Simulation of Parallel Programs on Application and System Level

Dissertation

zur Erlangung des akademischen Grades
Dr. rer. nat
an der **Fakultät für Mathematik, Informatik und Naturwissenschaften**
der **Universität Hamburg**

eingereicht beim Fachbereich Informatik von

Julian Martin Kunkel
aus Heidelberg

5. Februar 2013

Gutachten:

Prof. Dr. Thomas Ludwig
Prof. Dr.-Ing. Stefan Olbrich

Datum der Disputation: 11. Juli 2013

Abstract

Computer simulation revolutionizes traditional experimentation providing a virtual laboratory. The goal of high-performance computing is a fast execution of applications since this enables rapid experimentation. Performance of parallel applications can be improved by increasing either capability of hardware or execution efficiency. In order to increase utilization of hardware resources, a rich variety of optimization strategies is implemented in both hardware and software layers. The interactions of these strategies, however, result in very complex systems. This complexity makes assessing and understanding the measured performance of parallel applications in real systems exceedingly difficult.

To help in this task, in this thesis an innovative event-driven simulator for MPI-IO applications and underlying heterogeneous cluster computers is developed which can help us to assess measured performance. The simulator allows conducting MPI-IO application runs in silico, including the detailed simulations of collective communication patterns, parallel I/O and cluster hardware configurations. The simulation estimates the upper bounds for expected performance and therewith facilitates the evaluation of observed performance.

In addition to the simulator, the comprehensive tracing environment HDTrace is presented. HDTrace offers novel capabilities in analyzing parallel I/O. For example, it allows the internal behavior of MPI and the parallel file system PVFS to be traced. While PIOsimHD replays traced behavior of applications on arbitrary virtual cluster environments, in conjunction with HDTrace it is a powerful tool for localizing inefficiencies, conducting research on optimizations for communication algorithms, and evaluating arbitrary and future systems.

This thesis is organized according to a systematic methodology which aims at increasing insight into complex systems: The information provided in the background and related-work sections offers valuable analyses on parallel file systems, performance factors of parallel applications, the Message Passing Interface, the state-of-the-art in optimization and discrete-event simulation. The behavior of memory, network and I/O subsystem is assessed for our working group's cluster system, demonstrating the problems of characterizing hardware. One important insight of this analysis is that due to interactions between hardware characteristics and existing optimizations, performance does not follow common probability distributions, leading to unpredictable behavior of individual operations.

The hardware models developed for the simulator rely on just a handful of characteristics and implement only a few optimizations. However, a high accuracy of the developed models to explain real world phenomenons is demonstrated while performing a careful qualification and validation. Comprehensive experiments illustrate how simulation aids in localizing bottlenecks in parallel file system, MPI and hardware, and how it fosters understanding of system behavior. Additional experiments demonstrate the suitability of the novel tools for developing and evaluating alternative MPI and I/O algorithms. With its power to assess the performance of clusters running up to 1,000 processes, PIOsimHD serves as virtual laboratory for studying system internals.

In summary, the combination of the enhanced tracing environment and a novel simulator offers unprecedented insights into interactions between application, communication library, file system and hardware.

Zusammenfassung

Die Computersimulation löst in vielen Bereichen traditionelle Experimente ab. Ziel des Hochleistungsrechnens ist die schnelle Ausführung von Anwendungen, da dies die Durchführung der Experimente beschleunigt. Die Leistung einer parallelen Anwendung kann erhöht werden, indem die Fähigkeiten der Hardware verbessert werden oder durch die Steigerung der Effizienz während der Ausführung. Um verfügbare Hardwareressourcen bestmöglich auszunutzen, werden sowohl in Hardware- als auch in Softwareschichten verschiedenste Optimierungsstrategien implementiert. Ihr Zusammenspiel resultiert in einem hochkomplexen Systemverhalten. Dies erschwert eine Bewertung des Laufzeitverhaltens der Anwendungen. Eine Analyse der Leistungsfähigkeit einzelner Komponenten ist darüber hinaus unabdingbar, um Engpässe zu identifizieren und zu optimieren.

Aus diesem Grund werden in dieser Doktorarbeit Werkzeuge entwickelt, mit deren Hilfe das Verhalten besser bewertet werden kann. PIOsimHD ist ein innovativer ereignisbasierter Simulator für die Ausführung von MPI-IO-Anwendungen auf heterogenen Clustercomputern. Der Simulator beinhaltet eine detaillierte Simulation von kollektiven Kommunikationsmustern, paralleler Ein- und Ausgabe und Modellen für die zu Grunde liegende Hardware. Mit Hilfe der Modelle können obere Schranken für die erwartete Leistungsfähigkeit geschätzt werden; dies ermöglicht eine Bewertung der beobachteten Leistung.

Darüber hinaus wird mit HDTrace eine Umgebung für das Erfassen von Anwendungsverhalten in Spurdaten präsentiert. HDTrace bietet neuartige Möglichkeiten der Analyse von paralleler E/A. Beispielsweise gestattet HDTrace die Aufzeichnung des internen Verhaltens von MPI und des parallelen Dateisystems PVFS. Programmläufe können auf bestehenden Systemen aufgezeichnet und in PIOsimHD in beliebigen virtuellen Clusterumgebungen ausgeführt werden. Dieses Zusammenspiel von HDTrace und PIOsimHD gestattet die Lokalisierung von Ineffizienzen, erleichtert die Optimierung von Kommunikations- und E/A-Middleware und das Studium von Kommunikations- und E/A-Verhalten der Anwendungen auf künftigen Systemen.

Diese Arbeit zielt systematisch darauf ab, die Einsicht in komplexe Systeme zu verbessern. Zunächst werden Hintergrundinformationen zu verschiedenen Themen vorgestellt. Diese beinhalten Informationen zu parallelen Dateisystemen, grundlegenden Leistungsfaktoren von parallelen Anwendungen, der Nachrichtenaustauschschnittstelle MPI, Optimierungsmöglichkeiten und ereignisbasierter Simulation. Das Verhalten von Speicherzugriffen, Netzwerk und E/A-System des Clusters useres Arbeitsbereichs wird analysiert, hierbei werden einige Schwierigkeiten bei der Charakterisierung der Hardware praktisch demonstriert. Eine wichtige Erkenntnis der Analyse ist, dass das Zusammenspiel zwischen Hardwarecharakteristika und Optimierungen in einer unvorhersehbaren Geschwindigkeit einzelner Operationen resultiert, die keiner typischen Wahrscheinlichkeitsverteilung folgt.

Die entwickelten Hardwaremodelle des Simulators stützen sich auf eine Handvoll Charakteristika und konzentrieren sich auf die wichtigsten Optimierungen. Dennoch zeigt sich in der Validierung des Simulators eine hohe Übereinstimmung der Simulationsergebnisse mit den beobachteten Phänomenen. In diversen Experimenten wird der neu entwickelte Simulator genutzt, um das Verhalten des Systems besser zu verstehen. Hierbei werden zahlreiche Engpässe in parallelen Dateisystemen, MPI und Hardware aufgespürt. Zusätzliche Experimente demonstrieren die Anwendung der neuen Werkzeuge, um alternative Kommunikations- und E/A-Algorithmen zu entwerfen. Mit der Möglichkeit, effektiv die Leistungsfähigkeit von beliebigen Clustern mit bis zu 1000 Prozessen zu evaluieren, dient PIOsimHD als virtuelles Labor, um die Interaktionen zwischen System, Anwendung und Kommunikationsbibliothek zu studieren.

Acknowledgments

I would like to thank Prof. Dr. Thomas Ludwig, who has supported me ever since I attended his seminar "Betriebsystemaspekte beim Cluster-Computing" in the 2003/2004 winter term.
I want to appreciate the support of my parents Ursula and Paul, who did not become tired of asking when I finally graduate with my Ph.D.

I also want to thank to all the students who contributed to PIOsim and HDTrace, and to all those who supported me directly or indirectly throughout this project. Special thanks to Nathanael Hübbe and Michaela Zimmer who reviewed parts of my lengthy thesis.

This last sentence should pay tribute to all the programmers who developed the software that helped me to write this thesis; this includes software which I evaluated to prepare for this project. While no software is perfect, without those software packets this thesis would not have been possible.

[illegible]

Table of Contents

Chapter 1

Introduction

In this chapter an overview of high-performance computing is given based on illustrating examples. The compartments of a supercomputer, the general usage models, and the interplay of hardware components and software layers is introduced. The execution of a simple parallel application illuminates some of the difficulties a system is confronted with when attempting to achieve optimal performance.

A deeper understanding of the system allows for assessing performance limitations and shortcomings when designing and programming a supercomputer and its software stack. Analysis of the observed behavior in order to understand the causes for performance degradation is cumbersome which ultimately motivates this thesis.

1.1. High-Performance Computing

Supercomputers combine the performance of hundreds or thousands of office computers to tackle problems which could not otherwise be solved on PCs in a reasonable amount of time. With the capabilities offered by supercomputers, scientists no longer have to conduct time-consuming and error-prone experiments in the real world. Instead, the modeling and simulation of the laws of nature within computer systems offers a well-defined environment for experimental investigation. Models for climate, protein folding or nano materials, for example, can be simulated and manipulated at will without being restricted by the laws of nature. This method leads to new observations and understandings of phenomena which would otherwise be too fast or too slow to comprehend in vitro [KS92].

With the improvement of computing performance, better experiments can be designed and conducted. As such, a thorough understanding of hardware and software design is vital to providing the necessary computing power for scientists. This understanding has developed into its own branch within the computer science field: *High-Performance Computing* (HPC). *High-performance computing* is the discipline in which supercomputers are designed, integrated, programmed and maintained.

Supercomputers are tools used in the natural sciences to analyze scientific questions in silico. Indeed, HPC provides a new model of scientific inquiry – that is, a new way to obtain scientific knowledge. Mahootian and Eastman state:

> *"The volume of observational data and power of high-performance computing have increased by several orders of magnitude and reshaped the practice and concept of science, and indeed the philosophy of science"* [ME09]

Exponential growth of performance Depending on the processing power of the system, a number of varying scientific questions can be tackled. For example, the prediction and simulation of microbial systems in order to produce an ecological alternative to fossil fuels is one such scientific question that could be solved with future supercomputers [JKN09].

Because supercomputer performance is the crucial factor in successful modeling and simulations, performance is one of the key metrics to compare the various supercomputers currently in use. Faster machines allow bigger and more complex experiments to be conducted in silico. Therefore, methods which speed up systems are of interest.

Since 1993 the *Top500* list [Meu08] has been gathering information on the achieved performance of supercomputers, and this database provides a rich set of tools to analyze information regarding vendors, system architectures and trends of the fastest computers worldwide.

Up to now an exponential growth in computing performance, which is proportional to Moore's "law"[1] [TP06], is observable and the observable speed even exceeds it slightly.

As of 2011 the fastest system offers a theoretical peak performance[2] of 10 Petaflop/s. Thus, between 1993 and November of 2011 the performance of the fastest listed supercomputer improved from 60 Gigaflop/s to about 10 Petaflop/s, which is a factor of roughly 2^{17}.

On the one hand, the factor correlates to the improvement of chip design and miniaturization that has been achieved in these 18 years and is analogous to Moore's "law". On the other hand, it is a result of the improved clustering of independent chips into a large supercomputer and of increased investments.

This is a basic consideration since the designer, in order to increase the performance of a supercomputer, can either increase the processing speed of a single processor or the total number of collaborating processors. The latter is achieved by fusing more chips into a single computers and by interconnecting multiple independent computers into a so-called cluster computer.

While obstacles to achieve petascale have been overcome, designing and implementing machines and parallel algorithms scalable to exascale[3] supercomputers is a task which must yet be solved.

Utilizing available transistors In regards to performance, transistors available on only a single silicon chip can be used to improve performance of a single thread[4] or by adding more cache or vector units. Also, a chip can be partitioned into (almost) independent processing units which allows concurrent processing of multiple threads[5].

In the past, the application performance was improved by packing more functionality into a single chip and by incrementing the clock frequency of this processor. However, power consumption of a processor is proportional to the square of the clock frequency. Thus, combined with the steady miniaturization, at some point chips would have been designed that produce more heat than a nuclear power plant or even the sun [RMM+01]. As a result of this observation, a change in strategy was implemented. In order to utilize available transistors, multiple processing units are replicated on a single chip and clocked with a lower frequency. With this approach the aggregated performance of all cores is higher than by increasing the complexity of a single core. Nowadays, all commodity chips consist of multiple cores.

The multi-core trend rapidly increased the numbers of processors in cluster systems. As of June 2011 the fastest system on the Top500 is the *K computer*. It consists of more than 20,000 boards each equipped with four 8-core SPARC64 chips leading to an aggregated number of 705,024 cores. In comparison with this highly parallel system, the fastest supercomputer in 1993 was driven by a mere 1024 processors.

Efficiency of algorithms Potential performance of a supercomputer and the efficiency of an algorithm running on the machine are two separate issues. Utilizing a high number of processing cores efficiently is problematic. Applications must be rewritten to deal with multiple threads which is a challenge for the software developer. Naturally, the number of parallel threads which can be used efficiently to tackle a problem depends on both the problem and the algorithm used to solve that problem.

To give an example of a fast but inefficient algorithm consider the problem of sorting N distinct elements. For a sequential processor this task has a time complexity of $O(N \cdot \log N)$[6]. Jindaluang's parallel sorting

[1] Gordon E. Moore observed that the number of transistors which can be placed on an integrated circuit roughly doubled every year between 1958 and 1965, and he predicted that this trend would continue in the future. This trend is referred to as Moore's law by computer scientists. However because it is no law of nature the author prefers to quote the term.

[2] A flop is a **fl**oating point **op**eration. Peta is the prefix for 10^{15}. Thus, a petascale computer executes 10^{15} operations per second. In supercomputing the term FLOPS is often used to describe flop/s, but the author claims Flop/s is more intuitive.

[3] Exascale computers perform 10^{18} Flop/s.

[4] A thread can be thought of as a single stream of instructions that is processed sequentially on the chip.

[5] This principle is called 'thread-level parallelism'.

[6] In practice, quicksort is most often used because it yield this complexity. However, new theoretical results show an improvement to the complexity of $O(N \log\log N)$ [Han02].

algorithm for completely overlapping networks uses N processors to solve this problem in a time complexity of $O(N)$ [Jin04][7]. However, the improvement of the complexity from sequential to parallel system with a completely overlapping network is low. By adding more processors the amount of work to compute the solution increases from $O(N \log N)$ to $O(N \cdot N = N^2)$, which is the number of processors multiplied with the time complexity. Consequently, the efficiency, that is, the effective work used to solve the problem divided by the number of executed operations, decreases on the parallel system.

Low utilization of hardware In addition to the potential lower efficiency of a parallel algorithm, the resources provided by a machine are unfortunately not evenly utilized by parallel applications [SH94]. A near-optimal utilization of all resources provided on a single chip (or processor) is already a challenging task for a developer, a compiler and the software infrastructure. This list includes the operating system as well, which manages the low-level hardware and dispatches tasks to the available resources.

The complexity of designing and implementing efficient parallel applications is of an even higher degree than of their sequential versions. This is due to the additional synchronization and communication because the independent processors must exchange intermediate results to cooperate. This communication process incurs some overhead to the computation and might cause processors to stall while waiting for (new) data to process. Consequently, on the one hand, careful attention must be given to balance the work evenly among the processors to keep them busy; On the other, the communication must be performed efficiently to ensure a steady computation.

While some parallel numerical algorithms are capable of utilizing computing resources on a supercomputer to a high degree, most applications exploit only a small fraction of peak performance [OCC+07]. Therefore, tuning and optimization of applications to exploit the available resources is an important task in the effort to improve performance of the supercomputer and efficiency of the compute facility as a whole.

Benchmarking of supercomputers Because of the suboptimal utilization of the system, the maximum theoretical peak performance, which can be determined by multiplying the processing capabilities of a single chip with the number of available chips, is not sufficient to compare two supercomputers. Therefore, the effective performance is determined by running an efficient numerical algorithm as a benchmark.

A highly-optimized and well-parallelized algorithm that solves a system of linear equations is the *LINPACK* benchmark [DLP03], which typically achieves about 60-80% of the theoretical *peak performance* of the hardware.

The LINPACK benchmark is used to determine the ranking of supercomputers in the Top500 List, where the highest performance achieved with the HPC LINPACK is labeled *Rmax*. The theoretical performance is also listed in the Top500 List and is labeled with *Rpeak*. The *K computer*, for example, has a peak performance of 11.3 PFlop/s. By dividing both values a metric for a system's efficiency can be derived[8]. For example, with a large problem size the optimized HPC LINPACK version for the *K computer* achieves a phenomenal 93% of its theoretical peak performance.

Since the architectural design of the supercomputer has a major impact on the observable performance and its characteristics, the compartments of a supercomputer and their arrangement are sketched next.

1.1.1. Architecture of Supercomputers

In the past, a supercomputer was a big monolithic computer built using tightly coupled components, similar to a mainframe. The number of components in a supercomputer rose quickly, and for various

[7] For a Parallel Random Access Machine (PRAM) that offers Concurrent Read Exclusive Write (CREW) memory access, a parallel merge sort completes in $O(\log N)$ time complexity [Col86]. But, to illustrate overhead in an interconnected network topology Jindaluang's algorithm has been chosen.

[8] Note that this metric is rather artificial, typically scientific applications achieve only a fraction of a system's peak performance.

reasons it was no longer affordable to build one big monolithic system. Instead, independent components now form building blocks which can be put together in arbitrary configurations to handle the requirements of a given computing center.

Those, so-called compute *clusters*[9] combine several independent resources into one big computer. In a cluster computer autonomous servers are referred to as *node*. A schematic view of a cluster computer is shown in Figure 1.1: *Compute nodes* run the (parallel) application which accesses required input data and stores the computed results on a *storage system*. All resources are interconnected by at least one *network*.

Applications are usually compiled and prepared on a cluster *frontend* on which users log in over the Internet (or just the local area network). Once logged in to the frontend, the user can prepare parallel applications and access produced data. In the preparation step the parallel application is compiled to generate a sequence of machine instructions which can be executed on the particular hardware of the compute nodes. Further, a job description is prepared. This description includes the required compute resources for the parallel job and a sequence of parallel applications that shall be executed on those resources. Once the applications are ready to run, the scientist submits the job description to the cluster's *batch system*. The batch system exclusively assigns compute nodes to the job and initiates startup of the job script.

Clusters could be built from identical hardware, or from inhomogeneous hardware. In the former case homogeneous components, that is, identical components with the same characteristics, are deployed. While cluster nodes are usually equipped homogeneously, multiple storage systems are connected when dealing with specific workload requirements. Individual components of a cluster can either be special (proprietary) hardware, or standard consumer and enterprise hardware. Typically, the latter components are cheaper. Clusters built using those *Commercially available Off-The-Shelf* (COTS) components are widely known as *Beowulf cluster systems* [GLS03]. Nowadays, many systems use both types of hardware. For example, in 2010 the fastest supercomputer was *Jaguar*, a Cray XT which uses off-the-shelf AMD processors but a proprietary network technology from Cray.

Traditionally, the supercomputer is built independently of the applications, which are to be executed later on in the system. Thus, the demands of the software and the system capabilities may not match optimally. Recent studies show the importance of software-hardware *co-design* to cope with the challenges of applications on very large scale systems [JKN09, SHS09]. With this approach, software aspects are kept in mind as the hardware is designed, but the hardware design also influences the programming model of the software.

Computation Typically, processing of data in a cluster computer follows the *von Neumann architecture* [vN93]: A *processing unit* executes one instruction after another. Each instruction causes one of the available processing units to perform modifications of the data stored in a memory system. In fact, most processors implement a *memory-hierarchy*: To speed up access of the slow, but large, main memory, several faster *caches* should hold the working set. Operations are performed in the fastest memory – the so-called registers, which are embedded in the processing unit itself.

Nodes of a cluster computer are equipped with at least one microprocessor (*Central Processing Unit*) and one memory system. Modern multi-core processors provide not only multiple cores – each core is a full featured processing unit – but also provide multiple functional units, which can be operated simultaneously to manipulate data. Thus, a CPU can execute multiple instructions concurrently. Therefore, a CPU (or the compiler for the system) keeps track of data dependencies to ensure that the computation result is identical to a sequential execution.

In a cluster computer the nodes by themselves are independent components. Thus, memory of one node cannot be accessed directly from CPUs of other nodes. For that reason a cluster is a *distributed memory architecture*. However, data of one node can be transferred over the network to the memory of a remote node.

[9]Sometimes a compute cluster is referred to as a cluster computer as well.

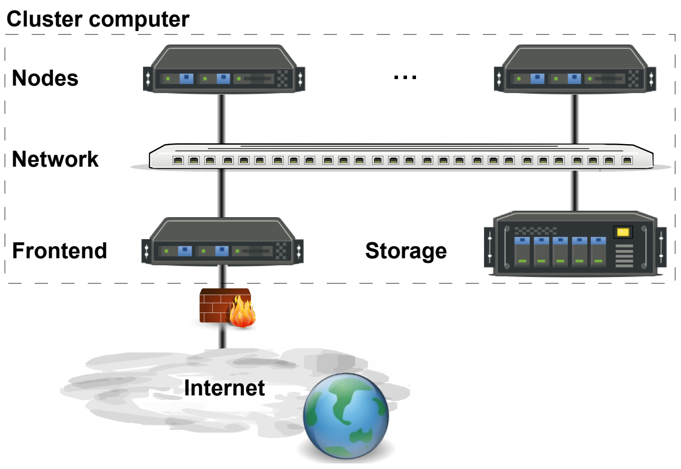


Figure 1.1.: Schematic view of a cluster computer. A scientist can connect to the frontend and work interactively.

In contrast, within a single cluster node or a supercomputer like the *Cray XT5*, all memory can be accessed from all CPUs. Therefore, the type of system is a *shared memory architecture*. In a shared memory system the time required to access data can vary with the CPU trying to access data. Some CPUs might be directly connected to the physical memory location while others have an indirect access. Such a system is called *non-uniform memory access architecture* (NUMA), otherwise it is called *uniform memory access architecture*.

Multi-socket systems with newer multi-core processors from Intel and AMD are already NUMA architectures by themselves. The available memory space is partitioned among the available processors, which are interconnected with a fast network to exchange data. Thus, CPUs (cores) of a processor need slightly more time to access memory assigned to another processor.

Network Design and performance of a network depends on the technology deployed and the layout of the interconnect between the network components. Gigabit Ethernet [Int01] is a commodity technology provided in consumer hardware. Contrary to the cost-effective Ethernet, the Infiniband technology [Tec00], for instance, is specifically designed for high performance.

Layout of the interconnection between nodes is defined by the system's *network topology*. In the best case, a node is connected to all other nodes with a direct connection, requiring a quadratic number of links to interconnect all nodes. In this case, partitioning of nodes into two disjoint groups leads to maximum throughput – half of the nodes can communicate with the other half without interference. Thus, the *full bisection bandwidth* is available for communication. Since cluster computers are deployed with node counts in the tens of thousands, the number of links must be reduced for practical reasons.

One way to reduce cost is to share network links, and therewith, multiple nodes relay messages over a shared link. However, when multiple messages should be transferred over the same link at the same time, then a *congestion* occurs; the available resources must be multiplexed among all of them resulting in lower performance. There are several communication methods that allow concurrent data transfer of messages over a shared network link. For instance with *packet switching* messages are partitioned into packets (or frames), a network component transfers one packet after another but can interleave packets from multiple messages.

To give an example for congestion, consider the Jaguar supercomputer that uses a 3D-torus SeaStar-2 network. An example of a *2D-torus* of nine nodes is shown in Figure 1.2. In this example the network interface of a node itself relays messages from other nodes to the destination node. Due to this network topology, a congestion might occur when messages from multiple sources are routed over one device or link. For instance, in the sketched 2D-torus it could happen that the upper left (0,0) and right nodes (0,2)

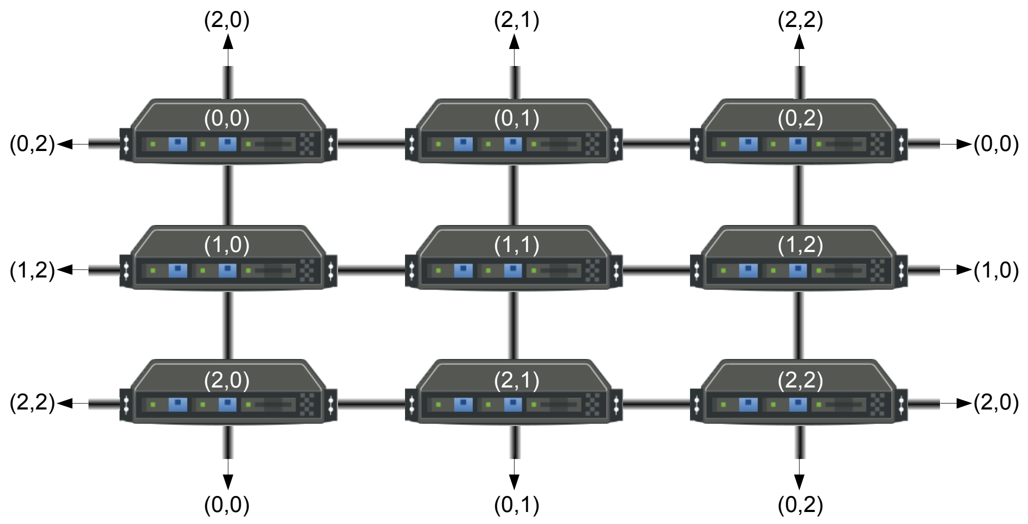


Figure 1.2.: Network topology of a 2D-Torus. Leftmost nodes (X,0) are interconnected with the rightmost nodes (X,2). Top nodes (0,X) are connected with the bottom nodes (2,X).

send a message to the center node (1,1). The routing algorithm might route both messages via node (0,1); however, the node has only one link to node (1,1).

Another frequently deployed class of network topologies are *hierarchical networks* which add intermediate layers of hardware that relay the data; a switch connects many links and forwards incoming data into the direction of the intended receiver. This approach provides a higher aggregated bandwidth between the nodes, but increases the latency of the communication. One hierarchical network topology is a *Clos* network, where a high number of links between switches offers the maximum available network bandwidth between all connected hosts. This network topology is often deployed with Infiniband technology. Refer to [AK11] for more details on network technologies and topologies.

Storage Data management in bigger clusters is performed by distributed file systems, which scale with the demands of the users. Usually, multiple file systems are deployed to deal with disjoint requirements.

In a typical setup two file systems are provided: a fast and large scratch space to store temporary results, and a slower, but highly available volume to store user data (e.g., for home directories). Looking at a particular distributed file system, a set of storage servers provides a high-level interface to manipulate objects of the namespace. The *namespace* is the logical folder and file structure the user can interact with; typically it is structured in a hierarchy. Files are split into parts which are distributed on multiple resources and which can be accessed concurrently to circumvent the bottleneck of a single resource. Replication of a part on multiple servers increases availability in case of server failure. Truly *parallel file systems* support concurrent access to disjoint parts of a file; in contrast, conventional file systems serialize I/O to some extent[10].

An example of the *hierarchical namespace* and its mapping to servers is given in Figure 1.3. The directory "home" links to two subfolders which contain some files. In this case, directories are mapped to exactly one server each, while the data of logical files is maintained on all three servers. Data of "*myFile.xyz*" is split into ranges of equal size and these blocks are distributed round-robin among the data servers. Each server holds three ranges of the file.

Storage devices are required to maintain the state of the file system in a persistent and consistent way. A storage device can be either directly attached to a server (e.g., a built in hard disk), or the server controls devices attached to the network. In contrast to the interface provided by file systems, the interface to

[10]More details on file systems in HPC systems are provided in Section 2.1.

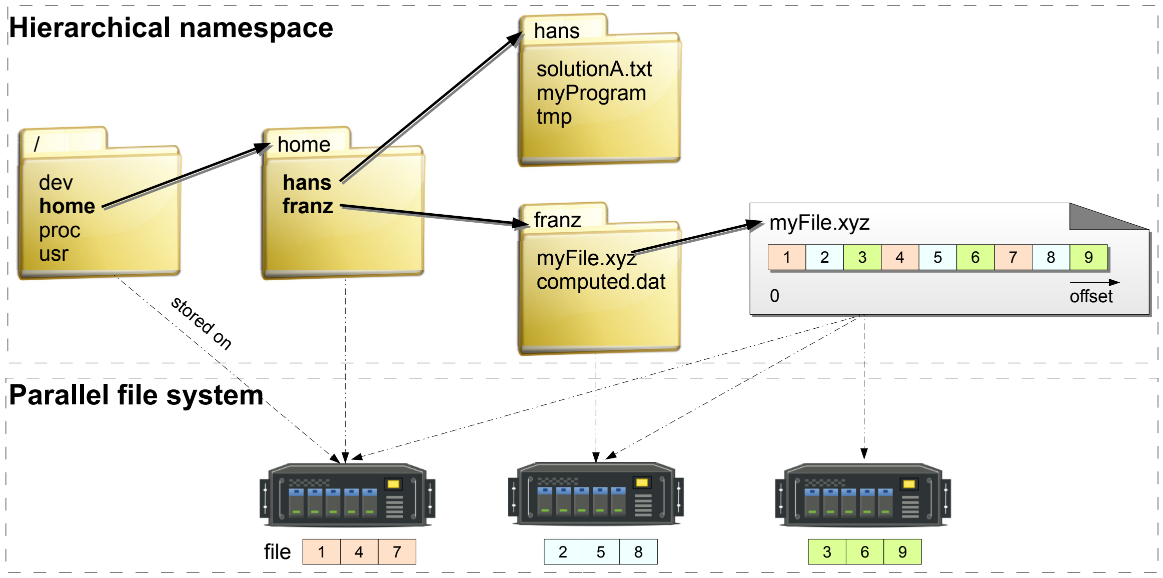


Figure 1.3.: Example hierarchical namespace and mapping of the objects to servers of a parallel file system. Here, metadata of a single logical object belongs to exactly one server, while file data is distributed across all servers.

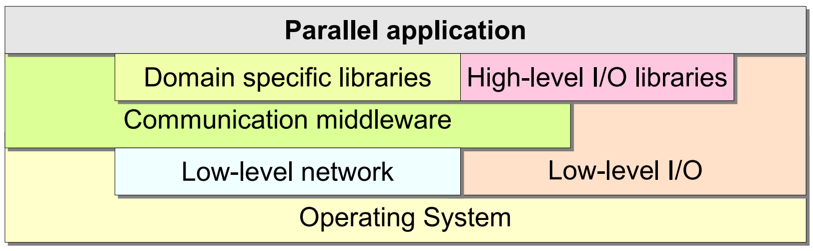


Figure 1.4.: Representative software stack for parallel applications.

storage devices is low-level. At the block-level, data can be accessed with a granularity of full blocks by specifying the block number and *access type* (read or write).

In a *Storage Area Network* (*SAN*) the block device seems to be connected directly to the server. To communicate with the remote block storage device, servers use the *Small Computer System Interface* (SCSI) command protocol. A SAN could share the communication infrastructure, or another network just for I/O can be deployed. Fibre Channel and Ethernet are common network technologies with which to build a SAN. In the latter case, SCSI commands are encapsulated into the IP protocol, that is, the so-called *Internet SCSI* (iSCSI). Therefore, the existing communication infrastructure can be used for I/O, too.

1.1.2. Software Layers

Several software layers are involved in running applications on supercomputers. A representative software stack is shown in Figure 1.4.

A *parallel application* uses a domain-specific framework and libraries to perform tasks common to most applications in that field. For example, the numerical library Atlas[11] is widely adopted by scientists.

Since collaboration between remote processes of an application requires communicating data, a conveniently programmable interface and an efficient implementation is important. The service to exchanged data is offered by the *communication middleware*. Several programming paradigms and models exist for

[11]Automatically Tuned Linear Algebra Software

each architectural type of the parallel computer, and those models often explicitly address the way communication is performed. Communication models can be classified according to the characteristics of the model. For instance, a classification made by the level of abstraction for communication distinguishes whether data exchange happens automatically whenever necessary, (i.e., is hidden from the programmer), or if data exchange must be encoded explicitly. The middleware might offer a high-level interface that abstracts from the physical location of processes, instead the user just specifies the particular communication partner.

Two models used to program the distributed memory architecture of cluster machines are MPI and PGAS. With the *Message Passing Interface* (MPI) [Mes09] the programmer embeds instructions in his code to explicitly send and receive messages. MPI also offers routines for parallel input and output of data. With *Partitioned Global Address Space* (PGAS) a process can access data that is stored in remote memory by using the syntax of the programming language, such as array access. Run-time environments ensure that, if required, data is transferred between the systems. However, the data partitioning is still encoded by the programmer. The language extensions which enable remote memory access in C and Fortran are called *Unified Parallel C* (UPC) or *Coarray Fortran* (CAF), respectively.

It is common practice to use MPI to communicate within a distributed memory architecture, and to use *Open Multi-Processing* (OpenMP) [BC07] to collaborate within a single node. Since developers may use both programming paradigms at the same time, the efficiency of this hybrid-programming model is specific to the problem being addressed and to the underlying hardware.

Input data and results are accessed either by harnessing *high-level I/O libraries* or by using low-level I/O interfaces such as the *Portable Operating System Interface* (POSIX). The *Network Common Data Form* (NetCDF) [HR08] and the *Hierarchical Data Format* (HDF5) are common high-level I/O libraries, that hide the complexity of defining low-level data formats from the user and offer features such as automatic data conversion. I/O can be performed with semantics close to the data structures used in the code. Domain specific libraries or high-level I/O libraries could be parallelized with MPI or OpenMP in order to utilize available cluster resources.

Low-level network communication enables the node to transfer data with another node. In contrast to a communication middleware, it operates directly with the network device and uses the network specific address formats. Programmers of an application usually work with the communication middleware, which provides a higher level of abstraction to address remote processes that is closely related to application logic. For example, the programmer should not have to care about the address (i.e., host name and port) on which the process might be listening. Distributed file systems provide their own *low-level I/O* interface to transmit operations and data between the storage servers and the node calling for I/O.

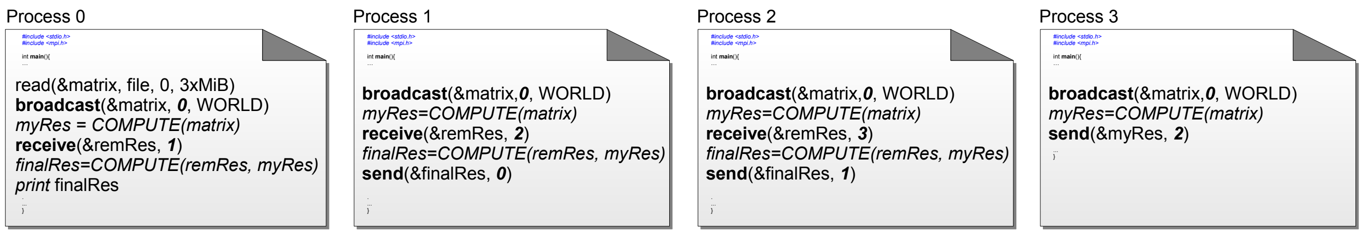
An *operating system* (OS) controls the local hardware and provides a software basis to run applications on a node. Often, the POSIX standard is supported by the OS. However, to reduce the overhead and complexity of background interaction a few supercomputers provide a reduced operating system. BlueGene, for example, offers a compute kernel with restricted threading support for processes [AAA^{+}02].

The figure is representative of most applications, but theoretically a layer could use all underlying layers directly. For example, a parallel application could call the functionality of the operating system to drive a network interface, however, programming of the communication would be cumbersome.

Software layers can be provided by the vendor of the technology, a supercomputer vendor, a cluster integrator or by the open source community. For performance reasons a cluster's integrator often gears the software stack towards the cluster's hardware.

1.1.3. Example Application Execution

Consider a simple parallel MPI program running on four processors; each node has two logical processors which execute commands concurrently. Figure 1.5 shows the relevant code for each process: An input matrix is read on Process 0 by a high-level I/O library and broadcasted to all processes. Each process



Process 0
```
#include <stdio.h>
#include <mpi.h>

int main(){
...
read(&matrix, file, 0, 3xMiB)
broadcast(&matrix, 0, WORLD)
myRes = COMPUTE(matrix)
receive(&remRes, 1)
finalRes=COMPUTE(remRes, myRes)
print finalRes
.
...
}
```

Process 1
```
#include <stdio.h>
#include <mpi.h>

int main(){
...
broadcast(&matrix,0, WORLD)
myRes=COMPUTE(matrix)
receive(&remRes, 2)
finalRes=COMPUTE(remRes, myRes)
send(&finalRes, 0)
.
}
```

Process 2
```
#include <stdio.h>
#include <mpi.h>

int main(){
...
broadcast(&matrix,0, WORLD)
myRes=COMPUTE(matrix)
receive(&remRes, 3)
finalRes=COMPUTE(remRes, myRes)
send(&finalRes, 1)
.
}
```

Process 3
```
#include <stdio.h>
#include <mpi.h>

int main(){
...
broadcast(&matrix,0, WORLD)
myRes=COMPUTE(matrix)
send(&myRes, 2)
.
}
```

Figure 1.5.: Pseudo code for a simple parallel application.

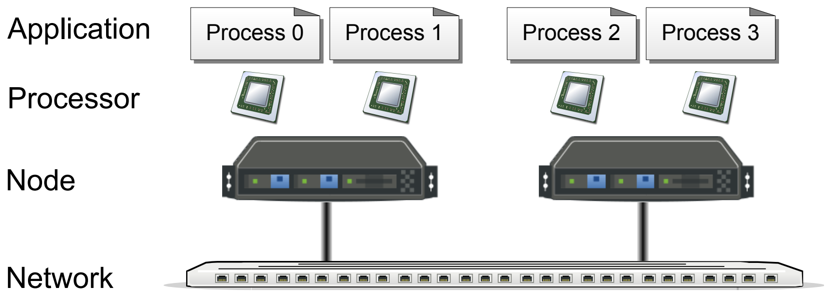


Figure 1.6.: Mapping of the application processes to two nodes of a cluster.

performs some computation on the matrix and then receives and incorporates results from later processes. Finally, the first process prints the results.

Commands in bold font are provided by the communication middleware. With the message passing paradigm sending and receiving of data must be explicitly encoded in the program. All processes are enumerated upon application startup, which enables addressing of a communication partner by specifying its process number.

Mapping the program to available hardware The mapping of the application to a homogeneous cluster is outlined in Figure 1.6. In this scheme a process is mapped exclusively to a processor, and applications are mapped exclusively to nodes, that means nodes and processors are dedicated to one application or process, respectively. Widely spread in HPC, this concept of *space sharing* of compute resources isolates concurrently running applications as those might interfere which each other. In general, this kind of cross-talk is unpredictable because each application has individual requirements for different hardware resources such as network, I/O or memory[12]. As scientific applications include inter-processor communication, the goal of mapping processes to available resources is to place processes which collaborate more with each other, close together – in the best case, on the same processor (but on different cores). Task scheduling on cluster systems is still under research – for example, recent papers such as [JM10] investigate scheduling algorithms.

Reading the input data Process 0 reads the whole matrix (3 MiB) with a single I/O request. Hardware and operations involved in retrieving the data from the file are sketched in Figure 1.7: The file data is split among three I/O servers; given a round-robin distribution of the file, the parallel file system could request one MiB per server for the data objects. Each server translates the offset to block position on the storage devices and requests access via SCSI commands from the storage device attached to the server. At the block-level, file data could be scattered across the device, requiring multiple low-level requests to fetch all blocks – this is illustrated for the rightmost storage server. Note that the file system objects composing the logical file, and the location of the physical data on the storage devices are hidden from the application developer.

[12]Further information about scheduling in HPC can be found in [CCS+06].

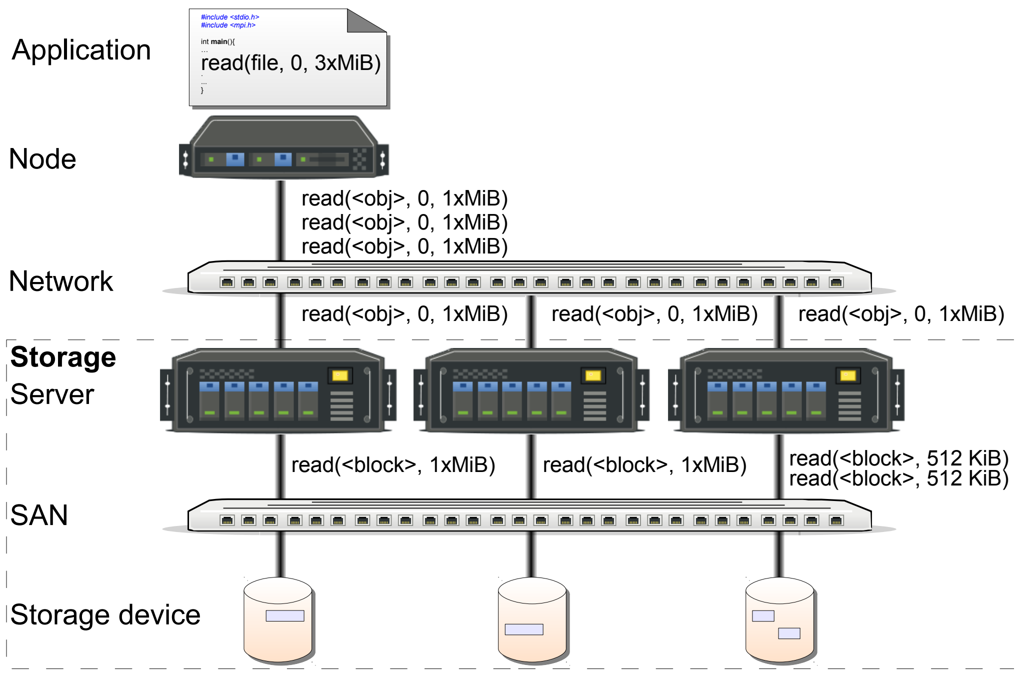


Figure 1.7.: Involved hardware and operations to retrieve 3 MiB of data from a file stored on a parallel file system. In the example file servers access storage devices attached to a SAN.

Broadcasting input data to all peers In the meantime the other processes execute the broadcast operation. A broadcast command will transfer data from a root process (in this case, Process 0) to all other processes. The MPI implementation could choose between one of multiple algorithms to achieve this goal. One algorithm is to simply transfer the data from Process 0 to all other processes in a for loop. Another, and more sophisticated algorithm, is to transfer data along edges of a spanning tree among the processes. In our toy example this is realized as follows: first Process 0 sends data to Process 2, then Process 0 and Process 2 have the data. Next, both processes concurrently copy data to their neighboring processes. Since Process 1 is hosted on the same node as Process 0 and the other two processes are hosted on the other node, data can be copied locally, which reduces the network communication between the nodes.

Whatever implementation is chosen, once a process finishes the broadcast operation, semantics define that the data has been received from the root. Therefore, the root process must have initiated the broadcast operation before data can be received; synchronization is not mandatory. Similar, semantics of send and receive operations do not require explicit synchronization between sender and receiver. As a consequence a send operation could complete, even though the recipient is not yet ready to accept the particular message. Internally the recipient could just keep the message in a buffer.

Execution variants Next, the execution of a parallel program, the caused activities and data dependencies are illustrated. Depending on the MPI implementation and duration of the individual operations, various interaction patterns are observable. The illustrating example visualized in Figure 1.8 on page 12 demonstrates four execution variants. The figure shows an interaction diagram alike to the *Unified Modeling Language* (UML): Activities of the components over time are printed in rectangles below the process. Actually, the quantitative timing is not so important, but to foster the discussion the timestamps are provided. Dependencies between the processes are shown as arrows – a receiver has to wait for the sender's data. For the moment assume the main computation needs 6 time units and every data exchange and I/O operation takes 1 time unit. Several factors define the actual speed of the communication and computation process, those are further explained in Section 2.2. The internal implementation of the broadcast operation within MPI, i.e., the sequence of send and receive operations, is outlined by arrows as well.

In Figure 1.8a the naive and sequential broadcast is performed. Due to the data dependency the send operation of Process 3 is blocked an additional time unit until the receive operation of Process 2 is called.

The execution of the example needs 15 time units since Process 0 finishes at that time. It is also possible to communicate data in the reverse order, that means Process 0 sends data first to Process 1. This scheme would increase the total runtime by 1 time unit, because the time critical Process 3 would be the last process that finishes the broadcast.

The tree communication scheme is used in Figure 1.8b. While Process 0 and 1 finish the broadcast quicker, completing the broadcast on the last process takes longer because it receives data later as when using the sequential broadcast implementation. Overall, due to the data dependencies the same execution time as with the naive algorithm is observable. If the process mapping as depicted in Figure 1.6 is used in this theoretical consideration, then the communication between processes hosted on the same compute node will be faster, which decreases the time from 1 time unit to a fraction, for example to 0.1 time unit. Since in that case one slow intra-node communication is saved, this scheme would outperform the naive algorithm.

Load balance With both implementations the processes have to wait to receive data from their successors – a *late sender* defers program execution on the first two processes both execution variants. In Figure 1.8c the potential computation time, which does not increase the run-time of the algorithm, is provided[13]. Computation could be done during the wait time, therefore, 30 units of work could be done compared to the 24 units which are actually used.

Since the program just needs to compute for 24 time units the load must be distributed differently among the processors. The ratio in which work is distributed among processors is called *load balance*, and *load balancing* [SKH95] is the methodology to balance work among the processors in such a manner that all available resources are well utilized. In example (a) and (b) the main computation of the processes takes the same amount of time – 6 units of work – therefore the work is perfectly balanced among the processes.

To avoid idle time an optimal load balancing would have lead to configuration (d), illustrated in Figure 1.8d. In this configuration the 24 time units of computation are distributed such that less computation on the processes 2 and 3 cause an earlier reception of the intermediate results. Computation on Process 0 is finished just in time to receive the aggregated results from Process 1. With this configuration Process 0 finishes after 13.5 time units, which is faster than for the other execution variants which need 15 time units.

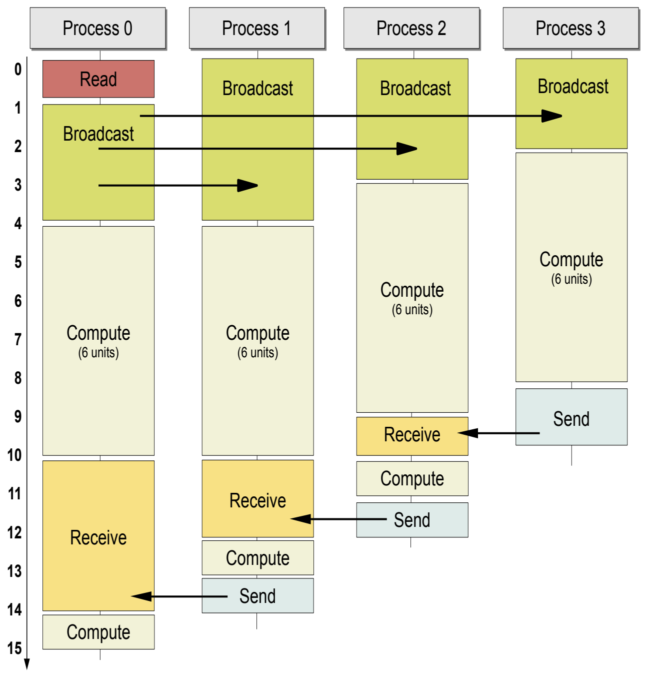
Nevertheless, optimally balancing the load in scheme (d) depends on the execution of the broadcast algorithm, the actual hardware configuration and the knowledge about the broadcast algorithm within the application. Without an estimate for the processing time of the broadcast operation on the processes data cannot be distributed optimally. One possible approach to balance load is to time the broadcast on all processes and distribute work according to the measured timings. Even a slight variation in the process-to-hardware mapping, or changes in the MPI internal broadcast algorithm, would lead to completely different distribution of idle time on the processors, which in turn would require another workload.

1.2. Motivation

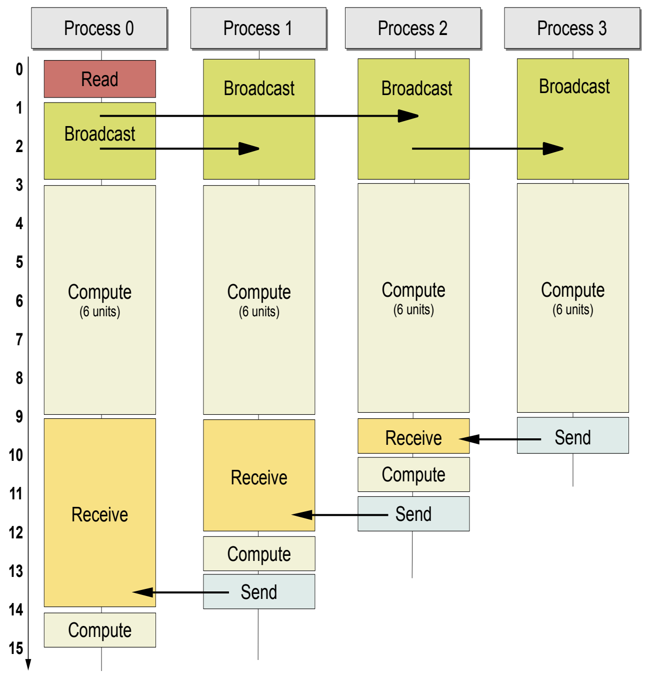
Since understanding hardware and software performance is the foundation for optimizing application and system behavior, providing innovative insights into these fields is the driving force of this thesis.

With regard to a computing facility, existing hardware resources should be utilized as well as possible in order to justify its acquisition and the resultant maintenance costs. Users of computing resources have a keen interest in reducing the runtime of their applications in their effort to solve scientific questions quickly; therefore, an application is *scaled* up to run on as many resources as possible. However, a typical

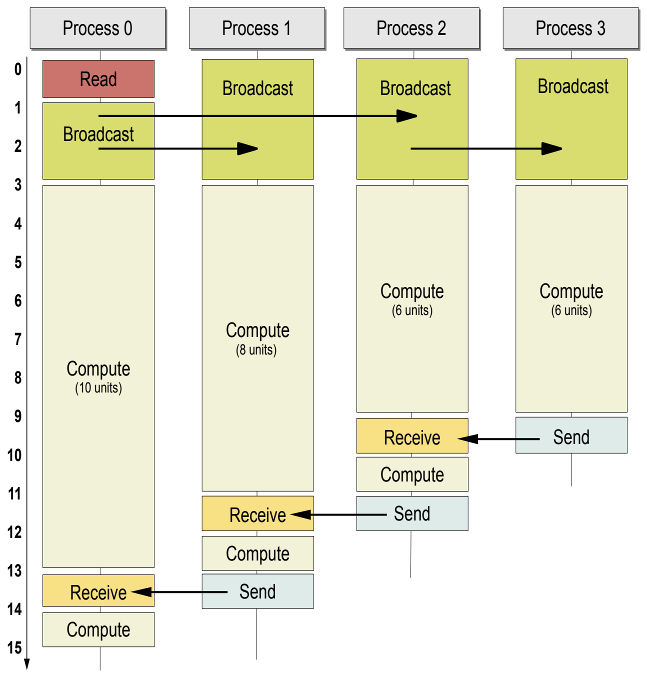
[13]In general, the data dependency of an algorithm restricts the possible communication patterns. For the trivial example application the data aggregation phase at the end could be implemented with a better collection algorithm. However, to illustrate the concepts in this discussion, the given communication pattern (and the algorithm) is considered to be fixed.



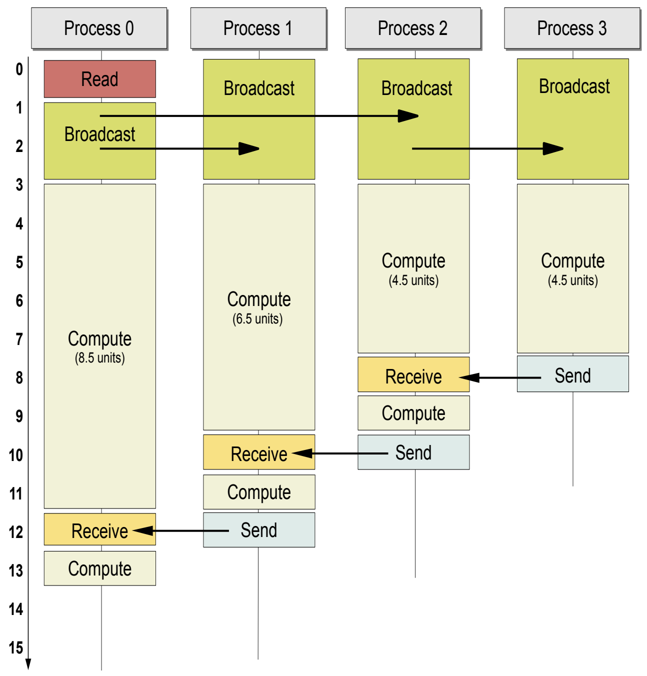
(a) Balanced computation using the naive sequential broadcast algorithm.



(b) Balanced computation using the tree broadcast algorithm.



(c) Maximum computation using up existing wait time in the communication.



(d) Perfect distribution of work among processors to avoid idle time.

Figure 1.8.: Some possible interaction pattern between the four processes of the example application. The actual observable pattern depends on the hardware and software configuration. In this example all processes are mapped to disjoint nodes, i.e., the communication between processes always takes 1 time unit.

HPC user has a limited amount of time to adapt an application to a given system and to configure the environment in an optimal way.

Consequently, even if very efficient algorithms are implemented resources are often wasted. The following lists a few reasons why resources might be utilized suboptimally:

- Limited scalability of the algorithm adds computational and communication overhead.
- A wrong mapping of processes to hardware leads to unnecessary communication.
- An inefficient I/O access pattern degrades performance of the parallel file system.
- Load imbalance in I/O or computation causes bottlenecks and idle resources on the system.

Efficiency also depends on the system characteristics, that is the hardware characteristics of the basic components: network, storage and compute nodes; the topologies of all interconnects, software layers involved and finally, on the configuration made by the administrator and user.

In software layers the system's algorithms for communication and I/O have a large influence on the observable performance[14]. Performance of the communication library itself is influenced by a number of factors. First, it is obvious that the library should be geared towards a given system, after which the right communication algorithm must be chosen. However, the tuning is complicated because performance of each communication algorithm depends on the parameters specified by the user. For example, while the broadcasting of small messages could be faster on the given hardware by sending data sequentially from the root, a tree algorithm might perform better for larger data, and a pipelined peer-to-peer alike algorithm would be preferable for big data. While the performance of the algorithm could be optimized for itself, efficiency also depends on the context of the application, that is, the number of processes and the current and future instructions executed by the processes.

Heterogeneous systems like Grids and Clouds increase complexity of the optimization exponentially, as more inhomogeneous components are involved.

It certainly is not easy to understand the interplay between all the hardware characteristics and potential bottlenecks, and unfortunately, mechanisms designed to optimize the system make it even harder to assess achieved performance and relate that performance to the system's capability. Further, on a high level of abstraction the communication interface implementation and I/O layers use techniques that, in most cases, improve performance. This is achieved by manipulating the request, deferring the operation, or fusing operations into one compound operation.

The complexity of analyzing an application on a distributed supercomputer becomes clear when a developer tries is to understand performance of a particular sequential code, which is only executed on a single processor. On the one hand, observed performance depends on the CPU architecture – branch-prediction, caches, the translation lookaside buffer, to name just a few mechanisms for optimizing hardware. On the other hand, it also depends on the compiler, which tries to transform the given high-level code to machine instructions in the best way possible.

From the user's perspective, post-mortem performance optimization is state-of-the-art. In this process performance of an application is measured on an existing system and analyzed to identify bottlenecks. Nowadays, developers are happy to achieve 10% peak performance on a given system[15]. It is important to optimize from the most promising and performance-boosting bottleneck to the least one. When confronted with this issue an important question arises concerning how much work is necessary to tune or modify the algorithm, the code and the system, and what will be gained by these modifications.

Estimating performance Modeling the system allows assessing obtained performance and therewith estimate the performance potentially gained by optimizations. Simple approaches to optimize floating point operations are to measure them and compare them with the theoretical peak performance of the system.

[14]More information of relevant performance factors is given in Section 2.2.

[15]This information is derived from available presentations of compute centers and scientific applications.

Since this theoretical value might not be achieved in practice due to inter-process communication and control structures, performance of an application can be also compared with the floating point values achieved with the high-performance LINPACK. If only 1% of LINPACK is achieved, then it might be advised to optimize the code; in the best case, performance is expected improve by a factor of 100. One should start on the code sections that run most of the time; it does not help much to optimize a function which runs for only a fraction of the total run-time.

While systems can be upgraded to enhance the performance of applications, it is important to consider the amount of improvement that could be observed using this approach. By replacing CPUs with newer generations that have twice as many cores per CPU, the performance of a particular algorithm is not increased twofold. If scalability of an application is limited by memory bandwidth or network, it might not benefit from an extension at all. Projecting application behavior on a potentially extended or future system could be helpful in deciding whether particular hardware should be purchased. By estimating application performance on possible configurations of a future system, the most promising extensions could be deployed.

Similarly, it may be advantageous to analyze the performance of applications before a new system is built from scratch. Carrying out such an evaluation during the design of a new system could guide the process of development in an effort to avoid later disappointments.

While inefficiencies in the code can be assessed once a target machine is up and running, this optimization process is difficult and time-consuming. To justify operating costs of a multi-million-euro supercomputer it is absolutely necessary to provide the community with tools to assess application and system performance as well as efficiency during the design of an algorithm, and to further assess and optimize systems and the software stack before they are built.

Modeling and simulating a cluster computer and the applications running on it addresses these issues. With the help of simulation the behavior of applications and systems could be predicted for arbitrary configurations. With this approach we go one step towards understanding system behavior, which is also a step in the direction of the integrated development of algorithms and supercomputers. All natural sciences harness models of the world and use simulations as frames through which to gain knowledge, computer science can keep pace by designing supercomputers and their software stacks with the same tools. This thesis aims to address these issues.

1.3. Goals of this Thesis

The main goal of this thesis is to simulate the execution behavior of MPI programs that run on arbitrary (virtual) cluster systems.

The simulator developed for this purpose should assist us in the following use-cases:

- Understanding of performance factors in cluster systems and application execution.
- Localization of bottlenecks in a cluster configuration and their causes in application logic and the software stack.
- Evaluation and optimization of the I/O path, the client-server communication and the server cache layers.
- Extrapolation of system performance towards future systems.
- Experimentation with various MPI-internal algorithms to gear them towards application and systems.
- Evaluation of new MPI commands and alternative MPI semantics.
- Teaching of the above aspects.

These goals are achieved by composing a model for cluster computers and MPI applications which should be easy to understand, yet powerful enough to represent major performance aspects of a cluster system. The created simulator implements the composed model. Consequently, the computed theoretical performance estimate should enable us to assess observed performance qualitatively. This thesis does **not** aim to provide a full-featured low-level simulation of system and application activity.

To ease evaluation of existing applications, tools should permit to record application behavior and to replay the behavior in the virtual environment. Furthermore, the simulation results should be assessable with the same methodology when analyzing parallel program execution.

As the software environment of this thesis evolved the goals have been extended to further evaluate novel visualization techniques that simplify the analysis of parallel MPI programs. Both the simulation and the visualization extensions allow to gain new insight into the behavior of applications and systems.

1.4. Outline of the Thesis

Since this chapter plunged deep into the topic of high-performance computing, several aspects relevant for this thesis are described in more detail in Chapter 2. Those aspects include parallel file systems, MPI details, and performance aspects of parallel applications and hosting systems. Further, the state-of-the-art methodology to analyze and optimize performance issues is introduced, as well as simulation concepts and existing tools.

Performance of the working group's cluster is characterized in Chapter 3. By assessing computation, communication and local I/O performance, the complex interplay of performance aspects as discussed in Chapter 2 should become more clear. Additionally, performance behavior of the experimental system is measured and assessed to characterize it for later experiments.

In Chapter 4 the new software environment created to record MPI activity and to simulate system and application behavior is presented. A tool that offers several unique capabilities to visualize and compare the simulated behavior with observations from real application runs is also introduced. System and application models and the simulator implementing them are discussed in detail in Chapter 5. The general workflow of simulating application and system behavior is then illustrated. For the interested reader, some implementation details fostering modularity and configuration of the simulator are given in Chapter 6. This includes: model creation, configuration and selection of modules, abstract event processing in components, and execution of arbitrary application commands.

HDTrace and the simulator are evaluated in Chapter 7. This evaluation includes validation of the basic cluster model. Also, the chapter assesses the differences between observations presented in Chapter 3 and the results computed by analytical models behind the simulator. Additional experiments with the simulator verify the implemented model against the conceptual model. Furthermore, sophisticated experiments are performed with the simulator, which indicate how the environment can be applied to analyze synthetic benchmarks and real applications.

The summary in Chapter 8 presents the scientific achievements in condensed form and concludes the thesis. Finally, Chapter 9 discusses potential improvements of the environment and outlines future work.

Chapter summary

This introductory chapter outlined the scope and complexity of high-performance computing as well as highlighting some system characteristics and several application aspects related to performance. The basic usage of supercomputers, and several problems in their development were briefly discussed, and the aim to address these issues via a new simulation environment has been sketched.

In the next chapter existing concepts and related work to relevant topics are discussed in more detail.

Bibliography

[AAA$^+$02] N. R. Adiga, G. Almasi, G. S. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. M. Cipolla, P. Crumley, K. M. Desai, A. Deutsch, T. Domany, M. B. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. E. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. A. Haring, D. Heidel, P. Heidelberger, L. M. Herger, D. Hoenicke, R. D. Jackson, T. Jamal-Eddine, G. V. Kopcsay, E. Krevat, M. P. Kurhekar, A. P. Lanzetta, D. Lieber, L. K. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. E. Moreira, B. J. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. B. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. K. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. D. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. B. Tremaine, M. Tsao, A. R. Umamaheshwaran, P. Verma, P. Vranas, T. J. C. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. T. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. K. Seager, J. S. Vetter, and K. Yates. An Overview of the BlueGene/L Supercomputer. page 60, 2002.

[AK11] Dennis Abts and John Kim. *High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, March 2011.

[BC07] Ruud Van Der Pas Barbara Chapman, Gabriele Jost. *Using OpenMP: Portable Shared Memory Parallel Programming*. Mit Press, 2007.

[CCS$^+$06] Jiannong Cao, Alvin Chan, Yudong Sun, Sajal Das, and Minyi Guo. A Taxonomy of Application Scheduling Tools for High Performance Cluster Computing. *Cluster Computing*, 9:355–371, 2006.

[Col86] R. Cole. Parallel Merge Sort. In *27th Annual Symposium on Foundations of Computer Science*, pages 511–516. IEEE, October 1986.

[DLP03] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, Present, and Future. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[GLS03] William Gropp, Ewing Lusk, and Thomas Sterling. *Beowulf Cluster Computing with Linux*. The MIT Press, 2nd edition, 2003.

[Han02] Yijie Han. Deterministic Sorting in O(nlog log n) Time and Linear Space. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 602–608, New York, NY, USA, 2002. ACM.

[HR08] Edward Hartnett and R. K. Rew. Experience With an Enhanced netCDF Data Model and Interface for Scientific Data Access. *24th Conference on Interactive Information Processing Systems (IIPS)*, 2008.

[Int01] Intel. Gigabit Ethernet - Technology and Solutions. `http://www.intel.com/network/connectivity/resources/doc_library/white_papers/gigabit_ethernet/gigabit_ethernet.pdf`, 2001.

[Jin04] W. Jindaluang. Time-Optimal Parallel Sorting Algorithm on a Completely Overlapping Network. In *Proceedings of the First Thailand Computer Science Conference (ThCSC2004)*, Bangkok, Thailand, 2004.

[JKN09] Wayne Joubert, Douglas Kothe, and Hai Ah Nam. PREPARING FOR EXASCALE: Application

Requirements and Strategy. Technical report, ORNL Leadership Computing Facility, December 2009.

[JM10] Emmanuel Jeannot and Guillaume Mercier. Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In Pasqua D'Ambra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 199–210. Springer Berlin / Heidelberg, 2010.

[KS92] William J. Kaufmann and Larry L. Smarr. *Supercomputing and the Transformation of Science*. W. H. Freeman & Co., New York, NY, USA, 1992.

[ME09] Farzad Mahootian and Timothy E. Eastman. Complementary Frameworks of Scientific Inquiry: Hypothetico-Deductive, Hypothetico-Inductive, and Observational-Inductive. *World Futures: Journal of General Evolution*, 65:61–75, 2009.

[Mes09] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard – Version 2.2. Technical report, September 2009.

[Meu08] Hans Werner Meuer. The TOP500 Project: Looking Back over 15 Years of Supercomputing Experience, 2008.

[OCC$^+$07] Leonid Oliker, Andrew Canning, Jonathan Carter, Costin Iancu, Michael Lijewski, Shoaib Kamil, John Shalf, Hongzhang Shan, Erich Strohmaier, Stéphane Ethier, and Tom Goodale. Scientific Application Performance on Candidate PetaScale Platforms. In *International Parallel & Distributed Processing Symposium*, IPDPS, 2007.

[RMM$^+$01] Ronny Ronen, Senior Member, Avi Mendelson, Konrad Lai, Shih lien Lu, Fred Pollack, John, and P. Shen. Coming Challenges in Microarchitecture and Architecture. volume 89, pages 325–340. IEEE, 2001.

[SH94] W. Schönauer and H. Häfner. Explaining the Gap Between Theoretical Peak Performance and Real Performance for Supercomputer Architectures. *Scientific Programming*, 3(2):157–168, 1994.

[SHS09] Vivek Sarkar, William Harrod, and Allan E Snavely. Software Challenges in Extreme Scale Systems. *Journal of Physics: Conference Series*, 180(1):012045, 2009.

[SKH95] Behrooz A. Shirazi, Krishna M. Kavi, and Ali R. Hurson, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[Tec00] Mellanox Technologies. Introduction to Infiniband. Document Number 2003WP, `http://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf`, 2000.

[TP06] Scott E. Thompson and Srivatsan Parthasarathy. Moore's Law: the Future of Si Microelectronics. *Materials Today*, 9(6):20–25, 2006.

[vN93] John von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15:27–75, October 1993.

Chapter 2

Background and Related Work

This chapter provides solid background to a rich variety of topics[1]*. After discussing each topic, the state of the art and related work are discussed. In this context recent research and several software tools are shown*[2]*.*

The first three sections provide more detail about HPC hardware and software, and their performance implications. First, the concepts of file systems and several real world representatives are introduced in Section 2.1. Then, aspects involved in the performance of parallel applications are discussed in Section 2.2. This indicates the complexity of those systems, but also enables classifying of relevant aspects. Next, the Message Passing Interface is introduced in Section 2.3 – special emphasis is put on the optimization potential within MPI.

Fast execution of an application is of main interest to the user since scientific goals should be achieved in time. To design fast programs, a user must understand the run-time behavior of the program on the system on which the program will be executed. Nowadays, users typically measure run-time behavior of their application, locate the bottleneck and try to improve performance in these long-running and thus critical code sections. There are also software-engineering concepts that can be applied during the whole software development cycle that take performance factors into account. Methodologies that lead to improved run-times in the implemented application are discussed in Section 2.4.

In Section 2.5 background about creation and verification of a model for a system is provided. The concept of simulation is introduced, too. A simulator implements a model and allows analyzing its behavior in silico. Further, several simulation tools that ease the implementation of models are introduced. While this section is not related to HPC, it provides relevant background to the art of simulation.

At last, in Section 2.6 a couple of simulators related to the simulation of cluster and I/O systems are presented. None of them allow to simulate parallel I/O and MPI at the level of detail required for this thesis.

2.1. Parallel File Systems

In contrast to a distributed file system, a *parallel file system* is explicitly designed for achieving performant and concurrent access to files. Therefore, internally data of a file is physically scattered among a subset of the available servers and their I/O subsystems. This enables those servers to participate in one I/O operation thus bundling their hardware resources to achieve higher aggregated performance.

This section extends the description of storage in Chapter 1 in three directions. First, by introducing representative enterprise and parallel file systems, important fundamental concepts are described. This enables assessing alternative I/O architectures and client-server communication protocols. Therewith, it helps us to define the scope of the architectural model and communication protocol that should be implemented in the simulation; the model should be flexible enough to represent several file systems. With the *Parallel Virtual File System* (PVFS) the architecture of one representative parallel file system is discussed in detail. Then, the I/O path of PVFS is described. This archetypal path is general enough to represent communication optimization strategies available in parallel file systems. With this knowledge, the I/O

[1]Note that a few passages are based on descriptions given in the author's master's thesis [Kun07].

[2]It is my personal opinion that all mentioned third-party software and published paper deserve a tribute because of the time spent by the authors to provide helpful tools and since they drive computer science forward. It is also my belief that software is never static and must be maintained to deal with new user and platform requirements. Such a dynamic software is probably never bug-free, and at any time some desirable features are missing. If I argue about missing features or suboptimal solutions in this chapter, then it is done in order to distinguish this thesis from existing approaches – all of the references deserve to be honored. It is not my intention to highlight or to criticize any of the papers or software mentioned. And I hope I succeeded in honoring work cited in an objective and constructive way.

path used in the simulator can be assessed better. Furthermore, it supports performance considerations which are discussed in the next section.

But first, a few basic terms:

Hierarchical namespace For convenient access of data, sequences of bytes are organized in file systems. The namespace is a concept that describes the organization scheme. Traditionally, file systems organize data in logical[3] objects: files and directories[4]. Files contain raw data. Internally, a *file* is like an array of bytes that can grow or shrink at the end. Thus, data must be serialized into a sequence of bytes to be stored. Directories structure the namespace by allowing to put other file system objects into them and give them a name, which results in a hierarchy of "labeled" objects.

This common organization scheme for file system objects is called *hierarchical namespace*. An example namespace is provided in Figure 1.3. By knowing the absolute path name within the hierarchical namespace, a logical file is unambiguously identified. Specific bytes of data can be addressed by referring to the offset within the "array" of data and the number of bytes (the size) to read or write . This type of addressing is referred to as *file-level* interface. However, the interpretation of the accessed bytes must be known to the program accessing it. For convenient access, a hierarchical namespace supports typically alternative names for individual objects, i.e., a single object can be found under different absolute file names. This is achieved by storing a reference to the original data (or directory) under an alternative name – this reference is called link. With a *global namespace*, the file system hierarchy can be accessed from multiple components in a distributed environment.

Beside the hierarchical namespace, there are other data access paradigms: In the cloud storage provided by *Amazon Simple Storage Service* (S3), data is referenced by a bucket (which can be thought of as a folder) and by using a key (similar to a file name). Arbitrary information can be stored for a given key. With the *Structured Query Language* (SQL), databases offer a high-level approach to access and to manipulate structured data. In contrast to a file-level interface data is managed on a higher level of abstraction: stored data is structured, every element of the structure has a label and datatype. Also, with SQL a user specifies the logic of the operations to perform with data and not the control flow that defines execution[5].

Client and Server Applications (and their processes) that access objects of a distributed file system are referred to as *clients*. In the context of hardware, the term describes a node hosting at least one process accessing the distributed file system. In this sense a node that provides parts of the parallel file system to a client is referred to as *server*.

(Meta)data Data refers to the raw content of a file. *Metadata* refers to the information about files and other file system objects themselves – the organization of objects in the namespace and their attributes. Attributes like timestamps or access permissions describe file system objects further. Usually, data and metadata are treated differently within the file system because of the semantic difference and the amount of stored information.

Block storage File systems use *block storage* to persist data. Block storage offers a block-level interface: Storage space is partitioned into an array of blocks that can be read or written individually. Access must be performed with a granularity of full blocks (typically 512 byte or 4 KiB), that means no block can be accessed partially. A number in a linear space specifies which block to access – this scheme is called *logical block addressing* (LBA). The relation between the blocks a file system object is made of is not defined on the block-level and must be managed by the file system.

[3]The term logical refers to the fact that this kind of object is accessed by using the file system interface. Internally, the file system might use several objects that work together to look like the logical object (e.g., file) to the user.

[4]Directories are also referred to as folders.

[5]Although there are procedural extensions, SQL is a declarative programming language.

A block device is a single hardware component that offers such a block-level interface. Multiple devices can be combined into a larger block storage that looks like a single block device.

2.1.1. Capabilities of Parallel File Systems

There are several requirements for file systems: persistence, consistence, performance, and manageability, just to name a few. *Persistence* describes that stored data can be accessed any time later. Also, the namespace should be in a correct state and all data read should also match the data that has been stored (*consistence*). Both requirements are vital because production of data is costly, and reading of corrupted (wrong) data can be disastrous. *Performance* describes the requirement that the file system should be able to utilize the underlying block storage efficiently[6]. Tools must be supported to mount the file system, to check the correctness or to repair a broken file system.

Additionally, for parallel file systems scalability, fault-tolerance and availability are of interest. *Scalability* of the file system allows deployment of the file system to provide sufficient performance for arbitrary workloads and to operate with any number of clients; performance just depends on the provided hardware resources and is not limited by the software. A *fault-tolerant* file system can tolerate transient and persistent errors to a certain extent without corrupting data, although the file system might be unavailable while errors are being corrected and it might crash when an error occurs. *Availability* describes a file system that continues to operate in the presence of hardware and software errors (usually with degraded performance).

The concepts of scalability and fault-tolerance will be briefly illustrated. In the context of this thesis other features are not relevant and therefore subsequently disregarded.

Scaling performance and capacity The architecture of most existing file systems and appliances can handle the demand of any customer ranging from small to very big (and fast) systems – these architectures are considered to be *scalable*. Furthermore, the requirements for an installed storage system might change over time. Since a storage system is costly and management of multiple systems is difficult, it is also important that the existing system can *scale* with increasing storage demands, either in performance or capacity. Otherwise, the money invested will be lost. Therefore, major enterprise file system vendors offer seamless upgrades of already installed solutions.

In the example storage system depicted in Figure 2.1 – it can be imagined that the single NAS server may be a bottleneck. With a scalable architecture, the storage system could be upgraded easily, with minimal modifications to the existing infrastructure; in the example additional NAS servers could be integrated.

There are two orthogonal principles to extend the capabilities of an existing system: *scale out* and *scale up*. The term *scale out* (or *scale horizontally*) is used to advertise modular systems which allow adding more storage nodes as they are needed – additional storage nodes add further capacity and performance at the same time. In contrast *scale up* (or *scale vertically*) means to equip the existing infrastructure with faster components; for example, by replacing a complete node or a single hardware component of the server. While upscaling is limited by the capability of available hardware technology, a scale-out solution provides high extensionability by adding more components. Both principles also apply to cluster computers.

To enable scaling, vendors typically put a distributed file system on top of storage nodes that aggregates all resources into a global namespace; data of a single file is distributed across all nodes. Thus, with an increasing number of storage nodes, the available capacity and performance can be scaled horizontally to any demand.

[6]Performance aspects of file systems are discussed explicitly in section 2.2.

Fault-tolerance Fault-tolerance as a requirement for high availability in parallel file systems is usually provided by replicating data over multiple devices in such a manner that permanent failure of a single component does not corrupt the file system integrity. Keeping a complete copy of data (mirroring) is costly as twice as much space is required. Therefore, mainly error-correcting codes such as the *RAID* levels 1 to 6, or *Reed-Solomon* codes are implemented.

Traditionally, error-correcting codes are applied on the block-level, in most cases multiple block devices are combined into a *redundant array of independent disks* (RAID). The RAID looks like a regular block device to the file system. If an error occurs and a block device of the array must be replaced, then the data of the broken device must be reconstructed by reading data from all disks and writing the lost information to the new block device.

File-level RAID is a software concept in which the file system controls the redundancy explicitly – for every file, the locations of the file blocks including the redundant data blocks are known. In contrast to block-level RAID, this has the advantage that hardware problems in the system only trigger a rebuild of the currently used space. Thus, empty (not-allocated) space of the system is not rebuilt. Failures that happen during the rebuild of a file, do not invalidate the whole RAID and thus file system – instead, the broken file can be identified and reported.

2.1.2. State of the Art

This section describes some parallel file systems that are deployed in enterprises and in a HPC center. This will allow us later to design an abstract communication protocol to simulate their interactions. Since performance of a file system depends on the communication path between client and servers, the focus of this section lies on the I/O path.

Enterprise storage In enterprise business, often a complete storage "solution" or *appliance* is purchased that is a bundle of hardware and software which can be integrated seamlessly into the existing communication network. Several commercial vendors for enterprise and datacenter storage sell *Network Attached Storage* (NAS) systems. Well-known vendors of network storage are: Panasas [NSM04], Netapp, Xyratex, BlueArc and Isilon [Kir10].

A customer connects the purchased storage system to the existing network infrastructure and accesses storage on the new system via standardized remote network protocols like the *Network File System* (NFS) or the *Common Internet File System* (CIFS)[7].

With NFS and CIFS, a remote node or workstation connects to exactly one file server (this scheme is illustrated in Figure 2.1). Clients forward file system operations to this server which executes them on a local file system. The block storage persisting this file system can be either integrated into the server, attached to it or provided in a *storage area network* (SAN)[8]. Recently, distributed file systems rely on *Object-based Storage Devices* (OSD) [Pan04] as underlying storage. Compared to low-level block devices, object-based storage provides a higher level of abstraction – access is performed by addressing file system objects directly. See Figure 1.7 and the descriptions on Page 9 for an example data access on a file system and the underlying block storage.

Panasas ActiveStor With *ActiveStor* Panasas delivers a performant and extensible system. The customer buys blade enclosures, which are equipped with a number of metadata servers (so-called director blades) and/or storage servers (so-called storage blades). Internally, the parallel file system *PanFS* distributes data across the available hardware. Metadata operations are performed on director blades, while data is stored on storage blades. The storage grows by adding further metadata and storage server blades or new

[7] An advantage of using those protocols is that they are available for most operating systems.

[8] A SAN is an additional network for block oriented storage. Devices that is part of a SAN, is usually addressed with the SCSI protocol. A single storage can be utilized from multiple servers, although not concurrently.

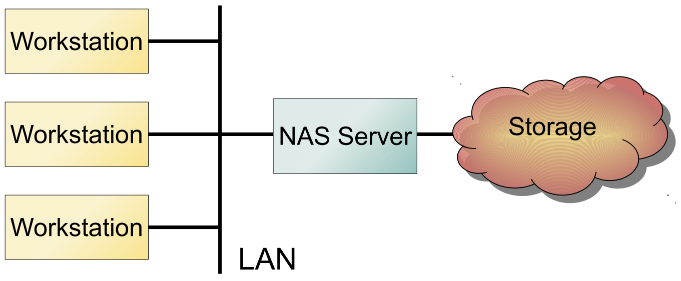
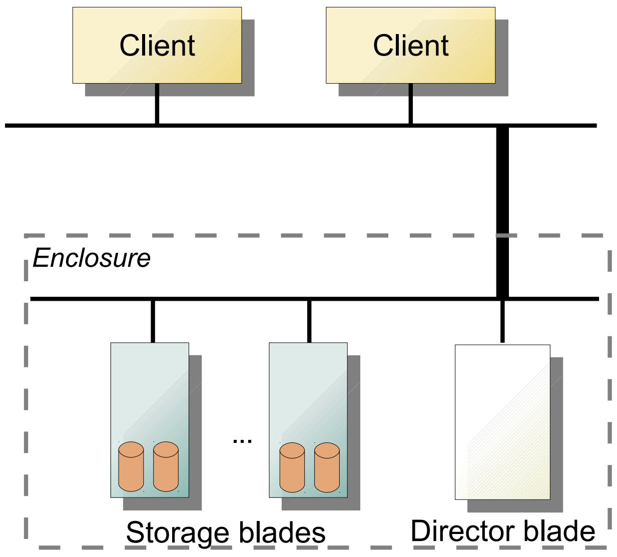
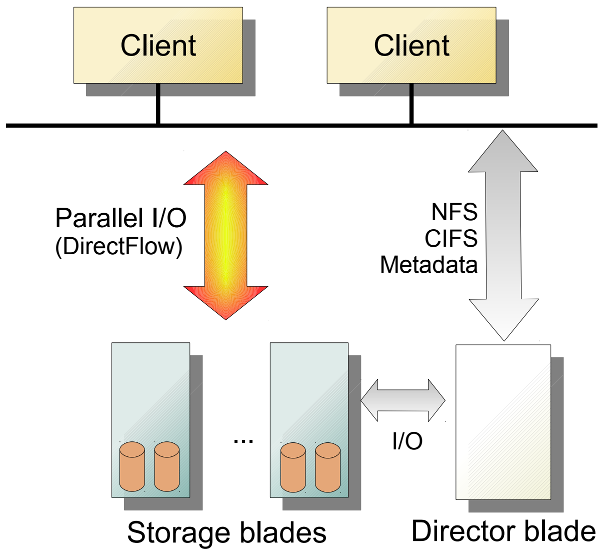


Figure 2.1.: Logical view of network attached storage. Multiple clients can access a central server that manages the persistent storage.



(a) Physical view – an enclosure holds 11 independent blade servers interconnected via a fast network link to the LAN. This example depicts a situation with a single director blade.



(b) Logical view. Clients communicate via a director blade when they use NFS or CIFS. By utilizing DirectFlow, they can communicate with the storage blades directly.

Figure 2.2.: Panasas ActiveStor system.

enclosures. A logical and physical view of an example system consisting of one enclosure is shown in Figure 2.2.

Enclosures have a fast interconnection to the *clients*, for example via 10 gigabit Ethernet (10 GbE). Internally, the enclosure is interconnected with each blade system by a slower 1 gigabit Ethernet – the name corresponds to the nominal data rate, for example, 10 GbE supports a rate of 10 GBit/s. Each blade holds two disk drives (or SSDs) to persist the file system.

Proprietary software can be installed on a client to allow it to interact directly with PanFS. In essence, the so-called DirectFLOW communication protocol implements a variant of the parallel NFS (PNFS) [HH07][9]; block I/O is performed with iSCSI, while an OSD protocol addresses the file system objects.

A client can also access storage via the NFS or CIFS protocol. In this case, clients simply mount the file system on one of the director blades with the one of this protocols. However, all I/O is communicated via the selected director blade, which forwards the I/O to the storage blades. Internally, the director blade translates the client operations into file system calls, and thus accesses PanFS (basically like a client which supports DirectFLOW).

[9]In fact, Panasas are member of the consortium for PNFS and share their file system knowledge to establish an industrial standard that might replace DirectFLOW in the future.

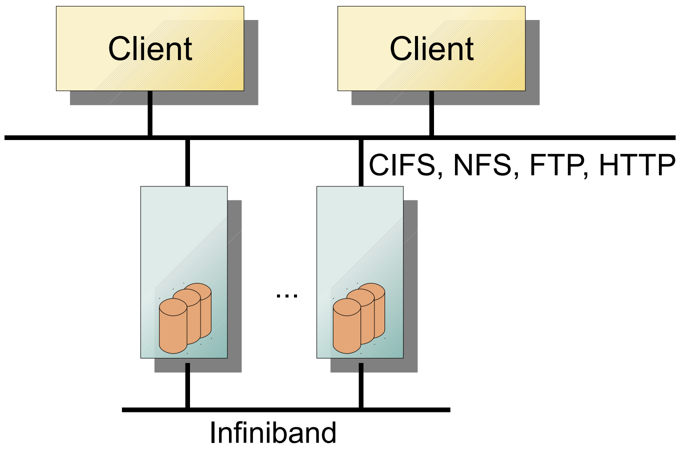


Figure 2.3.: Abstract view on an Isilon scale-out solution.

Isilon S-Series Isilon offers a NAS storage system in which the storage nodes have an additional Infiniband interconnect as a backend that balances load within the storage cluster (the *S-Series* [Har09]). Clients transfer data to one storage node by using either the NFS, CIFS, HTTP or FTP protocol. Clients and servers are interconnected with either Ethernet or Infiniband technology. The parallel file system *OneFS* orchestrates the servers into a coherent global namespace: Data is distributed across all servers of the storage system, the servers exchange data by using the internal Infiniband network. A schematic view is provided in Figure 2.3.

By supporting protocols in which clients connect to only one server, the server might become a bottleneck. To allow coarse grained load balancing, the assigned storage node is rotated by using *DNS round-robin*, that is, each DNS request is assigned to a different storage node. Each node hosts up to 36 storage devices and the scale-out system is expandable up to 10 PByte.

Both Isilon and Panasas support RAID protection on file-level.

Parallel file systems in HPC For HPC it is important that all processes of a parallel program can access the file system efficiently. A storage solution of the enterprise market might not match the architectures of supercomputers well, and thus may not provide the required performance. Performant concurrent access of multiple processes to a single file is especially difficult. For those reasons, only a subset of the existing enterprise solutions can handle HPC workloads.

The parallel file systems Lustre [SM08, YV07] and IBM's General Parallel File System (GPFS) [SH02, BICG08] are widely deployed in HPC environments – currently, 15 out of the 30 fastest systems use Lustre. GPFS is installed on many lower ranked systems of the Top500 list, too. Besides those file systems, mostly enterprise systems serve the storage needs of smaller supercomputers.

Xyratex offers Lustre based storage servers. In contrast to solutions mentioned from other vendors, this enables truly parallel access without modification of the client, i.e., the standard Lustre client can be installed. In the long term perspective, the parallel NFS (PNFS) will become widely supported, thus, allowing parallel I/O across I/O systems from various vendors.

The Parallel Virtual File System (PVFS) is an open source file system developed for efficient reading and writing of large amounts of data. There are many file systems designed with similar principles as PVFS, therefore, PVFS introduced as an archetype for parallel file systems.

2.1.3. The Parallel Virtual File System PVFS

PVFS is designed as a client-server architecture – servers provide storage and the client contains the logic to access this distributed storage. Multiple file systems can be served by one server infrastructure. According

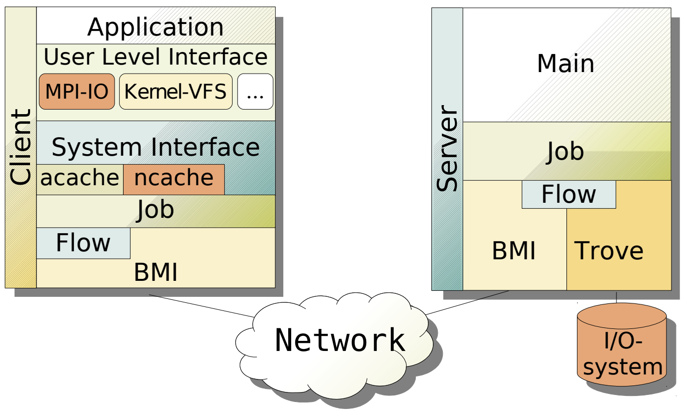


Figure 2.4.: PVFS software architecture.

to the type of storage provided, servers can be categorized into data servers and metadata servers. Data servers maintain parts of logical files. Metadata servers store attributes of logical file system objects – their metadata, and the namespace. A server operates either as metadata server or data server, or both at once.

Logical file system objects which can be stored in PVFS are files, directories and symbolic links. Internally, additional system-level objects exist: *metafiles* contain metadata for a file system object, *datafiles* contain pieces of the file data and *directory data* objects map names of logical objects to internal objects. A logical file is represented as a metafile which refers to several datafiles – file data is partitioned among the datafiles which are then placed on the available data servers. In PVFS neither file data nor metadata is replicated explicitly. Instead, configurations aiming for high availability rely on shared storage, i.e., storage that is accessible from multiple servers over a network – SAN storage for example.

Architecture PVFS uses the layered architecture illustrated in Figure 2.4. One of the main advantages of a layered architecture is that an implementation for a layer can be replaced, in order to match the underlying hardware.

The *user-level interface* allows access to a file system with a standardized interface – PVFS provides interfaces for the Linux kernel, the *Filesystem in Userspace* (FUSE)[10], and MPI. The kernel interface integrates PVFS into the kernel's Virtual File System Switch (VFS), a user-space daemon connects to the kernel module via the `/proc` interface and communicates with the servers.

The *system interface* enables direct manipulation of file system objects, yet some internal details are hidden from the user. Internally, the processing of file system operations is modeled with finite state machines – states can fetch information from a server, or send a request to a server to modify the file system.

Several *client-side caches* are incorporated to reduce the number of requests to the servers. An attribute cache (acache) maintains metadata, the name cache (ncache) buffers mappings from file names to internal objects.

The *job layer* is a thin layer consolidating all lower interfaces – BMI, Flow and Trove, into one interface.

The *Buffered Message Interface (BMI)* provides a network independent interface for message exchange between two nodes. Clients communicate with the servers by using the so-called *request protocol*, which defines the layout of the messages for every file system operation. Depending on the underlying network technology like TCP or Infiniband, the appropriate communication method can be selected.

Trove interfaces with the storage subsystems to store the system level objects. Data can be either stored as a key/value pair or as a bytestream. Key/value pairs are used to store metadata, while byte streams keep the file data. By default, PVFS uses multiple Berkeley databases to store the metadata and regular UNIX files to store bytestreams.

[10]`http://fuse.sourceforge.net/`

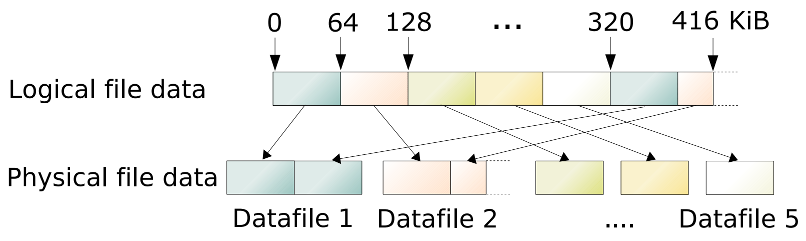


Figure 2.5.: Exemplary file distribution for 5 datafiles – data is split in 64 KiB chunks and striped over the datafiles in a round-robin fashion.

The *I/O subsystem* indicated in the figure can be any block storage – disks integrated in a SAN or direct attached storage, usually RAID systems. When a SAN is deployed, PVFS can be configured for high availability, i.e., to tolerate server crashes.

A server's *main loop* drives the processing of communication and I/O layers. Also, the server checks for new requests and initiates state machines to process them.

On the server side, it is important to overlap I/O with communication, the orchestration of data streams between two endpoints is performed by *Flow*. Data might flow between memory, BMI, or Trove. Data on the server flows between network and I/O subsystem, while on the client side it is transferred between memory and the network. The flow protocol defines how data is transferred between two endpoints including the caching strategy. By default, PVFS allocates 8 data buffers for every I/O request, a buffer is either accessed by the network or the storage layer. That is, up to 8 operations can be pending on Trove or BMI – once all buffers are filled due to a congestion on I/O subsystem or network, further operations are deferred.

Data distribution of a logical file A selectable *distribution function* defines how data of a file is distributed among the different datafiles (often also referred to as *stripes*). By default, file data is split in 64 KiB chunks which are distributed over all file servers. Similar to a RAID-0, the first chunk is mapped into the first datafile, the second chunk into the second datafile, until one chunk has been mapped to each datafile. Then, the next chunk is assigned to the first stripe again, until all chunks are assigned. The mapping of a logical file with a size of 416 KiB into 5 datafiles is illustrated in Figure 2.5. A datafile is mapped to exactly one server; this way file data is distributed among the available servers.

When a user creates a new logical file in PVFS, the datafiles are created and assigned to the file servers. Actually, PVFS tries to pre-create logical files and keeps them in a pool to avoid the overhead of creating and mapping of datafiles.

Small files might never use all datafiles; thus, the empty datafiles cause overhead during file access and they crowd the underlying local file system. Therefore, some parallel file systems like PanFS from Panasas grow the number of stripes dynamically. In this concept, small files are first stored within the metadata, then the number of stripes grows with the file size – while appending new data, existing data could be redistributed among the servers to re-balance it optimally, or just be kept in place.

2.1.4. Client-Server Communication

Due to the distributed nature of the clients and servers, a protocol defines how the clients interact with the servers to access and manipulate the file system. In contrast to Isilon's OneFS, with PVFS (or GPFS) a server accesses only its own storage and thus, knows local objects. Therefore, no server has an overview of the complete file system. To perform an operation, a client gathers intelligence where metadata and data reside and communicates with the required servers. A slightly simplified client-server protocol of PVFS is discussed here, but it is close to the request protocol actually used.

If a client wants to access file data, it first figures out on which metadata servers the metadata about this file is found by traversing the namespace hierarchy. In the metadata the information about the data distribution, i.e., the RAID layout, is found and the object IDs for all datafiles the file is composed of.

Now, depending on whether a read or write is requested, a different sequence of operations is performed; either way the client contacts each data server that holds data of this particular file.

In the *read path*, the server sends an acknowledge including the amount of data which can be read, the server then transfers the data to the client. When all data is received, the client knows that the read has been completed.

In the *write path*, an acknowledge is sent indicating that the server permits the write operation. Then, the client sends its data to the server which starts to write back the data; usually, data is cached in main memory first. Once all data is written to the cache, the server sends a completion message to the client.

In the client-server communication, a small amount of data can be piggy-backed to the initial request or response message and thus avoids additional data flow. Bigger requests start the rendezvous protocol mentioned above. The amount of data which is transferred with the initial request depends on the network module, for TCP/IP 16 KiB are allowed.

To support concurrent network and file operations, the accessed data is split into a stream of buffers transferred between client and server. By default, up to 8 streams of 256 KiB each are used. Write operations on the I/O subsystem are initiated once all required data of the current buffer is received from a client. Once a read request is received, all eight operations are issued to Trove. When one of the read operations complete, the buffer is ready to be transferred by BMI.

PVFS supports *non-contiguous I/O* to reduce the number of messages and the file system overhead – i.e., with one request a user can access several non-overlapping file regions. Without this feature one request must be created for each file region. There are several possible methods to announce the non-contiguous file regions. The term *ListIO* indicates that a list of non-contiguous accesses is transferred in a single request. The user can invoke an I/O call with a list containing multiple size and offset tuples. PVFS additionally supports several derived datatypes similar to the structured data types of MPI, in fact the structures of PVFS are some kind of lightweight replacement for MPI datatypes.

Usually, multiple remote clients access a parallel distributed file system simultaneously. Therefore, a mechanism must be implemented, either in the clients or in the servers, to prevent potential corruption of the file system. Especially concurrent modifications of the namespace are dangerous. In PVFS a *request scheduler* defers conflicting metadata operations. Concurrent access to a file's data is allowed but concurrent writes might corrupt data (further details of the semantics are given in Section 2.3.3).

2.2. Performance of Parallel Applications

In this section factors contributing to application performance are discussed. First, in Section 2.2.1, relevant components and layers are identified from the perspective of the application. Then, particular hardware characteristics and software strategies to mitigate negative hardware characteristics are discussed in Section 2.2.2.

In this discussion the performance of the application is attributed to four areas: the raw performance is supplied by the *hardware*; the *computation performance* is the speed at which the desired calculation of the result is performed; the *communication performance* addresses inter-process communication of a parallel application; finally, the *I/O performance* is required to read input data and to stage results.

Communication is actually an undesired byproduct of the collaboration – at best it can be avoided – as it does not contribute to computing of the parallel application, instead it just enables several independent

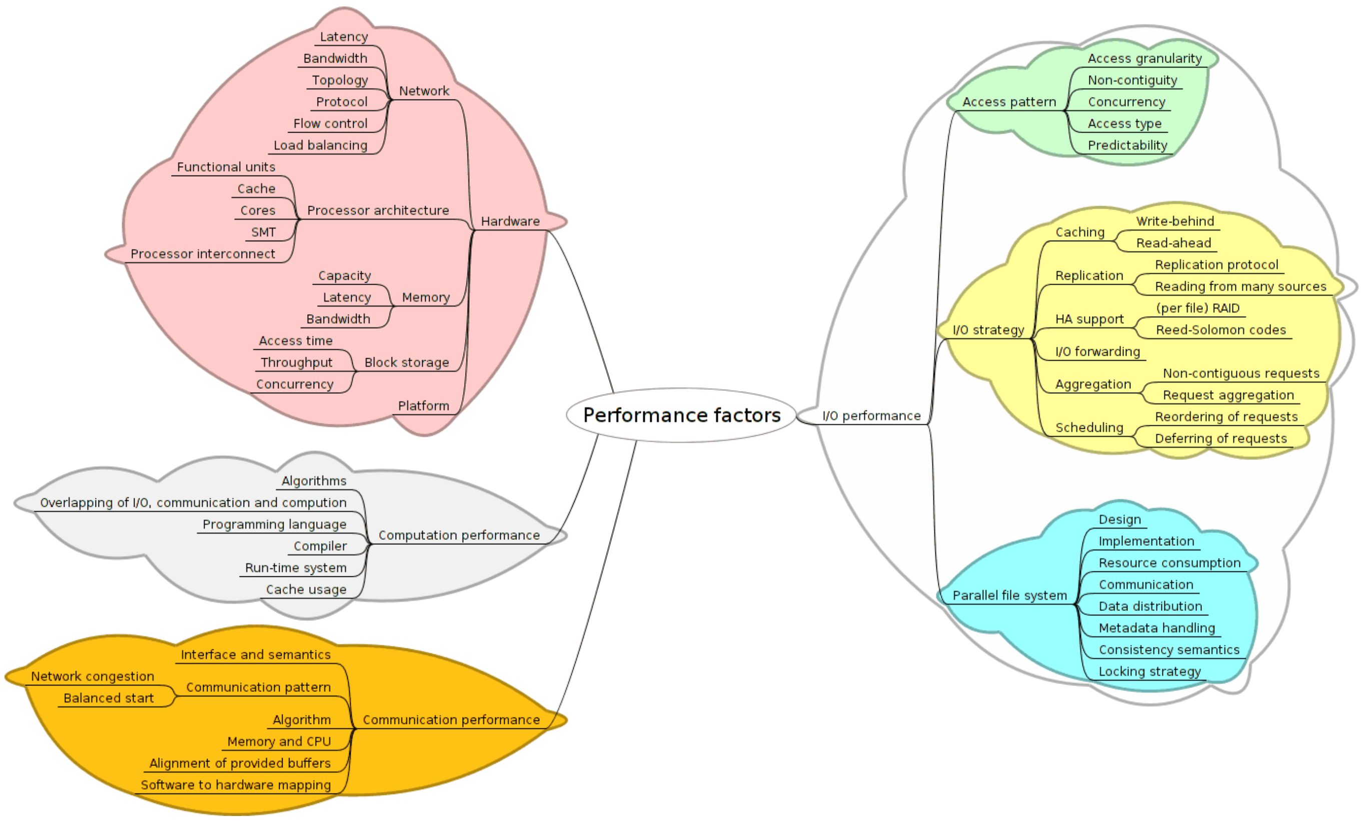


Figure 2.6.: Performance factors. Hardware aspects are orthogonal to computation, communication and I/O performance.

hardware components to contribute to one computation[11]. Intermediate I/O such as for checkpointing are similar to communication in that respect.

Generally, scientific applications are typically composed of *phases* of activity. Consequently, some regularity of computation, communication and I/O phases is expected. Former evaluations showed that most scientific applications tend to regularly access data with similar patterns [MK91, PP94, SACR96, Rot07, CHA+10].

Relevant characteristics and factors are summarized in Figure 2.6. Note that all hardware and software components have diverse character and performance limitations and thus this list does not claim to be complete. Main goal of the discussion is to increase awareness of the diversity of the performance influences and optimization potential.

The optimization strategies introduced are a double-edged sword: On the one hand, they posses the potential to improve performance. On the other hand, they might be counter productive by degrading performance in some cases. To mitigate the risk of performance degradation, exact knowledge of the processing layers below is necessary; then, optimizations could be turned off if they are not appropriate for the situation.

2.2.1. Performance Relevant Hardware Components and Software Layers

In an abstract view all hardware components and software layers which are involved in executing an application contribute to the observable activity and measurable performance – refer to Section 1.1.2 on Page 7 for more detail on software layers in parallel scientific applications.

[11] Reduction operations, as part of many parallel programming models actually perform some wanted calculation, but this could be done without communication as well.

Either an application triggers activity on other components directly or indirectly by calling intermediate software layers, or concurrent background activity of hardware and software interfere with the desired computation of the result. Activity, which is caused by the application neither directly nor indirectly is referred to as *external* activity in this thesis. Both the internal (application caused) activity and external activity is performed on the same computer system.

As resources provided by any system are limited, a program and background activity (or to be more formal, the sequence of instructions actually executed) could utilize some resources to a bigger extent than others. If the hardware does not match the characteristics required by a sequence of instructions, then the limiting characteristic is a bottleneck for those instructions – execution of further instructions is deferred until the required operations are finished. Since background activity requires resources as well, internal activity is influenced by external activity.

Due to the characteristics of the application it is typically impossible to utilize all resources at 100%; consequently, some resources are utilized in a suboptimal way. As the requirements might change during run-time, the bottleneck can be another component or layer. In the application example in section 1.1.3, for instance, the application reads data, communicates, computes the local results and then collects the local results and aggregates them into the desired output. While the computation is performed, the speed of the CPUs determines the performance, during communication the network infrastructure and communication library define how long the communication takes – the other components are not required and idle[12].

Another view of the layers and components involved in application performance is given in Figure 2.7. Here, several of the software layers are grouped by their basic function:

- The *application*'s internal behavior has certainly a high impact on performance, the programmer could modify the design, algorithm, source code or assembler code to achieve better performance. This is also true for all further software layers, such as libraries, run-time system or OS. Source code is also a good starting point for the analysis because resource consumption and execution time can be assigned to each line in the source code.
- *Libraries* provide all kinds of functionality – usually provided by third parties – which are added to the original application. These could be domain specific libraries or high-level I/O libraries.
- The *run-time system* is the software that provides an environment to execute the application in. Python and Java are examples of programming languages that translate the application code into machine code at run-time. Therefore, efficient intermediate code representation and translations are provided. *Partitioned global address space* (PGAS) languages provide their own run-time system to transfer data between remote processes; optimizations for network communication, which will be discussed later, apply to them as well.
- Access to hardware is controlled by the *operating system*, also basic functionality is provided. The operating system incorporates several strategies to optimize throughput and concurrent processing of programs. On this level, a proper configuration of the hardware towards the needs of the upper layers is required. Also, background activity of concurrently running software or operating system daemons can disturb the execution of the user's application.
- *Hardware* provides the resources which are used for computation and I/O. The orchestration and concurrent utilization of all hardware components is desirable to complete the execution as fast as possible. In this sense, knowing that performance is limited by a specific hardware component, and that measured performance is close to theoretical capabilities, provides the foundation to performance optimization and tuning.

In additional to local layers and components, applications might access remote (hardware) resources like persistent storage in a *Storage Area Network* (SAN) or rely on *remote services* like a parallel file system, databases or online visualization. Thus, depending on the environment and application, remote activities might be of interest.

[12]However, in a real system during communication some CPU time is required for the interrupt handling, memory transfer and inter-process communication. But here, the influence is expected to be rather low.

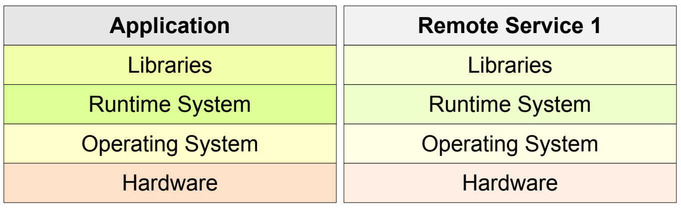


Figure 2.7.: An abstract view of layers and components involved in application execution.

Next, the hardware aspects contributing to performance are examined. Software aspects contributing to performance are discussed in more detail in Section 2.2.3.

2.2.2. Hardware

Clearly, observable performance is limited by the physical capability of the hardware to perform computation, communication and I/O. In the following, a subset of relevant hardware characteristics is provided.

Network Since a network transfers data between two independent nodes, it limits the communication capability. To share the available links among all communication streams, most technologies utilize the communication method of *packet switching*: Data is fragmented into *packets*[13], which are transported between source and target. This allows to multiplex available network links among multiple streams. Common network technologies are *Ethernet*, *Infiniband* and (for storage servers) *FibreChannel*. Many supercomputers deploy their own custom network interface.

Within the scope of this section, important hardware characteristics are[14]:

- *Latency*: the delay between the moment network transfer of a message is initiated, and the moment data is starting to be received. Latency can be referred to as *propagation delay* as well. It is measured either one-way, or *round-trip*, which is the one-way latency from source to destination plus the one-way latency from the packet destination back to the original source. Round-trip time can be easily measured between two machines by inducing low-overhead network communication. Tools like *netperf* are designed to determine the network latency and throughput on top of the network protocols. Latency has a high influence on small messages, e.g., metadata operations in file systems, but it becomes negligible for large messages. Each inter-process message involves latency, therefore it is favorable to reduce the number of messages by packing more data into one big message.
- *Bandwidth/Throughput*: In computer networking, the term bandwidth refers to the data transfer rate supported by a network connection or interface – how many bits are transferred over the wire. Bandwidth can be thought of as the capacity of a network connection. Network protocols and encoding of information add overhead to the actual transferred data and thus the effective *throughput* of the hardware is lower than the actual available bandwidth. For this reason, the term throughput is preferred in this thesis.
- *Topology*: is the physical (and logical) arrangement of a network's elements like nodes and interconnections between them. In a fully-meshed network each node is able to communicate directly with each other node. Unfortunately, realization of this topology is very expensive in reality and almost impossible for large numbers of nodes. Common topologies try to mitigate this problem by reducing the number of links, but still aim to provide a high bandwidth and a low latency. To reduce the number of communication links an indirect network topology either restricts communication to a subset of neighbor nodes[15], or it uses hierarchies such as trees.

[13]Packets on an Ethernet link are called *frames*.

[14]Further information on the topic of networking is offered in [PM04, PD11].

[15]These nodes, in-turn, are able to route the packets further towards to the destination (for instance the 2D-Torus, which has been shown in Figure 1.2).

A *star topology* interconnects all components via an intermediate switch resulting in a distance of one hop for every communication. It is frequently deployed on small clusters. With this kind of indirect communication, however, the bandwidth of the intermediate network components and links is shared amongst all packets which are routed through the same network path. Thus, the topology determines the maximum throughput which can be achieved transferring multiple data streams between different nodes concurrently.

Tightly coupled applications need a matching topology to communicate efficiently. For a parallel file system, it is important that the amount of data which can be shipped between clients and servers is at least slightly higher than the maximum aggregated physical I/O throughput. Otherwise the available devices cannot be fully saturated, and caching mechanisms on the server-side are not beneficial.

Bisection bandwidth is a valuable metric which provides insight into the fundamental performance limitations of a topology. Bisection bandwidth is defined as the minimum sum of the bandwidth of the links, when the network is partitioned into two sub-graphs of equal size.

- *Protocol*: Network protocols like the Transmission Control Protocol (TCP) define how communication is established between sender and receiver and define rules for data exchange between the participants. Furthermore, guaranteed properties for the data transfer (quality of service) like reliability or minimum throughput of the data transfer could be provided by a protocol.

  Effectiveness of the communication is a crucial factor. It is desirable that hardware is exploited already with small messages, and that a high transfer rate can be achieved in the presence of transfer errors, for instance caused by package loss.

- *Flow control*: Amounts of data larger than the packet size are partitioned into smaller chunks which are then transported over the network topology. There are several ways to realize the flow of individual packets.

  > *"Flow control is the process of managing the pacing of data transmission between two nodes to prevent a fast sender from outrunning a slow receiver. It provides a mechanism for the receiver to control the transmission speed, so that the receiving node is not overwhelmed with data from transmitting node."* [Wik11]

  The typical network communication method of packet switching, as used by Ethernet and Infiniband, does not provide flow control by itself. In detail, data streams (or messages) are fragmented into packets and transferred between the network components in the following way: In the simple *store-and-forward switching*, packets are stored completely on an intermediate node, i.e., the whole packet content is stored in a buffer, once the link towards the target turns idle, buffered data is transferred. One drawback is that with slower sinks the buffer space might not suffice and thus packets are lost – this must be covered by the network protocol to ensure reliable communication. A common approach is to check for lost packets and resend them.

  Cut-through switching tries to avoid receiving the complete packet. Instead, once the packet header is received it determines the target link. If the target link is not busy right now, then the header is sent to the target and the packet data is streamed directly to the outgoing node. If the link is busy, then the packet is buffered similar to the store-and-forward switching.

  In *wormhole switching*, the route of a message is defined during startup of the communication, then subsequent packets of the message are all shipped over the same route. Since a virtual channel is reserved to transmit data from a particular sender to an receiver, effectively only one communication occurs over a link[16]. An advantage in reserving virtual channels is that no buffer space is required, data can flow directly from source to sink – without buffering it. Thus, wormhole routing provides flow control on the network layer; packet loss that is observable with the two other switching methods cannot happen. Also, the latency is lower because the intermediate nodes do not have to receive the

[16]With virtual channels multiple concurrent communications are possible and share the resources, but a discussion is out of the scope.

whole packet before resending it. Bandwidth reservation is an extension to this concept that allows utilizing a link only partly.

Flow control can be directly implemented by a network protocol such as TCP to avoid outpacing a slow receiving process. Therewith, flow control on the network level is achieved, too.

- *Load balancing*: Depending on the topology, a dynamic load balancing could transfer packets over a less utilized path. For example, in the 2D-Torus topology sometimes two neighbors are possible candidates to transfer data towards the destination node. Fat-trees do not permit load-balancing as there are no alternative routes between source and destination. In a hierarchical topology like the Clos network, multiple paths to one target exist as well, but often load balancing is performed implicitly by distributing packets in a round-robin fashion among all switches. There is a rich variety of approaches available to balance the network load; refer to [LFL09] for further reading.

Processor architecture Capabilities of the CPU determine the time needed to process the instructions of the application and a parallel file system. While there are so many aspects of a processor microarchitecture, the following characteristics are contributing a lot to the performance:

- *Functional units:* Depending on the micro-architecture, multiple instructions can be dispatched to independent *functional units* in one cycle. IBM's Power6 and Intel's Nehalem architecture are instances of those so-called *superscalar* architectures. For example, the Power6 can process two *fused multiply add* instructions per cycle. Several CPU internal optimizations try to utilize the functional units in the best way; widely known approaches are branch-prediction, translation look-aside buffers and speculative execution.
- *Cache:* Offers very fast, but small memory. Several levels of cache and the main memory form the memory hierarchy of the system. To achieve optimal performance, the size of the cache must suffice to hold the *working set* of an application. The working set can be thought of as the data that is accessed frequently during an execution phase. If it does not fit in the caches, the data must be fetched from main memory which reduces performance.
- *Cores:* Nowadays, all processors contain multiple interconnected logical *CPUs* on one silicon die – called *cores*, each is capable of processing one independent thread. Clearly, a higher number of cores enhances raw computational power. *Simultaneous multithreading* (SMT) enables a CPU to schedule multiple threads concurrently; the functional units are shared among them. With SMT, latency of memory access from one thread can be hidden by processing instructions from the other threads in the meantime. Hence, an SMT CPU tries to utilize the existing functional units by scheduling instructions from multiple threads.
- *Interconnect:* When multiple processors are installed in one node, then those processors are interconnected with each other to provide a *cache-coherent*[17] view of data in memory. The processor *interconnect* defines the efficiency of the inter-processor communication. In systems with more than four processors, their layout forms a topology similar to the network topology. Actually, a communication network interconnects the microprocessors.

  Some processors might be attached to the bus system of the I/O devices while others must communicate via intermediate processors. Hence, the communication is made more efficient if the right processor is used for communication and I/O.

Memory In the *von Neumann architecture*, memory contains data to process. Memory supports random-access to individual data words. See also Page 4 for the interplay between CPU and memory. The memory technology implies a specific latency and bandwidth, while the capacity (the amount of memory) can be extended depending on the processor and system architecture.

[17]This means that data manipulations done by one processor are immediately visible to all other processors and prevents that an old state of the data is accessed accidentally.

Capacity: Available memory can be either used to hold processing data or to cache communication and I/O. By caching data performance can be improved significantly. For a parallel file system, a large server cache is vital because it allows performing optimizations that help to saturate the I/O subsystem (see below).

Typical main memory has a *latency* in the order of 100 processor cycles. Therefore, if the processor requires a particular piece of data, the computation is stalled for several 100 cycles until the data becomes available in the registers.

Bandwidth is the speed at which data is accessible. Similar to network communications, the effective throughput is typically lower.

Block storage The block storage provides a random-access storage which is used to persist data. Block devices are only able to access blocks of a certain size aligned to the block boundaries[18]. Modification of a few bytes of a block requires to read the full block, then modify it and write it back. Also, unaligned reads require more data to be fetched than required. As a consequence, unaligned accesses degrades performance.

Block storage might be either a single directly attached block device, a RAID system consisting of multiple block devices, or a SAN. In the latter, the network interconnect plays an important role, but block storage must be deployed in the SAN as its backend, too. *Direct-attached storage* is either a *hard disk drive* (HDD) or a *solid state drive* (SSD). Nowadays, SSDs are mostly built with flash technology, therefore, the following brief discussion focuses on hard disk drives and flash storage.

Important characteristics of a block device are:

- *Access time*: The time needed until the storage is ready to transfer data. A hard disk needs a few milliseconds to move the access arm to the correct position (*seek time*) and to wait until the block sought rotates under the platter's disk head (*rotational latency*). The time the access arms has to wait until the data is available depends on the speed at which the disk spins – the *revolutions per minute* (RPM). The actuator, which moves the heads of an HDD, accelerates depending on the distance from the current to the next track. Therefore, the time to seek from one position to another is non-linear.

  The disk's characteristics include the *average access time*, which is the average time needed to move the access arm from one random location on the disk to another; this measure is the sum of the *average seek time* and the *rotational latency*[19]. *Track-to-track seek time* is the time needed to move the access arm just to an adjacent cylinder. Full-stroke seek time is the time needed to move the access arm from the first cylinder to the last one. Good values for disks are around 5 ms for an average seek time, 0.5 ms track-to-track seek time and 10 ms for full-stroke seeks.

  An SSD has different characteristics. The access time is usually very low and independent of the previously accessed block on the device. However, write performance can vary depending on the controller and actual flash memory. On the one hand, the slower write path is due to the fact that flash storage must be erased before it can be written, on the other hand, *wear-leveling* costs some performance as it tries to utilize all blocks to the same extent to increase the life-time. In a flash device data can be erased only in blocks of a larger granularity, like 256 KiB.

- *Sustained transfer rate*: Once the heads are placed on the disk, subsequent blocks can be read or modified very fast, which results in a higher transfer rate. However, depending on the underlying local file system and disk fragmentation, additional seeks might be necessary. If the data being accessed is spread over the entire drive, extensive access arm movement is required resulting in low aggregated *throughput*. Thus, it is preferable to issue I/O requests for data that is physically close to other data that has been requested recently. Also, the possible speed of read and write operations

[18]Nowadays, a block has a size of 4 KiB.

[19]In some datasheets and benchmarking results, the average access time is confused with the average seek time. In this thesis, the terms are used according to this definition.

may differ. Hard disks rotate with a fixed speed and contain more data on the cylinders with a larger circumference, therefore subsequent blocks can be accessed faster on the outer cylinders than on the inner cylinders. For an SSD, the throughput depends on the speed data is transferred between the chips and the controller, which should utilize multiple flash chips concurrently.

- *Concurrency*: Depending on the access pattern and deployed block device, it can be useful to issue multiple requests at a time to increase aggregated performance. In case of small requests, a disk scheduler might optimize throughput by reordering requests – for SATA disks this is called *Native Command Queuing* (NCQ). On SANs, multiple pending requests keep the disk(s) busy. Thus, additional network latency due to communication with the SANs does not show up in the aggregated performance, although individual operations take longer.

All technologies deploy mechanisms that enable them to tolerate errors occurring while data is accessed. For instance, a HDD verifies data written to disk by re-reading the magnetic information. Any read and write error detected will cause the disk to spin again over the same track while trying to recover the information. SSDs deploy wear-leveling algorithms to increase the life time of flash memory, but also have reserve cells to tolerate failure to a certain extent. RAID schemes on top of block storage may increase capacity or throughput, and might protect the system from failures – depending on the scheme. While all those mechanisms mitigate and protect from errors, recovering from an erroneous state requires time and resources. Thus, an error degrades performance in a real I/O subsystem, but might be transparent by the user. More information about HDD's is available in [Mey07].

Platform A computer is built with a particular processor platform in mind. The platform includes chipset, internal technology and interconnect to peripheral devices such as I/O devices. For example, *PCI-Express* is a bus-system that is deployed with newer processor platforms. Thus, a platform determines how fast data can be shipped between the different hardware components of a computer. In most cases, the performance provided by the platform suffices to saturate network, disk or memory. However, very fast networks like Infiniband put the internal bus systems under pressure.

Certain optimization techniques and technologies might also be provided by a platform. For example, *Direct Memory Access* (DMA) is an access method that enables data to be transferred between peripheral devices and main memory without copying it through the CPU. Together with the network card, the platform might enable *remote direct memory access* (RDMA). This technique transfers data directly into a remote main memory, without involving the operating system of any of those systems. Thus, data need not be copied between application buffers and kernel[20]. RDMA reduces the communication latency, but usually requires data to be aligned contiguously in memory, i.e., only a contiguous region of data can be transferred with RDMA.

2.2.3. Computation Performance

From the user's perspective, an application computes some valuable result; I/O and communication just help to get the work done. Therefore, computation performance is prime importance. The following aspects are relevant on all software layers and include the operating system. The experience of the developer is important to pick the appropriate algorithm, programming language and compiler for solving a given task. Finally, the coding in the programming language expresses how a problem is solved.

Algorithms are blueprints describing how computation will be performed to calculate the results. An algorithm determines the data dependency and thus communication patterns. Furthermore, certain data structures might be required to perform an algorithm efficiently. Therefore, the algorithmic specification restricts the formulation in a programming language. An algorithm has a computational complexity,

[20]Since data can be transferred without copying buffers is often called *zero-copy*.

approximately defining how the computation time grows with an increasing problem size, i.e., the data itself.

Overlapping of I/O, communication and computation In the best case, the application spends all its time to compute valuable results. In this sense time needed for I/O and communication are wasted. Therefore, any communication and I/O should take place concurrently to computation. Usually, communication and I/O libraries provide non-blocking calls. When such a call is invoked, it initiates communication (or I/O activity) and returns immediately. The actual activity is performed in the background. The possible speedup achievable by overlapping I/O and computation is bounded: if I/O takes the same time as the computation, a speedup of two can be achieved [BKL09]. If communication and I/O can be perfectly overlapped a speedup of three is possible. However, overlapping requires additional memory space to temporarily buffer the I/O (or communication) data until the background activity completes. Also, the programming and debugging of the application becomes usually more difficult.

Programming language The programming language is a formal notation in which the developer tells the system what to execute (in imperative paradigms) or the result he/she is interested in (in declarative paradigms). Expressibility and semantics limit possible formulations of code to solve the problem. The programming language is tightly coupled with the potential optimizations a compiler can apply.

Compiler The compiler parses a programming language and transforms it into another programming language, typically a sequence of machine code, which can be executed on the target machine. While the compilation is performed, the compiler tries to optimize the code in order to increase performance[21]. Compilers provide different sets of available optimizations. Users have to select the appropriate optimizations to achieve efficient machine code; for example, a brief evaluation of compiler optimizations for stencil operators is provided in [KN10].

Run-time system The transformation of *interpreted* languages or *byte-code* into machine-specific instructions depends on the capabilities of the run-time system. Java's just-in-time compiler aggressively optimizes the code regions that are used frequently, while at its first execution, code it is translated quickly in order to reduce translation time. A PGAS language needs a run-time system to manage remote memory access; buffers are used to store local results. In this case, the strategy, such as potential background transfer of data, reduces the communication overhead. Making use of the platforms' RDMA feature of platforms is favorable as it permits modification of remote data without disturbing the remote processing.

Cache usage The ratio of memory bandwidth per floating point operation is rather low at around 1 byte per Flop, hence optimization of memory access is crucial. Caching is a standard practice in computer systems to exploit access locality of computer programs. Most programs need the same data or instruction sequence multiple times (the working set). If the data is already held on media that is faster accessible, it is not necessary to request it from a slower one. As a CPU cache is much faster to access than the main memory, the algorithm and program should operate on a small working set – the more local accesses are performed, the better. Random access to small pieces of data degrades memory performance due to the fact that memory access is performed in larger cache lines of, e.g., 64 bytes. Thus, if only one byte is needed per cache line then effectively 63/64 of the memory bandwidth is wasted. The efficiency of cache re-use depends on the characteristics of memory and the platform (see above).

[21] However, the language semantics must be obeyed in this step even if relaxed semantics would be valid for the given code. If too many optimizations cannot be applied due to the semantics, then typically the programming language is adjusted. For example, this has led to the new `restrict` keyword in C99.

2.2.4. Communication Performance

Cooperation between processes requires information exchange – inter-process communication transfers data between processes. The observable performance is restricted by the semantics of the interface, the actual communication pattern, the internal communication algorithms, memory alignment of the provided data and placement of the processes which communicate. During the communication local resources such as CPU and memory might be required.

Interface and semantics The API and semantics of the communication paradigm defines how inter-process communication is performed. Usually a rich variety of potential operations is provided; this enables the user to pick the best abstraction to realize his communication pattern. Two interaction patterns are common, either two processes communicate with each other in a *point-to-point* operation, or a set of processes exchange information together in a *collective* operation. Operations might follow rather strict semantics, sometimes more strict than needed, which degrades performance. Internally, the library and run-time system can use any (additional) provided information to gear the communication pattern towards the available hardware.

Communication pattern The communication pattern refers to the observable inter-process communication, which is the sequence of collective or point-to-point operations invoked by the program and the communication partners. In most scientific codes, the communication pattern varies between the processes. Typically, processes communicate sparsely with other peers; only rarely, all processes talk to each other.

In point-to-point communication, a late start of the receive operation or the send operation cause a ready communication partner to wait – late processes are referred to as *late sender* and *late receiver*, respectively.

The arrival pattern determines the temporal order in which processes start a collective operation. A process is referred to as *early starter* if it starts a collective operation earlier than other processes (in terms of wall-clock time); in comparison, a *late starter* joins an already started collective operation. Optimally, all processes invoke the collective operation concurrently; this *balanced start* yields the best performance[FPY07].

When a program tries to send data from multiple processes to one process at a given time, then the single receiver imposes a bottleneck. Therefore, implementations of collective operations try to reduce the *network congestion* and balance network transfer among the participants.

Algorithms Internally, the communication must be realized by implementing some inter-process communication protocol. Several algorithms can be thought of for point-to-point communication: By using a rendezvous protocol, a sender waits until a receiver is ready to start transmission, or a sender can start transfer immediately without knowing the actual state of the receiver.

Collective operations are implementable in several communication patterns; the algorithm that realizes the semantics of the collective operation has a major impact on the observable communication pattern. An efficient algorithm follows the semantics, yet utilizes provided hardware resources at any time and minimizes redundant operations. Further information about the optimization of MPI is provided in Section 2.3.4.

Memory and CPU Performance of network data transfer is limited by memory performance because data from the network card must be copied from, or to, main memory. Without *direct memory access* (DMA) capability of the platform the CPU must also move data explicitly, which implies a performance limitation for fast networks. Also, the CPU might be necessary to initiate or control communication, or it does some bookkeeping such as computing checksums for the network packets. Therefore, many network adapters accelerate communication protocols by supporting DMA and by providing so called *offload engines*, which perform most of the required computation inside the adapter.

Data must be copied at least between network device and memory. However, often data cannot be transferred directly between the network device and the buffer that is specified in the application. Therefore, multiple copies are needed and the available memory throughput is degraded further. On the one hand, this might be necessary because the user space does not have direct access to the network device. On the other hand, protocols such as TCP embed further control data, the Internet Protocol, for example, requires post-processing to compute and to verify checksums which are embedded in the data packets. *Zero-copy* is a feature of the platform which enables direct data transfer between network device and user-space [KT95]. Without zero-copy capability of the platform, the OS must copy data between an OS internal buffer, in which the data is exchanged with the network device, and the buffer of the process. In the best case, data is copied from the memory of a process to the memory region of the remote process via RDMA.

Alignment of provided buffers Hardware technology and te platform can offer the capability to transfer contiguous data via RDMA which enables zero-copy. Typically, transfer of multiple non-contiguous regions in memory either requires to pack them in additional memory buffers for RDMA, or to send them without RDMA support. On some systems, copying data in memory from (or into) a fragmented buffer takes a considerable amount of time. Also, buffers not aligned to cache lines might waste memory bandwidth.

Software to hardware mapping Processes and threads of a parallel application must be assigned to available CPUs and nodes. Typically, the inter-process communication between some pairs of processes is more efficient than between others. The reason for this may be the network topology, or that intra-node communication is faster than communication via the network interface. In most cases in HPC, the user will start more processes than fitting onto one node, hence multiple nodes must be used. The communication path between two communication partners defines the performance of the inter-process communication.

Assigning the processes to the processors in an appropriate way yields high potential for optimizing applications with well-known communication and I/O behavior. Two processors which communicate often should exchange data in an efficient way. Especially, for tightly coupled applications[22], the inter-process communication should be as efficient as possible.

Complexity and potential of the mapping in the context of MPI is discussed on Page 51.

2.2.5. I/O Performance

There are several aspects involved in delivering high I/O performance to parallel applications. While given hardware characteristics are discussed on Page 37, this section focuses on the methods applicable to manipulate workloads to improve achievable performance.

Since many file systems such as PVFS2 and Lustre use local file systems like Ext4 to store their data, they are influenced by the mapping of logical blocks to physical blocks on the block devices. To ensure persistency and consistency, file systems write additional data to the block device. Those add overhead when modifying data or metadata. Further, file systems are implemented in the operating system which deploys strategies to improve performance such as scheduling, caching and aggregation.

Therefore, the observable I/O performance depends on more than the capabilities of the raw block device. In this thesis, the term *I/O subsystem* is used to refer to all the hardware components and the software layers

[22]Tightly coupled applications are programs which processes must communicate frequently with each other due to data dependency. In contrast, an *embarrassingly parallel* problem can be solved by processes which compute their result independent from other processes.

involved in providing node-local persistent storage[23]. Typically an I/O subsystem consists of the operating system which provides a file system on top of the block storage. The file system maps file system object to available blocks by using the block-level interface; it also schedules the raw block I/O. A parallel file system adds a layer on top of many independent I/O subsystems. The I/O path inside the operating system and the (bus) systems involved to transport data between memory and block storage are also addressed by the term I/O subsystem.

In the following, general considerations about the influences to I/O performance are discussed. Further information specific to local file systems is given in Section 3.6.1.

Access patterns The access pattern describes how spatial access is performed over time. With an access pattern, the I/O of a single client process can be described, but also the actual observable patterns on the I/O servers, or on a single block device. The pattern on the I/O servers is caused by all clients and defines the performance of the I/O subsystems.

An access pattern can be characterized by:

- **Access granularity:** Depending on the context, this is the amount of data accessed per I/O call or request. It is desirable to access a large amount of data per request, to reduce the influence of latency induced by I/O subsystems and network. In the best case, a single large contiguous I/O region is accessed per call.

  To access multiple non-contiguous regions of a file, either one request is issued per region, or a so-called non-contiguous I/O request bundles them into a single request. Non-contiguous I/O enables optimizations on the I/O subsystem such as scheduling, because the server has more pending operations to choose from. Also, the setup time for the initial link establishment and preparations of the I/O is reduced. Access to non-contiguous data in memory and on disk requires additional computation and may lead to in-memory aggregation of the contiguous regions into a temporal buffer. The extra computation time increases with the number of fragments while the costs of in-memory copies depend on their sizes.

- **Randomness:** Defines how close the accessed bytes are in the file (or on the block device). When multiple operations are issued in a sequence, then the locality of the accessed data matters. Disks achieve best performance with contiguous accesses where physical locations on the persistent storage are close together. For an SSD, random access is not so harmful, well designed controllers handle sequential and random workloads with the same efficiency; still access granularity should be in the order of the erasure-block size.

- **Concurrency:** Describes the number of concurrently issued I/O requests. The hardware resources must be multiplexed among all requests. Even if multiple sequential accesses to files are issued, an I/O scheduler might interleave these requests. Thus, from the perspective of the I/O subsystem it might look like a non-sequential access pattern.

- **Load balance:** This is the distribution of the workload among the servers in a parallel file system (or multiple block devices in a RAID).

  While distribution of a file among the servers is assigned by the parallel file system, the actual usage of servers depends on the access pattern. Depending on both the data distribution and the client usage, the requests to parallel or distributed file systems may access data only from a subset of the servers. This may potentially lead to a different utilization of the servers – a load imbalance. It is expected that the aggregated throughput of these requests is reduced. Intuitively, the highest throughput is achieved if data is accessed on all servers and if the amount of data accessed on each server depends on the server's current capabilities.

[23]In literature, the term I/O subsystem is often used in a relaxed manner; for example, to refer to a software layer which performs I/O. In this thesis, the term is used to highlight that performance and observable behavior is determined by more than the block device.

- **Access type:** All layers can either perform read or write accesses. There are other types of access for file servers, such as *flushes*, which cause cached data to be written to the block devices, or metadata operations. The I/O path might distinguish the different types of access; depending on incorporated optimizations and observed access patterns, other optimizations might be involved.
- **Predictability:** This measure indicates if data access follows a more or less regular pattern over time. The easier a pattern, the better predictable it is. Checkpointing[24], for example, stores a big amount of data in a well defined period and thus it is easily predictable. Predictable access patterns might be recognized and automatically enable comprehensive I/O optimizations in the operating system or file system like read-ahead, for example.

I/O strategy In general, the mechanisms introduced in this paragraph are orthogonal to the hardware and the architecture of the parallel file system. On the client-side, for instance, requests could already be tuned to improve the access pattern which will be observed on the servers. Similar to optimizations found in communication, these strategies could be applied on any layer involved in I/O.

A set of potential strategies are:

- **Caching algorithm:** Physical I/O is much slower than access to main memory. Therefore, it is favorable to hide the slow device performance by buffering I/O in main memory. Caching of requests is also a necessary prerequisite for many further optimizations like reordering or aggregation of requests.

  Write bursts fitting into the memory can be delayed – an early acknowledge to the writer permits it to proceed with processing. In the presence of bursts, this *write-behind* strategy can saturate the slower media constantly; while the writer continues with its computation, the data is slowly persisted on the storage. Unaligned writes might benefit from data already cached by allowing to combine multiple small accesses into a full block; this avoids the need to read the old block from the block device.

  Read-ahead is a strategy in which data is read in advance into a faster cache. Later reads access the cached data and avoid loading data from the slow I/O subsystem. However, in case the data is not needed in the near future it was fetched from the I/O subsystem unnecessarily. Thus, a balance of the read-ahead size is important and depends on the predictability of the access patterns and costs of the additional operations.

  Distributed systems could build a coherent *distributed cache* in which cached data not necessarily belongs to the local disk subsystem. This strategy increases the total available size of the cache at the expense of inter-process network communication. As network is often faster than the I/O subsystem, this boosts performance of applications whose working sets fit into the cache. On the client side, these strategies can be applied to avoid creation of requests to servers at all – all data is simply held locally. One difficulty with caching strategies is that it is necessary to update cached data to ensure a coherent view – this is often defined by the consistency semantics of the system that provides the caching.
- **Replication:** This strategy creates distributed copies of data, which then can be used to satisfy read requests. This effectively balances read-mostly workloads among the available components by reading the data from replicates located on idle servers. The number of copies can be varied depending on the level of load-imbalance and regarding the costs of the replicas' distribution. Replication is also a method to provide high availability: if one server crashes, data of the file is still accessible on a replica.

  The replication protocol defines the approach by which data is replicated: In a parallel file system typically either the clients or the servers are responsible for mirroring data. Also, data written could

[24]A checkpoint contains all the data that is required to continue computation later. This is especially useful to restart long-running applications that crashed due to hardware errors.

be replicated synchronously – during the write call, or asynchronously in the background. Since more data must be communicated than necessary, the chosen algorithm has a big impact on performance.

- **High availability (HA) support:** Mechanisms to increase availability of the computing environment imply additional expenses for hardware, and they reduce performance of the system by requiring it to write more data than necessary.

  Redundant components like hot spares and redundancy by storing data on multiple devices with error correction codes tolerate a number of failures of components or physical data. In case of a hardware failure, reconstruction of the original data with the redundant information implies a performance impact, though. Complete data replication multiplies the amount of stored data but has the advantage to provide load-balancing for files that are mostly read as well. HA could be provided just on a per server basis, or could be built into the parallel file system itself by spanning multiple servers to tolerate complete server faults.

  In large data sets the so-called *silent data corruption* [Mic09] becomes important[25]. There are methods to identify data corruption, for example, by storing checksums of the data. Another approach for fault tolerance is *end-to-end data integrity*. It validates that data accessed by the client is the same as stored on the I/O subsystem. However, transmitting the checksums and verification may reduce performance slightly.

- **I/O forwarding:** An I/O server can handle only a limited number of concurrent clients because each connection requires resources on the network interface and in memory. I/O forwarding [ACI+09] is a technique in which clients do not communicate with servers directly[26]. Instead they communicate with intermediate I/O nodes – the so-called I/O forwarders. Those nodes channel the I/O of all responsible clients to the file system and thus, reduce the burden on the file system.

  I/O forwarding could reduce performance because data must be communicated to another node, and might be a bottleneck in a system. However, this depends on the network topology and placement: In indirect network topologies an I/O forwarder can be placed on a node that is directly on the path between client and server. Thus, in such a configuration the I/O forwarding does not imply additional network communication. This is true, for example, for the network topology of a BlueGene system.

  However, optimization techniques, such as caching, aggregation or scheduling, can be implemented on the I/O forwarders that improve performance of I/O servers.

- **Aggregation:** This technique combines a set of smaller requests to a larger request containing all the data and operations. Aggregation can be performed on various layers: on the client side, on intermediate I/O forwarders, on the server or on the I/O subsystem.

  This technique is very useful, when independent operations are interleaved or overlapping, multiple operations can be combined into a bigger sequential access. It is also possible to integrate independent requests into one non-contiguous request, even across disjoint applications, if they access the same file. However, additional buffer space is required to merge the data into the new request. Also, the number of requests to decode is lowered – reducing computation time to process operations.

- **Scheduling:** A scheduler queues pending work and dispatches it according to a strategy. Requests can be deferred to avoid flooding of layers which are unable to process all the pending requests concurrently. In the meantime, the outstanding requests can be reordered or aggregated in order to optimize the access pattern.

  Scheduling algorithms can be applied on different abstraction layers. For example, with NCQ the block device incorporates reordering schemes for block accesses as discussed in the hardware section.

[25] Data corruption means read data does not match the written data any more – this can happen due to bit flips or hardware errors. Silent refers to the fact that this data corruption cannot be detected by the system. Thus, the user is not aware that data has been corrupted.

[26] Note that sometimes in literature the intermediate nodes are referred to as I/O aggregators, too.

Also, scheduling strategies could be applied not only on block-level but also on file-level.

Parallel file system Performance of a parallel file system highly depends on its design as it provides the frame for the deployed optimization strategies. Several aspects like consistency semantics also apply to higher level interfaces like domain specific I/O libraries.

The following important factors tend to vary over the available parallel file systems:

- **Design:** Software architecture of a parallel file system defines the I/O paths within the file system and the distribution of data among the available servers. Naturally, the parallel file system should be able to saturate all participating components (network, servers and I/O subsystem); even with a low number of concurrent clients. Therefore, it is necessary to utilize all components to the best extent. This requires to perform operations on disk and network at the same time. An efficient path for performing I/O on the client side to the server and back is crucial. The higher the abstraction layer of the file system the more intelligent optimizations could be integrated, and the more background information a user can provide the better potential for optimization exists.
- **Implementation:** Usually, computation is not the bottleneck of the file system as CPU's are much faster than the I/O subsystem. However, it is important to be able to saturate both network and disk subsystem; to schedule and optimize operations, data structures are required. Efficient basic data structures like hashes or trees, which scale well with the number of concurrent requests, are necessary to reduce the overhead of the operations. Regarding compute performance, in principle, all the aspects mentioned in Section 2.2.3 apply.
- **Resource consumption:** The parallel file system itself needs some of the available hardware resources to operate. On the client side, these resources are not available for the application and considered overhead. The amount of memory consumed for internal data structures and allocated buffer space reduces the available I/O (and metadata) cache. The number and complexity of instructions which have to be processed in order to prepare or serve requests increase utilization of a CPU – thus, a perfect overlapping of computation and I/O may not be possible. Examples for communication overhead are additional control information transferred within the request and response messages, or server-to-server communication needed for replication.
- **Communication:** Performance factors of the communication between client and server are the steps necessary to establish a connection, to provide data encoding into a common format (in order to support heterogeneous environments), data transfer and the communication protocol. Additional control information must be transferred to the server to initiate the request. Protocols that permit bursts of commands to be transferred in a single request reduce the processing overhead. Support of parallel streams between client and server can improve performance. Several aspects like non-contiguous communication are also discussed in Section 2.1.4.
- **Data distribution:** Distribution of data among the available servers defines the maximum concurrency and highly influences the potential usage of the I/O subsystems. Large contiguous data access is preferable, hence data must be stored in large chunks on a server, yet the number of servers involved should be high to provide a good aggregated performance by utilizing all of them concurrently. Adaptive striping depending on the file size meets both requirements. In most cases automatic re-striping is not supported by the file systems, instead the user (and system administrator) can define the way data is distributed on file creation to match the expected access patterns.
- **Metadata handling:** Regular file system operations involve metadata about the objects, as it must be acquired before I/O can be performed. Metadata performance is crucial to many workloads. Metadata is rather small, therefore it has to be organized in a manner that allows bulk transfers between the servers and persistent storage. Network and I/O subsystem should be utilized, which requires a large number of metadata operations to be initiated per request.

  On the server side, an efficient metadata management is necessary. For example, logical files could be pre-allocated to already contain all information; thus avoiding additional communication to data servers during file creation. PVFS supports some bulk operations: when listing a directory, for example, a large number of directory entries are transferred in one response. Also, PVFS incorporates read-only client side caches for metadata.

- **Consistency semantics:** The semantics of a file system define how the API calls are translated to file system manipulations and accesses. Of special interest is how and when a server (and a client) realizes concurrent modifications made to file system objects. Relaxed guarantees open the potential for additional optimizations. For instance, when a client does not need to realize all modifications made in the past, it could save communication that would refresh the state of cached objects. Also, operations could be deferred on a client to bundle them with future requests, then other clients would not realize modifications until they were communicated to the servers.

  Assume a client which tries to create a number of files; this client could request file creation of every file individually, or it could avoid some of the communications by assuming the state of the directory is still valid. With strong consistency semantics, locking is mandatory to perform bulk operations because with a lock modifications of the file system status could be prevented. For example, Lustre allows to issue a set of metadata operations at once by locking metadata, and Lustre pre-creates files, too.

  With relaxed consistency semantics, modifications will be lost, if client or server crash prior to execution and thus persistence is not ensured. However, in this case, locking mechanisms might become costly, too; the crash has to be detected and the lock must be released to continue operation.

  Consistency semantics are defined by the application program interface. POSIX for example has very restrictive semantics since I/O operations have to be applied in the same order as issued, even if they are non-overlapping. Thus, communication with the I/O subsystem must be serialized to guarantee proper handling. This is a major drawback of the POSIX interface. Most programs could be adapted to loose semantics to exploit parallelism. PVFS does not provide guarantees for I/O data in case of concurrent operations, data could be a mixture of data blocks of different requests. However, with PVFS all metadata operations are ordered in a specific way to guarantee metadata consistency of objects in the accessible namespace[27].

  Further information on concurrent access is given when MPI-IO semantics are discussed (see Page 49).

- **Locking strategy:** Locking as a mechanism can be used to prevent concurrent access to one (or multiple) file system objects. Depending on the required file system semantics, measures must be taken to ensure that clients access the current state of the file system. While GPFS and Lustre provide a locking protocol, PVFS does not offer locking; hence, when using PVFS the application must avoid concurrent access to file regions. In presence of a locking mechanism, the granularity of the lock defines the potential concurrency with related data and metadata – a lock could be valid for a file region, a logical file itself or whole directories. Different lock types could be available – depending on the type of the lock, another access type could be permitted. A particular lock, for instance, could prevent modifications to a file but allow concurrent reads, while another one may restrict the type of access possible. Locking algorithms can become expensive in a distributed environment, since they require a consistent view.

2.3. Message Passing Interface

The Message Passing Interface [Mes09] is a standard to enable inter-process communication. With MPI the user explicitly encodes the sending and receiving of data in the source code as function calls to MPI.

[27]There could be broken objects not yet linked to the namespace and inaccessible, though. These broken objects can be identified and cleaned by performing a file system check.

In this section the focus is placed on semantical aspects and communication performance implications imposed by the interface; features are only listed if they are relevant to this thesis.

The standard offers a rich set of *point-to-point* and *collective* functions to address many use-cases while keeping communication efficiency in mind. In point-to-point communication two processes communicate, one sends a message and the other receives it. With collective operations an arbitrary number of processes can participate to exchange data. In general, collective operations can be grouped into three classes depending on the number of processes that provide the initial data and the number of receiving processes: all-to-one (for example *MPI_Gather()* and *MPI_Reduce()*), one-to-all (e.g., *MPI_Bcast()*), all-to-all (e.g., *MPI_Allreduce()*).

With MPI-2 a set of functions to manipulate files are standardized. Directory operations are provided to a limited extent, only creation and deletion of files is supported. Basically, MPI-2 stretches communication concepts of MPI to I/O. I/O functions rely on communication between MPI processes with the I/O servers; in the context of I/O the MPI processes performing an I/O call are referred to as *clients*. MPI defines individual and collective I/O operations and permits non-contiguous access to file data.

First, Section 2.3.1 describes the rules how transferred messages are matched by a receiver. Then some MPI functions are introduced in Section 2.3.2. An excerpt of semantical aspects is provided in Section 2.3.3. Based on the semantical aspects, available optimization potential is elaborately discussed in Section 2.3.4. In Section 2.3.5 related work is presented to gear the MPI run-time towards system and application.

2.3.1. Matching Sends to Receives

A program can perform multiple collective calls and point-to-point operations at the same time. To ensure proper operation, data must be received by the intended receiver and not by an arbitrary process. Therefore, several rules are defined by MPI that match send messages to receives – the rules were chosen to permit efficient implementations but still offer a comfortable programming interface. One basic rule, for instance, is that messages are usually received in the order they are sent, i.e., whenever two messages match all rules of the receiver, then the one received first is given preference over the others.

Point-to-point communication Messages include metadata in an envelope which contains the *source*, *destination*, *tag* and *communicator*. A communicator describes a group of processes that participate and a context. The source and the destination are the *rank* of the respective processes in the specified communicator, i.e., their numeric ID within it. The tag is a numeric identifier that distinguishes messages from a sender, typically a tag has a user-specific semantic. For example, a message with Tag 1 could mean that this message contains input data, while Tag 2 indicates that computation is completed.

According to the parameters supplied to the receive operation, incoming message envelopes must match some or all of the information to be received. Wildcards can be used to receive a message from any rank of the communicator, or one with an arbitrary tag. However, the *communicator* must always match as specified by the MPI standard:

> *"A communicator specifies the communication context for a communication operation. Each communication context provides a separate "communication universe:" messages are always received within the context they were sent, and messages sent in different contexts do not interfere."* [Mes09]

Further, in a message envelope information about the transferred data types is contained. This information is not used to match sends to receives, but it helps the library to check the correct data transmission: When a message matches the conditions, received data is copied from the message into the output buffer on the receiver side. During this process the elementary datatypes specified on the sender side and receiver side are compared to ensure consistency. When they do not match the received data is invalid. Also, the number of elements to receive must match the number of elements transferred. While it is not possible to receive more elements than specified, because usually the provided buffer does not suffice, it is possible to receive less data than expected. For example, a sender can send 10 integers while the receiver specifies a buffer for 50 integers; the actual number of elements received can be checked in the program after the receive completed.

Derived datatypes in the MPI standard allow developers to directly transmit non-contiguous regions in memory – that means multiple non-overlapping memory areas can be sent with one MPI call. Non-contiguous communication avoids the need to pack data explicitly in a contiguous buffer. MPI defines a large number of datatypes to ease definition of vectors, structures or expansions of primitive datatypes. By referring to already defined datatypes the user can compose arbitrarily nested datatypes. In the communication process the datatypes are converted to the receiver's machine architecture if necessary, and copied from/to the specified memory regions. Datatypes on sender and receiver side can be different, although the elementary datatypes below must match (it does not make sense to transfer an integer and receive a double value).

MPI-2 applies this concept to enable non-contiguous I/O (see Section 2.3.3 for details on how to use it). See section *Type Matching Rules* in [Mes09] for more details.

Collective calls MPI forbids interference of messages from point-to-point operations with messages sent by collective operations. To quote the standard:

> *"Collective communication calls may use the same communicators as point-to-point communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication."* [Mes09]

Interference between collectives shall not happen as well, as it makes no sense for one collective operation to receive a message from another.

Different datatypes on sender and receiver are permitted, but the amount of data must match, according to the standard:

> *"The type-matching conditions for the collective operations are more strict than the corresponding conditions between sender and receiver in point-to-point. Namely, for collective operations, the amount of data sent must exactly match the amount of data specified by the receiver. Different type maps (the layout in memory [...]) between sender and receiver are still allowed."* [Mes09]

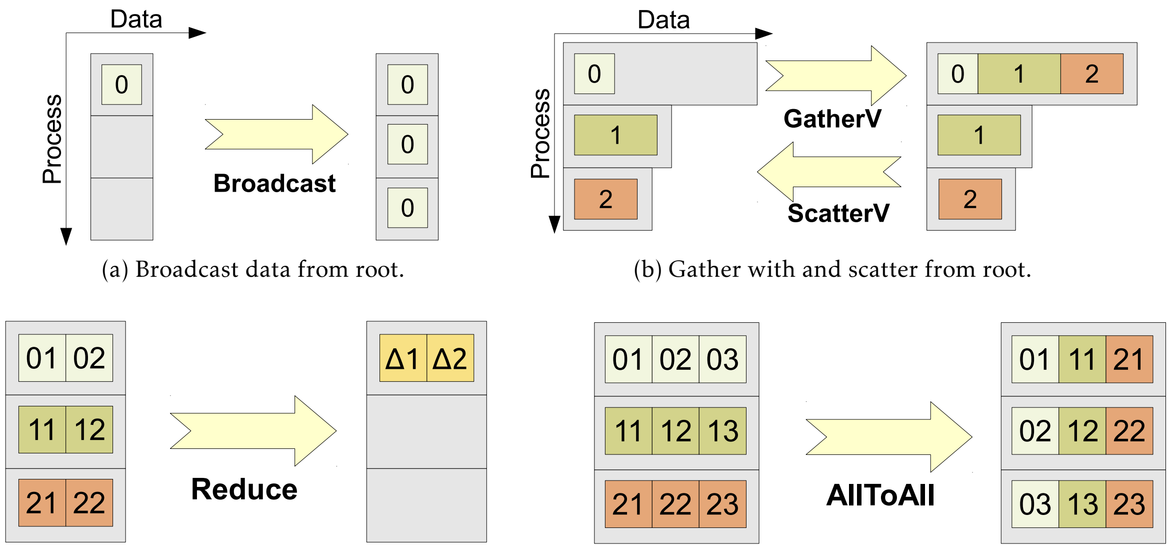
Consequently, MPI collective calls receive a message from another collective call iff the communicator is identical and the collective call matches. When one process of a communicator starts one collective call and another process starts a different call, then a deadlock occurs. MPI-3 will allow asynchronous collective calls; it is likely that multiple non-blocking starts of the same collective operation will be serialized to avoid false matching of messages.

2.3.2. Exemplary Collective Operations and Semantics

To foster discussion of performance aspects, the parameters and the invocation semantics for at least one representative of each collective pattern is listed. Data exchanged between processes is visualized in Figure 2.8. Gray boxes symbolize buffer space for input and output buffers – for the sake of simplicity it is not distinguished between input and output buffer. In the figure it is assumed, that *one-to-all* or *all-to-one* operations are performed with Rank 0 as root/target. The definitions and semantics are taken from [Mes09].

Synchronization behavior Once a collective function returns, the required operation for the local process has been performed in its output buffer; this implies that collective functions are blocking[28]. Collective operations do not imply synchronization. Synchronization between processes depends on the implementation of the specified semantics; for example, with a broadcast operation all processes must wait for

[28]With MPI-3 non-blocking collective calls will be introduced.



(a) Broadcast data from root.

(b) Gather with and scatter from root.

(c) Reduce data of multiple elements in root. Δi is computed from p_i (for all processes p in the communicator).

(d) All ranks exchange data.

Figure 2.8.: Data transfer scheme of a few collective operations.

the root process to be ready. But theoretically other processes could complete even if some processes did not call the collective operation, yet.

To quote a lengthy, but informative general introduction to collective calls:

> *"Collective routine calls can (but are not required to) return as soon as their participation in the collective communication is complete. The completion of a call indicates that the caller is now free to modify locations in the communication buffer. It does not indicate that other processes in the group have completed or even started the operation (unless otherwise implied by the description of the operation). Thus, a collective communication call may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier function.*
> [...]
> ***Advice to users.***
> *It is dangerous to rely on synchronization side-effects of the collective operations for program correctness. For example, even though a particular implementation may provide a broadcast routine with a side-effect of synchronization, the standard does not require this, and a program that relies on this will not be portable.*
>
> *On the other hand, a correct, portable program must tolerate a synchronizing side-effect of a collective call. Thus, though it cannot be relied on the synchronization effect of a collective call, one must formulate the program to deal with it."* [Mes09]

Several functions can replace existing data in the input buffer directly with new data[29], that means input data is replaced with the result of the collective call. This enables the MPI library to conserve about half of the memory. Internally, the implementation might provide additional data buffers. However, buffer capacity should be minimized as the description of *All-to-All Scatter/Gather* in [Mes09] states: *"Users may opt to use the "in place" option in order to conserve memory. Quality MPI implementations should thus, strive to minimize system buffering."*

[29]To do so the user has to set the buffer to `MPI_IN_PLACE`.

Broadcast The `MPI_Bcast()` operation transfers data from a *root* process to all other processes in the group (Figure 2.8a), upon return of a callee the data is available in its local output buffer. The *signature*[30] of the call is:

```
MPI_BCAST( buffer, count, datatype, root, comm )
  INOUT   buffer     starting address of the buffer (choice)
  IN      count      number of entries in buffer (non-negative integer)
  IN      datatype   data type of buffer (handle)
  IN      root       rank of broadcast root (integer)
  IN      comm       communicator (handle)
```

The datatype can be an elementary datatype or a derived datatype.

Gather A gather operation transfers data of a fixed size from each process's buffer to the root buffer (Figure 2.8b). Data of the processes is stored subsequently in the buffer of the root, that is data of Process i is stored right in front of data from Process $i+1$. `MPI_GatherV()` customizes the amount of data sent by each process to the root.

Signature of the call:

```
MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype,
            root, comm)
IN  sendbuf     starting address of send buffer (choice)
IN  sendcount   number of elements in send buffer (non-negative integer)
IN  sendtype    data type of send buffer elements (handle)
OUT recvbuf     address of receive buffer (choice, significant only at root)
IN  recvcounts  non-negative integer array (of length group size) containing the
                number of elements that are received from each process
                (significant only at root)
IN  displs      integer array (of length group size). Entry i specifies the
                displacement relative to recvbuf at which to place the incoming
                data from process i (significant only at root)
IN  recvtype    data type of recv buffer elements (significant only at root)
                (handle)
IN  root        rank of receiving process (integer)
IN  comm        communicator (handle)
```

Gather can be thought of to be the inverse function to an scatter operation, which distributes subsequent blocks of data from the *root* among all processes. Broadcast is quite similar to scatter. Consider that Rank 0 transmits individual data to each process, which is indicated on the right side of Figure 2.8b. After the operation completed each process has its part of the data locally (left side of the figure).

Another explanation to the execution of collective operations is to specify the operation in terms of point-to-point operations – the definition for `MPI_GatherV()` is:

> *"[...] the outcome is as if each process, including the root process, sends a message to the root,*
> ***MPI_Send**(sendbuf, sendcount, sendtype, root, ...),*
> *and the root executes n receives,*
> ***MPI_Recv**(recvbuf* $+$ *displs*$[j] \cdot$ *extend(recvtype), recvcounts[j], recvtype, j, ...)*
> *The data received from process j is placed into recvbuf of the root process beginning at offset displs[j] elements (in terms of the recvtype)."* [Mes09]

By setting the *displs* vector accordingly it is possible to change the order in which the data is stored, i.e., it is possible to store data from Process 2 at the beginning of the output buffer. However, it is erroneous to receive data from multiple processes in the same memory location.

[30]In the signature the description of the input and output parameters is given in a language independent notation according to the standard. Parameters are prefixed with the type (input, output or both).

Reduce Aim of a global reduction operation is to perform an associative operation like sum or minimum, across data provided by the processors of a group. A vector of *count* elements can be supplied in each process, then a reduction is performed for each element of the vector individually. In Figure 2.8c, each process has two elements, after the reduce the root process has the two independent results (Δ1 and Δ2) computed with the associative operation.

Signature of the call:

```
MPI_REDUCE( sendbuf, recvbuf, count, datatype, op, root, comm )
  IN   sendbuf    address of send buffer (choice)
  OUT  recvbuf    address of receive buffer (choice, significant only at root)
  IN   count      number of elements in send buffer (non-negative integer)
  IN   datatype   data type of elements of send buffer (handle)
  IN   op         reduce operation (handle)
  IN   root       rank of root process (integer)
  IN   comm       communicator (handle)
```

The user can define further operations, which might also operate on derived datatypes – pre-defined operations work only on elementary datatypes. To quote [Mes09] regarding the semantics of the operation:

> *"The operation op is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.*
> ***Advice to implementors.***
>
> *It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processors."*

The advice to the implementors impacts the performance, as numerically unstable floating point operations must be executed in the same order. Therefore, independent of the process mapping, the same operations must be executed, thus locality of the processes cannot be exploited in all cases. `MPI_Allreduce()` is a variant of reduce in which the result is stored on all processors – not only on the root process.

All-To-All With this operation each process transfers individual data to each other process; the amount of data exchanged between all processes is the same, see Figure 2.8d.

Signature of the call:

```
MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
  IN   sendbuf     starting address of send buffer (choice)
  IN   sendcount   number of elements sent to each process (non-negative integer)
  IN   sendtype    data type of send buffer elements (handle)
  OUT  recvbuf     address of receive buffer (choice)
  IN   recvcount   number of elements received from any process (non-negative integer
    →)
  IN   recvtype    data type of receive buffer elements (handle)
  IN   comm        communicator (handle)
```

In this all-to-all function the user may specify that the data in one buffer is used as input and output, thus it can be replaced with new data. Effectively, this halves memory consumption.

2.3.3. Excerpt of MPI-IO Functions and Semantics

The I/O interface of MPI can be thought of as an extension of POSIX I/O for parallel programs. While the basic idea of file manipulation is connected to POSIX, the interface and semantics enable coding of portable and efficient parallel applications.

There are several MPI functions that deal with input and output of data. The semantics differ in the way the file pointer is used – either all processes share a common file pointer, or each process has its own file pointer. An orthogonal aspect is whether the call is blocking or non-blocking (and shall be executed concurrently with further program activity). Further, each process can perform I/O independently, or multiple processes can participate during the I/O (so-called *collective I/O*). As the functions are quite similar, only a subset of the available functions is introduced to enable later performance discussion: opening of a file, to enable non-contiguous file access by defining a file view, the non-blocking read operation and the collective write operation.

At last, the access semantics for multiple processes accessing the same file are discussed.

Opening of files With the collective function `MPI_File_open()` all processes of the communicator open a file. Once the file is opened individual (independent) or collective I/O can be performed by referencing the returned file handle. It is expected that all processes of the communicator participate in subsequent collective I/O operations with that file handle. The implementation must ensure that communication routines do not interfere with I/O operations. Therefore, no message from communication routines shall be received by an I/O operation and vice versa (see also Section2.3.1).

Signature of the call:

```
MPI_FILE_OPEN(comm, filename, amode, info, fh)
  IN comm       communicator (handle)
  IN filename   name of file to open (string)
  IN amode      file access mode (integer)
  IN info       info object (handle)
  OUT fh        new file handle (handle)
```

The specified filename on all processes must reference the same logical file – for example it is invalid for multiple processes to collectively open files on a temporary directory that cannot be accessed from all processes. The implementation defines how the filename is interpreted, ROMIO uses prefixes to distinguish various file systems. To access files in the PVFS namespace, for instance, the prefix "PVFS2://" is specified.

The file access mode is similar to the POSIX open call, with `MPI_MODE_CREATE` the call creates a new file and thus it alters the namespace of the file system if the file does not yet exist. Besides a delete function, there is no other way to alter metadata and the namespace with MPI functions.

Users can supply optimization parameters, which are in the form of key/value pairs, to the info object. Those parameters are referred to as *hints*. The `access_style` is an example of a reserved hint, the user can specify the temporal and spatial access patterns in which file access is performed until the file is closed – possible values are `read_once`, `read_mostly`, `write_once`, `write_mostly`, `sequential`, `reverse_sequential`, and `random`. Multiple access patterns can be defined, e.g., to announce that a file is read and written exactly once. Note that file hints can be changed with the `MPI_File_set_info()` function at run-time. Further information about hints is given in Section 2.3.5.

Defining a file view A file view defines how and which areas of the file are accessible by a process. This supports a better interpretation to the raw bytes of the file – by setting the data types contained, the implementation could perform file type conversion automatically. Further, non-contiguous file accesses are

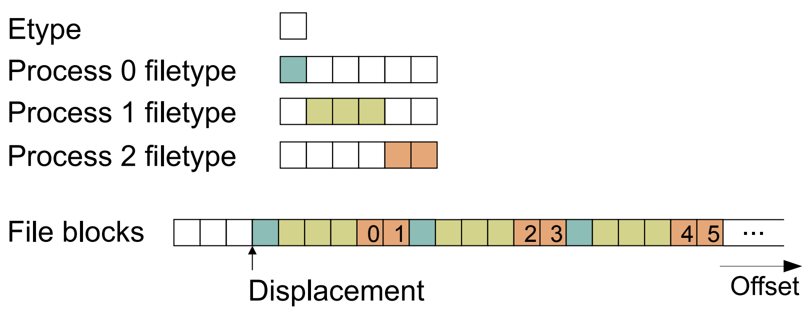


Figure 2.9.: Example partitioning of a file by using a file view. The accessible area for each process is colored, the offsets for accessible etypes are printed for Process 2.

implemented with the help of file views. First, the file is partitioned into regions which can be accessed, then non-accessible regions are skipped by subsequent I/O calls.

This function is collective – all processes which opened the file set their own view. While the file is open the file view can be altered by invoking `MPI_File_set_view()` again. It is invalid to change the view of a file while non-blocking I/O is performed.

Signature of the call:

```
MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)
  INOUT fh       file handle (handle)
  IN disp        displacement (integer)
  IN etype       elementary datatype (handle)
  IN filetype    filetype (handle)
  IN datarep     data representation (string)
  IN info        info object (handle)
```

The *elementary datatype* defines the unit of positioning and data access. From the program's point of view access to parts of an *etype* does not make sense, e.g., to read half of an integer. Only full etypes can be read/written and data positioning is relative to this datatype, i.e., the first etype is accessed, then the second etype and so on. For I/O calls the datatypes are restricted to use non-negative and monotonically increasing displacements.

The *filetype* sets how etypes form the whole file. Unimportant file regions are skipped, those non-accessible areas are usually referred to as *holes*. By using datatypes with holes non-contiguous file access is possible.

In *datarep* the interpretation of data types is set. Either the system's native representation is used – that means no conversion will be performed, or datatypes can be converted to a fixed representation. There are two alternatives that guarantees portability of the files created: An MPI implementation specific representation, which is portable across architectures as long as the same MPI implementation is used. And a standardized data type representation that is understood by all MPI implementations, but potentially slower.

An illustrated example of a collective file view for three processes is given in Figure 2.9. In this example the etype could be any portable datatype starting from a single integer to a complex record. Process 0 accesses the first etype, Process 1 the next three etypes and Process 2 the two following etypes. The datatypes are repeated infinitely in the file. In the figure the offsets for Process 2 are printed into the data blocks.

Independent non-blocking read With independent I/O, each process performs I/O without knowledge of the other processes' state. The operations are marked with an "*I*" in their name, read and write calls are treated similarly.

In the following example, the signature of the read call is provided:

```
MPI_FILE_IREAD(fh, buf, count, datatype, request)
  INOUT fh        file handle (handle)
  OUT   buf       initial address of buffer (choice)
  IN    count     number of elements in buffer (integer)
  IN    datatype  datatype of each buffer element (handle)
  OUT   request   request object (handle)
```

Accessed data is read from/written into the provided buffer with the specified memory datatype. If a file view has been applied like in Figure 2.9, then a process accesses only its visible data, e.g., if Process 2 reads 6 elements, then all 6 etypes of the example file are read in one call – thus, non-contiguous file access is performed. A potential data conversion is performed on the fly, too.

An independent non-blocking I/O operation just extends the signature of the blocking call by an opaque request object[31]. This opaque request object can be used to check the completion of the operation by MPI-1 functions such as `MPI_Wait()`.

Collective write In collective operations all processes which open a file might work together to access file data; this enables certain client-sided optimizations.

From the signature, a collective call is differentiated from the independent calls only by its prefix "`_all`". Depending on the implementation collective I/O operations might be synchronizing or not; the author of this thesis, however, has not seen non-synchronizing collective I/O in an MPI implementation, yet.

Signature of the call:

```
MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
  INOUT fh        file handle (handle)
  IN buf          initial address of buffer (choice)
  IN count        number of elements in buffer (integer)
  IN datatype     datatype of each buffer element (handle)
  OUT status      status object (Status)
```

Access semantics Compared to the consistency model of POSIX, the MPI standard follows relaxed I/O semantics.

In the POSIX I/O semantics, each read and write operation is an atomic operation – concurrent operations are serialized and follow the *strict consistency* model. When a call finishes all manipulations become immediately visible – the first executed call is executed first and has to terminate before the next one can start. Consequently, when a `write()` and `read()` call are performed concurrently, then one of those two operations is scheduled first. Partly written data is not returned in the read function – if the read finishes first, then the old data is read, if the write finishes first, then read will retrieve all new data. Concurrently scheduled processes obey these semantics as well, ultimately this requires to serialize all file operations.

Sequential consistency relaxes the strict model slightly: The order in which the calls are invoked (in terms of wall-clock time) can now vary from the actual execution order. However, all processors have the same picture of the execution order, and from the perspective of a processor its operations are executed in the same sequence as they have been issued. These semantics are a consequence of the distributed nature of the system; it is hard to tell which operation started first. Since all processors see the same execution order, the operations are serialized, but not necessarily the first call issued is executed first.

Three consistency levels are available in MPI: sequential consistency of one file handle, sequential consistency with atomic mode and explicit, user-imposed, consistency.

[31]Non-blocking MPI communication routines such as `MPI_Isend()` use a similar technique.

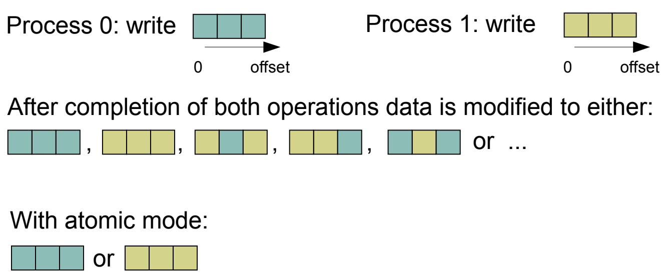


Figure 2.10.: Example of concurrent file operations, two processes write the same file regions.

In terms of MPI, the sequential consistency model implies that the behavior of accesses performed on one file handle follows any of the potential serializations of the execution, as implied by the synchronizations of the parallel program's processes: Read operations enforced to occur later in program execution than a write operation to the same file handle read the altered data, and writes to the same file regions overwrite previously written data. To illustrate this issue, consider two processes opening one shared file, one process writes some data, then both processes invoke a `MPI_Barrier()`[32] before the second process reads the data back. With the barrier, the sequential consistency enforces that the first process reads the same file region which was written before because in any execution the read always follows the write operation. Without the barrier, some old data (if the file existed before it was opened) or a mixture of already modified and old data could be returned to the other process.

This example illustrates that while each I/O operation by itself produces a well defined result, concurrent I/O to overlapping file areas result in unpredictable data – parallel accesses to overlapping file regions are not handled atomically as in POSIX.

Assume two processes opening one file, then each process writes three elements of local data. In theory the observed data in the file can be any combination of the data stored by the two processes, e.g., the first two elements from the first process and the remaining datatype from the other process – this example is illustrated in Figure 2.10. This kind of mixed data might break integrity of file records. Not only the datatypes but even the bytes in the file could be interleaved in the same fashion. In practice, file systems usually update data in the granularity of a full block. Thus, data of two processes is not mixed within one block but across block boundaries.

Atomic mode changes the behavior by serializing independent I/O to the same file handle if necessary, concurrent modifications by third-party processes or by using another file handle to access the same file are still under responsibility of the programmer (and user). If supported by the MPI library and the file system, atomic mode can be set by calling the collective function `MPI_File_set_atomicity()` with the file handle.

The sequential consistency of MPI does not specify the state of the file in regards to concurrent modifications by other programs – MPI might be unaware of these modifications. Also, the atomic mode does not protect the integrity of records in case multiple file handles point to the same logical file – it is undefined what data is accessed by independent processes and file handles. To deal with this issues the collective function `MPI_File_sync()` enforces to persist and to synchronize manipulations of one file handle with the file system. After a synchronization is performed, subsequent read operations access data updated by third-party software (or by the same program that used other file handles) in the background. Both opening and closing of a file imply this kind of synchronization.

The MPI semantics are quite comparable to the NFS semantics, as in NFS there is no cache consistency between different clients and the server. In MPI the set of processes which collectively open a file share the access with the sequential consistency model, this is exactly the behavior NFS enforces for one client

[32]A barrier is a synchronization point for all processes of the communicator. The processes wait until all of them reach an `MPI_Barrier()`, then they continue.

as each node uses local cache to increase performance.

MPI can share a single file pointer among multiple processes. Whenever a process reads or writes data, the file pointer is modified accordingly. The current semantics of a shared file pointer require a strict serialization of concurrent calls to avoid that two processes access the same file regions. However, the semantics define that the file pointer could be updated immediately after an I/O call is initiated. Thus, theoretically, the actual data transfer can happen concurrently.

2.3.4. Optimization Potential Within MPI

Several issues mentioned in Section 2.2 are related to performance of communication and I/O. The semantics of *Message Passing Interface* enable additional concepts that improve performance, some of them have been integrated on purpose into the interface. The potential that resides in using MPI is now discussed on an abstract level before details are provided on the state of the art.

In general, the usage of MPI within the application limits how well an MPI library could optimize interprocess communication – in the best case, the programmer uses the highest-level function to perform his tasks and ensures a balanced startup of all communication partners, i.e., all processes start synchronously. To reduce the complexity of the discussion, let us assume the hardware setup and the communication pattern of the application are fixed.

The performance of inter-process communication certainly depends on the hardware characteristics – the specific hardware configuration limits potential network throughput, computation power and available memory bandwidth. Therefore, it is our goal to exploit the available hardware resources to reduce application run-time.

There are two orthogonal concepts that have a large impact on performance: Either the process mapping to the existing hardware can be adjusted, or the MPI internal algorithms are adapted. Actually, many minor optimizations exist that may gear an implementation towards a system and a specific (application's) workload. For I/O calls all these options permit to optimize throughput, but even further optimizations can be applied to improve the access pattern and to decrease the burden on the I/O servers. A discussion of the state of the art that addresses the optimization potential mentioned is provided in Section 2.3.5.

Process mapping The logical MPI processes must be mapped to the available *processing elements*[33]. Without modifying MPI internally, *placing the MPI processes* in an appropriate manner on the processors yields a high potential to optimize communication and I/O behavior. However, selecting an appropriate mapping requires that the access patterns of the application are known before the program is executed.

Consider the example in Figure 1.8 (page 12). Assume, the 4 processes should be placed on two nodes with dual core processors (the example in Figure 1.6, page 9)). There are $4 \cdot 3/2$ options to map the 4 processes to the two nodes – the two processes on the first node are either $(0,1), (0,2), (0,3), (1,2), (1,3)$ or $(2,3)$. Actually, if the two nodes are equivalent, then the mapping $(0,2)$ is equal to $(1,3)$, the same with the mappings $(0,1)$ and $(2,3)$. In the example only neighboring processes communicate via point-to-point communication, but additionally a broadcast operation is performed. It can be seen that the mapping $(0,1)$ (or $(2,3)$) requires less inter-node communication than a mapping like $(0,2)$ because this way only one message is sent between process 2 and 1 over the slower network. The other two send operations can be performed within a single node. Therefore, this placement is typically faster.

Coming back to the difficulty of the process mapping, even under homogeneous hardware the number of possible permutations of the process mappings grows exponentially with the number of processes. It has been shown that the mapping problem is NP-hard. See [LB98, CCH$^+$06] for a discussion of issues related to task scheduling. Tools have been developed that apply heuristics to place processes on the system topology.

[33]The mapping from processes to hardware is introduced on Page 36.

Communication algorithm Under the assumption that the placement is fixed, i.e., either because it is optimal or pre-defined, then there is potential to adjust the MPI internal algorithm towards the given application and network topology. To cope with the complex interplay of system and application, many different algorithms must be implemented in each MPI library. However, choosing the right algorithm for a given function call remains difficult. Presumably, suboptimal performance can be observed on every system (for at least some applications).

Point-to-point operations are of simple nature, as just two processes participate; therefore, the low-level network driver has a major impact on the performance. From the algorithmic point of view an MPI implementation has only a few options if it sticks to the standard's semantics. The semantics of `MPI_Send()` permit a completion of the operation, even if the receiver did not yet get the message; this allows the implementation to buffer the message either on the side of the sender or the receiver. Many variants of the point-to-point communication exist which enforce a certain buffering or synchronization behavior.

Collective operations are of much more interest. Related work shows the impact of the arrival pattern on performance and that tuning for individual message size is important [FPY07]. From a library's point of view, optimization can be done based on the parameters provided by the programmer. Typically, this includes the *memory datatype*, the *communicator*, *target/source rank* (for all-to-one or one-to-all operations), and the actual *amount of data* shipped with the call:

- The memory datatype is relevant because contiguous datatypes can be transferred via RDMA. Datatype on sender and receiver side can be be different, requiring to pack or unpack data.
- The communicator defines the participating processes in a given communication, even if an algorithm is optimal for one defined group of processes, for another set of processes a different algorithm might be faster. The problem of choosing the appropriate algorithm for a given group of processes is linked closely to network topology.
- The target rank is relevant because the implementation must perform the last operations on the target rank, therefore, for instance a spanning tree algorithm should put its root on the target rank. Similarly, for a one-to-all operation the source rank provides some information which must be transferred to all other clients.
- Depending on the amount of data the communication time varies. A basic consideration is that latency becomes more important for small messages, while larger messages are dominated by the bandwidth between the communication partners.

Concepts to optimize I/O Due to the complexity of I/O, that involves communication as well as server activity, there exists a rich variety of opportunities to optimize access patterns and metadata operations. First, a few hardware considerations are recapitulated from section 2.2.2. Those help in assessing the mentioned optimizations. From the perspective of the servers it is important that clients access as much data as possible in one I/O request, because it keeps the pipelines on the server busy and, furthermore, it enables sophisticated server-sided optimizations. A server can access data on block-level in a granularity of full blocks – modification of a few bytes of a block requires reading the full block – then modify it and write it back. Hence, performance is wasted. Due to the characteristics of hard disk drives, random access achieves only a small fraction of available performance (at least on hard disk drives) – the mentioned optimizations are all designed having disks as persistent storage in mind. Processing of a request requires some CPU and memory resources. As requests must travel over a network from client to server, processing involves additional delay.

To tackle the issues mentioned, the non-contiguous operations and collective calls have been defined in MPI-IO. In [TGL02] access to data with MPI is classified into *four levels of access*. The levels are characterized by two orthogonal aspects: contiguous vs. non-contiguous data access, and independent vs. collective calls. Thus, the levels are: contiguous independent, contiguous collective, non-contiguous independent and non-contiguous collective. Depending on the level a different set of optimizations can be thought of.

When using collective I/O calls, clients can collaborate to access data on I/O servers efficiently. Therefore, the clients exchange information about their spacial access pattern, then MPI orchestrates the I/O operations. Thus, optimizations are possible with collective I/O which are impossible to perform with individual I/O. For example, the MPI implementation might aggregate requests for neighboring blocks into a few larger request. This not only reduces the amount of requests, but also the resulting requests have a more sequential access pattern, than the individual operations. Also, the schedule for the requests can be adjusted. For example, the order in which data is requested can be swapped.

Further, to overlap computation or communication with I/O, analogous to non-blocking communication the non-blocking I/O is defined in the standard. A non-contiguous read or write operation usually triggers a sequence of contiguous data accesses within MPI. Internally, MPI could start the requests in another order, or perform a single larger access to avoid spamming the server with a high number of operations.

Our discussion for the access semantics of MPI on Page 49 revealed that its sequential consistency requires an implementation to persist file modifications by calling `MPI_File_sync()`. The access semantics can be exploited further to cache data in faster devices of the memory-hierarchy, to aggregate accesses, or to keep the file's data in an suitable internal data structure. However, until function invocation all data access could be cached in a buffer specific to the particular file handle.

Another possibility for the user is to provide additional information to MPI about the access pattern, or to change internal defaults to values which prove to improve performance. The implementation can honor this information to perform additional and better internal operations. Technically, in MPI-2 an opaque *info object* is defined, which basically consists of key-value pairs; this info object can be supplied to some MPI-IO functions to change their internal behavior. Note that the implementation is not forced to support or even obey hints; but the implementation can use it to (hopefully) tune the internals. Unfortunately, the hint is not incorporated in the function prototypes of collective calls.

2.3.5. State of the Art

Optimized collective calls With the upcoming of MPI many algorithms were proposed to optimize collective communication [KHB+99, TRG05]. Depending on the hardware, especially network topology and interconnect, the algorithm which achieves best performance varies.

Proprietary systems could add special hardware to realize or just support collective operations. The MPI library could be adjusted to utilize the hardware. On the SciCortex machine, for example, the network devices forward network packets for broadcast operations directly to their neighbors[34]. The new Mellanox Infiniband adapter enables offloading of communication operations. This way, the communication protocol can be handled directly on the network adapter [KST+11].

As algorithms must be handcrafted towards the system, for instance for a BlueGene [AHA+05, MK05], one major problem is to pick the best algorithm for a system. Heterogeneous systems complicate this issue, inhomogeneous hardware within a cluster makes it a tedious task to determine the best algorithms because the algorithm depends on the nodes that participate in the communication. Another issue is the interconnection of homogeneous clusters in a WAN or Grid environment. Potentially different MPI implementations on each homogeneous part of such a system must collaborate in an efficient way [BTR03, ITKT00].

The influence of the starting times of processes invoking a collective operation is measured and discussed in [FPY07]. In their work process synchronization and the pattern of process arrival is proven to make an important contribution to the efficiency of selected collective algorithms and it is claimed that application developers cannot control the arrival pattern.

[34]Unfortunately, SciCortex is out of business.

Choosing the best collective algorithm Several approaches have been developed that assist in determining the best algorithm and MPI configuration. The search for the best algorithm and its configuration is an optimization problem. Theoretically, all available implementations (and configurations) could be evaluated by *brute-force*. However, due to the large number of potential configurations often *historic knowledge* about previous execution is used to predict performance. To reduce the parameter space promising candidates could be identified by a *performance model*, or by classifying configurations manually. For example, the topology of a 3D-torus restricts the number of algorithms that could achieve best performance.

The *Abstract Data and Communication Library (ADCL)* [GH07] uses historic knowledge during the application run. ADCL assumes a program performs operations iteratively – in the first few iterations ADCL evaluates a set of MPI functions to determine which one is best suited for the given problem, then this function is applied to subsequent invocations. Evaluation is performed on the fly during the program run, every time a function is invoked another internal algorithm is evaluated, until the winning function is decided. Therefore, no artificial data is communicated to evaluate the performance of the functions. Compared to other solutions ADCL provides a new interface with higher abstraction than MPI, for instance, to allow users to specify neighbor communication explicitly. Communication is performed internally by calling `MPI_IRecv()`, `MPI_Recv()`, `MPI_Put` and other point-to-point calls. The function which performs best for a problem is called for subsequent invocations. Astonishingly, already on the abstract level of MPI function calls, the function best suited depends on application, problem size and system [GFBR10]. ADCL does not care about MPI-internals, it just treats MPI functions as a black box.

In [FG08], ADCL is extended to use historic knowledge across program executions. Therefore, achieved performance is recorded in XML files and used to predict performance of future executions. This reduces the number of high-level MPI functions to evaluate. Especially, functions which are well known for their slow execution on similar problems could be skipped, speeding up the run-time decision process of ADCL and consequently reduce application run-time.

Compared to ADCL, the *Self-Tuned Adaptive Routines for MPI Collective Operations (STAR-MPI)* provides a rich set of MPI implementations for collective operations by itself [FYL06], for instance a set of 13 algorithms is supplied for `MPI_Alltoall()`. Those algorithms are inspired by the algorithmic research on efficient collective communication. Similar to ADCL, STAR-MPI evaluates the performance of the implementations on the fly during program execution. In the first stage, during each collective function call STAR-MPI evaluates the performance of an alternative implementation. Once all implementations are tested multiple times, the processes pick the best function and use it for subsequent calls. While the winning function is used to realize the MPI call, the time spent for subsequent calls to this function call is monitored in order to deal with changes in workload and environmental conditions. The algorithm is adaptive because it tries to keep the current selection as long as possible but still adjusts to changes. For instance, if the difference between previous run-times is too high, then the second best algorithm will be chosen for further calls. STAR-MPI selection of algorithms depends also on the message size, for each size a best algorithm is kept. To reduce the training phase, algorithms which are expected to behave similarly are partitioned into groups, e.g., a group for inter-node communication, or for a switched network. Now, instead of checking all algorithms of all groups, the group is selected by testing only one algorithm of each group and choosing the best group for further evaluation. On average between two to three algorithms are grouped together.

Selecting the appropriate methodology for benchmarking and to interpret measured performance results is tedious and thus there are tools which assist in creating new benchmarks. *MPIBlib* [LRO08] is a library which bundles common features of MPI benchmarking suites including timing, repetition of experiments and calculation of statistics. Features of such a library can be embedded directly into the application or into the tools selecting the fastest communication implementation.

Process placement and topology awareness Besides tuning MPI internals, the mapping of the processes to the available CPUs and nodes plays an important role – as the application communication behavior should match the node-internal and external network topology to minimize communication time (see Page 51). The *Moab* scheduler knows the physical layout of the cluster, which enables it to provide a mapping with low communication costs [CCS+06]. Mapping of processes to processors is done in *Hypertool* by a heuristic taking the communication traffic into account [WG90].

An automatic profile-guided approach is introduced in [CCH+06]. The *MPI Process Placement toolset* provides a set of tools: one tool records the communication profile, another one explores the system topology, at last an optimization algorithm determines a viable solution for the application mapping. This solution can then be applied in subsequent application executions by the user itself.

Neither of those mapping tools takes process I/O to servers into account, although at first glance it could be handled similarly to communication.

When a placement is chosen, the topology must be explored by MPI to utilize it. Starting with the recent Open MPI 1.3, *Carto*, a framework which determines the topology of the inter-process communication partners, is incorporated. By using *Carto*, for example a graph is built for the shared memory interconnect and one for the remote network. For each interconnect, the topology is stored in a weighted graph and available for other internal frameworks to tune their interconnection to remote processes. As an example, Open MPI can use the NIC closest to a particular sender process, or shared memory collective operations can take the memory distribution into account.

MVAPICH2 also aims to become topology aware to tune collective operations according to the topology. Recent work in this area has demonstrated the potential of this approach [KSVP10], for `MPI_Gather()` and `MPI_Scatter()` up to 50% improvement could be achieved on an Infiniband cluster.

Individual I/O Client-side I/O optimizations for MPI are presented in [TGL99]. *Data-sieving* is introduced to optimize independent non-contiguous I/O – it operates by issuing one request to access required data together with intermediate *holes*; unwanted data is just discarded. This avoids unnecessary seeking on hard disk drives and can improve performance, especially for very small blocks. However, it causes unneeded data to be read and transported to the clients.

General write-behind for MPI files is realized in [LCCW05, LCC+07], in their implementation all clients dedicate a piece of their memory to form a cooperative cache.

Non-blocking I/O enables overlapping communication (and computation) with I/O, the efficiency is analyzed theoretically and empirically in [BKL09]. Theoretically, a perfect overlapping of I/O with computation halves execution time.

To reduce the number of messages for non-contiguous I/O, Ching et al.[CCC+03] extend ROMIO to use ListIO in PVFS client-server communication.

In [KRVP07] a method to optimize non-contiguous (random) writes is introduced. Basically all write operations are appended to a logfile in the order they are performed. Consequently, non-contiguous operations are converted into sequential operations. This manipulation does not preserve the logical order of data in the physical file because the file format is different. However, when the file is read from the same file handle the actual file format must be recreated by replaying the log. Once a file is closed, it could be converted to the correct logical file and thus the sequential consistency of MPI is satisfied. Hence, this optimization has a good use-case for write-mostly access patterns, e.g., for checkpoints.

With shared file pointers, the actual data transfer could happen concurrently, however, in most cases I/O is suboptimal. Most implementations are not performing well because file system constructs such as a file lock or hidden files are used to realize the semantics of a shared file pointer. Some purely MPI-based implementations have been investigated and evaluated [LRT07].

Collective I/O The PVFS implementation of ROMIO reduces the metadata overhead of opening a new file by delegating the file open (and the potential create operation) to Rank 0 of the communicator specified during the open call. All other participating processes receive the information required to access the file by a broadcast from the root and thus the burden of the metadata server is reduced.

The collective optimization of the *Two-Phase* protocol is discussed in [CCC+03]. With Two-Phase, processes exchange their spatial access pattern and coordinate amongst themselves, which process will access a sequential file domain – data to be accessed is partitioned among all clients. Then, (for a read operation) the clients repeat two phases: a set of the processes reads the assigned file domains sequentially, then during the communication phase data is shipped to the clients which needs it[35].

Multiple-Phase Collective I/O, an extended version of the Two-Phase protocol, is presented in [SIC+07, SIPC09]. With Multi-Phase I/O, the communication phase is split up in several steps, in which pairs of clients communicate with each other in parallel. These multiple steps are used to progressively increase the locality of the data to be accessed by aggregating more operations into larger blocks. Then, during the I/O phase, sequential blocks can be accessed; the block size of the sequential access depends on the number of steps performed.

In [HYC05], the collective I/O scheme is adjusted to exploit the features of Infiniband.

A cooperate cache is integrated in [PTH+01], this allows GPFS to cache read and defer write operations. With a write-behind strategy, write operations are buffered. Effectively, data updates and storing the changed data on the file system happens concurrently. To ensure consistency, every physical data block is assigned to exactly one client which performs all I/O operations.

In [Wor06] an adaptive approach is introduced which automatically sets hints for collective I/O, based on the access pattern, topology and the characteristics of the underlying file system.

Higher-level I/O optimizations The following two approaches are not optimizing MPI-IO by itself, instead they provide a layer above MPI that enables further interesting optimizations.

Initially, *SIONlib* [FWP09] was developed to deal with I/O forwarding and communication topology of a BlueGene system. This library channels I/O operations to a logical file from POSIX via MPI, i.e., in a program regular POSIX (e.g., `fwrite()`) calls are used, which are mapped to a set of shared files by using the MPI-IO interface. Depending on the underlying system, one or multiple files are generated to optimize performance. Conflicts on parallel access of a shared file are reduced, yet, the number of files is less than the number of processes which minimizes metadata overhead on the underlying file system. For instance, on a BlueGene, all processors routing to a particular I/O aggregator can be mapped to one physical file. Hence, the I/O forwarder does not have to share access to the file with other aggregators. The mapping from logical to physical files is hidden behind the library and transparent to the application. The user just specifies the number of files in the library open call and whether or not collective calls should be performed.

The *Adaptable IO System* (ADIOS) [LKK+08, LZKS09] provides an abstract I/O API and library that decouples application logic from the actual I/O setting. In an XML file, the desired I/O method can be selected and parameterized – I/O operation can be realized either with HDF5, MPI (collective or independent), POSIX or several asynchronous staging methods. With ADIOS, each I/O performed in a C (or Fortran) program is annotated with a name that can be referred to in the XML file. The amount of data accessed, datatype and further attributes of the call are defined in the XML[36]. An advantage of decoupling the underlying I/O procedure is that the best-fitting implementation can be selected for a group of files.

Settings for the implementation, e.g., buffer size, can be defined without changing code. Moreover, data could be forwarded to a *visualization system*, simply discarded, or even multiple I/O methods can be selected to visualize and store data at the same time. Similar to SIONlib, the system is able to either write a shared file or to split logical I/O into several file system objects. With ADIOS, the *BP* file format is proposed that improves data locality for a single process and minimizes collisions between the processes.

[35]The communication and exchange phases are swapped for write operations.

[36]Elementary datatypes or arrays of arbitrary dimension are supported.

The API provides functions to the programmer to indicate when the computation starts or ends, or when the scientific application main loop occurs. On the one hand, this enables efficient communication to the servers without disturbing application communication. On the other hand, the pace in which data is created and written back is announced to the library. Concluding, ADIO provides a completely new API in which the programmer is forced to deal with I/O related aspects consciously – but due to the XML, system optimizations are possible without source-code modifications.

In our paper [KMKL11], the ADIOS interface is explained in detail. In this paper the interface is extended to offer visualization capabilities and improved energy efficiency.

Tuning library settings MPI libraries like IBM's *Parallel Environment* or Open MPI offer a rich set of environment specific parameters to tune the library internals towards a system or application. For example buffer sizes for the eager message protocol can be adjusted to the network characteristics. In Open MPI, the *Modular Component Architecture (MCA)* provides more than 250 parameters on a COST Beowulf cluster. The libraries provide empirically chosen defaults, which might be determined for a completely different system than the system the library is deployed on. Thus, the defaults might achieve only a fraction of theoretical performance. To provide a starting point for application specific tuning of those values, an administrator should provide appropriate values for the given system.

Chaarawi et.al. developed the *Open Tool for Parameter Optimization (OTPO)* for Open MPI which uses the automatic optimization algorithm from ADCL to determine the best settings of available MCA parameters for a given cluster system [CSGF08]. OTPO could be configured and run by administrators to set up efficient cluster defaults.

2.4. Performance Analysis and Tuning

In computer science, *performance analysis* refers to activity that fosters understanding in timing and resource utilization of applications. In terms of a single computer, the CPU, or to be more formal, each functional unit provided by the CPU, is considered to be a resource. Therefore, understanding resource utilization includes understanding run-time behavior and wall-clock time. For parallel applications, the concurrent computation, communication and parallel I/O increase the complexity of the analysis. Therefore, many components influence the resource utilization and run-time behavior; those have been discussed in Section 2.2.1.

Computational complexity theory is the field of computer science that provides methods to classify and estimate algorithm run-time depending on the problem size. Theoretical analysis of source code is usually infeasible as utilization of hardware at run-time can only be roughly estimated. Therefore, in practice, theoretical analysis is restricted to small code-pieces or clear application kernels, and typically software behavior is measured and assessed.

Programs can be classified according to their utilization characteristics and demand – important algorithms are categorized into 13 *motifs* [ABC$^+$06]. Most applications could be thought of as a combination of the basic functionality required by those motifs. But even so, the characteristics of each real program must be analyzed individually.

In this section, it is first shown how application design and software engineering can assist in developing performance-demanding applications (Section 2.4.1). Those methods focus on integrated and automatic development to achieve efficient and performant applications.

As scientific programs usually require a huge amount of resources, one could expect them to be especially designed for performance. Unfortunately, that is not the case. One reason is that many scientific codes evolved over decades at a time when performance has been of low priority. Usually performance is analyzed after the correctness of the program has been evaluated. At this late stage, a functional version of the code exists, which has been tested to some extent. Thus, a complete redesign is usually out of reach for the scientist because it is time-consuming.

Once the behavior of an application is understood, it can be modified to make the code more efficient with respect to resource consumption. *Tuning* refers to the iterative process in which the current status is systematically analyzed and optimized, it is described further in Section 2.4.2).

Plenty of tools exist which assist the developer and user in tuning an application and the underlying run-time system according to this *closed loop of performance tuning*. In Section 2.4.3, several tools to analyze sequential programs are introduced, followed by tools suitable for parallel applications in Section 2.4.4. Capabilities of each tool are illustrated on the same parallel code, i.e., all tools are tested with the same application and parameters. The experiment run uses our partial differential equation (PDE) solver `partdiff-par`[37], which implements the Jacobi and Gauß-Seidel methods. The sequential version called *partdiff-seq* excludes the MPI calls; apart from these modifications, the code base is identical.

Since behavior of a program is often analyzed after the program terminates, at last a common file format is introduced, which stores program activity for such a *post-mortem* analysis (Section 2.4.5). Some parts of this chapter have been published in the book [MMK+12].

The simulation environment that is presented later in this thesis will assist in post-mortem performance analysis but also in a performance oriented development of applications. Therefore, understanding these areas is important.

2.4.1. Developing Applications for Performance

In the industry the process of system and application tuning is often referred to as *performance engineering*. Software engineers designed special methods to embed performance engineering into the application development. With these approaches, performance is considered explicitly during the application design and its implementation.

Important existing approaches that embed performance engineering into the development cycle are presented next. Unfortunately, the research and processes in industry are not integrated in state-of-the-art HPC application development, although there are a few tools which assist in the development of parallel applications.

Computer-aided software engineering Tools and methodologies serve the developer during the lifecycle of software. In 1982, the term *Computer-aided software engineering* (CASE) was formed:

> *"Computer-aided software engineering is the scientific application of a set of tools and methods to a software system which is meant to result in high-quality, defect-free, and maintainable software products"* [Wik11], Wikibook: Introduction to Software Engineering/Tools/Modeling and Case Tools

CASE tools assist the programmer in the software development process, which basically consists of the following phases: requirement engineering, analysis, design, coding, documentation and testing. Information of one phase is linked to other phases by those tools. This way, information about early phases is accessible in later phases; for example, the requirements are referenced directly in the source code.

Automatic code generation from a model The *Model-Driven Architecture* (MDA) is a design methodology in which abstract models of the desired application are refined until a code skeleton representing the model is generated. Multiple existing technologies are bundled together with tools to assist in top-down development. For example, developers can create the models in appropriate description languages like the

[37]This PDE solver is frequently used in the lectures of our working group as an exercise to implement various parallel programming models. More information about the program is given in Section 7.9.1.

Unified Modeling Language (UML). UML class diagrams are then translated into Java skeletons; a reverse engineering of modified classes into the model is also possible.

CASE and MDA do not explicitly honor the non-functional requirement for performance. Certainly, due to the difficulty to meet this requirement in later stages of the development, as it is hard to estimate performance of complex applications beforehand.

Integrating performance aspects into code development In the industry, the direction towards design for performance – *Software Performance Engineering* [CUS01] (SPE) – is a newer development. Performance tuning guides or best practices on the other hand, are usually available for all programming languages.

Performance evaluation is anchored into the software development cycle; for example, see [CUS01, BGM03, BFG$^+$04, PW05, SSM$^+$09]. The book [CUS01] discusses performance-oriented design and provides best-practice and a methodology for predicting performance of object-oriented software quantitatively. Microsoft targets performance modeling of the .NET language explicitly in [Cor04] and discusses software development for performance.

Continuous performance management of Java or .NET applications is made available in the product *dynaTrace* [dyn10]. During the entire software life-cycle, the performance is monitored and compared with earlier results to detect performance degradation due to code modification.

The *Tuning and Performance Utilities* have been recently integrated into the *Eclipse Parallel Tools Platform* [SSM$^+$09] to make them available to a wider audience (refer to section 2.4.4 for a description of TAU).

Modeling and estimating performance The *Object Management Group*, the standardization consortium which defined UML, extended the diagrams by defining annotations for performance requirements [Obj03]. Several systems exists which feed these annotations into performance models and simulators, this is discussed in [BGM03, BFG$^+$04, WPP$^+$05, PW05].

For example, in [PW05], interactions between actors and system components are evaluated by converting a use case map into a performance model. *PUMA* [WPP$^+$05] is a tool set which actually translates arbitrary design models with performance annotations into a *Core Scenario Model* (CSM) containing the performance relevant aspects. This CSM can then be evaluated with various performance models. In their paper, an example is given in which behavior is modeled in UML and converted into *Petri Nets* and a *Queueing Network Model*.

Resource and network utilization of distributed systems is accounted for in the *Layered Queueing Network* (LQN) model: An entity performs activities which consume some resource, need time, impose wait delay or communicate with other entities via synchronous or asynchronous messages. As a third possibility, a request can be forwarded to another entity. The LQN model honors resource contention. It has been used to study many kinds of distributed systems [Fra00]. Simple models can be solved analytically to compute mean values of *service time* and *utilization* of it's components. To solve more sophisticated models, simulation is needed.

The Trace-Based Load Characterization (TLC) in [HWRI99] records trace information in a production environment and feeds it into an LQN model.

The integrated modeling and simulation environment CoFluent Studio [CoF09, CoF10] allows designing embedded multiprocessor systems and their software. Developers formulate requirements and specifications and finally specify their algorithm as an abstract sequence of functions. Similar to MDA, the model is synthesized into architecture specific code, but here the functions are considered as well. Then the performance characteristics of software under the given hardware are simulated and analyzed prior to implementation. The system model for embedded systems contains models for computing units, system buses, network routers, point-to-point links, memory and I/O interfaces including schedulers. Full system simulation of the hardware/software system in a virtual prototype is also supported but does not target HPC.

Designing hardware and software together In hardware/software *co-design*, both the perspective of software and hardware are considered which is especially important for embedded systems [DM94]. This means that the hardware is designed with the software in mind which will run later on the hardware. Through this concurrent development, a synergy is achieved – the hardware supports the required functionality and the software is geared towards the hardware.

With this approach performance critical software functionality can be directly implemented in the hardware. To avoid the high costs involved in the design of an *Application-Specific Integrated Circuit* (ASIC), often *Field-Programmable Gate Array* (FPGA) technology is deployed. An FPGA can be reprogrammed online to implement a particular function.

Recently, hardware/software co-design became a hot topic for designing HPC environments [Ebc05, HMD+10, KWM+10] as it promises to increase energy efficiency of applications and paves the road for exascale:

> *"The traditional model of single-domain research activities where hardware and software techniques are explored in isolation will not address the current challenges."* [KBB+08]

The similarities and differences between HPC and embedded systems are discussed in [HMD+10], for example, in HPC the processor is not designed for a specific application as it could be for embedded systems.

On a side note, another, similar approach to co-design is to configure a system towards the requirements of the software. However, that means the system provides capabilities to be reconfigured according to the demand of the software. In earlier times this was called *configurable computing* [MSHA+97]. Slight reconfiguration and modification of hardware was recently deployed; for example, with the *Fermi* architecture of its GPUs, NVIDIA can partition available memory into shared memory and as local cache, depending on the user configuration [Pat09].

Performance-oriented development of parallel applications An early tool for semi-automatic development of parallel programs was *Hypertool* [WG90]. Hypertool takes a sequential data-partitioned program and generates code for parallel execution based on the user-supplied partitioning. Additionally, performance estimates and quality measures for the code are computed. In their paper, state-of-the-art software shows that the development issues were already a topic in the 90th.

Software Engineering for Parallel Processing [WK94] (SEPP) was an European project which tried to realize the design for performance during development of parallel applications by developing new CASE tools. Among these tools, a simulator should allow rapid prototyping and benchmarking. The contribution to the *EDPEPPS* environment [DZV+97] simulate parallel programs that use the *Parallel Virtual Machine* (PVM) message passing paradigm.

Computer Aided Parallel Software Engineering (CAPSE) was a similar approach, developed in parallel to SEPP, it is discussed in [GHKV96]. N-MAP is an environment developed within CAPSE that predicts application performance already during the development phase [FM95, FJ95]. In [FJ00], N-MAP is claimed to be superior to performance models, because a model abstracts from reality which they state *"is suspect to fail"*. Instead they propose that the developer prototypes the distributed algorithm directly in a N-MAP specification. The N-MAP specification is basically a language dialect derived from C in which inter-process communication is programmed explicitly. Functions can be implemented that represent the real algorithm, or the computation time can be specified by a particular function which might include random variables of some distributions. This implicit performance model is executed inside a simulator that multiplexes a single processor among all parallel processes and executes them sequentially. With this approach, the computation time will be accurately represented iff the real algorithm is encoded. Further, with this approach, the simulation prototype can be transformed into a real PVM executable. With N-MAP, the workload of application processes can be optimized online: performance metrics are measured by sensors, future states and utilization are estimated, and then appropriate actions are initiated pro-actively.

Pllana et al. [PBXB08] propose performance-oriented program development instead of code-based performance tuning. With *Performance Prophet*, they describe a system which combines mathematical modeling and discrete-event simulation to predict application performance. In their workflow, a developer specifies the program behavior in an UML extension which is geared towards modeling parallel applications. Then, the program is translated into simulator code and executed to predict its performance. The execution time of code blocks is modeled mathematically by assigning a cost function to each code block. Cost functions could be designed by measuring behavior of the code for some inputs and extrapolating them to arbitrary inputs. In their approach, inter-process communication and the control flow are modeled with discrete-events.

All the mentioned tools try to change the way of software engineering to incorporate the performance relevant aspects early in the development cycle. In contrast, the closed loop of performance tuning optimizes programs after a running version exists.

2.4.2. Closed Loop of Performance Tuning

The localization of a performance issue on an existing system is a process in which a hypothesis is supported by measurement and theoretic considerations. Measurement is performed by executing the program while monitoring run-time behavior of the application and the system. In general, tuning is not limited to source code, it could be applied to any system.

The schematic view of the typical iterative optimization process, the *closed loop of performance tuning*, is shown in Figure 2.11:

1. To measure performance in an experiment, the environment consisting of hardware and software including their configuration is chosen, also the appropriate input, i.e., problem statement, is decided. While theoretic considerations allow projecting run-time behavior of arbitrary input sets, monitoring is limited to instances of program input. Thus, it might happen that optimizations made for a particular configuration degrade performance on a different setup or program input. Therefore, multiple experimental setups could be measured together. Often, the measurement itself influences the system by degrading performance, which must be kept in mind. Picking the appropriate measurement tools and granularity can reveal the relevant behavior of the system.
2. In the next step, obtained empirical performance data is analyzed to identify optimization potential in the source code and on the system. As execution of each instruction requires some resources, the code areas must be rated. First, hot-spots – code regions where execution requires significant portions of run-time (or system resources), are identified. Then, optimization potential of the hot spots is assessed based on potential performance gains and the estimated time required to modify the current solution. Knuth describes this issue excellently:

   > *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified. "* [Knu79]

   Changing a few code lines to improve run-time by 5% is more efficient than to recode the whole input/output of a program, especially if I/O might account for only 1% of the total run-time. However, care must be taken when the potential is assessed, depending on the overall run-time, a small improvement might be valuable. From the view of the computing facility, decreasing run-time of a program which runs for millions of CPU hours by 1% yields a clear benefit by saving operational costs in form of 10.000 CPU hours (which is about 1.5 CPU years).

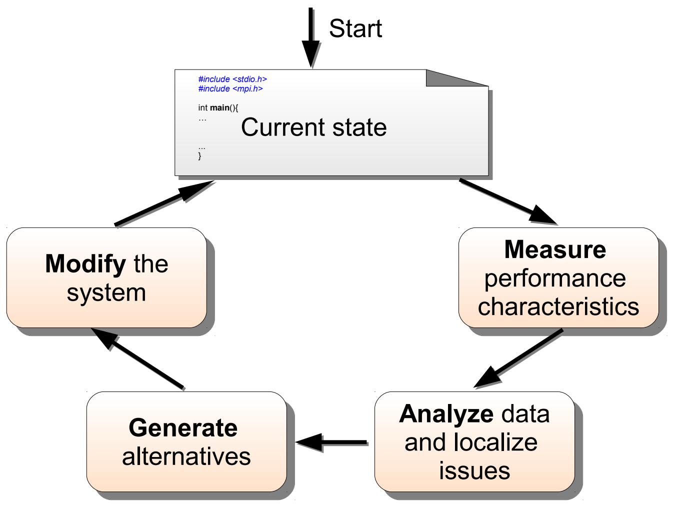


Figure 2.11.: Closed loop cycle of optimization and tuning.

"The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being practiced by penny-wise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs. In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering." [Knu79]

3. Based on the insight gained by the analysis, alternatives for the implementation are generated, or system modifications are considered, respectively, which apparently mitigate the observed performance issue. This is actually the hardest part of the tuning because it requires that the behavior of the new system can be predicted or estimated. However, often in practice multiple potential alternatives are evaluated, and based on the results the best one is chosen. Sometimes a modification requires modifications that affect other parts of the program and might even degrade overall performance. With increasing experience and knowledge of the person tuning the system, the number of alternatives is reduced as the future behavior can be better anticipated.

4. At the end of a loop the current system is modified, i.e., one of the performance relevant layers is adjusted, to realize the potential improvement of the new design. The system is reevaluated in the next cycle until the time required to change the system outweights potential improvements, or potential gains are too small because the performance measured is already near-optimal. However, in practice, in most cases the efficiency of the current solution is not estimated; instead, the current run-time is considered to be potentially saved.

Large systems/applications are often complex; thus it is important to reduce the complexity. Therefore, application logic can be reduced to the core of the algorithm – the application *kernel*, which is then implemented in a *benchmark*. Such a benchmark can already provide a rough estimate for running the full application on a system.

In the following paragraphs, a few aspects of the loop are highlighted and discussed: the layers and components that are relevant, then the nature and diversity of performance data, concepts that enable collecting performance data, and the analysis of collected data.

Relevant layers and components In the context of this thesis all of the layers and components participating in an application execution[38] are subject to the optimization process. The information about state and activities of all of them could be important to assess the observations.

[38]See Section 2.2.1 on Page 27 for the layers involved in parallel programming.

Especially, if a component's characteristics contribute to the observed performance to a large extent, understanding its internals might enable tuning of future run-time behavior. To identify reasons for a bottleneck it is mandatory to look inside the particular component (or activity). Without this knowledge, one can deduct only that the process within a particular component causes the suboptimal behavior, but not the reason. It could happen, for instance, that dependencies of the spotted behavior are the cause, while a layer itself is very efficient. Also, *external* (background) activities influence the observable performance. Therefore, during a measurement observations must be accounted to the right activity, otherwise the resource consumption might be attributed to the wrong cause.

Consequently, striving for the best analysis capabilities requires us to obtain a transparent view of all concurrent activities to a very detailed level, however, in a complex system such as a parallel computer this is infeasible because every event must be accurately captured with a global time. Therefore, a promising compromise is to look at relevant activity on all layers and components and the application code, this way performance bottlenecks can be assigned to the particular layer. Furthermore, such a concepts permits identifying the particular application code that limits performance.

The implementation of a *stub*[39] for a layer which pretends to perform the operation allows an implicit performance evaluation of all layers above or below the replaced layer. An example of this technique can be found in [Kun06]; in this thesis the persistency layer of PVFS has been replaced with a stub to reveal the achievable metadata and data performance of PVFS.

Performance data In the closed loop, data is collected which characterizes the application run and system utilization. There are many types of data that can be collected. For example, the operating system provides a rich set of interesting characteristics such as memory, network and CPU usage. These characterize the activity of the whole system, and sometimes usage can even be assigned to individual applications.

The semantics of this data can be of various kinds, usually, a *metric* defines the measurement process and the way subsequent values are obtained. For example, *time* is a simple metric, which indicates the amount of time spent in a program, function or hardware command. The *resident set size* (RSS), i.e., the amount of occupied memory, is another metric.

Depending on the metrics, data can be obtained for a subset or all layers involved in application execution. For example, data can be collected from hardware devices or generated within software, either by the operating system, the application or from additional performance analysis tools.

The expressiveness of a metric depends on its measurement process: A metric can be fine-grained – just describing a single operation executed on a particular component at a given time; or it can be coarse-grained – aggregating all activity that is observed on a given component or even on the whole system.

One way of managing performance information is to store *statistics*, e.g., absolute values like number of function invocations, utilization of a component, average execution time of a function, or performed floating point operations. Statistics of the activity of a program is referred to as *profile*. A profile aggregates events by a given metric, for example by summing up the inclusive duration of function calls. In contrast to a *profile*, a *trace* records events of a program together with a timestamp and thus it provides the exact execution chronology and, therewith, allows analysis of temporal dependencies. External metrics like hardware performance can be integrated into traces as well. Tracing of behavior produces much more data, potentially degrading performance and distorting attempts of the user to analyze observation; therefore, in many cases only profiles are recorded. A combination of both approaches can be applied to reduce the overhead while still offering enough information for the analysis: Events that happen during a timespan can be recorded periodically as a profile for an interval – this allows analysis of temporal variability; By generating profiles for disjoint code regions, behavior of the different program phases can be assessed.

Once a way of aggregation is chosen, the performance data must be correlated to the interesting application's behavior and source code. Depending on the measurement process the assignment of information to

[39]In software engineering a stub refers to code that implements a method (or interface). It may simulate the behavior of existing code for some inputs to ease testing of other components that rely on the method (or interface).

the cause can be impossible, for example a statistic cannot reveal the contribution of concurrent activity.

Typically, it is not possible to gather detailed information for hardware activity because that would imply much overhead. Hardware sensors are available in some devices, which measure internal utilization or other metrics like energy consumption or error rate. For example, all decent consumer CPUs have built-in programmable performance counters. Depending on the CPU architecture, a wide range of counters are available which measure efficiency of cache and branch-prediction, the number of instructions run or the number of floating point operations performed. Modern processors provide a variety of over one hundred counters covering different aspects of a CPU. However, using performance counters of a CPU for analysis has a flaw – the CPUs provide only a limited number of registers to manage the counter values. Effectively only 4 to 6 counters can be active at any given time. This issue can be tackled by multiplexing the metrics from time to time and thus over long running processes a good estimate of the values can be provided. Some of the recorded counters actually capture the aggregate events for the full microprocessor, that means multiple physical CPUs share one event. Consequently, this statistical data cannot be associated with a particular process. Newer network interface cards accumulate the number of packets and the amount of data received and transmitted.

Collecting performance data There are several approaches of measuring the performance of a given application. A *monitor* is a system which collects data about the program execution. Approaches could be classified based on *where*, *when* and *how* run-time behavior is monitored.

A monitor might be capable of recording activities within an application (e.g., function calls), across used libraries, activities within the operating system such as interrupts, or it may track hardware activities; in principle, data can be collected from all components or layers described in Figure 2.7. Most *monitors* rely on software to measure the state of the system, data from available hardware sensors is usually queried from the software on-demand. Hardware monitors are too expensive, complicated and inflexible to capture program activity in detail.

Usually, changes are made to the program under inspection to increase analysis capabilities; the activity that alters a program is called *instrumentation*. Popular methods are to modify source code, to relink object files with patched functions or to modify machine code directly [SMAb01]. During execution, such a modified program invokes functions of the monitoring environment to provide additional information about the program execution and the system state. The described instrumentation functionality could also be supported directly by the (operating) system and, therewith, it could be possible to collect performance data without modifying the application.

As a software monitor requires certain resources to perform its duty (those can be considered as overhead), monitoring of an application perturbs the original execution. Observed data must be kept in memory and might be flushed to disk if memory space does not suffice. Additionally, computation is required to update the performance data. The overhead depends on the characteristics of the application and system – it might perturb behavior of the instrumented application so much that an assessment of the original behavior is impossible. Therefore, to reduce the overhead users enable only a subset of the potential features in a typical optimization setup.

Automatic instrumentation by tools usually tries to gather as much information as possible, therefore, the overhead is higher than with an approach in which the user modifies the source code manually. Normally, the user starts with an automatic instrumentation, then if the overhead is too high filters are applied until the trace file includes just enough information for the analysis. Some tools automatically filter activity if an event is fired to often, if the overhead of the measurement system itself grows too high. If filtering still incurs too much overhead, then interesting functions can be manually instrumented, i.e., by inserting calls to the monitoring interface by hand.

Additionally, a selective activation of the monitor can significantly reduce the amount of recorded data. Also, a monitor could sample events at a lower frequency, reducing the overhead and the trace detail level on the same extent.

Analyzing data Users analyze the data recorded by the monitoring system to localize optimization potential. Performance data is either recorded during program execution and assessed after the application finished, this approach of *post-mortem* analysis is also referred to as *offline* analysis. An advantage of this methodology is that data can be analyzed multiple times and compared with older results. Another approach is to gather and assess data *online* – while the program runs. This way feedback is provided immediately to the user, who could adjust settings to the monitoring environment depending on the results.

Due to the vast amount of data, sophisticated tools are required to localize performance issues of the system and correlate them with application behavior and finally identify source code causing them. Tools operate either manually, i.e., the user must inspect the data himself; a *semi-automatic tool* could give hints to the user where abnormalities or inefficiencies are found, or try to assess data automatically. Tool environments, which localize and tune code automatically, without user interaction, are on the wishlist of all programmers. However, due to the system and application complexity those are only applicable for a very small set of problems. Usually, tools offer analysis capability in several *views* or *displays*, each relevant to a particular type of analysis.

2.4.3. Available Tools for Analysis of Sequential Programs

There exist plenty of tools that assist in performance analysis and optimizations of sequential code, a handful of tools of different categories are briefly introduced: *GNU gprof* generates a profile of the application in user-space. *OProfile* can record and investigate application and system behavior including activity of the Linux kernel. CPU counters can be related to the individual operations. *PAPI* is a library which accesses CPU counters and provides additional hardware statistics. *Likwid* is a lightweight tool suite that reads CPU counters for an application. *LTTng* traces and visualizes activity of processes and within the kernel. However, compared to OProfile symbolic information of the application program is not supported.

All tools mentioned are licensed under an open and free license. The state of the latest stable versions available is discussed as of February 2011.

GNU gprof

GNU compilers can be instructed to include code into a program that will periodically collect samples of the program counter. During run-time profiles of function call timings and the *call graph*[40] are stored in a file. This profiling data can then be analyzed by the command line tool `gprof`.

To demonstrate the application of the tool, an excerpt of the `gprof` output for a run of the `partdiff-seq` PDE solver is given in Listing 2.1. In the *flat profile* (up to Line 12), the time is shown per function. While 6 functions have been invoked once (Lines 6 to 11), all run-time is spent in the function `calculate()` (Line 6). The textual representation of the call graph is also provided starting with Line 15 of the output. In Line 20 and Line 21, it is shown that the function `calculate()` is called from main once, additional invocations from different functions would generate further sections. The main function calls all 6 subroutines (Lines 24-30).

Most platforms provide tools alike to `gprof` to analyze performance of sequential programs. To analyze a parallel application, a profiler must be aware of the parallelism and provide an approach to handle it, for example, by generating one output for each of the spawned processes.

Listing 2.1: Excerpt of a `gprof` output for partdiff-seq

```
1 Flat profile:
2
3  Each sample counts as 0.01 seconds.
4   %   cumulative   self              self     total
5  time   seconds   seconds    calls  s/call   s/call  name
6 100.05     30.95    30.95        1    30.95    30.95  calculate
7   0.00     30.95     0.00        1     0.00     0.00  AskParams
```

[40]The call graph is a directed graph providing information about function invocation, the nodes of the graph represent functions and the edges function calls.

```
  0.00      30.95      0.00         1      0.00      0.00  DisplayMatrix
  0.00      30.95      0.00         1      0.00      0.00  allocateMatrices
  0.00      30.95      0.00         1      0.00      0.00  displayStatistics
  0.00      30.95      0.00         1      0.00      0.00  initMatrices

[...]

                       Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.03% of 30.95 seconds

index % time    self  children    called     name
                30.95    0.00      1/1           main [2]
[1]    100.0   30.95    0.00       1         calculate [1]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00   30.95                 main [2]
               30.95    0.00       1/1           calculate [1]
                0.00    0.00       1/1           AskParams [3]
                0.00    0.00       1/1           initMatrices [7]
                0.00    0.00       1/1           allocateMatrices [5]
                0.00    0.00       1/1           displayStatistics [6]
                0.00    0.00       1/1           DisplayMatrix [4]
```

OProfile

OProfile[41] provides a sophisticated system-level profiling for the Linux operating system. Compared to `gprof`, OProfile gathers information from all running processes at the same time. Also, kernel internals are captured, and a configurable set of hardware performance counters. Profiling must be enabled and started by the super user, then all activities are recorded. Several tools are provided that analyze data recorded from user-space.

An example for system-level profiling of a desktop system running our PDE is given in listing 2.2. In this profile, the concurrent activities can be identified, also the time spent in kernel space[42] and in certain libraries becomes apparent. For each application, the user can create an individual profile, covering activities of the particular application as well as activities triggered by library and system calls.

A very handy feature of the OProfile system is that source code (and additionally assembler) can be annotated with the performance observations. This makes it easier to localize the time-consuming lines. In Listing 2.3, the source code of `calculate()` is shown with the sample count. The time spent in individual code lines becomes apparent, for example, 12% of the run-time is spent in Line 20, showing the optimization potential within this line.

Accessible hardware counters on the desktop system are shown in Listing 2.4[43].

Listing 2.2: Excerpt of a system-wide OProfile output while running partdiff-seq

```

CPU: Intel Architectural Perfmon, speed 1199 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (No unit mask) count 100000
samples  %        image name               app name                 symbol name
1068951  71.0457  partdiff-seq             partdiff-seq             calculate
185685   12.3412  no-vmlinux               no-vmlinux               /no-vmlinux
26927     1.7896  libgstflump3dec.so       libgstflump3dec.so       /usr/lib/gstreamer-0.10/libgstflump3dec.so
16738     1.1125  libspeexdsp.so.1.5.0     libspeexdsp.so.1.5.0     /usr/lib/libspeexdsp.so.1.5.0
13980     0.9292  Xorg                     Xorg                     /usr/bin/Xorg
12617     0.8386  libQtGui.so.4.7.0        libQtGui.so.4.7.0        /usr/lib/libQtGui.so.4.7.0
12110     0.8049  libQtCore.so.4.7.0       libQtCore.so.4.7.0       /usr/lib/libQtCore.so.4.7.0
10177     0.6764  libxul.so                libxul.so                /usr/lib/firefox-3.6.13/libxul.so
8375      0.5566  libmozjs.so              libmozjs.so              /usr/lib/firefox-3.6.13/libmozjs.so
8311      0.5524  libflashplayer.so        libflashplayer.so        /usr/lib/flashplugin-installer/libflashplayer.so
6670      0.4433  libdrm_intel.so.1.0.0    libdrm_intel.so.1.0.0    /lib/libdrm_intel.so.1.0.0
6097      0.4052  libpulsecommon-0.9.21.so libpulsecommon-0.9.21.so /usr/lib/libpulsecommon-0.9.21.so
4185      0.2781  oprofiled                oprofiled                /usr/bin/oprofiled
4111      0.2732  libpthread-2.12.1.so     libpthread-2.12.1.so     pthread_mutex_lock
3819      0.2538  libgstreamer-0.10.so.0.26.0 libgstreamer-0.10.so.0.26.0 /usr/lib/libgstreamer-0.10.so.0.26.0
3255      0.2163  libpthread-2.12.1.so     libpthread-2.12.1.so     pthread_mutex_unlock
```

Listing 2.3: Excerpt of the annotated partdiff-seq source code

```
1                :/* ************************************************************************** */
```

[41]Visit `http://oprofile.sourceforge.net/` for further information.

[42]By providing the kernel symbol table, all activity inside the kernel is accounted in a fine-grained manner to the kernel-internal symbols; in the listing the `no-vmlinux` symbol aggregates all kernel activities.

[43]The output was created by using `opcontrol -list-events`.

```
                :/* calculate: solves the equation                                  */
                :/* ******************************************************************* */
                :void calculate(void)
                :{ /* calculate total: 1068951 99.9979 */
                :  int i,j;                                 /* local variables for loops  */
                :
                :  while(term_iteration>0)
                :  {
     2 1.9e-04  :    maxresiduum=0;
  5088  0.4760  :    for(i=1;i<N;i++)                                 /* over all rows      */
                :    {                                                /*                    */
 74459  6.9655  :      for(j=1;j<N;j++)                               /* over all columns   */
                :      {
484776 45.3497  :        star=                 -Matrix[m2][i-1][j]
                :            -Matrix[m2][i][j-1]   -Matrix[m2][i][j+1]
                :                              -Matrix[m2][i+1][j]   +4.0*Matrix[m2][i][j];
                :        residuum=getResiduum(i,j);
   315  0.0295  :        korrektur=residuum;
134782 12.6086  :        residuum = (residuum<0) ? -residuum:  residuum; // if (residuum<0) residuum=residuum*(-1);
135906 12.7137  :        maxresiduum = (residuum < maxresiduum) ? maxresiduum : residuum;
                :
 66943  6.2624  :        Matrix[m1][i][j]=Matrix[m2][i][j]+korrektur;
                :      }
                :    }
                :    stat_iteration=stat_iteration+1;
    56  0.0052  :    stat_precision=maxresiduum;
                :    checkQuit();
                :  }
                :}
                :
```

Listing 2.4: Available OProfile events (accessible hardware counters) on an Intel Nehalem system

```

OProfile: available events for CPU type "Intel␣Architectural␣Perfmon"

See Intel 64 and IA-32 Architectures Software Developer's Manual
Volume 3B (Document 253669) Chapter 18 for architectural perfmon events
This is a limited set of fallback events because oprofile doesn't know your CPU
CPU_CLK_UNHALTED: (counter: all)
        Clock cycles when not halted (min count: 6000)
INST_RETIRED: (counter: all)
        number of instructions retired (min count: 6000)
LLC_MISSES: (counter: all)
        Last level cache demand requests from this core that missed the LLC (min count: 6000)
        Unit masks (default 0x41)
        ----------
        0x41: No unit mask
LLC_REFS: (counter: all)
        Last level cache demand requests from this core (min count: 6000)
        Unit masks (default 0x4f)
        ----------
        0x4f: No unit mask
BR_INST_RETIRED: (counter: all)
        number of branch instructions retired (min count: 500)
BR_MISS_PRED_RETIRED: (counter: all)
        number of mispredicted branches retired (precise) (min count: 500)
```

PAPI and Likwid

There are several approaches that access CPU counters. *PAPI*, the *Performance API* [MCW+05, TJYD09], is a portable library which enables programs to gather performance events of all modern x86 and Power processors.

PAPI evolved from an interface for CPU counters to the component-oriented PAPI-C [TJYD10], which extends the original PAPI to capture counters from a multitude of sources – ACPI, `lm_sensors` for temperature, or specific network interfaces (Myrinet). PAPI-C leverages the existing inhomogeneous vendor (and software) interfaces.

An alternative library and user space program to profile hardware counters for an application is Likwid [THW10]. Likwid is a user space tool that profiles hardware counters for the whole application execution. When the tool is started, it initializes the counters, then starts the application, and once the application terminates, the counters are stopped and a brief report is created. A small API is offered that allows a developer to restrict the measured code regions and, furthermore, it supports to split execution into phases that can be assessed individually.

An exemplary profile of the floating point group of `partdiff-seq` is given in Listing 2.5. In this example the Intel processor operated on an average clock of 3.427 GHz (Line 28), the number of *cycles per* *instruction*(CPI) is 0.52, which means every other cycle one instruction is *retired*[44] on the core. The number of double precision floating point operations per second is about 1.7 GFlop/s. One can easily conduct from the average clock speed and CPI that if we execute one instruction every other cycle, then effectively 1.7 Gigainstructions are executed per second, which means most operations were floating point instructions. This little example already demonstrates the power of hardware counters and simple theoretic considerations.

Available groups for Likwid are shown in Listing 2.6. The group to measure is selected upon startup of Likwid. Note that the memory group is aggregated for all cores on a given chip.

Listing 2.5: Excerpt of the likwid output for partdiff-seq

```

-------------------------------------------------------------
-------------------------------------------------------------
CPU type:        Intel Core Westmere processor
CPU clock:       2.79 GHz
Measuring group FLOPS_DP
-------------------------------------------------------------
/opt/likwid/bin/likwid-pin -c 1 ./partdiff-seq 0 2 100 1 2 10000
[likwid-pin] Main PID -> core 1 - OK

[...]
< program output >
[...]

+-----------------------------------------+-------------+
|                  Event                  |   core 1    |
+-----------------------------------------+-------------+
|            INSTR_RETIRED_ANY            | 2.02167e+11 |
|          CPU_CLK_UNHALTED_CORE          | 1.04973e+11 |
|           CPU_CLK_UNHALTED_REF          | 8.55369e+10 |
|     FP_COMP_OPS_EXE_SSE_FP_PACKED       | 1.16896e+06 |
|     FP_COMP_OPS_EXE_SSE_FP_SCALAR       | 5.22953e+10 |
| FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION    | 1.0281e+07  |
| FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION    | 5.22862e+10 |
+-----------------------------------------+-------------+
+---------------------------+-----------+
|          Metric           |  core 1   |
+---------------------------+-----------+
|        Runtime [s]        |  37.5892  |
|        Clock [MHz]        |  3427.22  |
|            CPI            |  0.51924  |
| DP MFlops/s (DP assumed)  |  1696.62  |
|      Packed MUOPS/s       |  0.037923 |
|      Scalar MUOPS/s       |  1696.55  |
|        SP MUOPS/s         |  0.333533 |
|        DP MUOPS/s         |  1696.25  |
+---------------------------+-----------+
```

Listing 2.6: Available Likwid groups on an Intel Nehalem system

```
BRANCH: Branch prediction miss rate/ratio
CACHE: Data cache miss rate/ratio
CLOCK: Clock of cores
DATA: Load to store ratio
FLOPS_DP: Double Precision MFlops/s
FLOPS_SP: Single Precision MFlops/s
FLOPS_X87: X87 MFlops/s
L2: L2 cache bandwidth in MBytes/s
L2CACHE: L2 cache miss rate/ratio
L3: L3 cache bandwidth in MBytes/s
L3CACHE: L3 cache miss rate/ratio
MEM: Main memory bandwidth in MBytes/s
TLB: TLB miss rate/ratio
VIEW: Double Precision MFlops/s
```

LTTng

The *Linux Trace Toolkit*[45] provides a user-space and a kernel-space tracer [FDD09]. The user space tracer (UST) provides an API by which a developer can instrument the source code. Also, the *GNU Project De-*

[44]Many modern CPUs execute operations that might depend on operations that are still in the pipeline, for example, speculative execution of branches before the branch condition is actually evaluated; if the prediction is correct, the results of the executed instructions are stored, otherwise these results are invalid and must be discarded. When there are no conflicts, a processor retires the instruction allowing write back of the results. For the speed of the execution the speculatively executed instructions are irrelevant, therefore, they are not covered by the CPI metric.

[45]Documentation of the *Linux Trace Toolkit* including all tools is elaborate and available on `http://lttng.org/`. A quick-start tutorial can be found here: `http://omappedia.com/wiki/Using_LTTng`.

bugger (gdb) can use UST to record GDB tracepoints. The kernel tracer captures activity for all processes and the system.

Traces are recorded in the *Common Trace Format* (CTF)[46]. The user-space tracer records events with zero-copy[47], resulting in a low overhead of 700 ns per event (according to the documentation). The overhead for the kernel-tracer is even lower.

LLTV is the *LTTng Viewer*, a tool providing statistical, graphical and text-based *views* of the recorded traces. Internally, the individual views are realized as modules. It has been shown capable of handling traces with a size of 10 GiB. Another viewer is incorporated into the *eclipse*[48] IDE as a plugin. With *LTTng*, several traces can be visualized concurrently; this technique has been used to view traces of a virtual machine together with a trace of the XEN host system [DD08].

Exemplary screenshots of LLTV for the desktop system running partdiff-seq are discussed next. In Figure 2.12, the *event view* and *control flow view* are presented. The *event view* – shown in the upper half of the screenshots – lists individual events in a textual representation while the *control flow view* – visible in the lower half of the screen – shows the activity in a timeline for each individual process. Graphical representation of the trace encodes the activity of each process in colors; green, for instance, encodes that the process on the left executes in user space (this is *partdiff-seq*).

The *statistical view* and *resource view* are shown in Figure 2.13. In the statistical view, aggregated information of all kinds of activity is provided. In the resource view, the activity on each CPU and interrupt can be observed, white color encodes user-space activity dispatched on a CPU. For easy analysis, the screenshot has been modified to show the likely[49] execution of *partdiff-seq* on the CPUs with green color. The vertical line marks the start of the program, then the single threaded program is migrated between all 4 cores on the quad-core system.

Consequently, with the tool, interactions between processes and system activity at a given time can be analyzed extensively. Note that while the overhead of the tracer is very low compared to other tools tracing 50 seconds of system activity created a set of files with the aggregated size of 57 MiB.

The tool provides capabilities to filter events, which is unfortunately not available for the graphical trace representation.

2.4.4. Available Tools for Analysis of Parallel Programs

Compared to the tools for sequential programs, tools for parallel applications are more involved. In the following, a few tools are presented which are aware of the programming and execution models of parallel applications, hence, they explicitly support the analysis of parallel applications. *VampirTrace* records execution behavior of HPC applications in the *Open Trace Format*[50]. Popular post-mortem performance analysis tools that analyze these trace files are *TAU*, *Vampir* and *Scalasca*. Those tools provide several ways to assist a developer in assessing application behavior.

An experimental tool environment which should be named is PIOviz [LKK+06a]. This environment is capable of tracing not only parallel applications, but also triggered activity on the *Parallel Virtual File System* version 2 (PVFS). In the following, brief introductions and screenshots are provided for all the mentioned tools.

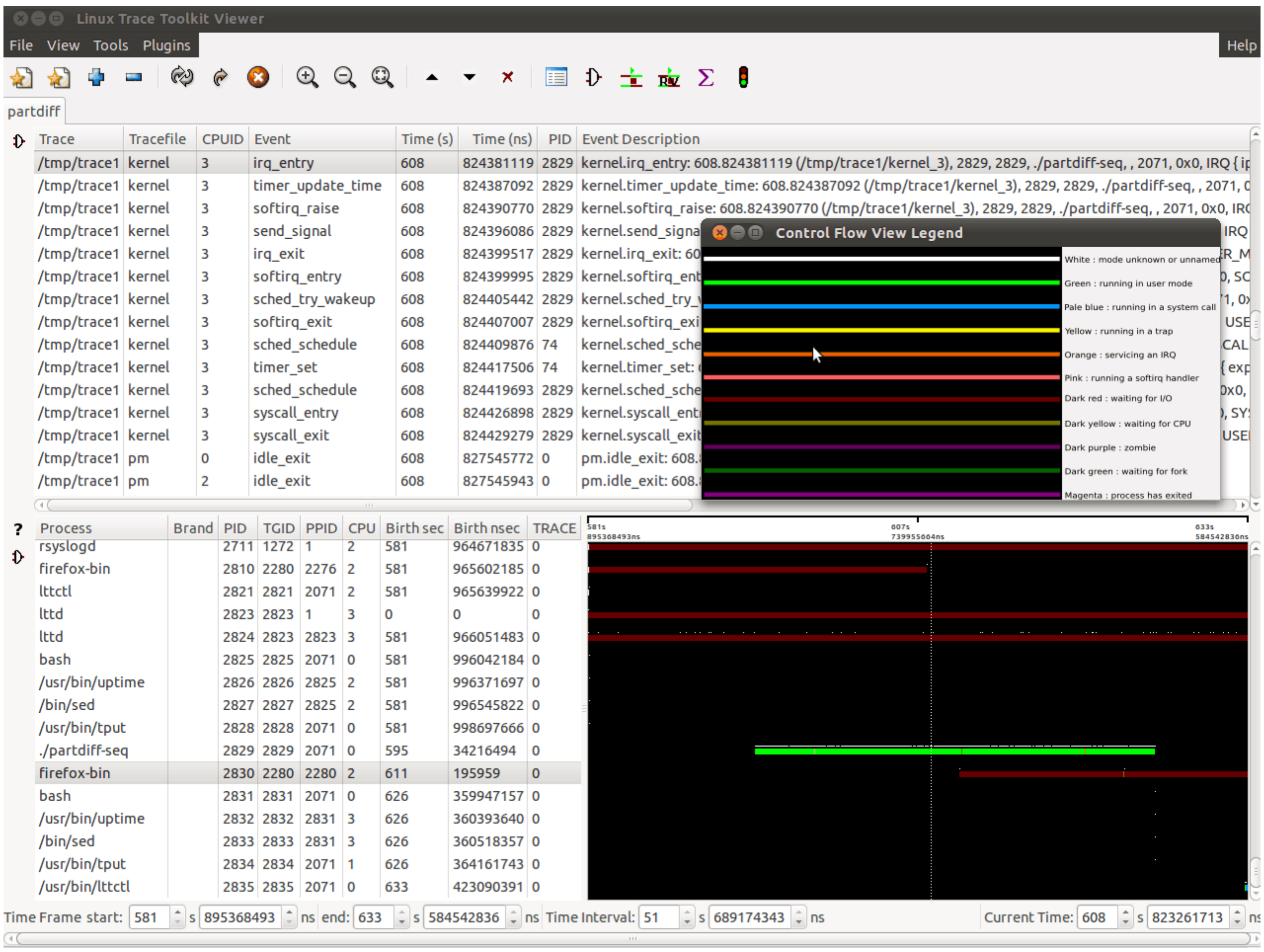
[46]http://www.efficios.com/ctf

[47]Zero-copy is a technique in which data is communicated without copying it between memory buffers – in this case no copy between kernel-space and user-space is necessary.

[48]http://www.eclipse.org/

[49]It is not possible with the tool to actually relate applications with the resource view, therefore, the execution is guessed by the author based on the information provided by the control flow view.

[50]Further information is provided on Section 2.4.5.



Trace	Tracefile	CPUID	Event	Time (s)	Time (ns)	PID	Event Description
/tmp/trace1	kernel	3	irq_entry	608	824381119	2829	kernel.irq_entry: 608.824381119 (/tmp/trace1/kernel_3), 2829, 2829, ./partdiff-seq, , 2071, 0x0, IRQ { ip
/tmp/trace1	kernel	3	timer_update_time	608	824387092	2829	kernel.timer_update_time: 608.824387092 (/tmp/trace1/kernel_3), 2829, 2829, ./partdiff-seq, , 2071, 0
/tmp/trace1	kernel	3	softirq_raise	608	824390770	2829	kernel.softirq_raise: 608.824390770 (/tmp/trace1/kernel_3), 2829, 2829, ./partdiff-seq, , 2071, 0x0, IRQ
/tmp/trace1	kernel	3	send_signal	608	824396086	2829	kernel.send_signa IRQ
/tmp/trace1	kernel	3	irq_exit	608	824399517	2829	kernel.irq_exit: 60 ER_M
/tmp/trace1	kernel	3	softirq_entry	608	824399995	2829	kernel.softirq_ent 0, SC
/tmp/trace1	kernel	3	sched_try_wakeup	608	824405442	2829	kernel.sched_try_ 71, 0x
/tmp/trace1	kernel	3	softirq_exit	608	824407007	2829	kernel.softirq_exi USE
/tmp/trace1	kernel	3	sched_schedule	608	824409876	74	kernel.sched_sche CAL
/tmp/trace1	kernel	3	timer_set	608	824417506	74	kernel.timer_set: { exp
/tmp/trace1	kernel	3	sched_schedule	608	824419693	2829	kernel.sched_sche 0x0,
/tmp/trace1	kernel	3	syscall_entry	608	824426898	2829	kernel.syscall_ent), SY
/tmp/trace1	kernel	3	syscall_exit	608	824429279	2829	kernel.syscall_exit USE
/tmp/trace1	pm	0	idle_exit	608	827545772	0	pm.idle_exit: 608.
/tmp/trace1	pm	2	idle_exit	608	827545943	0	pm.idle_exit: 608.

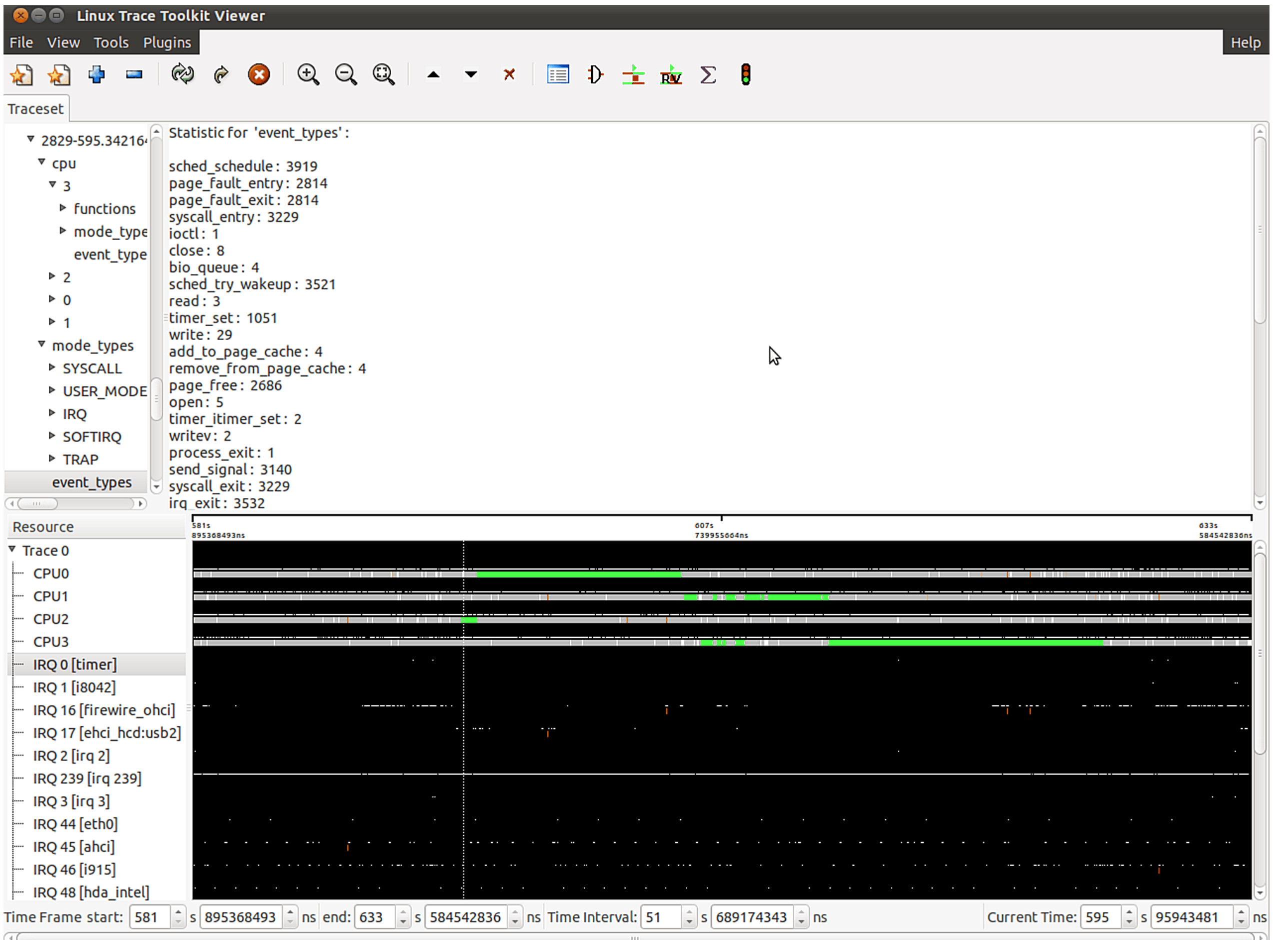
Control Flow View Legend

- White : mode unknown or unnamed
- Green : running in user mode
- Pale blue : running in a system call
- Yellow : running in a trap
- Orange : servicing an IRQ
- Pink : running a softirq handler
- Dark red : waiting for I/O
- Dark yellow : waiting for CPU
- Dark purple : zombie
- Dark green : waiting for fork
- Magenta : process has exited

Process	Brand	PID	TGID	PPID	CPU	Birth sec	Birth nsec	TRACE
rsyslogd		2711	1272	1	2	581	964671835	0
firefox-bin		2810	2280	2276	2	581	965602185	0
lttctl		2821	2821	2071	2	581	965639922	0
lttd		2823	2823	1	3	0	0	0
lttd		2824	2823	2823	3	581	966051483	0
bash		2825	2825	2071	0	581	996042184	0
/usr/bin/uptime		2826	2826	2825	2	581	996371697	0
/bin/sed		2827	2827	2825	2	581	996545822	0
/usr/bin/tput		2828	2828	2071	0	581	998697666	0
./partdiff-seq		2829	2829	2071	0	595	34216494	0
firefox-bin		2830	2280	2280	2	611	195959	0
bash		2831	2831	2071	0	626	359947157	0
/usr/bin/uptime		2832	2832	2831	3	626	360393640	0
/bin/sed		2833	2833	2831	3	626	360518357	0
/usr/bin/tput		2834	2834	2071	1	626	364161743	0
/usr/bin/lttctl		2835	2835	2071	0	633	423090391	0

Time Frame start: 581 s 895368493 ns end: 633 s 584542836 ns Time Interval: 51 s 689174343 ns Current Time: 608 s 823261713 ns

Figure 2.12.: Screenshot of the LTTV viewer for a trace of the system – event view and control flow view.



```
Statistic for 'event_types' :

sched_schedule : 3919
page_fault_entry : 2814
page_fault_exit : 2814
syscall_entry : 3229
ioctl : 1
close : 8
bio_queue : 4
sched_try_wakeup : 3521
read : 3
timer_set : 1051
write : 29
add_to_page_cache : 4
remove_from_page_cache : 4
page_free : 2686
open : 5
timer_itimer_set : 2
writev : 2
process_exit : 1
send_signal : 3140
syscall_exit : 3229
irq_exit : 3532
```

Time Frame start: 581 s 895368493 ns end: 633 s 584542836 ns Time Interval: 51 s 689174343 ns Current Time: 595 s 95943481 ns

Figure 2.13.: Screenshot of the LTTV viewer for a trace of the system – statistic view and resource view. The likely scheduled execution of the program on the CPUs is marked in green.

VampirTrace

VampirTrace [MKJ+07, KBD+08] is a library and tool set which generates traces or profiles for MPI and OpenMP applications. It instruments applications to write files in the *Open Trace Format* (OTF) at run-time. Depending on the type of instrumentation, function calls of the application, MPI and OpenMP activity are recorded. Additionally, low-level POSIX I/O calls and POSIX Threads are intercepted, CPU counters are supported. Additional counters can be supplied by plugins [STHI10].

VampirTrace utilizes *MPI Profiling Interface* (PMPI), a profiling interface provided by MPI[51]. VampirTrace stores a subset of parameters and context information from MPI calls. This enables identifying communication partners and provides information about the exchanged data, e.g., the message size. OpenMP provides an interface that eases instrumentation, the *OpenMP Pragma and Region Instrumentor* (OPARI) [MMSW02], which is used by VampirTrace.

Furthermore, it supports performance counters provided by PAPI. In case the tracing of performance counters is enabled by the user, selected counters are recorded for every generated event.

To perform source code instrumentation, the application must be recompiled using provided build scripts. These are wrappers for the compiler and include a source-to-source compiler which modifies functions in order to generate events at every function entry and exit. *Dyninst*[52] [BH00] can be used to instrument binaries, too. However, the binary instrumentation has the drawback that only limited information about application internals are accessible.

TAU

The *Tuning and Analysis Utilities* [SM06], are the swiss army knife for performance analysis. They support not only many languages (C, C++, Java, Python, CUDA, ...), but also a rich variety of capabilities like profiling and tracing, sampling, throttling of event generation, extended information for POSIX I/O and communication and hardware counter support via PAPI.

Several tools are shipped with TAU which include: converters to export data to other trace or profile formats, automatic instrumentation for source code, and a visualizer for trace profiles (*ParaProf*). *Jumpshot* [ZLGS99] or *Vampir* can be used to visualize traces. The Jumpshot viewer[53], shipped with TAU, is originally part of the *MPI Parallel Environment* (MPE) and visualizes the *SLOG2* trace format.

ParaProf is a Java GUI which displays profiling results. Compared to the tools for sequential analysis, like *gprof*, with ParaProf all processes and threads of a parallel application can be analyzed together. The tool focuses on analyzing behavior of the processes and threads for a single metric at a time. Values for the selected metric can be compared, additionally average and standard deviation is provided. The metric under investigation can be chosen – for example, time or floating point operations, and the data is provided either for the executed functions or a group of functions. Additionally, derived metrics like Flop/s can be created. To analyze time-dependent behavior, TAU provides an API that can be used in a program to divide execution into phases, which are profiled and analyzed independently.

The available views in ParaProf are histograms, a communication matrix showing the interaction between the processes, textual or a 3D visualization. The 3D visualization shows the value of a selected metric for each pair of thread and function.

Performance results of experiments can be archived, either directly in ParaProf or in a (remote) database. Metrics can be compared across experiments to monitor performance over the application development, or

[51]The MPI Profiling Interface defines that every MPI function is exported with two names, an MPI prefixed call and a PMPI prefixed version. Users use the MPI call; by providing an instrumented version of the MPI function, all user calls can be intercepted. Instrumented functions can perform appropriate logging and they usually call PMPI functions that are directly provided by the MPI library.

[52]`http://www.dyninst.org/`

[53]See Page 77 for further information on Jumpshot.

run-time behavior of various parameter sets can be assessed. *PerfExplorer* is another tool shipped with TAU, it eases statistical analysis of profiles such as clustering and correlation across experiments. It can be used to create diagrams, for example to illustrate achieved speedup. Furthermore, application performance can be assessed over multiple runs, and even a longer time period.

In order to demonstrate a few of the available views, a profile of our PDE was generated for 4 processes and visualized in ParaProf. In Figure 2.14, a few windows are shown. In the upper window, the experiment is chosen and information about run-time environment characterization is given which identifies the experimental setup and run-time settings. The legend windows (on the left) show the color-code and name of the recorded events and available groups. To improve user interaction, the events and function calls are grouped into sets – in the example, the MPI group can be selected to assess communication overhead quickly. In TAU, the call-graph is just encoded into the event names. Thus, the number of events in the function legend is rather large; in the example, calculate invokes several functions.

The main data window (in the center) with the unstacked bars in the middle of the figure displays a metric for all functions and processes. In the screenshot the *time* metric has been chosen and the mouse hovers over the calculate function. By clicking an interesting function, a window pops up that just shows this metric for all threads. In the figure, the window below shows the time for `MPI_Sendrecv()` which is called by the exchange function.

Interaction between the 4 processes can be analyzed with a communication matrix, in Figure 2.15 the number of calls is encoded in a heat-map. In the figure, a diagonal communication pattern of the PDE can be observed – a process communicates with its two neighbors. A few communications from the ranks to Rank 0 are seen, this is the finalization step in which the results are gathered.

Text statistics of Rank 0 for I/O and messages are provided in Figure 2.16, showing the number of I/O calls and amount of data.

Note that the standard automatic instrumentation, which intercepts every function entry and exit, causes serious overhead; by tracing, the wall-clock run-time of the PDE increases from 13 s to 576 s. In the same run, more than 1.8 GiB of trace data will be generated per process, if tracing is enabled. Consequently, this demonstrates the need to filter unimportant events during the instrumentation or at run-time. TAU provides tools to automatically filter events, and it throttles an event when it is fired too often.

For more information about visualization and usage refer to the *TAU User Guide* [Tau10].

Vampir

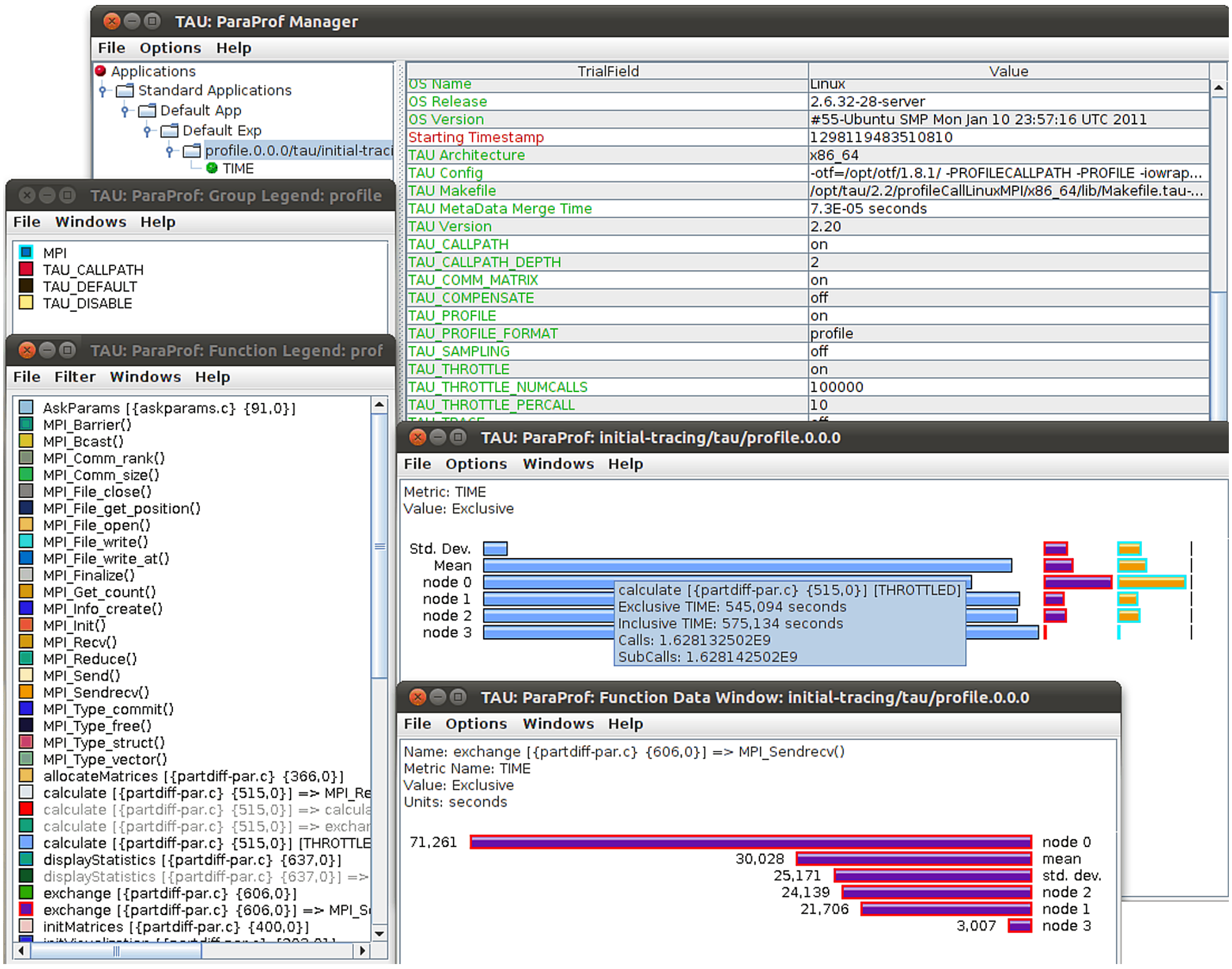
Vampir [MKJ+07, KBD+08] is a mature commercial trace analysis tool, which visualizes files that have been generated with VampirTrace. However, it has a proprietary code base and cannot be extended by third parties.

Large trace files may not fit into the memory of one machine and take a long time to visualize and preprocess, therefore, the software called *VampirServer* extends the performance of Vampir by outsourcing analysis capabilities and trace handling to a set of remote processes. The number of server processes can be scaled to match the size of the trace file and the required performance.

In Vampir, multiple displays, each performing a specific visualization, can be arranged to a *workspace*. Displays can be configured, e.g., to show a particular process, to filter information or to visualize another metric. Zooming on a timeline is propagated to all displays.

For a PDE run, a workspace containing all displays has been created and is shown in Figure 2.17. Here, only a fraction of time is actually traced as the overhead with the default instrumentation that traces invocations of all functions is overwhelming[54].

[54]The PDE for the trace runs 100 iterations instead of the 10.000 that have been executed with an instrumented program for the other tools.



TAU: ParaProf Manager

File Options Help

- Applications
  - Standard Applications
    - Default App
      - Default Exp
        - profile.0.0.0/tau/initial-traci
          - TIME

TrialField	Value
OS Name	Linux
OS Release	2.6.32-28-server
OS Version	#55-Ubuntu SMP Mon Jan 10 23:57:16 UTC 2011
Starting Timestamp	1298119483510810
TAU Architecture	x86_64
TAU Config	-otf=/opt/otf/1.8.1/ -PROFILECALLPATH -PROFILE -iowrap...
TAU Makefile	/opt/tau/2.2/profileCallLinuxMPI/x86_64/lib/Makefile.tau-...
TAU MetaData Merge Time	7.3E-05 seconds
TAU Version	2.20
TAU_CALLPATH	on
TAU_CALLPATH_DEPTH	2
TAU_COMM_MATRIX	on
TAU_COMPENSATE	off
TAU_PROFILE	on
TAU_PROFILE_FORMAT	profile
TAU_SAMPLING	off
TAU_THROTTLE	on
TAU_THROTTLE_NUMCALLS	100000
TAU_THROTTLE_PERCALL	10

TAU: ParaProf: Group Legend: profile

File Windows Help

- MPI
- TAU_CALLPATH
- TAU_DEFAULT
- TAU_DISABLE

TAU: ParaProf: Function Legend: prof

File Filter Windows Help

- AskParams [{askparams.c} {91,0}]
- MPI_Barrier()
- MPI_Bcast()
- MPI_Comm_rank()
- MPI_Comm_size()
- MPI_File_close()
- MPI_File_get_position()
- MPI_File_open()
- MPI_File_write()
- MPI_File_write_at()
- MPI_Finalize()
- MPI_Get_count()
- MPI_Info_create()
- MPI_Init()
- MPI_Recv()
- MPI_Reduce()
- MPI_Send()
- MPI_Sendrecv()
- MPI_Type_commit()
- MPI_Type_free()
- MPI_Type_struct()
- MPI_Type_vector()
- allocateMatrices [{partdiff-par.c} {366,0}]
- calculate [{partdiff-par.c} {515,0}] => MPI_Re
- calculate [{partdiff-par.c} {515,0}] => calcula
- calculate [{partdiff-par.c} {515,0}] => exchar
- calculate [{partdiff-par.c} {515,0}] [THROTTLE
- displayStatistics [{partdiff-par.c} {637,0}]
- displayStatistics [{partdiff-par.c} {637,0}] =>
- exchange [{partdiff-par.c} {606,0}]
- exchange [{partdiff-par.c} {606,0}] => MPI_S
- initMatrices [{partdiff-par.c} {400,0}]

TAU: ParaProf: initial-tracing/tau/profile.0.0.0

File Options Windows Help

Metric: TIME
Value: Exclusive

Std. Dev.
Mean
node 0
node 1
node 2
node 3

calculate [{partdiff-par.c} {515,0}] [THROTTLED]
Exclusive TIME: 545,094 seconds
Inclusive TIME: 575,134 seconds
Calls: 1.628132502E9
SubCalls: 1.628142502E9

TAU: ParaProf: Function Data Window: initial-tracing/tau/profile.0.0.0

File Options Windows Help

Name: exchange [{partdiff-par.c} {606,0}] => MPI_Sendrecv()
Metric Name: TIME
Value: Exclusive
Units: seconds

Value	
71,261	node 0
30,028	mean
25,171	std. dev.
24,139	node 2
21,706	node 1
3,007	node 3

Figure 2.14.: Screenshot of ParaProf – PDE profile including experimental information.

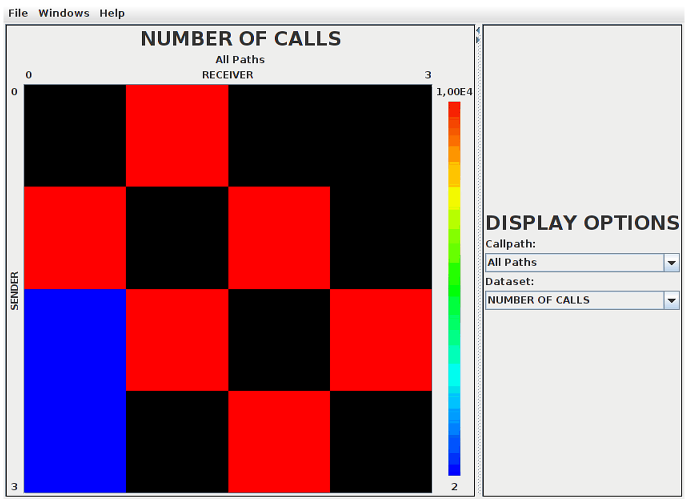
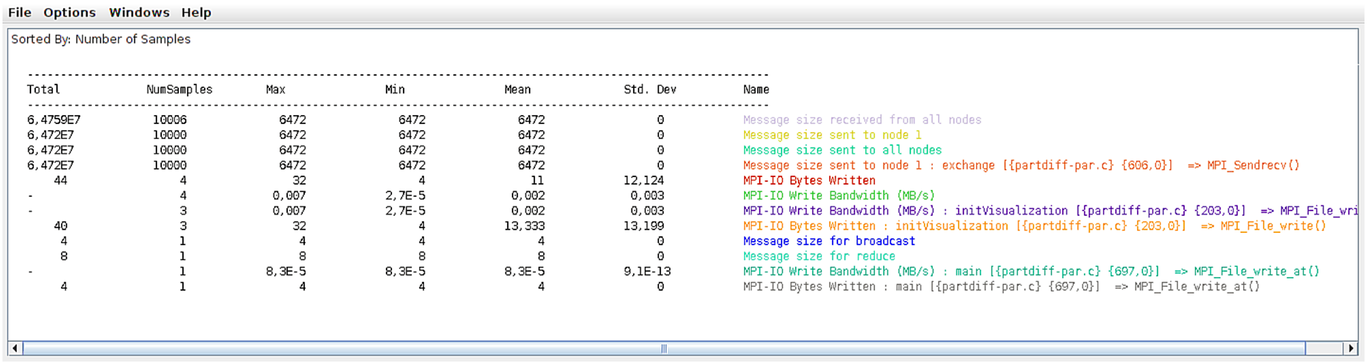


Figure 2.15.: Screenshot of ParaProf – PDE communication matrix.



Sorted By: Number of Samples

```
---------------------------------------------------------------------------------------------------------
Total          NumSamples      Max          Min          Mean         Std. Dev      Name
---------------------------------------------------------------------------------------------------------
6,4759E7        10006         6472         6472         6472            0         Message size received from all nodes
6,472E7         10000         6472         6472         6472            0         Message size sent to node 1
6,472E7         10000         6472         6472         6472            0         Message size sent to all nodes
6,472E7         10000         6472         6472         6472            0         Message size sent to node 1 : exchange [{partdiff-par.c} {606,0}]   => MPI_Sendrecv()
     44             4           32            4           11       12,124         MPI-IO Bytes Written
 -                  4        0,007       2,7E-5        0,002        0,003         MPI-IO Write Bandwidth (MB/s)
 -                  3        0,007       2,7E-5        0,002        0,003         MPI-IO Write Bandwidth (MB/s) : initVisualization [{partdiff-par.c} {203,0}]   => MPI_File_wri
     40             3           32            4       13,333       13,199         MPI-IO Bytes Written : initVisualization [{partdiff-par.c} {203,0}]   => MPI_File_write()
      4             1            4            4            4            0         Message size for broadcast
      8             1            8            8            8            0         Message size for reduce
 -                  1       8,3E-5       8,3E-5       8,3E-5      9,1E-13         MPI-IO Write Bandwidth (MB/s) : main [{partdiff-par.c} {697,0}]   => MPI_File_write_at()
      4             1            4            4            4            0         MPI-IO Bytes Written : main [{partdiff-par.c} {697,0}]   => MPI_File_write_at()
```

Figure 2.16.: Screenshot of ParaProf – *user event statistics* for Process 0 including MPI-IO statistics.

As Vampir has a huge number of displays, they are enumerated in the screenshot and described individually:

1. Next to the symbol toolbar is an overview timeline, which is visible all the time and does not change with zooms. This timeline shows the observed activity of all processes over time in a condensed form. The height of color in the bar encodes the number of processes that execute an operations of a given kind at a given time. Functions are grouped during the tracing and represented by different colors, example groups are I/O, communication or application. By default, MPI functions are colored in red whereas time spent in the application is colored in green. If all processes perform operations of the same group, then a column contains one color, for example at the beginning all processes perform MPI calls. To add a certain display, a user can click on the specific icon in the toolbar.

2. The *Master Timeline* shows the activity for each individual thread in colors according to the group. Inter-process messages are visualized by black lines. In Vampir there is no concept which associates the processes to the hardware topology. Therefore, the user must know the mapping of the processes to hardware.

3. In the *Process Timeline*, the call-stack of an individual process is given (here Process 0).

4. A *Counter Timeline* shows the values of one PAPI counter for one process. The minimum, maximum and average values can be plotted into one graph. In this display, the total number of instructions which were performed is plotted. The observed spike during startup might be an artifact caused by an overflow of the counter.

5. This is also a counter timeline that visualizes the Flop/s for Process 0. Note that there could be as many replicates of the displays as fit on the screen.

6. For one selectable metric, the *Performance Radar* encodes the counter values in a given color, similar to a heat map. In contrast to the counter timeline, all processes can be visualized together. Unfortunately, due to the counter overflow in the example, all processes are drawn in blue.

7. The *Call Tree* shows the call-graph together with a profile for the visualized time interval.

8. In the *Context View*, more information of the selected object is provided – most visible entities, such as an event, function or process, can be selected. In the example, the whole function group is clicked.

9. The *Function Summary* provides information on a set of processes. In the left window, the exclusive time of all processes is accumulated while the right window prints the number of invocations per group.

10. Available groups and their color scheme are listed in the *Function Legend*. By instrumenting the application manually with VampirTrace, more groups can be created.

11. An overview of the number of messages or message sizes is given in the *Message Summary*.

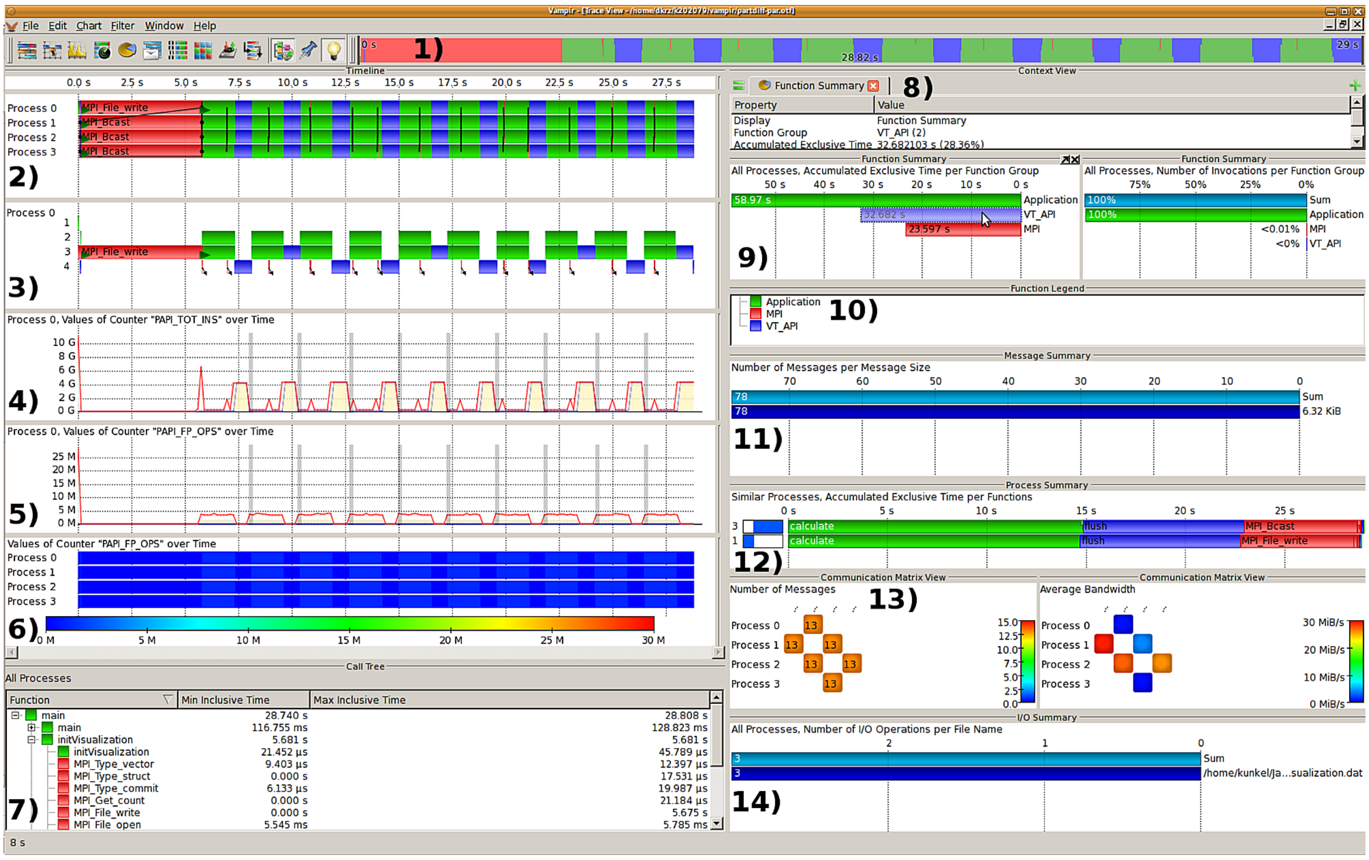


Figure 2.17.: Screenshot of a Vampir workspace.

12. In the *Process Summary*, a profile is generated and visualized for every process. If the space does not suffice, the display automatically clusters similar processes. In the example, it detected three processes with the upper function profile and one with the lower function profile – the master process which performs the actual I/O.

13. Similar to TAU, the inter-process behavior is visualized in the *Communication Matrix View*. Two displays have been created, the left window just shows the number of messages exchanged between two processes, while the right window indicates the average bandwidth.

14. The *I/O Summary* finishes our tour through the available displays: Several metrics are available and related to the file name: accessed data volume, number of operations or bandwidth.

Scalasca

Scalasca [GWW+10], is a performance analysis toolset which automatically analyzes large-scale parallel applications that use the MPI, OpenMP or a hybrid MPI/OpenMP programming model. It has been successfully applied to applications running with 200.000 processes on a BlueGene/P system [WBM+10].

Scalasca can be run in two modes, either a summary of the parallel program is created at run-time, or the application activity can be recorded in the *EPILOG* trace format and then analyzed post-mortem.

In the trace mode, Scalasca searches automatically for a common class of run-time communication bottlenecks, for example, for late senders. Scalasca ships with the sequential analyzer `expert` [WM03] and the parallel analyzer `scout` that identify wait-state patterns. The parallel analyzer runs with the same number of processes as the original application. While the analysis is performed, it replays the communication pattern of the original program and updates statistics accordingly. Therefore, `scout` scales similarly to the original application. However, the sequential tool `expert` detects more inefficient communication patterns.

Scalasca can instrument the application automatically, either by using compiler options, by transformation of the source code or by linking the program with an already instrumented library. Similarly to the other tools, an API for manual instrumentation is provided.

In contrast to previously referenced tracing tools, with Scalasca the application is typically started with an additional software monitoring system. After the application terminates the system can automatically run `scout` to perform the parallel trace analysis.

Process statistics and identified bottlenecks are displayed in the *Qt* application *Cube3*, which allows browsing through the analysis results.

To assess the features, the PDE configuration with 4 processes is instrumented with Scalasca and instructed to generate summaries.

A screenshot of the summary is provided in Figure 2.18. The view is split into three columns: the first column shows the available metrics, the second column displays all functions in the call-graph and the contribution to the metrics, the last column shows the contribution of every process of the supercomputer to the value of the function (and metrics). In the analysis session, the user selects a metric on the left, then localizes the relevant function and, at last, analyzes the distribution of the selected metrics among the processes. In the given example, the number of send operations is selected in the metrics tree; sends are invoked in the call-graph by `exchange()` which is called by `calculate()` which in turn is called by `main()`.

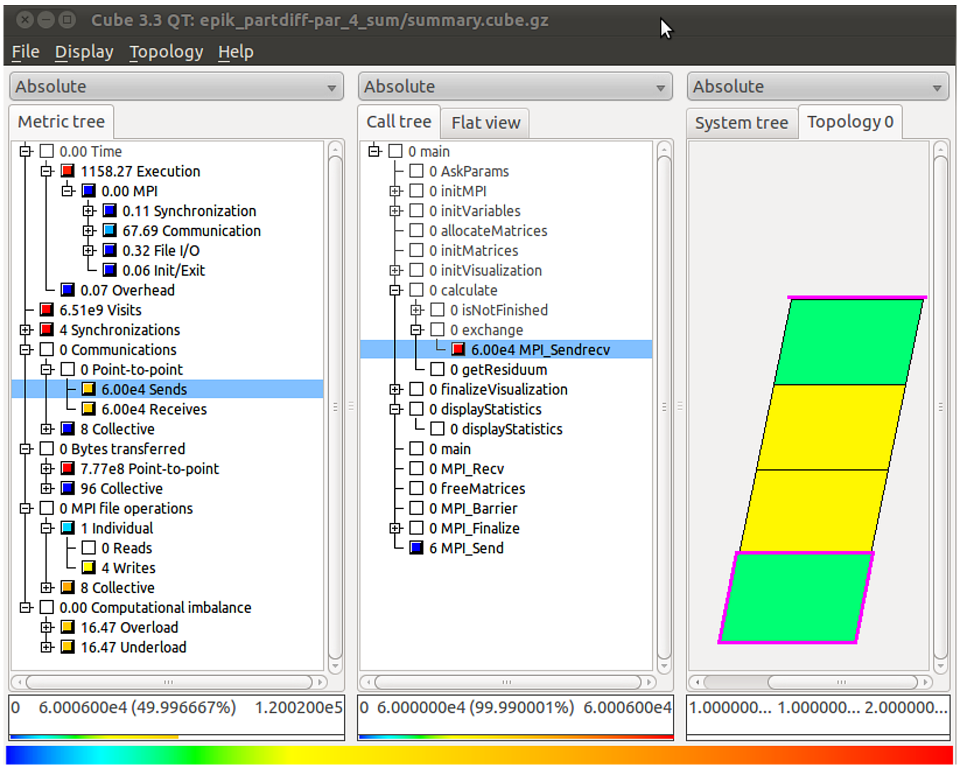
The view on the left aggregates the metrics among all functions and processes. Also, note that in the hierarchical view, a collapsed node of a column aggregates the values of all children. That means each node shows the inclusive metric, while an expanded node displays the *exclusive metric*, i.e., the contribution to the metrics which is not caused by any of the child nodes. Child nodes show their share by themselves (compare the *Execution* node and children in the left column). A color scheme between blue and red encodes the values similar to a heat map in all three columns. This assists in spotting bottlenecks in the metrics, code and unbalanced processes on the hardware.

For figure Figure 2.18, 60.000 `MPI_Sendrecv()` functions have been invoked from the `exchange()` function. As the inner processes transfer twice as much messages, the four processes call the function 10.000 times, 20.000 times, 20.000 and 10.000 times, respectively. The right view visualizes either the topology of the machine – here a flat topology – or the numerical values as shown in Figure 2.19.

By instrumenting all user functions, the wall-clock time increased from 12 s to 307 s, by instrumenting just MPI with the provided PMPI library, the overhead is not measurable. With Cube, the bottleneck can be identified by looking at the (run-)time metric of the functions and processes in Figure 2.19. A total of 1158 s for all 4 processes is divided into 68 s MPI activity and the remaining time is spent for user activity. In the call-graph, 558 s is spent in `getResiduum()` and another 600 s in the calculate function itself; the right column shows that the load is almost balanced across all processes. As `getResiduum()` has a tiny function body, the overhead of the measurement system dominates run-time. In a real scenario, this function should not be instrumented and therefore would be filtered; Scalasca provides tools to filter events.

For the existing metrics, a short description is provided in the online help that assists the user in understanding them. For example, a screenshot of the metrics for computational imbalance between processes and the corresponding help is given in Figure 2.20.

Periscope [GO10] and PerfExpert [BKD+10] are other automatic tools. They scan performance properties at run-time; appropriate metrics are measured and evaluated automatically. Ultimately, as in Scalasca, this assists in automatic localization of certain types of bottlenecks. However, while all those automatic tools provide some hints, the user might be forced to use a visualization tool such as Vampir to really understand the behavior of the application.

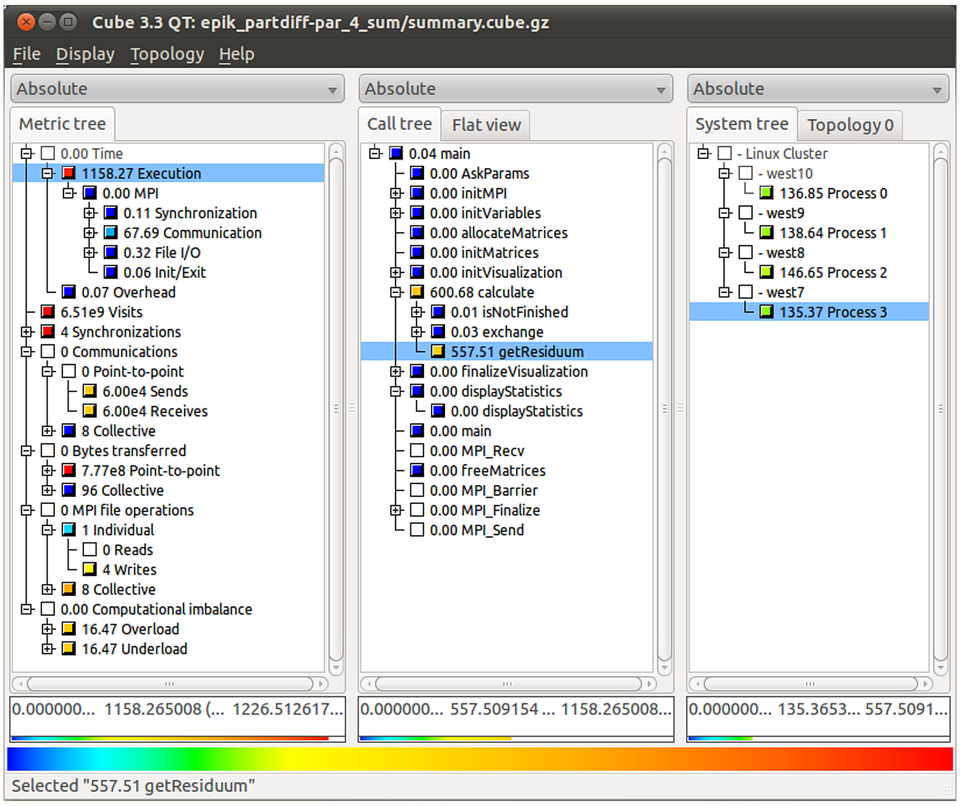

Cube 3.3 QT: epik_partdiff-par_4_sum/summary.cube.gz
File Display Topology Help
Absolute
Absolute
Absolute
Metric tree
Call tree
Flat view
System tree
Topology 0

```
0.00 Time
  1158.27 Execution
    0.00 MPI
      0.11 Synchronization
      67.69 Communication
      0.32 File I/O
      0.06 Init/Exit
    0.07 Overhead
6.51e9 Visits
4 Synchronizations
0 Communications
  0 Point-to-point
    6.00e4 Sends
    6.00e4 Receives
  8 Collective
0 Bytes transferred
  7.77e8 Point-to-point
  96 Collective
0 MPI file operations
  1 Individual
    0 Reads
    4 Writes
  8 Collective
0.00 Computational imbalance
  16.47 Overload
  16.47 Underload
```

```
0 main
  0 AskParams
  0 initMPI
  0 initVariables
  0 allocateMatrices
  0 initMatrices
  0 initVisualization
  0 calculate
    0 isNotFinished
    0 exchange
      6.00e4 MPI_Sendrecv
    0 getResiduum
  0 finalizeVisualization
  0 displayStatistics
    0 displayStatistics
  0 main
  0 MPI_Recv
  0 freeMatrices
  0 MPI_Barrier
  0 MPI_Finalize
  6 MPI_Send
```

```
0    6.000600e4 (49.996667%)    1.200200e5
0    6.000000e4 (99.990001%)    6.000600e4
1.000000...  1.000000...  2.000000...
```

Figure 2.18.: Scalasca's Cube3 browser – the left column shows available metrics, the middle column assigns the metric's values to functions of the call-graph, the right column shows the contribution of every process to the function.


Cube 3.3 QT: epik_partdiff-par_4_sum/summary.cube.gz
File Display Topology Help
Absolute
Absolute
Absolute
Metric tree
Call tree
Flat view
System tree
Topology 0

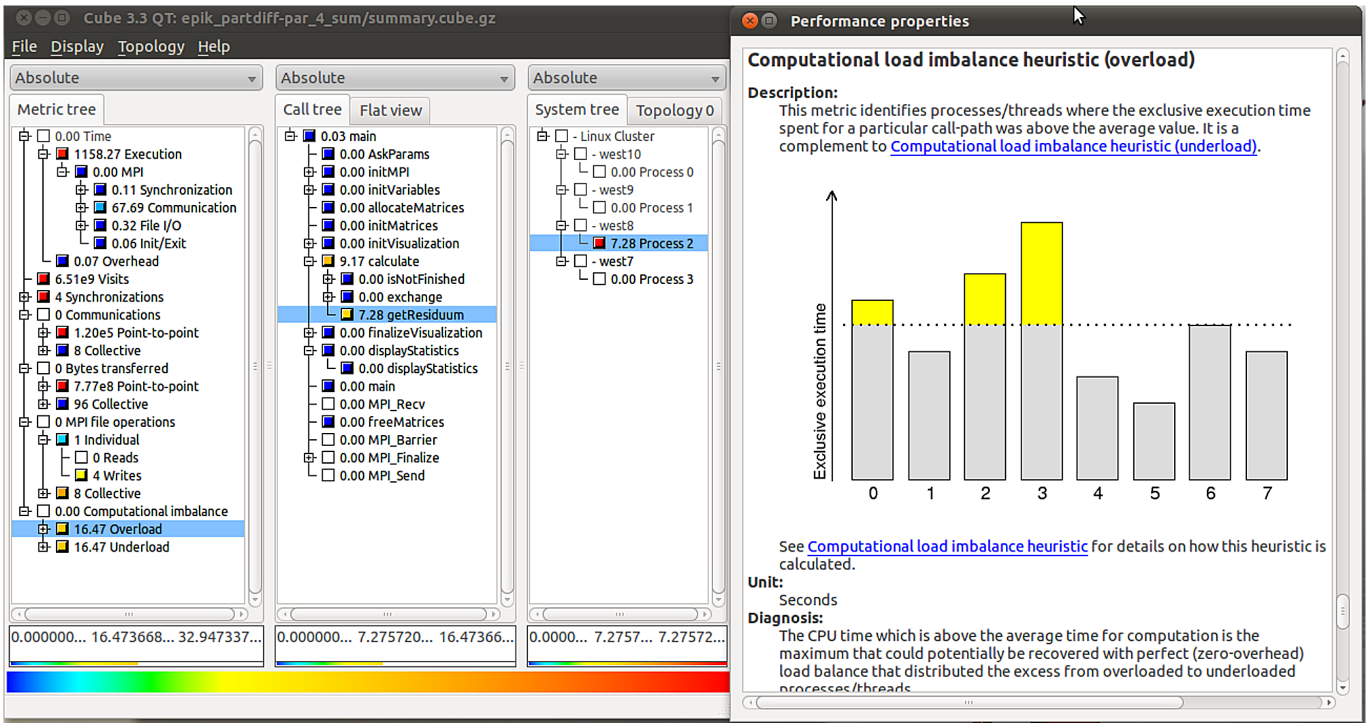
```
0.00 Time
  1158.27 Execution
    0.00 MPI
      0.11 Synchronization
      67.69 Communication
      0.32 File I/O
      0.06 Init/Exit
    0.07 Overhead
6.51e9 Visits
4 Synchronizations
0 Communications
  0 Point-to-point
    6.00e4 Sends
    6.00e4 Receives
  8 Collective
0 Bytes transferred
  7.77e8 Point-to-point
  96 Collective
0 MPI file operations
  1 Individual
    0 Reads
    4 Writes
  8 Collective
0.00 Computational imbalance
  16.47 Overload
  16.47 Underload
```

```
0.04 main
  0.00 AskParams
  0.00 initMPI
  0.00 initVariables
  0.00 allocateMatrices
  0.00 initMatrices
  0.00 initVisualization
  600.68 calculate
    0.01 isNotFinished
    0.03 exchange
    557.51 getResiduum
  0.00 finalizeVisualization
  0.00 displayStatistics
    0.00 displayStatistics
  0.00 main
  0.00 MPI_Recv
  0.00 freeMatrices
  0.00 MPI_Barrier
  0.00 MPI_Finalize
  0.00 MPI_Send
```

```
- Linux Cluster
  - west10
    136.85 Process 0
  - west9
    138.64 Process 1
  - west8
    146.65 Process 2
  - west7
    135.37 Process 3
```

```
0.000000...  1158.265008 (...  1226.512617...
0.000000...  557.509154 ...  1158.265008...
0.000000...  135.3653...  557.5091...
Selected "557.51 getResiduum"
```

Figure 2.19.: Scalasca's Cube3 browser – identifying the computational overhead in `getResiduum()`.



```
Cube 3.3 QT: epik_partdiff-par_4_sum/summary.cube.gz
File  Display  Topology  Help

Absolute                          Absolute                        Absolute
Metric tree                       Call tree | Flat view           System tree | Topology 0

0.00 Time                         0.03 main                       - Linux Cluster
  1158.27 Execution                 0.00 AskParams                  - west10
    0.00 MPI                        0.00 initMPI                      0.00 Process 0
      0.11 Synchronization          0.00 initVariables              - west9
      67.69 Communication           0.00 allocateMatrices             0.00 Process 1
      0.32 File I/O                 0.00 initMatrices               - west8
      0.06 Init/Exit                0.00 initVisualization            7.28 Process 2
  0.07 Overhead                     9.17 calculate                  - west7
6.51e9 Visits                         0.00 isNotFinished              0.00 Process 3
4 Synchronizations                    0.00 exchange
0 Communications                      7.28 getResiduum
  1.20e5 Point-to-point             0.00 finalizeVisualization
  8 Collective                      0.00 displayStatistics
0 Bytes transferred                   0.00 displayStatistics
  7.77e8 Point-to-point             0.00 main
  96 Collective                     0.00 MPI_Recv
0 MPI file operations               0.00 freeMatrices
  1 Individual                      0.00 MPI_Barrier
    0 Reads                         0.00 MPI_Finalize
    4 Writes                        0.00 MPI_Send
  8 Collective
0.00 Computational imbalance
  16.47 Overload
  16.47 Underload

0.000000... 16.473668... 32.947337...   0.000000... 7.275720... 16.47366...   0.0000... 7.2757... 7.27572...
```

Performance properties

Computational load imbalance heuristic (overload)

Description:
This metric identifies processes/threads where the exclusive execution time spent for a particular call-path was above the average value. It is a complement to Computational load imbalance heuristic (underload).

See Computational load imbalance heuristic for details on how this heuristic is calculated.
Unit:
Seconds
Diagnosis:
The CPU time which is above the average time for computation is the maximum that could potentially be recovered with perfect (zero-overhead) load balance that distributed the excess from overloaded to underloaded processes/threads.

Figure 2.20.: Scalasca's Cube3 – assessing load imbalance and the online help for this metric.

MPE

The *MPI Parallel Environment* (MPE) is a loosely structured library of routines designed to support the parallel programmer in an MPI environment. It includes performance analysis tools for MPI programs, profiling libraries, graphical visualization tools and the trace visualization tool *Jumpshot* [ZLGS99]. MPE is shipped with MPICH-2 and contains different wrapper libraries, which use the PMPI profiling interface of MPI to replace the MPI calls with new functions.

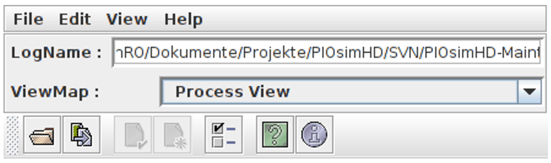
The trace visualizer of MPE is *Jumpshot*, which reads files in the SLOG2 format. SLOG2 tries to be scalable by storing aggregated information about intervals directly inside the format – further information is provided in [ZLGS99].

Upon startup of Jumpshot, the main window is shown in which a trace file can be loaded, a screenshot is given in Figure 2.21a. Jumpshot distinguishes between three types of entries: an *event*, a *state* that has a well defined start and end, and arrows which mark causal relations between states. A trace entry belongs to one named *category*. All available categories, assigned colors and whether they shall be visualized or searchable, are listed in the *legend window* (Figure 2.21b).

The *timeline window* shows the activity of each process over time. Processes are enumerated in a tree view and mapped to the timelines according to a *ViewMap*. A screenshot of the timeline window is given in Figure 2.21c. On the left side, the tree view shows two processes, the activity is drawn in the center. A horizontal timeline renders the activity of one processor in one row, the activity is encoded with colors as defined in the legend window. White areas in the activity correspond to computation by the client processes, the colors show the MPI function; violet, for example, represents `MPI_Reduce()`.

A user can select an interval and open a *profile window*, which aggregates the time of the states over each category and process. An example is provided in Figure 2.21d; the violet color indicates that most time is spent in `MPI_Reduce()`.

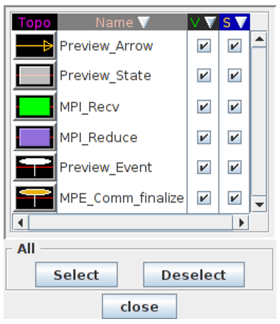
Both windows offer functionality to zoom and scroll in the window, this is provided by the icons in the toolbar. Timelines can be enlarged with the slider on the right of the windows. Additionally, individual timelines can be deleted or moved around; to move a timeline, it must be cut by the user and inserted after



LogName : ıRO/Dokumente/Projekte/PIOsimHD/SVN/PIOsimHD-Maint

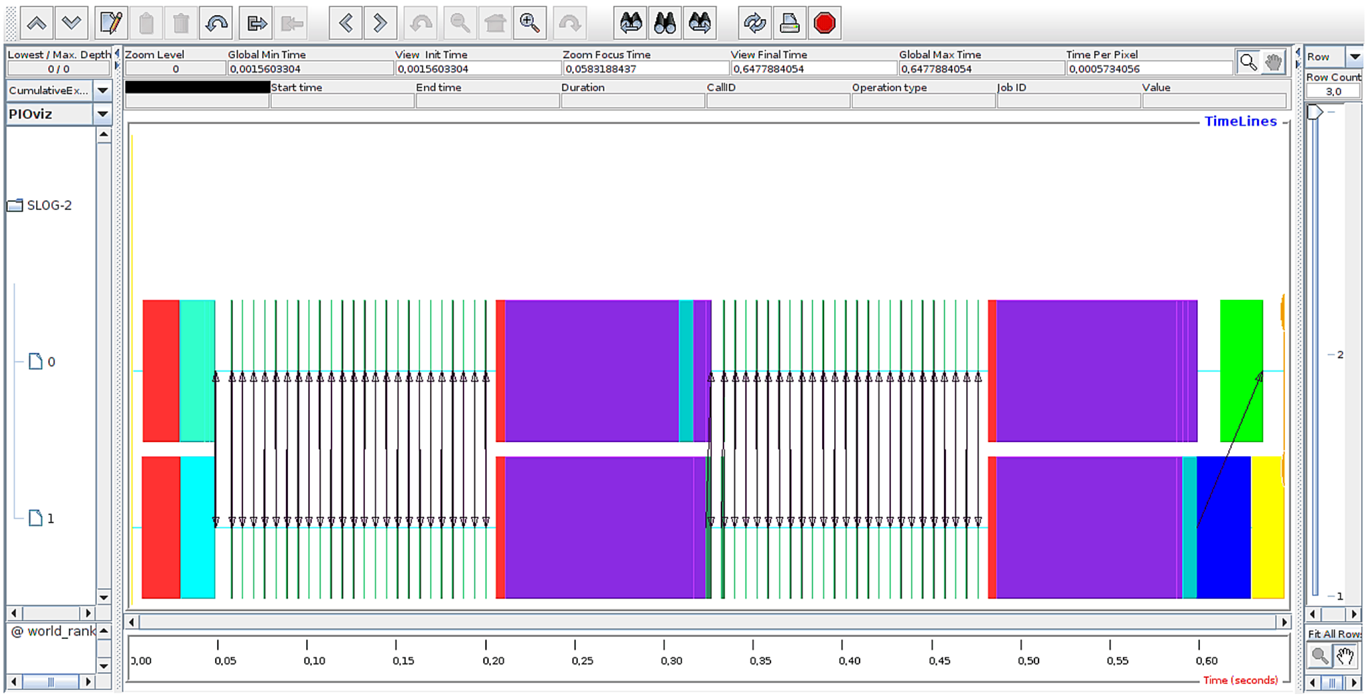
ViewMap : Process View

(a) Main window



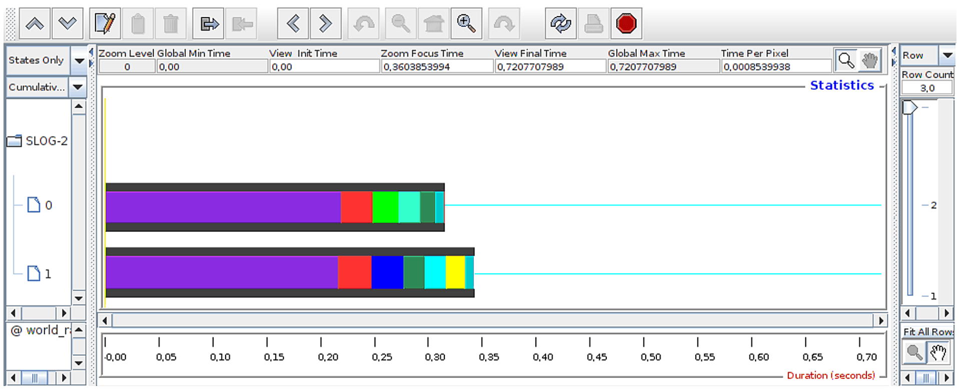
Topo	Name	V	S
	Preview_Arrow	✓	✓
	Preview_State	✓	✓
🟩	MPI_Recv	✓	✓
🟪	MPI_Reduce	✓	✓
	Preview_Event	✓	✓
	MPE_Comm_finalize	✓	✓

(b) Legend window



Lowest / Max. Depth 0 / 0
Zoom Level 0
Global Min Time 0,0015603304
View Init Time 0,0015603304
Zoom Focus Time 0,0583188437
View Final Time 0,6477884054
Global Max Time 0,6477884054
Time Per Pixel 0,0005734056
Row Count 3,0

(c) Timeline window



Zoom Level 0
Global Min Time 0,00
View Init Time 0,00
Zoom Focus Time 0,3603853994
View Final Time 0,7207707989
Global Max Time 0,7207707989
Time Per Pixel 0,0008539938
Row Count 3,0

(d) Profile window

Figure 2.21.: Jumpshot windows.

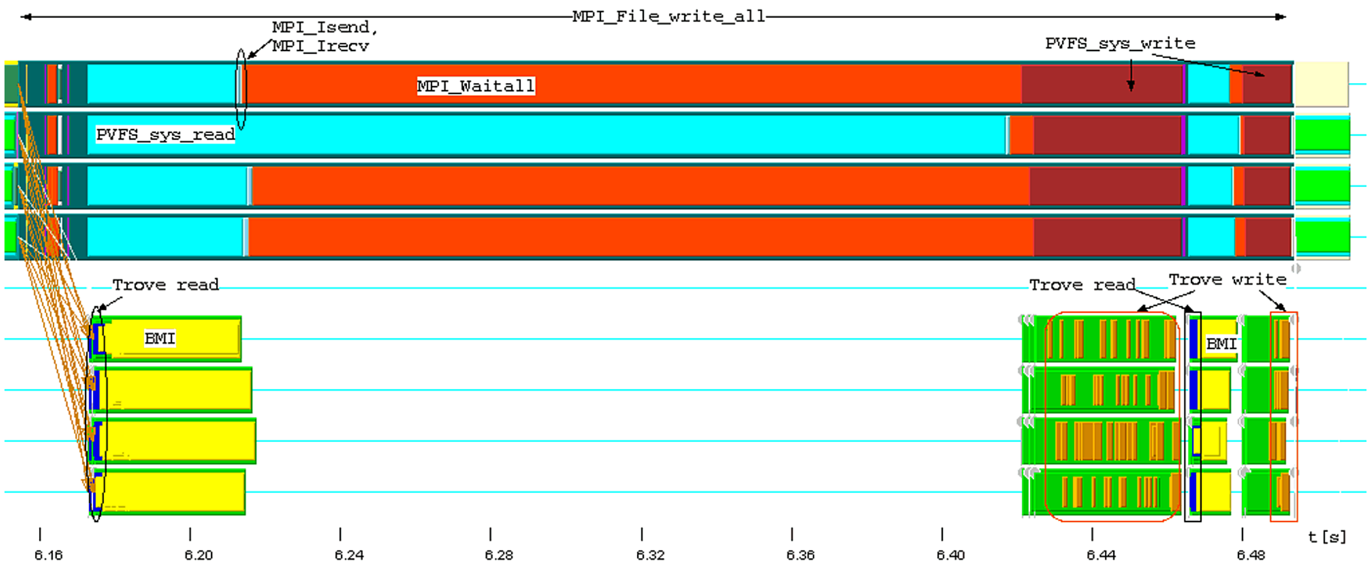


Figure 2.22.: Modified screenshot of PIOviz visualizing the interaction between 4 clients and 4 servers with explanations [KTML09].

another timeline.

PIOviz

The *Parallel Input/Output Visualization Environment* [LKK+06b, KTML09] (PIOviz) is able to trace and visualize activities on the servers of the parallel file system PVFS in conjunction with the client events triggering these activities. PIOviz correlates the behavior of the servers with program events. Developers can use these features to analyze and optimize MPI-IO applications together with PVFS. Additionally, PIOviz also collects device statistics, such as network and disk utilization, from the operating system, and computes a few PVFS-internal statistics [KL08].

The PMPI wrapper provided in MPE is used to trace the MPI function calls. However, with the original MPE only MPI activity is recorded and analyzed. PIOviz extends its capabilities by recording communication inside MPI calls, client-sided PVFS activity, and corresponding operations in PVFS servers. Compared to other tools, PIOviz is considered experimental, however, it provides novel capabilities that are not available elsewhere.

In brief, the environment consists of a set of user-space tools, modifications to the I/O part of MPICH2 and MPE, and logging enhancements to PVFS. To relate client and server activity, the following changes are made: PIOviz modifies the MPE logging wrapper to add a call-ID to each I/O request. Patches to MPI-IO and PVFS transfer this ID to the server and through the different layers, and record interesting information. The call-ID allows us to associate the MPI call with the PVFS operations triggered by this call. Also, the environment introduces additional user-space tools that transform SLOG2 files depending on this extra information. It contains modifications to Jumpshot providing additional information in the viewer.

To assess performance in the workflow, independent trace files are created on client-side and server-side once a user executes an MPI(-IO) program. Then a set of tools post-process these files and finally merges them into a single file containing all enriched information about client and server activities. PIOviz uses MPICH's SLOG2 format, therefore a user can analyze trace information with Jumpshot.

An example screenshot of PIOviz is given in Figure 2.22. Client process activity is given in the upper four timelines, and the lower five timelines show the activity for one PVFS metadata server and four data servers.

By looking at the screenshot it can be observed that the second process spends about 20 ms more time in the `PVFS_sys_read()` operation of the `MPI_File_write_all()` and thus the process waits for the read operation to complete. However, the server activity finished already, therefore, the servers are waiting for requests from clients and are not the cause of the inefficiency. Instead we claim the client library caused the observed behavior. Without knowledge of the server activity, a hypothesis for a potential bottleneck could not exclude the network, the server, or the client-server protocol.

Besides PIOviz, to our knowledge, there is no trace environment available which can gather information of client and server activity and correlates them – a recent funded project aims to extend TAU towards this goal [BCI+10], though.

2.4.5. Trace Formats

There are several trace formats available because many performance analysis tools rely on their own trace format. Usually, command line tools are provided to convert the tool-specific trace format to other well-known trace formats. Therefore, already existing tools can be applied to process the (converted) trace. Scalasca and TAU, for example, provide converters into the *Open Trace Format* (OTF).

A few general aspects in designing a trace format and its interface are discussed at first, the list is a lose collection of aspects and does not aim for completeness. Then, the concepts behind OTF are introduced.

Basically, all trace formats have been created with specific design goals in mind, however, several goals are common to most of them. The following abstract requirements and concepts represent the author's view; they are defined after looking at several trace formats and existing tools[55]:

- Recording of all *relevant information* for post-mortem analysis must be possible. This includes the possibility to record arbitrary data that is necessary to characterize an event in detail. A *context* provides more information about the recorded events or the utilized resources. For example, timestamps are required to understand the temporal causality; an identifier can specify the processor a thread is (currently) executed on. Timestamps are especially difficult: Since local clocks are not as accurate as a primary reference clock, timestamps of different components are slightly incorrect thus sorting events by the locally created timestamp can lead to a wrong order of events. Therefore, either all clocks must be synchronized accurately with one reference clock, or mechanisms are provided that are able to fix incorrect ordering. *Metadata* describes the invariant properties of the environment in which the trace is recorded. This kind of description of system configuration and experiment is important when traces are kept for a long time.
- In heterogeneous environments *portability* between machine architectures becomes relevant.
- Methods to *reduce the trace file size* and to handle large traces should be available. One simple approach that reduces the file size is permit selective activation and deactivation of the run-time tracing, that means either the application activates tracing for the relevant area or the system deactivates itself automatically after a threshold is reached. Another method is to support compression. If that does not help to reduce the file size, then tools should be capable to process large trace files.
- Required post-processing of the trace files should be of *low overhead*. For example, the PIOviz environment relies on several post-processing stages in which the trace data is read completely. For larger traces this is infeasible.
- Analysis of the trace files should require *limited resources*. A subset of the data should be loadable. Loaded information might be a subset of the processes or the record types, or just a restriction of the time interval.
- *Efficient* parallel access. Technically, this can be achieved by splitting the trace information into multiple files which can be accessed independently to prevent locking and synchronization between

[55]The inspected trace formats are RLOG2, SLOG2, TAU trace files and OTF. To encourage further reading in the design goals, two literature references are provided explicitly.: The attempt for CTF [Des10] and design considerations for OTF [KBB+06].

processes.

- *Partitioning of trace data* into sections of related information. This fosters the previous points by enabling independent and probably faster generation and analysis of those sections. In general, a section should be independent of others but it might be related to activities recorded in another section.
- *Stability* of the interface and the format. An adjustment of recorded events and their related information or an extension of the basic trace format should be possible without rewriting the complete tracing and analysis toolchain.
- *Flexibility* in terms of the stated requirements, that means gearing the format towards any specific purpose should be possible.

By itself, each requirement can be realized with simple approaches. However, several requirements are conflicting with each other, for instance recording of all potentially relevant information results in large files, which must be analyzed later.

When work on this thesis has been started, no trace format existed that fulfilled all requirements for the simulator, therefore, a new trace format has been developed (the format is described in Section 4.2). In the following, the concepts behind OTF are introduced, to foster understanding of restrictions imposed by trace formats and their interfaces.

Open Trace Format

OTF [KBB+06, KBB09] is developed in cooperation between the TAU tool group of the *University of Oregon* and *The Center for Information Services and High Performance Computing* (ZIH) of the TU-Dresden. Initially, OTF is designed to record trace information for MPI and OpenMP parallel applications. The API and implementation and contained tools are licensed under a BSD open source license. First, a few basic concepts of OTF are discussed:

- Traces of processes and threads are split into *streams* (consider them as independent sections), which are mapped into a set of files, i.e., each file contains the trace information from one or multiple streams. A *master file* contains information about the mapping from processes to streams.
- Within a trace file, an event is recorded as a single text line, the detailed structure of a record is defined separately. A new type can be defined for a stream in a *local definition*, or it can be stored globally. Definitions are stored in additional files. For some functions, such as MPI functionality, the types are predefined. To reduce the amount of redundant data, several common properties, like timestamps and information about threads, are written independently of event information, they are valid for all subsequent events until they are reassigned.
- Files are encoded in ASCII to ensure platform independence and readability by third-party tools. Random access is realized by performing binary search over the timestamp. To decrease the file size blocks of data can be transparently compressed with *zlib*[56].
- Auxiliary information, locally defined records, snapshot records and statistics can be optionally added. Since information about the state of the program is stored implicitly in a stream, random access is difficult. For example, the call graph is recorded by issuing several nested start events. When a file is accessed at a random position, then the earlier events are skipped and should not be read but they are essential to understand the current state of the process. Thus, in OTF, derived information can be added in additional files, a *snapshot* holds the current state. That means the snapshot keeps all variable information about the nested function invocation, or I/O activities. *Statistics* store summary information from beginning up to the current time, arbitrary intervals can be derived from these values by subtracting start from end statistics. Thus, information before the snapshot is not needed and random access is possible.

[56]`http://zlib.net/`

- A low-level and a high-level C-interface are provided to access OTF files.
- Several tools are provided to access and manipulate trace files: to filter processes, to compress or decompress traces, to dump tracefiles as text, to add statistics or snapshots, or to change the number of streams per file.

API The low-level API is briefly introduced based on the source code of OTF and the included documentation (version 1.7.0rc1 as of 2010-03-30).

Important OTF functions and an excerpt of the API documentation are provided in Section A.1. Exemplary usage of the API to write a trace file is shown in Listing 2.7. In the example, a single function entry and exit for the main function of one process are written.

Listing 2.7: OTF example code (as provided by the API documentation)

```c
#include <assert.h>
#include "otf.h"

int main( int argc, char** argv ) {
// Declare a file manager and a writer.
        OTF_FileManager* manager;
        OTF_Writer* writer;
// Initialize the file manager. Open at most 100 OS files.
        manager= OTF_FileManager_open( 100 );
        assert( manager );
// Initialize the writer. Open file "test", writing one stream.
      writer = OTF_Writer_open( "test", 1, manager );
      assert( writer );
// Write some important Definition Records.
// Have a look at the specific functions to see what the parameters mean.
      OTF_Writer_writeDefTimerResolution( writer, 0, 1000 );
      OTF_Writer_writeDefProcess( writer, 0, 1, "proc␣one", 0 );
      OTF_Writer_writeDefFunctionGroup( writer, 0, 1000, "all␣functions" );
      OTF_Writer_writeDefFunction( writer, 0, 1, "main", 1000, 0 );
// Write an enter and a leave record. time = 10000, 20000
// process = 1 function = 1
// Sourcecode location doesn't matter, so it's zero.
      OTF_Writer_writeEnter( writer, 10000, 1, 1, 0 );
      OTF_Writer_writeLeave( writer, 20000, 1, 1, 0 );
// Clean up before exiting the program.
      OTF_Writer_close( writer );
      OTF_FileManager_close( manager );
      return 0;
}
```

The API does not support mapping of processes to hardware topologies. However, a potential mapping could be recorded by forming process groups, or by storing comments according to the OTF developers. Relations between two processes are created explicitly by invoking the functions `OTF_Writer_write[Recv|Send]Msg()`, or for a collective MPI call via `OTF_Writer_writeBeginCollectiveOperation()`.

OTF is under constant development but the developers worry of modifying the existing interface or file format because that could break comparability. Support to add arbitrary data to an event was added recently, in new versions key-value pairs can be added to a call [KGS+10]. An example with these extensions is provided in listing 2.8. First the key and the datatype of the value pair are defined by using `OTF_Writer_writeDefKeyValue()` (Line 22), then a value must be set (Line 28). To additionally store the defined key/value list to the trace records, the API function with the suffix *KV* can be invoked (Line 30). Those new functions have been added for most previously existing API functions. Prior to this approach, it was cumbersome to extend existing records, eventually comments could be used, but that was tedious and inefficient.

Listing 2.8: Adding arbitrary information with key/value pairs (as provided in the API documentation)

```
#include <assert.h>
#include "otf.h"

int main( int argc, char** argv ) {

  OTF_FileManager* manager;
  OTF_Writer* writer;
  OTF_KeyValueList* KeyValueList;

  manager= OTF_FileManager_open( 100 );
  assert( manager );

  writer = OTF_Writer_open( "mytrace", 1, manager );
  assert( writer );

// Initialize the prior declared OTF_KeyValueList.

  KeyValueList = OTF_KeyValueList_new();

// Write a DefKeyValue record that assigns key=1 to name="first_arg" with
  →description="first argument of function" and type=OTF_INT32.

  OTF_Writer_writeDefKeyValue( writer, 0, 1, OTF_INT32, "first_arg", "first␣
    →argument␣of␣function" );

// Append a signed integer for key=1 to the initialized KeyValueList.

  OTF_KeyValueList_appendInt32( KeyValueList, 1, 25);

// Write the entries of the KeyValueList together with the enter record.
  →Afterwards the KeyValueList will be empty!

  OTF_Writer_writeEnterKV( writer, 10000, 100, 1, 0, KeyValueList );

// Clean up before exiting the program. Close the OTF_KeyValueList.

  OTF_KeyValueList_close( KeyValueList );
  OTF_Writer_close( writer );
  OTF_FileManager_close( manager );

  return 0;
}
```

In the Birds of a Feather [KWGS10], the future direction of (OTF) tools has been discussed. OTF evolves into a new (OTF2) format with funding from the BMBF and DOE. A unified performance measurement infrastructure is envisioned in form of the Score-P[KO11] measurement environment, which handles creation of event traces in OTF2 and call graph profiles for Cube-4. Similar approaches to unify the infrastructure have been tried in the past, OMIS [LWSB97] is an example for an earlier interface, which also supported debuggers and load management systems.

2.5. Discrete-Event Simulation

In this section, the concept of modeling and simulating real world *systems* is briefly introduced. Then a few simulation engines are presented that assist in formulating models of systems and their analysis.

2.5.1. Modeling

"A model is a system's representation within a chosen experimental frame; i.e., a model must have a purpose or set of questions it can answer" [PK05]. In other words, a system model is an abstraction and simplification of the existing system. The purpose of such a model is usually to either explain a system's behavior or to predict it. Compared to the real system, a model makes it easier to conduct experiments and to understand results.

The modeling process by itself already increases understanding of the system's behavior. Besides gaining knowledge of the system, models allow scientists to perform experiments which are impossible in reality either because system manipulation is infeasible, too time-consuming, too expensive, or too dangerous.

During the creation of a model, an abstraction and simplification is made in such a way that "factors" which are relevant to answer the questions are considered – irrelevant factors are removed. *Modeling* is the process of constructing a model for a real system. It involves identifying *entities*, i.e., compartments of the system which might interact, and their characteristics. The importance of the interactions and characteristics depends on the scientific question. On the one hand, characteristics include a set of attributes, each with potential states, and on the other hand, rules define how these states change over time and upon interaction with other entities.

There are many formalisms that assist in system modeling. Further, these formalisms provide concepts to analyze the created models. Many models use mathematical concepts and language to describe and analyze the system. Example model formalism are: *queueing models* [Tak82], *artificial neural networks* [AB09], *agent-based models* [MN10], *game theory* [Ras07] or modeling via *differential equation* (see [Ger99] for more information about *mathematical modeling*).

For example, a cash desk in the supermarket could be represented by a queuing model: a random distribution defines the time between the arrival of two customers, they queue up in a line and a cashier, who needs a variable (random) time to scan the goods. Such a model, for example, would allow analyzing the wait time for the customers to optimize the number of cashers in the shop depending on the arrival times of the customers. Several advantages and limitations of modeling are discussed in [PK05] (Section 1.7.1).

Conceptual domain model Usually similar systems of a domain share similar processes and entities; these common processes and entities can be described in isolation from a specific system with a model that is conceptual. This conceptual *domain model* contains models for the relevant processes and entities, and describes their characteristics. Thus, those models can be combined to model a class of real systems[57].

For example, in cluster computers, behavior of nodes and network is similar for many existing systems, these entities and processes can be abstracted into a domain model. In the process of *parametrization*, a domain model is instantiated to create a representation of the real system; this is done by identifying and defining all parameters necessary for the specification of the particular real system. Thus, to model a specific real cluster, a model can be created by instantiating entities of the domain model and adjusting their characteristics. The result of this process is a specific model representing the particular system.

[57]In literature related to processing there is no common abbreviation or description for something like a conceptual *domain model* – typically it is just called model. The term *domain model* is used in software-engineering in a similar way as that intended by the author of this thesis, though. The author claims the description helps the reader to distinguish a specific model that represents a system and a model that represents multiple instances in that domain.

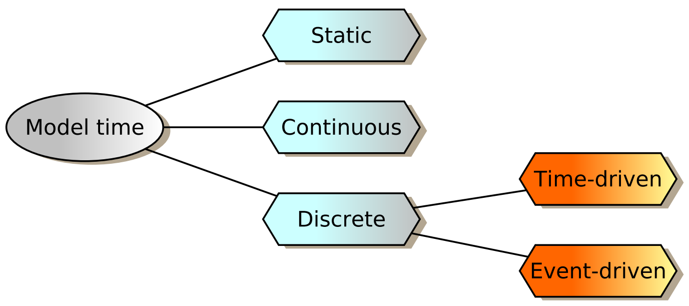


Figure 2.23.: Alternatives of passing time within a model.

Model time A model of a system can include dynamic behavior that is time dependent. Typically, the *virtual time* of the model increases while states of entities are manipulated. The virtual time passing by is also referred to as *model time*. Model time represents the time in the model, while the *wall-clock time* is the time needed to run the simulation – in a simple model, years of model time can pass within one second of simulation.

Several time representations are possible, the alternatives are shown in Figure 2.23. *Static* models do not consider time at all and thus the model is time invariant. With a *continuous* time model, the states and interactions change steadily over time. For example, this is true for formal mathematical models based on differential equations with time as a variable. In analytical models those differential equations are solved directly, that means the exact analytical solution to the equation is determined, then this function can be evaluated for arbitrary model time.

Discrete time model By using a *discrete* time model, the state of an entity changes only at specific points in time, that means no modification of the system occurs in between. Discrete time models can be subdivided into *time-driven* and *event-driven* models. In a time-driven model, the time steps forward by a fixed increment, e.g., 0.1 s. For example, the time of numerical models is incremented in these small steps, in each step the equations are evaluated with approximations such as the *finite-difference method* or the *finite element method*.

Event-driven models advance time by setting it to the next timestamp at which the system's property changes. In the context of event-driven models the cause of the system state manipulation is called *event*. Therefore, an event-driven model skips intervals in which the system's state is invariant:

> *"In discrete-event simulation, the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of state in the system"* [Pid04]

Often, in discrete models stochastic behavior is involved, for example, the duration of a process or the manipulation of a state is changed by a random influence. This causes nondeterministic behavior.

Checking the correctness of models As a model is an abstraction of reality it cannot serve all purposes. There are methods for checking that the model resembles system behavior closely enough – in respect to the scientific question. To ease assessment of results and to increase insight, complexity of a model should be minimal.

One of the first papers about systematic correctness checking is [Ste79]. In this paper, a conceptual model, which is implemented by a computer, is referred to as *computerized model*. Terminology about correctness checking between *reality*, *conceptual model* and *computerized model* is provided, the definitions given are:

- Model qualification: *"Determination of adequacy of the conceptual model to provide an acceptable level of agreement for the domain of intended application."*
- Model validation: *"substantiation that a computerized model within its domain of applicability posses a satisfactory range of accuracy consistent with the intended application of the model"* .

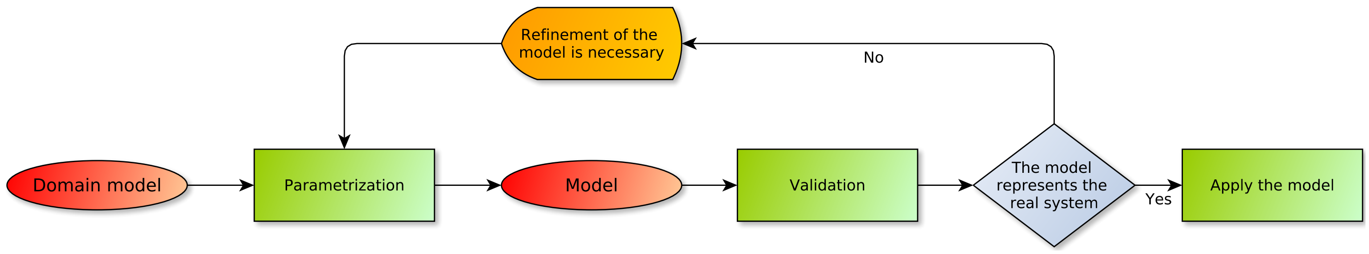

Refinement of the model is necessary
No
Domain model
Parametrization
Model
Validation
The model represents the real system
Yes
Apply the model

Figure 2.24.: Process of creating a model for a real system.

- Model verification: *"substantiation that a computerized model represents a conceptual model within specified limits of accuracy."*

When a model is built parameters can be added or tuned until the model and the real system provide similar results. This loop of refining a model until it resembles behavior of the real system is visualized in Figure 2.24. While a model of a system is extended and evaluated, important processes and parameters are identified. Eventually the domain model is adjusted, taking more or different characteristics of the real system into account (this loop is omitted in the figure).

2.5.2. Simulation

> *"Simulation is the imitation of some real thing, state of affairs, or process. The act of simulating something generally entails representing certain key characteristics or behaviors of a selected physical or abstract system."* [Wik11]
>
> *"We define simulation as the modelling of dynamic processes in real systems, based on real data, and seeking predictions for a real system's behavior by tracing a system's changes of states over time (starting from some initial state). In computer-based simulations models are represented by (simulation) programs, and simulation experiments ("runs") are performed by a model's execution for a specific data set."* [PK05]

The latter definition given by Page is tailored to computer aided simulation. Compared to an analytical model, computer aided simulation requires a conceptual model which is then translated into a computer model – often directly into source code. Finally, the parameterized computer model can be executed to simulate the system's behavior. *Verification* of the computer model evaluates how well the implemented model resembles the conceptual model. And *validation* proves that the (computerized) model is applicable to answer the scientific questions.

A refined definition for validation in the domain of computer simulation is given by Page:

> *"Model validation is the activity of establishing that a model is a suitable substitute for a real system with regard to the goals of a simulation study.*
> [...]
> *Model verification in the wider sense is the activity of establishing that a model is correctly represented and consistently transformed from one form of representation into another[...]. Model verification in the narrower sense is the activity of formally proving the correctness of model representations and their transformations – relative to some canonical model (e.g., a formal specification)."* [PK05], page 198

In this newer definition the previously used term qualification has been merged into validation; a simulation program is considered to be a model for reality.

Discrete-event simulation With discrete-event simulation the state of the system is changed by events, an event is executed at a specific point in time. The abstract processing of a discrete-event simulator is visualized in Figure 2.25. Upon start of a simulator, the specific model of the system under analysis must

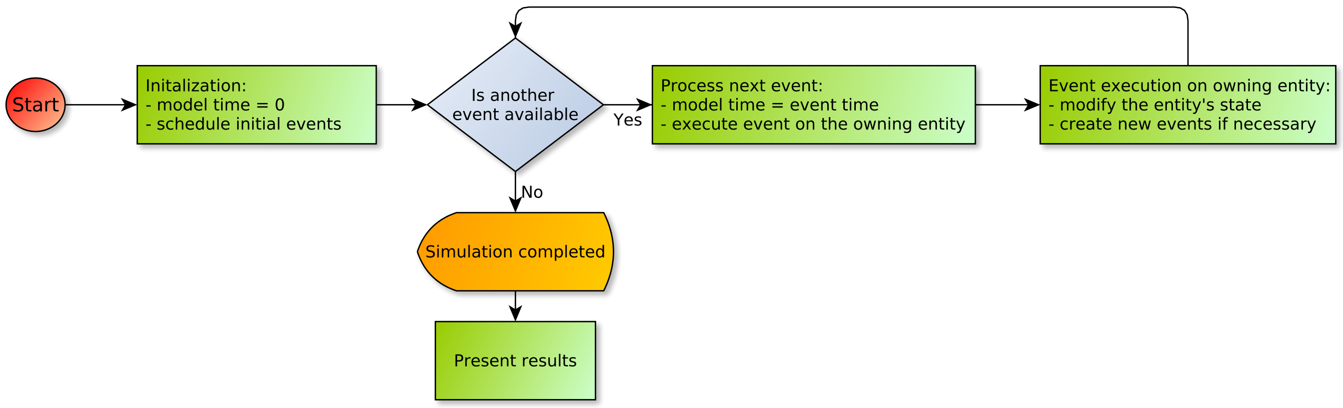

Start
Initalization:
- model time = 0
- schedule initial events
Is another event available
Yes
No
Process next event:
- model time = event time
- execute event on the owning entity
Event execution on owning entity:
- modify the entity's state
- create new events if necessary
Simulation completed
Present results

Figure 2.25.: Processing of a discrete-event simulator.

be initialized, in this process the simulator adds initial events to an *event queue*. While an event is available there, the simulator processes the next (future) event in the queue. In general, an event is associated with a particular entity of the model. In this process, the model time is incremented to the event's time and according to the type of the event and its content, the associated entity can then modify its internal states and might submit new events to other entities (or itself).

Once all events have been processed, the simulation stops – presenting the results of the observed behavior. Some models generate infinite events. To prevent endless execution in this case, the termination of the simulation can be enforced, for example, if the accuracy of stochastic processes is sufficient, or if enough events have been processed.

Interested readers find a solid background of discrete-event simulation in *The Java Simulation Handbook* [PK05] and in [Pra04].

Characteristics used to assess discrete-event simulators and relevant design variations To compare and understand the basics of related work, first, a set of design aspects and characteristics is introduced that will be used to further describe it. Additionally some design principles and alternatives are introduced – the presented content is chosen because it is relevant for simulating parallel applications and the system underneath. In brief, this includes: the environment the simulation framework runs on, the way a system model is specified and read by the simulator, the model describing activity that shall be executed by the entities, the design of the simulation core, and the assistance provided by the framework to assess simulation results. A description of these characteristics and some further explanation of common techniques is given in the following:

- **Environment**: Aspects related to the environment in which the simulation framework operates are mentioned – certain software and hardware requirements. Also, license terms are given, especially whether the tool is open source or under a proprietary license.

- **System model**: Typically, a simulation engine can execute models of many discrete (or mixed) mathematical systems. All presented tools are suitable for discrete-event simulation and mixed systems and simplify the modeling process; for example, various random number generators are supplied in each framework to ease modeling of stochastic processes. While there are many approaches to model a system, a simulator might require a certain formulation of the entities and their interaction partners. On the one hand, the simulator must support constructing new model entities and defining their behavior. On the other hand, for an experiment domain, entities must be instantiated and parameterized to represent a specific system.

  There are two dominant styles for modeling interaction of entities in discrete-event simulation. Behavior and interactions of a simulated entity can be either modeled by defining all transformations for the entity – this is called *event-oriented* modeling – or by specifying the life-cycle, i.e., the sequence of activity and interaction, for each entity. In the latter, so-called *process-oriented* modeling style, a simulation framework must offer an API to advance model time, to wait for a given event, and to create a new event. To give an illustrated example for a process, consider the life-cycle of an application which first reads input data, then computes, and at last outputs the result. Whenever the application performs one operation it passes model time until the operation has completed – for example, it starts the read process and continues when the data is ready. A sophisticated process-oriented example is provided for the CSIM20 simulation library on Page 92.

During the modeling-process, this domain model and a specific system model must be formulated in a representation understood by the simulator. One possible representation is to describe the entities and behavior in an abstract format which is parsed by the simulator at run-time, or it is transformed into a simulator specific format. This format can be just a simple text format, or expressive such as UML diagrams (especially state-diagrams and interaction diagrams are suitable to describe the behavior of the entities). Often, it is possible to build the relevant entities by using a simulator specific API. Thus, the developer formulates the model in a programming language.

- **Activity model**: With the system model, entities and their interaction partners are defined. However, as the system model fixes the entities and their behavioral scheme, the actual input which drives execution must be described – e.g., in our supermarket example, the queue of customers and their attributes must be provided to the system model. In this thesis, the methodology of supplying input to the simulator is referred to as *activity model*. The model that translates activity into reactions (entity behavior) must be implemented by the simulator. In a simulation run certain activities can be initiated upon creation of entities; during the execution, those initial events can trigger further activities.

  Similar to the system model, a developer can encode the activity in a programming language or directly into the entities; stochastic descriptions are also possible. To model activity in complex systems, three modeling concepts are imaginable:

  - A *stochastic* model computes state modifications with probability distributions, e.g., in our example, the arrival rate of customers and the goods in their shopping cart.
  - *Trace-driven simulation* [Kae93] is an approach where the activities are provided by a file; the simulator executes the activities sequentially as they are described in the trace. Activities of an existing system can be recorded to generate a trace, which is then fed into a simulation engine to initiate activities based on the trace.
  - The *execution-driven simulation* is dedicated to simulating compute systems. With this approach, an existing system is modified to invoke the simulation API whenever an activity takes place. While the real activities are performed, a simulator can compute model time and all relevant metrics based on a system model that computes them in the background – therewith, predicting the behavior for an arbitrary system. As an illustrated example, consider a network interconnect that should be modeled for a client-server infrastructure. Whenever a client communicates with the server, the simulator computes how long the communication took. Thus, the model time is completely independent from the wall-clock time; at the end of the “simulation”, the virtual communication time and computation time are accumulated and returned to the user – therewith, predicting these characteristics for another system.

In respect to simulation of parallel applications, it is probably not useful to use a purely stochastic model because that would imply that applications perform random operations; however, random effects could be added to the other two activity models, too. Both execution-driven simulation and trace-driven simulation reduce the burden to simulate realistic activities on a system’s model. The execution-driven simulation reduces the time to port an application to the simulator because it just requires a slight modification of the real application. In contrast to trace-driven simulation, it depends on real processes and interactions – the original code and the simulation framework must be run together on an existing system. Basically, trace-driven simulation adds another layer of abstracting – while the activity in the execution-driven simulation is fixed by the real system, trace-driven simulation offers the freedom to create arbitrary traces and to modify them.

- **Simulation core**: This characteristic gives more details about the core of the simulation engine. Since performance and scalability of a simulator are critical, focus is put on the parallelization potential of the core which is typically low.

  Performance of the simulation main loop, presented in Figure 2.25, can be increased by parallelizing the simulation core. In parallel discrete-event simulation, the entities are distributed among the available processors, each processor maintains its own model time, additional communication is necessary to transfer events between two processors. On each processor, local events are executed and model time increases independently from other processors. To guarantee accurate results, the events must be executed in the right order – the event with the lowest timestamp must be dispatched first.

  During an iteration of the simulation loop, the current event might spawn new events which are communicated to the responsible processor. Consequently, a synchronization between two processors takes place. The receiving processor enqueues the new events; still, any processor must execute events in the order of their start-time. Due to the distributed nature of a parallel simulation, it could turn out that a remote processor sends an event which has a timestamp before already executed events – this would invalidate the simulation.

  There are two approaches to ensure correct causality: conservative or optimistic event execution. A *conservative* algorithm prevents execution of further events if they might break causality. Therefore, the next local event is scheduled iff it is clear that further generated events on all processors have later timestamps than the current event. With this realization, it is evident that a parallel simulation ensures the right order of events, although a processor must pause execution in case another process might generate earlier events for this processor. This limits the scalability of a conservative implementation. In contrast, an *optimistic* approach can speculatively schedule events, causality violations are detected and repaired a by a roll-back mechanism. The roll-back mechanism works as follows: If an event with an earlier timestamp than the current model time is received, then the current state of the simulation is reverted to a valid state of the entities before the conflict occurred – a state of the system before the wrong event got scheduled.

  Depending on the specific model one or the other approach works better; if just a few events violate the causal event order, then the optimistic approach typically will yield better performance.

- **Report generation**: After a simulation experiment is conducted, the modeler must assess the simulation results. Thus, the simulator must provide information about the simulation run in some kind of report.

  A framework can simply maintain certain metrics, like model time, globally, or collect statistics, like server response time and utilization, per entity. Those metrics could be printed upon completion of the simulation.

  Analogous to performance optimization, the progress of executed events can be recorded in *traces* and visualized post-mortem – showing the event order for each entity in a separate timeline similar to the displays used for performance analysis. However, storing the accurate state of each entity during each simulation step is expensive in terms of performance and storage capacity.

  Another approach is to illustrate low-level processes and events in a domain-specific graphical view. For instance, in a simulation of a traffic system a road map can be overlayed with the lights and cars. With this so-called *animation*, processes and suitable key characteristics of the entities are presented in an understandable way to the modeler.

2.5.3. State of the Art

Discrete-event simulation is widely used in industry and science and thus many simulation engines and tools exist to build domain specific simulators. Five mature frameworks under active development are provided in this section, their differences and capabilities are discussed based on the characteristics stated before.

CSIM 20 The *Parallel Process and Diagrams Simulator* [MS09] is a toolkit for process-oriented discrete-event simulation.

- **Environment**: The CSIM library operates on all common platforms: UNIX flavors, Mac OS X or Windows. Several GUIs are shipped with the simulator to generate models or to visualize and analyze simulation results. The simulator is primarily coded in C and supports C and C++ models. A Java version is available, which has slightly reduced capabilities compared to the C version. CSIM is commercialized by a company and thus not freely available.
- **System model**: The modeler encodes the model in C (or C++) by calling the CSIM API to create entities and to describe the behavior of the processes. With the framework, many basic models are supplied which assist in building more complex processes and entities.

  In the following, a brief overview of the basics objects is given: A *process* represents an active element of a system. Due to the process-oriented scheme, all active entities of a system are modeled processes. Also, a message which is transferred between client and server over a network is such a process. Each process has its own internal state encoded in arbitrary local variables, a priority and an activity state. The state of a process is either executing, pending, holding (until a well-defined amount of time passed by), or waiting for a particular event.
  A *facility* is an execution unit which performs a task and realizes a queue model: processes can queue jobs at a facility; once a facility becomes idle it executes the next pending job in an exclusive way, similar to a server which serves pending requests sequentially. Real world instances of a facility are the cashier in the supermarket or machines of an industrial factory – during the manufacturing of a product, several different types of machines are occupied for a period of time. Multiple servers can be provided by one facility.
  A *storage* has limited capacity which can be allocated by processes. Main memory is an example of a storage object. Whenever a storage or a facility does not contain the resources to satisfy the demand, processes are stopped and queued until enough resource become available.
  A *buffer* is a resource with a maximum capacity in which a process can store some resource while other processes use them. Compared to the storage, both producer and consumer will be queued if the storage capacity does not suffice to store/fetch the product. A warehouse with limited capacity is an example buffer. If it is full then no more goods can be stored; if it is empty the consumer must wait until goods are available.

  Process communication is possible via *mailboxes*, a process can send a message to a mailbox or try to receive a message. Sending of a message queues the message in the mailbox, the sender continues without waiting for reception. Receiving processes are suspended until a message can be received. In CSIM, messages are integers (or pointers to arbitrary data).
- **Activity model**: The activity is encoded directly in the C(++) system model in a processes-oriented style. To increase model time, processes can wait for a period of time, or until a certain event occurs. The purpose of an *event* is to synchronize processes. Remote processes can signal the event to start either all waiting processes or to wake up the first process.
- **Simulation core**: In the documentation there is no information supplied regarding the simulation core. Presumably, the simulation core is not parallelized at all, instead events and processes are executed sequentially. However, theoretically, processes could be realized by spawning one thread per process and activating them once an event is scheduled.

- **Report generation**: Provided components contain a basic reporting mechanism to collect statistical information or histograms about element-specific metrics, for example, throughput and response time of a facility can be provided. The simulation can be run until the metrics satisfy a certain confidence interval. In debug mode, an ASCII trace of simulator activity can be printed.

To understand the process-oriented simulation better, we discuss a simple model for a factory which assembles goods of specific sizes depending on the orders of customers. The owner of a factory could use such a model to evaluate and optimize the factory setup consisting of a number of machines and workers to satisfy demand in a cost-efficient manner.

Assume that in the production process two types of machines are needed. In the factory described in the particular model below, one machine of type *A* and four machines of type *B* are available. When one of the type B machines should be operated, a number of workers equal to the size of the order is required because each worker places and assembles a compartment of the product; in total, 15 workers are employed. After the product is produced one worker operates machine A with the product, which packs the completed product and prepares it for shipping. Once the packaging completes, the worker transfers the product to a truck, which is out of the scope of the simulation and the border of the *factory system*. Then the worker is ready to handle the next product. This simple model implementation including timings for arrival of orders and the processing is given in Listing 2.9. Normal and exponential distributions represent the variance of the interval between order arrival and the processing time due to variability in the human worksteps.

Listing 2.9: Simple factory model for CSIM20. The example is created with the specifications provided in the starter guide [MS09].

```
#include <cpp.h> // include the CSIM C++ header file

// In the factory, there is only one machine of type A.
facility * machineA;

// There are multiple machines of type B; to operate one of them several
// workers are required, depending on the size of the product which is ordered.
facility_ms * machineB;

// Manages the available workers.
storage * workers;

// The simulation main routine is declared in a C header and is defined in C++.
extern "C" void sim()
{
  // Create the main process of the simulation.
  // Each process has its own stack.
  create("sim");

  // Instantiate the system model:

  // One machine of type A is available.
  machineA = new facility("machine␣A");

  // We have 4 machines of type B.
  machineB = new facility_ms("machine␣B", 4);

  // 15 workers are available.
  workers = new storage("workers", 15);

  // The system model is now completely set up.

  // Create the incoming orders, activity and system model are mixed.
  // Loop until 5000 seconds have been simulated.
  while(simtime() < 5000.0) {
    // Wait for an amount of time as defined by the exponential distribution (mean
      →= 130).
    // Thus, mean time between two orders is about 130 seconds.
    hold(exponential(130.0));

    // Assembly of a product for an order requires between 1 and 10 workers,
    // depending on the size of the object to produce.
    // All sizes are in equal demand, therefore we use rand().
    // Call the order function to start an order process (see below).
    order(rand() % 10 + 1);
  }

  // Create a report for this simulation run.
  report();
}

// Life-cycle of a single order in the factory.
void order(int size)
{
  // Define this as an independent process: The current instruction pointer and
  // stack are saved and stored in the next event list.
  // Control is returned to the calling process.
  create("task");

  // Request a number of workers; if they are not available wait until they are.
  workers->alloc(size);

  // To utilize machineB, an additional worker is required to operate the machine.
  // The time is normally-distributed with a mean of 150s and a variance of 10s.
  // Each size of the product adds 10 seconds to the average time needed.
  machineB->use( normal(150.0 + size * 10.0, 10 ));

  // All workers finished their work carrying the product.
  // One worker is needed to package the product on the truck.
  workers->dealloc(size - 1);

  // A product for this order utilizes machine A for 5s times its size.
  machineA->use(5.0 * size);

  // Complete the order by waiting until it is transported outside the factory.
  hold(exponential(10.0));

  // The last worker who brought the product outside to the truck finishes
  // his work and returns to the worker "pool".
  workers->dealloc(1);

}
```

Parallel Object-oriented Simulation Environment POSE [WK04] is a library which eases the creation of simulators.

- **Environment**: POSE is coded in *Charm++*, which is a parallel programming interface for C(++) and a run-time system. The POSE library and several examples are shipped with the Charm++ source code distribution.
- **System model**: System model and activity model are explicitly encoded in the Charm++ programming paradigm. While this freedom allows modeling of arbitrary systems, there is a lack of support

to handle common tasks of a simulator.

Process-oriented behavior of entities and reactions to events are basically coded in C++. A small language extension is made to the C++ standard to increase the readability of the model code.

To model an entity – called *poser*, an interface description which specifies methods to accept events and a C++ implementation are necessary. With a source-to-source compiler, the model objects are translated to the C++ language. Entities are classes derived from POSE superclasses, messages (events) transmit data and information between two posers.

- **Activity model**: New instances of posers can be scheduled for creation in the simulator with `POSE_create()`, a message can be submitted to configure the entity according to its intended purpose.

  Future events can be issued by calling the language extension `POSE_invoke()` with the event function, data (the argument to the function), a reference to the entity and the future timestamp at which the event (the function) should be triggered with the data. With the `elapse()` function, an entity can pass its model time. Both event-oriented and process-oriented simulation are possible.

- **Simulation core**: By leveraging Charm++, the simulation is parallelized, also automatic load balancing of the simulation objects is built-in.

  Besides a conservative synchronization strategy, several optimistic and adaptive synchronization strategies are contained in the parallelized core. To handle optimistic synchronization a roll-back mechanism requires the programmer of a poser to guarantee that method invocation is free from side effects. Internally, to perform a roll-back, the inverse function is invoked on all posers with the events which must be undone.

- **Report generation**: Upon completion of a simulation statistics about the simulator itself are printed, e.g., the run-time, the number of actual processed events and speculative events. There is an infrastructure provided that eases reporting, however, the distributed code uses `printf()` to print simulation behavior for further assessment.

OMNet++ is a community simulation framework which provides powerful tools for the analysis of all kinds of networks [VO10, VH08].

- **Environment**: While the simulation core is written in C++, several Eclipse GUIs and wizards assist in developing, executing, debugging and analyzing the results. They are running on Linux, Mac and the Windows platform.

  OMNet++ is under an *Academic Free License*. For non-profit intentions, OMNet++ grants rights akin to the GPL. *OMNEST* is a commercialized version of OMNet++, which permits commercial usage.

- **System model**: In OMNet++ models are built from *modules*. Simple modules from a palette of disjoint archetypes are provided similarly to CSIM20. Existing simple models are interconnected, their composition forms behavior and real-world entities. Inter-module communication takes place by exchanging messages over connection links. A link can be used to model packet transmission and, therefore, supports the parameters: data rate, propagation delay, bit and packet error rate. Events are represented by messages.

  Model communication topology is defined in a textual language, the *NEtwork Description* (NED). This hierarchical format supports reuse of existing descriptions as templates and it permits a hierarchical structure – simple modules form compound modules with a given purpose. A NED is created by utilizing a library, or directly within a GUI editor similar to Labview.

  If the model is not fixed in the NED, further parameterization of the modules can be provided in a textual configuration file in the *INI file format*. Also, an automatic executing of experiment ensembles can be defined. At run-time, the program can adjust the communication topology and even instantiate new modules.

There are many publicly available models for OMNet++, especially in the area of low-level packet simulation of networks. Those can be used to derive a model for new components and systems.

- **Activity model**: Activity is encoded in simple modules, either in a process-oriented or event-oriented style. Upon instantiation of an object, its constructor can inject initial events. In the process-oriented approach a single method of the module encapsulates the whole process; with `wait()`, control is returned to the simulator which keeps all the pending processes in memory.

  The event-oriented style calls a function whenever a message (event) is received. However, an API is provided to rewrite the event-oriented style with state-machines. State-machines can be nested to increase readability and reuse.

- **Simulation core**: The simulation core and the simple modules are written in C++. MPI is supported for parallel simulation. Internally, the parallel simulation offers a conservative synchronization strategy.

- **Report generation**: The framework can record automatically several metrics' human-readable statistics and histograms, as well as the activity of modules (including message exchange and module debug output) of the basic archetypes. In self-written modules, new statistics can be added by specifying them in the NED of the module, the values can then be set within C++ by emitting signals. Traces are recorded in so-called *log files*.

  A viewer in the IDE displays the log files; activity per object is rendered as a timeline similar to off-line visualization tools. Edges in the graph represent causal relations between the network layers. Additionally, a visualization plugin animates and replays activity, e.g., the packet routing via network links and nodes is overlayed to the network topology.

Layered Queueing Network Solver The Layered Queueing Network Solver software package (LQNS) provides an analytical solver and a simulator tool to assess queuing systems [Woo02, FMW+11].

- **Environment**: The analytical solver and simulator are coded in C and C++, and they are available on the Macintosh and Linux platforms.

  In order to download LQNS, one must agree to a proprietary software license, sign it and send it to Careton University, then username and password to access the executables of LQNS are provided. The license permits personal evaluation within a period of 6 month, after that period annual reports sketching LQNS usefulness must be sent to Careton University.

  With the *Java Layered Queueing Network Definition tool* (JLQNDef), a Java swing GUI is provided to create and manipulate models. The LQNS simulator is built on the ParaSol [MKR95] discrete-event simulation system.

- **System model**: A queueing system contains waiting lines (*queues*) and *servers* which execute pending requests. While a server performs the activities triggered by a request, it might spawn new requests on arbitrary queues. Delays are represented in service times of servers. Layered queueing networks extend the queueing model in the notion that queues and servers can form complex hierarchical entities in which requests are passed between the layers[58]. Service times of an entity depend on the definition by the components' underlying layers. This eases the understanding of an entity and allows a better reuse of entities within templates.

  In the context of LQNS a simplified description of the elementary modeling concepts are: A *processor* represents a pure server – executing requests on demand which requires time. Scheduling of pending requests is done by selecting one of the tasks with the selected queueing discipline (FIFO, preemptive priority based, *completely fair queue* (CFQ) or random).
  A *task* encapsulates the activities and resources to process a particular type of request – a request of a task can be parameterized according to a task description. Each task has a queue for incoming

[58]See also the description of layered queueing networks on Page 59.

requests. Upon execution computation time on a processor can be requested, or new requests are issued to other tasks[59]. Processors with CFQ scheduling bundle several tasks into groups, then execution time is distributed equally among all tasks of a group.
Model components can be replicated, e.g., to distribute requests of one queue among multiple servers.

In LQNS models are defined in the XML format – processors host tasks and perform activities. Developers use the programming language *LQX* to control the execution of multiple experiments, input and output parameters of the analysis and simulation can be defined. The language is similar to PHP syntax and permits access to model components, for instance, to query metrics of a particular processor.

- **Activity model**: Each task assigns an *activity* graph describing the sequence of operations which get executed by a request. The activity sequence is specified in a directed graph, nodes represent operations to perform, including the possibility to join or fork multiple operations or to repeat (loop) activities, and edges symbolize the transition. Initial requests are created by *reference tasks*.

  A *request* can be either synchronous (waiting for a reply) or asynchronous. Synchronous mode does not require the operation to finish before the reply is created. Instead upon arrival operations needed for the response can be performed, then further activity can be executed while the requester continues. Replies can also be delegated to another task, which then responds to the requester.

- **Simulation core**: As the simulation core of LQNS utilizes the *Parallel Simulation Object Library* (ParaSol) and the limitations and advantages of ParaSol apply. ParaSol is coded in C++ and permits parallel simulation by using PVM or MPI with an optimistic synchronization. Development of the library seems to have stopped before the millennium, therefore, ParaSol is not described further in this thesis.

- **Report generation**: Analytical solutions and simulations calculate statistics like the processor utilization, queueing delays, service time or mean delay for (a)synchronous requests. This data is provided in XML or human-readable text files. Additionally, ASCII histograms for activities and tasks can be created. During a simulation run internal activities can be filtered and output to standard-out, the user specifies the interesting events with regular expressions.

An introduction to layered modeling of software performance and further documentation is available on the project's web-page `http://www.sce.carleton.ca/rads/lqns/`.

Discrete-Event Simulation and Modeling in Java DESMO-J [PK05] is a framework for discrete-event simulation.

- **Environment**: DESMO-J and all models are build in Java. A customizable GUI called *Experimental Starter* helps parameterizing the model and to assess experimental results. DESMO-J is distributed under the Apache License (comparable to the GPL).

- **System model**: Developers encode the model explicitly in source code, both process-oriented and event-oriented modeling styles are supported. For each entity and event type an own class is created and inherits from the provided framework classes.

  Analog to the CSIM several advanced modeling elements like queues and resources are provided, they are called *higher-level modeling constructs* in DESMO-J.

- **Activity model**: Both event-oriented and process-oriented simulation are supported. In the event-oriented model, the execution of an event starts an event method, which then manipulates the states of the entities. In contrast, with the process-oriented style, the simulator performs all activity in the

[59]In fact, in LQNS a task is related to a resource and it might offer multiple *entry* points. Depending on the type of the request, a particular activity can be started. In that sense, the terminology of a task seems different from the common notion.

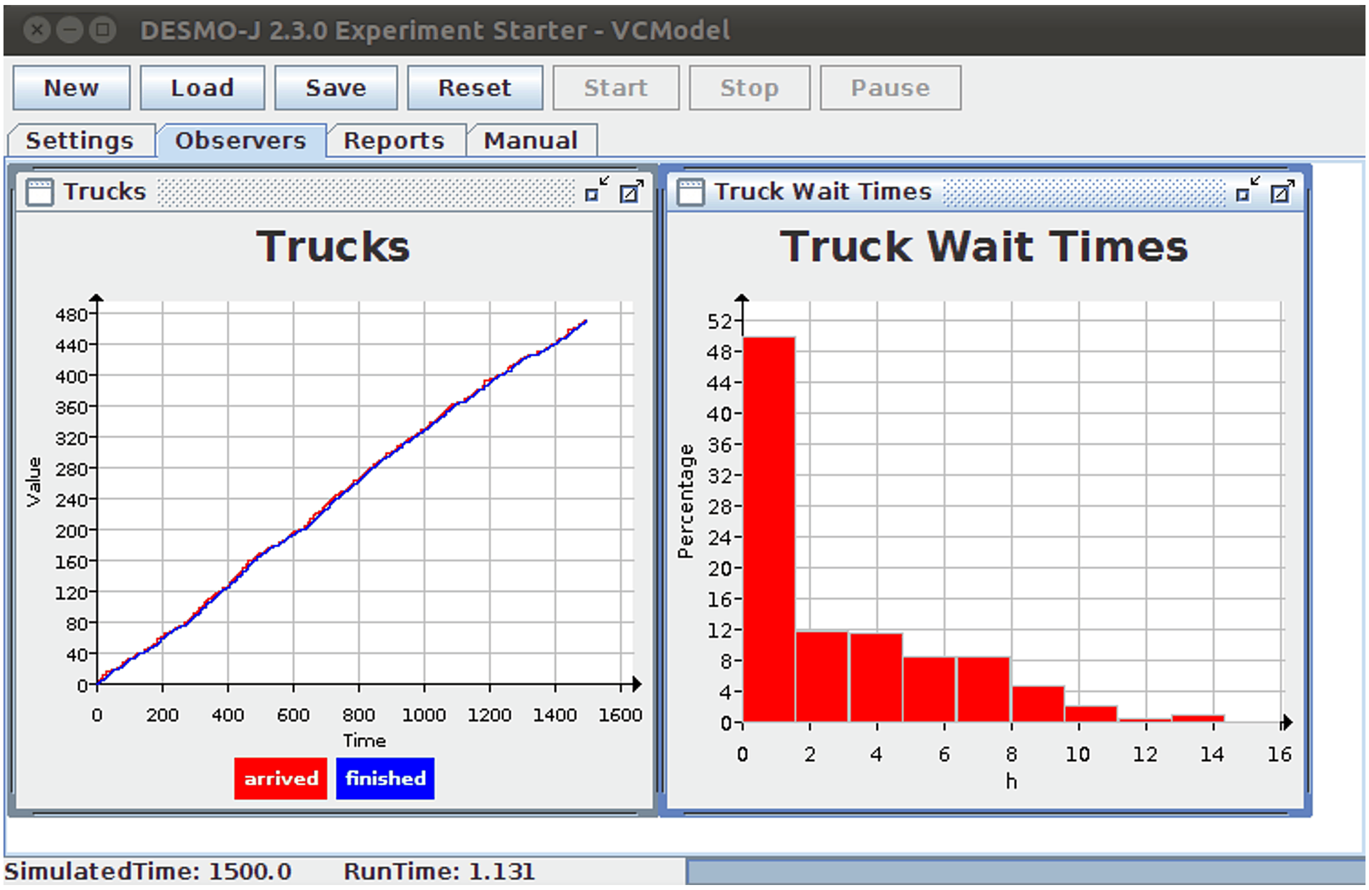


Figure 2.26.: DESMO-J GUI that can start an experiment and assists users to evaluate the outcome. Here truck arrival and departure at a container terminal is shown (in the left diagram) and the truck wait times until they were loaded (in the right diagram).

`lifeCycle()` method of an entity. An API supplies required functions to wait for a signal from other entities or to pass model time. Therefore, it is quite comparable to CSIM20.

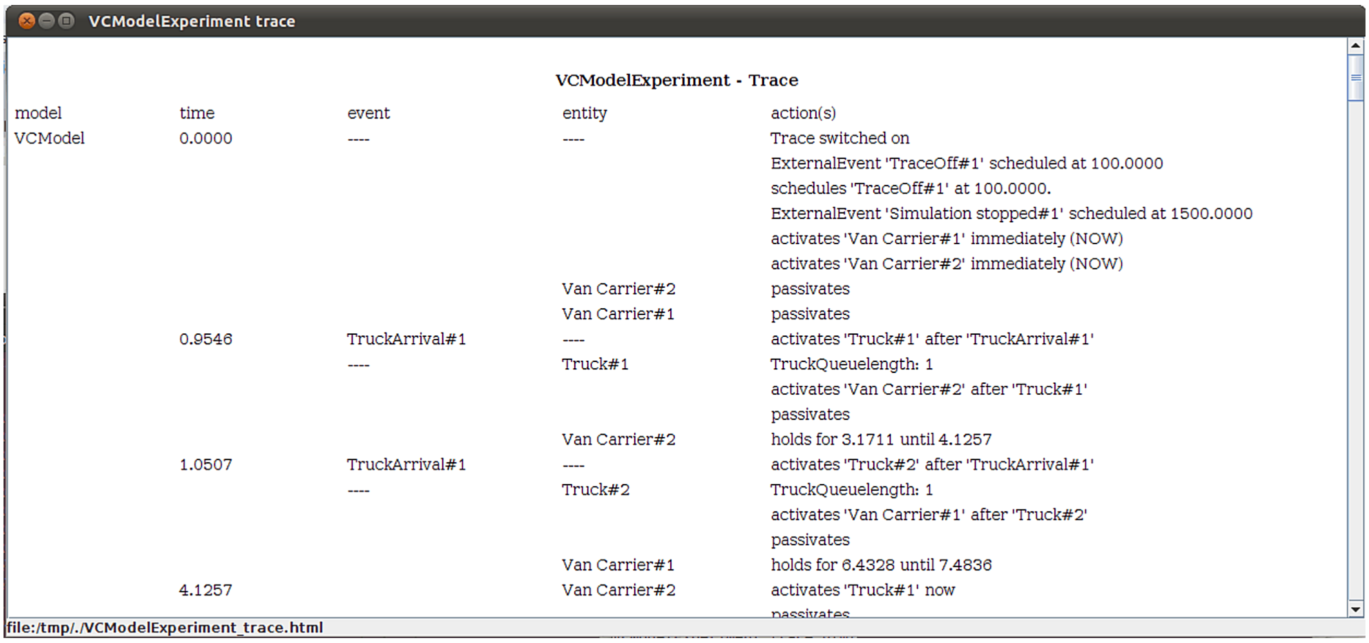
- **Simulation core**: The simulation core processes the events sequentially.
- **Report generation**: The framework provides classes to ease collection of statistics and histograms. Higher-level modeling constructs report relevant metrics, such as the average time in a queue, automatically,. Upon completion of an experiment the results including debugging and error messages are output to HTML files. Additionally, a user can request to record scheduled events and actions in an HTML (or XML) trace file.

To illustrate some capabilities example screenshots of the *Experimental Starter* and the evaluation are provided. In this experimental run a queueing model shipped with the DESMO-J distribution[60] is executed. This model simulates a container loading station on a harbor: trucks load containers – when a truck arrives it requests a container; only one truck can be loaded on the loading zone. A van carrier picks up the requested container and moves it onto the truck. Finally, the truck leaves the loading station.

The results are shown in Figure 2.26 and the HTML trace is displayed in Figure 2.27. In the left window of Figure 2.26 the arrival times and completion times of each individual truck are marked; in the right window, a histogram of the average truck wait time is plotted.

Model-specific adjustment of the GUI is done by modifying the GUI's XML file. The latest version (April 2011) includes support for 2D and 3D animations.

[60] `http://desmoj.sourceforge.net/tutorial/events/1.html`



```
                                VCModelExperiment - Trace
model      time      event            entity          action(s)
VCModel    0.0000    ----             ----            Trace switched on
                                                      ExternalEvent 'TraceOff#1' scheduled at 100.0000
                                                      schedules 'TraceOff#1' at 100.0000.
                                                      ExternalEvent 'Simulation stopped#1' scheduled at 1500.0000
                                                      activates 'Van Carrier#1' immediately (NOW)
                                                      activates 'Van Carrier#2' immediately (NOW)
                                      Van Carrier#2   passivates
                                      Van Carrier#1   passivates
           0.9546    TruckArrival#1   ----            activates 'Truck#1' after 'TruckArrival#1'
                     ----             Truck#1         TruckQueuelength: 1
                                                      activates 'Van Carrier#2' after 'Truck#1'
                                                      passivates
                                      Van Carrier#2   holds for 3.1711 until 4.1257
           1.0507    TruckArrival#1   ----            activates 'Truck#2' after 'TruckArrival#1'
                     ----             Truck#2         TruckQueuelength: 1
                                                      activates 'Van Carrier#1' after 'Truck#2'
                                                      passivates
                                      Van Carrier#1   holds for 6.4328 until 7.4836
           4.1257                     Van Carrier#2   activates 'Truck#1' now
                                                      passivates
file:/tmp/./VCModelExperiment_trace.html
```

Figure 2.27.: HTML trace file of the container simulation – rendered in the DESMO-J GUI.

Related to simulation tools, there are plenty of more relevant aspects which assist comparability, those criteria and some other simulation tools are introduced in [PK05], chapter 9. A long list of available simulation tools, engines and languages is given on `http://www.idsia.ch/~andrea/sim/simtools.html` (last checked: Jan 2012).

2.6. Simulation of Computer and Cluster Systems

While generic frameworks, such as those referenced in the last section, allow modeling of many different types of systems, in this section an small excerpt of domain-specific tools for simulation of computer and cluster systems are introduced in more detail. A few tools which are just peripherally relevant are given in brief at the end of this section.

Simulation as introduced in Section 2.5 depends on a model built with a particular goal in mind. In the context of this thesis, we are interested in the simulation of cluster hardware and the execution of a parallel program. One question that should be answered in this thesis is whether an application access pattern is able to utilize the deployed parallel file system for a given cluster file system. In this sense, the simulation could assist in spotting bottlenecks in the real cluster; thus it could explain the observation, and it could predict performance of the parallel file system for arbitrary hardware. Key characteristics of the real hardware and software should be represented in the simulated model to allow to conduct experiments on optimization of parallel middleware and I/O behavior. Since mathematical models are not suitable to model behavior of the dynamics of those systems, discrete-event simulation is the preferable tool.

To compare the simulation tools, the criteria from Section 2.5.3 are extended and concretized slightly towards the application domain. In the system model, additionally the abstractions for relevant cluster components are given, effectively splitting the model into CPU, node, memory, network and I/O models. In the context of the simulator, core hints to the scalability of the solution are provided. The application model is part of the activity model, in case other concepts such as the application trigger activities by themselves – e.g., autonomously acting (active) storage, this fact is mentioned explicitly. It is required that simulation experiments match the intentions of the specific simulator, otherwise the provided model might lead to an insufficient representation of the aspects that are relevant for the experiment – thus leading to a low accuracy. During the validation, typical accuracy for the mentioned simulators is in the order of several percent, it is not uncommon to see a difference of 5 % in performance for simulation of network and application behavior.

DiskSim simulates the behavior of block I/O in a fine grained manner [BSS+10]. It has been maintained and extended since 1993 and is now available in version 4.

- **Environment**: The storage simulator is written in C and provided under a BSD-alike license. A simulation library and a library to access storage models are provided. The *fast lexical analyzer generator* (Flex) and the GNU parser generator *Bison* are used to build parsers for the model descriptions. The simulator can be run under Linux and on the Windows platform.

  Dixtrac, a supplementary tool, extracts characteristics from a real HDD. Those parameters can be loaded into DiskSim to characterize the behavior of an HDD closer to reality.

- **System model**: DiskSim simulates a storage subsystem, components not involved in low-level storage are not modeled at all. Simulated devices are device drivers, controllers, bus systems and the underlying block oriented storage device such as an HDD.

  Topologies of components can be specified in a simple textual representation, exactly one device driver connects to one or two controllers, a controller connects storage devices in a hierarchy of system buses. Orthogonal software services such as I/O scheduling and disk caching are supported. Components are instantiated as defined by the input component specification.

  The description of the simulated components is provided next. A *bus* is either occupied exclusively – e.g., only one component can acquire the bus and then transfer data, or it shares available bandwidth equally among all pending transfers. Modeled characteristics include: time to acquire the bus, transfer time to read a block and transfer time to write a block. The time to transfer a block over a bus between two components is the maximum of the block transfer times of the components.

  Controllers have an I/O scheduler and a caching strategy, and include parameters for the maximum queue length and block transfer time.

  Disk devices can use either a simple or an accurate device model. The accurate HDD model considers the mechanics in the device and bus communication realistically and thus the actual position of the actuator on the platter and position of the target block determines the duration of data access. There are 65 parameters to configure a device in the accurate model. In contrast, the simple model does not track the accessed blocks, instead it is parameterized by: the time to transfer a block, a constant access time and a bus latency.

  Twenty-seven *schedulers* have been implemented and evaluated in DiskSim – ranging from *First-Come-First-Served* over *VSCAN* to experimental algorithms. In order to schedule the appropriate operations, a scheduler can have access to the cylinder mapping strategy, i.e., the mapping of LBA to physical location. Within a sequence of requests, a read request might overlap completely with a previously scheduled request, in that case caching mechanisms provide the data for the second request.

  The *block cache* of a HDD simulates many aspects among which write-through/write back, pre-fetching, flushing and replacement policy are controllable.

  Further features permit to apply RAID schemes to storage devices, also the rotation of a set of devices can be synchronized (spindle synchronization). Further, an I/O device can be configured to act as a cache for another device.

  Since 2008 there is a patch available which supports SSDs. The patch considers a multitude of characteristics of flash storage including the hierarchical design of flash packages, lanes and (erasure) blocks. Just to give an impression of the complexity of such a model, in total, 5131 lines of C source code are required. Also, a model for *Microelectromechanical systems* (MEMS) [CGN00] is shipped with DiskSim. In this model, energy consumption of I/O is considered.

- **Activity model**: Either recorded trace files or synthetic workloads can drive the modelled I/O subsystem. To replay a trace file, the file contents is read on demand. In a simple case, the trace consists of a sequence of requests, each characterized by arrival time, access type, the device to access, the block number and the request size in blocks.

  With a synthetic workload, the logical block number and the access size for the next I/O operation are determined randomly with choosable probability distributions, for example with a normal, exponential or poisson distributed random variable. Multiple synthetic workloads can be executed to simulate concurrent "applications".

- **Simulation core**: DiskSim is a sequential program with supports application level checkpointing. By injecting a specific event during the run, a checkpoint of the simulation state is requested and created. Experiments are configured with a parameter file, specifying the storage subsystem and further simulation parameters.

- **Report generation**: During the experiment DiskSim collects statistics for the storage components. For each component, an individual set of statistics is maintained. For example more than 30 metrics are available for the HDD model. A warm-up period can be specified, during which data collection of values is not performed.

File System Simulation (FSS). In the PhD thesis [She99], a file system simulator is introduced, unfortunately it has been abandoned after the PhD thesis finished. It seems the source code is not accessible in the WWW any more and thus all the descriptions are based on the thesis.

- **Environment**: The file system simulation, which is distributed under the GNU GPL, uses OMNet++ as backend.

- **System model**: Provided layers are discussed from top (application) to bottom. The *disk request regenerator* simulates application behavior by creating requests to an IO library. Interfaces for an IO library, system call interface and file system are defined, yet they just pass the operations without implementing a realistic behavior. Usually, in the file-system layer, the file to block mapping is performed, according to the description, the current implementation is a simple layer mapping blocks sequentially.

  There is support for a *block cache* which orchestrates write-back of dirty pages; implemented caches include *FIFO*, *LRU* and and *Fair Share* algorithms.

  The *disk driver* layer should translate the logical block numbers to physical disk geometry, however, the models take block numbers directly. Thus, operations are passed through the layer without modifications. Another variation of the layer realizes a mutex, which queues pending operations and dispatches just one operation at a given time.

  A *disk scheduler* queues outstanding requests and schedules them according to its policy – among the supported algorithms are FIFO, CScan and Preemptive Fair Share. The layer handles cases in which completion of a request satisfies multiple pending requests. For instance, blocks read could be requested by various pending read requests. Multiple writes to the same block only persist the data written last. A pending write and read operation of one block can be satisfied from cache, even if the write has not finish.

  The *physical disk* model is either a simple fixed access time model, or an implementation of the *HP 97560 Disk* model based on previous descriptions in [RW94]. This advanced disk model respects seek-time including head-positioning, rotation position of the platters, disk cache and read-ahead capability.

- **Activity model**: To generate requests, FSS supports either a sequence of read and write requests supplied by a trace file, or one out of three synthetic workloads: One is a random request generator. Another one simulates an application that reads/updates a 2D clipping of a larger map and that also records some log file. The last synthetic workload represents a network printer, which spools a remote job onto the disk while a printing engine reads pages when the printer is idle.

- **Simulation core**: FSS utilizes OMNet++ and thus parallel simulation might be possible.
- **Report generation**: The simulator gathers statistical information of all layers and includes request time, seek distance, access time, cache hit/miss ratio and wait time.

SimSANs This simulator models storage area networks (SANs) and their administration.

- **Environment**: SimSANs is built with the OMNet++ framework. Additional GUIs support creation of the model and analysis. Recent versions require the Windows platform and the *.NET* framework because the GUIs are coded in *C#*. The source code is not available for the public.
- **System model**: In the system model, hosts access storage devices via the Fibre Channel protocol and address them with their *logical unit number* (LUN). Besides simulation of the Fibre Channel technology, SimSANs supports the tunneled *Fibre Channel over Ethernet* (FCoE) protocol.

  Each LUN is accessed exclusively by one host. A *host* has a number of equal processors characterized by cache, processing speed and front-side bus; the host defines a SCSI timeout and device queue depth[61]. *Network adapters* connect the host via a switched topology to the Fibre Channel endpoints.

  Data rate, MTU, frame payload size and receive buffer count are attributes of a *port*. Ports are also part of a *switch*. Virtual ports are supported as well.

  Storage devices are specified by a range for the sequential and for the random response time; the range is defined by a minimum and a maximum access time and a probability distribution.
- **Activity model**: I/O traffic is injected by assigning at least one *I/O generator* to a host. An I/O generator creates requests randomly; it is characterized by the read ratio, the sequential ratio, the maximum number of outstanding operations, the amount of data requested per operation, burst length and burst delay.

  With SimSANs, many parameters of the system configuration can be adjusted at run-time; one can even adjust the system by adding new hosts and I/O generators.
- **Simulation core**: The simulation tool is coded in OMNet++. The author has not found further information about the simulation core.
- **Report generation**: The GUI can display and assess Fibre Channel packets in the way common protocol analyzers, like *Wireshark*[62], capture and show real protocol traffic. In fact, traces can be exported into the *Wireshark* file format and analyzed directly in the network analyzer. Additionally, the GUI can be configured to draw performance charts for each host and SCSI device.

BigSim [ZWJK05, ZKK04, WZB+05] is an environment consisting of the machine emulator *BigEmulator* and the trace-based simulation tool *BigSimulator*. BigSim was initially designed to assess performance of a BlueGene/C system.

- **Environment**: Both tools are written in Charm++ and are offered under an open source license, BigEmulator is shipped within the tarball of Charm++. Compile scripts for many UNIX platforms and Windows are enclosed. The BigSimulator depends on the POSE framework (also included in the distribution of Charm++).

  The emulator realizes the execution-driven approach, thus it runs existing applications and estimates communication time. The BigSimulator loads and replays trace files generated by the emulator. A

[61] The maximum number of pending requests. A pending request is one that has been issued to the SCSI device but is yet unfinished.

[62] `http://http://www.wireshark.org/`

tool permits to adjust computation time of the trace files, e.g., to simulate compute nodes that are twice as fast as the original ones.

- **System model**: BigEmulator and BigSimulator only support homogeneous architectures. Parameters of a simulation experiment are defined in a simple text file, specified parameters are used for all model components. The emulator model consists of interconnected single chip nodes, a chip has a number of threads each supplied with an integer execution unit.

  BigEmulator computes communication time for point-to-point and broadcast operations while the computation is performed on a real system. For communication, it uses an algorithmic network model – performance depends on an initial latency, the latency per hop and the bandwidth. The distance between two nodes depends on the network topology, congestion effects are not addressed. Several topologies are supported: 2D/3D grids and tori, a fully meshed network, or machine-specific routing algorithms on a 2D/3D node arrangement.

  With BigSimulator, either a simple algorithmic model or a network contention model can be used in experiments. The simple model computes the communication time from a latency, bandwidth, and optionally, with an overhead per packet, it does not depend on the actual network topology. The network contention model is made of links, switches and network interfaces. Arbitrary topologies are supported, several models like 3D grids, tree topologies or hypercubes are already templated. A *compute node* is connected to one *NIC* which fragments messages into packets. The NIC is characterized by two delay values representing DMA overhead – one for small and one for large messages. Also, latency and bandwidth attributes are supported.

  Store-and-forward *switches* route the packets with one of the available strategies, switches buffer packets either on the incoming or the outgoing *port*. The routing algorithm can be static or adaptive; in the latter case, the outgoing port with the lowest load is chosen.

  Modeled switches perform multicast, barrier and reduction operations in hardware. Supported MPI operations are send, receive, barrier and allreduce. Communicators are not modeled. Startup of a barrier operation adds a delay, which is hard-coded and depends on the number of processes per node.

- **Activity model**: Real applications can be linked with the BigEmulator to mimic the BlueGene scheduling and communication layer – including 3D grid topology. Supported programming models are Charm++ and MPI 1.0, a BlueGene low-level API for the emulator is also supplied. The emulator is initialized with the dimensions of the 3D torus and the number of communication and computation threads per node – these parameters are specified via command line. Or, if the low-level emulation API is used, values are set in initialization functions embedded in the application. While the computation is executed on the available processors, the virtual time is managed by the emulator. When the simulation finishes, the model time is output thus estimating performance for a real BlueGene system.

  During execution, the emulator can record activity in a trace file. To predict performance of non-existent machines and architectures, a tool is supplied which alters the compute time of recorded functions individually.

  Three models are supported to adjust computation time: first, all times can be scaled by a fixed factor, hence modeling a faster or slower processor. Also, execution time for a function call can be estimated by an equation that computes the complexity of the function based on the supplied parameters. Also, the tool can extrapolate measured timing information automatically with the least squares method. At last, BigEmulator provides an API to instrument the compute kernels; for each execution, timing and hardware performance counters such as floating-point, integer and branch instructions are recorded and can be used in the extrapolation tool.

  At last, cycle-accurate simulation of the routines for a target system can be combined with parameterized complexity of a routine. Alternatively, hardware performance counters on the traced machine can be extrapolated to estimate the speed for the target machine's execution units and its cache performance.

BigSimulator additionally provides some stochastic generators for network traffic.

- **Simulation core**: Parallel simulation is supported by utilizing capabilities of POSE. By running on 100 physical processors, BigSim demonstrated parallel simulation of machines with 100.000 processors.

- **Report generation**: The simulator can gather statistics of link utilization and message arrival. Also, both simulator and emulator can record activity in trace files. A command line tool extracts communication information of trace files and prints statistics.

  Charm++ ships with the *Projections* analysis framework for those trace files. This framework consists of a trace writer API, its implementation and a Java viewer. Multiple views are provided: profiles, histograms and timelines, a utilization graph shows the CPU usage of the individual logical processors, and the communication activity (number of messages, messages bytes ...) for all remote methods can be stacked for each processor. Further, the aggregated communication activity of all processors over time, a 2D visualization of processor usage – colors encode the utilization at any given time, clustering functionality to spot outliers in processor utilization via a k-means clustering algorithm, and at last a noise detection tool to identify OS or computational noise in the data.

SIMCAN The *SIMCAN* [NFG+10] simulation project aims to simulate large storage networks in order to predict performance and scalability. SIMCAN has been developed in parallel to this thesis – both projects started at the same time. Recently, we found this interesting project when updating the bibliography and references. Although there are similar features as those developed in this thesis, the approaches are different; components of PIOsimHD operate on a higher level of abstraction and focus on simulating alternative MPI algorithms. The reported characteristics represent the status of the project as of April 2011.

- **Environment**: SIMCAN is based on OMNet++. The *INET* framework is utilized for simulation of communication networks. INET models network technology and protocols including Ethernet and TCP/IP.

  By itself, SIMCAN is licensed under the GPL, INET is licensed under GPL or LGPL. However, OMNet++ by itself is provided with the academic private license that prevents commercial usage free of charge. SIMCAN is available for Windows and Linux.

- **System model**: Modeled system components are largely along the lines of cluster systems: racks, nodes, operating system, CPU, file system layer, communication and services represent their real systems. The abstraction level of a component can be varied. The developers explicitly mention simple stochastic models and realistic implementations. While the stochastic models are easier to understand, the realistic models provide more accurate and comparable results.

  Modeled components are as follows: A *rack* consists of a number of node boards each with an equal number of nodes. Each *node board* contains a number of nodes and interconnects these nodes (characterized by a data rate and latency). Nodes have a number of CPUs of a given type (attributed by instructions per second and core count) an OS layer with several I/O layers and, optionally, a local storage device. *CPUs* schedule computation either in FIFO, or with a Round Robin scheme in which each job is processed for a time slot – called quantum – before the next pending job is dispatched.

  Memory can be allocated by the application and is used for buffer space; messages can allocate or release memory. The access time is modeled as a function with the access size as a parameter – when more data is accessed, time increases.

  A *file system* maps the logical block of an operation its physical block address; there are three models provided that define the processing time of metadata operations and block I/O. In a simple model, the latency to open, create or close a file is hard-coded and a straight forward mapping of bytes from a file to disk is realized. An advanced model for ReiserFS and Ext2 distributes the file across the disk similar to these file systems. There is a parallel file system model without explicit metadata servers, the model uses NFS servers to maintain data. In this implementation, metadata is considered to be part of the file's data stream, i.e., the first few KiB represent metadata. Metadata operations involve communication with all servers: open and delete require one message exchange per server – open reads the metadata. Create needs two messages; with the first, the file is created, and with the next request, metadata is written to the file. Block I/O is distributed with a RAID 0 concept among the servers.

NFS servers operate on block-level and realize a thin layer relaying operations between operating system and requester. The *OS* contains a cache layer and it controls modules for the I/O – right now, only FIFO scheduling is supported. Data blocks of files can be cached; one memory model provides write-behind and read-ahead capabilities.

Implemented *disk* models are: a simple model – characterized by latency and transfer time for read and write, respectively. The detailed disk model contains tables with the measured throughput for read and write access in relation to the distance of the access. Therefore, it tracks the last accessed offset; when an access shall be performed, the knowledge of the next offset allows picking the right elements of the table. Since the values in the table are samples, they are linearly interpolated to match the actual distance and size implied by the request. There is no scheduling algorithm implemented for the disk model.

Nodes are interconnected with a switched network topology provided by the *INET* framework. INET models Ethernet switches and communications protocols in high detail. High-level protocols, such as TCP, are fragmented into frames of the lower MAC layer and are communicated with store-and-forward switching. *Switches* have a number of CPUs relaying the frames between the ports and a buffer for frames, processing of a frame takes some time. If this buffer is full then frames are dropped.

- **Activity model**: A node hosts client or server processes. All workload is generated by simulated client applications. On the client-side, there are several modules provided for independent or sequential programs, and for parallel MPI applications. To be more specific: an NFS client – receiving I/O requests via TCP and generating NFS requests, a master/server MPI-IO application, and replay engines for sequential and parallel applications. In their paper, a PMPI wrapper has been presented which intercepts MPI-IO calls to record the activity, those activity can be later replayed. Computation time is simulated by waiting the time between MPI calls. The sequential replay engine supports open, close, delete, read and write calls.

  Collective MPI calls are supported to gather, scatter, send and receive data; for synchronization, the `MPI_barrier()` is provided. The MPI routines to open, close, delete, read and write files are provided. A drawback of the implemented functions is that for each function, only the naive implementation is provided, that means all communication of collective operations happens between the root of the collective and all other nodes. Communicators are not supported, also all implementations are hosted in one file. Each processing step of a complex operation, for example, to wait for receiving a message, is implemented in its own method.

- **Simulation core**: Although OMNet++ is used, the simulation core is executed sequentially.
- **Report generation**: There is no explicit report generation provided, the MPI replay module maintains additional statistics about the time spent in computation, IO and communication.

Scalable Simulation Framework (SSF) [LNBG03] is a process-oriented discrete-event simulation framework which is mainly used to simulate large networks – SSFNet aims to simulate Internet traffic.

- **Environment**: This project has been abandoned in 2004, although the proposed API of SSF has been implemented by several third parties in C++ and Java, for example in the *Darthmouth Scalable*

Simulation Framework (DaSSF); further commercial implementations of the API for Java and C++ exist.

- **System model**: The model is specified in their *Domain Modeling Language* (DML). This language is a public-domain standard. DML can be thought of as an XML-like language with features to support inheritance; parameters are specified in textual form in hierarchically organized key-value pairs.

  SSFNet simulates protocols and networks based on the Internet Protocol (or above). Models for several protocols such as IP, UDP, TCP, BGP[63] closely resemble the real behavior of an IP network. The basic network model consists of hosts, routers, NICs and links. NICs are characterized by bitrate and latency, a link is characterized by latency. Internally, these components are implemented in a process-oriented scheme. Networks can be arbitrarily nested and connected by routers to form autonomous systems. Layer 2 switches are supported, but network contention seems not to be addressed by a switch.

- **Activity model**: Activity is specified in so-called *protocol graphs* and *protocol sessions*, each individual connection starts a protocol session controlling message exchange.

  Several protocol sessions to generate workloads are provided, they can be instantiated and parameterized in DML. Common application schemes include: HTTP and TCP client & server – the TCP servers accepts client sessions and the TCP test client sends a fixed amount of data. HTTP client activity is simulated with probability functions. The communication partner, that is, the server network interface and port, for a client is defined in another section of the DML; upon initialization of the model, SSFNet assigns IP addresses automatically.

- **Simulation core**: Parallel execution on shared memory machines is anticipated with the API specification. However, the model developer must indicate the dependency of entities in the DML of the model specification. Otherwise the partitioning is unclear and the model runs on just one processor.

- **Report generation**: SSFNet provides monitoring extensions to capture and visualize all states of the network activity. For instance, TCP or IP packets can be dumped by all components – similar to the protocol analyzer *Wireshark*. Also, the queue length of pending packets in routers can be output periodically, and an animation of traffic is provided in a 2D representation of the network model. To start the recording of activity, in the DML code of the model, probes and monitors must be defined.

Structural Simulation Toolkit The SST [RHB+11, HRR+11] is a recent attempt to co-design hardware structure and software technologies. Besides performance, SST estimates parameters for energy consumption. Recently it has been used to evaluate temperature and space requirements of a processor, the authors envision to estimate costs and reliability as well. This project has been developed in parallel to this thesis.

- **Environment**: SST is coded in C++ and generously uses the *Boost* libraries. It works on the Linux and Unix platforms. The code is provided under a BSD license. Native MPI applications can be executed inside the simulated environment.

- **System model**: SST provides a modular architecture to simulate memory hierarchies, bus, computation, network and I/O at arbitrary levels of abstraction. On one hand, it provides cycle-accurate simulation, on the other hand, abstract analytical models are supported. Coarse-grained simulation with *SST/macro* [JAC+10] estimates performance of large scale systems while fine-grained cycle-accurate simulation aims to understand system details. Especially for fine-grained simulation, SST incorporates and interfaces many existing simulators to avoid redundant development. In the modular concept, each component has its own local virtual time and synchronizes with other components to exchange events and update its local time.

  For the cycle-accurate simulation, some integrated simulators are mentioned briefly[64]. DiskSim is

[63] Routers use the *Border Gateway Protocol* to exchange routing information.
[64] Although many simulators are mentioned in the context of SST, only a few are provided in the source code distribution.

adjusted and incorporated into the source tree to estimate single disk I/O. However, currently, parallel I/O is not supported.

The compute model permits execution of native programs (with small modifications) in the simulation environment. Both the activity and system models can be chosen independently. Execution of native programs is handled by one of the following simulators. The hardware processor framework *Multicore Power, Area, and Timing* (McPAT) [LAS+09] simulates the microarchitecture of many-core processors from 90 nm to 22 nm including caches and execution units. Due to the high level of detail of this simulation, McPAT permits assessing of energy characteristics. Alternatively, Monte Carlo models for the *Niagara* or *Opteron* processor quantitatively resemble the microarchitecture. A stochastic model does not maintain the complete virtual state of a processor; instead, the activity of instructions depends on the probability distribution, e.g., a load instruction is satisfied by the L1 cache in 50% of the cases. Also, with *genericProc*, an adjusted version of the *SimpleScalar* open source computer architecture simulator is provided. SimpleScalar models a complete virtual computer system executing the *PowerPC instruction set architecture* (ISA). Finally, *QSim* is mentioned in the SST documentation, it acts as a front-end for the *QEMU* processor emulation that emulates the x86 ISA.

In SST/macro, a processor executes jobs characterized by a number of load/store instructions, which are interpreted as memory operations, and the number of floating point operations to be executed.

Memory simulation is supported by *DRAMSim2*, which models a complete DDR2 (or DDR3) memory subsystem including controller and DRAM devices.

The network simulation contains models for the *Portals* API[BMRH02], the *SeaStar* network and *HyperTransport*. Network attributes are primarily latency and bandwidth; in the SeaStar network, the queue depths and packet flow can be adjusted as well. According to the documentation, their network congestion model shares links among the data streams – the available bandwidth is allocated in the order the communication is started. To be more precise, the oldest flow operates at maximum possible bandwidth, then the remaining bandwidth is available for the next flow, until all flows are handled. One router model, that is provided with the source distribution, models congestion by queueing messages and scheduling them in FIFO order. This model characterizes the performance by its bandwidth and a router delay that represents the costs for an additional hop.

Templates for network topologies allow simulation of large networks. Predefined topologies are supplied, e.g., 3D meshes, binary fat tree, hypercube or a fully connected graph. A command line tool eases creation of large networks.

An experiment is set up in an XML file that describes the components and parameters, and the links between the components.

- **Activity model**: Two ways to execute applications are supported, either a native executable is loaded into the virtual environment and started, or a replay mechanism executes MPI calls according to the trace information. The former scheme is used to prototype application behavior in skeletons that use MPI. In the latter, *DUMPI*, an MPI trace library, intercepts and records MPI function calls and their parameters for a replay within *SST/macro*. To evaluate network architectures, there are also components which inject communication patterns on the network layer to avoid time-consuming simulation of processes.

  Regarding MPI implementations, in the SST distribution a subset of MPI functions are realized: currently `MPI_Bcast()`, `MPI_Reduce()`, `MPI_Allreduce()` and `MPI_Barrier()` are implemented. In the SST/macro package, many more functions are implemented with a naive communication pattern. For a few collective operations, sophisticated collective algorithms like Rabenseifner's reduce algorithm[Rab04] are provided.

- **Simulation core**: Parallel simulation is supported by using MPI. A conservative synchronization occurs between the processors – the minimum latency of a (simulated) link between two components determines the synchronization frequency. Static load balancing is done by partitioning the components into groups which have many links between them. Every group is then mapped onto a logical processor. This scheme also maximizes the latency between components placed on disjoint processors. Checkpoint/restart of the simulation is implemented by serializing the state of all components into a file in configurable intervals.

- **Report generation**: Supported simulators output their statistics independently, e.g., DiskSim maintains the total number of operations, the average service time and standard deviation, and DRAMSim2 outputs the energy consumption. The genericProc simulation module can record the activity of the simulated application in text files; among the traced activities are MPI point-to-point communication, DMA, network activity and internal operations.

  In SST, derived components and several connected simulators can disclose named statistics and other data. Data collection can be triggered by events, or data can be queried by other components explicitly. Supported metrics, for instance, are: core temperature, L2 cache reads and floating point operations (supplied by McPAT). For a router, the number of processed messages and the aggregated delay of all communication are supported. Energy consumption and power information are monitored for DRAM, router and processor.

LogGOPSim The *LogGOPSim* [HSL10] is a recent project for simulating the LogGPS performance model of homogeneous machines. This project has been developed in parallel to this thesis.

- **Environment**: LogGOPSim is coded in C++ and utilizes several existing libraries: The *graphviz* library to draw simple communication timelines, the scanner generator `re2c`, and several projects hosted on the university of Illinois. There is no licensing information provided.

- **System model**: The LogP model family supports analytical models with parameters for the maximum latency of two processors, the CPU overhead per message, the costs per byte of a message. It distinguishes between eager and rendezvous sends – messages with a size below a threshold set as a parameter are performed in eager mode. In the paper written by Hoefflers et.al., the LogGP model is extended to allow overlapping of communication and computation, this model is implemented in the simulator. Note that all components share the same characteristics.

  Topologies of switches and NICs can be created and read from a file in the annotated *DOT format*[65]. Besides the analytical model based on LogGP, a simple congestion scheme is offered. In the implementation, the startup and completion of a message update all messages routed over the same path and change the congestion level – the bandwidth is shared among all messages equally. The simulator processes 1 byte of the message per time-step of the simulation – the model parameters are not yet supported. I/O is not addressed in the simulator.

- **Activity model**: Three basic operations are supported: Sending and receiving of messages with a matching close to MPI, and computation. In the latter activity, time is defined which elapses on the selected processor. In this process, a random time can be added to simulate operating system noise. A recorded noise profile can be injected as well to study the impact of OS jitter.

  Application communication behavior is supplied to the simulator in a file of the *Group Operation Assembly Language* (GOAL), an example for a binary tree between four processors is given in Listing 2.10. Every operation is labeled and dependencies between the operations are explicitly coded with the `requires` keyword. To simulate more complex MPI operations, these operations must be translated into a sequence of send and receive primitives. The simulator is shipped with some basic communication patterns for MPI collectives, such as linear gather, binomial tree patterns and the dissemination pattern.

  MPI application profiling traces can be recorded and fed into the simulator. The tracing library offered by Torsten Hoeffler intercepts PMPI calls and records the activity in a simple text format. Then, this format can be converted into the GOAL input schedule for the simulator. During this procedure,

[65]`http://www.graphviz.org/content/dot-language`

collective operations must be translated into a sequence of elementary send, receive and compute operations. Currently, only collective operations running on all processes, e.g., MPI_COMM_WORLD, are supported. Also, the way MPI operations are translated is fixed in the provided *Schedgen* tool. That means, for instance, that `MPI_Allreduce()` operations create a dissemination pattern.

Listing 2.10: GOAL example input for LogGOPSim

```
num_ranks 4

rank 0 {
l0: calc 1000 cpu 0
l1: send 1b to 1 tag 0
l2: send 1b to 2 tag 0
}

rank 1 {
l1: recv 1b from 0 tag 0
l2: send 1b to 3 tag 0
l2 requires l1
}

rank 2 {
l1: recv 1b from 0 tag 0
l2: calc 1000 cpu 0
l2 requires l1
}

rank 3 {
l1: recv 1b from 1 tag 0
}
```

- **Simulation core**: The simulator is a sequential program. Due to its fast execution – 1 million events per second, the simulator has been successfully used to study existing applications. With memory-mapping techniques, the out-of-core execution of a GOAL input file is possible [HSL10].
- **Report generation**: The simulator outputs the virtual end-time for the simulation and the model parameters of LogGOPS. Optionally, the simulated end-time of every process can be printed. The output for the schedule in Listing 2.10 is given in 2.11

  For small simulations, a trace file can be created which is then visualized with the help of *Graphviz*. The output for our example schedule is illustrated in Figure 2.28. Due to the default settings of Graphviz, the behavior and text is barely readable in this example. Since the overlapping parameter O is set to 0, the computation and communication are not overlapped.

Listing 2.11: Output of LogGOPSim for the GOAL input in Listing 2.10

```
LogGP network backend; size: 4 (1 CPUs, 1 NICs); L=2500, o=1500 g=1000, G=6, O
  →=0, P=4, S=65535
PERFORMANCE: Processes: 4        Events: 11      Time: 0 s       Speed: inf ev/
  →s
Times:
Host 0: 4000
Host 1: 7000
Host 2: 8000
Host 3: 11000
```

Other tools A few more simulation tools with peripheral relevance to this thesis are introduced in brief.

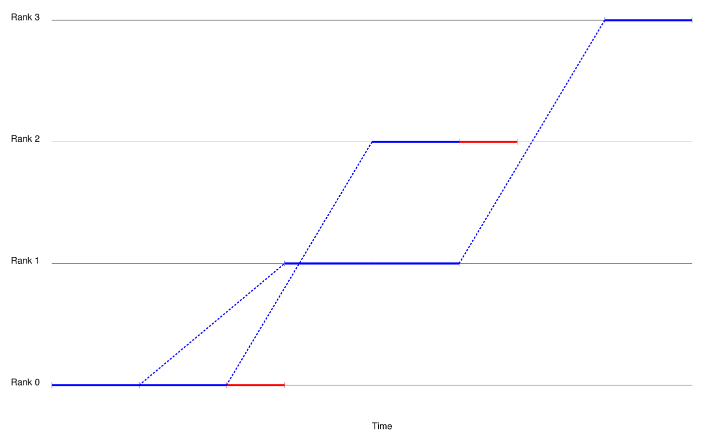


Figure 2.28.: Visualized binary tree pattern for four processes as created by Graphviz.

JitSim [DM10] simulates the influence of jitter in large systems. The application behavior is modeled as a sequence of computation, communication and jitter phases. Jitter is either traced on a real system and replayed in the simulator, or synthetically created. Synchronization points of the program are simulated by passing send and receive messages in a tree topology. Currently, the barrier is the only synchronization primitive supported. The system model defines latency between cores, sockets and nodes. To be able to simulate very large systems, homogeneity of the systems is assumed and network congestion is ignored.

N-MAP [FJ95], as mentioned in Section 2.4.1, is an early discrete-event simulation tool which predicts performance of SPMD programs. From a program specification, that is, a sketched communication and execution scheme, the tool generates either a simulation program – which upon execution writes a simulation trace, or a skeleton for an instrumented real program, which writes a trace file during run-time. In their workflow, assessment of the obtained performance leads to an iterative refinement and improvement of the specification. The program is specified in a C-like dialect, which is transformed into a C simulator or executable. The tool is not specifically designed for MPI, yet the programmer uses some kind of simplified message passing interface.

PS [AMMV98] is a trace-driven PVM simulator. Simulation runs are analyzed for instance with the visualizing tool *ParaGraph*. Further, in some cases, traces can be derived from source code by using static code analysis. In their system model, the relative processing speed of each processor is specified, the model considers one PVM application, the hosts running the applications, and NICs communicating via TCP/IP.

In [BDCW91], *PROTEUS*, an execution-driven simulator for MIMD processors is presented. PROTEUS executes instrumented message-passing code or shared-memory code – the code calls the simulator which processes mode-time accordingly. The system model includes nodes, one processor, a network chip, cache and memory. The sequential simulator offers multiple implementations for memory and network. Depending on the accuracy and performance requirements, the user can pick the right implementation. Shared-memory as well as distributed memory machines are supported. Independent nodes are connected, either via a direct or an indirect network. Network congestion is addressed, but the user can select an analytical model for the network as well. Programs are written in a C dialect, extensions to C include ways for declaring data to reside in shared memory and controlling placement of data structures. Further, library routines for message passing, thread management and memory management are provided. A GUI assists to define the architectural characteristics, then an application and an architecture-specific simulator are generated and compiled. Running the executable simulator generates some output and a trace file, which can be analyzed with another GUI. Real applications, which do not invoke the simulator, can be created from the C dialect. Thus, production runs with designed programs can be made.

The *Performance Prophet* [PBXB08] utilizes the CSIM engine and simulates message-passing applications. Activity of an application is specified in the UML language and translated into C++ simulation code. Possible message passing activity includes the following operations: blocking and non-blocking send, blocking receive, broadcast, barrier and OpenMP-like parallel regions. The simulation generates a trace file, which can be assessed in the Java GUI *Teuta*. This GUI also supports the workflow to generate the application model. In the model, a cost function is assigned to sequentially executed code blocks, this in turn determines the run-time. System parameters include the number of processors per node, the number of nodes, and the processes and threads per node. It seems that network activity is modeled mathematically, and the model does not take network congestion into account.

Several other approaches to predict performance of parallel computing systems are summarized in [PBB07]. Although the simulation of supercomputers recently gained inertia since it fosters co-design of exascale machines, none of the available tools simulate MPI-IO applications as offered by *PIOsimHD*, the simulator designed for this thesis. PIOsimHD handles parallel I/O activity and MPI application behavior in software, and allows concurrent execution of multiple parallel applications on heterogeneous cluster systems. With these features, PIOsimHD fosters parallel I/O analysis and assessment of alternative MPI collectives.

2.7. Chapter Summary

This extensive chapter introduced related topics and presented the relevant state of the art. On the one hand, with this knowledge, the novel scientific aspects of this thesis will become clear, on the other hand, results presented can be assessed better.

Some background to parallel file systems and their deployment was presented; this included important architectures of vendors. Since the Parallel Virtual File System is used as a reference to compare simulation results, an in-depth introduction to PVFS as an archetype for parallel file systems was given. Later, this knowledge will assists in validating the I/O model of the simulator. Further, it fosters understanding of measured behavior and observed I/O activity.

Then, aspects were listed that influence performance of parallel applications: Many hardware capabilities have an impact on performance. Mainly they can be attributed to CPU, memory, network and block device. Also, various characteristics for hardware and software on all different layers were explained together with optimizations that mitigate performance shortcomings of the hardware.

Details about the Message Passing Interface recalled its programming model and I/O semantics. The MPI semantics offer a level of freedom to vendors that allows to adjust an implementation to the architecture of a supercomputer. Performance implications of the programming model and the defined semantics were discussed, and related work showed a diversity of approaches that improve communication performance.

The section about performance analysis presented concepts that deal with integrating software engineering into the development process to achieve efficient applications. Those approaches require performance models and often use simulation tools to estimate performance before the application code is completely written. However, the state of the art in high-performance computing is to evaluate application's performance after they are coded. This process is illustrated in the closed loop of performance tuning which explains a general workflow to improve performance of a system: Achieved performance of the current state is measured and analyzed, leading to code modifications. This cycle is repeated until the code efficiency suffices, or further tuning is not worth the effort. Several tools were introduced which assist in analyzing serial and parallel programs in order to localize performance bottlenecks. Most of the introduced tools instrument an application to record information of interest into trace files. These trace files are then analyzed post-mortem.

The last sections highlighted the general concept of modeling and simulating complex systems. During modeling, an abstraction must be found from the real system that should be investigated. Discrete-event simulation implements a system model by maintaining a global model time that is incremented to the next event that should be executed. Therewith, an arbitrary accuracy of model time can be achieved while periods of inactivity are skipped.

A couple of frameworks, which are suitable for simulation, were introduced. The presentation of simulation tools for cluster systems (and I/O paths) showed the rich diversity of previous work. Recently, many new simulation projects were started that aim to estimate performance for exascale installations. However, this thesis can be *distinguished from existing work because it aims to fully simulate MPI-IO behavior on a virtual heterogeneous cluster system.*

In the next chapter, the behavior of our working group's cluster is analyzed. Several of the performance characteristics are determined and the complex interplay of hardware and software factors will be illustrated.

Bibliography

[AB09] M. Anthony and P.L. Bartlett. *Neural Network Learning: Theoretical Foundations.* Cambridge University Press, 2009.

[ABC+06] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: a View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[ACI+09] Nawab Ali, Philip H. Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert B. Ross, Lee Ward, and P. Sadayappan. Scalable I/O Forwarding Framework for High-performance Computing Systems. In *CLUSTER*. IEEE, 2009.

[AHA+05] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of MPI Collective Communication on BlueGene/L Systems. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS, pages 253–262, New York, NY, USA, 2005. ACM.

[AMMV98] Rocco Aversa, Antonino Mazzeo, Nicola Mazzocca, and Umberto Villano. Heterogeneous System Performance Prediction and Analysis Using PS. *IEEE Concurrency*, 6(3):20–29, 1998.

[BCI+10] Pete Beckman, Jason Cope, Kamil Iskra, Sam Lang, Kwan-Liu Ma, Chris Muelder, Robert Ross, and Carmen Sigovan. System Software Instrumentation to Support the Visual Characterization of I/O System Behavior for High-End Computing. Poster, 2010.

[BDCW91] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical report, Cambridge, MA, USA, 1991.

[BFG+04] Steffen Becker, Viktoria Firus, Simon Giesecke, Willi Hasselbring, Sven Overhage, and Ralf H. Reussner. Towards a Generic Framework for Evaluating Component-Based Software Architectures. In *Architekturen, Komponenten, Anwendungen - Proceedings zur 1. Verbundtagung Architekturen, Komponenten, Anwendungen (AKA 2004)*, volume 57 of *GI-Edition of Lecture Notes in Informatics*, pages 163–180. Bonner Köllen Verlag, 2004.

[BGM03] Simonetta Balsamo, Mattia Grosso, and Moreno Marzolla. Towards Simulation-Based Performance Modeling of UML Specifications. Technical report, Dipartimento di Informatica, University Foscari Venezia, 2003.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.*, 14:317–329, November 2000.

[BICG08] Francisco Blas, Florin Isailă, Jesús Carretero, and Thomas Großmann. Implementation and Evaluation of an MPI-IO Interface for GPFS in ROMIO. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 159–166. Springer Berlin / Heidelberg, 2008.

[BKD+10] Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[BKL09] David Buettner, Julian Kunkel, and Thomas Ludwig. Using Non-blocking I/O Operations in High Performance Computing to Reduce Execution Times. In *Proceedings of the 16th European*

PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 134–142, Berlin, Heidelberg, 2009. CSC - IT, Springer-Verlag.

[BMRH02] R. Brightwell, A.B. Maccabe, R. Riesen, and T. Hudson. The Portals 3.2 Message Passing Interface Revision 1.1. *Sandia National Laboratories*, 2002.

[BSS+10] John Bucy, Jiri Schindler, Steve Schlosser, Greg Ganger, and et al. The DiskSim Simulation Environment (v4.0). `http://www.pdl.cmu.edu/DiskSim/`. Last accessed 2010-06-13., 2010.

[BTR03] Daniel Balkanski, Mario Trams, and Wolfgang Rehm. Communication Middleware Systems for Heterogenous Clusters: A Comparative Study. *Cluster Computing, IEEE International Conference on*, 0:504, 2003.

[CCC+03] Avery Ching, Alok Choudhary, Kenin Coloma, Wei-keng Liao, Robert Ross, and William Gropp. Noncontiguous I/O Accesses Through MPI-IO. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, CCGRID, pages 104–, Washington, DC, USA, 2003. IEEE Computer Society.

[CCH+06] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. MPIPP: an Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS, pages 353–360, New York, NY, USA, 2006. ACM.

[CCS+06] Jiannong Cao, Alvin Chan, Yudong Sun, Sajal Das, and Minyi Guo. A Taxonomy of Application Scheduling Tools for High Performance Cluster Computing. *Cluster Computing*, 9:355–371, 2006.

[CGN00] L. Richard Carley, Gregory R. Ganger, and David F. Nagle. MEMS-based Integrated-circuit Mass-storage Systems. *Commun. ACM*, 43:72–80, November 2000.

[CHA+10] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, , and R. Ross. Storage Access Characteristics of Computational Science Applications. Preprint ANL/MCS-P1791-0910, September 2010.

[CoF09] CoFluent Design. The MCSE Methodology – Overview. White Paper, Online, 2009.

[CoF10] CoFluent Design. CoFluent Studio – System-Level Modeling and Simulation Environment. White Paper, Online, Dec 2010.

[Cor04] Microsoft Corporation. *Improving .Net Application Performance and Scalability (Patterns & Practices)*. Microsoft Press, 6 2004.

[CSGF08] Mohamad Chaarawi, Jeffrey Squyres, Edgar Gabriel, and Saber Feki. A Tool for Optimizing Runtime Parameters of Open MPI. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 210–217. Springer Berlin / Heidelberg, 2008.

[CUS01] Lloyd G. Williams Connie U. Smith. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2001.

[DD08] Mathieu Desnoyers and Michel R. Dagenais. LTTng: Tracing Across Execution Layers, from the Hypervisor to User-space. In *Linux Symposium 2008*, page 101, July 2008.

[Des10] Mathieu Desnoyers. [RFC] Common Trace Format Requirements (v1.4). Discussion on a Mailinglist, Oct 2010.

[DM94] Giovanni De Micheli. Computer-Aided Hardware-Software Codesign. *IEEE Micro*, 14:10–16, August 1994.

[DM10] Pradipta De and Vijay Mann. jitSim: A Simulator for Predicting Scalability of Parallel Applications in Presence of OS Jitter. In Pasqua D'Ambra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6271 of *Lecture Notes in Computer Science*, pages 117–130. Springer Berlin / Heidelberg, 2010.

[dyn10] dynaTrace software. Application Performance Managment. `http://www.dynatrace.com/en/`, 2010.

[DZV+97] T. Delaitre, M.J. Zemerly, P. Vekariya, GR Justo, J. Bourgeois, F. Schinkmann, and SC Winter. EDPEPPS: An Environment for the Design and Performance Evaluation of Portable Parallel Software. In *Proceeedings of the 5-th Euromicro Workshop on Parallel and Distributed Processing*, 1997.

[Ebc05] Kemal Ebcioglu. IBM PERCS Project: Hardware-software Co-design of a Future Supercomputer for High Programmer Productivity. Presentation, 2005.

[FDD09] Pierre-Marc Fournier, Mathieu Desnoyers, and Michel R. Dagenais. Combined Tracing of the Kernel and Applications with LTTng. In *Proceedings of the 2009 Linux Symposium*, July 2009.

[FG08] Saber Feki and Edgar Gabriel. Incorporating Historic Knowledge into a Communication Library for Self-Optimizing High Performance Computing Applications. In *Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 265–274, Washington, DC, USA, 2008. IEEE Computer Society.

[FJ95] A. Ferscha and J. Johnson. N-MAP: A Virtual Processor Discrete Event Simulation Tool for Performance Prediction in the CAPSE Environment. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, pages 276–285 vol.2. 1995.

[FJ00] Alois Ferscha and James Johnson. N-MAP – an Environment for the Performance Oriented Development Process of Efficient Distributed Programs. *Future Generation Computer Systems*, 16(6):571 – 584, 2000.

[FM95] Alois Ferscha and Allen Malony. Performance-oriented Development of Irregular, Unstructured and Unbalanced Parallel Applications in the N-MAP Environment. In Heinz Beilner and Falko Bause, editors, *Quantitative Evaluation of Computing and Communication Systems*, volume 977 of *Lecture Notes in Computer Science*, pages 340–356. Springer Berlin / Heidelberg, 1995.

[FMW+11] Greg Franks, Peter Maly, Murray Woodside, Dorina C. Petriu, Alex Hubbard, and Martin Mroz. Layered Queueing Network Solver and Simulator User Manual. Technical report, Department of Systems and Computer Engineering, Carleton University, 2 2011.

[FPY07] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan. A Study of Process Arrival Patterns for MPI Collective Operations. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS, pages 168–179, New York, NY, USA, 2007. ACM.

[Fra00] Roy Gregory Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Ottawa, Canada, Canada, 2000.

[FWP09] Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. Scalable Massively Parallel I/O to Task-local Files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC, pages 17:1–17:11, New York, NY, USA, 2009. ACM.

[FYL06] Ahmad Faraj, Xin Yuan, and David Lowenthal. STAR-MPI: Self Tuned Adaptive Routines for MPI Collective Operations. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS, pages 199–208, New York, NY, USA, 2006. ACM.

[Ger99] N.A. Gershenfeld. *The Nature of Mathematical Modeling*. Cambridge University Press, 1999.

[GFBR10] Edgar Gabriel, Saber Feki, Katharina Benkert, and Michael M. Resch. Towards Performance Portability Through Runtime Adaptation for High-Performance Computing Applications. *Concurrency and Computation: Practice & Experience - International Supercomputing Conference*, 22:2230–2246, November 2010.

[GH07] E. Gabriel and S. Huang. Runtime Optimization of Application Level Communication Patterns. In *International Parallel & Distributed Processing Symposium*, IPDPS, pages 1–8. IEEE, 2007.

[GHKV96] B. Gruber, G. Haring, D. Kranzlmueller, and J. Volkert. Parallel Programming with CAPSE – A Case Study. In *Proceedings of the 4th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, volume 0 of *PDP*, page 130, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[GO10] M. Gerndt and M. Ott. Automatic Performance Analysis with Periscope. *Concurrency and Computation: Practice & Experience – Scalable Tools for High-End Computing*, 22:736–748, April 2010.

[GWW+10] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Abraham, Daniel Becker, and Bernd Mohr. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience*, 22(6):277–288, April 2010.

[Har09] Shai Harmelin. Whitepaper: Isilon IQ Scale-out NAS for High-Performance Applications. Technical report, 2009.

[HH07] Dean Hildebrand and Peter Honeyman. Direct-pNFS: Scalable, Transparent, and Versatile Access to Parallel File Systems. In *Proceedings of the 16th international symposium on High performance distributed computing*, HPDC, pages 199–208, New York, NY, USA, 2007. ACM.

[HMD+10] X. Sharon Hu, Richard C. Murphy, Sudip Dosanjh, Kunle Olukotun, and Stephen Poole. Hardware/software Co-design for High Performance Computing: Challenges and Opportunities. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS, pages 63–64, New York, NY, USA, 2010. ACM.

[HRR+11] Ming-yu Hsieh, Arun Rodrigues, Rolf Riesen, Kevin Thompson, and William Song. A Framework for Architecture-level Power, Area, and Thermal Simulation and its Application to Network-on-chip Design Exploration. *SIGMETRICS – Performance Evaluation Review*, 38:63–68, March 2011.

[HSL10] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim: Simulating Large-Scale Applications in the LogGOPS Model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC, pages 597–604, New York, NY, USA, 2010. ACM.

[HWRI99] Curtis E. Hrischuk, C. Murray Woodside, Jerome A. Rolia, and Rod Iversen. Trace-Based Load Characterization for Generating Performance Software Models. *IEEE Trans. Softw. Eng.*, 25:122–135, January 1999.

[HYC05] Ding-Yong Hong, Ching-Wen You, and Yeh-Ching Chung. An Efficient MPI-IO for Noncontiguous Data Access over InfiniBand. In *Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks*, ISPAN, pages 140–147, Washington, DC, USA, 2005. IEEE Computer Society.

[ITKT00] Toshiyuki Imamura, Yuichi Tsujita, Hiroshi Koide, and Hiroshi Takemiya. An Architecture of Stampi: MPI Library on a Cluster of Parallel Computers. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 200–207, London, UK, 2000. Springer-Verlag.

[JAC+10] Curtis L. Janssen, Helgi Adalsteinsson, Scott Cranford, Joseph P. Kenny, Ali Pinar, David A. Evensky, and Jackson Mayo. A Simulator for Large-scale Parallel Architectures. *International Journal of Parallel and Distributed Systems*, 1(2):57–73, 2010.

[Kae93] David Kaeli. Issues in Trace-driven Simulation. In Lorenzo Donatiello and Randolph Nelson, editors, *Performance Evaluation of Computer and Communication Systems*, volume 729 of *Lecture Notes in Computer Science*, pages 224–244. Springer Berlin / Heidelberg, 1993.

[KBB+06] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang Nagel. Introducing the Open Trace Format (OTF). In Vassil Alexandrov, Geert van Albada, Peter Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3992 of *Lecture Notes in Computer Science*, pages 526–533. Springer Berlin / Heidelberg, 2006.

[KBB+08] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, and R. Stanley Williams andKatherine Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, DARPA, 9 2008.

[KBB09] Andreas Knüpfer, Holger Brunst, and Ronny Brendel. Open Trace Format Specification. Technical report, Center for Information Services and High Performance Computing (ZIH), April 2009.

[KBD+08] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing, Proceedings of the 2nd International Workshop on Parallel Tools*, pages 139–155. Springer, 2008.

[KGS+10] Andreas Knüpfer, Markus Geimer, Johannes Spazier, Joseph Schuchart, Michael Wagner, Dominic Eschweiler, and Matthias S. Müller. A Generic Attribute Extension to OTF and Its Use for MPI Replay. *Procedia Computer Science*, 1(1):2109–2118, May 2010.

[KHB+99] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP, pages 131–140, New York, NY, USA, 1999. ACM.

[Kir10] Nick Kirsch. Whitepaper: OneFS Operating System. Technical report, 2010.

[KL08] Julian Kunkel and Thomas Ludwig. Bottleneck Detection in Parallel File Systems with Trace-Based Performance Monitoring. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 212–221, Berlin, Heidelberg, 2008. University of Las Palmas de Gran Canaria, Springer-Verlag.

[KMKL11] Julian Kunkel, Timo Minartz, Michael Kuhn, and Thomas Ludwig. Towards an Energy-Aware Scientific I/O Interface – Stretching the ADIOS Interface to Foster Performance Analysis and Energy Awareness. *Computer Science - Research and Development*, 2011.

[KN10] Julian Kunkel and Petra Nerge. System Performance Comparison of Stencil Operations with the Convey HC-1. Technical Report 1, Deutsches Klimarechenzentrum GmbH, Bundesstraße 45a, D-20146 Hamburg, 11 2010.

[Knu79] Donald Knuth. *Structured Programming with Goto Statements*, pages 257–321. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[KO11] Andreas Knüpfer and Others. Score-P - Scalable Performance Measurement Infrastructure for Parallel Codes. Webpage `http://www.score-p.org`, 2011.

[KRVP07] Dries Kimpe, Rob Ross, Stefan Vandewalle, and Stefaan Poedts. Transparent Log-Based Data Storage in MPI-IO Applications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 233–241. Springer Berlin / Heidelberg, 2007.

[KST+11] Krishna Kandalla, Hari Subramoni, Karen Tomko, Dmitry Pekurovsky, Sayantan Sur, and Dhabaleswar Panda. High-performance and Scalable Non-blocking All-to-all with Collective Offload on InfiniBand Clusters: a Study with Parallel 3D FFT. *Computer Science - Research and Development*, 26:237–246, 2011.

[KSVP10] Krishna Chaitanya Kandalla, Hari Subramoni, Abhinav Vishnu, and Dhabaleswar K. Panda. Designing Topology-aware Collective Communication Algorithms for Large Scale InfiniBand Clusters: Case Studies with Scatter and Gather. In *International Parallel & Distributed Processing Symposium*, IPDPS, pages 1–8. IEEE, 2010.

[KT95] Yousef A. Khalidi and Moti N. Thadani. An Efficient Zero-Copy I/O Framework for UNIX. Technical report, Mountain View, CA, USA, 1995.

[KTML09] Julian Kunkel, Yuichi Tsujita, Olga Mordvinova, and Thomas Ludwig. Tracing Internal Communication in MPI and MPI-I/O. In *International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT*, pages 280–286, Washington, DC, USA, 12 2009. Hiroshima University, IEEE Computer Society.

[Kun06] Julian Kunkel. Performance Analysis of the PVFS2 Persistency Layer. Online `http://archiv.ub.uni-heidelberg.de/volltextserver/volltexte/2006/6330/pdf/PerformanceAnalysis.pdf`, 02 2006.

[Kun07] Julian Martin Kunkel. Towards Automatic Load Balancing of a Parallel File System with Subfile Based Migration. Master's thesis, Ruprecht-Karls-Universität Heidelberg, 08 2007.

[KWGS10] Andreas Knüpfer, Felix Wolf, Michael Gerndt, and Sameer Shende. The Future of the Open Trace Format (OTF) and Open Event Trace Recording. BoF `www.vi-hps.org/datapool/page/41/sc_bof_slides.pdf`, 12 2010.

[KWM+10] Alex Kaiser, Samuel Williams, Kamesh Madduri, Khaled Ibrahim, David Bailey, James Demmel, and Erich Strohmaier. A Principled Kernel Testbed for Hardware/Software Co-Design Research. In *USENIX Workshop on Hot Topics in Parallelism*, HOTPAR, 2010.

[LAS+09] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: an Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 469–480, New York, NY, USA, 2009. ACM.

[LB98] Oren La'adan and Amnon Barak. PhD thesis, 8 1998.

[LCC+07] Wei-keng Liao, Avery Ching, Kenin Coloma, Arifa Nisar, Alok Choudhary, Jacqueline Chen, Ramanan Sankaran, and Scott Klasky. Using MPI File Caching to Improve Parallel Write Performance for Large-scale Scientific Applications. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[LCCW05] Wei-keng Liao, Kenin Coloma, Alok N. Choudhary, and Lee Ward. Cooperative Write-Behind Data Buffering for MPI I/O. In Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *PVM/MPI*, volume 3666 of *Lecture Notes in Computer Science*, pages 102–109. Springer, 2005.

[LFL09] D. Lugones, D. Franco, and E. Luque. Dynamic and Distributed Multipath Routing Policy for High-Speed Cluster Networks. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID, pages 396–403, Washington, DC, USA, 2009. IEEE Computer Society.

[LKK+06a] Thomas Ludwig, Stephan Krempel, Julian Kunkel, Frank Panse, and Dulip Withanage. Tracing the MPI-IO Calls' Disk Accesses. In Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 4192 in Lecture Notes in Computer Science, pages 322–330, Berlin / Heidelberg, Germany, 2006. C&C Research Labs, NEC Europe Ltd., and the Research Centre Jülich, Springer.

[LKK+06b] Thomas Ludwig, Stephan Krempel, Julian Kunkel, Frank Panse, and Dulip Withanage. Tracing the MPI-IO Calls' Disk Accesses. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 4192 in Lecture Notes in Computer Science, pages 322–330, Berlin / Heidelberg, Germany, 2006. C&C Research Labs, NEC Europe Ltd., and the Research Centre Jülich, Springer.

[LKK+08] J. Lofstead, S. Klasky, Schwan K., N. Podhorszki, and C. Jin. Flexible IO and Integration for Scientific Codes Through The Adaptable IO System (ADIOS). In *CLADE 2008 at HPDC*, Boston, Massachusetts, June 2008. ACM.

[LNBG03] Michael Liljenstam, David M. Nicol, Vincent H. Berk, and Robert S. Gray. Simulating Realistic Network Worm Traffic for Worm Warning System Design and Testing. In *Proceedings of the 2003 ACM workshop on Rapid malcode*, WORM, pages 24–33, New York, NY, USA, 2003. ACM.

[LRO08] Alexey Lastovetsky, Vladimir Rychkov, and Maureen O'Flynn. MPIBlib: Benchmarking MPI Communications for Parallel Computing on Homogeneous and Heterogeneous Clusters. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 227–238. Springer Berlin / Heidelberg, 2008.

[LRT07] R. Latham, R. Ross, and R. Thakur. Implementing MPI-IO Atomic Mode and Shared File Pointers Using MPI One-Sided Communication. *International Journal of High Performance Computing Applications*, 21(2):132–143, 2007.

[LWSB97] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — On-line Monitoring Interface Specification (Version 2.0). Technical Report TUM-I9733, SFB-Bericht Nr. 342/22/97 A, Technische Universität München, Munich, Germany, July 1997.

[LZKS09] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, Metadata Rich IO Methods for Portable High Performance IO. In *International Parallel & Distributed Processing Symposium*, IPDPS, 2009.

[MCW+05] Shirley Moore, David Cronk, Felix Wolf, Avi Purkayastha, Patricia Teller, Robert Araiza, Maria Gabriela Aguilera, and Jamie Nava. Performance Profiling and Analysis of DoD Applications Using PAPI and TAU. In *DOD_UGC '05: Proceedings of the 2005 Users Group Conference on 2005 Users Group Conference*, page 394, Washington, DC, USA, 2005. IEEE Computer Society.

[Mes09] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard – Version 2.2. Technical report, September 2009.

[Mey07] Mike Meyers. Mike Meyers' CompTIA A+ Guide: PC Technician. Online `http://mrdanault.com/aplus/chap8.pdf`, 2007.

[Mic09] S. Michalak. Silent Data Corruption: A Threat to Data Integrity in High-End Computing Systems. In *Invited Panelist at 2009 National HPC Workshop on Resilience*, 2009.

[MK91] Ethan L. Miller and Randy H. Katz. Input/Output Behavior of Supercomputing Applications. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 567–576, New York, NY, USA, 1991. ACM Press.

[MK05] Sam Miller and Ricky Kendall. Implementing Optimized MPI Collective Communication Routines on the IBM BlueGene/L Supercomputer. Technical report, Iowa State University, 2005.

[MKJ+07] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In *Parallel Computing: Architectures, Algorithms and Applications, volume 15 of Advances in Parallel Computing*, pages 637–644. IOS Press, 2007.

[MKR95] Edward Mascarenhas, Felipe Knop, and Vernon Rego. ParaSol: A Multithreaded System for Parallel Simulation Based on Mobile Threads. In *Proceedings of the Winter Simulation Conference*, pages 690–697, 1995.

[MMK+12] Timo Minartz, Daniel Molka, Julian Kunkel, Michael Knobloch, Michael Kuhn, and Thomas Ludwig. *Tool Environments to Measure Power Consumption and Computational Performance*, pages 709–743. Chapman and Hall/CRC Press Taylor and Francis Group, 6000 Broken Sound Parkway NW, Boca Raton, FL 33487, 2012.

[MMSW02] Bernd Mohr, Allen D. Malony, Sameer Shende, and Felix Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *Supercomputing*, 23:105–128, August 2002.

[MN10] C.M. Macal and M.J. North. Tutorial on Agent-based Modelling and Simulation. *Journal of Simulation*, 4(3):151–162, 2010.

[MS09] Inc. Mesquite Software. Getting Started: CSIM 20 Simulation Engine. Online document: `http://www.mesquite.com/documentation/documents/CSIM20_Getting_Started-C++.pdf`, 2009.

[MSHA+97] William H. Mangione-Smith, Brad Hutchings, David Andrews, André DeHon, Carl Ebeling, Reiner Hartenstein, Oskar Mencer, John Morris, Krishna Palem, Viktor K. Prasanna, and Henk A. E. Spaanenburg. Seeking Solutions in Configurable Computing. *Computer*, 30:38–43, December 1997.

[NFG+10] Alberto Núñez, Javier Fernández, Jose Garcia, Félix Garcia, and Jesús Carretero. New Techniques for Simulating High Performance MPI Applications on Large Storage Networks. *The Journal of Supercomputing*, 51:40–57, 2010.

[NSM04] David Nagle, Denis Serenyi, and Abbie Matthews. The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC, pages 53–, Washington, DC, USA, 2004. IEEE Computer Society.

[Obj03] Object Management Group. UML Profile for Schedulability, Performance and Time. Online Document: `http://www.omg.org/spec/SPTP/1.0/`, sep 2003.

[Pan04] INC. Panasas. Whitepaper: Object Storage Architecture. Technical report, 2004.

[Pat09] David Patterson. The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges. Whitepaper `http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf`, September 2009.

[PBB07] Sabri Pllana, Ivona Brandic, and Siegfried Benkner. Performance Modeling and Prediction of Parallel and Distributed Computing Systems: A Survey of the State of the Art. In *Proceedings of the First International Conference on Complex, Intelligent and Software Intensive Systems*, pages 279–284, Washington, DC, USA, 2007. IEEE Computer Society.

[PBXB08] Sabri Pllana, Siegfried Benkner, Fatos Xhafa, and Leonard Barolli. Hybrid Performance Modeling and Prediction of Large-Scale Computing Systems. In *CISIS '08: Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 132–138, Washington, DC, USA, 2008. IEEE Computer Society.

[PD11] L.L. Peterson and B.S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, March 2011.

[Pid04] Michael Pidd. *Computer Simulation in Management Science*. Wiley, 5th edition, April 2004.

[PK05] Bernd Page and Wolfgang Kreutzer. *The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java*. Shaker Verlag, 12 2005.

[PM04] Michal Pioro and Deepankar Medhi. *Routing, Flow, and Capacity Design in Communication and Computer Networks*. Elsevier, July 2004.

[PP94] Barbara K. Pasquale and George C. Polyzos. Dynamic I/O Characterization of I/O Intensive Scientific Applications. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 660–669, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[Pra04] Herbert Praehofer. Simulation Technischer Systeme. Lecture notes, 2004.

[PTH+01] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. MPI-IO/GPFS, an Optimized Implementation of MPI-IO on Top of GPFS. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 17–17, New York, NY, USA, 2001. ACM.

[PW05] Dorin Bogdan Petriu and Murray Woodside. Software Performance Models from System Scenarios. *Performance Evaluation*, 61(1):65–89, 2005.

[Rab04] Rolf Rabenseifner. Optimization of Collective Reduction Operations. *Computational Science-ICCS 2004*, pages 1–9, 2004.

[Ras07] E. Rasmusen. *Games and Information: An Introduction to Game Theory*. Wiley-blackwell, 2007.

[RHB+11] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. The Structural Simulation Toolkit. *SIGMETRICS – Performance Evaluation Review*, 38:37–42, March 2011.

[Rot07] Philip C. Roth. Characterizing the I/O Behavior of Scientific Applications on the Cray XT. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing '07*, PDSW, pages 50–55, New York, NY, USA, 2007. ACM.

[RW94] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *Computer*, 27:17–28, March 1994.

[SACR96] E. Smirni, R. A. Aydt, A. A. Chen, and D. A. Reed. I/O Requirements of Scientific Applications: An Evolutionary View. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, HPDC, pages 49–, Washington, DC, USA, 1996. IEEE Computer Society.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies*, FAST, pages 231–244, 2002.

[She99] Joel Reece Sherrill. *Priority Inversion in Real-time File Systems*. PhD thesis, 1999.

[SIC+07] David E. Singh, Florin Isaila, Alejandro Calderon, Felix Garcia, and Jesus Carretero. Multiple-Phase Collective I/O Technique for Improving Data Access Locality. In *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP, pages 534–542, Washington, DC, USA, 2007. IEEE Computer Society.

[SIPC09] David E. Singh, Florin Isaila, Juan C. Pichel, and Jesús Carretero. A Collective I/O Implementation Based on Inspector–Executor Paradigm. *The Journal of Supercomputing*, 47(1):53–75, 2009.

[SM06] Sameer S. Shende and Allen D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[SM08] Inc Sun Microsystems. Peta-Scale I/O with the Lustre File System. Technical report, Oak Ridge National Laboratory, 2 2008.

[SMAb01] Sameer Shende, Allen D. Malony, and Robert Ansell-bell. Instrumentation and Measurement Strategies for Flexible and Portable Empirical Performance Evaluation. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA, pages 1150–1156, 2001.

[SSM+09] W. Spear, S. Shende, A. Malony, R. Portillo, P. Teller, D. Cronk, S. Moore, and D. Terpstra. Making Performance Analysis Tuning Part of the Software Development Cycle. IEEE Computer Society, 2009.

[Ste79] Schlesinger Stewart. Terminology for Model Credibility. *Simulation*, pages 103–104, 1979.

[STHI10] Robert Schöne, Ronny Tschüter, Daniel Hackenberg, and Thomas Ilsche. The VampirTrace Plugin Counter Interface: Introduction and Examples. In *Proceedings of the EuroPar 2010 - Workshops*, 2010.

[Tak82] L. Takács. *Introduction to the Theory of Queues*. Greenwood Press, 1982.

[Tau10] TAU User Guide. Online Document `http://www.cs.uoregon.edu/research/tau/tau-usersguide.pdf`, 2010.

[TGL99] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *FRONTIERS '99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, page 182, Washington, DC, USA, 1999. IEEE Computer Society.

[TGL02] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing*, 28:83–105, 2002.

[THW10] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A lightweight Performance-oriented Tool Suite for x86 Multicore Environments. In *39th International Conference on Parallel Processing Workshops*, ICPPW, pages 207–216. IEEE, April 2010.

[TJYD09] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing, Proceedings of the 3rd International Workshop on Parallel Tools*, pages 157–173. Springer, 2009.

[TJYD10] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting Performance Data with PAPI-C. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173. Springer Berlin Heidelberg, 2010.

[TRG05] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, February 2005.

[VH08] András Varga and Rudolf Hornig. An Overview of the OMNeT++ Simulation Environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[VO10] András Varga and OpenSim Ltd. OMNet++ User Manual Version 4.1. Technical report, June 2010.

[WBM+10] Brian J. N. Wylie, David Böhme, Bernd Mohr, Zoltán Szebenyi, and Felix Wolf. Performance Analysis of Sweep3D on Blue Gene/P with the Scalasca Toolset. In *International Parallel & Distributed Processing Symposium*, IPDPS. IEEE Computer Society, April 2010.

[WG90] M. Y. Wu and D. D. Gajski. Hypertool: A Programming Aid for Message-Passing Systems. *IEEE Transactions Parallel Distributed Systems*, 1:330–343, July 1990.

[Wik11] Wikipedia, the Free Encyclopedia, 2011.

[WK94] S.C. Winter and P. Kacsuk. Software Engineering for Parallel Processing. *IEE Colloquium on High Performance Computing for Advanced Control*, 12 1994.

[WK04] Terry L. Wilmarth and Laxmikant V. Kale. POSE: Getting Over Grainsize in Parallel Discrete Event Simulation. In *Proceedings of the 2004 International Conference on Parallel Processing*, ICPP, pages 12–19, Washington, DC, USA, 2004. IEEE Computer Society.

[WM03] Felix Wolf and Bernd Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003.

[Woo02] Murray Woodside. Tutorial Introduction to Layered Modeling of Software Performance – Version 3.0, 5 2002.

[Wor06] Joachim Worringen. Self-Adaptive Hints for Collective I/O. Presentation, available online, 2006.

[WPP+05] Murray Woodside, Dorina C. Petriu, Dorin B. Petriu, Hui Shen, Toqeer Israr, and Jose Merseguer. Performance by Unified Model Analysis (PUMA). In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 1–12, New York, NY, USA, 2005. ACM.

[WZB+05] Terry L. Wilmarth, Gengbin Zheng, Eric J. Bohm, Yogesh Mehta, Nilesh Choudhury, Praveen Jagadishprasad, and Laxmikant V. Kale. Performance Prediction Using Simulation of Large-Scale Interconnection Networks in POSE. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 109–118, Washington, DC, USA, 2005. IEEE Computer Society.

[YV07] Weikuan Yu and Jeffrey Vetter. Exploiting Lustre File Joining for Effective Collective IO. In *Proceedings of the 2007 7th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID, pages 267–274. IEEE Computer Society, Los Alamitos, CA, USA, 2007.

[ZKK04] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kalé. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In *International Parallel & Distributed Processing Symposium*, volume 1 of *IPDPS*, page 78b, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[ZLGS99] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward Scalable Performance Visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.

[ZWJK05] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant V. Kalé. Simulation-Based Performance Prediction for Large Parallel Machines. *International Journal of Parallel Programming*, 33(2):183–207, June 2005.

Chapter 3 III

Characterizing the Experimental System

Various performance factors and characteristics have been introduced in Section 2.2.1. Before a model of a system can be built the behavior and performance must be understood. Thus, the characteristics must be determined.

Measurement of system characteristics is tedious work and requires careful interpretation of obtained values. Therefore, in this chapter a measurement methodology is introduced in Section 3.1. Then, the working groups compute cluster and its configuration is described in Section 3.2. This cluster is used in later validation experiments. Next, performance of memory, inter-process communication and I/O subsystem is measured and assessed in Section 3.4, 3.3, 3.5 and 3.6 respectively. This will show the complex interplay between hardware components and operating system. Further, these results will be used to validate the behavior of the simulation of these basic components.

3.1. Measuring System Behavior

To obtain experimental characteristics of a system measurements are necessary. The term *characteristics* is used in this thesis to indicate qualitative but also quantitative observable behavior of a hardware or software component. The term characteristics is frequently used synonymously to the term *metrics*, which just indicates a quantitative measure of behavior, such as network latency.

A *metric* is an *"analytical measurement intended to quantify the state of a system"* ([Wik11], Metric). By interpreting metrics the qualitative characteristics of the system can be obtained. The definition of a metric includes instructions how to measure values for this metric. In the context of computer science typically small code pieces – called *benchmarks* try to stress relevant aspects of the system and quantify them with certain metrics. Often the term benchmark is referred to as a collection of several *kernels*. A kernel is a small code piece that is the essence of a specific use case. Typically, a kernel either mimics a particular aspect of an application, or it stresses a particular aspect of the system.

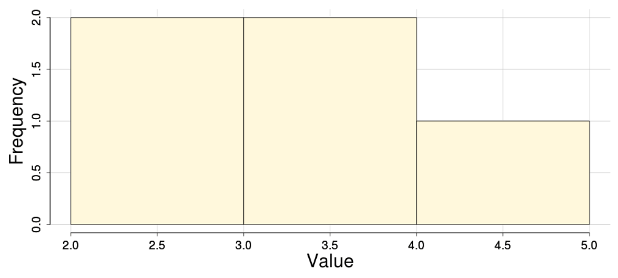
Unfortunately, measurement does not necessarily reveal the real value of a metric; systematic mistakes may bias the result, and background activity influences the utilization of hardware components. This error is common to all measurements and called *observational error*. An informal definition of this term is given in the Wikipedia:

> *"Every time we repeat a measurement with a sensitive instrument, we obtain slightly different results. The common statistical model we use is that the error has two additive parts: systematic error which always occurs (with the same value) when we use the instrument in the same way, and random error which may vary from observation to observation."* [Wik11], article about Observational Error

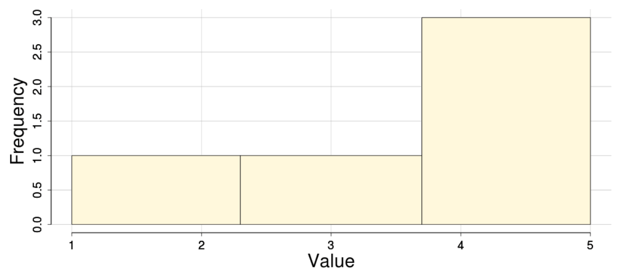
The terms *precisions* and *accuracy* state how close the measurements are to the real value:

> *"In the fields of science, engineering, industry and statistics, the accuracy of a measurement system is the degree of closeness of measurements of a quantity to that quantity's actual (true) value. The precision of a measurement system, also-called reproducibility or repeatability, is the degree to which repeated measurements under unchanged conditions show the same results. Although the two words can be synonymous in colloquial use, they are deliberately contrasted in the context of the scientific method."* [Wik11], Accuracy and precision

Both facets are relevant for measurements conducted in this thesis. To assess the accuracy and precision of obtained values mathematics can help.



(a) First bin starts with minimum value.



(b) Alternative representation.

Figure 3.1.: Possible histograms for the vector $m = (3, 5, 4, 2, 4)$ with three bins.

3.1.1. Mathematical Background

This small section offers a brief introduction related to the error estimation mathematics. The presented collection is a selection of practical methods to quantify metrics and the errors of the measurement environment.

Assume the small vector $m = (3, 5, 4, 2, 4)$ has been measured for the metric m; every value is an independent observation that is measured by repeating an experiment. When many measurements are made, listing the vector becomes tedious. It is interesting to see the rough distribution of values. This is possible with a *histogram*. In a histogram the range of the observed values are split into intervals – called bins, and the frequency of observations within every interval is given. While arbitrary intervals are possible, usually the range of the values is split into equidistant intervals – the user just selects the number of bins. In this case, the interval length is computed with the following equation: interval len = $\frac{\text{maxValue - minValue}}{\text{number of bins}}$.

Attention must be spent in the interpretation of a histogram. The look depends on the construction – the number of bins and the minimum and maximum value. Data for our example is visualized in histograms with equidistant binning and three bins in Figure 3.1. In Figure 3.1a the defaults of the statistical tool R has been chosen showing two values in the interval $[2, 3]$ and $(3, 4]$ – the minimum value is accounted for the first bin. However, by picking a minimum of 1, each interval has a length of 1.33 resulting in Figure 3.1b; suddenly three values are accounted for the last interval. This is pathological example because the first figure renders values that are the interval boundaries, also the number of observation is very low. However, it demonstrates that comparison between histograms must be done carefully.

Since experimental measurements are expected to be suspect to random error, statistical methods are necessary to estimate the true value. Often in natural science it is assumed that the random error leads to observations that are distributed with a *Gaussian probability distribution*. Thus, the *average value* of the observations is an estimate for the real value – in our example the average is 3.6. Frequently the *standard error of the mean* (SEM) is used as an indicator for the precision of the estimate. The *SEM* denotes how close the average value ($\overline{X}$) is to the real value (μ), see equation 3.3 for the computation of the SEM. The average value ($\overline{X}$), the so-called sample standard deviation (s) and the *standard error of the mean* are computed according to equations 3.1 and 3.2, respectively. In the equations n is the number of samples, X_i the observed value for the i-th sample.

The sample standard deviation is an estimate for the real standard deviation in the noisy environment. It can be used to reject observations that skew the average value – these values are expected to be skewed by processes and systematic mistakes. Outliers are then removed from the vector of observations and thus do not contribute to the estimated average. Then, the computation would be repeated with the remaining values to obtain the new average and an improved estimate for the standard error. It is expected that 95% of the values are in the interval $[\overline{X} - 2 \cdot s, \overline{X} + 2 \cdot s]$ and 3 standard deviations cover 99.7% of the samples[1]. In our example the sample standard deviation is 1.1. Thus, the interval is $[1.2, 5.8]$ and no value is rejected.

[1]Refer to [Wik11](Standard deviation) for further information.

With the help of the SEM and the estimated mean a confidence interval for the true mean can be determined ([Wik11] Standard error (statistics)). The real value of the mean and thus the true value of the metric is expected to be in the interval $[\overline{X} - 2 \cdot SEM, \overline{X} + 2 \cdot SEM]$. This is motivated by the fact that the estimated average ± 2 standard deviations already cover 95% of data points. And since SEM is an estimate for the standard deviation of the sample-mean's estimate, with a probability of 95% the true mean should be within the range of the sample mean ± 2 SEM. In the example the SEM is 0.51 and thus, with a probability of 95% the true value is in the range of 3.6 ± 1. With an increase in the number of samples the confidence interval becomes smaller, thus the real value can be approximated better. However, due to the square-root, reducing the interval to half the size requires four times the number of observations. Remember, the equations require that the measured data points are distributed with a Gaussian probability distribution.

Besides the average value, another description of observations is the concept of *percentiles*. *"In statistics, a percentile (or centile) is the value of a variable below which a certain percent of observations fall."* ([Wik11], Percentile). *Quartiles* are the 25th, 50th and 75th percentile. Thus, quartiles are the three data points that divide the data set into four (almost) equally sized groups of values. Roughly one quarter of the values lies in every group. To obtain the percentiles (or quartiles) all values are sorted, then the value of the sorted vector is picked below which a given percentage of the data points reside. In the example five values are considered; the sorted vector is $s = (2, 3, 4, 4, 5)$ and thus below the second value (and of course above the minimum value) are roughly 25% of values. Thus, quartiles for the example vector are $(3, 4, 4)$. Note that the second quartile is defined as the median.

$$\overline{X} = \frac{1}{n} \sum_{i=1}^{n} X_i \tag{3.1}$$

$$s = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} \left(X_i - \cdot \overline{X}\right)^2} = \sqrt{\frac{1}{n-1} \left(\sum_{i=1}^{n} X_i^2 - n \cdot \overline{X}^2 \right)} \tag{3.2}$$

$$SEM = \frac{s}{\sqrt{n}} \tag{3.3}$$

3.1.2. Measurement Methodology

When we characterize the hardware we are interested in real behavior, however, software and hardware optimizations influence the outcome and bias the results. The effect of long term background noise, that is background activity which occurs a substantial amount of time while the kernel runs, and singular effects can be mitigated by performing multiple measurements on independent components of the homogeneous cluster system. To increase statistical independence some time should pass between two measurements to reduce the impact of long term background noise. Consequently, a benchmark which consists of multiple kernels which are all repeated in a loop is more robust to background influence than one which repeats every test before it moves to the next kernel. All measurements made for this thesis were repeated at least three times, in most cases even on different nodes of the cluster. A node is reserved exclusively for a benchmark and thus other users could not cause further background noise.

To ensure correctness of a benchmark measurements are also conducted with multiple third-party tools. This increases the confidence in reported values by revealing potential measurement errors caused by programming mistakes. Hardware specification serves as a reference, this allows detecting whether systematic errors impair the measurements.

Unfortunately, as we will see, most performance metrics obtained in a computer system do not follow the Gaussian distribution. This is due to the fact that optimizations are not triggered randomly and several optimizations interfere when a performance metric is measured. Hence, the common methodology to compute the *standard deviation* to remove outliers is not applicable. To assess the fluctuation of observed data, the first and third quartile will be given for many important metrics.

Measurements require timing. However, the physical hardware timer of the system has a limited accuracy, that depends on the selected timer (depending on the hardware and operating system multiple low-level timers are provided). All used timers provide an accuracy of at least 1 μs. Consequently, the granularity of the timed events must be chosen to ensure that multiple μs are needed to perform the timed operations.

Timing in parallel benchmarks running on multiple nodes is problematic because clock synchronization in distributed system is non-trivial. On our cluster a *Network Time Protocol* (NTP)[2] service is installed to support synchronization of the clocks to some degree. See A.2 for details and a discussion of the accuracy of NTP on our cluster. To avoid skewed timing of the distributed clocks, the timing of parallel benchmarks is always measured on Rank 0 and typically shielded with a `MPI_Barrier()`. While this guarantees that all processes finished their work, the processing of the barrier adds overhead and thus it is source of a systematic error. However, we will see that this overhead is negligible for the conducted experiments.

3.2. Overview of the Cluster

In Figure 3.2 the available hardware resources and the network topology of the working group's cluster is illustrated. The *cluster* front-end is a gateway to an internal switched network that connects all available nodes with star topology. The front-end schedules operations on the available 11 nodes, 10 nodes (hostname *West1* to *West10*) are equipped with two Intel Westmere processors, also a 4 socket AMD Magny-Cours node (called *Magny1*) is available.

The cluster is installed with Ubuntu 10.04 and offers additional repositories with newer software versions for scientific software and middleware. Jobs are executed by the batch scheduler *Torque*[3] and the cluster status is monitored by *Munin*[4]. Both services depend on background daemons which periodically (in second intervals) communicate with the front-end. In experiments with MPI either *Open MPI* (version 1.5.3) or *MPICH2* (version 1.3.1) are used.

Since the testbed for all experiments are the *Westmere* nodes, those are analyzed in more detail in the following.

3.3. Processor

A *Westmere* node is equipped with two *Intel Xeon 5650* processors each offering 6 cores [Int11, Int]. Hyper-threading is enabled on our cluster and thus 24 logical processors are visible in Linux. Those processors operate at a nominal frequency of 2.66 GHz and utilize three memory channels.

Dynamic voltage and frequency scaling (DVFS) of the CPUs is activated on our cluster and the *ondemand governor*[5] is active. Therefore, the frequency of the processor is adjusted to match the utilization. When the frequency is low enough the processor reduces the voltage, which, in-turn, decreases power consumption. Basically, the processor supports Intel's *Turbo Boost* technology, which means the actual frequency could be above the nominal 2.66 GHz. However, this feature is disabled on our cluster by limiting the maximum frequency of the CPU governor because the frequency depends on the actual temperature of the processor and its production[6].

[2]`http://www.ntp.org/`
[3]`http://www.clusterresources.com/`
[4]`http://munin-monitoring.org/`
[5]Refer to the Linux kernel documentation enclosed in the tarball of the vanilla kernel (`cpu-freq/governors.txt`).
[6]Due to the production process of the small structures there are minimal differences between produced microprocessors.

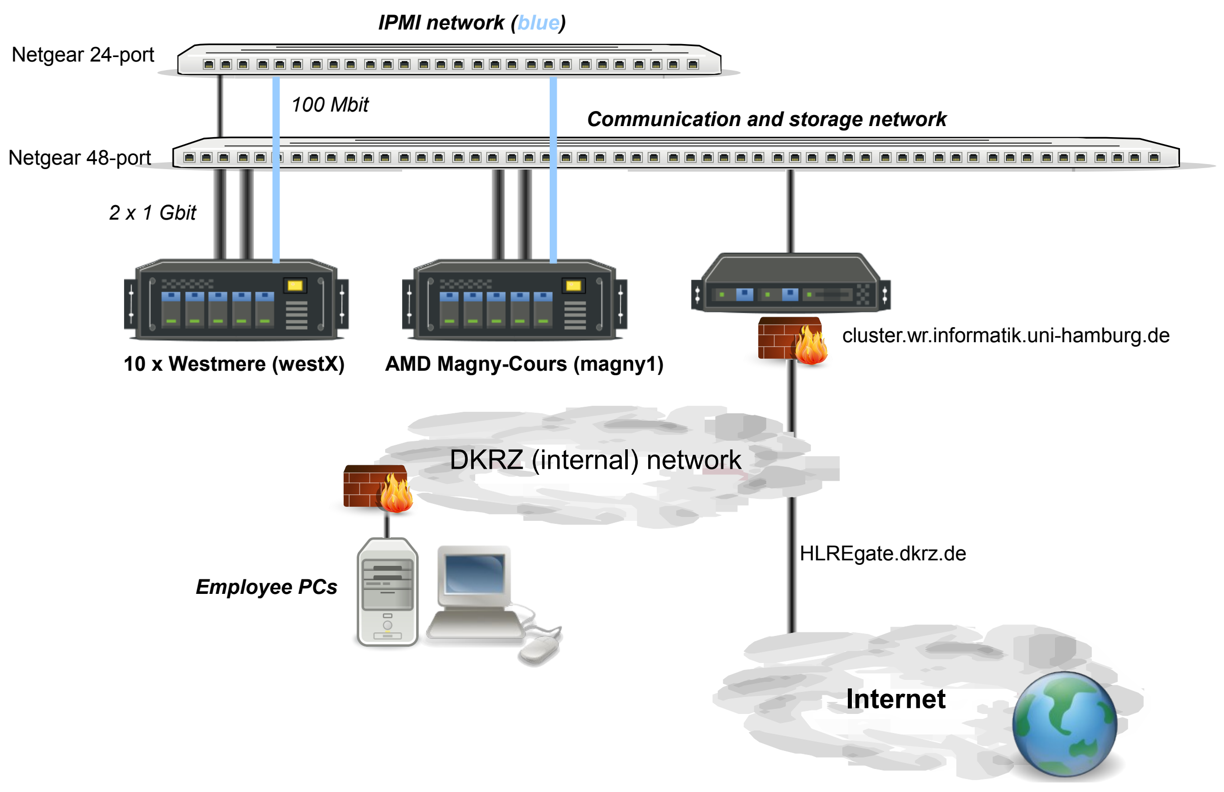


Figure 3.2.: Physical view of the working groups' cluster.

The amounts of cache available per physical core are 64 KByte L1 cache (split into a 32 KByte instruction and a 32 KByte data cache) and 256 KByte L2 cache. Every processor has an *inclusive* 12 MByte L3-Cache, thus all L2 data of the cores is also available in the shared L3 cache.

In the system architecture of a node, all PCI(-Express) components, such as the NIC, are connected to the *I/O Hub* (IOH) via Intel's *QuickPath* technology [Int09b]. To communicate between two sockets data is shipped via a 6.4 GT/s QPI interface. According to the specification 16 bit of data are transferred per cycle, thus the uni-directional speed is 12,800 MB/s[Int09a] (12,207 MiB/s). The communication protocol has an overhead of roughly 11 %, therefore, the effective I/O throughput is limited by 10,864 MiB/s.

3.4. Main Memory

A node is equipped with 12 GByte DDR-3 main memory (6 times 2 GByte modules). The *Nehalem* microarchitecture actually offers up to three memory-channels per socket, which is utilized in our case. Memory operates at 1333 MHz, and thus with the triple channel a maximal theoretical throughput of 32 GB/s is achievable (per socket). Data is transferred in the granularity (*cache line size*) of 64 bytes between processor L3 cache and memory.

To access memory from the remote socket data must be shipped via QuickPath and thus memory throughput from/to the remote processor is also limited by the performance of the QPI interconnect (which is 10,864 MiB/s). The achieved memory bandwidth is sensitive to the benchmark's kernel, and thus, varies with the specific access patterns.

When data is accessed it is likely to be located on the same processor, because Linux allocates memory with the *first touch policy*. With *first touch* space is allocated on the memory controller of the socket on which the data is written the first time[7]. Since every program used in the following experiments initializes its

[7]The allocation of memory in a program with `malloc()` does not count, because the OS defers allocation of physical pages until data is written the first time.

buffers, memory should be allocated on the right socket and thus no QPI transfer should be necessary. This expectation has been validated with `Likwid` for one of the memory benchmarks, the experiment showed that only a very small fraction of data data is accessed on the remote socket.

Although every socket has its own memory controller, the conducted experiments just utilize a single controller. An improved performance by reading from the memory of one socket and writing to the other socket might be possible, but has not been checked. This is evaluated later in the section about inter-process communication. In the following the quantitative and qualitative analysis of the memory performance of a single socket is conducted.

Further information about the architecture and some comparative analysis is available in [Rol09].

3.4.1. Memory Behavior

The general behavior of memory access is evaluated with the `bandwidth`[8] benchmark (version 0.26) which is run in Linux[9]. This tool contains assembler kernels to perform sequential and random read/write tests. It iterates over various amounts of data to determine caching and memory behavior. To measure a reliable average performance, the benchmark times the execution and loops over every test for at least 5 s.

Obtained throughputs for accessing data with a granularity of 64 bit and 128 bit are visualized in Figure 3.3a and Figure 3.3b, respectively. The benchmark permits to measure write accesses which bypass the cache and directly hit main memory, those results are given in Figure 3.3c.

Also, the benchmark has been run on several nodes to compare performance of the different servers. As expected by the homogeneous configuration of the nodes, those values fluctuate only slightly. Therefore, all figures show the results obtained on the first Westmere node.

Observations and interpretation

a) Performance drops when the accessed data is above 32 KiB, 256 KiB and 8 MiB. The three plateaus are caused when data does not fit any more in the processor L1, L2 and L3 caches, respectively. Sequential read and sequential write achieve similar results, except when data does not fit into L3 cache any more – in this case, reads are faster.

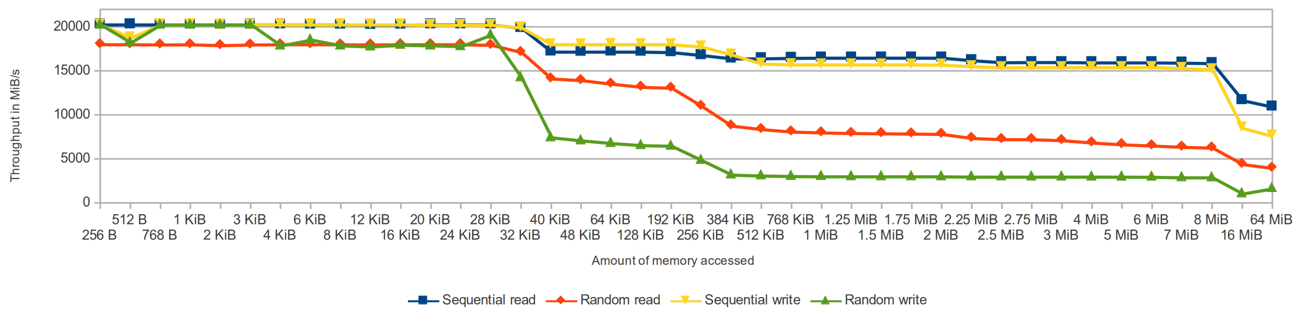
b) The single threaded benchmark is not capable to extract the memory bandwidth of 32 GB/s that is theoretical available on a single socket. The achieved sequential read and write performance measured for 64 MiB accesses is about 11 to 12 GiB/s. Performance could be lost if the bandwidth benchmark does not fit the platform well – for example, operations might not suffice to generate enough memory requests to utilize the memory controller, it might be dominated by latency. On the other hand, our Westmere platform and the configuration might degrade performance. This cannot be answered in the scope of this thesis.

c) Performance of 128 bit access is roughly twice as high as with 64 bit access – as long as data fits into L1 or L2 cache. Sequential L3 accesses show little improvement, but random access almost doubles performance.

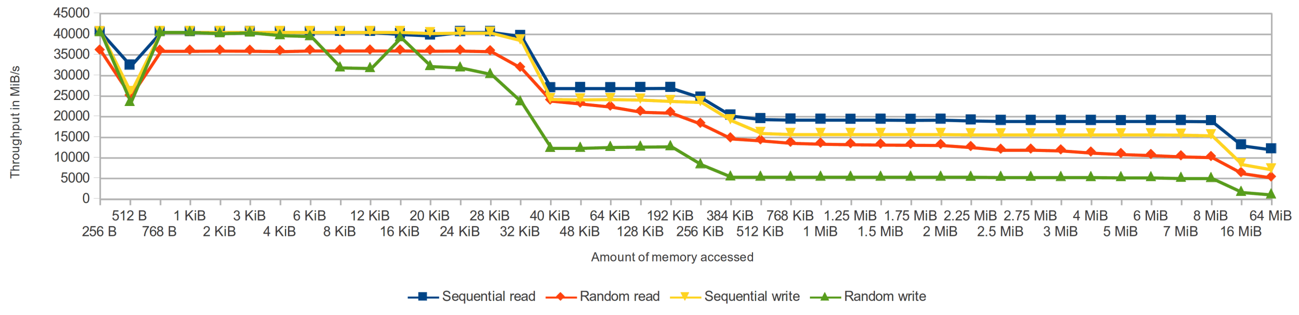
d) Random access is slower than sequential access. This is caused by the fact that data is transferred with a granularity of 64 bytes between the caches and main memory. When just 128 bit (16 bytes) are accessed, three times the accessed amount of irrelevant data must be transferred. Thus, performance is expected to drop to 1/4. In the measurement the write performance halves for L2 access and is

[8] `http://home.comcast.net/~fbui/bandwidth.html`

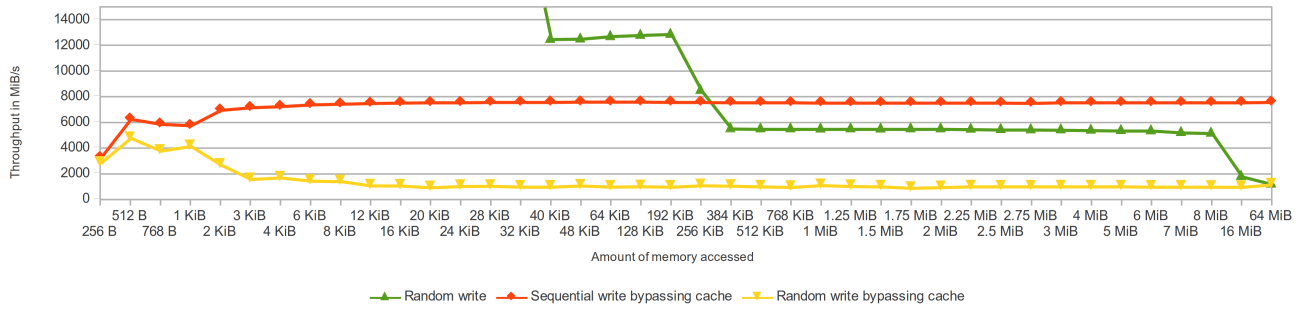
[9] Although the operating system and the glibc library are expected to influence the benchmark results, a benchmark running on top of an operating system has been chosen because this is the regular use-case. This includes the conducted experiments, which are also influenced by the OS. With Memtest86+ (`http://www.memtest.org/`), there is actually a tool which can run at boot-time. Originally intended to detect memory errors, it also reports the average performance of L1, L2 and main memory.



(a) 64 bit accesses.



(b) 128 bit accesses.



(c) 128 bit accesses bypassing the cache.

Figure 3.3.: Memory throughput measured with `bandwidth`.

roughly 1/4 for L3 access and for larger amounts of data. For an access granularity of 64 bit just 1/8 of data is typically accessed. Therefore, performance degrades by the same factor.

e) Interestingly, performance of random reads does not degrade to the same extent, this might be caused by the inclusive character of the L3 cache and the required write-back of data to the next cache level.

f) The impact of the cache hierarchy becomes apparent when cache is bypassed (see Figure 3.3c). Write throughput goes down to 7.5 GiB/s for sequential access, and to 1 GiB/s for random access already with small memory sizes. With 64 MiB of data, performance of random writes with and without bypassing cache behave almost equal.

Several interesting caching and memory behavior are quantified by the presented benchmark. With data fitting in L1 cache a rate of 40 GiB/s is measured with `bandwidth`, but larger amounts of data do not fit in the caches any more. This is a typical situation for memory-bound scientific problems. Therefore, more benchmark kernels are evaluated to determine and verify the memory throughput of the nodes.

3.4.2. Throughput

Several kernels and benchmarks are executed to assess the observed behavior and to derive a valid estimation for the maximum memory throughput. The resulting memory throughput for accessing large memory regions are presented in Table 3.1. In repeated tests a slight fluctuation of obtained performance (within a few percent) is noticeable.

The conducted experiments are as follows:

- `bandwidth` – the 64 MiB values from the previous experiments are taken.
- Writing data to the in-memory file system `tmpfs` via the command `dd if=/dev/zero bs=1024k count=2000 of=/dev/shm/test`. This value measures reading from a buffer and writing to memory and adds the overhead of the `tmpfs` file system. The `tmpfs` is an in-memory file system offered by the Linux kernel, all data of the file system is maintained in main memory.
- A local TCP connection with `iperf`[10]. This value approximates the performance of the TCP stack for an inter-node communication based on sockets, and, therefore, is lower than accessing main memory. Nowadays efficient MPI implementations communicate local messages directly via shared memory to avoid the additional overhead of the TCP protocol, but still the value serves as an approximate for the real performance.
- `memory-bandwidth`, a benchmark developed for this thesis which measures performance of three access patterns with a smaller access granularity to allow assessing the variability of performance. The benchmark loops over a larger amount of data and times subsequent groups of multiple data accesses. The average value is shown in the table, a detailed analysis of the results is given in Section 3.4.3.

Observations and interpretation

a) Memory throughput varies between 1 GiB/s and 10 GiB/s, it is sensitive to the actual access pattern and access granularity.

b) Writing data to `tmpfs` achieves roughly half the performance of `memcpy()` (measured with `bandwidth`). `memcpy()` copies data in user-space, while the transfer to `tmpfs` with `dd` copies data from kernel-space into a user buffer and from the user-space back to `tmpfs` in kernel-space. An additional internal processing of the data structures is required within `tmpfs` and two mode changes between kernel-space and user-space. Compared to `memcpy()` data is copied twice with `dd`. Therefore, it is expected that the performance of the I/O is below half the invocation of a plain `memcpy()` – when all operations hit main memory. This translates to an upper bound of roughly 2,100 MiB/s (half the

[10]`http://iperf.sourceforge.net/`

Benchmark and configuration			Measured throughput
bandwidth (64 MiB of data)	128 bit	sequential read	ca. 12000
		random read	5328
		sequential write	7642
		random write	1226
		sequential write bypassing cache	7616
		sequential copy	4517
	64 bit	sequential read	10947
		random read	4005
		sequential write	7582
		random write	1692
	library	memset	7464
		memcpy	4212
dd	writing to tmpfs		2087
iperf	local connection		1540
memory-bandwidth	sequential read		8347
	sequential write		7614

Table 3.1.: Memory throughput in MiB/s achieved with several tools.

performance of `memcpy()`). Since this performance can be seen in the measurements, caches do not help `dd` although only 1 MiB of data is transferred between `/dev/null` and `tmpfs`.

c) With `iperf` performance is just 1.5 GiB/s. Internally, since two processes communicate data must be copied from the first process to kernel-space, in this process it is segmented into TCP packets with an maximum size of 16 KiB. Then the context changes to the second process, the TCP packets are then unpacked by the second process and copied from kernel space to user-space. Therefore, two copies are necessary which translates to an upper bound of roughly 2,100 MiB/s (half the performance of `memcpy()`). Due to the context changes and the processing of TCP, performance is lower.

d) The benchmarks validate the results of `memory-bandwidth` because the sequential write performance is close to the results obtained with `bandwidth`. The cause of a degraded read throughput has been investigated; the loop which accesses memory in `memory-bandwidth` is not forced to be unrolled[11], while the assembler kernel in `bandwidth` has an unrolled kernel. When the code is unrolled similar results have been measured. Again, this shows the influence of the memory access pattern and the hardware architecture to the observable performance.

3.4.3. Analysis of Variability and Noise

This experiment fosters understanding the hardware variability between different servers, and the impact of background noise to memory performance.

The benchmark `memory-bandwidth`[12] measures memory throughput with three memory intense kernels: The first two tests write and read a consecutive chunk of memory, respectively. The last kernel (called RWW) reads a piece of memory then writes the read data and overwrites it again. Data is accessed in a (not unrolled) loop with a 64 byte granularity, since it has been observed with `bandwidth`, that this configuration yields a good performance. Each kernel iterates over a memory region of a given size multiple times. Additionally for a reference and burn-in phase, the memory is initialized with `memset()` for given number of iterations.

A main difference to other conducted benchmarks is that while the memory is accessed performance is

[11]The benchmark is compiled with `gcc -O3` which does not include `-funroll-loops`.
[12]The benchmark is part of the PIOsimHD distribution. The source code is given in Section A.5.

Experiment / Host	West1	West2	West3	West4	West5	West6	West7	West8	West9	Lap.
Write	7615	7618	7637	7505	7640	7630	7624	7619	7604	3024
Read	8347	8344	8339	8346	8333	8328	8347	8338	8345	3119
RWW	3031	3034	3036	3019	3036	3036	3035	3035	3034	1021

Table 3.2.: Average memory throughput in MiB/s to access 100 GiB of data with `memory-bandwidth`.

timed with a finer granularity and those timings are written out after the benchmark. This behavior of fine-grained timing is the main reason for the new benchmark; by looking at the distribution of timings background noise is revealed and, further, outliers of an experiment can be identified and removed later on. The physical hardware timer of the system has a limited accuracy, therefore, time cannot be measured accurately for just accessing a single byte. Thus, a consecutive memory region of a given size is accessed in this benchmark to obtain a single timing. Both the size of the memory chunk accessed and the frequency of the timing can be configured in order to measure cache effects or main memory performance.

In the conducted experiment 1000 MByte of data are accessed in a loop 100 times, and the time is measured for every Megabyte accessed[13]. That means in total the durations to access 100,000 samples are gathered. Clearly, the total amount of data accessed suffices to overrun the available L3 cache. The chosen granularity of one Megabyte ensures that the clock accuracy of the system suffices to provide accurate timings.

Coming back to the RWW benchmark, this benchmark reads data in the granularity of the timing, that means in this experiment 1 MByte of data is read, then written and re-written. Then the next MByte of the whole Gigabyte of data is processed. This amount of data can fit into the processors L3 cache, consequently, the performance of RWW is different than just executing a sequence of read and write benchmarks over the full data set.

All 10 homogeneous cluster nodes are compared by their average value and with histograms, to reveal machine differences. Further, the authors laptop system is measured for comparison. The samples are fed into the statistical analysis tool *R* and analyzed to assess variability and average performance.

Average performance First, the average performance results to access the 100 GiB of data is listed in Table 3.2. All systems achieve quite similar results. This is due to the fact that in total 100 GByte of data is accessed, which means short fluctuations and the impact of noise is mitigated. Since the values are close together the accuracy of the average values is good. The performance of the laptop differs significantly from the Westmere nodes.

Read performance is slightly higher than write performance; the average throughput of the nodes is 8,340 MiB/s and 7,610 MiB/s for reads and write, respectively. The average RWW throughput is 3,032 MiB/s which is higher than the performance of conducting the read test, and then executing two times the write test. If RWW kernel would behave like executing the other two kernels sequentially, then according to the measurements in the table, this combination should yield about 2,613 MiB/s. This shows the impact of caching for this simple workload.

Distribution of the measured values The throughput of the individual memory accesses of 1 MiB is visualized in histograms, that means the x-axis shows intervals (called bins) of observed values and the y-axis the number of samples which fit into the given bin.

Results from two nodes and the desktop machine are analyzed in the following; since all nodes reveal comparable behavior the first two are picked. Due to background activity additional memory load might be generated and thus some measurements might be delayed. In these experiments, values which are below the 1th percentile are considered to be outliers. While this value seems high, it ensures that results can be compared between the machines. For every benchmark and machine, two histograms with 100 bins are

[13]The benchmark is called with `./memory-bandwidth $((1000*1000*1000)) 100 1000`

generated: One histogram shows valid data points and another is restricted to outliers. The x-axis of the figure with valid data points is identical for all Westmere nodes – the x-axis is fit to the smallest data point among them.

The diagrams for the read, write and RWW kernel are shown in Figure 3.4, Figure 3.5 and Figure 3.6, respectively. In the read figure the outliers are provided to illustrate how they might look like. The x-axis is given in terms of the throughput that is achieved while accessing a chunk of one Megabyte.

Observations and interpretation

a) Time for accessing one MiB of data varies and variability can be quite large. For example, while it can be seen that most read accesses achieve a performance between 8300 and 8,450 MiB/s (see Figure 3.4c), on West2 some accesses achieve just about 2,000 MiB/s. A detailed analysis of the causes is difficult since it involves operating system, processor and its caches, memory controller and the physical memory[14]. Also, as the benchmark measures time for chunks of 1 MiB of data, variability of accessing individual cache lines cannot be revealed. If this variability would be statistically independent, these effects should be distributed evenly among the large blocks and result in a similar performance. Since performance variability is quite large in some cases, it can be concluded that either hardware behavior changes for longer periods – for example, when a memory controller is saturated – or a single operation is interrupted for a longer period. For example, the OS could suspend the benchmark for a while to dispatch background processes on the same core, or the processing of hardware interrupts could defer execution.

b) For the read and the write kernel the observed histograms of both cluster nodes look similar. While the read graph is identical, in the write kernel several data points between 7500 and 7,750 MiB/s occur for West2. The distribution of the laptop can be clearly distinguished from the nodes, and also shows significantly more variability. Interestingly, for the laptop the histograms for read and write are similar – besides the shift of the read performance by 100 MiB/s.

c) There are several outliers in the range between 6,500 and 7,000 MiB/s, although there are just a handful between 7,000 MiB/s and 8,000 MiB/s – overall the both graphs look similar (look at Figure 3.4c and Figure 3.4d). Since this behavior is observable for both machines, it is likely that it is caused by platform-specific behavior.

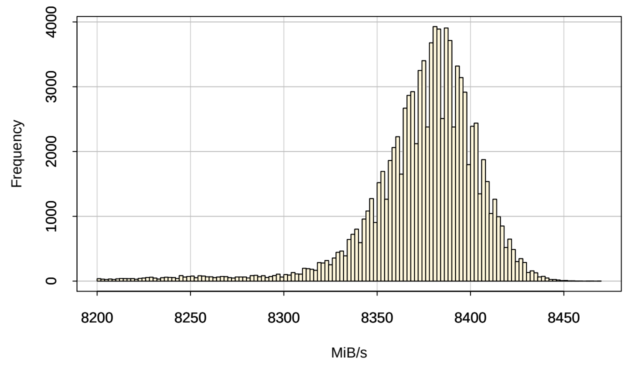
d) The histograms measured with the RWW kernel differs slightly on all cluster nodes. However, the valid data points of a diagram are between 2,800 and 3,150 MiB/s, which is a small range explaining the identical average value.

e) Although the main peak for reads looks almost like a Gaussian distribution, none of the results obtained for a kernel shows histograms in which data is distributed with a normal distribution or another well-known distribution. The different peaks indicate hardware influences such as CPU caching behavior, or they are caused by background noise of the OS and running daemons.

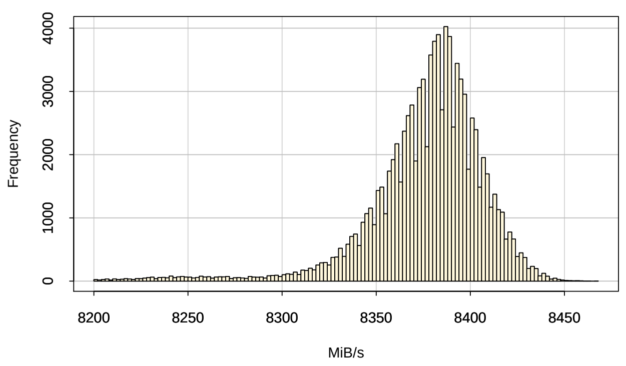
Platform dependency of the results A slightly modified benchmark[15] has been run one year after obtaining the initial results. It turned out that the results for the write kernel (and the RWW kernel) differ compared to previous results, although the read results are still similar to the previous results. Furthermore, different nodes now show a different behavior under the write kernels. For this reason an extensive test is conducted that expands the previous results.

Each executed run iterates 10 times more often over the data; by accessing a total of 1 TByte of data it is expected that potential background noise is leveraged because in this configuration the initial `memset()` and the write kernel take about 120 s. The experiment is repeated 15 times.

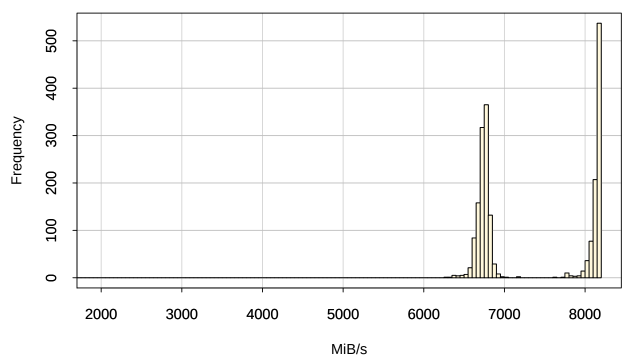
[14]Unfortunately, details about memory controllers seem to be treated confidential.
[15]The benchmark now uses a high-resolution timer.



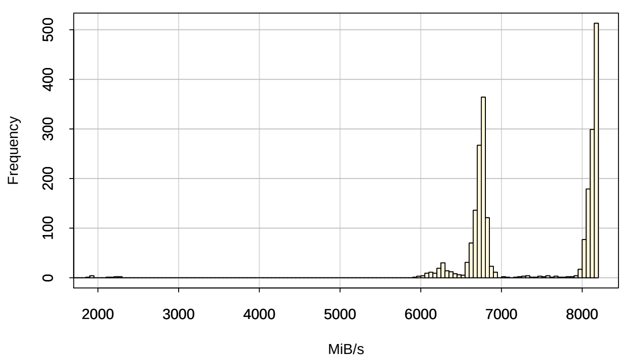
(a) West1 – valid data points.



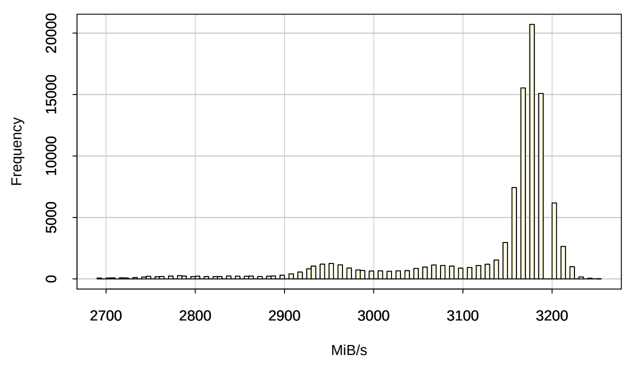
(b) West2 – valid data points.



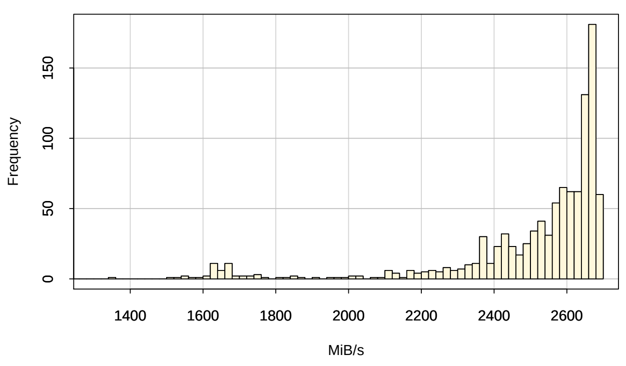
(c) West1 – outliers.



(d) West2 – outliers.

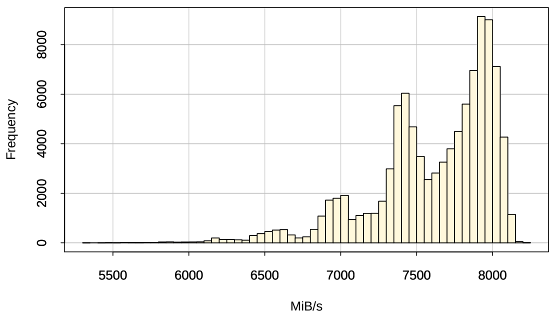


(e) Laptop – valid data points.

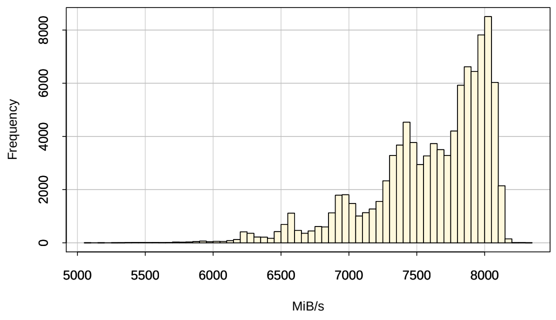


(f) Laptop – outliers.

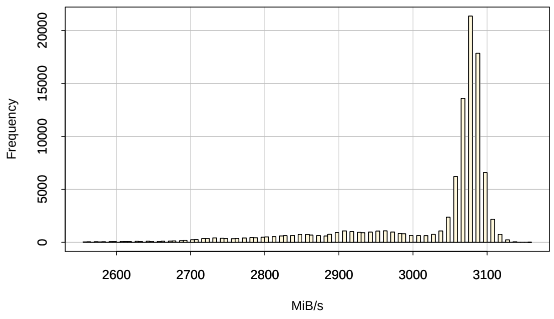
Figure 3.4.: Read performance – histograms show the timings obtained with `memory-bandwidth`.



(a) West1 – valid data points.

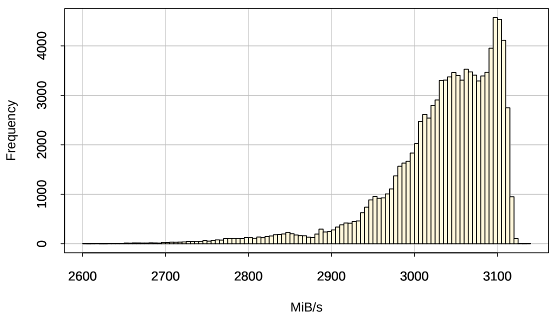


(b) West2 – valid data points.

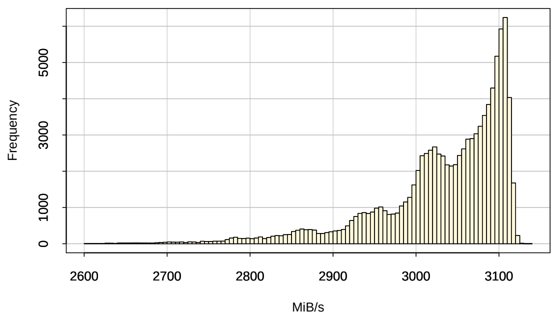


(c) Laptop – valid data points.

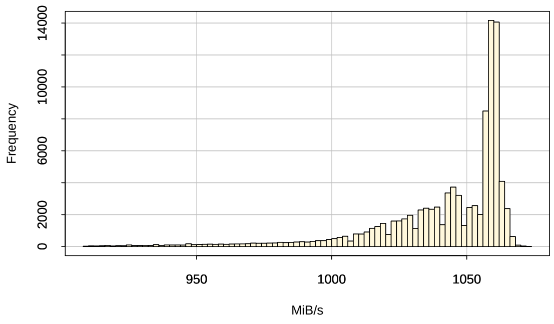
Figure 3.5.: Write performance – histograms show the timings obtained with `memory-bandwidth`.



(a) West1 – valid data points.



(b) West2 – valid data points.



(c) Laptop – valid data points.

Figure 3.6.: RWW performance – histograms show the timings obtained with `memory-bandwidth`.

Experiment / Host	West1	West2	West3	West4	West5	West6	West7	West8	West9
Write	7115	7121	7146	7100	6906	7622	7271	7449	7100
Read	8331	8332	8337	8339	8334	8340	8337	8331	8331
RWW	2936	2936	2942	2931	2888	3036	2968	3005	2931

Table 3.3.: Average memory throughput in MiB/s of 15 runs to access 100 GiB of data with `memory-bandwidth`.

This time by using Likwid the benchmark process is explicitly pinned to CPU 0 on all systems – this ensures that the process is run on the first physical core and never migrated[16]. Like before, Turbo Boost is disabled by restricting the maximum frequency the CPU governor can set.

An overview of the average performance is provided in Table 3.3. The average, minimum and maximum performance for all runs is illustrated in Figure 3.7. Histograms for three repeats of the big runs are presented in Figure 3.8, Figure 3.12 for the read and the RWW kernel – up to 300 buckets are used per histogram to generate diagrams with a fine resolution. Since the write kernel is discussed the most, 5 runs are presented for each node in a diagram for three nodes each (Figure 3.9, Figure 3.10 and Figure 3.11)[17]. For an easier comparison for every kernel the axes of all these diagrams are scaled identically.

Observations and interpretation

a) The read kernel behaves similar to the previous measurements. Overall, the average read performance fluctuates between 8,200 MiB/s and 8,360 MiB/s, but the average across all runs and nodes is about 8,335 MiB/s (look at Figure 3.7a). In Figure 3.8 one histogram is provided for a run on each node. The runs on West1 and West9 has been that result in the minimal observable throughput on these nodes (8,280 and 8,200 MiB/s). While the typical run on most nodes behaves alike, these two nodes result in histograms which values are shifted to the left. These could be caused by the physical memory pages that are actually allocated by the OS.

b) The write histograms look now much different to previous results. Depending on the run there are two to three spikes below 7,200 MiB/s, and a hill between 7,400 MiB and 8,000 MiB. The hill has been measured in the runs before, while the spikes are new.

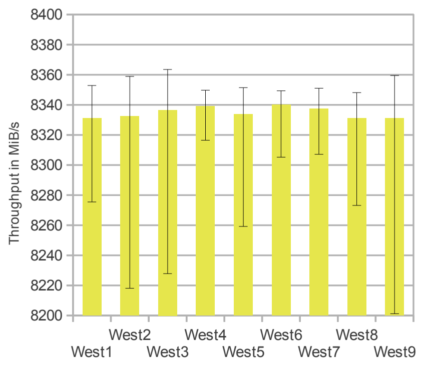
West5 does not show the hill, therefore, the average performance is lower. Apparently West6 is the only node that has not changed its behavior, its histogram is close to the ones previously measured on West1 and West2 (look at Figure 3.5), also the average performance and fluctuation is similar to the previous results. West7 and West8 show an intermediate behavior between the well working behavior of West6 and the behavior of West5.

c) By comparing the five independent runs on one node the variability of the measured data becomes visible. It is surprising to see that the first spike either appears completely or not at all between the runs (compare the runs of West1 to West5, West7 and West9).

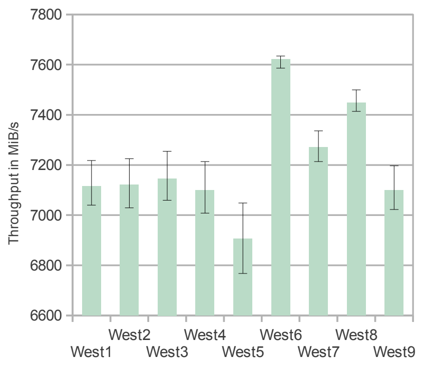
d) Overall the RWW kernel behaves a bit smother than the write kernel, but it mimics the behavior of the write kernel (see Figure 3.12). The fluctuation of performance is identical to the fluctuation of the write performance (compare Figure 3.7b and Figure 3.7c.

e) Since the hardware has not been modified, presumably it looked like the software causes this behavior – while the major version of Ubuntu is still 10.04 the software and kernel has been updated regularly. Also, modifications of the BIOS settings in the meantime seemed possible[18]. The observations suggest that the run has an impact on the result but also that the hardware of the systems behaves now differently.

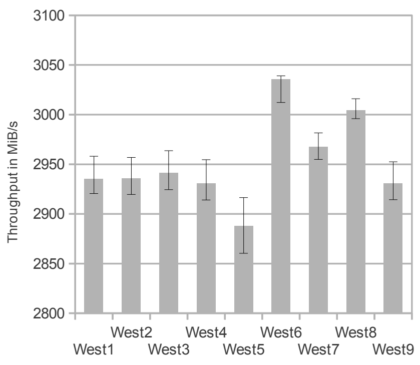
[16]In the initial measurement this fact has not been considered since the behavior is similar on all runs.
[17]The results for West10 are omitted.
[18]The author does not know whether the BIOS has been modified in the year between the measurements.



(a) Read.

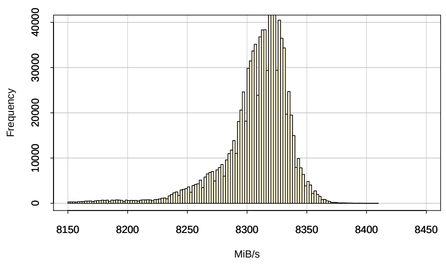


(b) Write.

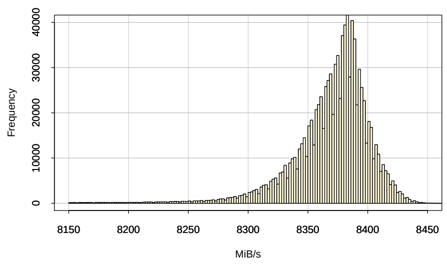


(c) RWW.

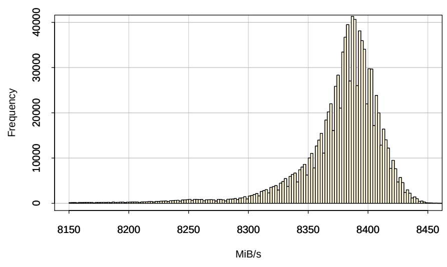
Figure 3.7.: Average memory throughput measured with `memory-bandwidth` – overwriting 1000 times a memory region of 1000 MB (error bars indicate the minimum and maximum of 15 runs).



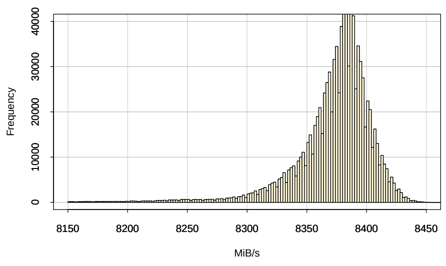
(a) West1.



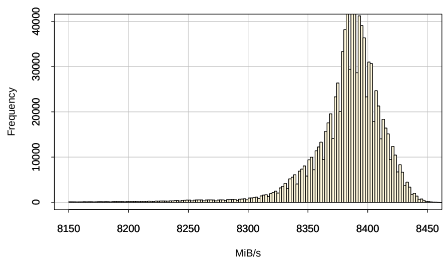
(b) West2.



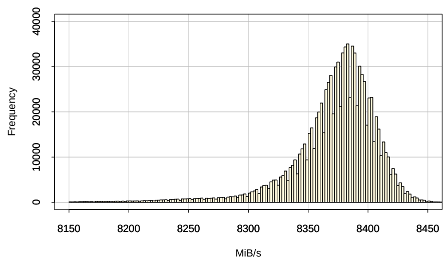
(c) West3.



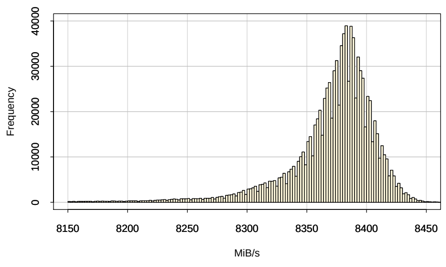
(d) West4.



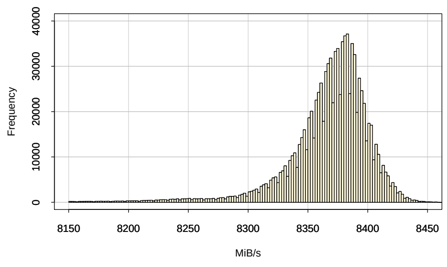
(e) West5.



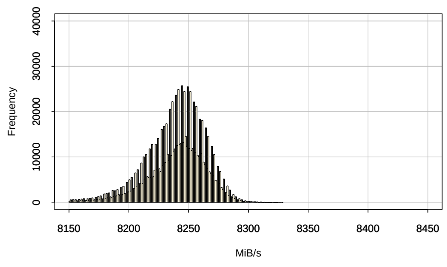
(f) West6.



(g) West7.

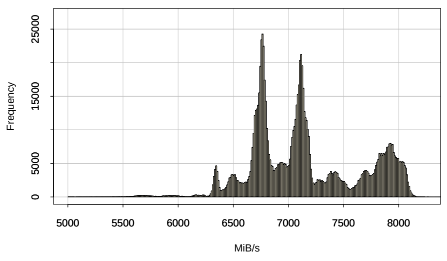


(h) West8.

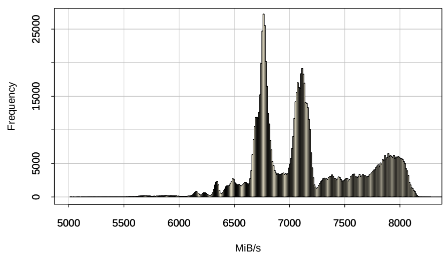


(i) West9.

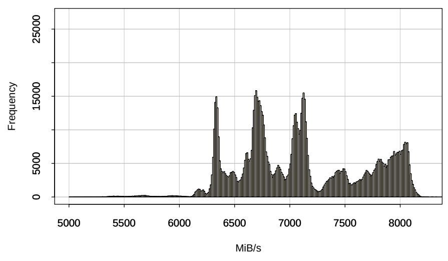
Figure 3.8.: Scaled histograms for accessing 1 TByte of data – read performance.



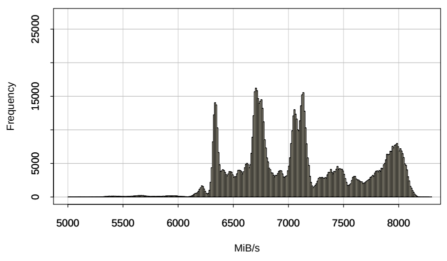
(a) West1 – Run 1.



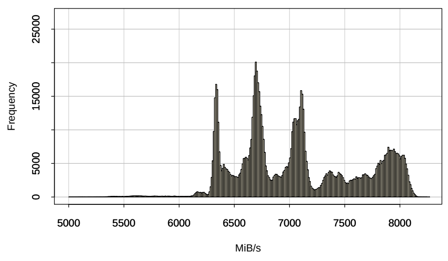
(b) West2 – Run 1.



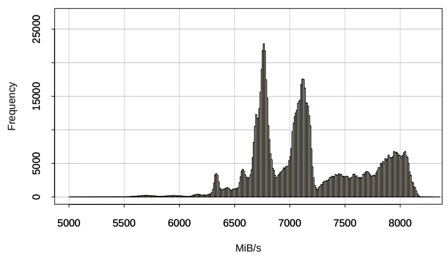
(c) West3 – Run 1.



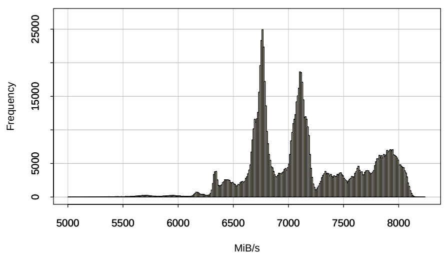
(d) West1 – Run 2.



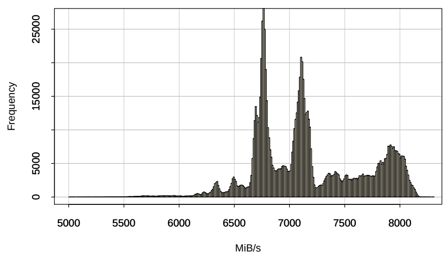
(e) West2 – Run 2.



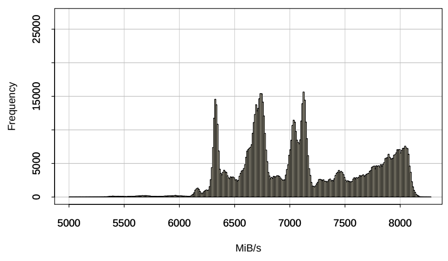
(f) West3 – Run 2.



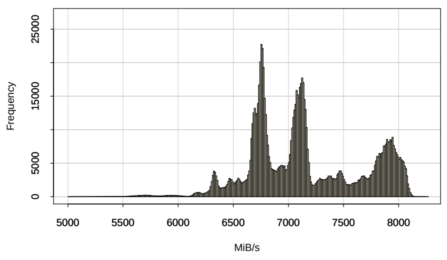
(g) West1 – Run 3.



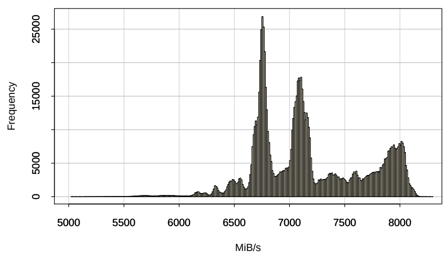
(h) West2 – Run 3.



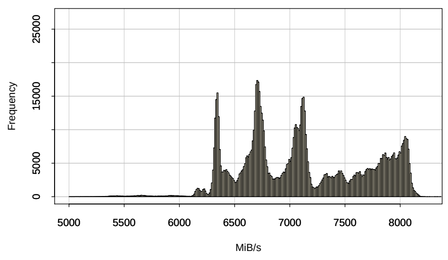
(i) West3 – Run 3.



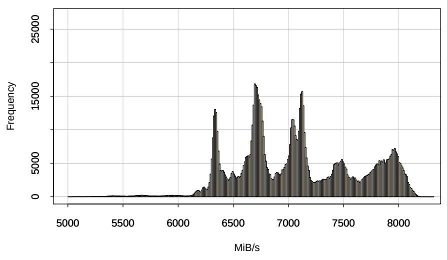
(j) West1 – Run 4.



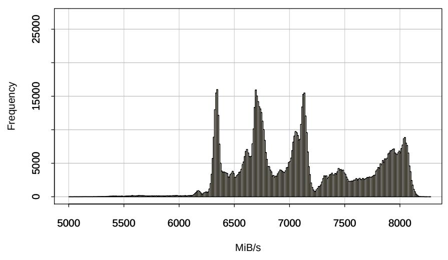
(k) West2 – Run 4.



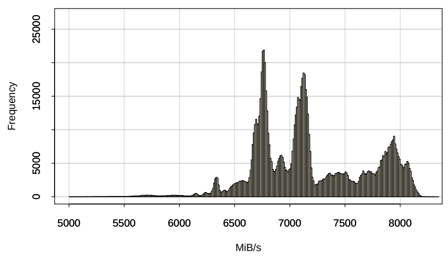
(l) West3 – Run 4.



(m) West1 – Run 5.

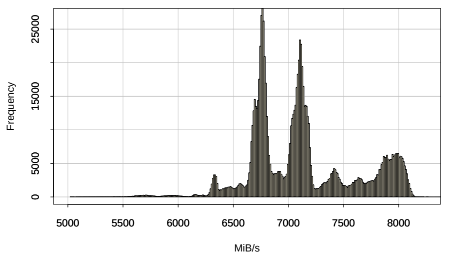


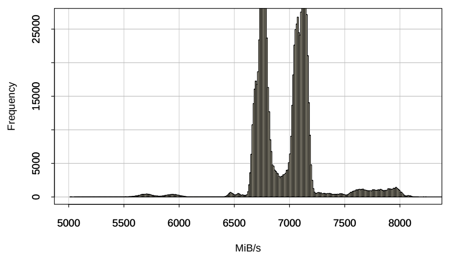
(n) West2 – Run 5.

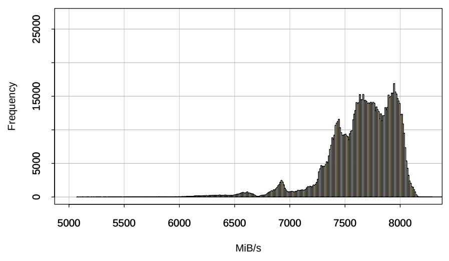


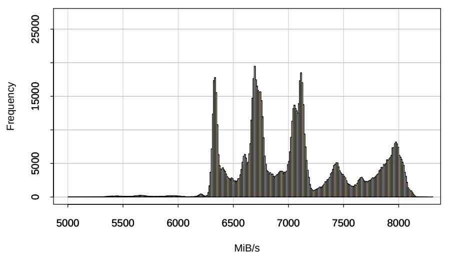
(o) West3 – Run 5.

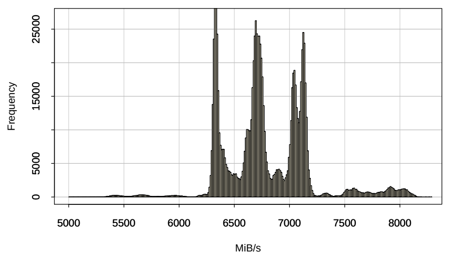
Figure 3.9.: Scaled histograms for accessing 1 TByte of data – write performance (1).

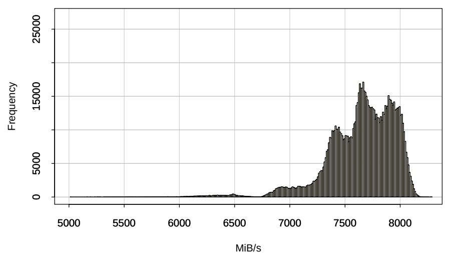

(a) West4 – Run 1.

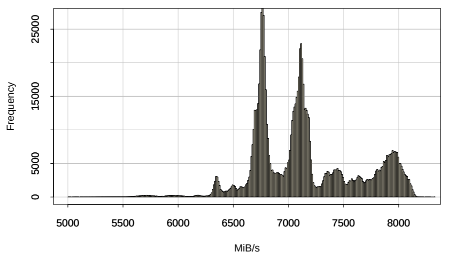

(b) West5 – Run 1.

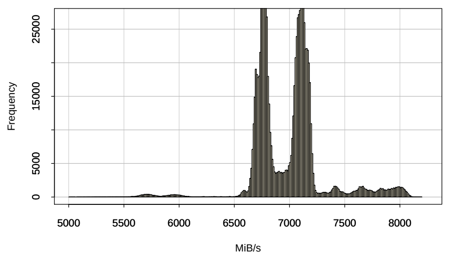

(c) West6 – Run 1.

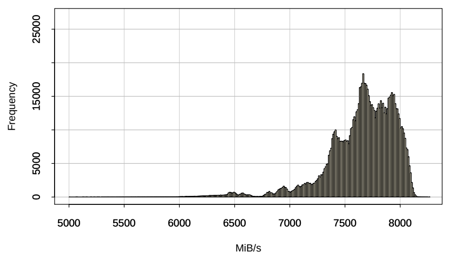

(d) West4 – Run 2.

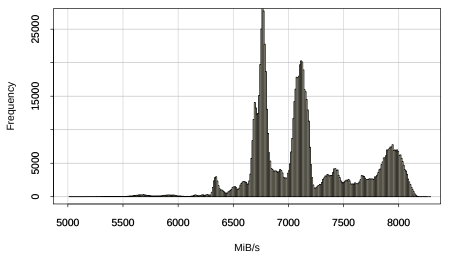

(e) West5 – Run 2.

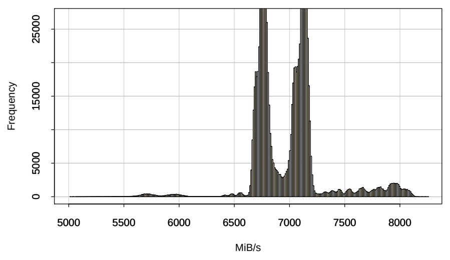

(f) West6 – Run 2.

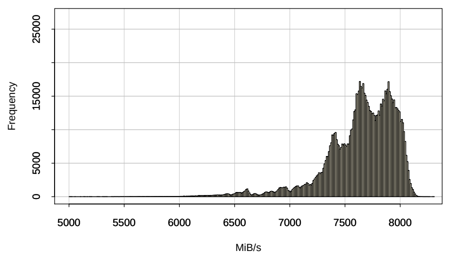

(g) West4 – Run 3.

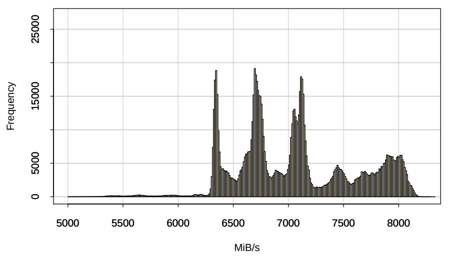

(h) West5 – Run 3.

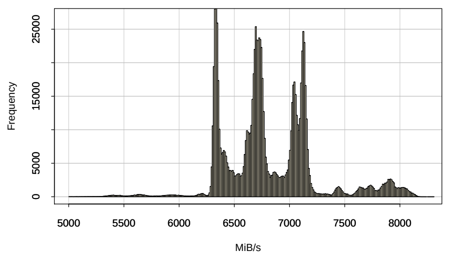

(i) West6 – Run 3.


(j) West4 – Run 4.


(k) West5 – Run 4.


(l) West6 – Run 4.


(m) West4 – Run 5.


(n) West5 – Run 5.

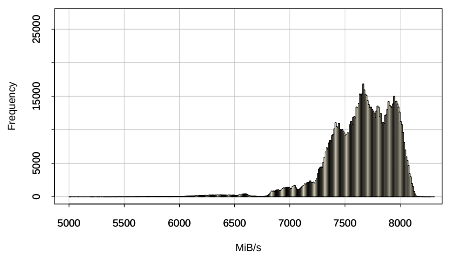
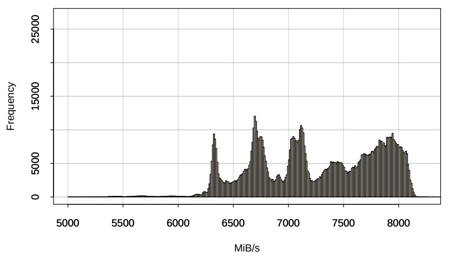
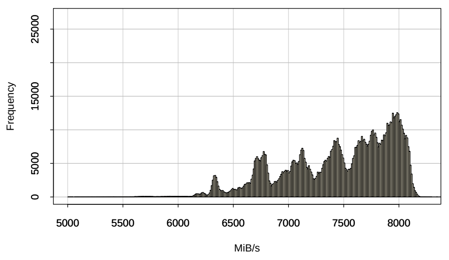
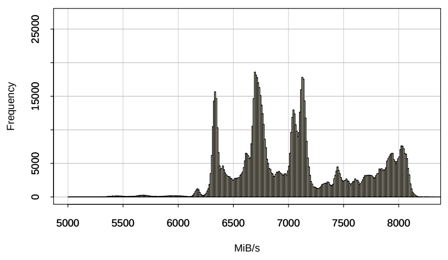
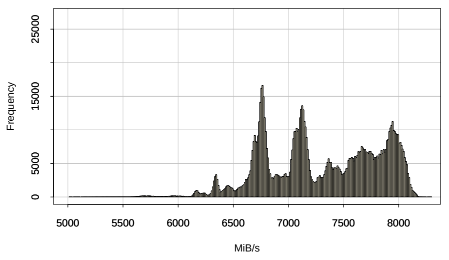

(o) West6 – Run 5.

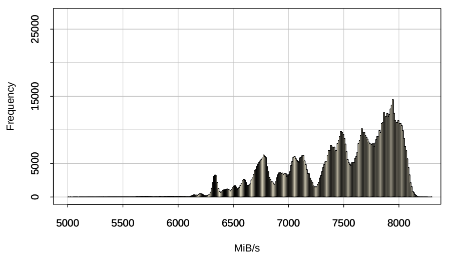
Figure 3.10.: Scaled histograms for accessing 1 TByte of data – write performance (2).

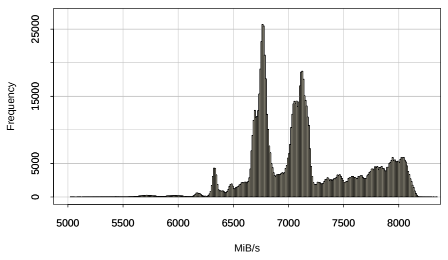

(a) West7 – Run 1.

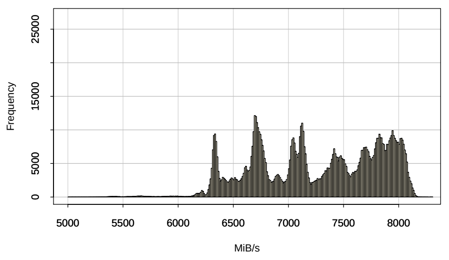

(b) West8 – Run 1.

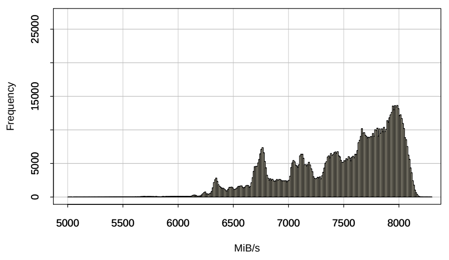

(c) West9 – Run 1.

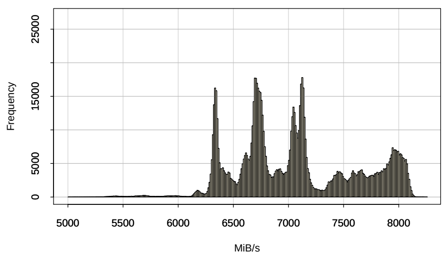

(d) West7 – Run 2.

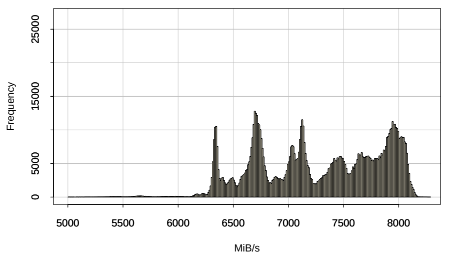

(e) West8 – Run 2.

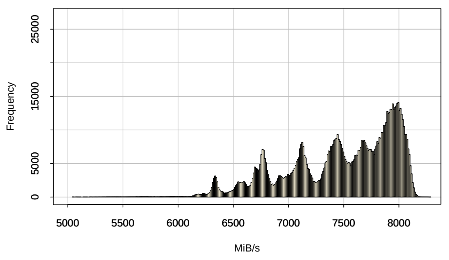

(f) West9 – Run 2.

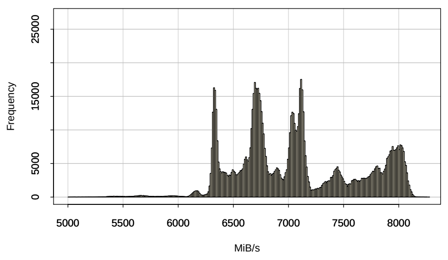

(g) West7 – Run 3.

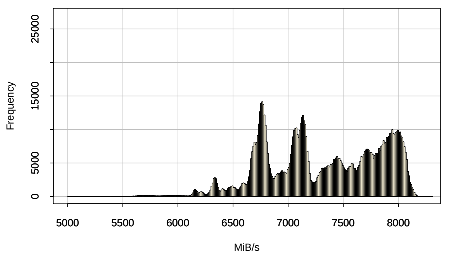

(h) West8 – Run 3.

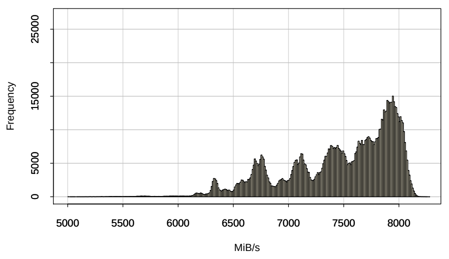

(i) West9 – Run 3.


(j) West7 – Run 4.


(k) West8 – Run 4.


(l) West9 – Run 4.


(m) West7 – Run 5.


(n) West8 – Run 5.

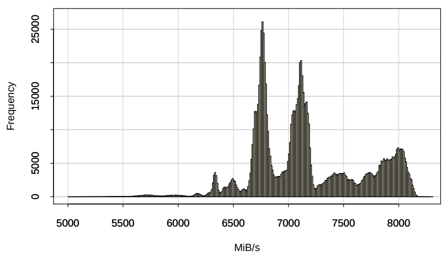
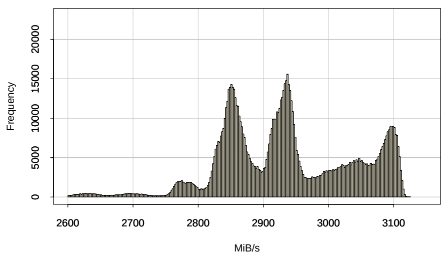
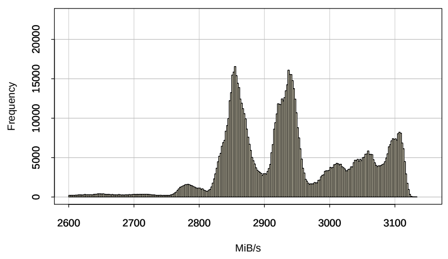

(o) West9 – Run 5.

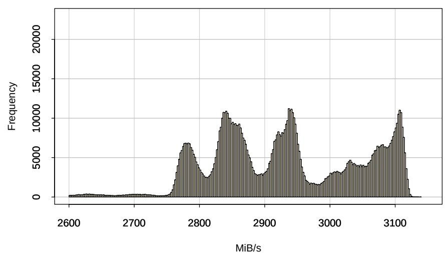
Figure 3.11.: Scaled histograms for accessing 1 TByte of data – write performance (3).



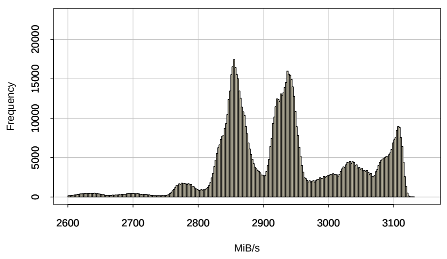
(a) West1.



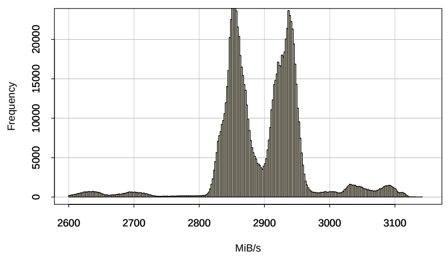
(b) West2.



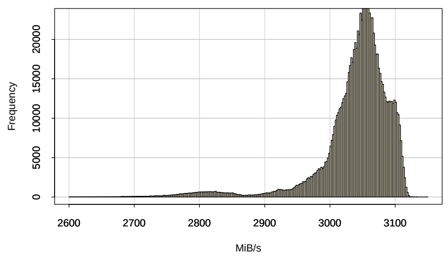
(c) West3.



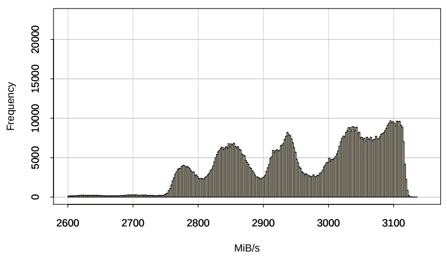
(d) West4.



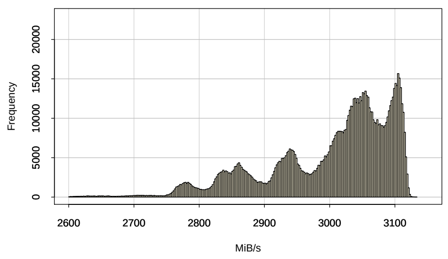
(e) West5.



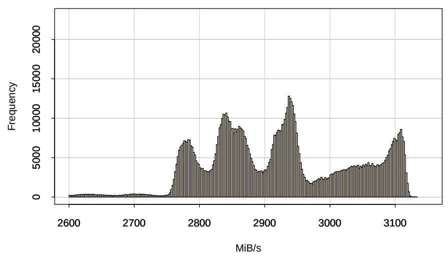
(f) West6.



(g) West7.



(h) West8.



(i) West9.

Figure 3.12.: Scaled histograms for accessing 1 TByte of data – RWW performance.

Potential reasons for observed variability In general, there are two potential sources for variability, the hardware and the software. Here is a short discussion of imaginable causes for the measured histograms. This list is not claimed to cover all causes, but it shows the complexity of the matter.

1) As the nodes are setup with the same hardware, BIOS revision, software and the configuration, these cannot influence the behavior. However, minimal fluctuations during the production process of the hardware might lead to a variability in microscopic behavior that is visible in the initial measurements of the write kernel. This even leads to variation in the thermal property of a multicore processor [CK11]. The process variability and a strategy to mitigate it is discussed in [RBOS08]. Microprocessor vendors cover a large part of the variability by testing the processor and by determining a frequency at which it can operate correctly. Still microprocessors are subject to timing fluctuations. The variability can be caused by phase lock loop (PLL) jitter, clock distribution skew, across chip line width variation and power supply noise [FRJ$^+$07, Mak08].

2) The mapping of the virtual memory pages to physical memory pages might be a cause – the assignment determines which memory controller to use and how data is put onto the physical DIMM. By using the first-touch strategy the kernel allocates memory pages on the local memory controller, this has been verified by looking at the performance counters with LIKWID[19]. This theory could be true for the new results, because individual runs show an additional peak with a lower performance. Also, the aging system and ECC-DRAM might change its behavior to ensure that data is written properly.

3) While an interruption of the OS can delay only access of a single memory chunk, congestion on the memory channel due to background activity could either cover a single access or delay multiple subsequent timings. Overall, periodic activity is likely to manifest itself in a peak of the histogram. Note that a throughput of 8,400 MiB/s corresponds to a duration of about 110 μs, between 6,500 MiB/s and 7,000 MiB/s the time to access one MB of data differs by a mere of 10 μs. Therefore, an influence is searched that delays operations by that extent.

4) Automatic C-state switching in the CPU or power saving strategies in the memory subsystem could also increase the time for individual operations – CPU frequency scaling alters the processing speed depending on the CPU load. However, the benchmark always uses 100% of the CPU time, therefore from the OS perspective CPU frequency is set the highest possible and automatic switching of CPU states is at least not initiated by the OS[20].

5) The scheduling strategy of the OS could have a major impact. Apparently, whenever a background activity such as an interrupt or background daemon disturbs access of a chunk, the processing takes more time. Background traffic on the memory subsystem can defer pending operations. Depending on the duration of an interruption the throughput of the measurement is degraded. With `vmstat` a low number of context switches and interrupts can be observed during the experiment (about 131 interrupts and 230 context switches per second). Apparently, they cannot be the cause of the overall behavior on the Westmere nodes, because the write kernel needs about 12 s to complete leading to at most 4200 accesses that are influenced by interrupts and context switches[21].

6) The Nehalem architecture provides hardware prefetchers to load the required data into the caches[22]. This could leverage potential latencies of the memory and the cache hierarchy.

7) The measurement by itself and the timer could also be a source of systematic mistakes. However, the initial results are consistent and many aspects of correctness have been validated by using LIKWID or by comparing obtained results with results measured by other benchmarks.

[19]The benchmark is run with `likwid-perfCtr -g MEM -c 0 likwid-pin -c 0 memory-bandwidth <ARGS>`. This shows that remote read bandwidth is around 150 KiB per second.

[20]This is verified by inspecting the pseudo file `/sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state`.

[21]Under the assumption that every access takes at most the same amount of time than the duration of an interrupt and context switch, this is feasible, because the performance in the histograms does not vary that much. Also, there are 12 cores in the system, which makes it unlikely that all the reported interrupts are executed on the first CPU.

[22]See `http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`.

Conclusions A consequence of the presented histograms is that common probability density functions such as the normal distribution do not suffice to describe memory performance of our system. Whenever processing of the benchmark is disturbed, the processing of the currently accessed memory chunk takes more time; periodic operations manifest in a peak. Initially, the observed noise revealed by the benchmark mostly behaves alike on all our cluster nodes, which is expected because identical hardware and software is deployed. However, performance measured with the RWW kernel differs between the nodes, although the nodes are equipped with the same memory system. Presumably, the cause of the observed variability is the hardware, especially the interplay between CPU cache, memory controller and DRAM. The degraded performance and change in the histograms under the new measurement is interesting, somehow this might be related to the aging system.

With the help of fine-grained histograms differences between systems become visible. By analyzing the latencies of basic operations one could derive a probability density function, which classifies the system and OS noise. We used this technique already to fingerprint systems and to classify LAN and Internet hosts by measuring ICMP-echo round-trip-times [KNL10]. The referenced paper also gives some hints how the *probability density functions* (PDFs) can be estimated and different PDFs can be compared quantitatively; the visual inspection done in this thesis is just a qualitative analysis.

However, as the performance for accessing a single chunk of memory varies depending on the node and run, a simulation of memory access by using a PDF for the distribution of access will not lead to accurate results. Presumably, an accurate simulation of memory throughput is expected to fail without knowing the exact system state and without simulating all the involved components in details. Clearly, this is infeasible.

Since I/O and network communication use memory operations to transfer data a slight variability is expected in these operations, too. It is probably impossible to determine if an observed performance fluctuation of these operations is caused by a fluctuation in memory or by the additional processes and technology that are involved.

Overall, the real cause for the observations are unclear and require further research. Unfortunately, an in-depth analysis of the hardware is out of the scope of this thesis. OS noise analysis for computation has been done for other systems in the past (for example see [TEFK05]).

3.5. Inter-Process Communication

When data between two processes is exchanged three placements of the processes can be distinguished: The processes are placed on the same processor socket, on two disjoint sockets of one cluster node, or the processes reside on two disjoint nodes. Consequently, the behavior of those three process mappings is assessed. Communication with a placement on two disjoint nodes is referred to as *inter-node* communication, with a placement on two sockets of a single node it is referred to as *inter-socket* communication, and placement on one socket is referred to as *intra-socket* communication.

In inter-node communication the deployed network technology on the cluster is Gigabit Ethernet and TCP/IP is used as a communication protocol in all conducted experiments. A Netgear *GS748TS* switch interconnects the components [Net08]. In intra-node message exchange the MPI implementation can choose to either transfer messages by using shared memory, or by using TCP/IP over the loop-back device.

The general MPI communication behavior is measured with the benchmark `mpi-network-behavior`, the benchmark is part of PIOsimHD and introduced in Section 3.5.1. Latency and throughput are important characteristics to describe data exchange (as discussed in Section 2.2). The performance of those characteristics is assessed in two separate sections, in Section 3.5.2 and Section 3.5.3, respectively. In those sections the results of `mpi-network-behavior` are validated and assessed with theory and verified by measurements obtained with third-party tools. Finally, the network communication behavior of `mpi-network-behavior` is assessed for a variable message size in Section 3.5.4. The analysis includes a discussion of the observed fluctuation.

3.5.1. Description of `mpi-network-behavior`

The pseudo code of `mpi-network-behavior` is given in Listing 3.1. With `mpi-network-behavior` the time for `MPI_Sendrecv()` and for the MPI PingPong is determined for message sizes with a power of two (between 128 bytes and 8 MiB and also for empty messages). The PingPong kernel uses `MPI_Send()` on one process to sends data to the second process. Then the role changes; the first process waits for the response of the other process with `MPI_Recv()`. Thus, the operation is analogue to the `ping` utility.

To ensure a long enough runtime the benchmark operates in two phases: First, the number of repeats is measured and set high enough to run for at least 1 s, at the same time this parameter is determined a multiplier is set which defines the number of operations which are timed together. The reason is that the system timer might be not accurate enough[23], hence as many operations as necessary are aggregated to take around 0.1 ms.

In the second phase four repeats over the determined parameters lead to a rough run-time of 8 s per configuration. After that, all measured timings are output for further analysis with `R`.

3.5.2. Throughput

The throughput is one major characteristic of the network. TCP/IP throughput between two nodes has been measured with `iperf`, `bandwidth` and with `nuttcp` to provide a reference. All these tests achieve a comparable average performance of about 70 MiB/s.

Throughput of interprocess communication is also measured with a simple MPI benchmark called `mpi-communication-test`, which is derived from `mpi-network-behavior`. In contrast to `mpi-network-behavior` this benchmark just starts 100 times a message transfer of 1,000 MiB data and times every transfer individually. PingPong or Sendrecv kernels are supported and thus in total 2,000 MiB of data is transferred in every iteration.

To test the variation between nodes and the robustness of the benchmark, it is run on three different node configurations. Quantitative results are provided in Table 3.4. The minimum and the maximum throughput is listed as well. Additionally, the measured average intra-node performance of all runs is visualized for MPICH2 and Open MPI in Figure 3.13.

Observations and interpretation

a) Achieved throughput of the inter-node communication sticks behind expectations. With Gigabit Ethernet performance is limited to a bandwidth of 1000 Gbit/s. TCP and the frame header add overhead to the potentially achievable throughput of 125 MiB/s. While in early tests a throughput of 117 MiB/s has been measured on the nodes, recent measures on the cluster yield a degraded performance of just around 70 MiB/s with all programs. Many software and kernel updates have been installed since the initial configuration, therefore, it is likely that a software configuration limits performance. The latest upgrade to kernel 2.6.38 improved performance from an average 67 MiB/s to the current 70 MiB/s limit. Recompilation and testing of various configure flags and the attempt to tune the Linux kernel and its TCP options[24] did not increase performance further, though.

Theoretically, the switch could limit performance. However, the deployed Netgear *GS748TS* switch offers full-bisection bandwidth of 96 GiB/s according to its documentation [Net08]. To validate the

[23]Internally, the benchmark uses `gettimeofday()` to determine the system time, this function returns the time in microseconds. However, testing revealed that some systems do not return time in microsecond accuracy, which includes our system.

[24]Refer to the kernel documentation (`networking/ip-sysctl.txt`).

Listing 3.1: Pseudo code of `mpi-network-behavior`

```
for size in [ 0, 128, 256, ... 8 MiB]:

  // First phase:
  // Adjust count until at least 1s is spentin the test.

  // Start with at least 1 iteration.
  count = 1
  while(true):
    start timing
    for i = 0 ; i < count ; i++:
      execute communication kernel (depends on the test: PingPong or Sendrecv)
    compute elapsedTime

    MPI_Allreduce( elapsedTime, MPI_MAX, MPI_COMM_WORLD)

    if elapsedTime is larger than 1s:
      break
    if runTime == 0:
      count = count * 10 // Quickly increase the number of operations.
    else:
      count = count / elapsedTime * 2 // After tuning the time should be around 2s.

  // We found the number of operations to run.

  // Determine multiplier, around 0.1ms per batch of operations.
  if count > 10000:
    multiplier = count / 10000
  else:
    // On average a single operation takes longer than 0.1ms.
    multiplier = 1


  // Second phase, execute with determined parameters:
  // 4 repeats, each with approximately 2s duration.
  for r = 0; r < 4 ; r++:
    // Compute elapsedTime for every batch of operations:
    start timing
    for i = 0 ; i < count ; i++:
      execute communication kernel
      for every batch of multiplier operations record and restart timing

  if rank == 0:
    print timing
```

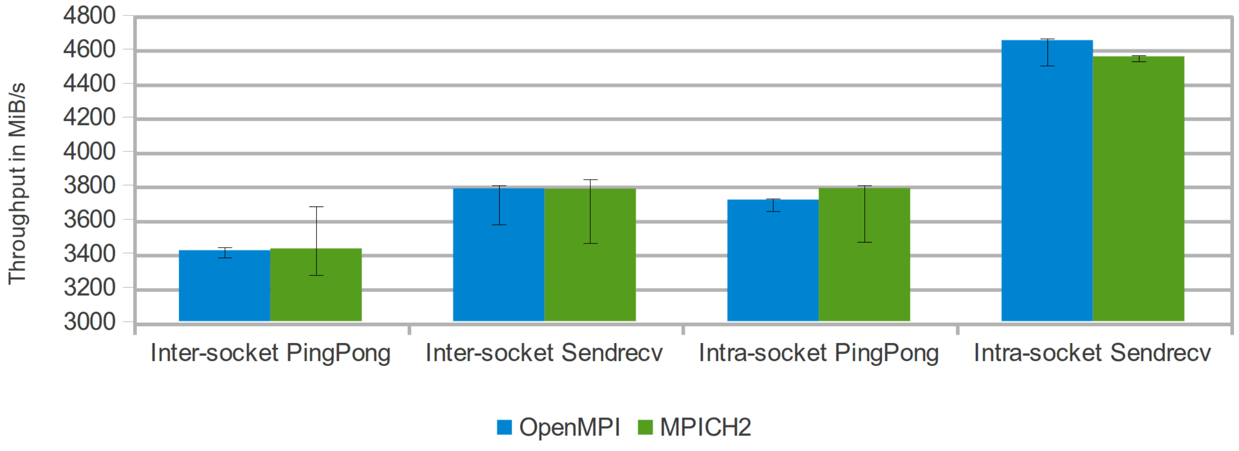


Figure 3.13.: Average intra-node throughput to transfer 1,000 MiB of data measured with `mpi-network-behavior` (error bars indicate the minimum and maximum).

speed the performance in the initial setup with `iperf` has been measured before – processes were started in groups of all 10 nodes concurrently with the result, that the achieved performance scaled linear with the number of pairs. Therefore, it is claimed that the full-bisection bandwidth is available.

b) In the best case, observed MPICH2 inter-node PingPong throughput and `iperf` achieve comparable performance. Since both rely on the TCP/IP stack this is expected.

c) With Open MPI and the PingPong kernel an average performance of 121.6 MiB/s is observable, which is above the theoretical possible performance. In fact, Open MPI uses all available network adapters for communication if possible, and because every node is connected with two independent adapters performance should roughly double. Therefore, the performance of Open MPI should be twice the performance of MPICH2 which is almost achieved.

d) Bi-directional communication between the nodes should double the observable performance, because the deployed Ethernet technology supports full-duplex mode. This can be observed for the average and maximum value of the Sendrecv test with both implementations. However, the minimum value stays on the same level as for the PingPong test. The maximum of the Sendrecv kernel is twice the minimum. Thus, the variance to transfer 1,000 MiB of data is quite high. Also, the bi-directional communication does not improve communication performance of Open MPI further in the average case, but for a few runs the performance doubles to 262 MiB/s.

e) Local communication is much faster than inter-node communication. This is because data can be copied between the processes in memory and does not have to be transmitted via Ethernet. Thus, the memory performance of roughly 4212 MByte/s (measured with `memcpy()`) is a good estimate for shared memory data exchange and verifies obtained results.

f) Open MPI and MPICH2 achieve similar performance in inter-socket and intra-socket communication. The bi-directional communication in memory is roughly 10% faster than uni-directional access. Intra-socket communication is slightly faster than inter-node communication. Performance between two cores of the same socket yields a 20% improvement for bi-directional communication, and 10% for PingPong.

3.5.3. Latency

Latency is determined with the MPI benchmark `mpi-network-behavior` and the measurements are verified with the `ping` utility.

The `ping` program sends ICMP echo requests to the target and measures the time for the arrival of the response – the round-trip time. Half of this duration is considered to be an estimate for the network

Communication	**MPI**	**Benchmark**	**Throughput in MiB/s**		
			Min	Average	Max
Inter-node	MPICH2	PingPong	71.3	71.9	72.2
		Sendrecv	70.8	139.7	147.1
	Open MPI	PingPong	109.1	121.6	124.0
		Sendrecv	112.6	130.3	262.7
Inter-socket	MPICH2	PingPong	3272	3427	3673
		Sendrecv	3456	3778	3834
	Open MPI	PingPong	3373	3416	3434
		Sendrecv	3457	3780	3799
Intra-socket	MPICH2	PingPong	3467	3781	3799
		Sendrecv	4524	4556	4561
	Open MPI	PingPong	3647	3714	3721
		Sendrecv	4502	4651	4661

Table 3.4.: Throughput to transfer 1,000 MiB of data measured with `mpi-communication-test`.

Communication	**Benchmark**	**Latency / operation**		
		Min	Average	Max
Intra-node	`ping -c 500`	5 μs	6 μs	10 μs
	`ping -A -c 5000`	4 μs	4.5 μs	14.5 μs
Inter-node	`ping -c 500`	54 μs	100 μs	125 μs
	`ping -A -c 5000`	27.5 μs	54.5 μs	8 ms

Table 3.5.: Measured network latency with ping.

latency[25]. Latency is measured for inter-node communication and intra-node communication. The experimentally determined latencies including minimum, average and maximum are provided in Table 3.5.

The `ping` utility supports to select the type of request and provides several modes of operation. Two different parameter sets have been used. In the first test a total of 500 packets is transmitted in sequence, by default `ping` waits for approximately 1 s between two consecutive packets to reduce the network load. With the adaptive mode (flag `-A`) `ping` sends an ICMP request once it receives the response for the previous packet. Hence it floods the network with packets[26]. Since this test runs faster it is measured for 5000 packets. Neither of those tests sends more than one request at a time.

Among other tests the benchmark `mpi-network-behavior` executes the PingPong kernel with an empty payload. Thus, the latency can be derived by halving the obtained round-trip-time[27]. The quantitative results of the determined round-trip-time are listed in Table 3.6. The table contains the minimum, the maximum and the quartiles for three independent runs with the benchmark; every run is conducted on a different cluster node.

Observations and interpretation

a) Interestingly, the average round-trip time for the adaptive mode is less than that for a regular `ping` which pauses for 1 s. For communication between nodes the response time roughly halves with the adaptive mode.

b) Average means for the communication of `mpi-network-behavior` is 84.39602 μs, 0.969855 μs and

[25]Actually, the processing by the program and by the operating system causes some overhead. For simplicity this overhead is considered to be part of the network latency.

[26]As a regular user, a small pause is added in this mode to prevent flooding, too. But this delay is not active if the program is run by the super-user.

[27]MPICH2 uses the eager protocol for this kind of messages, therefore, the round-trip time can be obtained.

Configuration	Run	Minimum	1st quartile	Mean	3rd quartile	Maximum
Inter-node	1	67.0 μs	82.0 μs	85.8 μs	90.0 μs	181.0 μs
	2	64.0 μs	81.5 μs	83.8 μs	86.5 μs	263.0 μs
	3	64.5 μs	80.5 μs	83.5 μs	88.0 μs	139.5 μs
Inter-socket	1	0.978 μs	0.9217 μs	0.9282 μs	0.926 μs	1.313 μs
	2	1.032 μs	1.053 μs	1.061 μs	1.058 μs	1.505 μs
	3	0.8986 μs	0.9124 μs	0.9204 μs	0.9217 μs	1.290 μs
Intra-socket	1	0.4148 μs	0.4271 μs	0.4293 μs	0.4292 μs	2.096 μs
	2	0.4253 μs	0.4336 μs	0.4354 μs	0.4357 μs	1.305 μs
	3	0.4130 μs	0.4235 μs	0.4277 μs	0.4277 μs	1.386 μs

Table 3.6.: Round-trip time statistics measured in three independent runs of `mpi-network-behavior`.

4.308 μs, for the inter-node, inter-socket and intra-socket communication. Consequently, the average latency of 42.248 μs is slightly better than the time determined with `ping`. Inter-node communication is much faster, this is due to the fact that shared-memory access is used in MPI but not for `ping`. The ICMP echo requires to communicate via the IP network stack which takes additional time.

c) Inter-node communication has a much higher variance than intra-node communication. The variance of the three configurations becomes clear when the range between minimum, first and third quartile are compared. In intra-socket communication 75% of the timings are between 0.41 μs and 0.43 μs. Communication between two sockets is about twice as high while the range for inter-node communication is between 64 μs to 90 μs (refer to Table 3.6).

d) Observed quartiles vary by the run. In most cases the mean value between the runs behave alike, but the first and third quartile vary by a few percent. The minimum time for the communication between two sockets in Run 2 is even higher than the third quartile of the two other runs. While in general the intra-node communications varies less, this high discrepancy of more than 10% of the average shows the need for an in-depth assessment of the variability between runs. Remember, every test is repeated to run for roughly 4 s. Thus, the fluctuation is expected to be caused by additional background activity, or it may depend on the node the benchmark is executed.

3.5.4. Performance

So far latency has been determined for empty messages, and the potential throughput is computed for very large data transfers. The performance of intermediate message sizes and the overall character of point-to-point communication is discussed in this section.

Therefore, results obtained by `mpi-network-behavior` are analyzed, the benchmark times data transfer for a variable message size between 128 bytes and 128 MiB. Diagrams containing the minimum, the average (mean) value, the first and third quartile are presented to assess the variance of the observations. For every configuration three runs of the benchmark are executed; the nodes the benchmark is run on are varied. If not mentioned explicitly, the presented results show values obtained in a single run. At the end of the section independent runs are compared to assess the deviation between runs.

For inter-node communication two diagrams show the observed communication timings as a reference (Figure 3.14a and Figure 3.14b). Since all figures look quite similar and achieved throughput is more of interest, the performance is computed from the timings and visualized for all cases in Figure 3.14. Throughput can be derived from the message size and the measured timing; 40 bytes are added to the message size to represent the message headers[28]. Thus, throughput is computed with the equation throughput = $2 \cdot (\text{message size} + 40)/\text{measured timing}$.

[28]The minimum size of the TCP header is 20 bytes and the minimum IP header adds another 20 bytes. The MPI envelope (header) depends on the implementation and therefore it is not considered.

Observations and interpretation

a) The logarithmic timing diagrams are hard to assess (look at Figure 3.14b). The maximum values are not plotted since they are much bigger than the average duration. In the computed throughput diagram the maximum can be drawn, because a very slow communication results in a low throughput. Also, the throughput diagrams can be assessed better because the maximum throughput is limited by the bandwidth of Gigabit Ethernet and the memory subsystem.

b) For several message sizes a low maximum throughput can be spotted in each diagram. Messages with less than a few MiB of data are more sensitive to these outliers and longer operations are rather stable. The reasons are fluctuations on the network and OS jitter that disturbs the communication. Larger messages are not influenced to a high degree because the duration of background activity and hardware fluctuations is limited, after some time the data is communicated.

c) Performance increases with the message size and saturates with about 64 KiB message sizes for all kernels and process placements. Overhead of the MPI implementation and latency in the communication path reduces performance for smaller message sizes.

d) The communication inside a socket or between two sockets looks alike (compare the two kernels, e.g., Figure 3.14e). Throughput for intra-socket communication is just higher than for inter-socket communication.

e) In intra-node communication a plateau appears up to 4 MiB of data. It is likely that this is caused by the cache hierarchy: The L3 cache can hold up to 12 MiB of data for all cores, which boosts performance for intermediate message sizes. Additionally, the Sendrecv kernel shows another plateau between 8 KiB and 32 KiB when data fits into the L2 cache. Since data must be copied between the shared memory and a process in both directions at most half of the cache is available for the message.

f) For message sizes below 64 KiB the PingPong kernel achieves half the throughput as the Sendrecv kernel for intra-node communication (compare Figure 3.14e with Figure 3.14d).

The PingPong kernel copies data sequentially from one process to another and then back; in this process data must be copied from the L1 cache of the sender to the L1 cache of the other processor (or main memory). Probably the concurrent processing of the Sendrecv kernel utilizes available L1 caches concurrently, which increases performance. Exact analysis would require to evaluate the CPU cache strategy and put it into relation to the shared memory implementation inside MPICH2, which is out of the scope of this thesis.

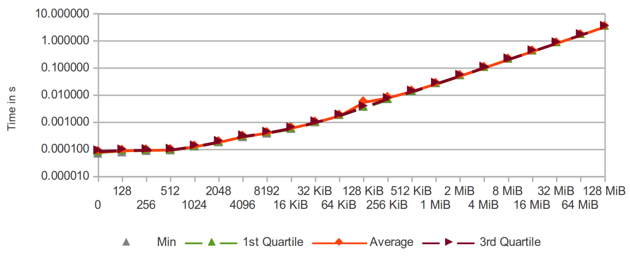
g) A high variance in data ranges between 2 KiB and 64 KiB can be identified for inter-node communication. In order to to understand the deviation between independent transmissions the variability is analyzed next.

Comparing MPICH2 with Open MPI Open MPI runtime is measured with `mpi-network-behavior` and compared to the results of MPICH2 to gain some insight in performance influences of different MPI implementation. In this comparison the average timings over four independent runs are calculated and used for the analysis.

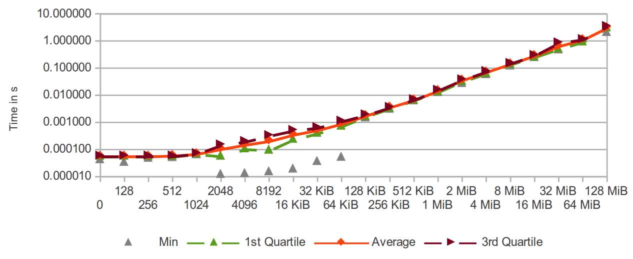
Figure 3.15 shows the throughput of inter-node communication and the relative performance for inter-node and inter-socket communication. To compare the two MPI implementations a relative performance is computed in which the performance measured with Open MPI is divided by the time for MPICH2. Thus, for a given message size values above 100% indicate that Open MPI performs better and values below 100% show that MPICH2 achieves better performance.

Observations and interpretation

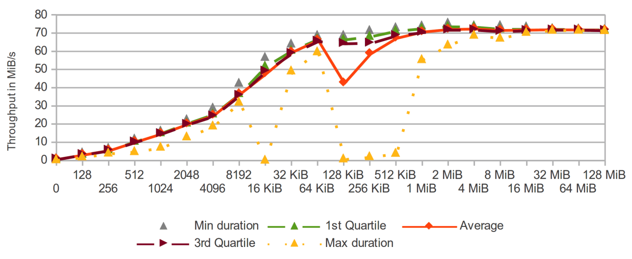
a) Overall, it can be observed that performance is highly depending on the implementation – the performance of both implementations differ by more than 50%. This is much higher than re-runs with



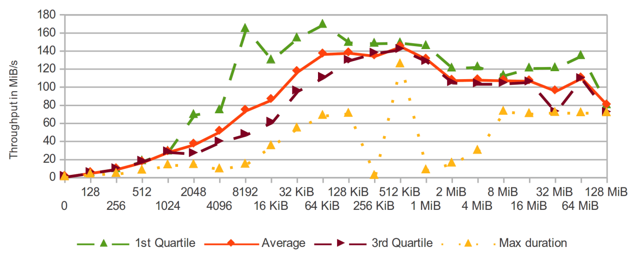
(a) Inter-node PingPong kernel timing.



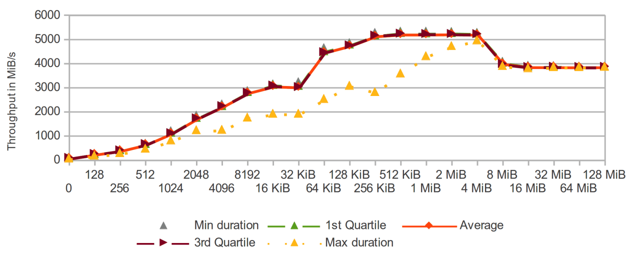
(b) Inter-node Sendrecv kernel timing.



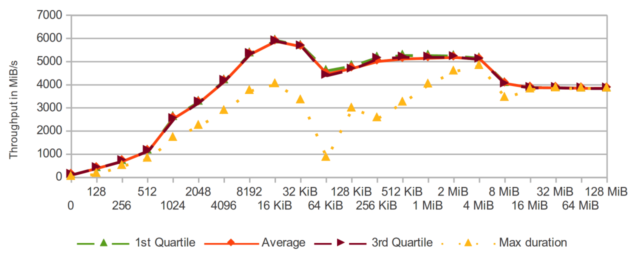
(c) Inter-node PingPong kernel throughput.



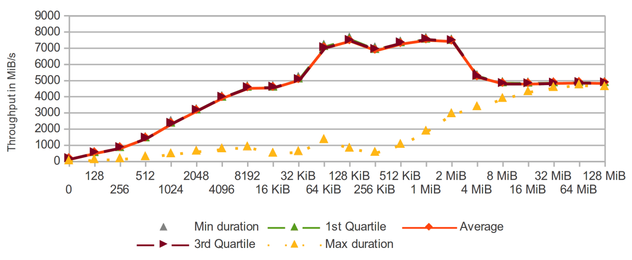
(d) Inter-node Sendrecv kernel throughput.



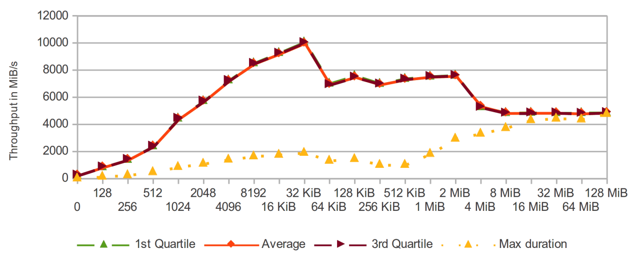
(e) Inter-socket PingPong kernel throughput.



(f) Inter-socket Sendrecv kernel throughput.



(g) Intra-socket PingPong kernel throughput.



(h) Intra-socket Sendrecv kernel throughput.

Figure 3.14.: Communication performance for a variable message size and the three quartiles.

MPICH2 which are in the order of 10%. Therefore, design and implementation details must cause this behavior.

b) The inter-node performance of both implementations is close together up to 64 KiB messages (see Figure 3.15c and Figure 3.15d). Then the behavior changes and OpenMPI is much faster.

c) Starting with larger amounts of data Open MPI needs roughly 60% of MPICH2's time (see Figure 3.15c). The cause for the faster communication with Open MPI is that Open MPI tries to use all available network adapters for communication and since each node has two independent networks performance should roughly double.

d) With 64 KiB messages and the PingPong kernel Open MPI achieves 10% less performance (see Figure 3.15c). The reason is the usage of the eager protocol: Starting with a payload of 64 KiB Open MPI switches to the rendezvous protocol in the default configuration[29], but MPICH2 uses the eager protocol up to 128 KiB. Therefore, two network round-trip times are necessary in Open MPI which reduces the performance compared to MPICH2.

e) With the Sendrecv kernel Open MPI is slower for data ranges between 32 KiB and 1 MiB (Figure 3.15d), transmitting more data improves its performance, while MPICH2's performance decreases.

f) MPICH2 dominates Open MPI in intra-node communication. A comparison is given in Figure 3.15g. The results measured for a message size of 64 KiB and the Sendrecv kernel is actually slightly better for Open MPI. The reason has not been identified for our cluster, but it seems that small message transportation in MPICH2 are handled better by MPICH2's Nemesis communication subsystem (refer to [BM06] for a description of Nemesis). In intra-socket communication the relative performance graphs look similar – additionally, the performance of Open MPI is slightly better for the Sendrecv kernel.

Conclusions In the diagrams a different behavior between the two MPI implementations becomes apparent. The discrepancy of inter-node communication behavior could be partly explained by the features of the implementations (usage of the eager protocol and network devices). However, the low performance of the Sendrecv kernel is surprising, because the hardware is full-duplex capable, and the character of the curve depends on the implementation. In intra-node communication both implementations behave differently, while in overall, Open MPI is worse than MPICH2 the relative performance reveals several spikes and small plateaus. This gives an impression of the influence an implementation has to achievable performance and reveals the non-linear character of the point-to-point communication.

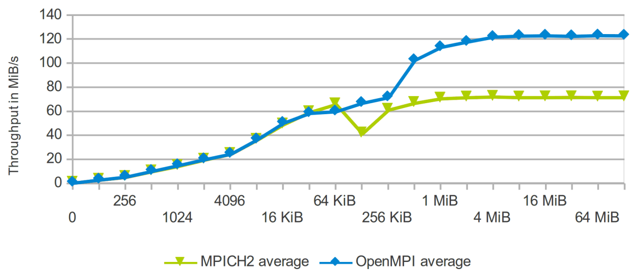
3.5.5. Variability

To assess the variability, the measurement differences of a single run and the results of three runs are compared for MPICH2.

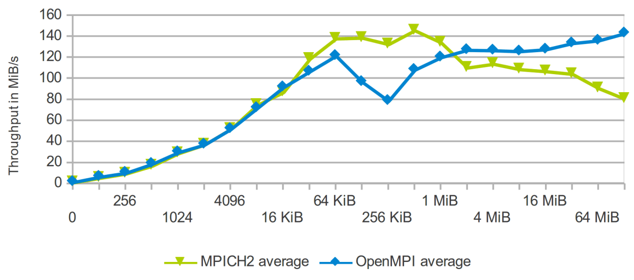
Fluctuations within a single run Diagrams are generated which show the relative performance of the duration for the first and third quartile to the mean value (see Figure 3.16). With the percentage of the relative performance in the diagrams the variance of the experiments can be quantified better than with a logarithmic timing diagram, or a throughput diagram. Presented results just show the values for one run, but are representative for the other two runs.

Observations and interpretation

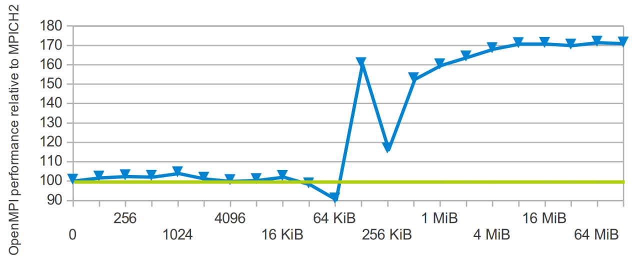
[29]This value can be determined with `ompi_info -param btl tcp`.



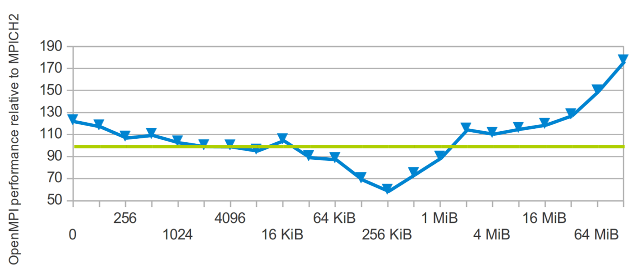
(a) Average inter-node PingPong kernel throughput.



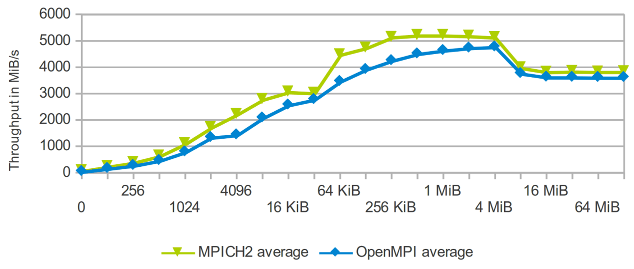
(b) Average inter-node Sendrecv kernel throughput.



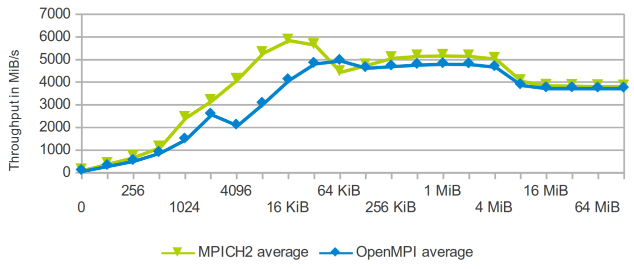
(c) Relative inter-node PingPong performance.



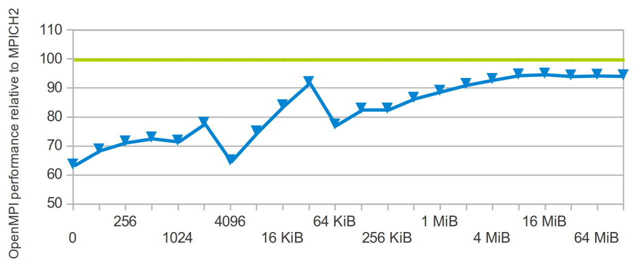
(d) Relative inter-node Sendrecv kernel performance.



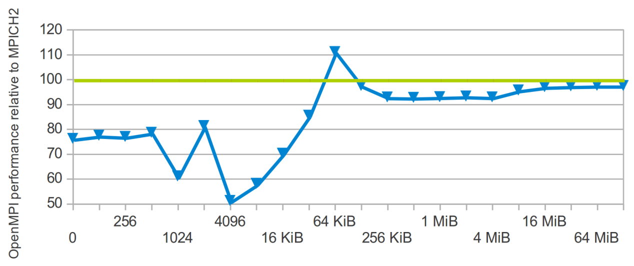
(e) Average inter-socket PingPong kernel throughput.



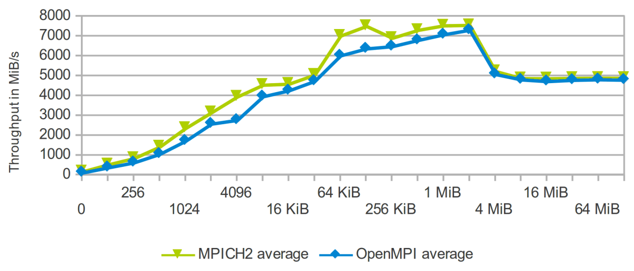
(f) Average inter-socket Sendrecv kernel throughput.



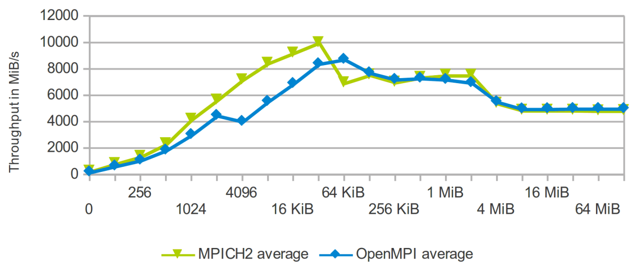
(g) Relative inter-socket PingPong kernel performance.



(h) Relative inter-socket Sendrecv kernel performance.



(i) Average intra-socket PingPong kernel throughput.



(j) Average intra-socket Sendrecv kernel throughput.

Figure 3.15.: Comparison of Open MPI and MPICH2 point-to-point communication performance for a variable message size. The green line marks the 100% relative performance of MPICH2.

a) For most inter-node data transfers the first and third quartile of the PingPong kernel are between 95% and 105% and thus the individual performance varies by more than 10% (see Figure 3.16a). For 128 KiB and up to 512 KiB the third quartile lies below the average indicating that some operations take a long time and thus increase the average significantly. For example, with 128 KiB operations three quarter of the operations need about half the average time. Note that by default MPICH2 starts to use the rendezvous protocol with 128 KiB of data, that might cause the performance degradation.

b) In intra-node communication (Figure 3.16c) this high variance is not observed. Instead most data points are between 99% and 101%. This is also true for the Sendrecv kernel. Therefore, the Ethernet network (and its handling by the OS) has a significant impact on the variability.

c) In the inter-node communication with the Sendrecv kernel a very high fluctuation can be seen when transferring between 2 KiB and 128 KiB of data (see Figure 3.16d). The explanation for the observation is simple – since timing is measured only on the first node background noise could cause an eager message from the second node to be in-flight before the local `MPI_Sendrecv()` is invoked. For this message the process is a so-called early-receiver. Then, if the MPI function is invoked the remote message is already available in a buffer which allows finishing the call quicker than network latency would theoretically permit. Therefore, in general the variance of the Sendrecv kernel is expected to be higher [30]. On average sender and receiver are expected to be synchronized, since one call can only complete when the message from the remote peer arrived.

d) With messages bigger than the limit for the eager protocol the processes are synchronized before data is exchanged. Therefore, it is not possible that a process completes a `MPI_Sendrecv()` while the other process still waits for data. The limit for eager messages is 128 KiB, it can be seen that the variance decreases with messages of that size. The reason for the tight timings up to 1 KiB messages are not completely clear[31].

Timing of individual operations Due to the unexpected high average for 128 KiB and the difference between the average latency between two runs (as listed in Table 3.6) the deviance of message communication times is assessed further. Therefore, a histogram and a few timelines are presented that focus on the fluctuation of inter-node communication. With the help of the timelines the behavior can be assessed over the whole benchmark run. Additionally, the deviations can be assessed over time and outliers can be spotted.

A histogram for transferring 2 KiB with the Sendrecv kernel is given in Figure 3.17d and the corresponding timeline of activity in Figure 3.17b. Every sample counted in a timeline represents one low-level communication by the kernel[32].

For empty messages and for 16 KiB data transfer the timelines of the observations are given in Figure 3.19 and Figure 3.18, respectively. In the diagrams the outliers, that are the uppermost 0.1% of data points, are removed. The diagram for empty messages is plotted with outliers and without to foster the discussion about this methodology. Timelines for intra-socket communication of big messages are provided in Figure 3.20. Other diagrams do not offer substantial new insights and are therefore omitted.

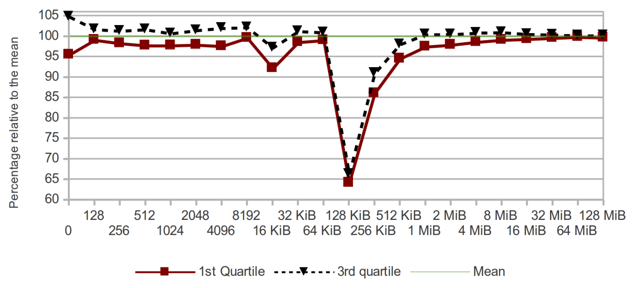
Observations and interpretation

a) Noise of the data transfer becomes visible by looking at the diagrams of the PingPong kernel (Figure 3.17a and Figure 3.18a). The histograms of communicating 2 KiB and 16 KiB of data show an

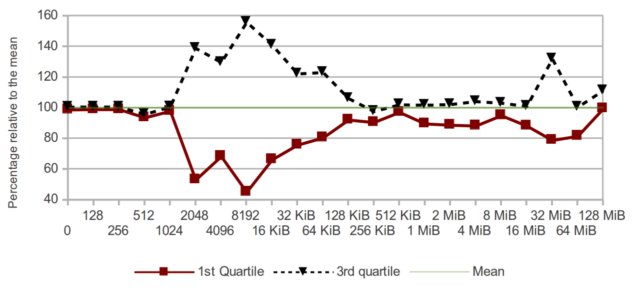
[30] Actually, the issue with early-receives could be mitigated by timing the operation on both processes and computing the average duration for both `MPI_Sendrecv()` operations. However, that would complicate comparison between PingPong and Sendrecv kernel.

[31] Maybe a single network frame is transferred so quick, that both processes operate synchronously. The Ethernet packet transmission packs messages smaller (or equal) to the *maximum transmission unit* (MTU) of the network into one Ethernet frame. On our cluster the MTU is 1500 bytes. TCP/IP controls the fragmentation into packets with a size of at most MTU.

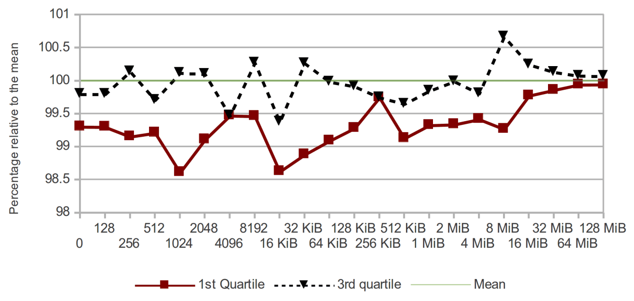
[32] While the number of calls per timing is adjusted to cope with the timing accuracy a single communication needs more time on average than required. Refer to the benchmark description in Section 3.5.1.



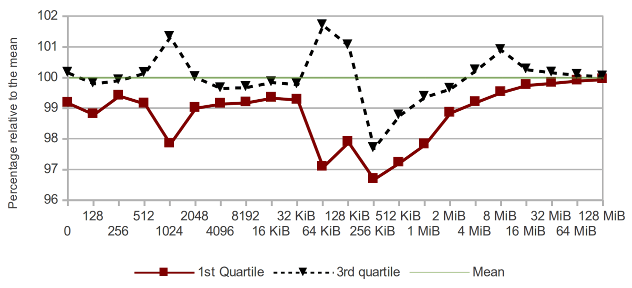
(a) Inter-node PingPong kernel.



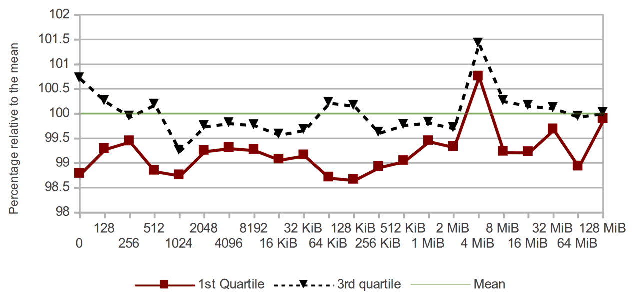
(b) Inter-node Sendrecv kernel.



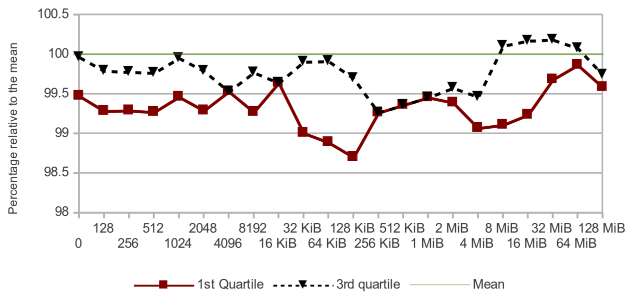
(c) Inter-socket PingPong kernel.



(d) Inter-socket Sendrecv kernel.

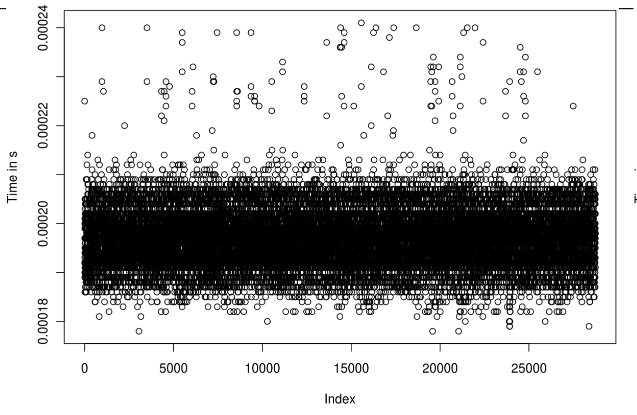


(e) Intra-socket PingPong kernel.

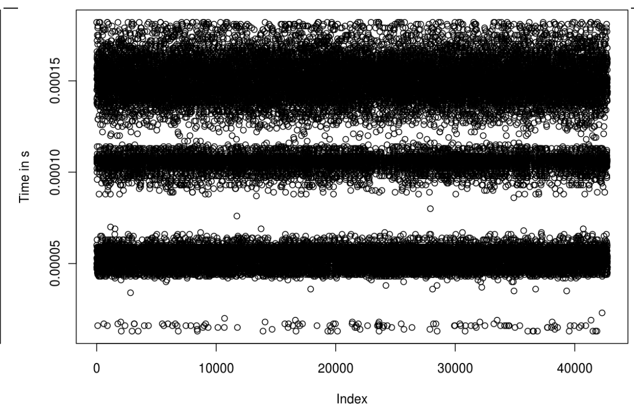


(f) Intra-socket Sendrecv kernel.

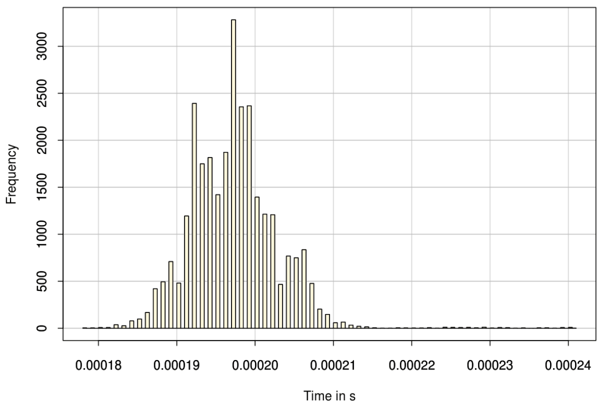
Figure 3.16.: Relative communication time of the first and third quartile to the mean for a variable message size.



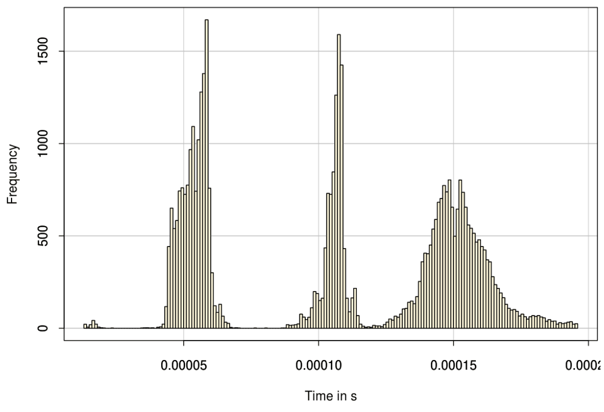
(a) PingPong timeline.



(b) Sendrecv timeline.

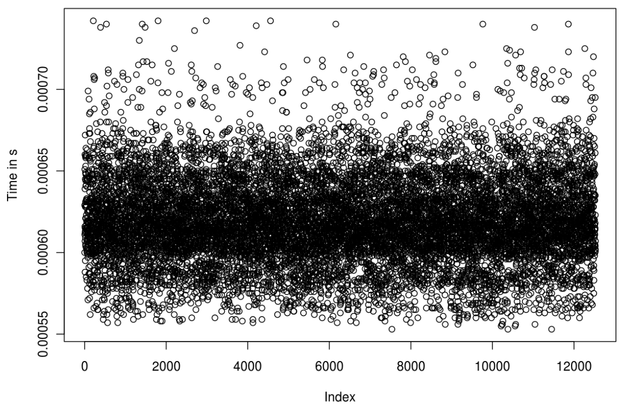


(c) PingPong timeline.

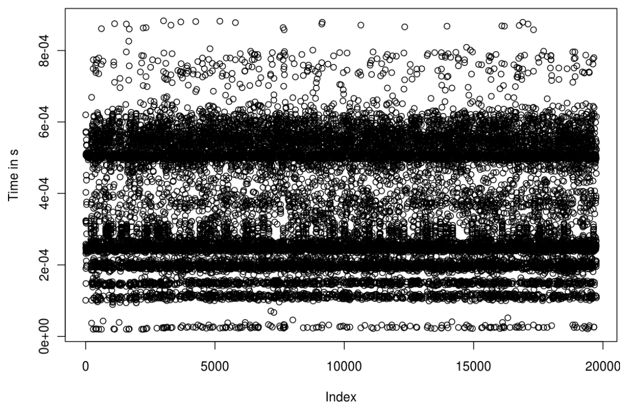


(d) Sendrecv histogram.

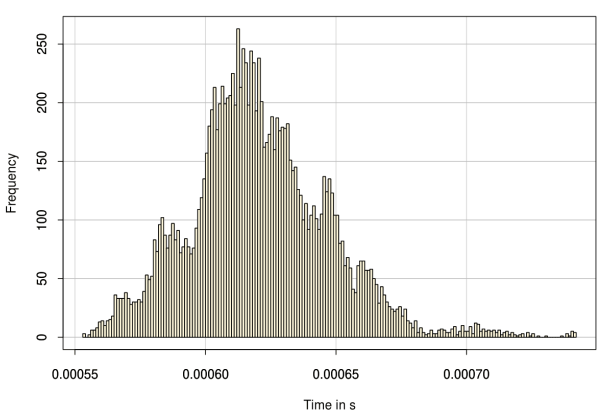
Figure 3.17.: Behavior of the inter-node communication transferring 2 KiB of data.



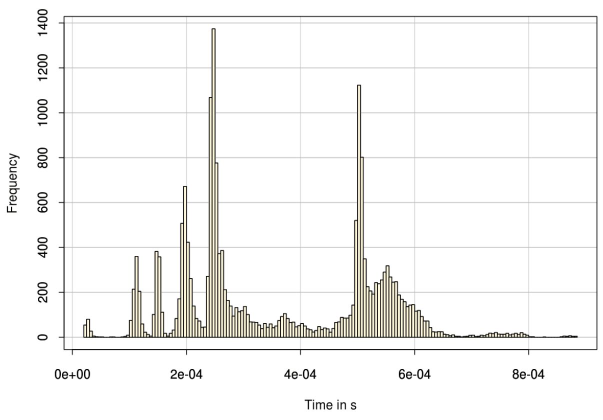
(a) PingPong timeline.



(b) Sendrecv timeline.

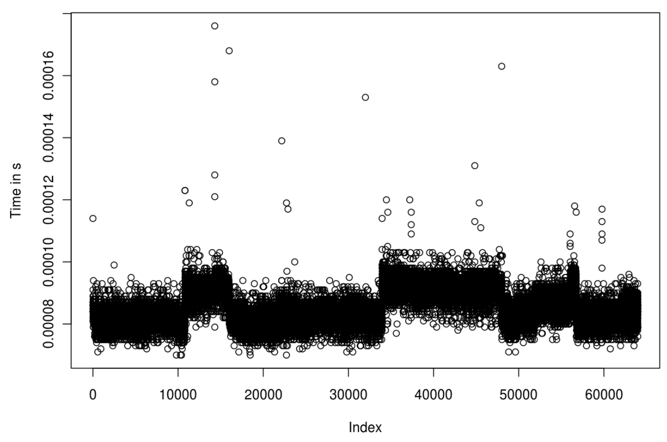


(c) PingPong histogram.

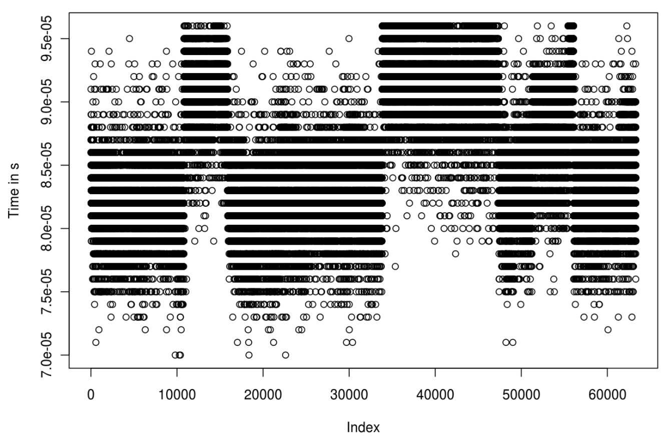


(d) Sendrecv histogram.

Figure 3.18.: Behavior of the inter-node communication transferring 16 KiB of data.

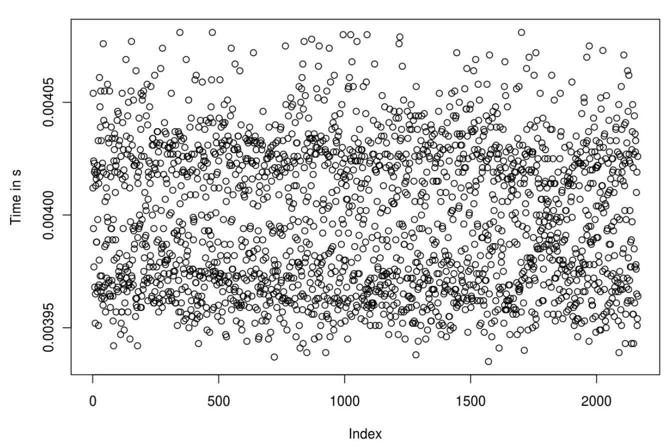


(a) PingPong timeline for empty messages – all data points.

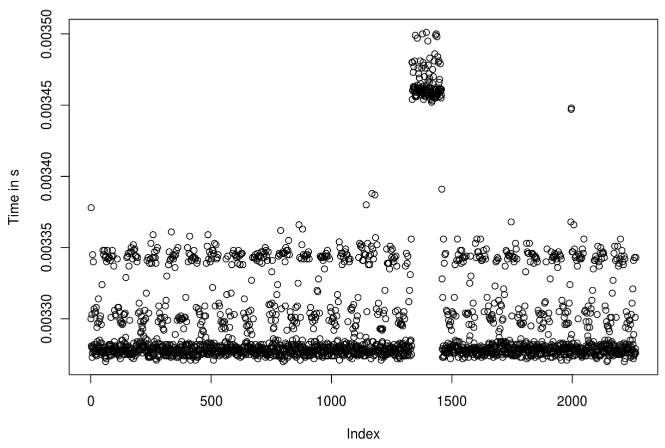


(b) PingPong timeline for empty messages – no outliers.

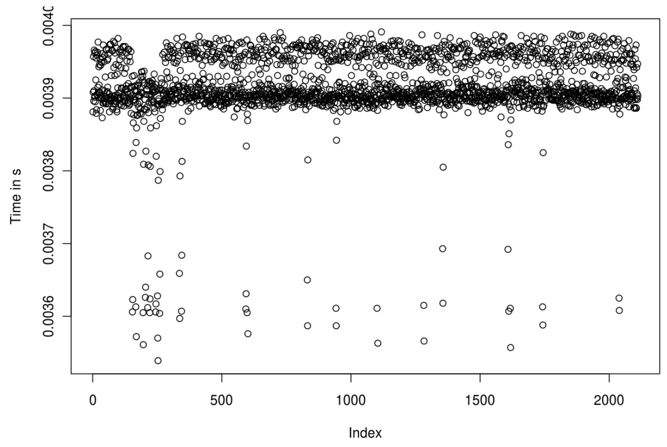
Figure 3.19.: Timelines of the inter-node communication for empty messages.



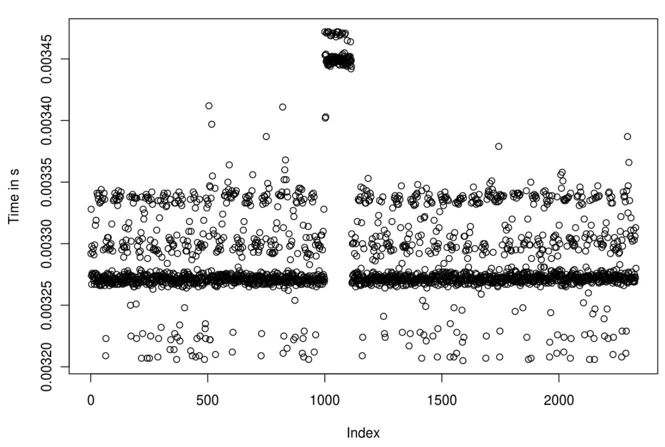
(a) PingPong timeline – inter-socket communication.



(b) PingPong timeline – intra-socket communication.



(c) Sendrecv timeline – inter-socket communication.



(d) Sendrecv timeline – intra-socket communication.

Figure 3.20.: Timelines of intra-node communication for 8 MiB messages.

interesting pattern with several connected spikes. This noise is caused by the interplay of the hardware (e.g., network, CPU and memory) and the software (e.g., TCP, Linux kernel, MPI).

b) Noise is visible in intra-socket communication of large data as well. In Figure 3.20d the timeline for bi-directional communication of 8 MiB of data reveals long-term background activity leading to a plateau with 5% higher transfer time for a sequence of 100 communications (duration of approximately 0.3 s).

It is possible that the processor executed some OS related services such as updating the *Munin* statistics during that time interval. This pattern manifests in many timelines of several intra-socket communications. Therefore, such a hypothesis must be assessed carefully – it is unlikely that the pattern occurs just in those tests and repeatedly.

c) By looking at the timeline for the same case four horizontal bands can be observed throughout the whole diagram (Figure 3.17b). Since the behavior is time-independent, the observation represents background noise caused by the system and/or the network.

Interestingly, the three clusters of the Sendrecv kernel have rather sharp borders (look at Figure 3.17d). The histogram for the Sendrecv kernel of 2 KiB of data in Figure 3.17d shows the time discrepancy that has been observed in the quartiles. The figure does not look like a Gaussian probability distribution instead three clusters can be identified: one at 50 μs, one at 100 μs and one at 150 μs. It is interesting to see that these clusters have a sharp border and thus background noise must have a non-linear influence. They correspond to approximately one to three times network latency (which is about 40 μs). Since at least 2 frames are transported by Ethernet for the 2 KiB Sendrecv kernel, random effects for individual packets should compensate to an average and thus a random pattern close to the one in Figure 3.17a is expected. We will discuss the observation in a minute.

d) Compared to the timeline with just 2 KiB, the bi-directional communication with 16 KiB of data (Figure 3.18b) even more horizontal bands are visible. While the lowest band is caused by a late receiver (all data is already received – consequently time is below network latency), the time for most operations vary between 0.1 ms and 0.8 ms during the whole test.

e) The exact cause of the horizontal bands in the Sendrecv kernels is unknown to the author – but some ideas are discussed in the following. Since the histogram for the PingPong kernel does not show these bands, it is unlikely that our full-duplex Ethernet technology is the cause.

One theory might be that the peaks are caused by the number of early-received packets; the transfer of 2 KiB of data requires 2 packets leading to four peaks: Either none, 1 or 2 packets are ready. At least a few fast operations due to an early-receive of the message are observable in the left bins of the histogram. However, the latency benchmark showed that the average latency is about 40 μs and transferring the 2 KiB of data with 71 MiB/s takes about 28 μs. Thus, the cluster around 50 μs represents the case in which the processes are synchronized – all the slower operations take longer than expected. The time difference between the visible spikes is also quite high – it is unlikely that that much time is needed to receive a single packet. For those reasons the three hills in the histogram are not expected to be caused by early reception of data.

A different view to the data is to compare the Sendrecv data with the PingPong time of about 20 μs. In this sense the last hill of the figure is surprising, because it is at 15 μs which is more than twice the expected time of 10 μs. This is also true for the 16 KiB messages, the highest band takes about 60 μs which is the same duration as exchanging two messages (look at the PingPong kernel). Most likely the MPI implementation requires additional time by executing the send and receive operations suboptimally on our system.

f) Timings for transferring empty messages with the PingPong kernel show a horizontal cluster around 80 μs. Subsequent timings are sometimes offset by roughly 10 μs from the average (Figure 3.19a). To carve out the behavior the handful samples which needed more time are removed in Figure 3.19b. In this figure the restrictions due to the timer resolution are visible, too.

The duration of such a shift is between 5,000 and 10,000 samples (corresponding to approximately 0.85 s). This indicates that some additional activity on the switch or the background daemons is degrading performance. However, the repeated runs reveal a similar behavior of the timeline and the effect lasts for a long time. Activity caused by daemon processes on the nodes is unlikely because these services transfer little data and thus they would last for a short amount of time.

g) This kind of bias is not visible for larger messages, for example see Figure 3.18a. This is surprising, because for larger messages the random behavior of transferring multiple small messages is expected to add up – which is explained in the following. While the payload of an empty message is transferred in a single Ethernet frame, TCP must fragment the 16 KiB message in at least 11 packets. On theoretic consideration is to predict the figure for 16 KiB of data by drawing 11 times from the probability distribution of a single packet (or by drawing 8 times from the 2 KiB histogram Figure 3.17d) – let us ignore latency for the moment. Although random effects would level out speed of individual packets, in this case the periods in which empty messages are transferred faster would add up for larger messages, too. Since this is not observable there are other issues leading to the behavior of empty messages. Presumably they are caused by our Ethernet network or by the operating system.

h) While in communication between two sockets two horizontal clusters can be identified, there are three in the intra-socket communication (compare Figure 3.20c to Figure 3.20d). There are many potential explanations, and due to the previous research some are unlikely. For instance, the behavior of the memory controller discussed previously has not caused this observation in this benchmark. L3 cache behavior, or more likely the OS or the MPI implementation could cause it by handling data structures of shared memory, or they could be due to context switches.

i) By comparing the figures for empty messages (Figure 3.19b and Figure 3.19a) the impact of removing outliers from the data can be studied; in the figure with all data points it could be seen that long executions happen occasionally and they are not clustered. Apparently, they are caused by short-term random effects and no systematic slowdown of the network by background activity is observable. Since they can occur at any time and they seem not be caused by background activity, they should not be removed from the computation of the mean.

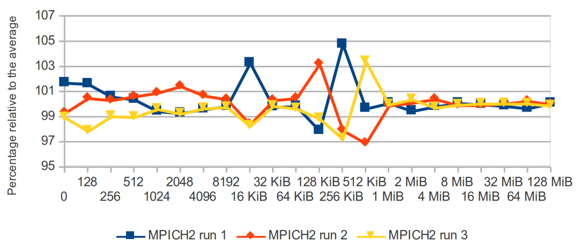
In contrast the plateau in Figure 3.20c is caused by some short-term activity that degrades performance. This can be identified as a real outlier that should be removed from the computation of the average value. An analysis of the reason behind this plateau would be interesting as well.

Variability between runs It has been observed that an average value can vary between different runs. This behavior differs from the fluctuations that are visible in a run and it is assessed briefly in the following. Remember, due to the construction of the benchmark computed average values are obtained for a test which runs at least 4 s (and typically 8 s) – during this time a loop communicates the same memory regions over and over between the two processes. The benchmark measures every data point for at least one second and repeats this process four times after which the average time is computed. Thus, the benchmark should be invariant to short-term effects such as hardware interrupts.

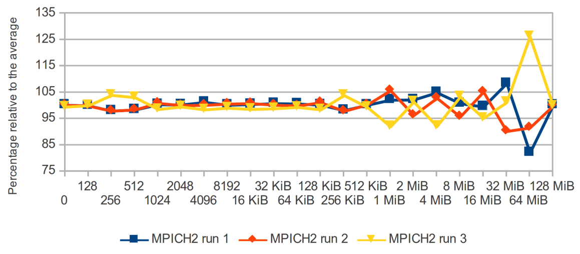
To assess the difference between multiple runs, the relative time of the three runs which have been used to compute the average, is compared with the time of every run. The first and the third run of the inter-node communication is conducted on the same node configurations (West9, West10) to see if the difference between those runs is similar to executing it on other node configurations. Runs for intra-node communication are executed on different nodes. The variability is just discussed qualitatively, for a detailed and statistical relevant analysis many more runs must be executed.

The relative performance for the conducted experiments is rendered in Figure 3.21. In the figures the y-axis offset is adjusted to highlight the variation between the runs. Note that for a proper analysis many more runs are necessary, this experiment just targets to increase awareness of variability between runs.

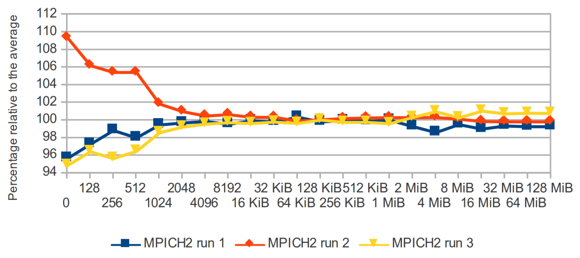
Observations and interpretation



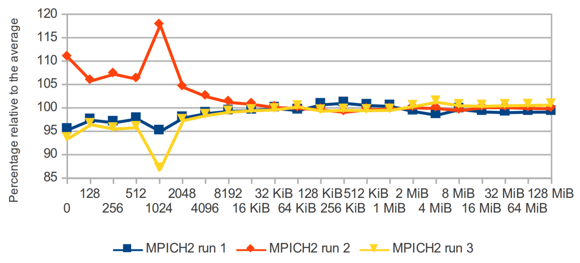
(a) PingPong kernel – inter-node communication.



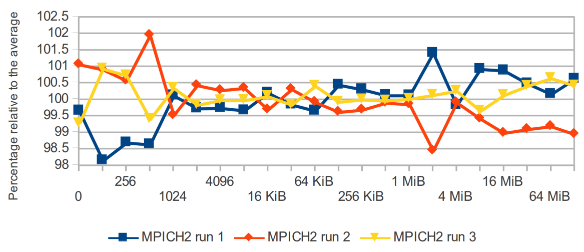
(b) Sendrecv kernel – inter-node communication.



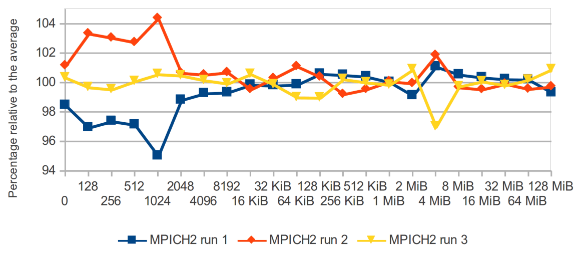
(c) PingPong kernel – inter-socket communication.



(d) Sendrecv kernel – inter-socket communication.



(e) PingPong kernel – intra-socket communication.



(f) Sendrecv kernel – intra-socket communication.

Figure 3.21.: Relative performance of the three runs to the average of all runs.

a) Overall the individual conducted PingPong averages differ by 8% when doing intra-node communication (case for 256 KiB), by 16% in inter-socket communication (case for 0 KiB), and by 3.5% for intra-socket communication. With the Sendrecv kernel the variance is even higher. Considering the long running experiment this result is surprising, especially since intra-node communication is independent from the variable Ethernet network.

b) In intra-node communication the variability between runs is higher than the spread between first and third quartile of a single run (compare the variance in Figure 3.16a). Consequently, averages of longer running applications are expected to vary slightly.

c) While there is a fluctuation on individual message sizes an overall trend that a single run achieves other performance values is not visible; for some message sizes a run is faster than another, sometimes it is worse. An exception to this observation is Run 2 in the inter-socket communication, here performance of small data points is much better than the two other runs. The data points for larger messages are close to the other runs; since performance is just better for very small messages there is no general advantage of Run 2 visible.

d) Typically, the fluctuation between repeated runs with the identical placement are in the order as running on another configuration (compare the three runs in Figure 3.21e, Run 1 and Run 3 are executed on the same host).

Conclusions It is obvious that noise disturbs individual communications; a variation of 10% in inter-node communication time happens frequently. For small messages the inter-process communication can need several the time of the previous operation. With a visual inspection of timelines atypical behavior could be spotted. This helps identifying true outliers that are probably caused by background activity.

It is interesting to see horizontal clustering of communication times. For larger messages it would be expected that these horizontal bands are hidden because random effects should add up and result in an "average" behavior. This indicates that non-linear hardware and software aspects are involved that could not be explained in the scope of this thesis. Consequently, similar to memory accesses a common random distribution is not capable to describe the observed behavior.

Even the average values that are determined by measuring performance for several seconds vary. Repeating a run results in a performance difference of up to 8%, 16% and 3.5% in inter-node, inter-socket and intra-socket communication, respectively. Sendrecv is even worse. This high variability leads to difficulties in determining the true average performance characteristics of the network.

3.6. I/O Subsystem

Understanding the behavior of a single data server is important to assess performance of parallel file systems, which use those capabilities. Our cluster nodes are equipped with a *Seagate Barracuda ST3250318AS* disk drive [Sea10], which is controlled by the Linux kernel.

The *I/O path* describes all the layers (and components) involved in I/O and how they interact. In brief, the I/O path of a write operation can be described as follows: When an application issues an I/O call, data is copied between the user-space buffer and the page cache that is offered in kernel-space. In kernel-space memory the data is cached and write operations can be deferred as long as free memory is available. At this point the write call can complete, because a programmer cannot modify kernel-space directly. A scheduler inside the kernel decides when modified pages are transferred to the block device. The mapping from offsets in logical files to addresses on the block storage is managed by a file system. Further, file systems provide the hierarchical namespace and offer additional management features.

Since the I/O path of the Linux kernel is even more complicated and offers several optimizations, important background information is provided in Section 3.6.1. Caching has a major impact on performance in most test cases. Therefore, the available memory has to be limited. This is achieved with a simple program called `mem-eater`, see Section A.4 for a description of the tool.

A reference value for the performance of the described write path is obtained by using the copy command `dd`; a throughput of approximately 100 MiB/s is observable[33]. Although the nodes are identical, the actual performance of the nodes varies between 92 MiB/s and 120 MiB/s[34]. On a single node a repeated measurement of the I/O throughput varies by a few MB/s.

All further reported benchmarks are made on the same node to understand the system's behavior. Since a file system must be deployed on the raw block device to store logical objects, further analysis is restricted to the *Ext4* file system that is deployed on our cluster nodes.

The observable performance of the prominent I/O benchmark IOzone[35] is given in Section 3.6.2. Assessing the average values with IOzone shows interesting effects and thus it serves as a seed point for in-depth analysis of individual operations. By default IOzone measures the average performance of repeatedly invoking a choosable I/O call. It is also interesting to see how the OS and disk activity interfere with individual I/O operations – later the simulator must assess performance of every I/O operation individually. In the follow-up sections timings for all I/O operations are recorded to understand the interplay of

[33]dd is executed as follows: `dd if=/dev/zero of=/tmp/test bs=1024k count=4000`. Main memory is limited to 1 GiB.

[34]In fact the disk drives on the nodes are partitioned identically, so are the system configurations. Further, the file system in `/tmp` is empty to ensure identical test conditions.

[35]`http://www.iozone.org/`

operating system and HDD better. Therefore, a little I/O benchmark called `posix-io-timing` has been written which times every sequential or random operation and exports timing for further analysis with the statistics tool R. Note that IOZone can time individual operations for sequential access, too[36]. But it is limited to sequential patterns and the new `posix-io-timing` benchmark uses timers with a nanosecond accuracy[37].

Timelines for the sequence of execution individual operations are generated and provide additional insight about the complex behavior of local data access. To understand the impact of internal optimizations the following experiments are conducted:

1. Accessing a file bypassing Linux and Disk cache (results are provided in Section 3.6.3). In this experiment overwriting of blocks is assessed, too.
2. Direct access to a file that bypasses the Linux page cache (see Section 3.6.4).
3. Regular file access that supports caching inside the OS (see Section 3.6.5).

Several of the discussed aspects are complicated and also depend on the underlying file system. Therefore, just a few possible explanations are offered but many facets could not be assessed further in this thesis.

3.6.1. Theoretic Considerations

Observable performance of our local I/O subsystem is determined by four factors:

- Hardware technology and characteristics of the block device
- Scheduling policy
- Caching
- File system

For all these aspects cluster specific information is provided in the following. Refer to page 32 for general information about I/O subsystems and for background information of factors which have an impact on (parallel) I/O performance.

Block device hardware The deployed *Seagate Barracuda* (*ST3250318AS*) has 7200 RPM. It is connected via a 3 Gb/s SATA interface and has a capacity of 250 GByte. The bus interface limits the performance of the disk drive to 300 MByte/s. According to the datasheet [Sea10] the disk is characterized by a sustained data rate of 125 MB/s, an average latency of 4.17 ms (derived by the RPM), a track-to-track seek time below 1 ms (for reads) and 1.2 ms (for writes), and an average seek time of around 8.5 ms and 9.5 ms for reads and writes, respectively. The drive uses a single platter with a single head and has 8 MB of cache. Note that the raw write performance of 125 MB/s, as provided by the vendor, is measured without a file system on top.

Scheduling policy All communication with the block device is controlled by a scheduler, on our cluster the *deadline* scheduler is enabled[38]. This scheduler may aggregate, defer and select operations which are then issued to the block device. While it performs these optimizations it usually serves requests in a fair manner.

Similarly, the hard disk drive deploys *Native Command Queuing* (NCQ), which optimizes the schedule to perform all currently pending I/Os. Due to fragmentation on disk even a single I/O call issued by an application may hit multiple non-neighboring blocks (LBAs) on the physical device. This enables a scheduler to actually optimize performance.

[36]By specifying the `-Q` parameter the offset and the timing in microsecond is recorded.
[37]Internally, `clock_gettime()` is used and the available accuracy is queried using `clock_getres()` and output.
[38]More information about the deadline scheduler is provided in the kernel documentation (`block/deadline-iosched.txt`).

Caching To understand the observations a brief introduction to I/O handling in the Linux kernel and the hard disk drive is provided.[39] The behavior which is introduced in the following can be parameterized by several files in the `/proc` file system, some relevant parameters are mentioned in the explanation; values of our system are provided.

Hard disk drives provide a read-ahead mechanism of blocks which is usually called read look-ahead (or just prefetch). Unfortunately, the specific caching algorithm for our hard disk drive is not documented, but the documentation for previous device generations is probably valid[40]. According to this scheme Seagate Barracuda has 8 MB of cache in which blocks are prefetched. Data written to the disk is copied to the internal cache. Once stored in cache it is acknowledged to the host, this write-behind strategy improves performance significantly. The cache of an HDD is typically not battery-backed, thus in case of a power outage the data stored in the cache is lost. To avoid an inconsistent state, SCSI defines *synchronize cache* commands which ensure that cached data of a range of blocks (or all blocks) is actually synchronized with the medium. The *ATA/ATAPI Command Set* defines *flush cache* which forces the cache to write out all data to the medium. In Linux, the protocol independent concept of write barriers is build on top of these concepts. A write barrier will complete if and only if all previous issued write requests have completed. Also, all later issued operations are deferred until the barrier completes.

In Linux, the file system uses pages of the virtual memory subsystem to cache data. In the write path, the kernel copies user-space data into this page cache; in the read path, data of the block device is read into the pages and kept for further usage. The file system maps the memory pages of a file to logical block addresses of the block device and performs additional metadata updates to record the state of the system. Background threads called `pdflush` try to asynchronously flush modified (dirty) pages to the I/O subsystem which are older than a given period of time (`/proc` parameter *dirty_expire_centisecs*: 500 $\Rightarrow$ 5 s threshold on our system), or if a certain percentage of main memory is filled with dirty pages (*dirty_background_ratio*: 10% of memory).

When a process writes data, it is written to cache and the process can proceed immediately until a certain percentage of pages is dirty [41]. Then future write calls will block and data is written back (*dirty_ratio*: 20% of memory). Also, in newer versions the kernel permits to set a number of bytes instead of these percentage values, but this is not the default behavior on our system.

Reads of data which is currently in the page cache happens without physical I/O by just copying data between kernel-space and the application and thus it executes at memory speed. Additionally, the kernel performs read-ahead of file's data, if it detects a sequential access pattern. Therefore, further data is fetched from the disk drive. Read-ahead is performed when data of at least 128 KiB is accessed sequentially – the algorithm grows the amount of data prefetched with the hit rate[42].

On our system the file system layer implements read-ahead of 256 sectors (each 512 bytes), prefetching 128 KiB of data [43]. Unmounting of a file system flushes dirty data to the underling devices and all cached data is dropped. This includes the page cache and the VFS caches. Therefore, all cached data is dropped and must be read from the block device if it is needed again.

The caching of the kernel can be avoided by specifying the *O_DIRECT* flag to the `open()` function. When this flag is set, then the data is copied directly between block device and the provided user-space buffer[44]. Thus, it omits the additional copy to the page cache in kernel space. As O_DIRECT bypasses the virtual memory management it also disables read-ahead of the Linux kernel.

[39]Refer to `http://www.kernel.org/doc/Documentation/sysctl/vm.txt` for information beyond this brief introduction.

[40]Refer to `http://www.seagate.com/support/disc/manuals/scsi/29471c.pdf` Page 21ff. for a description of the strategy.

[41]The concept of write-behind makes it difficult for an application to detect errors which occur while data is written back – an application might even be finished at this point. An application can use `fsync()` and check the return value to ensure that dirty data of a file has been persisted.

[42] In [GGJL10] the mechanism of Linux is explained in detail. Also, the paper introduces a methodology to adapt the buffer depending on observed access patterns. Refer also to [Wu10] for a description of the kernel implementation.

[43]The value is defined in `/sys/devices/.../sda/queue/read_ahead_kb`.

[44]An requirement for direct I/O is that the buffer is aligned to the page size.

Caching inside the I/O device is not prevented with *O_DIRECT*, for instance, write-behind can still be performed by a hard disk drive. A specification of the *O_SYNC* flag blocks the process writing to the file until data has actually been written by the hardware. Note that the *O_SYNC* flag enforces dirty metadata of the file system to be flushed to disk as well. Appending to the file causes updates of the file size and requires to allocate blocks which in turn alters metadata. Therefore, O_SYNC requires further modifications to the file system structure besides modifying the plain data blocks.

All the mentioned caches are related to file data, there are also several metadata caches that are provided on the layer of the virtual file system switch (inode and dentry cache), or within the file system itself.

File system Since a file system maps file system objects and the logical data of a file to the blocks on the block device it must manage additional information about this mapping. For every file metadata stores the list of blocks (LBAs) the byte range is made up of. Initially, upon creation of a file this list is empty and grows when data is appended to the file. Ext4 [MCB⁺07a, MCB⁺07b] stores ranges of blocks (extents) and tries to allocate neighboring blocks for the data of a file to improve performance on disk drives – typically, neighboring LBAs are also neighbors on the physical geometry of the disk[45]. Also, Ext4 defers the allocation of blocks to the moment data is written out from the page cache to the disk drive. For more information on the delayed block allocation scheme see [AKCSD08].

File system operations might update multiple LBAs on disk. Failure to replace one LBA on the disk, e.g., by a power outage, leads to an inconsistent state of the file system. This is critical if multiple metadata blocks are updated – in this case the file system can be corrupted leading to non-reachable files and folders. To ensure file system consistency in such operations transaction concepts are implemented to guarantee consistency even in case of power outage. For example, to ensure consistency Ext4 writes all metadata updates to a log that is stored in a reserved area of the underling device (the so-called journal) and then updates the LBAs. Thus, all activity triggered by a metadata update can be considered to be encapsulated by a transaction. If a power outage occurs while a block is overwritten, then the journal can be replayed which guarantees consistency of the file system.

The volumes of our cluster are formatted with Ext4[46] and mounted with the following options: noatime, barrier=1, data=ordered. The *noatime* flag means that accessing of a file system object does not update the `atime`, the timestamp that records the last access of the inode. Otherwise, reading of a file causes a modification of the inode which must be written. Write barriers are enabled, which means that a journal update enforces disk activity and waits for completion. With *data=ordered* first data blocks are updated on the disk, then metadata is committed to the journal. For further information refer to the kernel source documentation (*filesystems/ext4.txt*).

3.6.2. Average Performance

The IOzone benchmark supports in-depth analysis of file system performance. Many tests are provided: a variation of POSIX I/O operations is supported, multiple access patterns can be executed, and orthogonal aspects can be assessed[47]. An automatic mode allows measuring average performance for a variation in the access granularity (called record size) and the file size.

For this thesis three experiments are conducted with IOzone: one which remounts the file system between every test, one which uses all available memory (12 GB) and one in which main memory is limited to

[45]There is an exception in newer disk drive: Every drive provides spare blocks to cover for defect blocks, when a defect is detected the LBA of the physical block is remapped to one spare block – this is transparent to the operating system which can still access all LBAs.

[46]Features of the mounted file system: has_journal, ext_attr, resize_inode, dir_index, filetype, needs_recovery, extent, flex_bg, sparse_super, large_file, huge_file, uninit_bg, dir_nlink and extra_isize.

[47]For example, concurrent access by multiple threads, usage of the O_DIRECT flag or remounting of the file system.

one GiB [48] For every experiment the average performance for a variable file size, record size and a few access patterns is measured. The evaluated file size is between 10 MiB and 5 GiB and the record size between 16 KiB and 8 MiB. The following access patterns are executed in a sequence for every file size and record size: data is written sequentially, then read sequentially, then written randomly and, at last, read randomly. Actually, a random experiment accesses every record just once, a *Fisher-Yates card shuffle* creates a permutation of offsets.

Average performance of the experiments is rendered in graphs for every access pattern. Those graphs are shown in Figure 3.23, Figure 3.24 and Figure 3.22. An excerpt of the quantitative performance values are given as reference. The experiments without remounting are listed in Table 3.7 and the experiment with remount in Table 3.8. File sizes that achieve too high performance for all access patterns (that means they are fully cached) are not included in the tables. Values for the largest file size that achieves comparable performance to all sizes below is provided, for example, if performance is roughly 1,500 MiB/s up to files with a size of 1,280 MiB, then the file sizes below this limit are omitted.

Observations and interpretation

a) Performance for smaller file sizes is in the order of several GiB/s, but drops with increasing file sizes. See the read and write graphs in Figure 3.23 and Figure 3.22. With less main memory, performance drops earlier, e.g., at 320 MiB, while with the large amount of main memory it drops at 2,560 MiB.

The caching explains the high performance in the range of several GiB/s, which is at memory speed. Due to the settings of `dirty_ratio` the IOzone process is blocked when writing more than 2 GiB or 200 MiB of data, which reduces observable write performance for 12 GiB or 1 GiB of available memory, respectively.

b) Read performance is very high in the measurements up to a file size of 640 MiB and 2.5 GiB for 1 GiB and 12 GiB of main memory, respectively. This is due to the caching, after the write test has been executed this data is read but available in cache.

c) With file sizes bigger than the cache, for a few combinations of record sizes the observable performance increases while the file size increases. For example compare the read and write results for 16 KiB records for the 1 GiB, 2 GiB and 5 GiB file (look at Table 3.7). Performance of the 2 GiB file is lower than for a 1 GiB file (78 MiB/s vs. 86 MiB/s), and the 5 GiB file behaves like the 640 MiB file (about 94 MiB/s). This is a performance increase of about 16%.

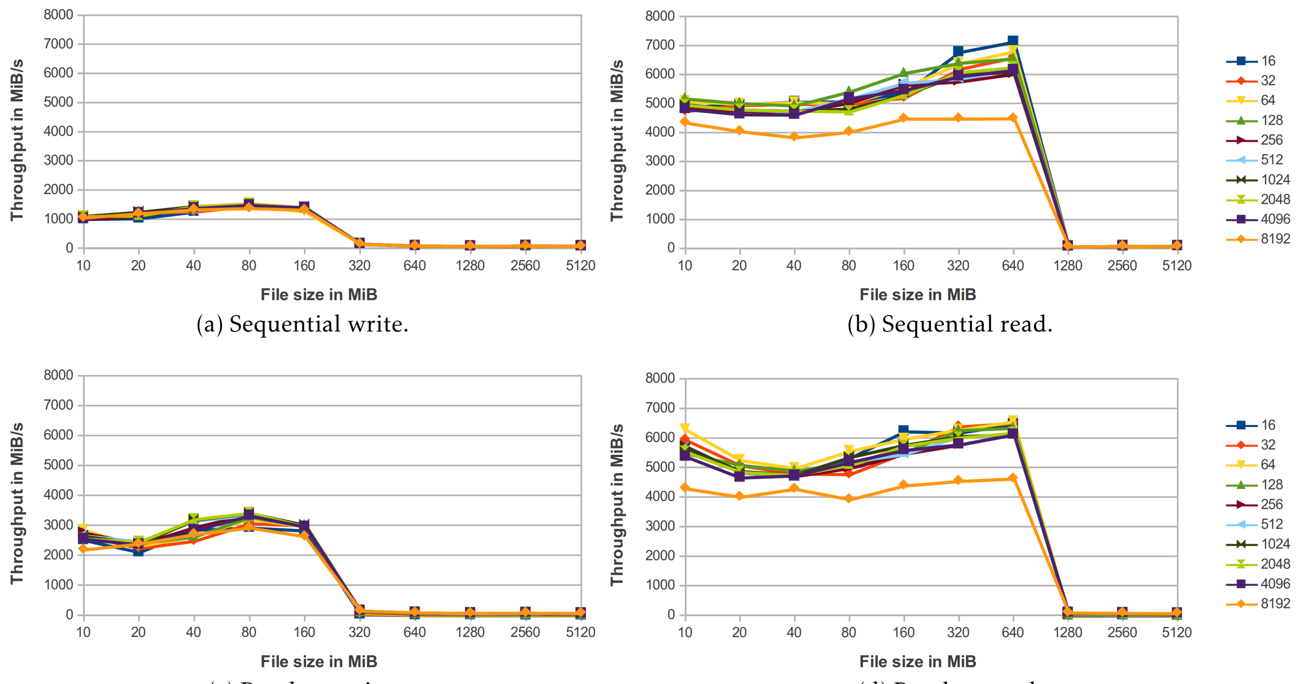
For larger record sizes than 128 KiB, the accesses to the 2 GiB file are faster than for 1 GiB or 5 GiB. Since this result is visible in the sequential read, write and the re-read and re-write of data it is probably not caused by a background activity. The cause is unknown.

d) The sequential (and random) read performance degrades for the 5 GiB file and larger record sizes even when 12 GiB of main memory is available. This indicates that some data is removed from the cache before it is read (see Figure 3.23d and Figure 3.23b). An interesting result is that 16 KiB accesses are cached efficiently for random read achieving 6,000 MiB/s while larger record sizes are hitting disk. Also, sequential and random read achieve a better performance for the 16 KiB record size than for 64 KiB records (look at Table 3.7). For example, 116 MiB/s is measured for 16 KiB sequential access but just 84 MiB/s for 64 KiB records.

e) In Figure 3.23 and Figure 3.22 the performance of all cached record sizes behave similar. The performance degradation due to shortage in cache is just shifted to smaller file sizes.

f) Sequential write and random write of the experiment with remounting (see Figure 3.24) look like the results obtained without remounting. Since the benchmark does not include the time for unmounting the file system both perform the identical operations. Thus, this is expected.

[48] The run parameters for IOzone are: `iozone -a -n 10m -g 5g -q 10m -y 16k -f /tmp/remount/test -i 0 -i 1 -i 2 -i 9 -i 10 -R [-U /tmp/remount]`. The `-U` flag commands IOzone to remount the file system between testing two access patterns.



(a) Sequential write. (b) Sequential read. (c) Random write. (d) Random read.

Figure 3.22.: Performance of several access patterns measured with IOzone – 1 GiB of memory is available.

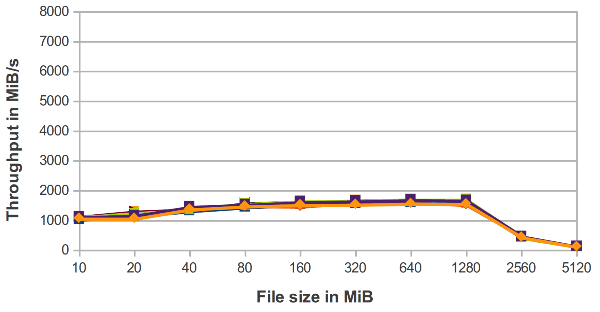
g) Sequential read performance of the remount experiment hits disk because the cache is purged before the data is read. Performance is relatively invariant to the block size and file size. In this case, read-ahead of the OS and disk works and improve observed throughput because the read-ahead window can grow until best performance is achieved.

h) When the random read performance is determined for the remount experiment, the disk characteristics are revealed because the data is not cached in disk drive or the operating system. Now, performance depends on the block sizes and increases with larger record sizes up to 90 MiB/s (see Figure 3.24). Latency of the disk dominates I/O time for small record sizes and thus the performance drops significantly for smaller records.

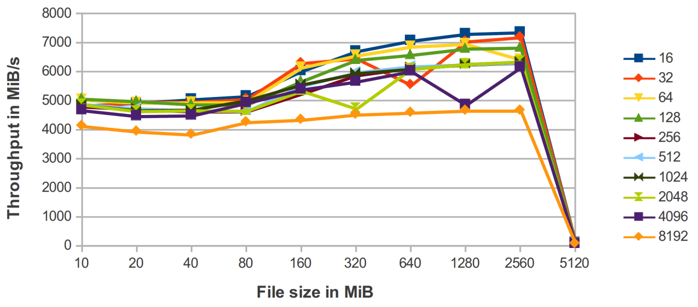
i) With larger file sizes random performance is significantly slower than sequential access, it depends on the record size and decreases further with increasing file sizes (see Table 3.7). Under the assumption that data of a single file is stored in a consecutive area, the average distance between two random offsets in the file increase with the file size. Therefore, the time to move the disk heads to the right track increases as well, which increases latency and thus reduces observable throughput.

j) For the largest file sizes (i.e., 5 GiB) measured throughput of random read for the 12 GiB remount run and for the run with 1 GiB are in the same order, but vary (see Table 3.7). Since in both cases memory cannot cache data any more, those cases are expected to behave alike, which is only true to some extent. In most cases remounting the file system increases performance slightly. For record sizes 16 and 32 KiB the bigger memory cache improves throughput to 5 MiB/s which is a factor of 2, for the 8 MiB records performance without the remount are a bit better.

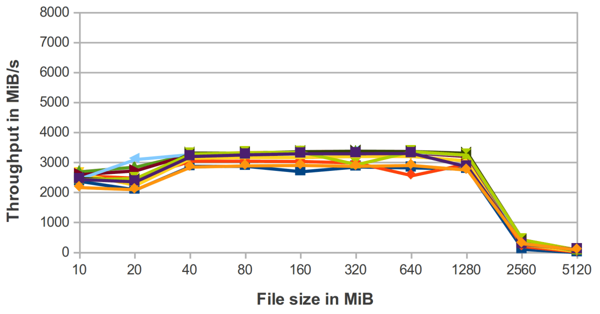
Theoretically, experiments which do not enforce remount could defer write operations. Then during the read kernel there are still some dirty pages available, which must be written-back. Thus, the additional writes performed in the read test spoil the results and reduce the observable performance. Consequently, the measured values of the experiment without remount do not represent the intention of the experiment. While this could be fixed by enforcing flush during the write operations, this would not allow to measure the throughput of the write-behind cache. To understand the internal processing better individual operations are timed in the next section.



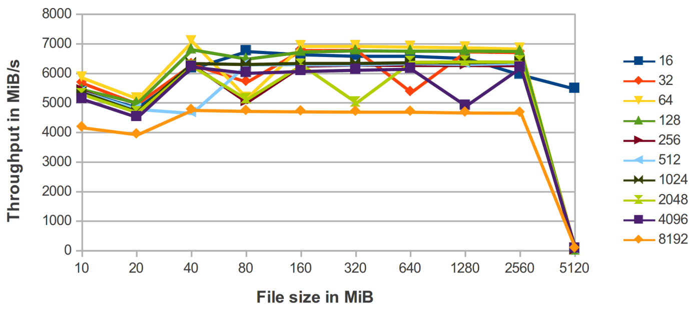
(a) Sequential write.



(b) Sequential read.

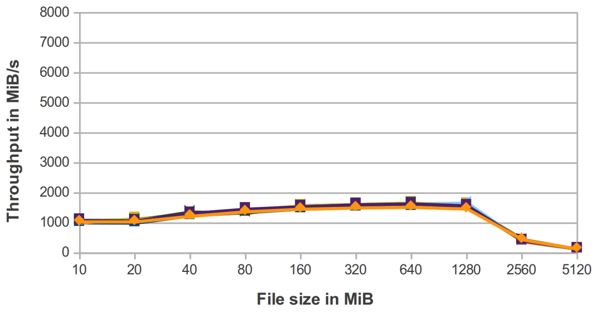


(c) Random write.

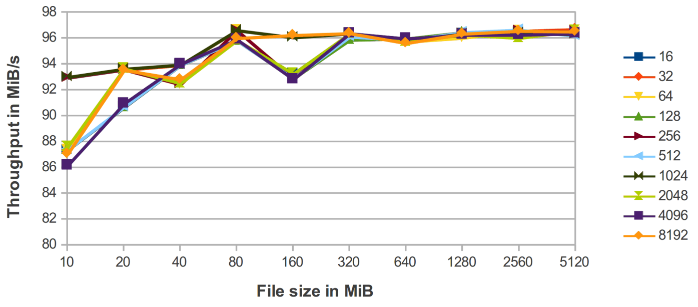


(d) Random read.

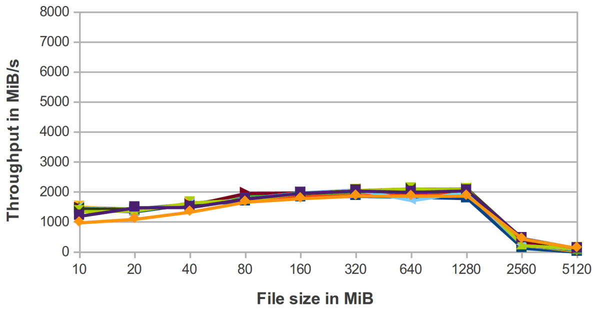
Figure 3.23.: Performance of several access patterns measured with IOzone – 12 GByte of memory is available.



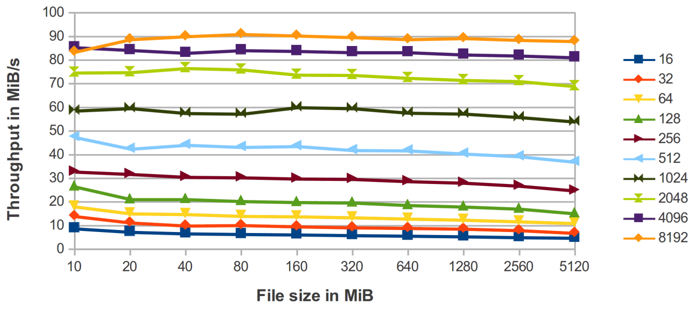
(a) Sequential write.



(b) Sequential read – note the different scaling.



(c) Random write.



(d) Random read – note the different scaling.

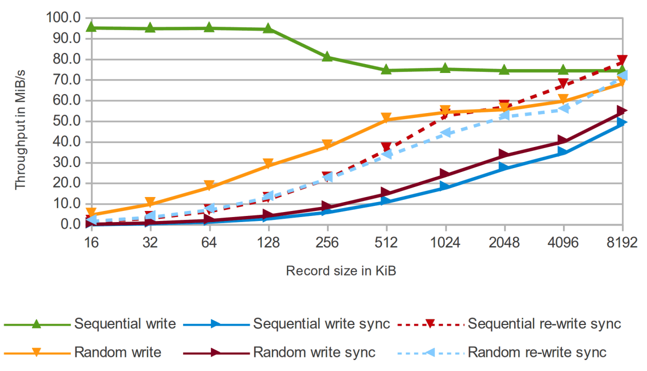
Figure 3.24.: Performance of several access patterns measured with IOzone – 12 GByte main memory is available, a remount is performed between two tests.

Configuration	Access pattern	File size	Record size in KiB									
		in MiB	16	32	64	128	256	512	1024	2048	4096	8192
12 GiB memory	Sequential write	1280	1600	1533	1623	1689	1690	1639	1708	1714	1678	1562
		2560	434.2	475.8	476.2	480.0	473.8	469.1	458.1	433.3	475.6	448.5
		5120	141.7	135.4	134.4	137.3	134.5	136.8	134.3	126.9	140.4	130.3
	Sequential read	2560	7361	7193	6424	6833	6315	6322	6351	6355	6159	4667
		5120	116.6	113.3	84.5	83.7	86.0	83.9	80.8	80.6	79.1	80.4
	Random write	1280	2810	2984	3084	3282	3229	3268	3338	3277	2885	2775
		2560	132.4	226.7	293.7	363.9	392.3	409.8	447.0	449.0	300.3	320.4
		5120	24.3	39.4	58.5	79.9	97.6	105.4	105.8	109.4	122.4	109.2
	Random read	2560	5973	6730	6850	6772	6289	6360	6414	6413	6212	4681
		5120	5490	35.3	10.9	17.9	28.1	45.5	60.1	88.2	84.2	89.1
1 GiB memory	Sequential write	160	1365	1352	1267	1412	1415	1427	1401	1408	1404	1330
		320	172.6	167.2	164.4	161.8	162.0	164.7	163.5	162.4	158.6	163.8
		640	94.0	95.5	94.8	96.7	96.3	96.1	105.3	106.5	95.3	95.0
		1280	86.6	85.2	86.1	86.0	83.8	85.6	82.8	85.5	82.0	85.8
		2560	78.5	77.3	83.2	99.8	98.6	99.5	100.0	98.1	99.0	97.6
		5120	93.7	93.8	94.2	91.5	90.3	90.6	90.6	89.2	87.1	86.2
	Sequential read	640	7143	6604	6808	6563	6032	6197	6106	6259	6165	4499
		1280	82.9	82.6	81.7	80.9	78.8	79.9	78.5	81.1	79.4	79.2
		2560	77.3	77.6	81.7	96.7	98.2	98.7	96.8	96.5	96.9	95.7
		5120	94.6	94.4	94.5	91.5	91.1	90.7	91.0	89.3	87.6	86.9
	Random write	160	2838	3006	2973	2988	2981	2983	3027	2973	2968	2644
		320	38.1	53.3	86.1	116.3	130.9	141.6	149.0	160.6	162.2	161.0
		640	14.2	21.5	36.0	53.3	71.7	82.5	89.8	96.2	97.5	101.2
		1280	8.5	15.7	24.5	39.5	52.3	63.4	70.8	76.7	79.5	79.9
		2560	6.8	12.8	21.1	36.3	53.1	65.9	78.7	85.8	90.3	91.2
		5120	6.0	11.5	19.9	31.8	46.5	58.3	66.7	74.6	72.7	78.8
	Random read	640	6478	6516	6570	6353	6139	6108	6136	6185	6124	4637
		1280	4.9	8.6	14.2	23.4	34.5	51.6	69.5	84.0	95.0	104.3
		2560	2.7	5.0	6.5	16.3	26.4	40.6	57.3	74.4	85.4	91.2
		5120	2.3	4.3	8.1	14.5	23.7	36.6	52.3	66.0	74.3	80.4

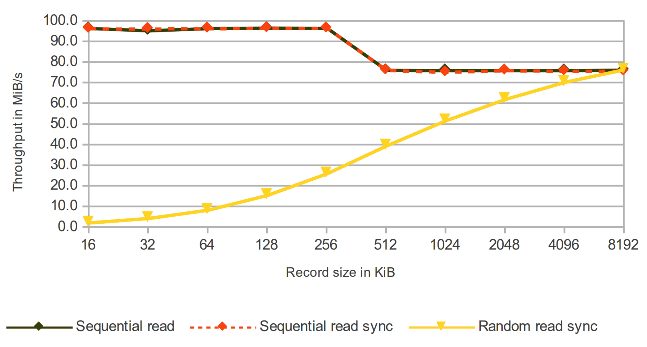
Table 3.7.: I/O throughput in MiB/s measured with IOzone for a variable record sizes with 12 GByte (or 1 GiB) of available main memory.

Access pattern	File size	Record size in KiB									
	in MiB	16	32	64	128	256	512	1024	2048	4096	8192
Sequential write	1280	1583	1636	1648	1612	1603	1692	1598	1596	1600	1494
	2560	422.2	408.2	435.8	484.2	452.1	430.7	455.5	439.6	439.1	472.4
	5120	158.9	159.3	156.0	158.7	157.3	156.8	158.3	157.8	159.4	158.3
Sequential read	10	87.4	87.3	87.4	87.4	93.0	87.4	93.0	87.6	86.2	87.1
	20	90.7	93.6	93.6	90.7	93.6	90.7	93.6	93.7	91.0	93.5
	40	93.9	93.9	92.4	93.9	92.5	93.9	94.0	92.5	94.0	92.8
	80	95.8	96.6	96.6	95.8	96.6	95.8	96.6	95.9	95.9	96.0
	160	93.3	92.9	92.9	92.9	92.9	92.9	96.1	93.3	92.8	96.2
	320	96.4	96.4	96.4	95.9	96.4	96.1	96.4	96.4	96.4	96.4
	640	96.0	96.0	95.7	96.0	95.9	95.9	95.9	95.7	96.0	95.6
	1280	96.3	96.3	96.0	96.5	96.3	96.5	96.3	96.3	96.3	96.3
	2560	96.4	96.6	96.6	96.3	96.6	96.7	96.4	96.0	96.3	96.6
	5120	96.4	96.7	96.4	96.4	96.6	96.2	96.3	96.6	96.4	96.5
Random write	1280	1808	1973	2077	2070	2109	2028	2124	2107	2067	1892
	2560	150.8	259.4	247.3	237.3	348.3	236.6	453.7	241.1	475.0	458.0
	5120	23.4	39.7	59.7	79.1	108.7	120.8	125.0	143.0	139.7	148.3
Random read	10	9.0	14.2	18.2	26.7	33.0	47.6	58.8	74.9	85.6	83.7
	20	7.5	11.4	15.3	21.3	32.0	42.7	59.8	75.0	84.4	89.0
	40	6.8	10.2	15.0	21.4	30.8	44.3	57.8	76.8	83.2	90.2
	80	6.6	10.4	14.3	20.5	30.6	43.4	57.5	76.2	84.3	91.2
	160	6.4	9.8	14.1	20.1	30.1	43.8	60.2	74.0	84.0	90.5
	320	6.1	9.4	13.6	19.9	29.9	42.1	59.8	73.9	83.5	89.8
	640	5.9	9.2	13.1	18.8	28.9	41.9	57.9	72.6	83.5	89.0
	1280	5.6	8.8	12.6	18.2	28.3	40.6	57.5	71.7	82.5	89.4
	2560	5.2	8.2	11.9	17.3	27.0	39.5	56.0	71.2	82.0	88.7
	5120	5.0	7.0	11.2	15.3	25.0	37.1	54.1	69.1	81.3	88.2

Table 3.8.: I/O throughput in MiB/s measured with IOzone for a variable record sizes in the experiment with 12 GByte available main memory. The file system is re-mounted by IOZone between measuring two access patterns.



(a) Write performance.



(b) Read performance.

Figure 3.25.: Performance of direct IO for a variable access granularity using *O_DIRECT* (and *O_SYNC*) measured with `posix-io-timing` for several access patterns.

3.6.3. Timing Uncached Data Access

In the following three sections the benchmark `posix-io-timing` is used to time I/O operations individually. The benchmark contains kernels to perform sequential or random access of either POSIX read or write calls. The benchmark iterates over record sizes between 16 KiB and 8 MiB. During each test a file with a size of 1 GiB is created and accessed with a fixed record size. In contrast to IOzone the kernel for random I/O of `posix-io-timing` accesses blocks with truly random offset, thus it may access the same record multiple times[49]. The random generator uses always the same seed to allow us to compare the observed performance.

In this experiment files are opened with the O_DIRECT and O_SYNC flags. By opening the file with O_DIRECT the kernel page cache is avoided and data is transferred directly between provided user-space buffers and the disk. With the O_SYNC flag the kernel will wait for the disk's notification of write operations, before it proceeds.

A general overview of the average performance achieved with direct IO is provided in Table 3.9, those results are also visualized in Figure 3.25. The read case and direct writes (without O_SYNC) are discussed in Section 3.6.4. Timelines are generated for interesting cases and discussed in the evaluation. In the timelines throughput of accessing a single record is plotted to simplify the comparison between record sizes and the theoretical peak performance. All timings are included in the timelines, that means outliers are not removed, because they provide insight about jitter in the system.

Observations and interpretation

a) The random read performance which adds the O_SYNC flag is similar because the flag does not alter the behavior and consequently the results are comparable (look at Table 3.9). Due to this fact and because write is complicated the discussion of the reads is done in the next section, where just O_DIRECT is used.

b) Synchronous sequential write performance is very low (about 0.4 MiB/s) and reveals several horizontal bands (see Figure 3.26a). At beginning of the random write performance just looks alike the sequential performance (see Figure 3.26b). The reason for the low performance is the necessary block allocation when appending to a file hosted on Ext4. Block allocation updates the file system's metadata to mark the blocks as used, and it also updates the file's metadata. By supplying the O_SYNC flag those modifications also require the metadata to be flushed to disk. This in turn requires the journal of the file system to be updated and flushed before the actual modifications are made. Since

[49]The file is partitioned into records of the given size, records are accessed completely.

Access pattern	Record size in KiB									
	16	32	64	128	256	512	1024	2048	4096	8192
Sequential write	95.6	95.3	95.4	95.0	81.2	75.0	75.6	74.9	74.9	74.9
Sequential write sync	0.4	0.9	1.7	3.3	6.4	11.5	18.4	27.6	35.1	49.3
Sequential re-write sync	1.8	3.6	6.9	12.9	22.8	36.8	53.2	57.5	67.9	79.2
Random write	5.3	10.4	18.6	29.1	38.3	51.3	54.9	56.1	60.3	68.9
Random write sync	0.6	1.3	2.5	4.8	8.8	15.4	24.4	33.9	40.7	55.0
Random re-write sync	2.1	4.1	7.6	13.7	22.7	33.8	44.3	52.7	56.0	72.0
Sequential read	96.7	95.5	96.6	96.8	96.7	76.4	76.3	76.2	76.2	76.4
Sequential read sync	96.5	96.4	96.5	96.7	96.6	76.3	75.4	76.2	75.9	75.8
Random read	2.3	4.6	8.5	15.8	25.9	39.7	52.0	61.0	67.7	73.7
Random read sync	2.4	4.6	8.6	15.8	26.3	39.8	52.0	62.3	70.6	76.7

Table 3.9.: Average throughput in MiB/s of direct I/O by using *O_DIRECT* (and *O_SYNC*) to access 1 GiB of data – measured with `posix-io-timing` for a variable access granularity.

in Ext4 certain blocks are reserved for this metadata, which are not in sequence, disk seeks are necessary in both cases. Consequently, to append a single record multiple blocks are updated that are spread over the disk. Therefore, both operations look like a random access pattern to the disk. Thus, by specifying O_SYNC performance degrades seriously.

c) When data is written to a file which already exists the performance is higher; compare the figures for sequential and random re-write of data blocks (Figure 3.26e and Figure 3.26f) with the first writes. Also, the horizontal clusters change.

Thus, presumably preallocation performed by Ext4 explains some of the horizontal bands. With the implemented preallocation algorithm a single allocation reserves a bigger consecutive extend than required and assigns it to the file. Therefore, a few further updates of the file do not require to allocate new blocks from the file system, instead just the file's metadata is modified. Since the files exist in the re-write test no blocks must be allocated at all. Therefore, the workload of updating the file system structures is reduced.

d) Average random write performance increases over time – data points spread up to 7 MiB/s on the right (see Figure 3.26c). Performance stays on one level for a random re-write of data (Figure 3.26f).

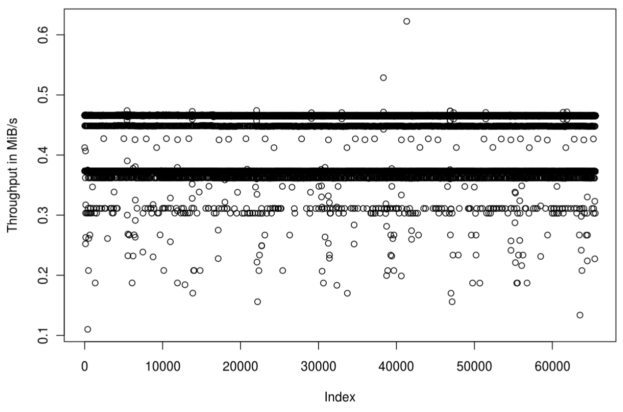
Again, the reason is the block allocation: As the benchmark accesses data at offsets of the record size either a record has been written or not. Overwriting of a record that has been written does not trigger block allocation and the metadata again, which reduces the workload.

Since the number of already written blocks increase over the run the average performance increases. Approximately 62% of the logical file blocks are written at the end[50].

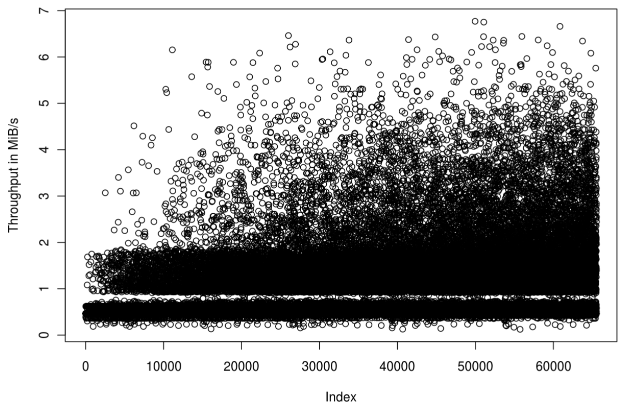
e) Writing with a larger record improves performance because all blocks can be allocated together requiring one update of the file's metadata and of the file system structures (compare the performance of synchronous I/O in Table 3.9).

A timeline for 2 MiB records is given in Figure 3.26d. Probably every data point in the figure which is in the horizontal line at either 30 MiB/s or 28 MiB/s is caused by a new block allocation and thus existing data is not overwritten in those cases.

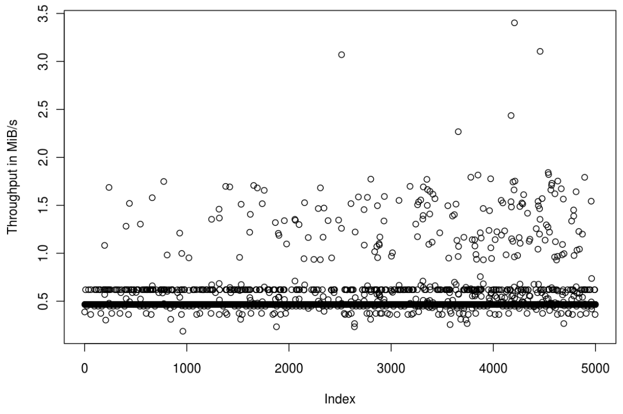
[50]Every written block reduces the probability that another write accesses a previously unwritten block. The value has been approximated with the command line tool bc: `echo "y = 0; for(i=0; i < 10000; i++) y=y+0.0001*(1-y); ; print y" | bc -l`.



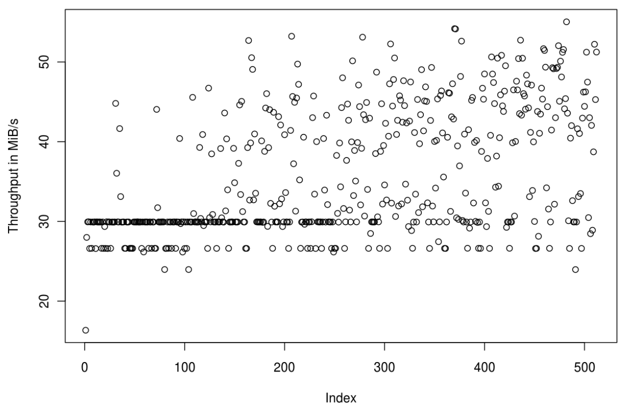
(a) Sequential write – 16 KiB records.



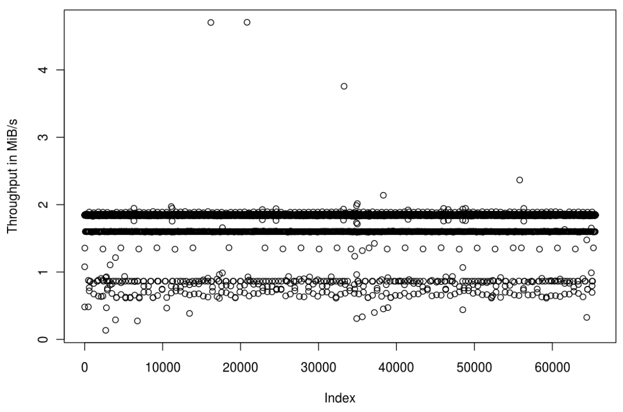
(b) Random write – 16 KiB records.



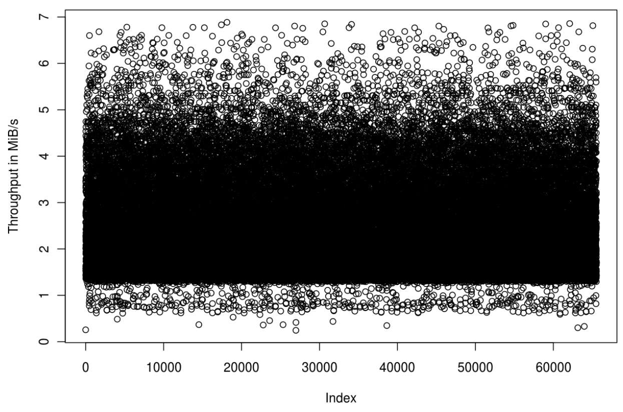
(c) Random write – 16 KiB records, first 5000 timings.



(d) Random write – 2048 KiB records.



(e) Sequential re-write – 16 KiB records.



(f) Random re-write – 16 KiB records.

Figure 3.26.: Direct I/O performance using *O_DIRECT* and *O_SYNC* measured with `posix-io-timing` for several access patterns.

3.6.4. Timing Accesses Bypassing the Linux Cache

In this section the benchmark `posix-io-timing` is used to time I/O operations that are performed by the block device, therefore, the O_DIRECT flag is used. In this case, I/O is done without involving the Linux page cache and thus all operations are handled by the disk drive. Read and writes are assessed. While the read path is identical to the previous results, in the write path updates of the file system's metadata (and journal) can be deferred (because O_SYNC is not specified), so that block allocation is done in memory.

All results are obtained with the `posix-io-timing` benchmark to time individual I/O operations. Timelines for all access patterns and 16 KiB records are given in Figure 3.27. Average throughput for all access granularities is listed in Table 3.9.

Observations and interpretation

a) Sequential read performance of 16 KiB records achieve already a high performance of 96 MiB/s (see Table 3.9). Since read-ahead of Linux does not work when a file is opened with O_DIRECT, the high performance must be caused by the disk internal read-ahead mechanism: Whenever data is fetched the disk reads the next data blocks while the data is returned to the callee. Therefore, when the next blocks are requested they are already available in the disk cache and copied from the internal cache.

b) The performance drops to 76 MiB/s when more data than 256 KiB are requested. At this point the read-ahead and the read call seem not to overlap any more. Probably the disk cache does not read-ahead that much of data. Actually, the Linux kernel reported a read-ahead window of 128 KiB for the hard disk drive, but the result suggest twice that much. This could be caused by the hard disk drive's look-ahead feature.

The behavior is probably influenced by the maximum amount of data that can be handled by a file system request, which is limited to 512 KiB on our system[51]. Effectively, this parameter limits the amount of data that can be exchanged between block device and Linux in a single request.

c) Compared to the discussed variation in read performance the observed throughput of cached sequential reads is stable (compare the throughput of IOZone in Table 3.8, typically 96 MiB/s). Direct I/O is similar to the IOZone results for small records, but it is slower for larger record sizes.

d) Sequential write performance is also very high when less than 512 KiB of data is written per access. Then performance drops similar to sequential read performance. Probably the `max_sectors_kb` limit mentioned previously, that hinders larger data transfers to occur in a single transfer between device and the kernel, is a performance bottleneck.

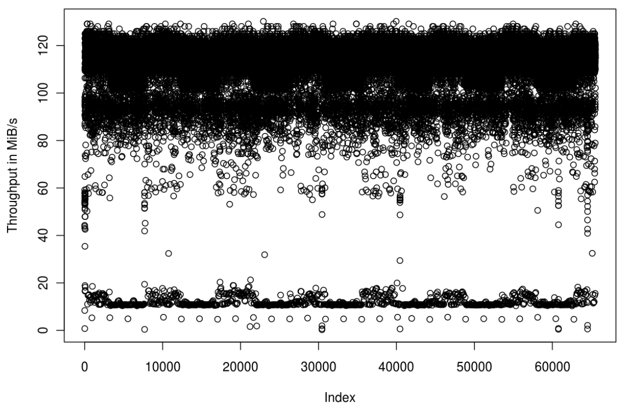
e) Random read performance is slightly better than synchronous random re-write performance (see Table 3.9). This is probably since Ext4 requires to update the file system metadata for writes, but not for reads. Also, the physical read and write process of the HDD differs slightly.

f) A few operations for sequential read and writes need much more time (see Figure 3.27b and Figure 3.27a). Thus, some slower random-like operations are caused by the sequential access patterns, those could be metadata reads or updates. In the write-path the disk cache could be full, degrading the next operations until it is ready – this would explain the periodic slow operations. To assess this further the first 5000 timings are printed in additional diagrams.

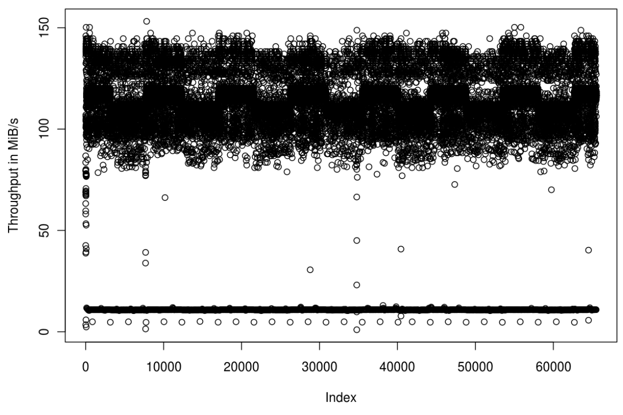
g) In the sequential read diagram an interesting step pattern is observable; many data points aggregate at 115 MiB/s or 105 MiB/s (see Figure 3.27d). Mainly, performance alternates between those two steps. Also, several very fast operations can be observed.

h) There are bursts of faster I/O visible when performing random writes – roughly every 1750 operations (see Figure 3.27d). The 1750 operations correspond to a data volume of roughly 28 MByte. The

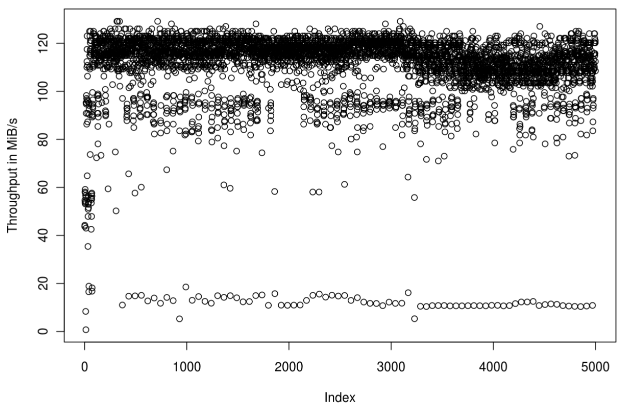
[51]This information is provided in the variable `/sys/block/<blk-device>/queue/max_sectors_kb` variable, according to the kernel documentation (`block/queue-sysfs.txt`).



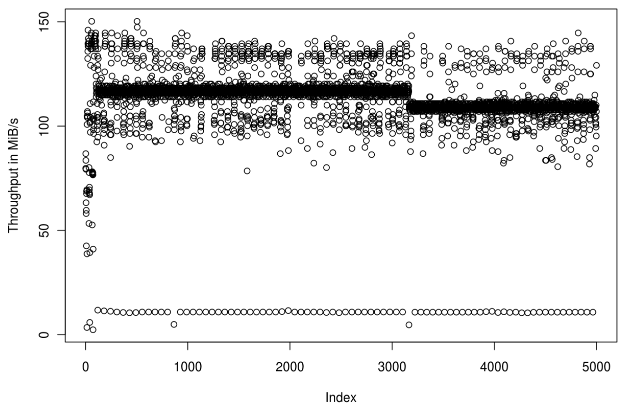
(a) Sequential write.



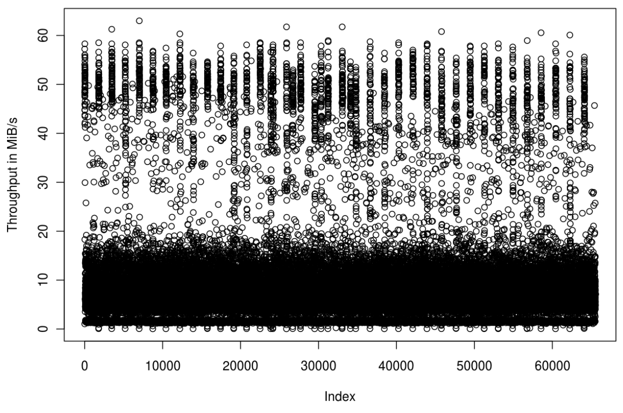
(b) Sequential read.



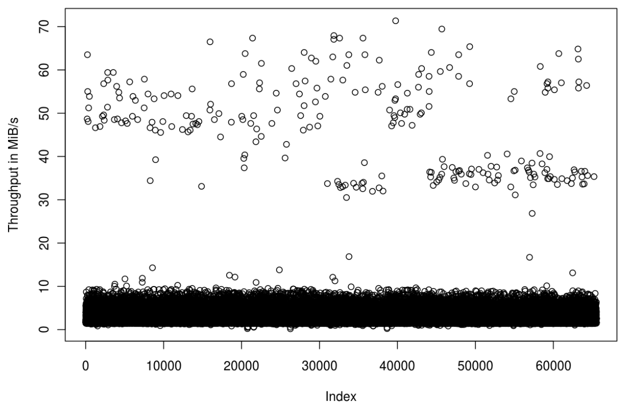
(c) Sequential write – first 5000 timings.



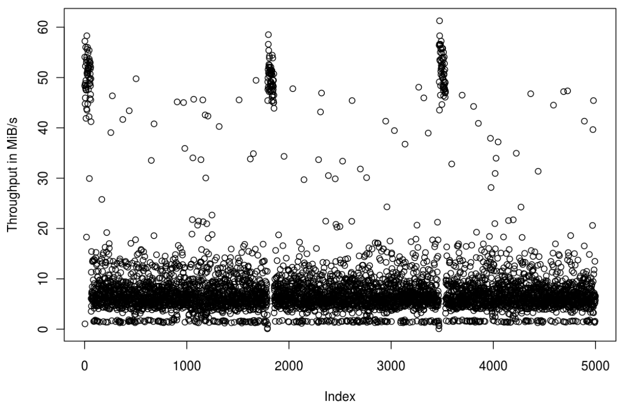
(d) Sequential read – first 5000 timings.



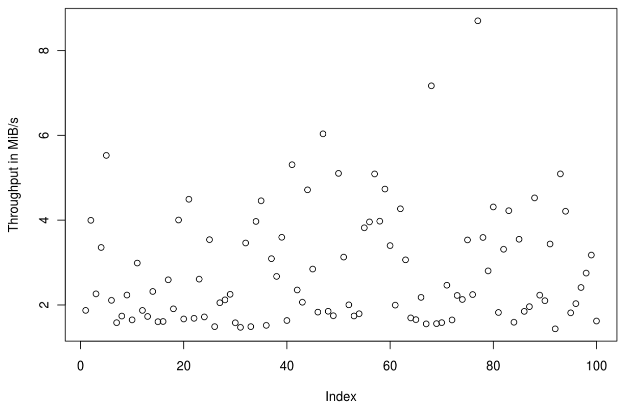
(e) Random write.



(f) Random read.



(g) Random write – first 5000 timings.



(h) Random read – first 100 timings.

Figure 3.27.: Direct I/O performance using *O_DIRECT* measured with `posix-io-timing` for several access patterns and 16 KiB records.

observed number of operations which are handled faster are 63 to 64 operations in every burst, which is about 1 MiB of data.

Without the sync flag the kernel can submit multiple operations to the disk without waiting for completion of an individual operation. With the NCQ capability offered by the drive up to 31 concurrent operations can be scheduled. Thus, NCQ would explain that 31 operations are transferred rapidly, then performance should drop. Thanks to the improved scheduling in the disk the average random write performance is much better than the random read performance (e.g., 5.3 MiB/s vs. 2.3 MiB/s for 16 KiB).

i) Since NCQ cannot be applied to random reads, obtained performance reveals the disks behavior (Figure 3.27f). Compared to the random write with the sync flag (see Figure 3.26f), some operations complete much faster than most operations which achieve less than 10 MiB/s. This is probably due to the true random access which might result in re-reading of cached information or sequential access.

3.6.5. Timing Cached Operations

In this experiment the timings for 16 KiB and 256 KiB accesses are measured with `posix-io-timing`. Several tests with a variable amount of free memory are conducted: not restricted (and roughly 12 GB is available), or memory is restricted to 200 MiB or 1 GiB. Data is stored on the Ext4 file system (like in the tests before). Further, a test of the in-memory file system `tmpfs` is performed that assists in comparing the performance of the Ext4 file system with `tmpfs`. In all cases, the test file has a file size of 1 GiB. Between two kernels data is kept in the caches and thus read benefits from the caching done in the write pattern before.

An overview of the quantitative values is given in Table 3.10a and Table 3.10b for West1 and West2, respectively. For all measured memory sizes and access patterns the average throughput of the whole run is provided, also the minimum, the maximum and the three quartiles. Results are listed for the two nodes to foster a qualitative intercomparison of repeated runs on distinct nodes.

Although, on `tmpfs` the benchmark just has a total runtime of about 0.2 s the correctness of the averages can be assessed qualitatively by inspecting the timelines. Timelines of the calls are given in Figure 3.28, Figure 3.29, Figure 3.30 and Figure 3.31: The first diagrams act as reference and show the timings when accessing data randomly from *tmpfs*; it also compares timings obtained on two nodes. In the second set of diagrams an overview of the access patterns is provided for one node and 16 KiB records. The last set of diagrams focuses on operations actually hitting disk, therefore, it cuts the y-axis of the previous diagrams at 20 MiB/s to reveal very slow operations.

Observations and interpretation

a) Quantitative results for accessing data on Ext4 depend largely on the amount of free memory (see Table 3.10a). With 200 MiB of memory the average small record performance is around 95 MiB/s for sequential access and in the order of 5 MiB/s for random access. This performance is similar to the one obtained with O_DIRECT. An increase in memory to 1 GiB improves the average performance of the random access to 18 MiB/s. When the cache is much bigger than the file size or if data is stored on `tmpfs`, then a performance of several GiB/s is observable, since operations causing disk activity are much slower than memory operations.

b) The spread between minimum, the quartiles and the maximum is very high, e.g., for sequential write and 200 MiB of cache a minimum of 0.06 MiB/s has been measured and a maximum of 1929 MiB/s. This is still true for fully-cached small accesses (compare the first and the third quartile in Table 3.10a, or look at Figure 3.28). Further, the minimum throughput can be quite low achieving only 20 MiB/s for individual records (0.8 ms per operation). Slow access seems to occur periodically (look at some of the figures in Figure 3.28). Because the I/O device is not involved on `tmpfs` it can be concluded that the fluctuations are already caused by OS jitter and/or the memory access. For

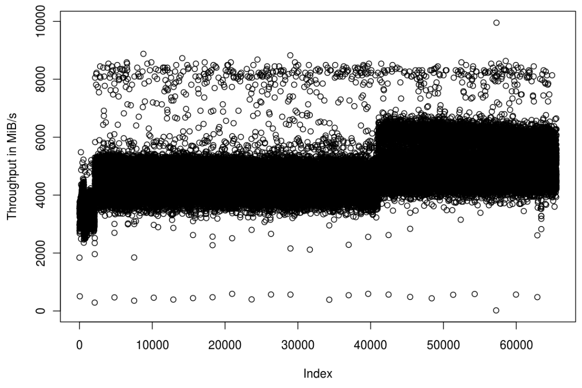
Memory	Pattern	Record size	*Average*	Min	1st quartile	Mean	3rd quartile	Max
200 MiB	Seq. write	16 KiB	93.0	0.06	863.3	1034	1177	1929
	Seq. read		95.8	1.58	3503.0	3416	4101	6483
	Rand. write	16 KiB	6.6	0.01	1134.0	1377	1606	5425
	Rand. read		2.7	0.20	1.8	394	4.4	4569
	Rand. write	256 KiB	47.6	0.67	966.4	1294	1530	4561
	Rand. read		26.5	3.57	20.9	259	34	7009
1000 MiB	Seq. write	16 KiB	107.6	0.019	1017	1261	1541	1717
	Seq. read		96.8	1.5	3685	3593	4257	7007
	Rand. write	16 KiB	11.1	0.002	1257	1749	2100	5333
	Rand. read		27.9	0.7	3100	3395	4080	8585
	Rand. write	256 KiB	67.7	0.2	1308	1882	2461	5532
	Rand. read		95.1	13.4	45.2	3230	5003	9416
unlimited	Seq. write	16 KiB	1527	67.7	1487	1546	1653	1736
	Seq. read		5530	162.7	5809	5759	6300	6621
	Rand. write	16 KiB	2101	236.7	1817	2256	2867	5351
	Rand. read		5743	143.4	4852	5993	6823	10210
	Rand. write	256 KiB	2334	1030.0	1987	2531	3406	5285
	Rand. read		6120	2346.0	6274	6198	6430	9513
on `tmpfs`	Seq. write	16 KiB	2380	19.55	2301	2421	2604	2775
	Seq. read		5463	285.8	5541	5557	5830	6104
	Rand. write	16 KiB	2122	19.49	1903	2309	2801	8778
	Rand. read		4561	19.56	4269	4676	4976	9952
	Rand. write	256 KiB	2995	1366	2542	3282	4437	8039
	Rand. read		5421	2254	5098	5516	5964	8384

(a) West1

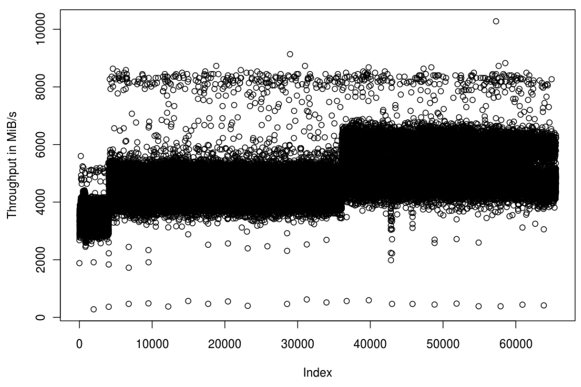
Memory	Pattern	Record size	*Average*	Min	1st quartile	Mean	3rd quartile	Max
200 MiB	Seq. write	16 KiB	103.3	0.05	835.1	1006	1158	1682
	Seq. read		116.0	1.52	3685.0	3559	4223	6706
	Rand. write	16 KiB	6.3	0.01	1105.0	1316	1492	5243
	Rand. read		2.6	0.17	1.8	382	4.3	4976
	Rand. write	256 KiB	44.6	0.46	969.2	1286	1487	5031
	Rand. read		27.2	12.10	21.5	250	35	7490
1,000 MiB	Seq. write	16 KiB	121.7	0.02	1027	1245	1517	1700
	Seq. read		116.0	0.8	3527	3425	4156	6706
	Rand. write	16 KiB	10.3	0.003	1282	1729	2126	5502
	Rand. read		27.0	0.8	3094	3353	4037	8828
	Rand. write	256 KiB	65.7	0.2	1312	1857	2365	5425
	Rand. read		100.0	11.1	46.3	3286	4920	8945
unlimited	Seq. write	16 KiB	1474	108.3	1440	1513	1653	1910
	Seq. read		5996	40.1	6104	6119	6430	6735
	Rand. write	16 KiB	2119	70.2	1834	2276	2888	5407
	Rand. read		6600	340.1	6565	6678	6944	12020
	Rand. write	256 KiB	2323	756.2	2000	2537	3441	5361
	Rand. read		6271	1277.0	6361	6333	6465	9498
on `tmpfs`	Seq. write	16 KiB	2350	103.9	2294	2405	2626	2780
	Seq. read		5493	316.8	5580	5590	5852	6200
	Rand. write	16 KiB	2167	38.91	1929	2357	2878	8980
	Rand. read		4634	279.9	4316	4751	5073	10280
	Rand. write	256 KiB	2969	509.9	2523	3276	4462	8010
	Rand. read		5478	2560	5130	5564	5991	8406

(b) West2

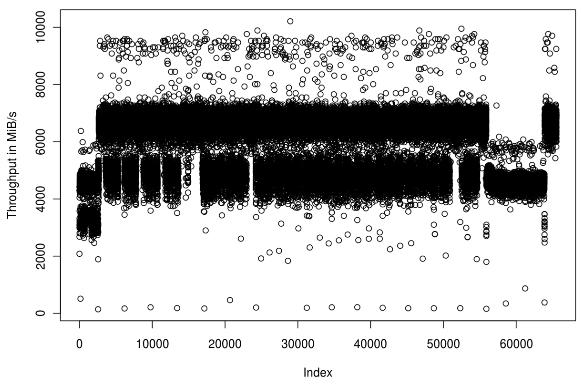
Table 3.10.: I/O throughput in MiB/s for individual operations as measured by `posix-io-timing`. The *average* value is the throughput overead the whole run.



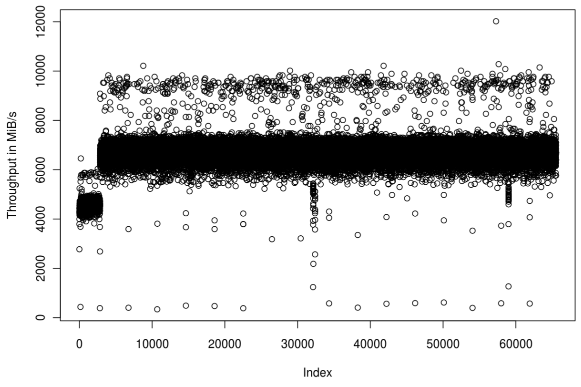
(a) West1 random read from *tmpfs*.



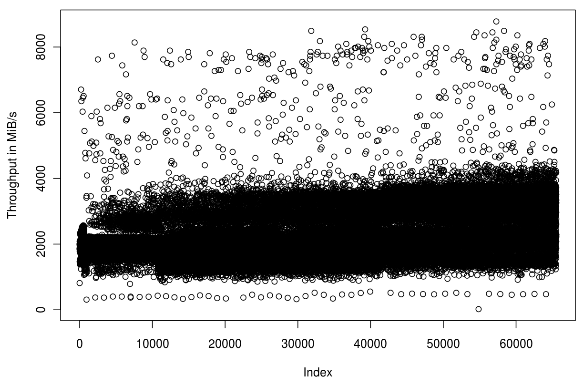
(b) West2 random read from *tmpfs*.



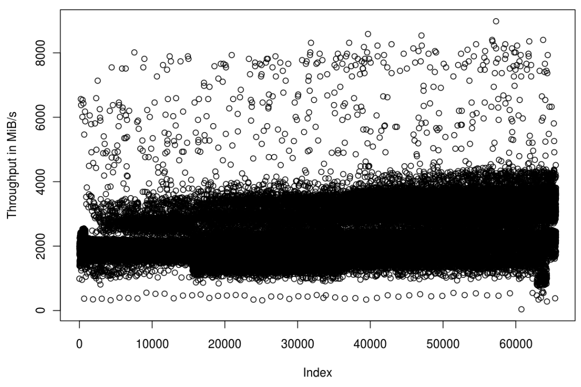
(c) West1 random read with unlimited memory.



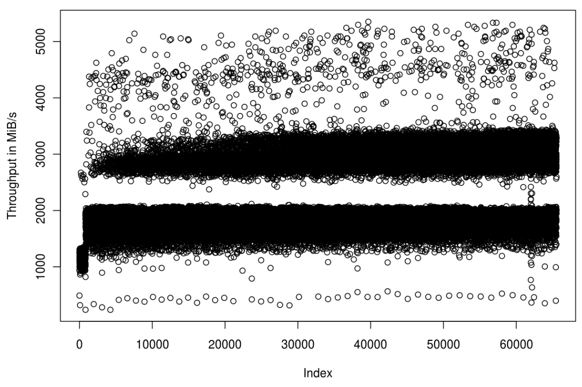
(d) West2 random read with unlimited memory.



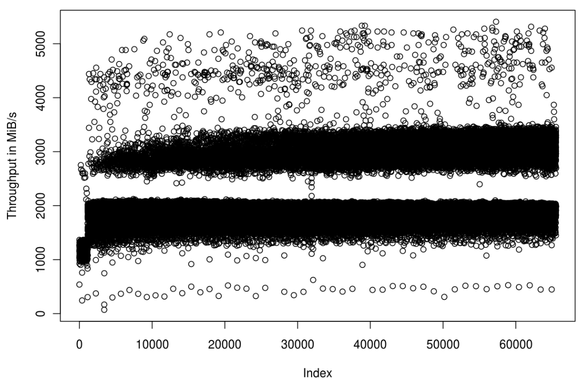
(e) West1 random write to *tmpfs*.



(f) West2 random write to *tmpfs*.

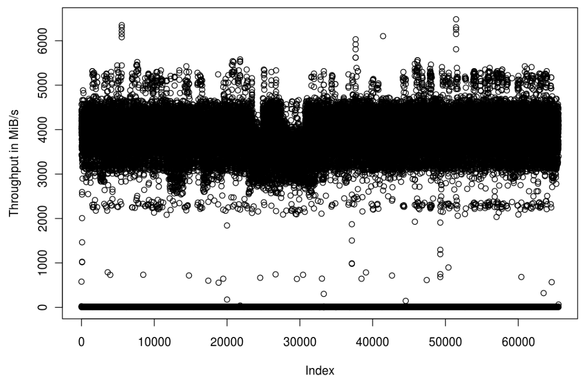


(g) West1 random write with unlimited memory.

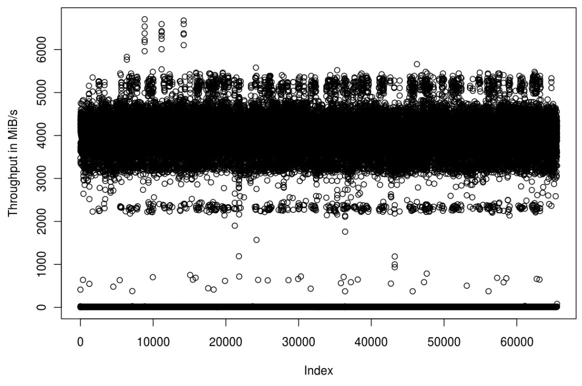


(h) West2 random write with unlimited memory.

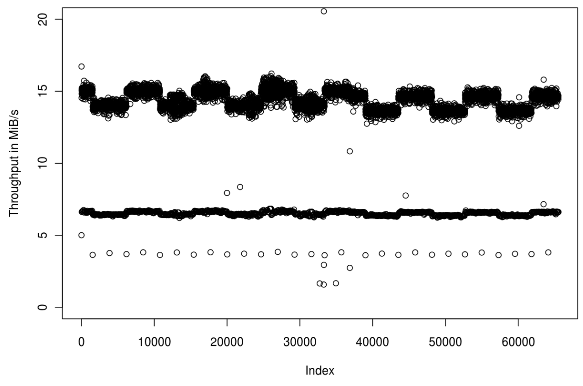
Figure 3.28.: Performance measured with `posix-io-timing` on two nodes for random access patterns and 16 KiB records. Results for accessing data on `tmpfs` and on Ext4 with unlimited memory are provided.



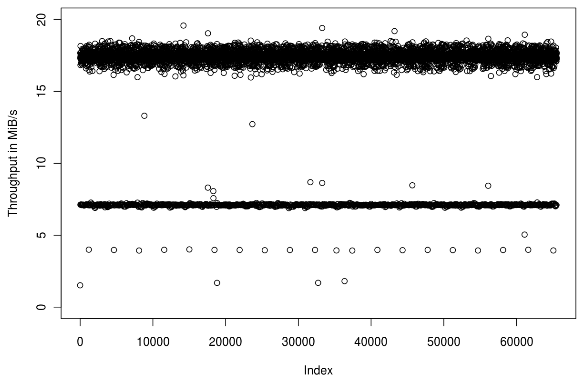
(a) West1 sequential read of 16 KiB records.



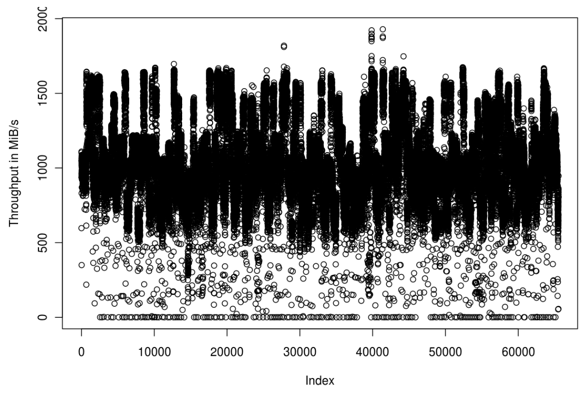
(b) West2 sequential read of 16 KiB records.



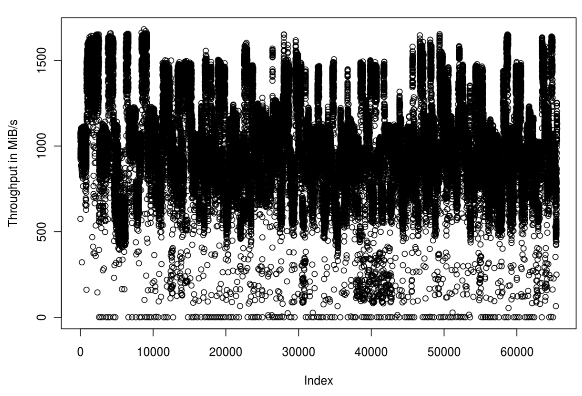
(c) West1 sequential read of 16 KiB records, the y-axis is limited to 20 MiB/s.



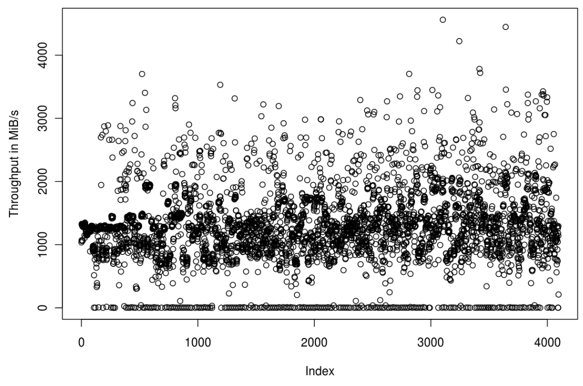
(d) West2 sequential read of 16 KiB records, the y-axis is limited to 20 MiB/s.



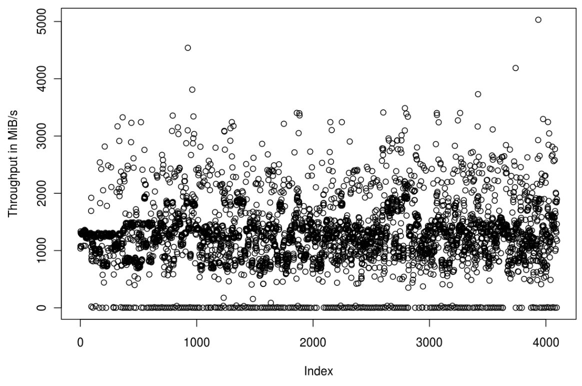
(e) West1 sequential write of 16 KiB records.



(f) West2 sequential write of 16 KiB records.

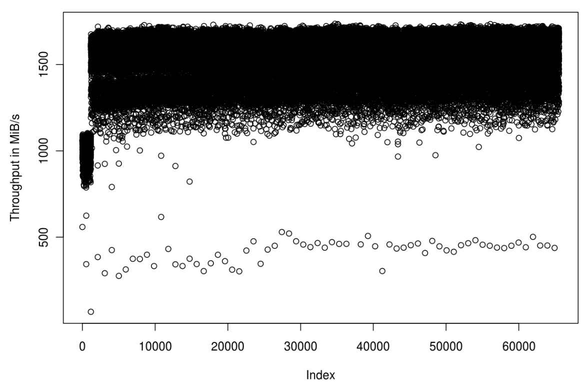


(g) West1 random write of 256 KiB records .

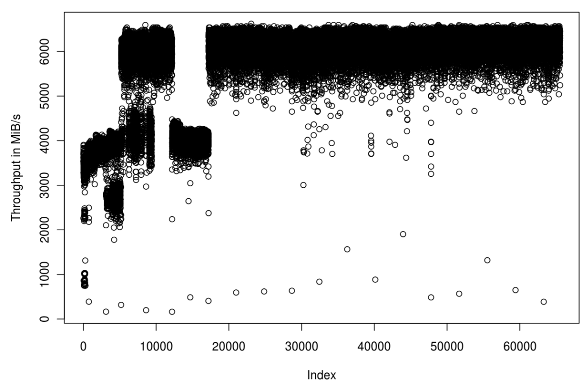


(h) West2 random write of 256 KiB records.

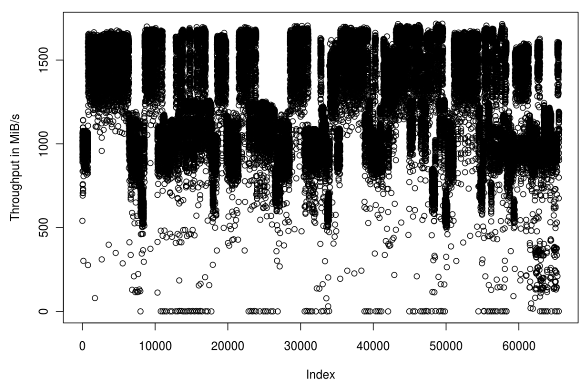
Figure 3.29.: Performance measured with `posix-io-timing` on two nodes for several access patterns – memory is limited to 200 MiB.



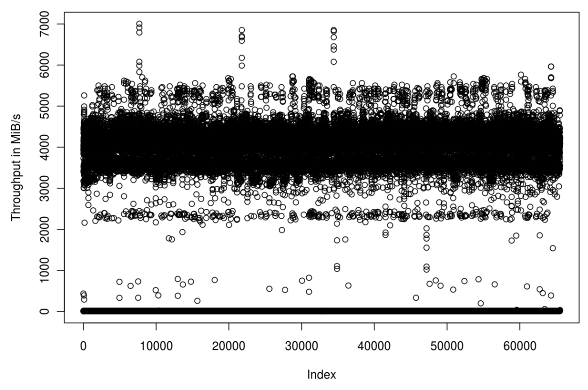
(a) Sequential write – memory is not limited.



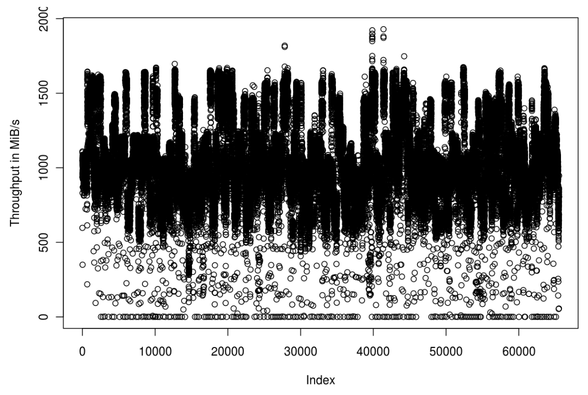
(b) Sequential read – memory is not limited.



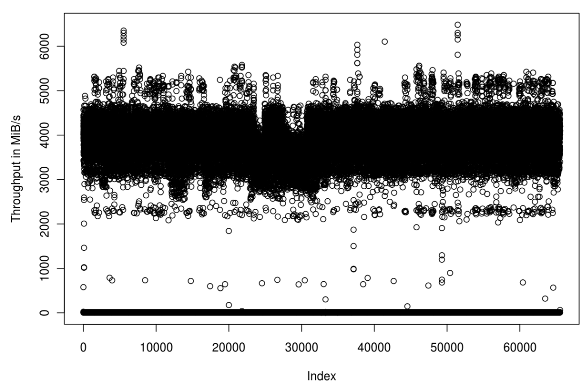
(c) Sequential write – memory is limited to 1 GiB.



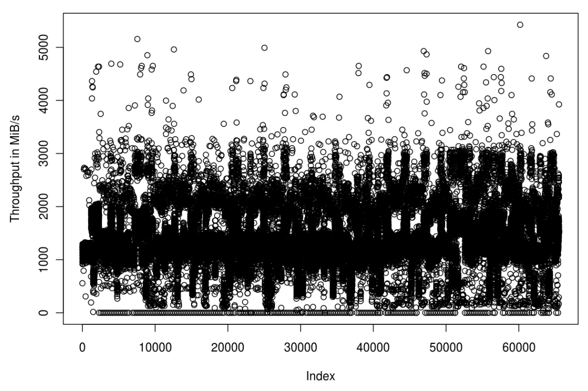
(d) Sequential read – memory is limited to 1 GiB.



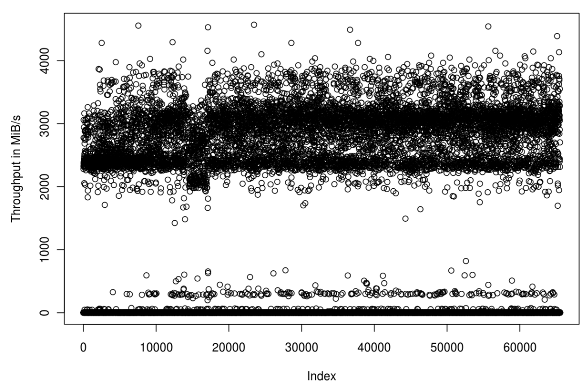
(e) Sequential write – memory is limited to 200 MiB.



(f) Sequential read – memory is limited to 200 MiB.

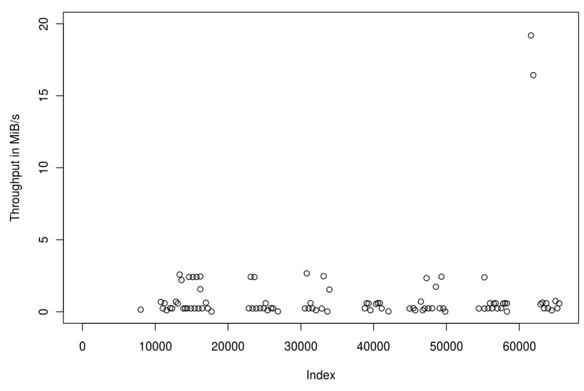


(g) Random write – memory is limited to 200 MiB.

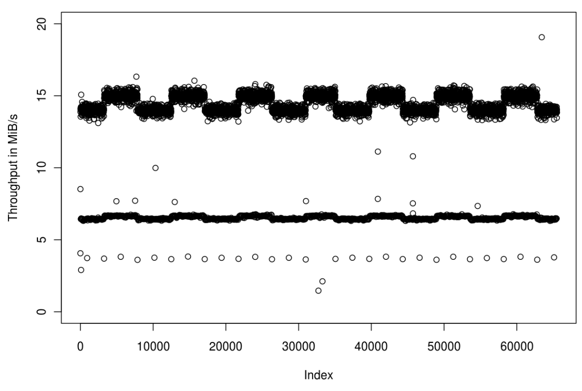


(h) Random read – memory is limited to 200 MiB.

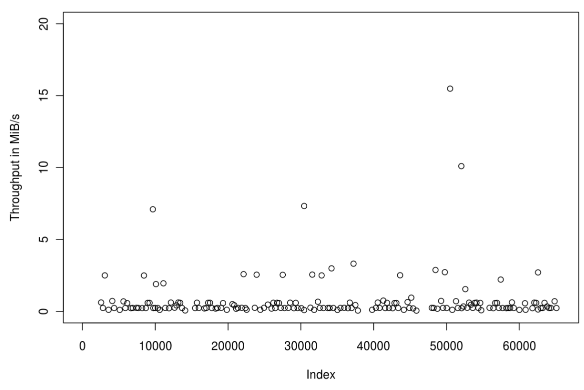
Figure 3.30.: Performance measured with `posix-io-timing` for several access patterns and 16 KiB records on West1 – all data points.



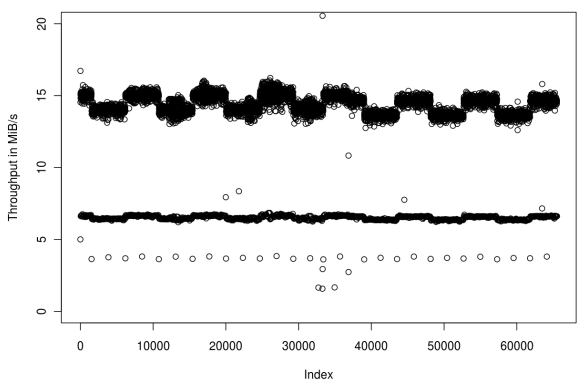
(a) Sequential write – memory is limited to 1 GiB.



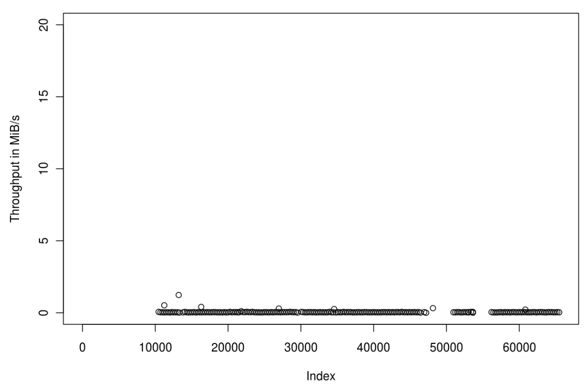
(b) Sequential read – memory is limited to 1 GiB.



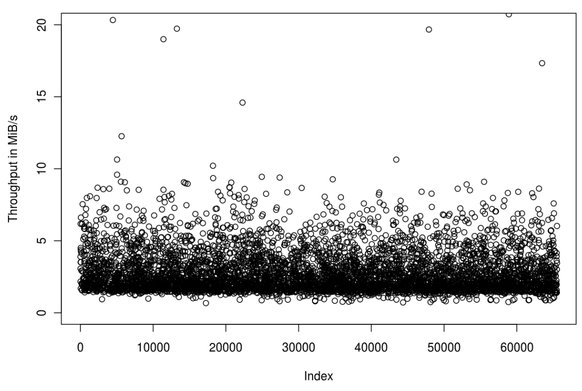
(c) Sequential write – memory is limited to 200 MiB.



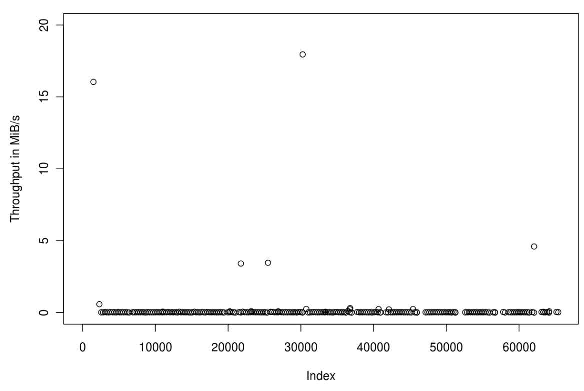
(d) Sequential read – memory is limited to 200 MiB.



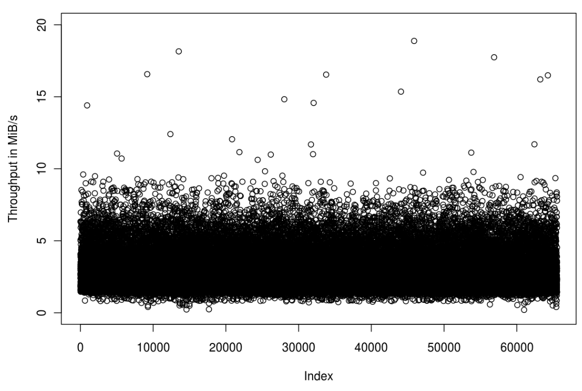
(e) Random write – memory is limited to 1 GiB.



(f) Random read – memory is limited to 1 GiB.



(g) Random write – memory is limited to 200 MiB.



(h) Random read – memory is limited to 200 MiB.

Figure 3.31.: Performance measured with `posix-io-timing` for several access patterns and 16 KiB records – the y-axis is limited to 20 MiB/s.

disk operations the situation is even worse, the first (and third) quartile is below the mean value for several access patterns. Presumably, because these 75% of operations are so slow (below 34 MiB/s), they are expected to hit disk.

c) The timelines' patterns are quite complex in their nature (for example look at Figure 3.28). Also, timelines for read and write patterns look substantially different. Even observed timings of accesses which are completely cached are rather complicated. Cached operations achieve a much better throughput and in general read access to cached data is faster than a write. Some operations are conducted at full memory speed (the measured throughput is around 8,000 MiB/s).

d) Next, the performance deviation between the two nodes is assessed quantitatively. By comparing the `tmpfs` results of both hosts it can be observed that the average values for all accesses are quite similar (e.g., 2,380 MiB/s vs. 2,350 MiB/s for sequential writes), so is the maximum and the three quartiles (compare Table 3.10a and Table 3.10b). A similar behavior can also be seen in the timelines (for example compare Figure 3.28a and Figure 3.28b). Due to the identical random seed of the benchmark and the hardware, this is expected.

e) The values for unlimited memory are also quite similar between West1 and West2, they are close to the values of `tmpfs` (compare Table 3.10a and Table 3.10b). A bit surprising is that cached read performance on the disk drives achieves better performance than reading from `tmpfs` in all cases because `tmpfs` is implemented in the page cache, which is used by all file systems, too.

The difference could be explained by the short runtime of the benchmark but also be caused by the implementation differences between `tmpfs` and the page cache implementation for regular file systems. This can be assessed with the timelines.

f) By comparing the timelines for the two scenarios it can been observed that the behavior of the two implementations differ qualitatively (see Figure 3.28). All four read plots start with a performance around 4,000 MiB/s and performance improves after 1,000 records have been read. Additionally, performance of reading data from `tmpfs` increases after 40,000 records while the performance of reading data from the page cache achieves a higher throughput. Although, the timelines for random read on Ext4 reveals an additional band on West1, it looks similar to the one obtained on West2. Further, most figures look very similar between the two nodes (for example see Figure 3.29). Presumably, the file system implementations are the reason.

g) The performance of sequential accesses with 16 KiB of data are an exception to the rule that both machines behave similar; West2 achieves better performance than West2 (compare Table 3.10a and Table 3.10b). It can be also seen that the average random access performance with a restricted amount of memory is higher. e.g., West2 achieves 116 MiB/s average read performance with 200 MiB of main memory, while West1 achieves 95.8 MiB/s.

To assess this further look at the timelines in Figure 3.29. First of all, the overall sequential read and sequential write behavior of the two nodes looks similar. However, when the y-axis is limited to 20 MiB/s difference between the two nodes can be observed. Due to the limitation on the y-axis these accesses are probably executed by the disk drive and not by the page cache: Performance of the sequential reads on West1 alternate in a zigzag pattern around 15 MiB/s while the performance is stable at 17 MiB/s for West2 (look at Figure 3.31d and Figure 3.29d). Two other horizontal bands can be identified on both timelines (one at 4 MiB/s and one at 6 MiB/s). While this explains the quantitatively higher performance of West2, the reason for the pattern is unknown. Since most other figures look quite similar further analysis is based on the results of West1.

h) The pattern from memory access is partially observed in all diagrams and the appearance implies that data is cached. In writes in principle data is cached for write-behind (refer to Section 3.6.1 for a description and the limitations). In reads a high performance indicates that data is already available, either due to read-ahead or that it is still in the page cache from the write-phase. Since the 12 GiB main memory suffices to cache all data, the sequential read is almost identical to the accesses from `tmpfs` (compare Figure 3.30b with Figure 3.28a).

With a decrease in available free memory more and more accesses hit the disk and less operations benefit from the caching (compare for example Figure 3.30a, Figure 3.30c and Figure 3.30e).

i) The random write pattern for 200 MiB has more variance in the range between 2000 and 0 MiB/s than the sequential read pattern (see Figure 3.30g and Figure 3.30e). There are also some calls which actually perform faster in the random write. These are probably caused by a re-write of already cached data blocks.

j) The higher deviation might be caused by the write-back policy of Linux: When enough pages are modified write-back of data is blocked by the Linux kernel and the process must write out data by itself. This would also explain the few long-running calls in Figure 3.31a, the frequency increases with memory pressure, e.g., when less memory is available for caching (Figure 3.31c).

Under the random write access pattern (Figure 3.31e) the first 10,000 writes of 16 KiB, i.e., roughly 160 MiB are stored in the cache without blocking the application. Then slow operations are observed, since the kernel blocks the operations, later there are two regions around index 50,000 in which write-back is again very fast. In the sequential write case clusters of slow operations can be seen in an almost regular pattern every 10,000 operations.

k) Writing bigger records reveals a comparable behavior, but varies more, see Figure 3.30g and Figure 3.29g. In this case, the time to access 16 times more pages than for 16 KiB, thus presumably random effects that manifest during access of 16 KiB of data add up. Thus, accessing a larger amount of data is a superposition of accessing individual pages from the cache. Since the experiment with larger block sizes has less samples it does not look so dense.

l) In random read patterns which cannot be cached a random distributed access pattern is expected, that means every data point should vary independently from the previous. For actual performed operations this can be seen in Figure 3.31f. Even for 200 MiB of main memory some operations can be cached, approximately 20% of accesses are expected to hit the cache and thus most operations hit the disk. With the 1 GiB of main memory almost all accesses can be cached. Due to the random character of the benchmark, the accesses that hit disk still look alike but are less dense (see Figure 3.31h). and most accesses aggregate around 2,200 MiB/s and 3,000 MiB/s.

Conclusions Assessing average performance of a system is important, however, it cannot reveal the complex interplay of hardware and software. Operations show a variability by several orders of magnitude. Since the mechanical parts of an HDD result in very slow performance, many optimization strategies such as read-ahead and write-behind are implemented in operating systems and hardware. In the best case, these optimizations can improve performance up to memory speed (up to 10,000 MiB/s), while operations with a speed of 0.01 MiB/s have been observed. Therefore, understanding behavior of the system requires a benchmark which measures performance of individual operations.

Several performance critical aspects are identified and discussed during the experiments. While many observations could be explained, some aspects remain unclear. The combination of optimizations in the operating system, the file system and the hardware cause very complex patterns of behavior. In the timelines, horizontal performance bands are visible. It can be assumed that the operations of each band share the I/O path including applied optimizations.

When an application wants to ensure data consistency with synchronous operations, performance degrades seriously showing the impact of the cache. Since a file system maps logical data of a file to physical blocks, its implementation and on-disk format interferes with the application's access pattern and thus influences achievable performance. For example, appending data to a file triggers block allocation on the medium and involves metadata operations to change the list of free blocks. In our case, this caused sequential write operations to behave similar to random I/O.

In principle, both identically configured nodes behave quantitatively and qualitatively similar in many experiments. However, in several tests differences can be observed in the timelines and even a divergence in the average performance becomes visible. Presumably, these effects are caused by minimal hardware and software fluctuations.

To predict performance of a single access accurately, even these hidden effects must be modeled, described and the exact system state must be known. Therefore, an accurate prediction of every single I/O operation seems to be impossible.

3.7. Chapter Summary

In this chapter the working groups cluster is introduced and an analysis is conducted in which the quantitative and qualitative behavior of several hardware components is discussed. To this end, benchmarks are introduced that measure performance of memory, point-to-point communication, and the I/O subsystem. Results of these benchmark are cross-validated with other benchmarks to ensure that results are trustworthy.

Several interesting and even counter-intuitive results have been found:

- *The operating system behavior can be parameterized with many switches in the* `/proc` *and* `/sys` *pseudo file system that influence performance of CPU, network and I/O subsystem. Also, performance depends on the kernel version and probably on the compile flags used to generate the kernel.*
- *Hardware components of a system can be characterized using timelines. In most cases the timelines obtained on different nodes are very similar, because the deployed hardware and software is identical. For a complex memory kernel (reading and two times writing data) and for operations that actually hit disk differences can be found between the nodes that exceeds the variability of a rerun of the benchmark. Thus, hardware and software causes non-neglectable fluctuation that results in a large variability of behavior.*
- *The time for a single operation (memory, network or disk) shows a high variability; the span between minimum and maximum can be very wide. For example the network latency can vary by a factor of 5. Also, almost any duration is imaginable for the slowest operation. In many experiments at least the 50% of timings between first and third quartile might be close together, but still the quartiles can vary to a large extent.*
- *With the help of timelines operations can be identified that are influenced by background activity – as long as the background activity changes the behavior for several subsequent operations.*
- *In point-to-point communication the average value determined by kernels that run several seconds are often stable, but may also vary between the runs. For the PingPong kernel and Open MPI the variability is around 10%, but the the time to transfer 1 GiB of data between two nodes with* `MPI_Sendrecv()` *varies between 71 MiB/s and 147 MiB/s.*
- *Performance of inter-process communication depends on the used MPI implementation. Beside the design difference in inter-node communication Open MPI and MPICH2 behave different in intra-node communication. The difference depends on the used message size.*
- *Due to the variance of the measurements a specification of results with a high accuracy, i.e., many positions after the decimal point, is not adequate and suggest a precision which is not measurable. However, since the variability has been assessed in depth in this chapter those reference values and their deviation can be assessed quantitatively.*
- *Access patterns have a large impact on memory and I/O performance. For instance CPU features such as L1, L2 and L3 cache behavior influence memory throughput to a large extent. Therefore, the intra-node communication and I/O which fits into these cache levels is likely to be handled much faster.*
- *I/O performance of individual operations depends on many factors: the OS cache, scheduler and the directly attached HDD. Optimizations in operating system and hard disk drive make it hard to assesses the cause for observed behavior. Just by looking at the measured timelines of cached I/O the system looks rather chaotic and unpredictable. By analyzing direct and synchronous I/O several file system and kernel dependent*

aspects have been isolated and discussed. While simulators such as DiskSim try to accurately simulate a single disks behavior an accurate simulation must cover the operating system that controls block I/O and the file system dependent behavior.

- *Most important, variance of the analyzed components cannot be described by a mathematical probability density function. Histograms of accessing 1 MiB of memory show multiple peaks and the timelines of I/O operations and even point-to-point communication reveal several horizontal bands on which timings aggregate. This kind of non-linear behavior is caused by the complex interplay of software and hardware effects. Performance of cached I/O measured with IOzone for instance improved with increasing file sizes but drops when cache does not suffice to hold data (see Figure 3.23).*

Consequently, exact simulation of a single node would require to deal with CPU and memory behavior but also requires to handle network protocol, the file system (here Ext4), kernel features for optimizations in TCP/IP, the virtual memory management, and background activity caused by services.

Clearly, an accurate cluster simulation of all those factors is not only tedious, but infeasible since all parameters of the system must be known and understood to some extent. Therefore, an abstraction to the real world system is needed which can be used for experimentation.

In the next chapter the HDTrace environment is introduced. This software eco-system contains a model of hardware and software, and a simulator implementing this model. The model tries to cover the most important performance factors and abstracts from the low-level aspects such as the network protocol. During the validation of the simulator the values measured in this section will be used.

Bibliography

[AKCSD08] KV Aneesh Kumar, M. Cao, J.R. Santos, and A. Dilger. Ext4 Block and Inode Allocator Improvements. In *Proceedings of the Linux Symposium*, pages 263–274, 2008.

[BM06] Darius Buntinas and Guillaume Mercier. Implementation and Shared-memory Evaluation of MPICH2 Over the Nemesis Communication Subsystem. In *Proceedings of the Euro PVM/MPI Conference*. Springer, 2006.

[CK11] C.Y. Cher and E. Kursun. Exploring the Effects of On-chip Thermal Variation on High-performance Multicore Architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(1):2, 2011.

[FRJ+07] R. Franch, P. Restle, N. James, W. Huott, J. Friedrich, R. Dixon, S. Weitzel, K. Van Goor, and G. Salem. On-chip Timing Uncertainty Measurements on IBM Microprocessors. In *Test Conference, 2007. ITC 2007. IEEE International*, pages 1–7. IEEE, 2007.

[GGJL10] Ekaterina Gorelkina, Sergey Grekhov, Jaehoon Jeong, and Mikhail Levin. Prediction of Optimal Readahead Parameter in Linux by Using Monitoring Tool. In *Proceedings of the Linux Symposium*, pages 83–91, July 2010.

[Int] Intel. Intel® Xeon® Processor X5650. Online: `http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-%2812M-Cache-2_66-GHz-6_40-GTs-Intel-QPI%29`.

[Int09a] Intel. An Introduction to the Intel® QuickPath Interconnect. Online: `http://www.intel.com/technology/quickpath/introduction.pdf`, January 2009.

[Int09b] Intel. Intel® 5520 Chipset and Intel® 5500 Chipset Datasheet. Online: `http://www.intel.com/content/www/us/en/chipsets/server-chipsets/server-chipset-5500.html`, March 2009.

[Int11] Intel. Intel® Xeon® Processor 5600 Series. Online: `http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-5000-sequence.html`, June 2011.

[KNL10] Julian Kunkel, Jan C. Neddermeyer, and Thomas Ludwig. Classification of Network Computers Based on Distribution of ICMP-echo Round-trip Times. Technical Report 1, Staats- und Universitätsbibliothek Hamburg, 09 2010.

[Mak08] TM Mak. Jitters in High Performance Microprocessors. In *Test Conference, 2008. ITC 2008. IEEE International*, pages 1–6. IEEE, 2008.

[MCB+07a] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium*, 2007.

[MCB+07b] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium*. sn, 2007.

[Net08] Netgear. ProSafe 48-port Gigabit Smart Switch DataSheet. Online: `http://www.netgear.com/images/gs748t_ds_19dec0818-5390.pdf`, December 2008.

[RBOS08] B.F. Romanescu, M.E. Bauer, S. Ozev, and D.J. Sorin. Reducing the Impact of Intra-core Process Variability with Criticality-based Resource Allocation and Prefetching. In *Proceedings of the 5th conference on Computing frontiers*, pages 129–138. ACM, 2008.

[Rol09] Trent Rolf. Cache Organization and Memory Management of the Intel Nehalem Computer Architecture. *Computer Engineering*, page 6, 2009.

[Sea10] Seagate. Product Manual – Barracuda 7200.12 Serial ATA. Online: `http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%207200.12/100529369h.pdf`, 2010.

[TEFK05] D. Tsafrir, Y. Etsion, D.G. Feitelson, and S. Kirkpatrick. System Noise, OS Clock Ticks, and Fine-grained Parallel Applications. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312. ACM, 2005.

[Wik11] Wikipedia, the Free Encyclopedia, 2011.

[Wu10] Fengguang Wu. Sequential File Prefetching in Linux. page 218, 2010.

Chapter 4 IV

HDTrace Environment

In this chapter software created and extended for this thesis is described.

The GPL licensed HDTrace environment is the tracing environment developed in the context of this thesis which gathers information necessary for simulation, to assess results and to increase insight in MPI activity. It consists of extensions of MPICH2 and PVFS for tracing internal behavior. Compared to existing tracing environments HDTrace concentrates on evaluation of new ideas and collects all information required for simulation. The sections of this chapter introduce individual components, and rationales and design considerations behind the current implementations.

In summary, HDTrace provides unprecedented insight in system behavior by offering:

- *A flexible trace format which allows recording of arbitrary information.*
- *A set of libraries for the instrumentation of applications and modification of middleware to permit recording application and PVFS activity, MPI internal communication and system behavior.*
- *A visualization tool which explores new analysis and visualization methods for system statistics such as energy metrics and for parallel I/O.*
- *A simulator with the capability to replay recorded application traces and I/O behavior in virtual cluster environments.*

By offering these features, HDTrace targets the stated research goals of this thesis in the following way:

- *Understanding of performance factors in cluster systems and application execution is fostered by the ability of the trace environment to record and analyze system behavior with the visualization tool and the simulator.*
- *With the extended insight into I/O and communication middleware, bottlenecks in a cluster configuration and their causes in application logic and the software stack can be localized more easily. Further, the behavior can be studied in silico by using the replay mechanisms of the simulator.*
- *Together, the advanced introspection capabilities into I/O activity and the simulation on system and application level permit a novel way of evaluating and optimizing the I/O path, the client-server communication and the server cache layers.*
- *Since the system characteristics can be defined at will during a simulation study, an extrapolation of system performance towards future systems becomes possible. Additionally, the modularity of the simulator allows experimentation with alternative hardware and software models that are not yet available in the market.*
- *The modular software model of the simulator and a global view over the system permits experimentation with MPI-internal algorithms to gear them towards application and systems. Together with the visualization tool, the simulation results can be evaluated and compared to the current behavior of a real system.*
- *Similarly, the modularity of the simulator and the visualization features ease evaluation of new MPI commands and alternative MPI semantics to assess their quality before they are actually implemented in a real system.*
- *Teaching of the above aspects is fostered by providing an environment which increases insight into a system while easing experimentation with it.*

This chapter is organized as follows. First, an overview of all software components is given in Section 4.1. Section 4.2 describes the developed trace format. This format provides all necessary information for enhanced visualization and for the developed simulator. The software which records the trace information from MPI applications is described briefly in Section 4.3. Sunshot, the visualization tool for traces and its capabilities are presented in Section 4.4.

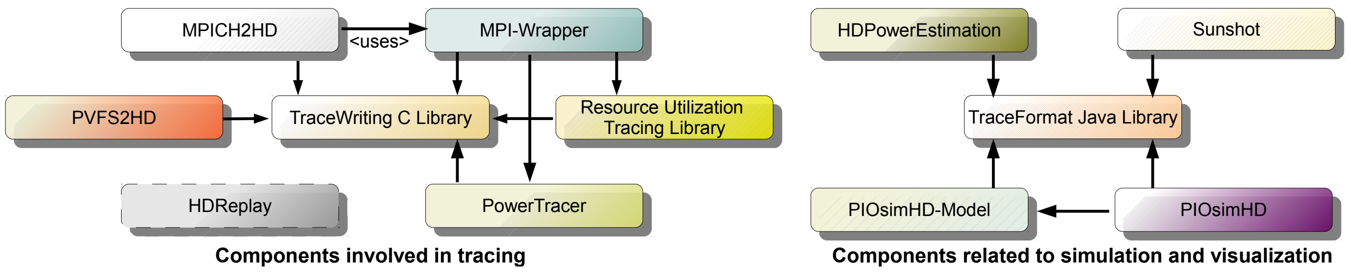


Figure 4.1.: HDTrace components and dependencies between components.

As the author supervised several contributions to HDTrace, those are listed in Section 4.5. While detailed descriptions of all components is beyond the scope of this thesis, these contributions offer further insight in several design and implementation aspects relevant to HDTrace. Since the simulator is of major interest for this thesis, further descriptions of the simulator are provided in a separate chapter.

4.1. Component Overview

The components of the *HDTrace environment* and their dependencies are shown in Figure 4.1. Two kinds of components can be distinguished: the components involved in tracing and the components responsible for simulation and visualization. The former software is primarily coded in C, the latter is written in Java.

Except for PVFS2HD and HDReplay, all tracing components are libraries which are linked into an application, at run-time each process has its own copy of all required libraries. With the exception of the MPI-Wrapper, libraries do not communicate with other processes, thus each library instance is completely independent. All libraries are controlled by the MPI-Wrapper. The MPI-Wrapper is the only component which communicates at run-time with other instances. HDPowerEstimation, Sunshot and PIOsimHD are independent programs. Details of the workflow are provided in Section 4.3.1.

Components involved in tracing

- The *TraceWriting C Library* is responsible to store events of an application and parallel file system in XML trace files and statistics about the execution in a binary format with an XML description header. A project file links together trace and statistic files of multiple sources. The format is designed to avoid post-mortem re-write of whole trace files. Instead, only the project file or trace file header must be adjusted. Statistics group a number of arbitrary counters together with a timestamp. Relations between activities of different programs can be recorded explicitly. HDTrace uses the local clock to generate timestamps for the data, therefore, nodes should have synchronized clocks (e.g., by using NTP).
  Additionally, to compensate drifts in the clocks, offsets between timestamps of the trace files can be fixed post-mortem by adjusting the file headers, or inside the trace visualization tool *Sunshot*.
- The *Resource Utilization Tracing Library* (libRUT) provides routines to start periodic gathering of system information and utilization from the operating system about network, I/O and CPU usage. This information is stored in statistics files and can be referred easily within a project file.
- *PowerTracer* is a library and a tool which periodically traces information about power usage from an external power meter in statistics files. Therewith, it becomes possible to visualize node energy metrics together with their MPI activities.
- The *MPI-Wrapper* uses PMPI to intercept MPI calls and the trace library to store the observed events. Additional information about file accesses, communicators and the mapping from application to hardware are recorded as well. It controls the functionality of the *Resource Utilization Tracing Library* and the *PowerTracer*.

- *PVFS2HD* contains a modified PVFS that allow tracing of internal activities within client library, kernel module and server. MPI activities of PVFS clients can be related to PVFS server activities. Further, statistics about utilization of PVFS-internal layers are computed and stored. Server and client activity can be visualized together to understand causal relations. PVFS2HD can use *libRUT* to gather information from the operating system on PVFS nodes and the clients as well. The currently instrumented version of PVFS is orangefs-2.8.3-20101113.
- *MPICH2HD* is a slightly modified version of the MPI implementation *MPICH2* that traces MPI internals: The point-to-point operations within collective calls are recorded and PVFS internal calls as well. It is based on MPICH2 1.3. HDTrace also supports tracing with Open MPI, but it does not instrument Open MPI to record communication details within the library.
- *HDReplay* is a program which is intended to replay recorded MPI-IO behavior on arbitrary environments. Therewith, potential bottlenecks of communication and parallel I/O can be evaluated before an application is ported. It is currently under development (more details are discussed below).

Components related to simulation and visualization

- The *TraceFormat* library provides interfaces to read and write trace files in Java.
- *PIOsimHD-Model* supports the simulator by providing an abstraction layer for application traces and for the cluster model. MPI commands which are read from the trace files are converted to a sequence of commands which can be executed by the simulator. Methods are provided to create and process models of components and commands; each model is represented by a Java (model) class. Actually, a model class can be considered as a container which carries the parameters for a modeled component or command that should be executed.
- *PIOsimHD* is the discrete-event simulator which simulates hardware and software behavior on application and system level. It offers several alternative implementations for hardware components and command behavior. For each modeled component or command one of the available implementations can be selected. For instance the `MPI_Bcast()` command can be implemented in multiple variants; while using the same parameters each implementation can operate with its own communication algorithm. The simulation generates trace files of the virtual execution for further inspection, information about processing on internal components such as network switches can be included to investigate the operation on all components.
- Trace files of application, PVFS or simulation runs are visualized by *Sunshot*, a Java-Swing application, which is a major rewrite of Jumpshot. The original Jumpshot viewer is distributed with the MPI implementation MPICH2, it visualizes the *SLOG2* trace format of the *MPI Parallel Environment* (MPE) (see Page 77 for a detailed description).
- *HDPowerEstimation* estimates the energy consumption of a node and, therewith, power saving strategies can be tested. Internally it reads the node statistics recorded by libRUT to estimate the energy consumed by the system components (CPU, network and disk).

4.2. HDTrace Format

When the project started at the end of 2008, the existing trace formats such as the OTF, the SLOG2 format and the TAU trace format were not suitable to provide all information necessary for the desired simulation. Thus, the design criteria for a new format are driven from the idea to support simulation. In the later stages of the project, the evaluation of new visualization ideas played a role as well. As a consequence a full featured tracing and visualization environment for arbitrary processes and activities has been developed.

4.2.1. Design Criteria

The following criteria guided the design of HDTrace. For each of them motivating examples are presented and some implementation details are provided.

- **Flexibility**: The trace format should permit to record arbitrary trace data, this especially includes MPI, OpenMP and parallel file system (client and server) activity. Semantics of recorded events is decoupled from the stored raw information. Instead, the tools writing trace information and the tools analyzing the trace must be aware of the semantics. It is the responsibility of any processing tool such as the Sunshot viewer to provide additional visualization capabilities based on the semantics. Most information which can be recorded is not mandatory, attributes recorded together with an event can be chosen at the moment the event is written.

  The mentioned flexibility enables users to extend the recorded information easily. It is used for instance to record any number of IDs for the `MPI_Wait*()` calls, or to include the important parameters for an MPI call. Once new attributes or XML tags are used their semantics should be defined to avoid misinterpretation of the stored information. Currently, this type of information is not stored explicitly. However, theoretically this information could be stored in a simple text file; whenever new information should be recorded its semantics and syntax could be added to the file.

- **Simple on-disk file format**: The simpler the format, the easier it is for users to quickly create new scripts and tools to process and analyze the traces from the command line. With an understandable and human readable format the user can manipulate the traces post-mortem with arbitrary programming languages or tools. Further, a simple trace format allows direct modification of existing trace events in an editor, and example trace files can be easily forged. The format must be simple enough to allow efficient parsing by a machine, thus a description via a context-free grammar is favorable.

- **Lightweight post-processing**: The simple on-disk format should foster a lightweight post-processing of the traces. Common manipulation of the trace files and project should not require to rewrite all trace files of the project since the trace files can be very large.

  For instance, with HDTrace a user can shift the timestamps of all trace events within a file by adjusting the reference time in the file header. The reference time is initialized with many digits that can be changed. Thus, only the first block of the file must be rewritten to change a time offset accordingly. Also, event sources, such as a process or a parallel file system server, can be added and removed to the project easily – in many cases by just moving the files into the right place.

- **Identity of the event source**: In many popular tracing formats the event source is enumerated, e.g., they are enumerated as Process 0 to Process N. Thus, it is not clear on which node or system a particular trace source is run. Also, an event source could be an MPI process or a process of a parallel file system.

  With HDTrace human readable identifiers are given to the event sources, also the event sources are arranged in a hierarchical *topology* tree. Information relevant to multiple event sources can be added to the parent node in the tree, then this information is valid for all children in the tree. For example, node statistics such as CPU utilization are valid for all processes and parallel file system servers that run on the node. Hence, with HDTrace the mapping of recorded information to processes (or event sources) becomes clear.

- **Causal relation of events**: An event source such as an MPI process might trigger activity on other event sources. As the cause-and-effect chains of events are generally of interest, with HDTrace the program developer can explicitly relate events with other events.

  Trace formats like OTF or TAU record explicitly the activity of “sending” and “receiving” an MPI message between two processes. The semantics of HDTrace is of a higher level, due to the fact that the causal relation do not have to be a simple message. For instance in the PVFS client/server communication, HDTrace needs to overhaul the message concept. The relation concept of HDTrace relates two events, one event being the cause of another one. This permits to relate client activity with server activity, and also to relate the activities of any intermediate layers with their callee.

- **Recording of bare activity**: In the trace format, there should be no information embedded which is used just for the sake of a post-processing tool like the visualization tool. Instead tools should derive any required information by themselves directly from the recorded trace files[1]. For instance, arrows which relate sending and receiving of MPI calls are computed directly in Sunshot, and thus not added to the trace explicitly.

  Since just the bare activity is stored, it might still be advantageous for some tools to derive information and store this information in additional files to speedup re-use of this information. But it is clear that it is not required to describe the original data. For example, the recorded traces do not permit access to random positions. This issue can be mitigated by indexing files post-mortem, eventually even automatically when they are read the first time.

- **Time synchronization**: Typically hardware clocks have a slight systematic drift depending on temperature and minimal hardware fluctuations. Therefore, from a global view, times of disjoint components always differ slightly, except if there is special hardware which synchronizes the remote timers.

  Local time should be accurate enough to allow assessing of trace results. Assume that all nodes have an identical offset to a global clock, then all recorded activity could be related between each other and thus this case is unproblematic. However, when one node has a positive time offset to the wall-clock and another has a negative offset, then the discrepancy accumulate when the events should be related.

  Synchronization of clocks in a distributed system is tedious. Existing tracing environments try to compensate for the time offset between the processes inside the tracing environment. HDTrace does not, because a *Network Time Protocol* (NTP) service is deployed on many cluster systems to ensure an synchronized clock for debugging/logging and services such as Kerberos. When the trace environment and NTP change the system clock they should work together, however, both approaches might interfere with each other leading to wrong timestamps. Thus, for HDTrace we rely solely on NTP to handle the time synchronization[2]. In case a source has an offset to other sources, this can be compensated by adjusting the XML header of the trace file.

- **Research friendly environment**: The mentioned aspects foster an environment which is suitable for research in tracing and visualization techniques as well as in simulation and the analysis of parallel file systems. All code in the context of the HDTrace environment is open-source and free of charge to allow researchers to realize their ideas.

4.2.2. On-Disk Format

There are three different types of files, each contains *events* with different properties. *Trace files* store activities of one event source and permit recording nested cause-and-effect chains of the events. A *relation file* permits to record multiple concurrent activities and the chain-and-effect of the events. Those activities can be related over process boundaries and even across nodes. *Statistics files* hold numerical values for an arbitrary number of metrics. A *project file* describes all the trace sources of interest for the analysis by linking all those files together.

In the following an instructional description of the various file types and concepts is demonstrated based on illustrating examples:

- **Trace**: A trace file contains information from a single trace source in plain XML. A trace source could be a process or any other entity which generates activity. An example trace file is given in

[1]An additional minor advantage of this strategy is also that avoiding redundant information saves disk space.
[2]See Appendix A.2 for details and discussion of the accuracy of NTP.

Listing 4.1. In the header the attribute "`timeAdjustment`" defines an offset (in seconds) that applies to all timestamps in the file. Thus, a later manipulation of this attribute shifts the recorded activity in the indented direction. The size of the field is big enough to permit a modification without rewriting the remaining parts of the file – when a clever editor is used. The fields "`processorModelName`" and "`processorSpeedinMHZ`" record information of the processor, currently this information is taken from `/proc/cpuinfo`. With "`processorSpeedinMHZ`" the current speed of the processor is indicated, this is needed for the simulator to approximate the number of cycles per compute step.

Activity is either an event or state that is marked with an XML tag. A *state* has a well defined beginning and end. Events are indicated with the reserved tag "`Event`". In comparison to a state they just have a single time; for instance, receiving of a signal is an event.

When a state invokes other states of interest they are nested, for instance the first nested section of the listing defines the operations within the `MPI_Barrier()` command of MPICH2. For the simulation and some post-processing tools the nesting areas are not of interest; with this annotation they can be stripped easily from the trace file. Attributes of arbitrary semantics can be added to each recorded state or event individually, this is used in the example to record all parameters of the MPI calls. Communicator, datatypes and files are referred to by a numeric ID. The ID and mapping is currently stored in the *project file*.

Activities can also record arbitrary information in the XML body of the tag. This is used for `MPI_Wait()` calls to indicate which asynchronous operation they matched. There could be an arbitrary number of asynchronous operations which are matched by an `MPI_Wait_all()` call. In our listing only one `MPI_Irecv()` call is matched by the wait. Also, the same mechanism records all hints associated with a file. Those hints could be set by the user or an intermediate high-level I/O library.

Listing 4.1: Example trace file for one process

```
<Program timeAdjustment='1293732233' processorSpeedinMHZ='1600' processorModelName='Intel(R)␣
  →Xeon(R)␣CPU␣X5650␣@␣2.67GHz'>
<Init  time='0.137998' end='0.137998'/>
<Nested>
  <Sendrecv size='0' toRank='2' toTag='1' fromRank='2' fromTag='1' cid='1' count='0' sendTid='1'
    →recvTid='1'  time='0.138009' end='0.138030'/>
  <Sendrecv size='0' toRank='1' toTag='1' fromRank='1' fromTag='1' cid='2' count='0' sendTid='1'
    →recvTid='1'  time='0.138032' end='0.142847'/>
  <Send size='1' count='1' tid='1' toRank='2' toTag='2' cid='1'  time='0.142852' end='0.142855'/>
</Nested>
<Barrier cid='0'  time='0.138007' end='0.142859'/>

...
<Irecv fromRank='7' fromTag='1001' cid='18' rid='0'  time='8.844311' end='8.844433'/>
...
<Wait  time='30.896758' end='30.896785'>
  <For rid='1' />
</Wait>
...
<Nested>
  <Allreduce size='4' cid='1' count='1' tid='1'  time='31.505869' end='31.506637'/>
  <Gatherv size='7' root='0' cid='1' count='7' tid='0'  time='31.506850' end='31.506873'/>
  <Bcast size='4' rootRank='0' cid='1' count='1' tid='1'  time='31.506881' end='31.506905'/>
  <Bcast size='20' rootRank='0' cid='1' count='5' tid='1'  time='31.506906' end='31.506923'/>
  <Type_commit tid='2'  time='31.691927' end='31.691965'/>
  <Bcast size='20' rootRank='0' cid='1' count='1' tid='2'  time='31.691968' end='31.692016'/>
  <Type_free tid='2'  time='0.692018' end='0.692023'/>
  <Allreduce size='4' cid='1' count='1' tid='1'  time='31.692025' end='31.692350'/>
</Nested>
<File_open cid='0' name='pvfs2:///pvfs2/visualization.dat' flags='5' fid='0'  time='31.470393'
  →end='0.692391'/>
<File_write fid='0' offset='0' size='4' count='1' tid='1'  time='31.692397' end='31.734785'/>
...
<Finalize  time='268.376843' end='268.377122'/>
</Program>
```

- **Relation**: A relation file permits to store multiple nested and also interleaved activities for one trace source. Each relation can be thought of as a single trace file manufactured for the purpose to handle a set of sequential and nested activities. The number of concurrent activities is not limited. This is important for file servers as the number of connected clients defines how many concurrent states are expected. Existing tools must open another trace stream for each concurrent activity. HDTrace permits to store all information in one file. Furthermore, it allows Sunshot to map the states to timelines automatically – concurrent activities can be mapped to one timeline or to as many as needed.

An example relation file is shown in Listing 4.2. Here only one *relation* is started with the "`rel`" tag and closed with the "`un`" tag. Similar to a trace file states of a relation can be nested and a tag can piggyback arbitrary attributes or XML entities. States are associated with a particular relation by a *token*; indicated with the "`t`" attribute in the XML.

Stored activity can be related to other relation events, this way the causality between multiple activities becomes clear – activity A triggers activity B which triggers C. These information can be forwarded to remote systems to relate events across processes and even systems. In the example the relation Token 0 relates to the activity created by the remote token with ID "west8:25555:5". The latter ID should be unique in the system and consists of the hostname, process ID (also referred to as local token) and a token ID[3].

Interleaved operations are omitted in the example, but the potential interleaving of the XML tags should be obvious. The relation concept has been used, for instance, to store the simulator output and the instrumentation of PVFS. The presented XML snippet shows the execution of the PVFS state machine which handles configuration requests.

Listing 4.2: Example relation file

```
<relation version="1" hostID="west9" localToken="25776" topologyNumber="0"
  →timeAdjustment="1293627993" processorSpeedinMHZ='1600' processorModelName='Intel(R)␣Xeon(R)␣
  →CPU␣X5650␣@␣2.67GHz'>

<rel t="0" time="10.368761" p="west8:25555:5"/>
<s name="pvfs2_get_config_sm" t="0" time="10.368771"/>
<s name="pvfs2_prelude_sm" t="0" time="10.368773"/>
<s name="setup" t="0" time="10.368777"/>
<e t="0" time="10.368779"/>
<s name="prelude_work_sm" t="0" time="10.368781"/>
<s name="req_sched" t="0" time="10.368782"/>
<e t="0" time="10.368784"/>
<s name="getattr_if_needed" t="0" time="10.368786"/>
<e t="0" time="10.368787"/>
<s name="perm_check" t="0" time="10.368788"/>
<e t="0" time="10.368789"/>
<e t="0" time="10.368790"/>
<e t="0" time="10.368791"/>
<s name="init" t="0" time="10.368792"/>
<e t="0" time="10.368793"/>
<s name="final_response_sm" t="0" time="10.368795"/>
<s name="release" t="0" time="10.368796"/>
<e t="0" time="10.368799"/>
<s name="send_resp" t="0" time="10.368800"/>
<e t="0" time="10.368847"/>
<s name="cleanup" t="0" time="10.368848"/>
<e t="0" time="10.368854"/>
<e t="0" time="10.368855"/>
<s name="cleanup" t="0" time="10.368856"/>
<e t="0" time="10.368859"/>
<e t="0" time="10.368860"/>
<un t="0" time="10.368861"/>
...
```

- **Statistics**: A statistics file contains the trend for a group of metrics which semantically belong together. Values of a group are stored together in one record which can be updated regularly. In typical tracing environments such as VampirTrace those statistics are referred to as counters.

[3]Although this scheme does not guarantee the uniqueness of the created ID it suffices for most use cases. An example in which it is not unique any more is the case in which a processes finishes, a new process is spawned dynamically and the OS reuses the PID.

Initially, statistics were designed to store average values over time as obtained, for instance, from the operating system and thus the name was obvious. Now, the statistics have also been used to accurately store the values of a metric, whenever the values of the metric changes new values are written. Hence, the accuracy of the statistics depend on the update approach.

A statistics file consists of an XML header describing the statistics group and a simple structured binary body carrying the timestamps and records. As the content of the group is fixed for a statistic file a binary format was chosen to minimize the overhead for data representation.

Listing 4.3 shows an exemplary XML header. The first line is an ASCII number which represents the length of the XML header. In the topology node the user can encapsulate information about the component on which the statistics have been stored. The *group* name specifies the actual statistics group, here the utilization of the system. Time can be adjusted in the group in the same fashion as in the other file formats, therefore a placeholder timestamp is provided. For each value of the group the name, the type and unit of measure can be specified; for example, in the listing "`CPU_TOTAL`" is a float which is labeled with percent.

The *grouping* attribute is made to associate similar metrics with each other. Identical strings means the metrics are related. For instance, *"MEM_FREE"* and *"MEM_BUFFER"* relate to the resource of main memory. When the statistics are visualized this semantical information can be used, for instance, to normalize across all values of a group. Thus, a user can normalize the network traffic to the maximum value of inbound and outbound speed to quickly spot which channel served more data.

Listing 4.3: Example file header of a statistics file

```
01381
<Statistics>
<TopologyNode>
  <Label value="west7"/>
</TopologyNode>
<Group name="Utilization" timestampDatatype="EPOCH" timeAdjustment="-0000000000.000000000">
  <Value name="CPU_TOTAL" type="FLOAT" unit="%" grouping="CPU" />
  <Value name="MEM_USED" type="INT64" unit="B" grouping="MEM" />
  <Value name="MEM_FREE" type="INT64" unit="B" grouping="MEM" />
  <Value name="MEM_SHARED" type="INT64" unit="B" grouping="MEM" />
  <Value name="MEM_BUFFER" type="INT64" unit="B" grouping="MEM" />
  <Value name="MEM_CACHED" type="INT64" unit="B" grouping="MEM" />
  <Value name="NET_IN_lo" type="INT64" unit="B" grouping="NET" />
  <Value name="NET_OUT_lo" type="INT64" unit="B" grouping="NET" />
  <Value name="NET_IN_eth0" type="INT64" unit="B" grouping="NET" />
  <Value name="NET_OUT_eth0" type="INT64" unit="B" grouping="NET" />
  <Value name="NET_IN_eth1" type="INT64" unit="B" grouping="NET" />
  <Value name="NET_OUT_eth1" type="INT64" unit="B" grouping="NET" />
  <Value name="NET_IN_EXT" type="INT64" unit="B" grouping="NET" />
  <Value name="NET_OUT_EXT" type="INT64" unit="B" grouping="NET" />
  <Value name="NET_IN" type="INT64" unit="B" grouping="NET" />
  <Value name="NET_OUT" type="INT64" unit="B" grouping="NET" />
  <Value name="HDD_READ" type="INT64" unit="B" grouping="HDD" />
  <Value name="HDD_WRITE" type="INT64" unit="B" grouping="HDD" />
</Group>
</Statistics>
... BINARY DATA ...
```

- **Project**: A project file maps multiple event files, statistics or relation files into a hierarchical organized namespace. In HDTrace this namespace is called **topology**. The project file contains also information that is frequently accessed and thus should be accessible without scanning through trace files. For example, the processes participating in each communicator are stored. Therewith, tools know all communication partners of collective operations without scanning through additional trace files. User defined datatypes and accessed files are also registered in the project file. This information is read by the simulator and enables the user to change the data distribution across the file servers for each file.

  Event files belonging to a project share the same prefix and are placed into one directory. When a project is loaded the topologies are scanned for all known file types. Thus, the association of the files to the project can be modified by the user easily by moving files into the right spot.

An example project file is provided in Listing 4.4. First, the accessed files are listed, then information about the topology is maintained. Here, the topology defines three ranks on two disjoint hosts – as illustrated in Figure 4.2. Since more information about the topology concept is provided on the next page, further description is omitted here. Two communicators are defined; the “WORLD” communicator contains all ranks, the other one is not named by the user and contains Rank 0 and Rank 2. Then, the constructed datatypes and basic datatypes are listed with their identifier, in the example the processes just use the basic datatypes `Char`, `Byte` and `Double`. Two statistic groups – energy and utilization, are known in the project. The existence of statistics files is checked when the project file is loaded.

In principle a project file must be created manually because arbitrary sources might generate information that should be visualized together. For example energy metrics might be generated externally and should be visualized together with the parallel application. HDTrace does not impose restriction on the information that might belong together.

However, there is a tool that helps users to automatically derive a project file for MPI applications. Therefore, small file snippets that contain the required information for executing the program are written by the *MPI-Wrapper* and read by the tool. This tool is described in Section 4.3.1.

Listing 4.4: Example project file

```
<?xml version="1.0" encoding="UTF-8"?><Application name="example">
<Description>Just a small description, this is an example for the project file contents
  →</Description>
<FileList>
 <File name="pvfs2:///pvfs2/visualization.dat">
  <InitialSize>4</InitialSize>
  <Distribution implementation="de.hd.pvs.piosim.model.inputOutput.distribution.SimpleStripe">
  <ChunkSize>64K</ChunkSize>
  </Distribution>
 </File>
</FileList>

 <Topology>
  <Level type="Hostname">
   <Level type="Rank">
    <Level type="Thread">
    </Level>
   </Level>
  </Level>

  <Node name="west3">
   <Node name="0">
    <Node name="0" />
   </Node>
  </Node>
  <Node name="west4">
   <Node name="1">
    <Node name="0" />
   </Node>
   <Node name="2">
    <Node name="0" />
   </Node>
  </Node>
 </Topology>

 <CommunicatorList>
  <Communicator name="WORLD">
   <Rank global="1" local="1" cid="0" />
   <Rank global="2" local="2" cid="0" />
   <Rank global="0" local="0" cid="0" />
  </Communicator>
  <Communicator name="">
   <Rank global="2" local="1" cid="1" />
   <Rank global="0" local="0" cid="1" />
  </Communicator>
```

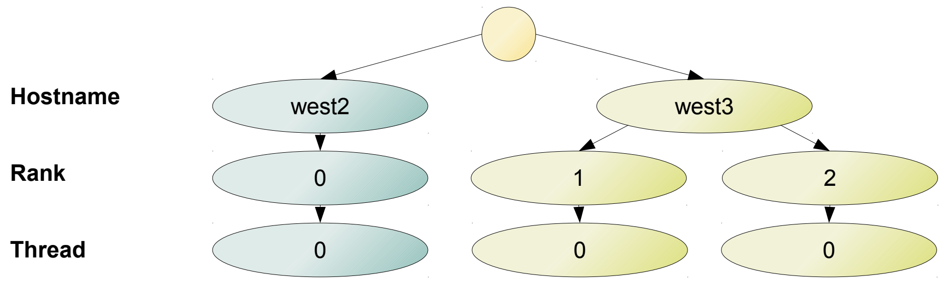


Figure 4.2.: An example MPI topology for two nodes and three processes. The labels for the levels are listed on the left; the instances of the tree nodes are shown on the right.

```
</CommunicatorList>

<Datatypes>
 <Rank name="1" thread="0">
 <NAMED id="0" name="MPI_CHAR"  />
 <NAMED id="1" name="MPI_BYTE"  />
 <NAMED id="2" name="MPI_DOUBLE"  />
 </Rank>
 ...
</Datatypes>

<ExternalStatistics>
       <Energy/>
       <Utilization/>
</ExternalStatistics>
</Application>
```

- **Topology**: Users can define an arbitrary tree for the namespace of the traces, for instance the MPI-Wrapper labels the levels of the tree with "Hostname", "Rank", "Thread".

  Labels on the levels are associated with a semantics, those semantical aspects can be interpreted by viewer plugins to permit content-specific operations[4].

  The concept could also be extended to associate multiple topologies with one project – for instance, to permit an additional physical view which also contains the racks. Since those multiple views could be of interest for the user, the user could pick and switch the displayed topology in the viewer.

  An illustration of the topology concept is provided in Figure 4.2, here the default MPI topology is shown for two nodes and three processes. The hostname identifies the node the process runs on, multiple MPI processes can run on one node, here *Rank 1* and *Rank 2* are placed on *host west3*. For a PVFS server the labels for the topology levels are "Hostname" and "Layer" to identify the internal part of PVFS which generated the events.

  When an event file is created it is attached to one node of the topology tree. The basic semantics requires that the information contained should be applicable to all child elements. For instance, the operating system can provide node local information, such as local network and I/O activity. Those values are caused by the activities of all processes on the node and cannot be assigned to one particular process. Thus, the topology feature permits to indicate shared system resources. Each node of the topology tree can hold at most one trace or relation file, and an arbitrary number of statistics groups. The file names of the event files are the prefix of the project file and a serialization of the topology node names and a suffix depending of the type of the file (.trc, .stat or .rel). A statistics filename contains the group name as well and, thereby, multiple statistics groups can be associated with one topology node.

  More details about the topology concept and statistics are given in [Kre09].

[4]There is no way to fix labels and their semantics, yet. To avoid false sharing of a semantics it seems mandatory to register labels and semantical aspects in the future.

4.3. MPI-Wrapper

The generation of MPI traces is of major importance for this thesis and thus the abstract capabilities of the *MPI-Wrapper* are listed in this section. Paul Müller implemented the first full-featured version from a prototype, see [Mü09] for implementation details beyond the descriptions provided in this section.

The basic MPI-Wrapper intercepts most MPI-2 calls by implementing the PMPI interface and it records MPI calls in trace files attached to a “Hostname”, “Rank”, “Thread” topology. Computation time is not recorded explicitly. A recorded trace file is provided in Listing 4.1. The MPI-Wrapper works for non-threaded MPI implementations and is tested with Open MPI[5] and MPICH2.

When an MPI program is traced the wrapper writes independent trace files for each process, also each process writes relevant information for the project file into one additional file. Those information files guide the `project-description-merger`, a python script which automatically generates a project file for an executed MPI program, this is discussed in section 4.3.1. The MPI-Wrapper is the only HDTrace library which communicates at run-time, but only during initialization.

The overhead of tracing depends on the frequency at which MPI calls are invoked by the application. Each function call adds a little overhead as the call is intercepted and recorded. The induced overhead of the wrapper and HDTrace is comparable to other PMPI interceptors such as VampirTrace, Scalasca or MPE (see Section 7.1). Since the wrapper buffers the latest events in memory to increase responsiveness and performance, cached and unflushed trace information is lost when a deadlock or crash of the application occurs[6].

Features which have been added later to the MPI wrapper include:

- Control of libRUT, the PowerTracer and the PVFS client tracing API. Those event sources are attached to the created topology at the right layer. During initialization, i.e., when `MPI_Init()` is called, the MPI-Wrapper analyzes the run-time environment by communicating with all MPI-Wrapper instances of the application. Then it configures these libraries and ensures that PowerTracer and libRUT are initialized exactly once for each node the program runs on.
- Fortran support: A python script generates wrapper code which intercepts Fortran calls and translates them into calls to the appropriate C pendants. The Fortran calling convention alters function symbols by prefixing one or two underscores or converting the function name into lower, or upper case (depending on compile time options). Parameters must be converted as well, since Fortran passes arguments as references (pointers). Additionally, strings are not null-terminated in Fortran, instead the string length is appended as an additional parameter to the C function signature.
- Information about the source code permits to record source file and line of code for the user activity and within MPI (ROMIO) internal activity. This has been realized by simple C macros which call an extended MPI interface. Thereby, file name and code line can be passed in – the extended interface embeds the desired information into the trace file. Another possible realization would be to use libraries to query debugging information from the object file, for example, by relying on *DWARF*[7].
- An experimental intercepting library for POSIX I/O calls. This library implements wrappers for I/O operations and records all parameters. Upon invocation of an wrapped function the information is traced and the original function is called by using the dynamic linking loader to open the shared library via `dlopen()` and `dlsym()`. Another alternative to this approach, for example, would be to use the GNU linker wrapper functionality [8]. Further, basic wrappers for NetCDF and HDF5 have been generated.

[5]`http://www.open-mpi.org/`

[6]This is the current behavior, a modification of the scheme would be to periodically flush the buffer as well – then the reason for a deadlock could be identified.

[7]The *Debugging With Attributed Record Formats* is a standardized debugging format used in UNIX systems.

[8]See the linker manual pages (`man ld`) for a description.

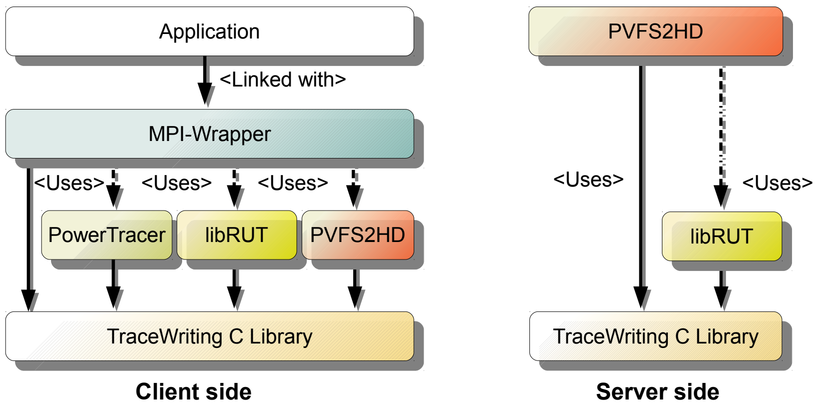


Figure 4.3.: HDtrace libraries involved in the creation of trace files for MPI-IO applications – all these libraries are linked into the application. The server side lists the libraries linked into PVFSHD.

- Hardware counter support by using *Likwid* [THW10]. With Likwid CPU counters for the *instructions per cycle* (IPC), *Flop/s* – partitioned in multiple categories (double precision, single precision, SSE) and memory bandwidth are gathered and recorded for consecutive compute section. Thus, between two recorded activities exactly one compute section is recorded and annotated with these attributes. This would permit *HDReplay* and *PIOsimHD* to simulate the computation characteristics on a system more accurately because the workload is roughly defined by these counters instead of just the runtime. Also, this information could be used to increase accuracy for estimates of the power consumption.
  Unfortunately, current processor counters aggregate memory demand for all hosted cores. Thus, if multiple processes run on one multiprocessor, it will be impossible to determine the memory utilization caused by one process. The extension is not ported for AIX systems, hence, there has not been the need to use the CPU counters in the simulator.

The MPI-Wrapper is integrated into the modified MPICH2HD distribution to automatically instrument all programs compiled with `mpicc` or `mpif90`.

4.3.1. Tracing Workflow

Components involved in the tracing-and-visualization workflow are illustrated in Figure 4.3.

An application is linked with the *MPI-Wrapper* which internally parameterizes the *Resource Utilization Tracing Library*, the PowerTracer and the PVFS client instrumentation. Usage of those libraries is optional, support must be built in during compile time. Several runtime parameters, such as the buffer size for the tracing or the usage of *libRUT*, can be set with environment variables.

The modified parallel file system *PVFS2HD* interfaces with *libRUT* to add statistics from the local node, further server behavior is instrumented directly in *PVFS2HD* to record relevant information. All of those components use the *TraceWriting C Library* to generate independent trace and statistics files. The server trace files are usually created on nodes disjoint to the client nodes.

The workflow to visualize trace files is illustrated in Figure 4.4. When the MPI-Wrapper generates trace files it also puts information about the client topology into so-called *info files*. All data which is relevant in the global environment is stored in these info files. The *project-description-merger* extracts this global information and *generates* a project file which links together all the trace files of the parallel application – application of this tool is indicated with *generate* in the figure. Once, the project file has been generated the info files are not needed any more. With a later modification of the project file arbitrary statistics groups, such as system utilization or power statistics, can be incorporated.

While the MPI-Wrapper generates trace files automatically, a user must activate tracing of PVFS2HD remotely (if desired) – this is due to the fact that the file system processes are spawned as daemons. Hence, they usually exist for a longer time than just for one application run. Currently, in order to visualize traces

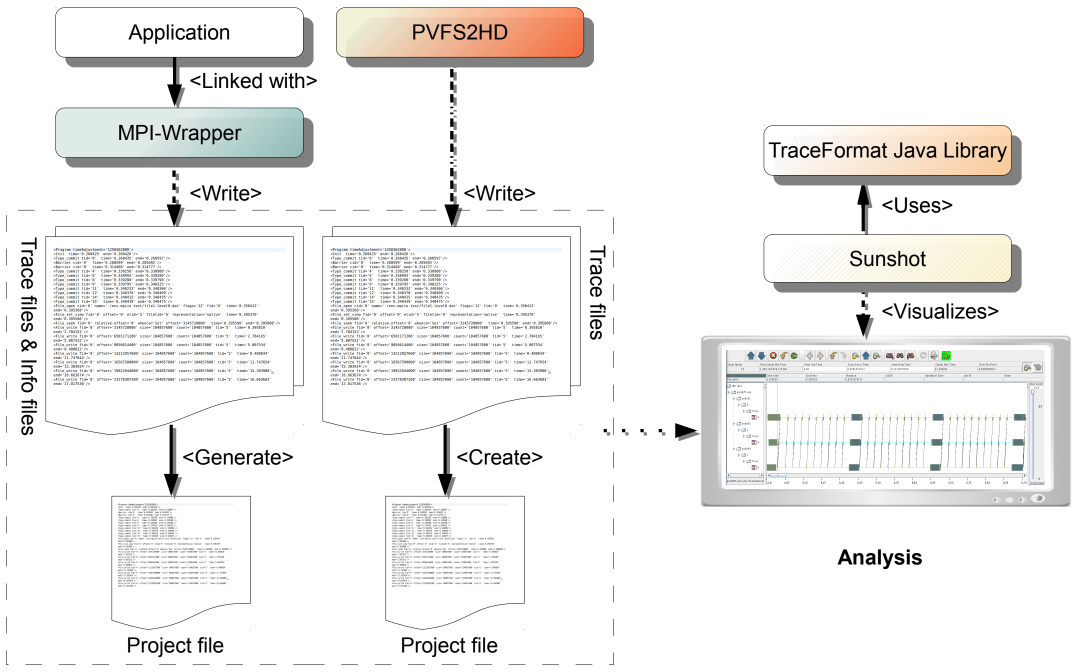


Figure 4.4.: Workflow to create and to visualize trace files.

of the parallel file system, the user has to create a project file for the PVFS traces manually. More details about the tracing of PVFS activities are provided in [Tie09].

Sunshot visualizes the projects, therefore, it uses the TraceFormat Java Library to load the project file and then it loads related trace files for analysis. In Sunshot multiple projects can be loaded and visualized at the same time – therefore, once the inital project is loaded further projects can be added. For instance, with this feature independent client and server traces can be related.

4.4. Sunshot

Sunshot is a Java-Swing application which visualizes trace activity. The code is a major rewrite of Jumpshot (see Page 77) to support HDTrace and includes many new features. By harnessing the Jumpshot code it was possible to reuse and extend the already existing graphical interface without writing a completely new interface.

The focus of this section is to introduce new features which are not not possible with other visualization tools. While Sunshot served as incubator for visualization ideas, all development was driven by the demand of the HDTrace projects.

Since Sunshot visualizes HDTrace, special care is taken to visualize statistics, the topology and I/O relevant aspects. Further, event details (e.g., MPI parameters) and additional features are incorporated into Sunshot. To my knowledge there are no other tools available which permit to visualize I/O datatypes[9], or which offer analysis capabilities for the statistics (counters) as sophisticated as Sunshot.

Extended functionality is presented and discussed based on illustrating examples: First, an overview of the main window types and basic handling is given. Then it is discussed how statistics are visualized and analyzed by the user (in Section 4.4.2). More information about user-interaction in the timeline window is given in Section 4.4.3. The visualization of MPI(-IO) datatypes is presented in Section 4.4.4. An example how to assess MPI communication inside collective calls of MPICH2 is listed in Section 4.4.5. At last, in Section 4.4.6, an example illustrates how MPI-IO and server activity are visualized.

[9]One other tool exists which visualizes memory datatypes: *MPIMAP* [May01] is a visualization tool which eases the creation of MPI datatypes, yet it just unrolls the datatype in a simple way. It is described in Section A.3 on Page 424.

4.4.1. Overview

When the Sunshot program is started the *main window* opens (see Figure 4.5a). This view of the GUI allows a user to load multiple projects at the same time. Therewith, MPI activity and PVFS server activity can be visualized simultaneously.

The *timeline window* visualizes timelines for every topology node and further renders statistics (see Figure 4.7). Activity and statistics at any given time are rendered as boxes, the colors encode the type of the event. White areas in the process activity correspond to activity which is not traced, which is typically some sort of computation.

The *legend window* of Jumpshot is extended to provide additional options for statistics (see Figure 4.5b, the lower part). A *profile window* can be opened (see Figure 4.6), in this view all events on each timeline are aggregated by a given metric; for example, by summing the inclusive duration of the events of a timeline. The interval for which the profile is computed is synchronized with the visible area in the *timeline window*.

A brief overview of the visualizable information is given next.

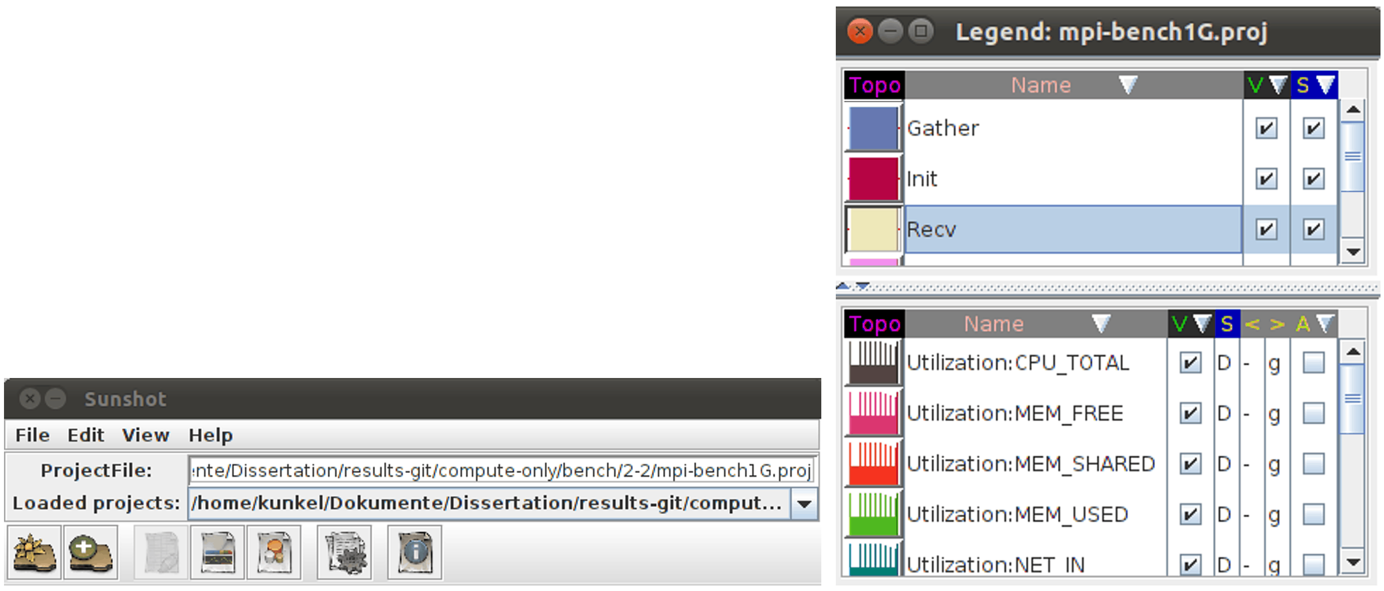
Topology mapping Timeline and profile windows show the topology of the loaded projects at the left. In our example the project *mpi-bench-1G* is loaded, this is a run of an MPI benchmark for two processes – one process (Rank 0) is run on the host *west4*, Rank 1 is executed on *west 3*. To provide a quicker overview states are labeled with their category names if the space inside the box suffices. When Jumpshot was extended to Sunshot the functionally to move timelines around has been removed, still timelines can be hidden by the user. A plugin permits to alter the topology for MPI processes, with this plugin the recorded default topology “Hostname”, “Rank”, “Thread” is altered to a “Rank”, “Thread” topology (application of the plugin can be seen in the topology in Figure 4.8). When the file contains the default topology labels, the plugin is activated – a menu option becomes available when the user right-clicks on the topology.

Statistics and counters Statistic groups of HDTrace are attached to the topology nodes they belong to. If the group consists of multiple values then a folder is created to contain all group metrics, otherwise the metric is drawn directly on the topology. In our example a few statistics about the *utilization* are attached to each host in the timeline window: Network traffic, used memory, aggregated CPU utilization and disk write operations are drawn in individual timelines, the height of each value drawn is proportional to the value at a given time. The handling of statistics is one major improvement over Jumpshot. Several features are supported that simplify the analysis of statistics, histograms and derived statistics, for example. This is discussed in detail in Section 4.4.2.

Relations between events Arrows are not recorded in the event files, instead they are computed on demand in the viewer. Plugins permit to draw arrows for MPI communication and for relations[10]. Figure 4.8 shows arrows between the nested `MPI_Sendrecv()` calls in `MPI_Allreduce()`.

Getting more information about an event When a user right clicks on a trace entry, additional information about the trace entry is presented in the *info box window* (center window in Figure 4.8). For relations and states the XML containing all attached attributes and nested tags are printed with syntax highlighting. In the figure the `MPI_Sendrecv()` transfers 65,536,000 elements (according to the `count` attribute) of the specified datatatype (sendTypeID – `sendTid` and receiveTypeID – `recvTid`) to Rank 1 with Tag 14. Plugins can extend the information shown in the window, one plugin maps the communicator ID to the user defined communicator and prints further information: the name (here “WORLD”), the global rank and the communicator specific local rank. Another plugin automatically decodes the memory datatype for many user-defined datatypes and visualizes them; this feature is presented in Section 4.4.4.

[10]The drawing of communication arrows is an experimental feature.



Sunshot

File Edit View Help

ProjectFile:	nte/Dissertation/results-git/compute-only/bench/2-2/mpi-bench1G.proj
Loaded projects:	/home/kunkel/Dokumente/Dissertation/results-git/comput...

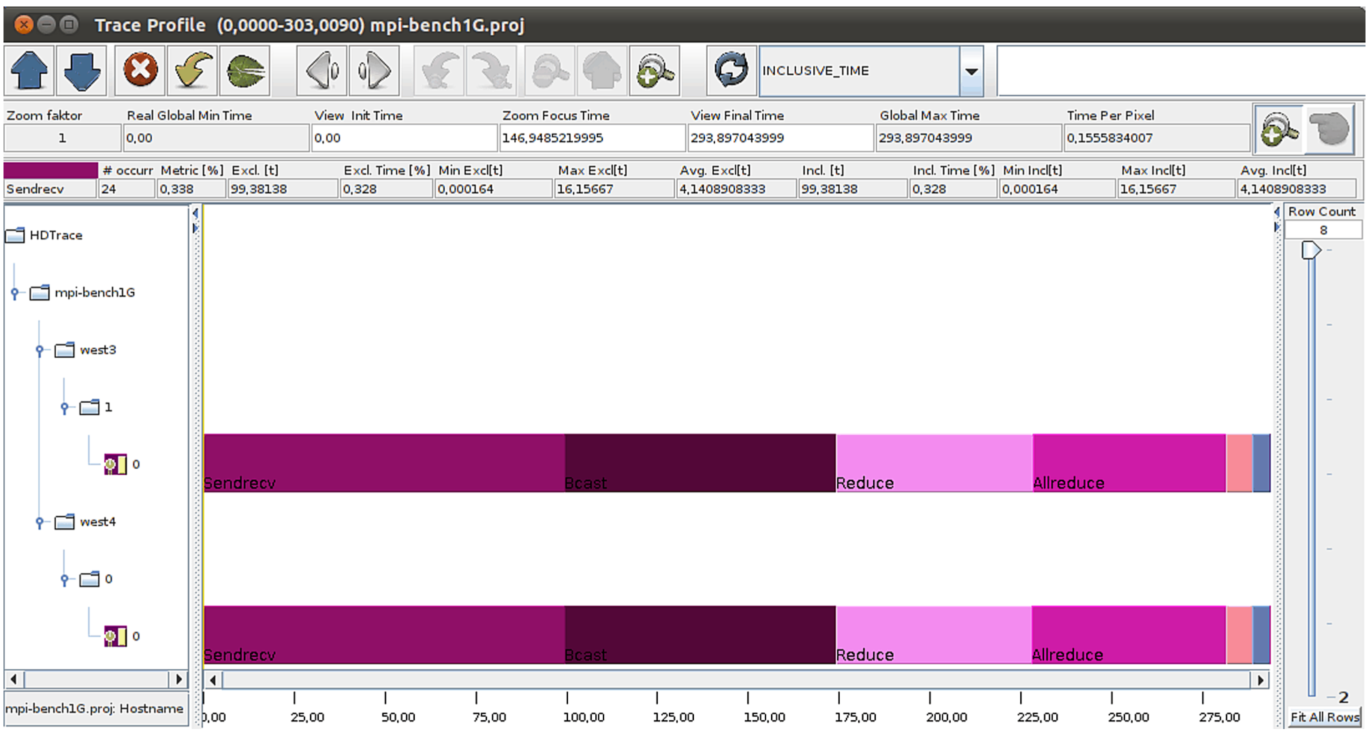
Legend: mpi-bench1G.proj

Topo	Name	V	S
🟦	Gather	✓	✓
🟥	Init	✓	✓
🟨	Recv	✓	✓

Topo	Name	V	S	<	>	A
	Utilization:CPU_TOTAL	✓	D	-	g	☐
	Utilization:MEM_FREE	✓	D	-	g	☐
	Utilization:MEM_SHARED	✓	D	-	g	☐
	Utilization:MEM_USED	✓	D	-	g	☐
	Utilization:NET_IN	✓	D	-	g	☐

(a) Main window (b) Legend window

Figure 4.5.: Sunshot main windows.



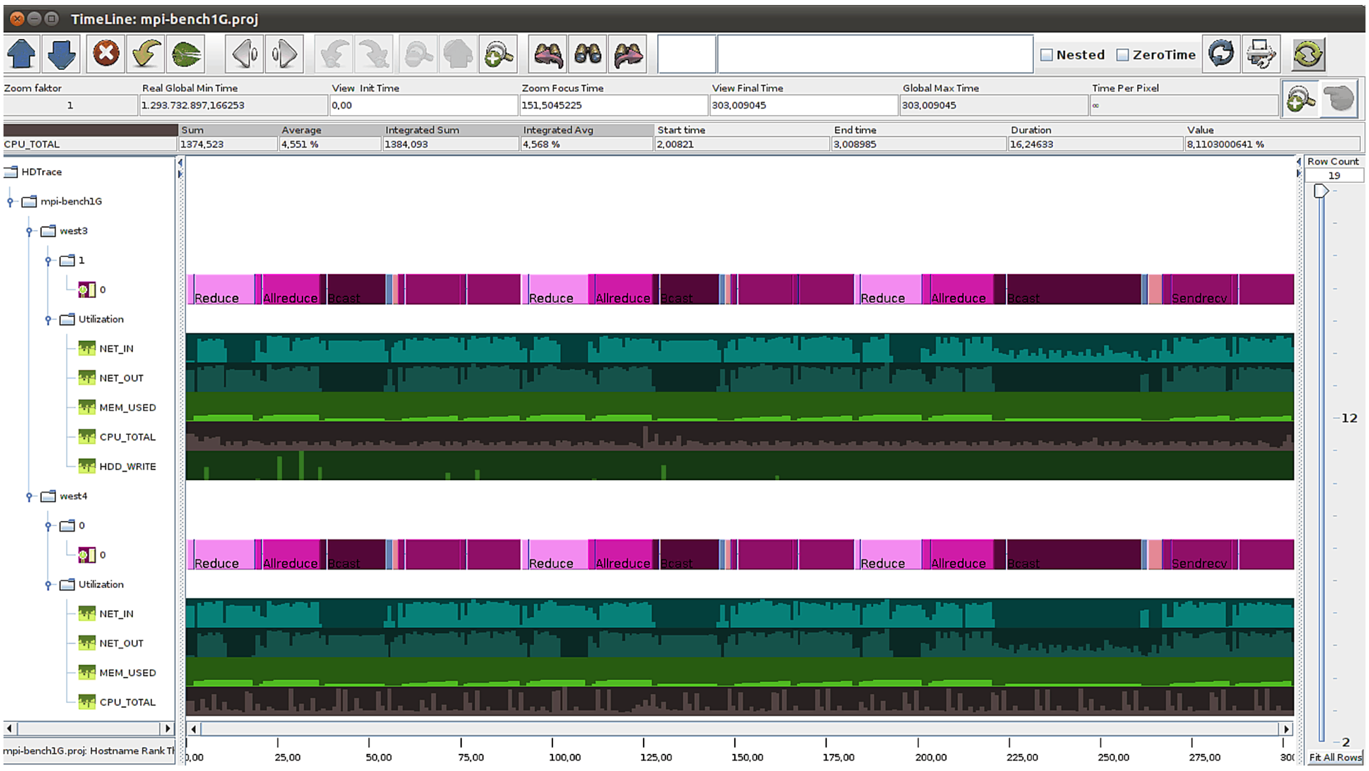
Trace Profile (0,0000-303,0090) mpi-bench1G.proj

INCLUSIVE_TIME

Zoom faktor	Real Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
1	0,00	0,00	146,9485219995	293,897043999	293,897043999	0,1555834007

	# occurr	Metric [%]	Excl. [t]	Excl. Time [%]	Min Excl[t]	Max Excl[t]	Avg. Excl[t]	Incl. [t]	Incl. Time [%]	Min Incl[t]	Max Incl[t]	Avg. Incl[t]
Sendrecv	24	0,338	99,38138	0,328	0,000164	16,15667	4,1408908333	99,38138	0,328	0,000164	16,15667	4,1408908333

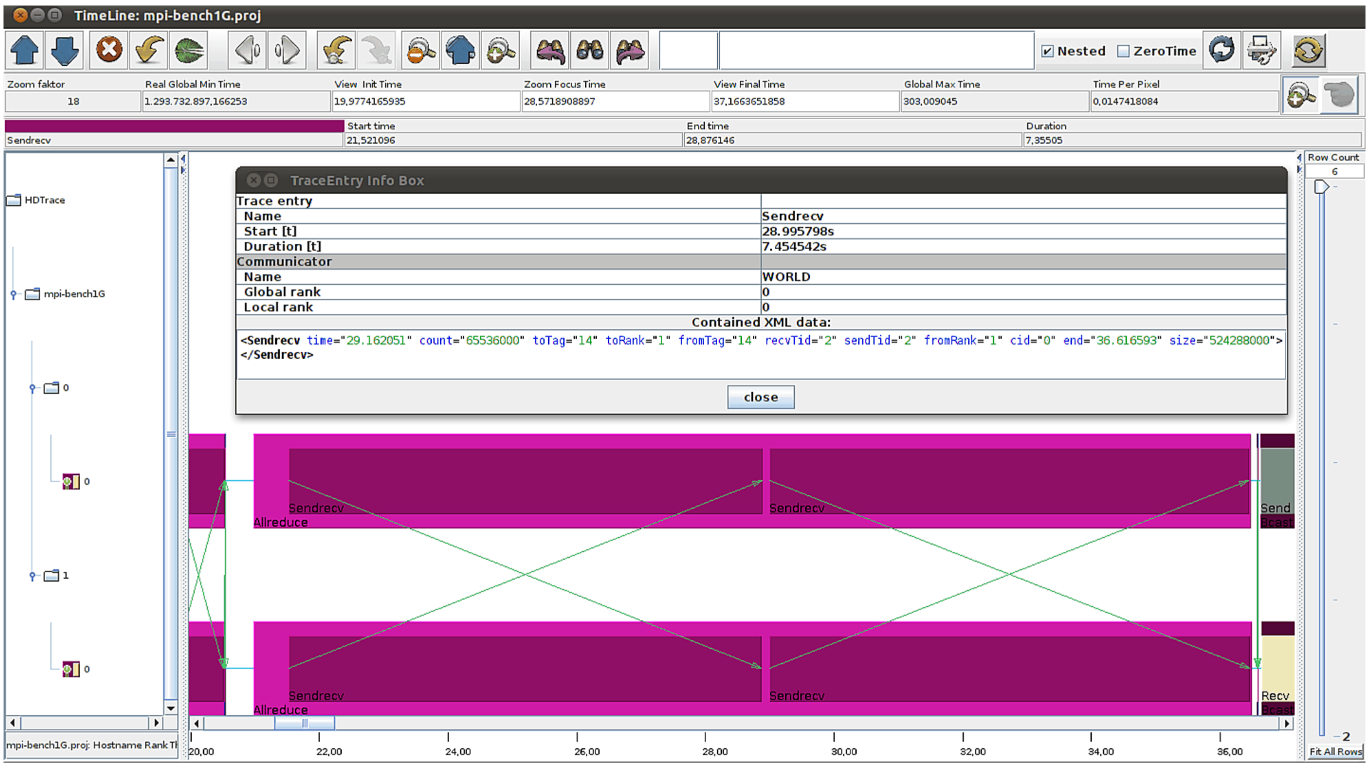
Figure 4.6.: Sunshot profile window showing the aggregated duration for each state.



Zoom faktor	Real Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
1	1.293.732.897,166253	0,00	151,5045225	303,009045	303,009045	∞

	Sum	Average	Integrated Sum	Integrated Avg	Start time	End time	Duration	Value
CPU_TOTAL	1374,523	4,551 %	1384,093	4,568 %	2,00821	3,008985	16,24633	8,1103000641 %

Figure 4.7.: Sunshot timeline window.



Zoom faktor	Real Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
18	1.293.732.897,166253	19,9774165935	28,5718908897	37,1663651858	303,009045	0,0147418084

	Start time	End time	Duration
Sendrecv	21,521096	28,876146	7,35505

TraceEntry Info Box

Trace entry	
Name	Sendrecv
Start [t]	28.995798s
Duration [t]	7.454542s
Communicator	
Name	WORLD
Global rank	0
Local rank	0

Contained XML data:

```
<Sendrecv time="29.162051" count="65536000" toTag="14" toRank="1" fromTag="14" recvTid="2" sendTid="2" fromRank="1" cid="0" end="36.616593" size="524288000">
</Sendrecv>
```

close

Figure 4.8.: Timeline Window with arrows and the *info box window*.

4.4.2. Analyzing Statistics

This section focuses on analyzing energy metrics and most parts (except the *derived statistics*) have been published as a chapter in the *Handbook of Energy-Aware and Green Computing* [MMK$^+$12]. Traces of the *HPC Challenge Benchmark* (HPCC)[11] demonstrate in the following how *Sunshot* visualizes energy metrics and especially statistics. Additionally, the HPCC results are discussed in brief.

A run of the *HPC Challenge* (HPCC) including energy metrics and client activity, is rendered in Figure 4.9. The topology on the left shows that *hpcc* was run on *node06* to *node09*, for example, Process 0 and Process 4 are run on node06 and the energy metrics (I, P, U) belong to node06. From this view a slight fluctuation of the power consumption can be observed between different nodes – during the broadcast operation (purple operation on the right) the power consumption is lower.

Moving the mouse over an state or statistics adds further information in the bar above the timeline canvas, this can be seen in Figure 4.10. In this figure only the timelines for energy metrics and some metrics for node local activity of node06 are shown. Here, the bar prints the yellow color and the text "Itrms" underneath, the average current over time on node06 is 0.838 Ampere and the value under the mouse pointer is about 0.92 Ampere.

The performance statistics offered by libRUT provide a hint how the power consumption is related to node behavior. On this system most energy is spent in the CPU and thus the power consumption is mainly a linear function – idle power plus the aggregated CPU utilization ("CPU_TOTAL") times a constant.

Scaling of statistics Since the height of each bar is scaled linearly between 0 and the maximum value of a timeline, the fluctuation cannot be seen easily in this figure. Sunshot supports adjusting the scale for metrics. For example, the values can be scaled logarithmically, or across multiple timelines of the same category and of the same grouping (e.g., one can scale to the maximum network activity, see Page 193). In Figure 4.11 the energy metric timelines are scaled to the minimum and maximum value of all nodes, therefore the minimal fluctuation between 229 V and 230.4 V of the voltage becomes visible as well.

During the first phase of the benchmark the observed power utilization between the nodes varies. However, the supply voltage stays the same for all nodes, this is expected because all nodes are connected to the same power distribution unit.

Overlapping statistics The resolution of the timeline is limited by by the monitor resolution. Therefore, if the sampling frequency of the value is higher than what can be drawn, then drawn pixel represents the average value for the timespan covering the particular pixel. Further, a shifted color scheme shows the minimum and the maximum observation during the timespan. This behavior is visualized in Figure 4.12a. Figure 4.12b is an excerpt of a zoomed timeline from Figure 4.12a that represents the values as they were observed on the measuring device.

Drawing histograms The values of a given statistic metric can be visualized in histograms with a configurable number of bins showing the distribution of values for a given timeline. To draw a histogram a user right-clicks on the statistics timeline in the topology viewer and selects the histogram. Multiple histogram windows can be opened at the same time. In the histogram the average value is drawn and the standard deviations for one, two and three σ.

Power and voltage of the first node are presented in Figure 4.13, the power fluctuates between 135 W and 220 W. Three peaks can be observed (at 138 W, 175 W and 208 W), this is due to the CPU activity, and more specifically the number of CPUs utilized. During the program run, there are phases in which both

[11] The *HPC Challenge Benchmark* consists of 7 applications: the High Performance LINPACK, DGEMM, Stream, PTRANS, RandomAccess and FFT [LDK$^+$05]. It can be obtained from: `http://icl.cs.utk.edu/hpcc/`.

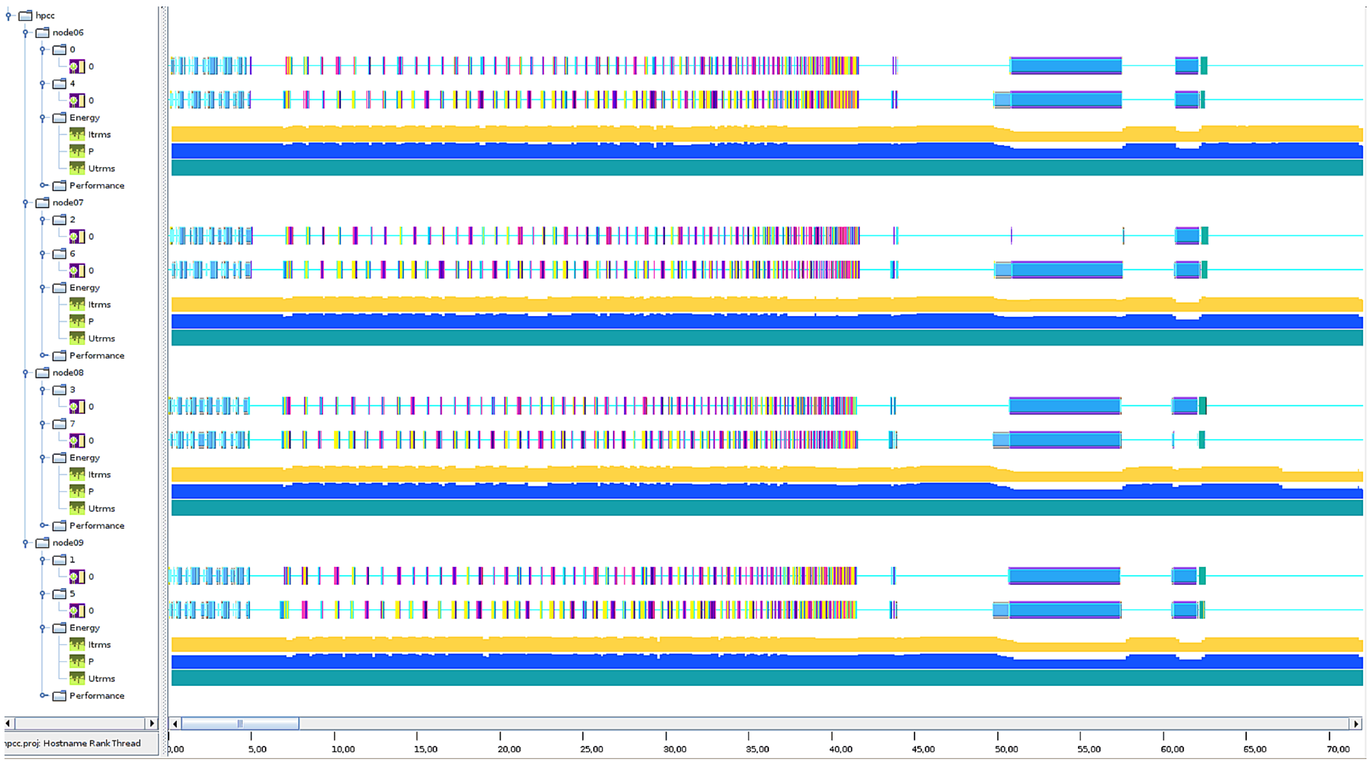


Figure 4.9.: Timelines of the first phase of the HPCC-run including energy metrics and client side communication activity.

CPUs are utilized, sometimes one CPU is completely utilized while the other process waits for input from another process, or both CPUs are idle because both wait for data from processes hosted on other nodes.

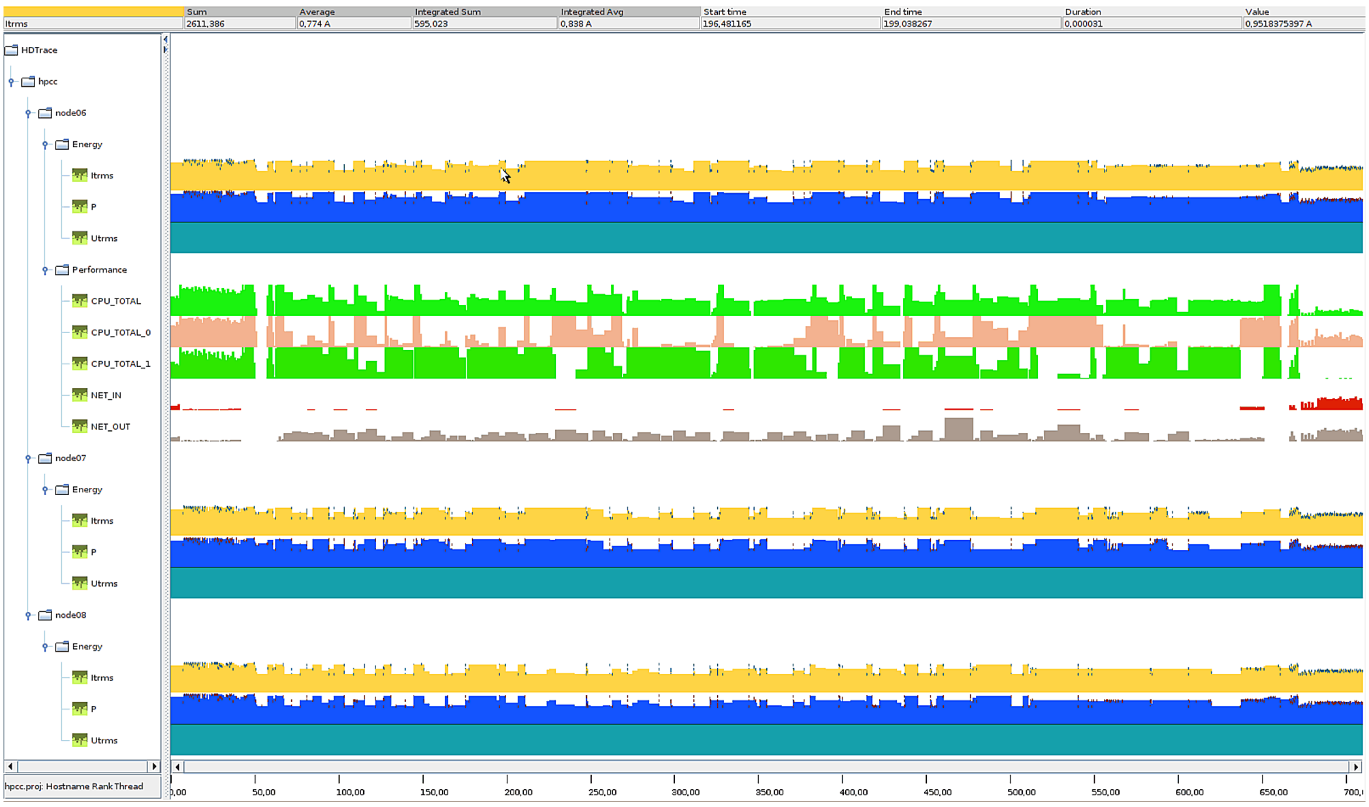
Similar to the profile window the histogram windows adjusts according to the area visualized in the timeline window. When a user zooms in or moves the clipping area, all other displays and histograms update accordingly, presenting content relevant only for the visible area.

Creating derived statistics The available statistics information can be used to derive new statistics timelines. Often a user might want to grasp the program state quickly, looking at a complex statistics can help to achieve this goal. To create a new statistics timeline a user can right-click on a topology label, provide a name for the new metrics and an equation how to compute it. The values of the new timeline are computed based on the user-supplied function and the values of existing timelines. In the previous example (in which the visualization of energy metrics is described), the power consumption could easily be derived by multiplying the voltage (V) with the current (I). Therefore, with derived statistics redundant information in the statistics file could be spared.

To give another example, Figure 4.14 extends Figure 4.7 by introducing two new metrics, the Net_Total aggregates the two node local network traffic statistics, that is, Net_Total = NET_IN + NET_OUT. Those node local metrics are aggregated by *NET_TOTAL* across all hosts and thus the aggregated network traffic of all nodes is visible at a glance.

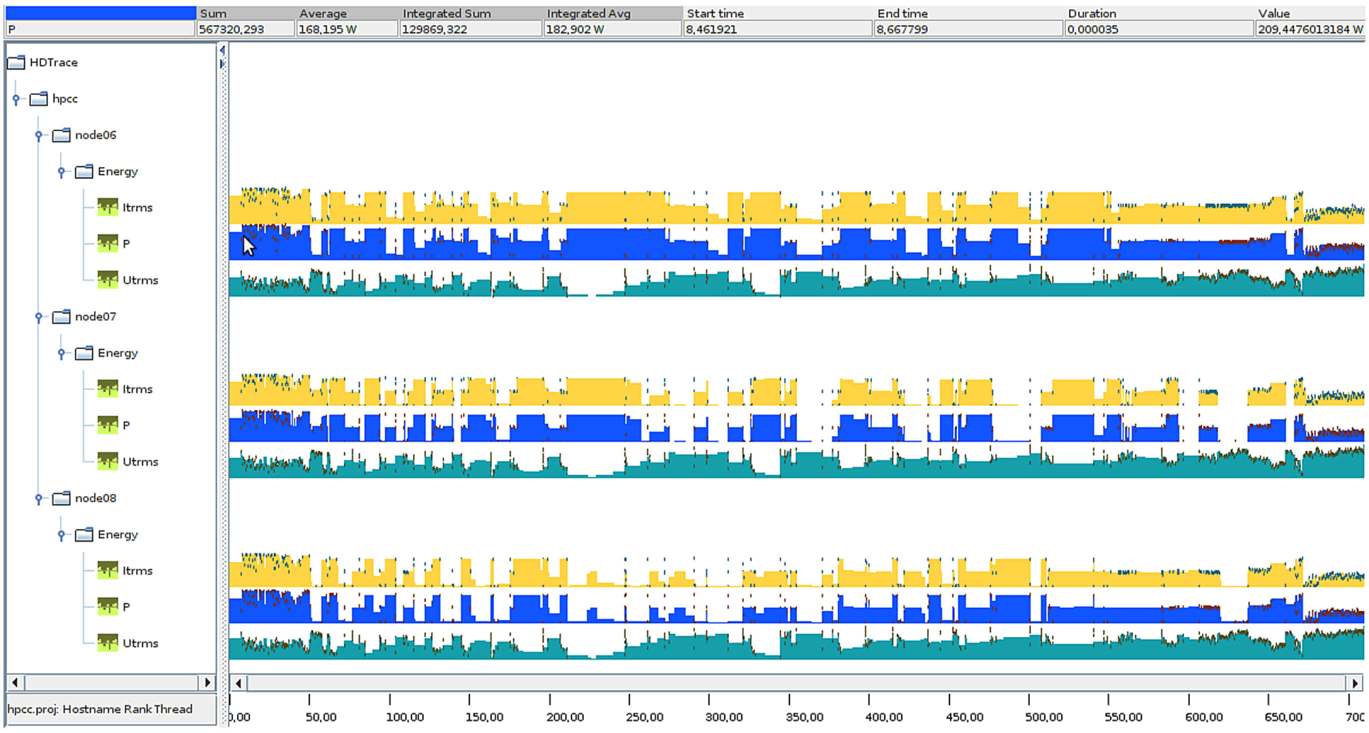
Once created, the user supplied function can be altered by the user at will, whenever it is modified the values are recomputed based on the specified function and the values of required statistics.

Specification of the function includes names of statistics, floating point constants and operators for addition, multiplication, subtraction, division, minimum and maximum (+∗−/,ˆ). The unary operators "+", "∗", minimum (",") and maximum ("ˆ") are also supported as functions. These functions work as reduction operations across multiple timelines. Metrics to use in those timelines are specified in parentheses after the operation. For instance to compute the aggregated value for the network traffic, the function of the



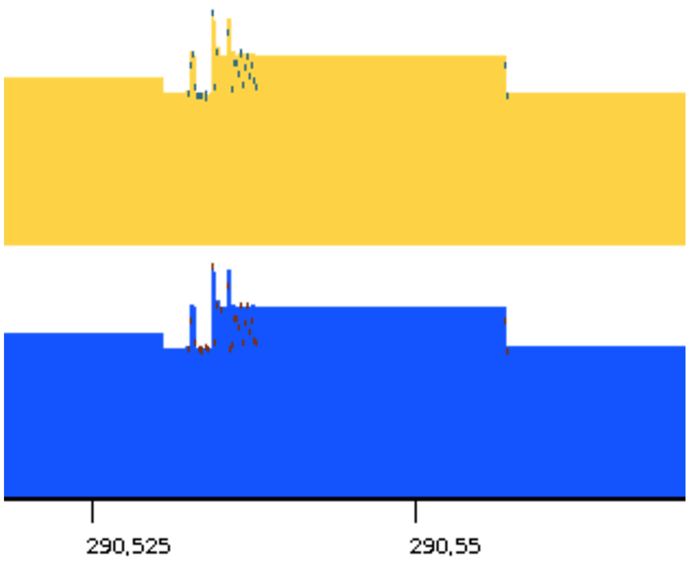
	Sum	Average	Integrated Sum	Integrated Avg	Start time	End time	Duration	Value
Itrms	2611,386	0,774 A	595,023	0,838 A	196,481165	199,038267	0,000031	0,9518375397 A

Figure 4.10.: Energy metric timelines of the HPCC-run.

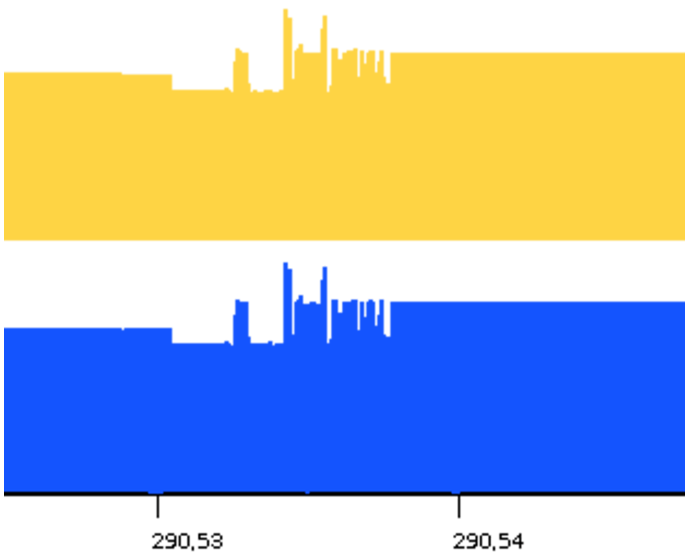


	Sum	Average	Integrated Sum	Integrated Avg	Start time	End time	Duration	Value
P	567320,293	168,195 W	129869,322	182,902 W	8,461921	8,667799	0,000035	209,4476013184 W

Figure 4.11.: Energy metric timelines of the HPCC-run – all timelines are scaled with the global minimum and maximum for each of the metrics (I, P, U).

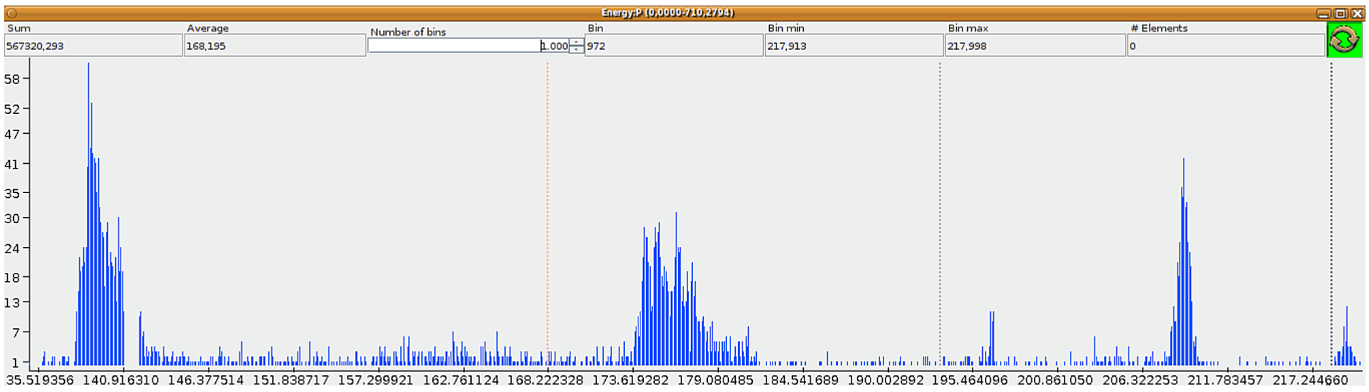


(a) Overview



(b) Zoomed

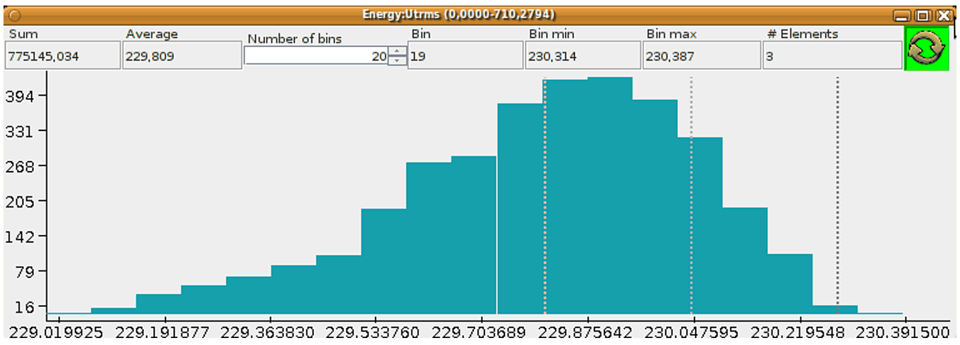
Figure 4.12.: Excerpt of the first process timelines of I and P to demonstrating how minimum, average and maximum are visualized in the timelines.



Energy:P (0,0000-710,2794)

Sum	Average	Number of bins	Bin	Bin min	Bin max	# Elements
567320,293	168,195	1.000	972	217,913	217,998	0

(a) Power

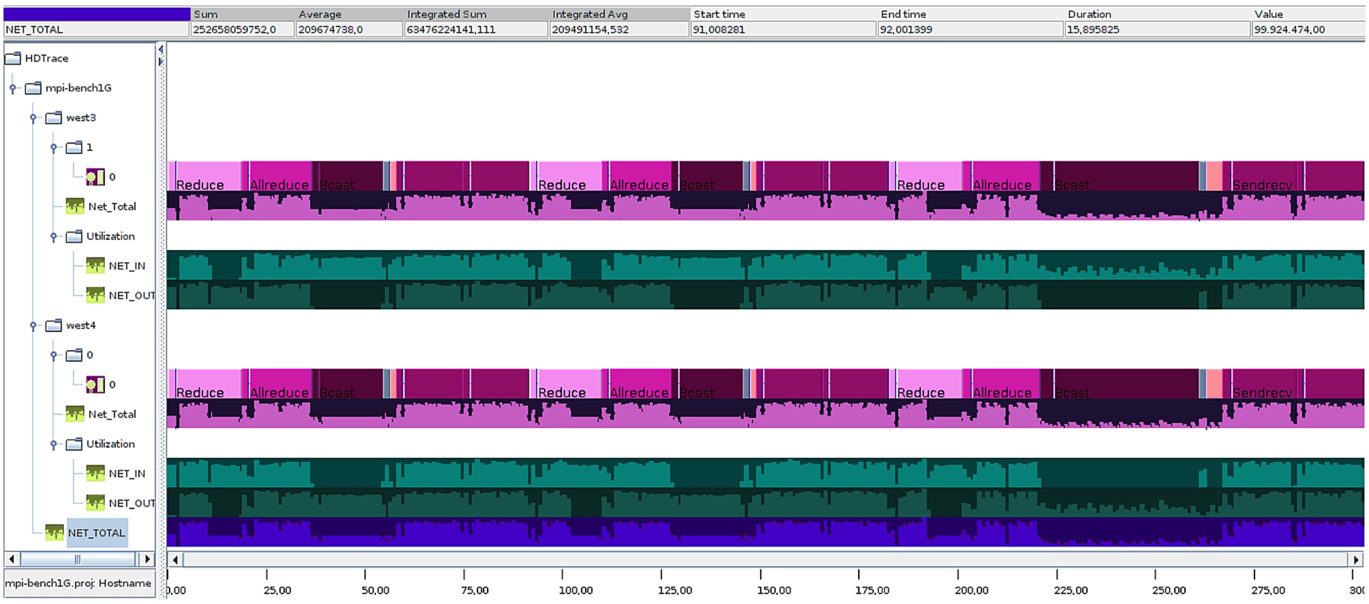


Energy:Utrms (0,0000-710,2794)

Sum	Average	Number of bins	Bin	Bin min	Bin max	# Elements
775145,034	229,809	20	19	230,314	230,387	3

(b) Voltage

Figure 4.13.: Energy histograms for a node. The pink dashed line is the mean value, gray lines indicate standard deviation for 1,2 and 3 σ.



	Sum	Average	Integrated Sum	Integrated Avg	Start time	End time	Duration	Value
NET_TOTAL	252658059752,0	209674738,0	63476224141,111	209491154,532	91,008281	92,001399	15,895825	99.924.474,00

Figure 4.14.: User derived statistics are created in Sunshot by aggregating network traffic of the nodes. This figure extends Figure 4.7.

NET_TOTAL metric was specified as $+(Net\_Total)$.

Required statistics are taken from the current topology node and all visible child nodes. Thus, on the one hand, a hierarchy of new statistics can be created in the topology, and, on the other hand, derived metrics can be introduced on the same topology node[12].

Modifications of the topology by the user, that is caused by expanding, collapsing or removing of topology nodes, are not automatically propagated to derived metrics. Instead users can request to recompute the user defined timeline by right-clicking on its topology node.

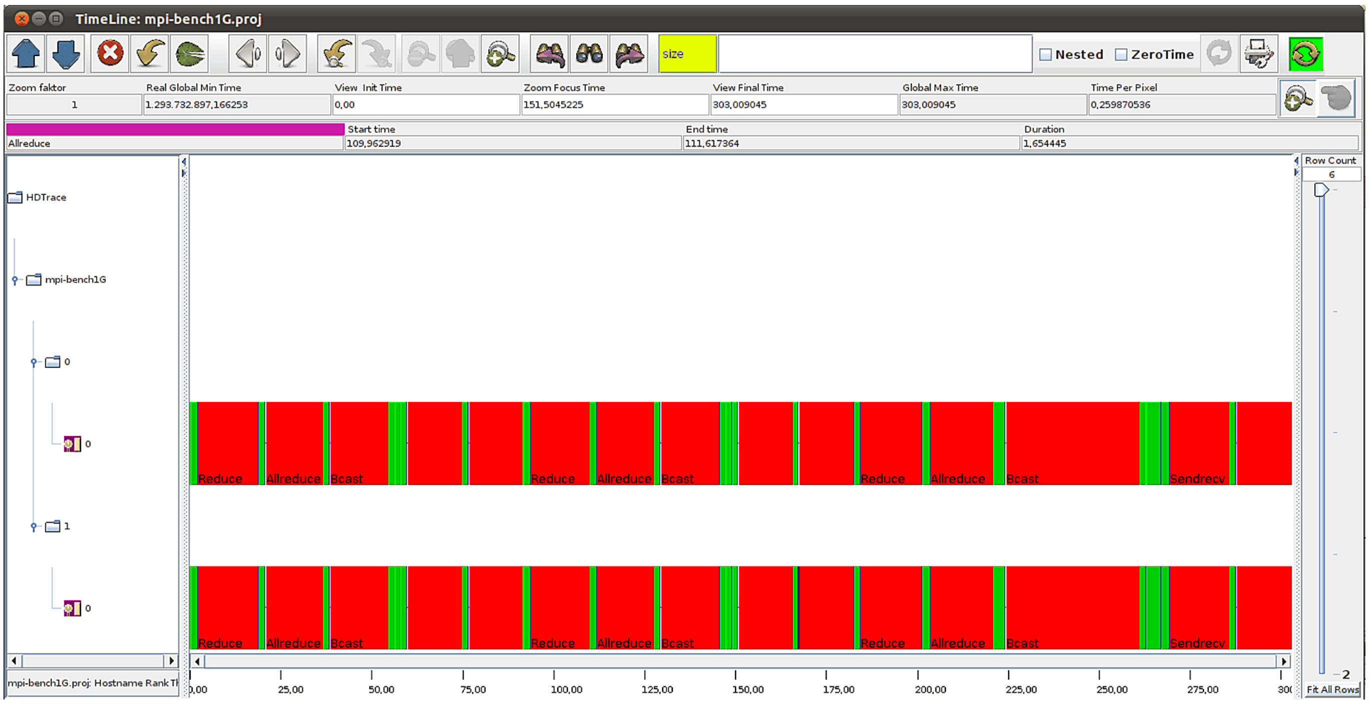
More sophisticated derived metrics are possible, as the function can consist of arbitrary mathematical expressions, e.g., $\hat{}\,(\text{NET\_IN} + \text{NET\_OUT} * 2.0)$ will compute for each child topology the weighted inner term $(\text{NET\_IN} + \text{NET\_OUT} \cdot 2.0)$, then the statistics is filled with the maximum computed value. Another example to assess energy is to divide the consumed power by the CPU utilization (plus a small value to avoid division by zero). Such a metric would indicate the energy consumption per work. Together with recorded CPU counters and utilization models for network and I/O subsystem, energy consumption can be estimated. Thus, model and measurement can be directly compared in Sunshot.

4.4.3. Highlighting Relevant Information

The huge amount of information presented in the timeline and profile view can be overwhelming for the user. Features that ease this task is color-coding of interesting events and user-defined filter mechanisms.

Color-coding Heat-maps are a straight forward way to visualize interesting values in a matrix; the matrix cells are colored depending on the value of the cell – a high value is usually colored in red while a low value is colored in green. This concept is stretched to timelines in Sunshot. Activity in the timeline window can be automatically colored based on a user-selected attribute – the color scheme can even be selected using an equation with multiple input attributes. A visualization based on the "size" attribute is shown in

[12]As data points of the new metric depend on a set of statistics (here NET_IN and NET_OUT), computation of the values for the new statistic timeline is performed as follows: starting from the initial time all required metrics are scanned, whenever one of the statistics values changes its value, then the derived value for the user defined metric is computed according to the function description and the current values. One consequence of this scheme is that the number of statistic entries of derived metrics is proportional to the number of changes in all required statistics.


TimeLine: mpi-bench1G.proj

size

Nested ZeroTime

Zoom faktor: 1
Real Global Min Time: 1.293.732.897,166253
View Init Time: 0,00
Zoom Focus Time: 151,5045225
View Final Time: 303,009045
Global Max Time: 303,009045
Time Per Pixel: 0,259870536

Allreduce
Start time: 109,962919
End time: 111,617364
Duration: 1,654445

Row Count: 6

mpi-bench1G.proj: Hostname Rank Tl

Figure 4.15.: Timeline window with an applied color-coding based on the size of the communicated data.

Figure 4.15 (the run from Figure 4.7 is adjusted accordingly). Thus, the figure colors collective calls and point-to-point messages based on the message size. With the more sophisticated equation of size/.duration, the throughput of all activities could be assessed directly.

All states which contain the attributes requested in the equation specified by the user are processed in two phases: First all entries are scanned and the resulting values are computed to determine the global minimum and maximum. Then, in the second phase, the states are colored depending on the value of the derived metrics; the gradient is between green, black and red, green indicates low values and red high values, black values correspond to average values. Since the example calls mainly communicate two amounts of data – a very small amount and a large amount, only the green and red color is visible.

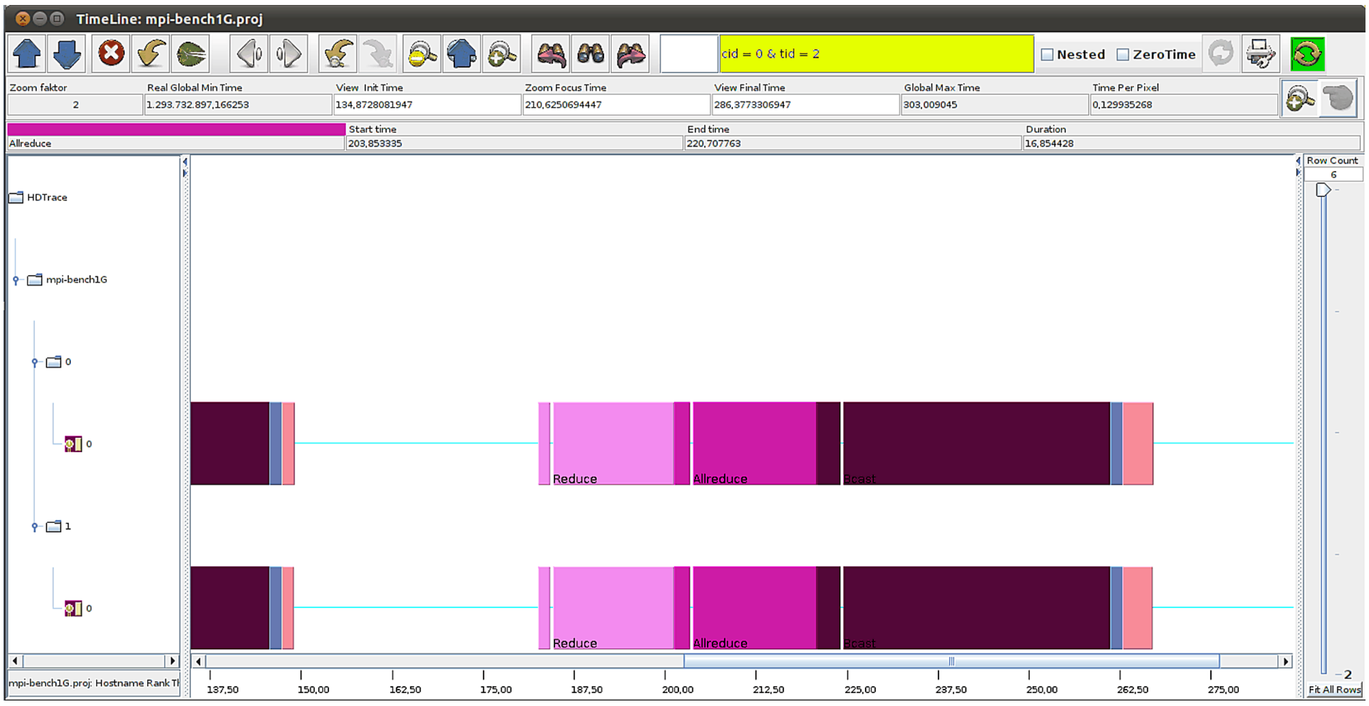
As the altered colors do not encode the category of the state any more, there are other ways to identify the category, e.g., by moving the mouse over a state. This reveals the real type in the information toolbar (here an `MPI_Allreduce()` is selected).

Color-codings (heat maps) are specified in the left text area in the toolbar of the timeline window, in the figure the area is colored in lime. This feature can be combined with a filter function which is discussed next.

Filter functions Timeline and profile window contain a text box in which a user-defined expression can be entered, states and statistics are filtered based on the entered expression. Users can specify expressions that use attribute names or statistic names. The computed values can be compared with floating point values; complex expressions can be constructed by using the boolean operators AND (&) and OR (|), and by nesting expressions with brackets. All states and statistics which do not match the specified condition are hidden, as are states which do not contain all required attributes.

Two reserved keywords exists that can be referred to as regular attributes: With `.category` the name of the category can be filtered quickly by specifying a regular expression, for example, the expression $.category = Send.*$ shows all categories which are prefixed by "Send". The actual duration of an activity is comparable by using the `.duration` keyword.

Figure 4.16 shows an applied filter to our example from the color-coding. The filter requires that an event uses the communicator with ID 0, and that the memory datatype is of Type 2. Thus, a user can quickly


TimeLine: mpi-bench1G.proj

cid = 0 & tid = 2

Zoom faktor	Real Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
2	1.293.732.897,166253	134,8728081947	210,6250694447	286,3773306947	303,009045	0,129935268

	Start time	End time	Duration
Allreduce	203,853335	220,707763	16,854428

Row Count 6

Figure 4.16.: Timeline with filtered communicator and type identifier.

see operations performed by one communicator, or the events involving a particular memory datatype[13]. Users can query the required information by looking at the trace entry info box.

Another use-case is to identify small message exchanges that takes longer than one second to execute. This indicates either an early sender in point-to-point messages or an early start of a collective operation, or it might be caused by a hardware bottleneck. Application of such a filter is illustrated in Figure 4.17.

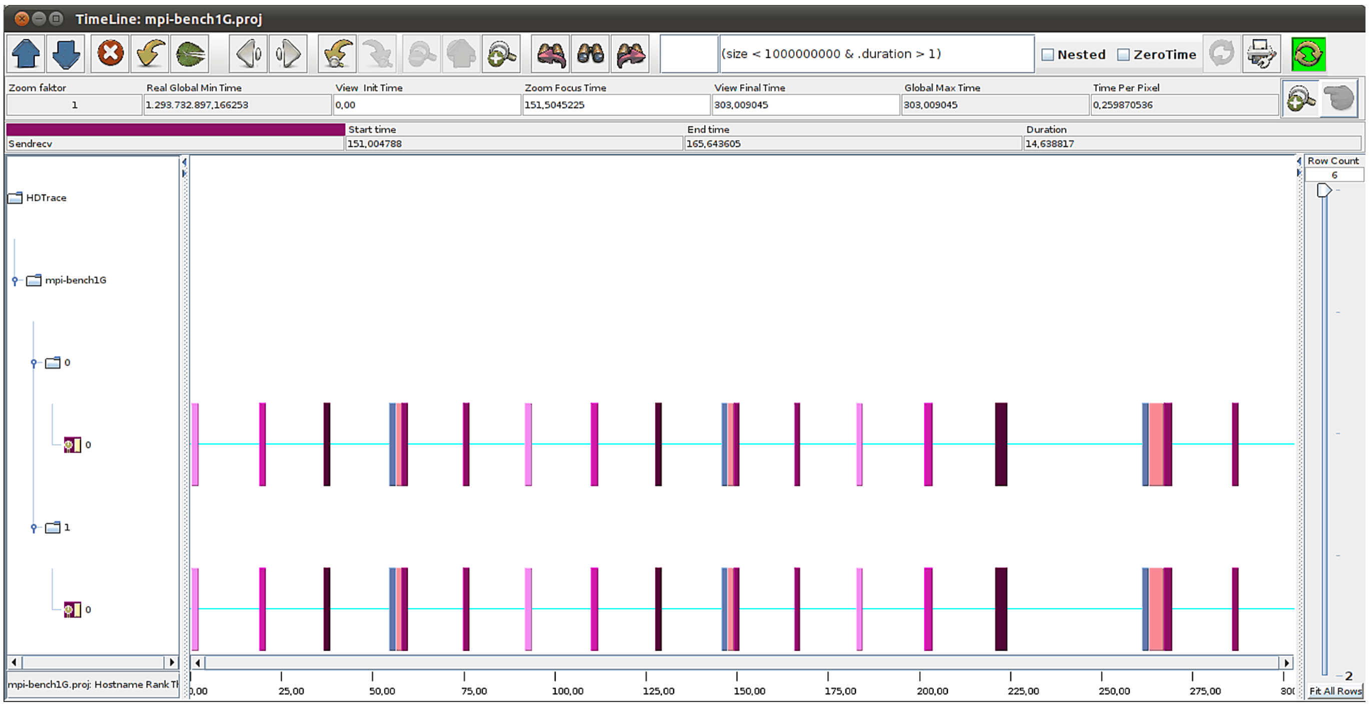
4.4.4. Visualizing of (I/O) Datatypes

Sunshot has the capability to visualize user-defined datatypes of MPI for memory and I/O access in a simple hierarchical view[14]. Many nested datatypes can be expanded and presented in a compact form. Also, file regions accessed in non-contiguous I/O calls can be visualized in terms of the original datatype. The introduced visualization assists developers and users during analysis to understand the spatial I/O access patterns of their application. Also, it reveals the memory regions accessed by a derived datatype.

Sunshot provides classes to handle the MPI datatypes: named[15], contiguous, vector and structure are fully supported. MPI datatypes can be constructed by referring to already defined datatypes. The visualization of Sunshot is restricted to the supported datatypes and to a subset of the memory datatypes: In the specification of memory datatypes it is permitted to refer to data with a negative offset, whereas datatypes for file I/O must refer to ascending positions in a file. For simplicity the visualization engine in Sunshot concentrates on the latter case because this eases the visualization, and, also, datatypes with a negative offset can be typically converted to a datatype with increasing offsets.

When a user selects a particular event to see details in the *info box* a plugin decodes information and shows information about the datatype used in this call. Memory datatypes are provided for all communication routines. If MPI operations access data of a file which has a view set, then the spatial access pattern of the

[13]In fact several calls such as collective calls might use multiple memory datatypes, one for send buffer and one for receive buffer – the correct attribute name must be specified.
[14]Work presented under this section has been published in [KL12].
[15]Those are the primitive datatypes such as an `integer`.


TimeLine: mpi-bench1G.proj

(size < 1000000000 & .duration > 1)

☐ Nested ☐ ZeroTime

Zoom faktor	Real Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
1	1.293.732.897,166253	0,00	151,5045225	303,009045	303,009045	0,259870536

Sendrecv	Start time	End time	Duration
	151,004788	165,643605	14,638817

Row Count 6

Figure 4.17.: Timeline with an filter to show messages which are below a given size but take longer than 1 s to be processed.

non-contiguous I/O operations will be visualized, too. First, the visualization is described for the memory datatypes and then for the non-contiguous I/O.

Memory Datatypes For visualization of memory datatypes a single datatype is unrolled in the viewer. Eventually nested datatypes are visualized in the same fashion and their usage is linked together by a directed acyclic graph. This corresponds to the way MPI datatypes are created.

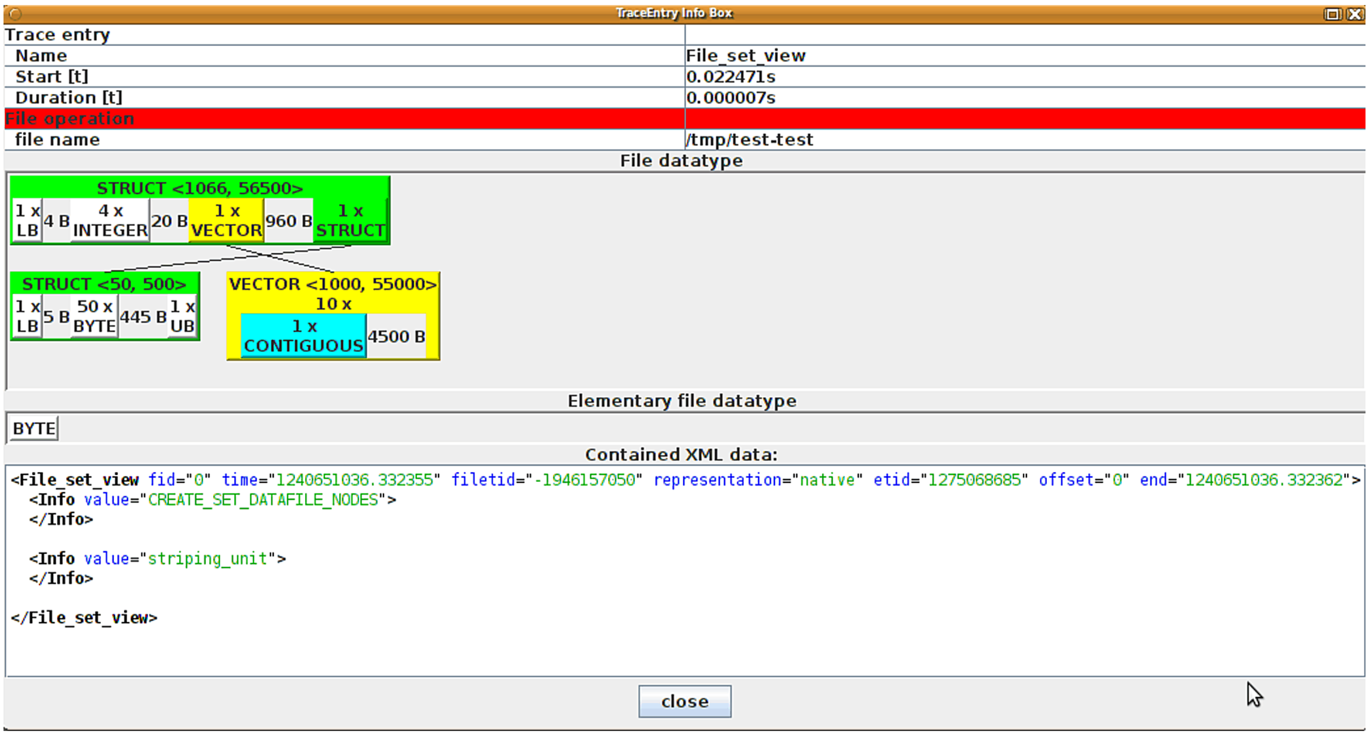
In Figure 4.18 an example screenshot for an `MPI_File_set_view()` operation is provided. The user can see the file the call refers to (here */tmp/test-test*), the file datatype and the elementary datatype. Furthermore, all parameters including `MPI_Info` arguments are listed in the XML information. For each datatype, the actual *size* and *extent* are drawn next to the type name in angle brackets (<size, extent>). Also, the compartments and holes – including their sizes, are given. Each kind of datatype is colored with an individual color, for example, structures are colored in green, vectors in yellow and holes in gray.

The displayed datatype represents a structure which has a hole of 4 bytes at the beginning, then 4 integers, a gap of 20 bytes, a nested vector datatype, another gap of 960 bytes and, finally, one structure. The user can dig into the nested datatype by double-clicking a datatype. Then the datatype expands into its elements; here the structure and the vector have already been expanded. Upon each expansion, the acyclic graph is redrawn: The initial (memory) datatype is drawn on the first row, nested datatypes are drawn in rows depending on their dependency. Lines show the usage of nested datatypes.

The fully expanded datatype is visualized in Figure 4.20. Here, it can be seen that both the contiguous datatype and the initial structure, refer to the structure shown in the bottom row.

Non-contiguous I/O In MPI-IO a user can position the file pointer with respect to the elementary file datatype, holes are skipped directly[16]. Sunshot unrolls the datatype to show which parts of the file are actually accessed by the file I/O. Unrolling of repeated datatypes is indicated by multiplying the unrolled region. As the access size might span multiple repeats of the file datatypes and the offset might start in

[16]Refer to Page 47 for more information about file datatypes and the MPI-IO semantics.



Trace entry	
Name	File_set_view
Start [t]	0.022471s
Duration [t]	0.000007s
File operation	
file name	/tmp/test-test

File datatype

Elementary file datatype

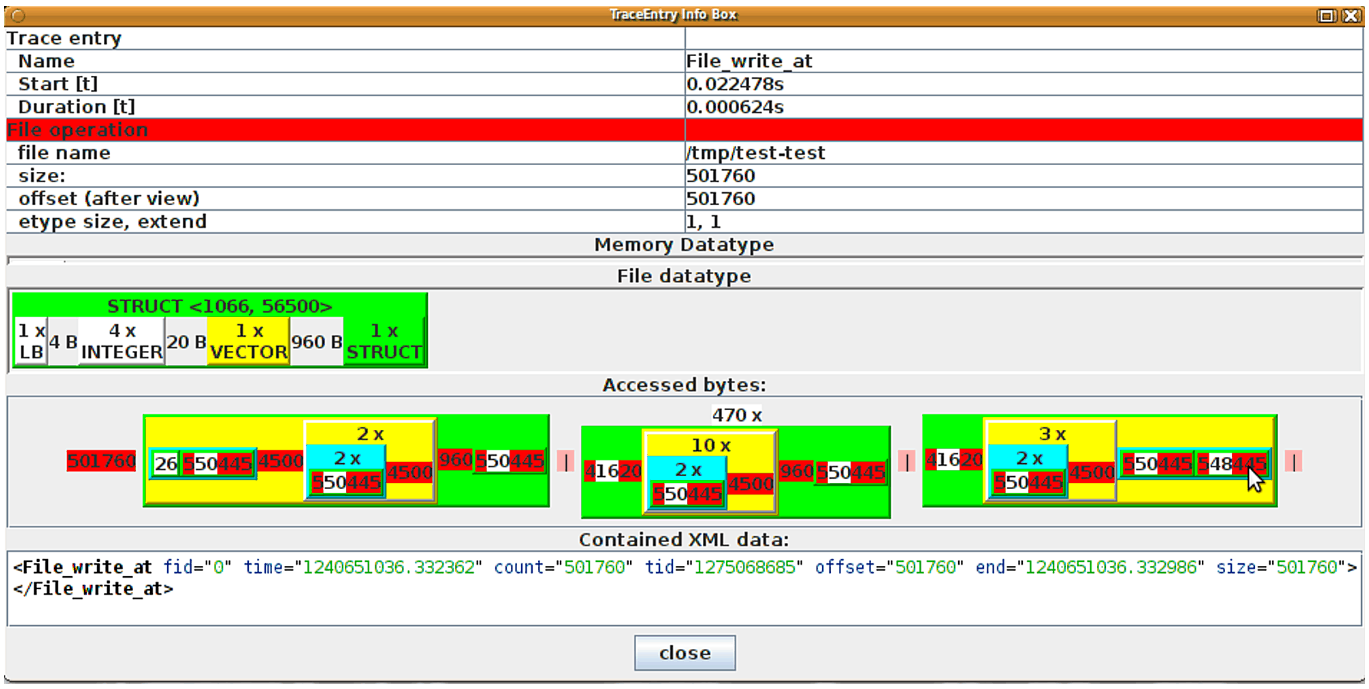
Contained XML data:

```
<File_set_view fid="0" time="1240651036.332355" filetid="-1946157050" representation="native" etid="1275068685" offset="0" end="1240651036.332362">
  <Info value="CREATE_SET_DATAFILE_NODES">
  </Info>

  <Info value="striping_unit">
  </Info>

</File_set_view>
```

Figure 4.18.: Sunshot info box for a `MPI_file_set_view()` function invocation with a partly expanded file datatype.



Trace entry	
Name	File_write_at
Start [t]	0.022478s
Duration [t]	0.000624s
File operation	
file name	/tmp/test-test
size:	501760
offset (after view)	501760
etype size, extend	1, 1

Memory Datatype

File datatype

Accessed bytes:

Contained XML data:

```
<File_write_at fid="0" time="1240651036.332362" count="501760" tid="1275068685" offset="501760" end="1240651036.332986" size="501760">
</File_write_at>
```

Figure 4.19.: Visualization of the spatial access pattern of a non-contiguous write operation.

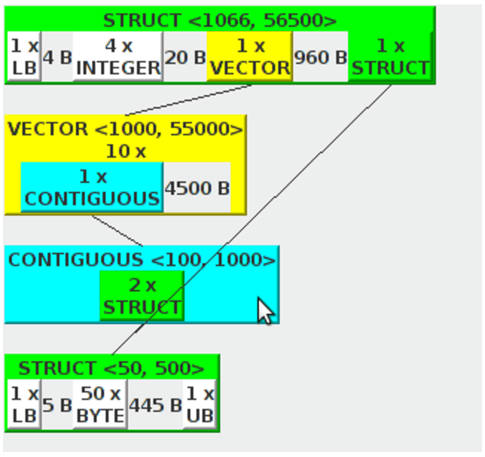


Figure 4.20.: Visualization of a fully expanded datatype as a directed acyclic graph.

the middle of a datatype, three parts must be identified – the partial start, repeats of the full matched datatypes and the end in which only the first part of the datatype is accessed. Each unrolled datatype is drawn in a box to indicate the nesting.

Visualization of our demonstration datatype is given in Figure 4.19. In this example the user begins to write at offset 501760 and writes 501760 bytes. These values are transformed by the view to skip the displacement of the view (here 0); only full elementary datatypes are matched. The write hits the datatype in the middle of the structure's vector datatype, which is just repeated 3 times. Inside the vector datatype the contiguous datatype is unrolled two times, from the first contiguous repeat 26 bytes are written, then a hole of 5 bytes. Next, the second structure of the contiguous datatype is unrolled and so forth. In the example the full datatype is repeated 470 times, then only the beginning of the datatype is written.

Both visualized datatypes assist the user in understanding the performed access pattern: to debug behavior and to compare it with the intended pattern and to understand occurring inefficiencies. This is especially useful, when PVFS2HD is used to reveal the server behavior.

4.4.5. Analyzing MPI Internals

As single collective calls can be slow due to several factors (see Section 2.2). An identification of the cause requires to look inside MPI and at the hardware utilization. While the latter is partly achieved by libRUT, the former requires instrumentation of the MPI library.

The MPI-Wrapper works with MPICH2 and Open MPI. MPICH2HD has been modified to also permit tracing of the MPI internal communication and MPI-IO operations of MPICH2. Most collective operations of MPICH2 issue a sequence of send, receive or of combined `Sendrecv()` operations. That usually means only one operation is performed at a given time. The tracing of the internals integrates seamlessly as nested operations into the MPI-Wrapper.

To illustrate the capabilities the visualization of a traced `MPI_Bcast()` with 100 MiB of data to 8 processes is listed in Figure 4.21. Each process is located on a separate node equipped with Gigabit Ethernet and two Intel Westmere processors. Without the internal tracing, the `MPI_Bcast()` would be a black box.

On our cluster MPICH2 performs a binomial tree scatter operation followed by an allgather. This explains the observed pattern: at the beginning Rank 0 sends 50 MiB to Rank 4, then 26 MiB to Rank 2, 13 MiB to Rank 1, then 7 times 13 MiB are sent to Rank 1 and received from Rank 7. Later in this thesis it will become clear that the observed communication pattern is suboptimal for our cluster. Consequently, the gained insight into the MPI calls foster understanding and identification of inefficient behavior.

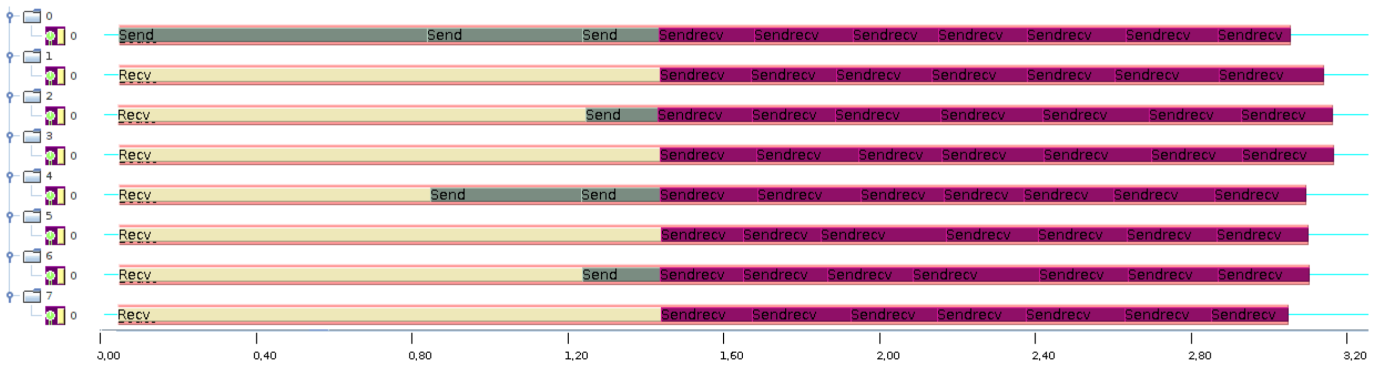


Figure 4.21.: `MPI_Bcast()` of 100 MiB of data to 8 processes.

4.4.6. Analyzing MPI and PVFS Interplay

Tracing parallel file system activities together with MPI activities allows spotting bottlenecks which remain hidden otherwise. The capabilities of HDTrace are demonstrated with an illustrating example.

In this experiment our parallel PDE solver for a 2-dimensional problem (`partdiff-par`, see Section 7.9.1) is instrumented with HDTrace. This PDE solver periodically stores the current state by writing diagonals of the matrix in a file with MPI-IO. This information then can be read-out during the processing in order to look at the convergence behavior of the algorithm. Full matrices are written after a configurable number of iterations to permit checkpoint/restart and to compare all values of the matrix at fixed numbers of iterations.

In this (artificial) problem the matrix has a dimension of about 8000x8000 double values, which corresponds to a resident set of about 450 MiB for a single matrix. The application is run on a single processor. Every 5 iterations a checkpoint of the full matrix is made and the progress information data (the diagonal has a size of 64 KiB) are written in every iteration. The PDE uses `MPI_File_write_at()` to store data without setting a file view, therefore access on disk is always contiguous. Access to the matrix diagonal in memory is performed by applying a derived memory datatype.

MPI and PVFS server activities are provided in Figure 4.22 and Figure 4.23. Activities of the PVFS client library are not included in these screenshots, because they resemble the behavior on the server side. In the first screenshot an overview of 20 iterations of the solver is provided. On top the client MPI activity including the long duration of the checkpointing can be seen. Below, the PVFS activity of one server is visualized. Each layer recorded events and/or statistics associated with the activities. Statistics encapsulate the number of concurrent operations of the particular layer. This information is not sampled or aggregated, therefore, at each point in time the value accurately represents the number of pending operations: *BMI* indicates the network activity, *FLOW* contains the regular I/O operations (very small requests are not included), *REQ* represents the number of outstanding requests. *SERVER* shows the pending state machines and processing of each state machine – here we can see performed activity by the write_sm. *TROVE* indicates the activity on the persistency layer of PVFS. In TROVE not only the number of concurrent operations are traceable. Additionally, each individual I/O operation that is issued to the operating system, can be traced with its offset and size. Concurrent operations will be expanded automatically to multiple timelines by Sunshot.

In the example up to 8 concurrent operations are observed – note this is the maximum number of outstanding operations FLOW enforces per request. Additionally several statistics of the OS are given (the folder is called “Utilization” on the left): CPU utilization, write throughput of the storage, and network activity.

While the process accesses data, it is possible to follow the server activity in detail – writing the data in the checkpoints takes most of the time. In fact it turns out the network configuration of the machines and switch degrades network throughput to 77 MiB/s while the disk has a performance of about 100 MiB/s. Therefore, with this configuration and the access pattern the network is the bottleneck. This can be iden-

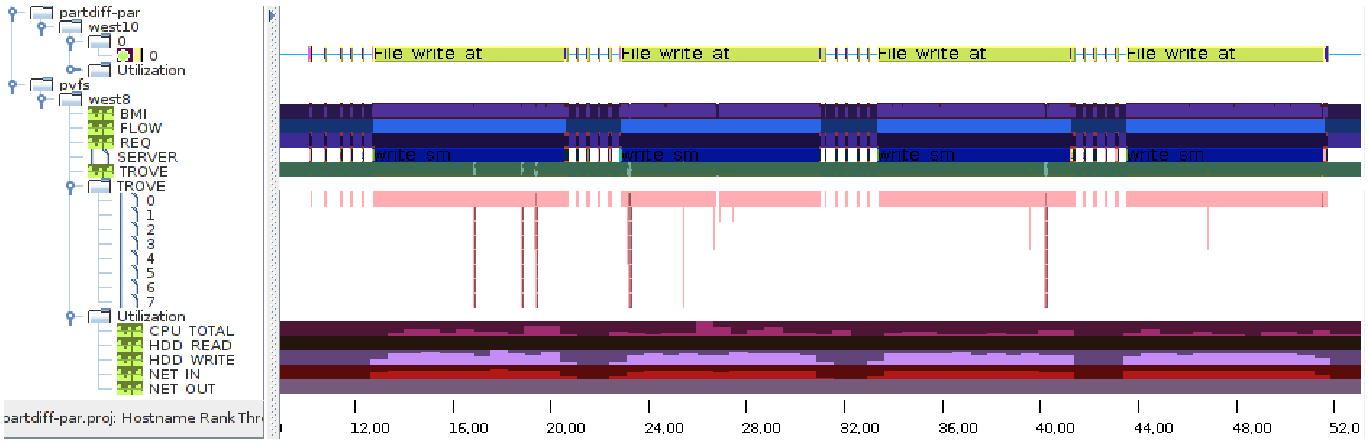
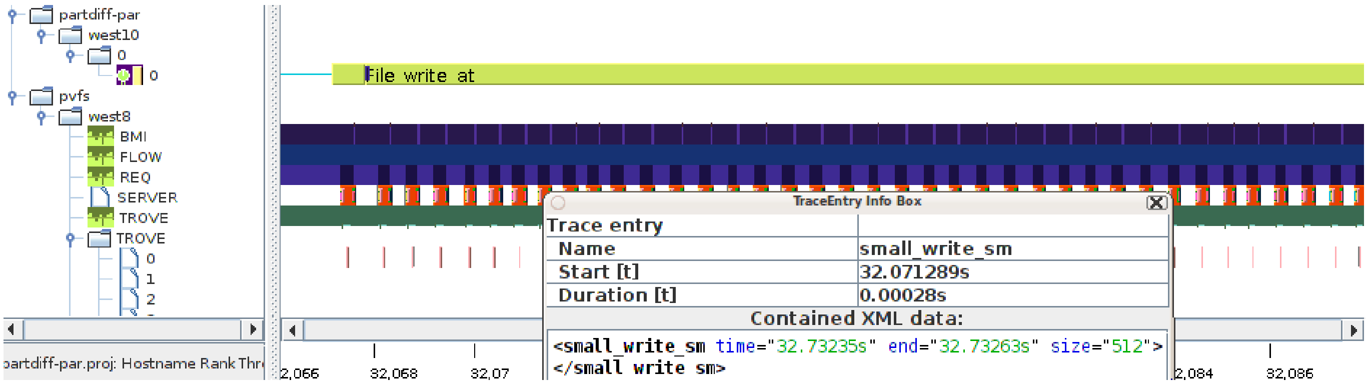


Figure 4.22.: Traced behavior of the 2D-PDE solver and the caused PVFS server activity. The first 10 iterations of the PDE solver are shown.



Trace entry	
Name	small_write_sm
Start [t]	32.071289s
Duration [t]	0.00028s

Contained XML data:

```
<small_write_sm time="32.73235s" end="32.73263s" size="512">
</small_write_sm>
```

Figure 4.23.: Traced behavior of the the 2D-PDE solver writing the progress information (the 64 KiB matrix diagonal) and the caused PVFS server activity. One single diagonal leads to 126 small requests.

tified also by looking at the BMI statistics; mostly network data is requested (BMI > 0) but the operations queue up on the TROVE layer. Effectively, the FLOW layer waits for completion of outstanding I/O before additional data can be transferred.

However, the shorter duration of writing the 64 KiB progress information is also of interest. Looking at one write operation, we can see a sequence of many requests, which are observed on the server; these are all small I/O operations (see Figure 4.23). In fact, writing the PDE diagonal generates 125 small I/O requests with a size of 512 bytes and one with 72 bytes.

Code inspection of PVFS and MPI revealed that this is due to the non-contiguous datatype in memory. MPI-IO avoids to copy the non-contiguous data in memory into an additional buffer, thus theoretically one requests would be necessary per memory region. PVFS permits to encapsulate a list of up to 64 non-contiguous operations with one ListIO request, which is used by the MPI module. This explains why 512 bytes are written per access ($64 \cdot 8$ bytes per double). PVFS also offers MPI alike memory and file datatypes. However, this feature must be explicitly activated with hints such as *romio_pvfs2_listio_write*. When this hint is enabled, then the time to write out the matrix diagonal is reduced from an average of 69 ms to 3.4 ms.

4.5. Research Activities Associated with the HDTrace Environment

In order to build and extend the HDTrace framework, the author supervised several student software labs and Bachelor and Master theses in the working group of Thomas Ludwig – at the University of Heidelberg and the University of Hamburg. These research activities are now mentioned to honor their contribution to the project. Some projects have evaluated some basic concepts before the thesis was officially started at the end of 2008. The student activity and publications related to HDTrace are listed in chronological order, this includes the aim of the activity in a brief description, the outcome and a reference to supplementary material.

In 2007 some preliminary work for this thesis has been started, in the software lab "Simulations-Prototyp für PIOsim" by Anton Ruff and Artjom König, a primitive prototype for simulation of network activity was built. The implementation achieved only suboptimal performance for the simulated network. An important result of their work was that the approach to simulate network contention and the re-transmit of packets caused too much jitter. Therefore, this early code has been discarded completely[17].

At the same time a first GUI to generate and alter cluster configurations was created in the software lab entitled "Java-GUI für die Modellierung eines Clusters im PIOsim" [Bra07]. Further, a simple graphical representation of component utilization was designed to demonstrate how that could be presented to the user.

In 2009 Dulip Withanage extended the GUI to handle the developed XML model format and to use the Java reflection API which is utilized by the simulation model. With provided annotations, the GUI automatically understood component parameters of the model classes automatically and allowed the user to parameterize them. Several ideas to easily generate and parameterize a machine configuration have been tested in his student lab [Wit09]. Components of the cluster and the interconnecting links were drawn as a *JPanel* which turned out to be problematic. Due to the layering of multiple links the identification of the link which is clicked by a user was error-prone (see his work for more details about the GUI). Later the idea of a GUI was dropped. One one hand, the complexity of GUI programming was too high and parameterization could be done quickly in Java, too. On the other hand, little further research seemed to be possible with the GUI that already realized many requirements.

A prototype for the python based automatic MPI-Wrapper generator, which was written by the thesis author, was largely extended by Paul Müller in his software lab [Mü09]. With this PMPI interceptor all parameters of most MPI functions could be recorded in the XML trace format, also communicator information and file information could be stored in detail.

At the same time the XML format for the tracing and the binary format for the statistics were co-designed with Stephan Krempel in his master thesis [Kre09] entitled "Design and Implementation of a Profiling Environment for Trace Based Analysis of Energy Efficiency Benchmarks in High Performance Computing". As the title suggests, the main goal of his thesis was to design an energy efficiency benchmark, therefore, measuring power was mandatory. Back then, existing performance tools lacked support to visualize those metrics. Therefore, I modified *Jumpshot* and designed *HDTrace* to support meaningful assessment of the consumed power together with program activity. Stephan implemented the *TraceWriting C library*, the *Resource Utilization Tracing Library* and the *PowerTracer* to generate the trace information.

Timo Minartz wrote the *HDPowerEstimation* tool for his master thesis "Model and simulation of power consumption and power saving potential of energy efficient cluster hardware" [Min09]. This tool reads the statistical utilization of components, computes the energy consumption and evaluates strategies to switch the components depending on the future utilization into power saving modes. This work has been published as a poster [MKL10a] and in a journal [MKL10b].

The development of HDTrace has been influenced by our previous project PIOviz [LKK+06]. During the development of HDTrace the tracing of internal communication in MPI and MPI-IO has been prototyped in PIOviz and was published [KTML09]. With this extended PIOviz version several internal shortcomings of MPI-(I/O) were revealed. This proved the usefulness of the approach, the features were then ported by the author to HDTrace.

[17]See Section 5.3 for the currently implemented network communication model.

The previously unique feature of PIOviz to trace client and server activity of parallel file systems has been ported in the Bachelor's thesis of Tien Duc Tien entitled "Tracing Internal Behavior in PVFS" [Tie09]. In this thesis Duc instrumented PVFS with the HDTrace API to visualize the internals of the file system. The modifications made gather even more information from the file system than before. Therewith, the author's knowledge of the PIOviz project got incorporated in this project and allows tracing and visualizing of activity in a better way than previously with PIOviz.

There is a vast number of I/O benchmarks available, yet, achieved performance could not be projected to any application workload. Therefore, the author started to write a programmable MPI benchmark already in his master thesis [Kun07].

This idea has been extended in the software lab "Programmable I/O-Pattern Benchmark for Cluster File Systems" [RS09] in which the customizable benchmark *Parabench* was created which is designed to mimic arbitrary access patterns and inter-process synchronization. Since then, this benchmark has been used in several conducted software labs [SK10] and will be used in the future. Parabench was published in 2010 [MRKL10].

To gather application benchmarks for scientific applications the software lab "Community Platform for Parabench" was started[RS10]. In this work a community portal where users can share I/O patterns and their results was created. This would not only help the users as they can determine on which machine their I/O workload runs best, but also would support vendors and scientists since it allows them to evaluate systems with realistic workloads.

A further extension of the programmable I/O benchmark was planned in which the recorded trace files from HDTrace are directly executed by an interpreter. With that approach any MPI(-IO) program could be recorded and replayed on any other system – without the need to port the original application. Thus, this tool would estimate the efficiency of I/O and communication patterns. As all parameters of MPI activities, communicators as well as mapping information are already provided by HDTrace, that tool seemed to be an ideal extension to the environment. Also, this is a similar idea to simulating behavior for any architecture. To seek for collaboration for the community platform of Parabench and the trace replay mechanism a poster has been presented on ISC 2010 [KMR+10]. Unfortunately, searching for collaboration was not fruitful and we started development ourselves. However, while a first version of *HDReplay* has been developed it turned out that another group already created a replay mechanism for MPI communication behavior [KGS+10]. Their version does not support I/O (which is our main focus). Nevertheless, the development of HDReplay and the community platform was then suspended until we find a collaborative approach for the community in the future. Negotiation with several researchers was unsuccessful and as of 2012 the development branch of OTF has been modified by their development to provide similar features.

With PIOviz [LKK+06] a suboptimal performance of collective I/O for various workloads and system configurations has already been shown. Michael Kuhn evaluated client-side and server-side I/O in his Master's thesis for PIOsimHD [Kuh09]. The thesis aimed to evaluate sophisticated I/O scheduling on the server side in conjunction with the client side collective I/O. It turned out that for many workloads the easier implementable server-side I/O is superior to the collective I/O and leads to an almost optimal performance. A research paper of his results has been published recently [KKL12]. After complementation of the theses the implementation of the cache layers and the client implementation inside the simulator has been refactored and rewritten by the author. On the one hand, in order to reduce redundant code and, on the other hand, to add some missing capabilities, for instance to combine partly overlapping accesses properly.

In 2011 Artur Thiessen analyzed the communication patterns in MPICH2 for several collective calls and started to mimic the existing communication patterns in PIOsimHD[Thi12]. While multiple algorithms are used in MPICH2 to realize a single collective operation, for every collective call one of the used algorithms was chosen and implemented.

4.6. Chapter Summary

This chapter introduces the HDTrace environment, a software ecosystem which supports tracing and visualizing of MPI activity, system statistics, and I/O behavior on client and servers. The components of HDTrace assist users in analyzing and understanding existing behavior in order to identify bottlenecks on application and system level. Especially focused on analyzing I/O behavior, HDTrace offers many unique features.

For these features information must be available in the recorded trace. However, due to the lack of support from existing trace formats when the thesis was started, a new trace format was developed for HDTrace. This chapter also discussed the creation of traces and the features of the Sunshot visualization tool. Compared to existing tools, the HDTrace format supports visualizing aspects of parallel applications which are not yet possible with other tools and environments.

Statistics, such as operating system activity and energy efficiency, can be investigated with histograms or by rescaling the timelines. Derived metrics permit to compute the metrics of interest; for example, the global network throughput and power consumption are computed based on other available statistics.

Filtering of relevant information is important, since rendering the vast amount of information requires the user to localize the relevant aspects. Sunshot offers custom color-codings and filtering with textual expressions, to reduce the amount of data that must be understood by the user.

Analysis of user defined data types and I/O access patterns is a new capability that eases the understanding of user composed datatypes – still, it preserves the nested character of the datatypes. A few screenshots indicate how derived datatypes can be visualized in a meaningful way. File views can be used to address non-contiguous data in a file with one contiguous MPI-IO call, the visualization allows users to see how a datatype is unrolled and ultimately which bytes of the logical file are addressed.

PIOsimHD simulates application and system behavior, among other use cases this capability can be used to compare simulation results with measured behavior to localize bottlenecks. In the next chapter the simulator and the underlying hardware and software models are introduced in more detail.

Bibliography

[Bra07] Mathias Braun. Java-GUI für die Modellierung eines Clusters im PIOsim. Technical report, Ruprecht-Karls-Universität Heidelberg, 10 2007.

[KGS+10] Andreas Knüpfer, Markus Geimer, Johannes Spazier, Joseph Schuchart, Michael Wagner, Dominic Eschweiler, and Matthias S. Müller. A Generic Attribute Extension to OTF and Its Use for MPI Replay. *Procedia Computer Science*, 1(1):2109–2118, May 2010.

[KKL12] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Simulation-Aided Performance Evaluation of Server-Side Input/Output Optimizations. In *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP. Munich Network Management Team, IEEE, 2012.

[KL12] Julian Kunkel and Thomas Ludwig. Visualization of MPI(-IO) Datatypes. In *ParCo 2011*, Amsterdam, New York, Tokio, 2012. University of Ghent, ELIS Department, IOS Press.

[KMR+10] Julian Kunkel, Olga Mordvinova, Dennis Runz, Michael Kuhn, and Thomas Ludwig. Benchmarking Application I/O in the Community, 06 2010.

[Kre09] Stephan Krempel. Design and Implementation of a Profiling Environment for Trace Based Analysis of Energy Efficiency Benchmarks in High Performance Computing. Master's thesis, Ruprecht-Karls-Universität Heidelberg, 08 2009.

[KTML09] Julian Kunkel, Yuichi Tsujita, Olga Mordvinova, and Thomas Ludwig. Tracing Internal Communication in MPI and MPI-I/O. In *International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT*, pages 280–286, Washington, DC, USA, 12 2009. Hiroshima University, IEEE Computer Society.

[Kuh09] Michael Kuhn. Simulation-Aided Performance Evaluation of Input/Output Optimizations for Distributed Systems. Master's thesis, Ruprecht-Karls-Universität Heidelberg, 09 2009.

[Kun07] Julian Martin Kunkel. Towards Automatic Load Balancing of a Parallel File System with Subfile Based Migration. Master's thesis, Ruprecht-Karls-Universität Heidelberg, 08 2007.

[LDK+05] Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John Mccalpin, David Bailey, and Daisuke Takahashi. Introduction to the HPC Challenge Benchmark Suite. Technical report, 2005.

[LKK+06] Thomas Ludwig, Stephan Krempel, Julian Kunkel, Frank Panse, and Dulip Withanage. Tracing the MPI-IO Calls' Disk Accesses. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 4192 in Lecture Notes in Computer Science, pages 322–330, Berlin / Heidelberg, Germany, 2006. C&C Research Labs, NEC Europe Ltd., and the Research Centre Jülich, Springer.

[May01] John May. MPIMap User's Guide, 2001.

[Min09] Timo Minartz. Model and Simulation of Power Consumption and Power Saving Potential of Energy Efficient Cluster Hardware. Master's thesis, Ruprecht-Karls-Universität Heidelberg, 08 2009.

[MKL10a] Timo Minartz, Julian Kunkel, and Thomas Ludwig. Simulation of Cluster Power Consumption and Energy-to-Solution, 04 2010.

[MKL10b] Timo Minartz, Julian Kunkel, and Thomas Ludwig. Simulation of Power Consumption of Energy Efficient Cluster Hardware. *Computer Science - Research and Development*, pages 165–175, 2010.

[MMK+12] Timo Minartz, Daniel Molka, Julian Kunkel, Michael Knobloch, Michael Kuhn, and Thomas Ludwig. *Tool Environments to Measure Power Consumption and Computational Performance*, pages 709–743. Chapman and Hall/CRC Press Taylor and Francis Group, 6000 Broken Sound Parkway NW, Boca Raton, FL 33487, 2012.

[MRKL10] Olga Mordvinova, Dennis Runz, Julian Kunkel, and Thomas Ludwig. I/O Performance Evaluation with Parabench – Programmable I/O Benchmark. *Procedia Computer Science*, pages 2119–2128, 2010.

[Mü09] Paul Müller. PIOsim MPIWrapper to Create XML-traces. Technical report, Ruprecht-Karls-Universität Heidelberg, 04 2009.

[RS09] Dennis Runz and Christian Seyda. Programmable I/O-Pattern Benchmark for Cluster File Systems. Technical report, Ruprecht-Karls-Universität Heidelberg, 11 2009.

[RS10] Dennis Runz and Christian Seyda. Community Plattform for Parabench. Technical report, Ruprecht-Karls-Universität Heidelberg, 06 2010.

[SK10] Jens Schlager and Marcel Krause. Evaluating Selected Cluster File Systems with Programmable I/O-pattern Benchmark Parabench. Technical report, Ruprecht-Karls-Universität Heidelberg, 04 2010.

[Thi12] Artur Thiessen. Simulation von MPI-Collectives in PIOsimHD. Technical report, Universität Hamburg, 04 2012.

[THW10] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A lightweight Performance-oriented Tool Suite for x86 Multicore Environments. In *39th International Conference on Parallel Processing Workshops*, ICPPW, pages 207–216. IEEE, April 2010.

[Tie09] Tien Duc Tien. Tracing Internal Behavior in PVFS. Bachelor's thesis, Ruprecht-Karls-Universität Heidelberg, 10 2009.

[Wit09] Dulip Withanage. Analyse und Erweiterung der JavaGUI für PIOsimHD. Technical report, Ruprecht-Karls-Universität Heidelberg, 02 2009.

Chapter 5 V

PIOsimHD – The MPI-IO Simulator

This chapter introduces PIOsimHD, the simulator for cluster systems. General considerations of modeling hardware and software components, and an appropriate level of detail are discussed in Section 5.1. The implemented models of the physical entities for a cluster system, are discussed in Section 5.2. Since network communication plays an important role in the performance of a distributed system the underlying communication model is assessed separately in Section 5.3.

The software model for executing parallel applications is presented in Section 5.4. Furthermore, models for layers involved in parallel I/O on client-side and server-side are included.

The workflow of conducting experiments in silico is explained in Section 5.5. This describes approaches to setup models, to control the simulator and to the process of visualizing and analyzing simulation results.

5.1. Modeling Behavior on Application and System Level

In *PIOsimHD* parallel applications and the system hardware on which those programs run, are modeled and simulated. The Java based simulator comes along with *PIOsimHD-Model*, which offers an abstraction layer for application behavior and for the system model, including simulated entities and their attributes[1]. While behavior of the modeled components is coded inside the Java simulator, the parameters offered by the model specify the characteristics of the implemented behavior; for example, the average seek time of a hard disk drive. When the simulator is started it takes the model of the system and instantiates the implementations that represent and realize these models. An advantage of this approach is a clean separation between abstract model and implementation details.

To answer a scientific question, such as the localization of bottlenecks during execution of parallel applications[2], the model must cover the right level of detail. Also, the concept of model time is a fundamental aspect to understand how processes are simulated. Hence, aspects for the discrete-event processing in PIOsimHD are discussed in this section: the level of detail, the event processing engine and the model time.

Appropriate level of detail The complexity of a live system is demonstrated in Chapter 3: memory access, point-to-point communication and I/O with a local hard disk drive are analyzed. Individual operations show various non-linear behavior and, additionally, jitter leads to unpredictable duration. To model the behavior of cluster hardware and software accurately a very high level of detail is necessary. However, a high level of detail is disadvantageous for various reasons:

- **Simulation complexity**: A detailed level of abstraction requires low-level behavior simulation (eventually down to the circuits of a component and every line of a software). Since real components are sophisticated pieces of hardware that consist of billions of transistors some level of abstraction is required for a cluster simulator.
- **Expensive simulation**: A high level of detail requires more CPU time to compute the next state of the model, and additional memory is required to maintain the internal states.
- **False accuracy**: When a component such as a hard disk drive is modeled close to reality, then during parametrization all required parameters of the real system must be known. Otherwise, the simulation of the real system is based on wrong facts which leads to an incorrect result. Thus, the level of detail is restricted to characteristics of the component that can be measured or determined by

[1]Refer to Section 2.5 for an introduction to the field of modeling and simulation.
[2]Refer to Section 1.3 for the list of questions that are addressed with PIOsimHD.

other means. To give an example, in modern hard disk drives the controller remaps bad blocks, this mapping cannot be obtained easily.

- **Tough interpretation**: Since the user must interpret simulation results a simple and understandable model is advantageous. Interpretation of a very detailed model is as hard as to understand the real behavior of the system: The cross-talk between the activities and the cause-and-effect chain increases the complexity of identifying the reasons for observed behavior.

Consequently, it is mandatory to focus on the right characteristics which represent the system well enough to answer the scientific questions. Therefore, one requirement for PIOsimHD is to support a variability of characteristics which is realized by a modular design.

With the knowledge about system behavior several characteristics that, in general, influence performance, and theoretic performance limitations are discussed in Section 2.2. This includes characteristics that might lead to non-linear behavior in the real system. The relevance of individual performance factors depends on the system and the workload. Most of these characteristics can be understood by the user. Therefore, these characteristics serve as a starting point for the developed models. Currently, all implemented models focus on the most important characteristics of the real hardware and software. Nevertheless, with the modular concept of PIOsimHD the models of components can be exchanged varying the level of detail – in many cases an individual model can be selected per component.

Event processing Two types of events are distinguished: *regular events* indicate the start or completion of work and *internal events* created by components to control their behavior, for instance to set up a work-timer which fires at a given model time. Typically, the processing of an event creates new (future) events.

Events are managed in a data structure, that permits to quickly retrieve the next (future) event. The events are processed in a loop which picks the next pending event, increments model time to the event start time and executes the event on the component it shall be executed (the abstract processing of discrete-event simulators is illustrated in Figure 2.25).

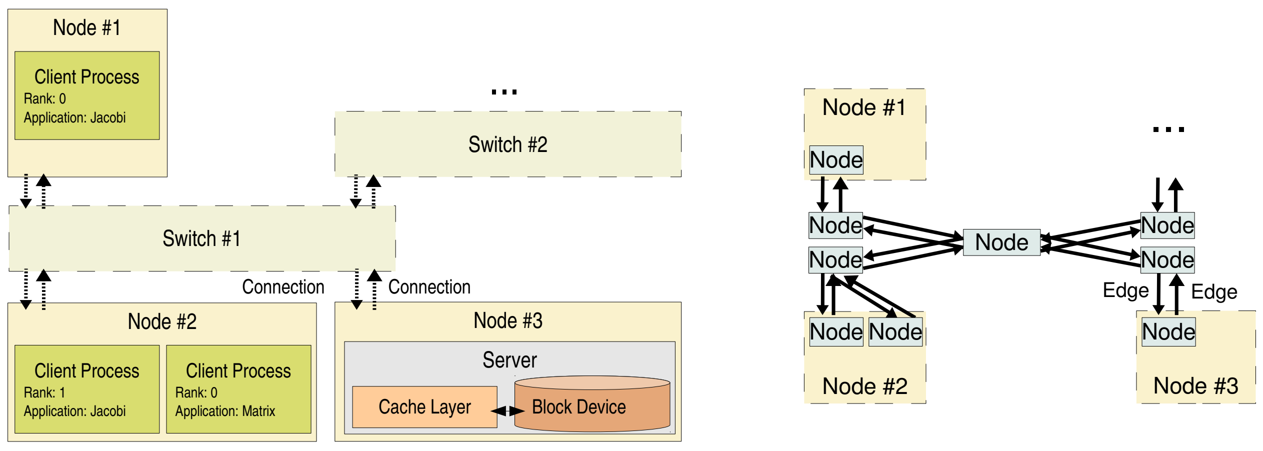
Most components are so-called *blocking components*, which means that they exclusively process one *job* of a given type after another. When jobs arrive faster than they are processed, they queue up in the component. A component can select one of the pending jobs with an arbitrary algorithm. Typically pending jobs are processed with the *first-come, first-served* strategy, hard disk drives, however, include complex schedulers that can combine multiple operations into one.

Currently, a CPU is the only component which shares available resources equally among all pending jobs, all other components process jobs sequentially – usually in the order they were submitted.

Model time The simulator maintains model time with a special datatype – the *Epoch*. *Epoch* manages time in nanosecond accuracy, in the simulator it is implemented with two integers, one that stores seconds and one that stores nanoseconds. Implementation of an own datatype has been chosen to guarantee extensibility in the future and simulation accuracy. First, the Epoch datatype can be adjusted in the future to increase the resolution of the model time on demand. Second, with Epoch numerical errors that could occur with floating point datatypes are not possible: While a model time of 1,000 s stored in nanosecond accuracy requires only 12 significant decimals and thus fit in the IEEE 754 double-precision floating-point format, a deterministic execution would require rounding of trailing decimals. Otherwise the simulation might change depending on the model time. Also, users may simulate long-running processes exceeding the accuracy of 15 significant decimals.

5.2. Hardware Model

The basic hardware model reflects the common sense of a cluster computer: One or multiple *client* processes of an MPI applications are executed on a *node*. Resources required for computation, such as CPU



(a) Components (b) Star topology of a switched network

Figure 5.1.: Illustration of the modeled hardware components and their interactions.

and memory are hosted on the node and shared among all *hosted processes*. On each node one I/O *server* of the file system can be placed. A server holds a *block device* which is controlled by the server's *cache layer*. The cache layer is an intermediate layer which does not perform time-consuming operations, instead it interfaces between network and block device, and it manages the available cache space.

An illustration of a regular cluster system with a sketched network is shown in Figure 5.1a: Three processes of two different applications are placed on two nodes, a third node hosts a single I/O server. Nodes are interconnected with a switched network topology, this is outlined in the figure by the two switches.

The network model is not limited to a switched network topology, instead arbitrary network graphs consisting of network nodes and edges can be created. A network topology defines how *network edges* are connected to intermediate (network) *nodes*. Each *hosted process* brings its own network interface (*NIC*), that must be connected. A central switch is modeled with the network topology presented in Figure 5.1b.

To cope with several levels of abstraction, the model for a component can be selected. In this context the model refers to the parameters that represent specific characteristics of the component's behavior – a model can use any number of parameters. Currently, the characteristics of provided models do not depend on stochastic processes, although it would be easy to extend the simulator in that direction.

Additionally, a modeled component can have several implementations that all rely on the same set of characteristics. When the model specification is made for a simulation experiment, the model parameters and the implementation are selected (more details are provided in Section 6.3) – during simulation the models and all characteristics are fixed. Usually, those parameters are provided in vendor specifications – in datasheets, for example, or they are obtained by benchmarking the existing system.

For this project several models have been implemented. The currently available hardware models and their implementations are discussed in more detail in the following: Nodes and process in Section 5.2.1, the block device in Section 5.2.2 and the network in Section 5.2.3. UML class diagrams of implemented models and their parameters are provided. For a sake of simplicity only a subset of the model class hierarchy is provided and interfaces are omitted in the class diagrams. As the implementation is usually more complicated, the behavior is illustrated and described in prose.

5.2.1. Node and Hosted Processes

The model for a node and for software processes, which can be hosted on a node, is shown in Figure 5.2. A *Node* offers memory and computation resources: it is equipped with an amount of memory and a number of

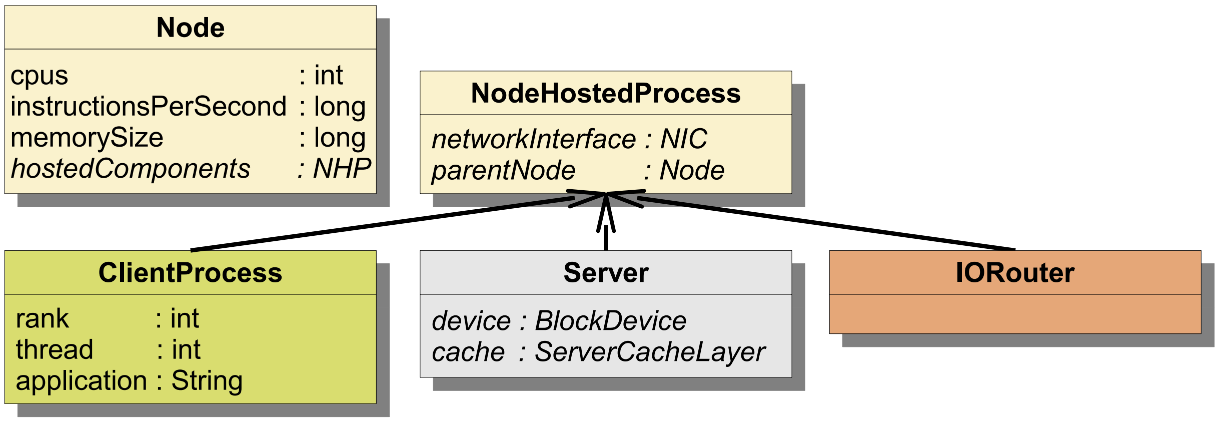


Figure 5.2.: Nodes and hosted processes that share the resources provided by a node.

CPUs[3]. Each CPU processes a fixed number of instructions per second. An instance of a *NodeHostedProcess* subclass can be placed on a node and use the provided resources. The resources of a node are shared among all processes that are hosted on the node.

A *ClientProcess* represents an MPI process which has a rank and an application assigned. A *Server* process is associated with a block device (see Section 5.2.2) and a cache layer that controls the block device and manages the available cache. *IORouters* (I/O forwarders) relay network traffic between client and servers of the simulated parallel file system – they are mentioned in the discussion of the I/O path. Execution of the MPI process and the client-server communication are part of the software model (which is discussed in Section 5.4.1).

Job execution Any NodeHostedProcess may start a *computation job* on the node. The node executes the jobs and drives the processing. If multiple jobs run concurrently and the number of CPUs does not suffice to permit an exclusive assignment of a job per CPU, the available execution time will be multiplexed among all processes in a fair manner.

This behavior is illustrated in Figure 5.3. In this example three independent processes submit compute jobs to one node which has two CPUs – a CPU processes one instruction per second. When more than two jobs are scheduled, all three run concurrently but each of the processes will proceed at a speed of 2/3 instructions per second. Starting of new jobs is indicated in the figure by an arrow, the number of instructions to compute (the workload) is printed next to the arrow. At t=0 process A issues a job which takes three cycles, at t=4 a new job with 3 instructions is issued and at t=5 process B issues a job. All those jobs can run concurrently at full speed on the two CPUs. At t=7 all three processes issue a job, now the processes must share their computation resources among them and thus a job which needs 2 cycles requires 3 seconds. During these three seconds process C reduces the remaining cycles of its 6 cycle workload by 2 (4 remain). The next two seconds the job can run at full speed as only one job is scheduled and thus two cycles remain. In the last phase, A and B again submit a 2 cycle job, therefore 3 seconds are required to process the last 2 cycles. Due to the overcommitment of the CPUs the job submitted by process C took 8 seconds instead of 6.

On real world systems, a scheduler dispatches processes in round-robin exclusively for a given time quantum. That means available computation time is split into small slices (e.g., 10 ms), and a process is assigned to a single CPU and executed during its turn. In contrast to a real system, the current implementation of the model shares CPU time equally among all processes. For long-running jobs the behavior of model and reality converge, but for short jobs the fair sharing leads to a different completion behavior. In a real system, for example, multiple jobs that require one time slice would finish one after another. It is faster

[3]The mapping of these CPUs to physical sockets and cores can be taken into account by creating an appropriate network topology for the processes, i.e., processes hosted on one socket are connected with faster links.

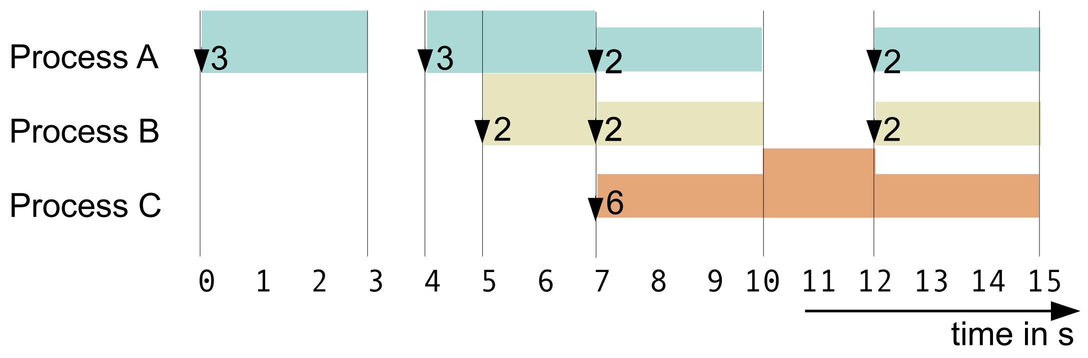


Figure 5.3.: Illustration of a node model for concurrent computation on two CPUs. Arrows indicate submission of compute jobs and the number next to the arrow corresponds to the workload (in s). The height of a bar indicates the CPU utilization of the job.

to do this analytical computation than simulating time sharing of a real CPU because an update is only necessary when a new job is submitted (or if a dispatched job completes).

Memory Right now, available memory is only used for caching I/O on the server side. The amount of memory required for an application is not considered, neither are additional buffers for network communication. Theoretically, memory usage for caching incoming messages could be taken into account, while application memory footprint in simulation would require a modification of the trace environment.

Simulation of memory access times required for a compute job is not performed because this information is not available in trace files. This, in turn is due to microprocessor design: Memory bandwidth is shared among all processor cores because all processes access memory through the L3 cache. Thus, memory accesses cannot be assigned to one process explicitly. Memory access times for data communication can be covered by the network model to some extent (this is described in the network section).

5.2.2. Block Device

Operations performed on a block device are characterized in the model by the type, the amount of data to access (called *jobSize*), the file to access (the *jobFile*) and the offset at which the data is accessed (called *jobOffset*). The type of an operation is either read, write or flush – the latter ensures that write operations are completed. This is the identical information that is available in the cache layer, in contrast to the traditional block-oriented interface, the models are based on objects. Therewith, a mapping from blocks to LBAs is not done in the cache layer. In fact, currently all block devices abstract from the block allocation.

Implemented models for the block device are shown in Figure 5.4. Two models are currently supported, the *SimpleDisk* model and the *RefinedDiskModel*. A block device might optimize the accesses when multiple requests are on the fly, the number of requests that can be issued concurrently is controlled by the cache layer. The base class of a *cache layer* offers a parameter to specify the maximum number of concurrent requests that can be issued.

Simple disk model In the primitive *SimpleDisk* model, the time for an I/O job depends on the amount of data accessed and the latency. The duration of an I/O is computed using the equation $t = \texttt{avgAccessTime} + \frac{jobSize}{\texttt{maxThroughput}}$. Variables in `typewriter` style are model parameters and parameters that depend on the job are *italic*. Requests are processed with the *first-come, first-served* (FCFS) strategy, read and write accesses are treated symmetrically. Dispatched flush operations complete instantaneously. This simplified model can be applied to resemble SSDs to some extent.

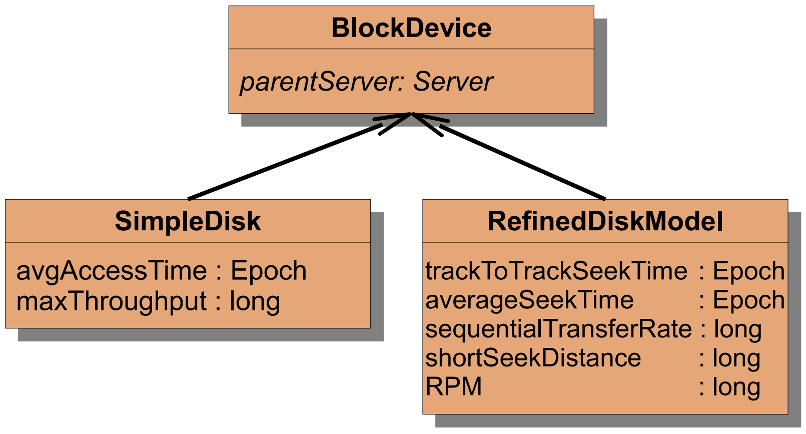


Figure 5.4.: Modeled block devices.

Refined disk model The *RefinedDiskModel* takes the following parameters into account: sequential transfer rate, an average seek time, track-to-track-seek time and the revolutions per minute (RPM). When an I/O operation is dispatched the time to seek over the disk is determined by the distance to the last accessed byte and the new offset. A simple block mapping is applied: Each file is assumed to be stored sequentially on disk; the implementation tracks the last file and offset in the variables *lastFile* and *lastOffset*.

The distance is determined in the model as follows: If the same file is accessed, then the distance of *lastOffset* and the *jobOffset* defines which type of seek to perform. When the offset matches then no seek is necessary, also it is assumed that data can be accessed via the disk cache – thus no latency applies at all. The maximum tolerable distance for a track-to-track seek operation is specified in the *shortSeekDistance* parameter of the model. Access to other files always enforce an average seek, this is motivated by the fact that files are considered to be larger than the *shortSeekDistance*. Equation 5.1 shows equations for the computation of access time.

$$\text{rotLat} = 30/RPM \tag{5.1}$$

$$\text{latency} = \begin{cases} 0 & \text{if lastFile} = \text{jobFile AND lastOffset} = \text{jobOffset} \\ \text{rotLat} + \mathit{trackToTrackSeekTime} & \text{if lastFile} = \text{jobFile AND} \\ & \|\text{lastOffset} - \text{jobOffset}\| < \mathit{shortSeekDistance} \\ \text{rotLat} + \mathit{averageSeekTime} & \text{otherwise} \end{cases} \tag{5.2}$$

$$t = \text{latency} + \frac{\text{amount of accessed data}}{\mathit{sequentialTransferRate}} \tag{5.3}$$

The implementation tries to reorder requests to optimize access. Pending requests are scheduled with an *elevator* alike algorithm[4] which works similar to *NCQ* of SATA disks.

In detail, the implemented algorithm groups pending jobs based on their filename in a queue. The list of files which have pending operations is processed sequentially – the first file is selected and all pending operations for that file are moved to a queue that is ready for execution (the *scheduled file queue*). Jobs for the scheduled file are processed from low offset to high offset in similar fashion to the elevator algorithm. When the execution of all scheduled operations is completed, the next pending file is scheduled and all operations that are currently pending are moved to the *pending file queue*. Therefore, operations for a file are queued until the file is actually dispatched. There is one exception: If a new job is issued that operates on the same file as *lastFile* and also with a higher *jobOffset* than *lastOffset*, it will be appended to the list of currently scheduled jobs as well. Otherwise, the file operation is appended to the list of pending files. Consequently, operations might be reordered.

[4]Often this algorithm is referred to as *SCAN*.

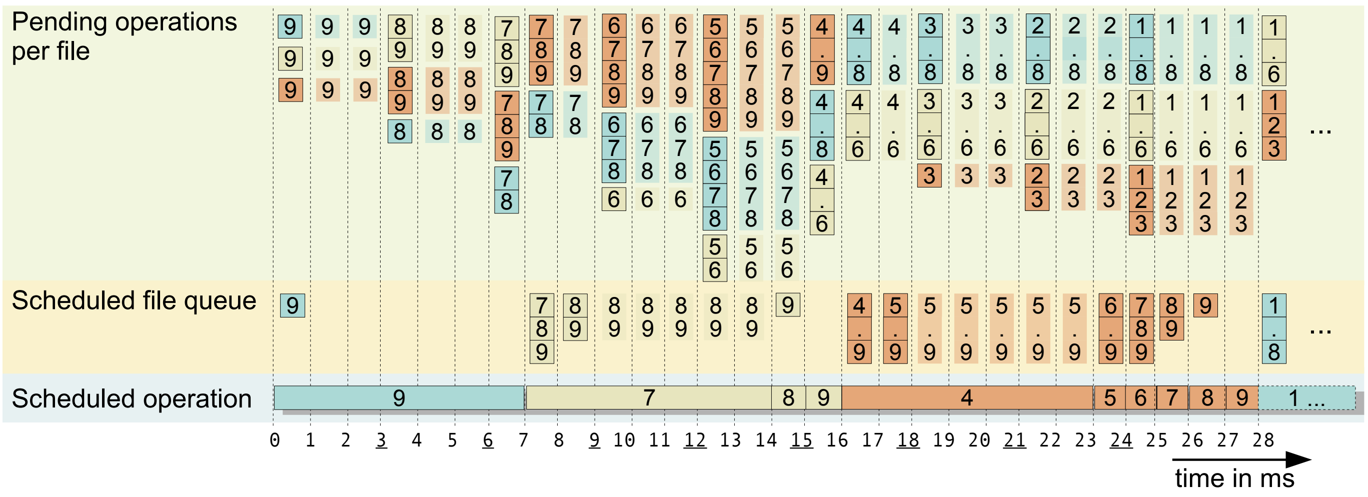


Figure 5.5.: Illustrated example for the elevator algorithm of the RefinedDiskModel. For every timestep the pending operations per file, the operations of the currently dispatched file and the scheduled operation is provided. Elements of modified lists are highlighted.

Flush operations are scheduled with other accesses and executed once all other operations have completed; execution of a flush operation takes zero time. They are always appended to the pending file queue which guarantees the synchronization character: New jobs that have a higher *jobOffset* could be executed before a previously submitted flush is performed, but never after.

Illustration of the processing for the refined disk model The operation of the algorithm is illustrated in a small example: Assume three files are accessed concurrently backwards in blocks of 512 bytes. The accessed blocks are numbered from 1 to 9, and one I/O job arrives every 3 millisecond for each of the files. The processing of an operation which does not involve a seek takes 1 ms and one which includes the average seek time needs 7 ms; short seeks are not performed in this example.

The elevator principle and simulated processing of this experiment is shown in Figure 5.5 – time increases from left to right, timestamps are printed at the bottom. The upper row indicates the pending jobs per file, the next row shows the operations of the scheduled file – jobs of the same file share one color. And the last row illustrates the processing of the scheduled operation – long operations represent accesses performed with average seeks, short blocks represent accesses executed without seeks.

At t=0, a job is submitted for the last block of every file. The first file is dispatched, however, it contains just one operation which takes 7 ms as it is the first scheduled file. In the meantime (at t=3 and t=6) operations for two new blocks are queued up for all files.

Actually, new operations to the file can be queued directly in the scheduled file queue, if the *jobOffset* if the new operation is bigger than the currently processed offset. This does not happen during the example, though, therefore new operations are always queued in the pending list and not in the scheduled file queue.

At t=7, the next file is scheduled; now operations for block 7, 8 and 9 are pending. While the first operation involves the average seek time, the other two are performed quickly (at t=14 and t=15) because the *jobOffset* matches *lastOffset*. It can be observed that operations queue up, if average seek time is involved, but due to grouping and dispatching a sequential list of blocks for every file, only 6 average seeks are performed in total.

Discussion of the models Internally, the models of a block device operate on file-level and extents rather than on block-level as block devices usually do. This permits to handle the mapping of blocks to the hardware inside the block device (if desired) and reduces the complexity of the cache layer.

Additionally, writing of a small amount of data does not involve a read-modify-write cycle in the presented model. However, in real systems a read-modify-write cycle might be necessary. This functionality could be realized in the implementation of a block device but has not been considered important, yet. In practice, applications should avoid accessing of partial blocks and, furthermore, caching algorithms on all layers of real systems try to aggregate operations into full blocks of data to mitigate degradation due to read-modify-write.

Right now there is no disk geometry oriented implementation because such a detailed model is quite complex and it is hard to assess its results for several reasons: The mapping from file offset to disk blocks is tedious, as explained before. Recording of block-level data access is not available inside the parallel file system. Thus, a comparison of simulation results with observations would not be possible. Furthermore, such a model can pretend false accuracy when the parameters of the disk cannot be determined correctly.

Later experiments will show that the current level of detail suffices to achieve a realistic and understandable behavior.

5.2.3. Network

Network connectivity is modeled by a graph of nodes and directed edges. Edges represent links between the nodes, a node routes packets over one of the outgoing edges or consumes the packet – messages are fragmented into packets. While packets are routed over the network graph a contention on a node or an edge defers processing of the packet (further details about the data flow between the components are provided in Section 5.3).

The class hierarchy of available network models is presented in Figure 5.6. These classes just hold the necessary attributes for the models, the connectivity between network components is stored in the network topology object.

Edges and nodes *Network edges* are characterized by latency and throughput. Implemented *network nodes* are derived from the *StoreForwardNode* which implements generic store-and-forward switching. Nodes are also characterized by a maximum throughput, which limits the speed data is relayed. A node or link is busy while it processes the data of a packet, once the job is completed the packet is transferred to the next component – the arrival of the packet is delayed by the network edge latency.

Network interface card A NIC is a special types of node that injects and destroys packets. Network interfaces interact with *NodeHostedProcesses*. They provide an interface for submission of high-level communication jobs, which are referred to as *messages*. Basically, messages which are sent are fragmented into packets of fixed size by the network interface. The packet flow from source to target is explained in detail in Section 5.3.

A *NICAnalytical* determines the duration of a submitted send job with an analytical methodology by computing arrival time for the target NIC upon job submission. For this the route to the target is determined when the message is initiated. Processing time for a job is determined with equation 5.4. In the equation components are referred to by the variable c. The network component on the route with the smallest throughput determines the effective throughput of the job, the latencies of intermediate hops are summed up.

The analytical approach causes less events in the simulator and thus computes faster. But the analytical approach cannot handle congestions in the network. However, the analytical model provides a rough performance estimate, and by comparing the analytical NIC model with a store-and-forward node, it can be used to assess the impact of network congestion.

Two implementations of an analytical model have been tested: In one implementation the message is transferred directly, in the other version data is segmented into packets. An advantage of the former

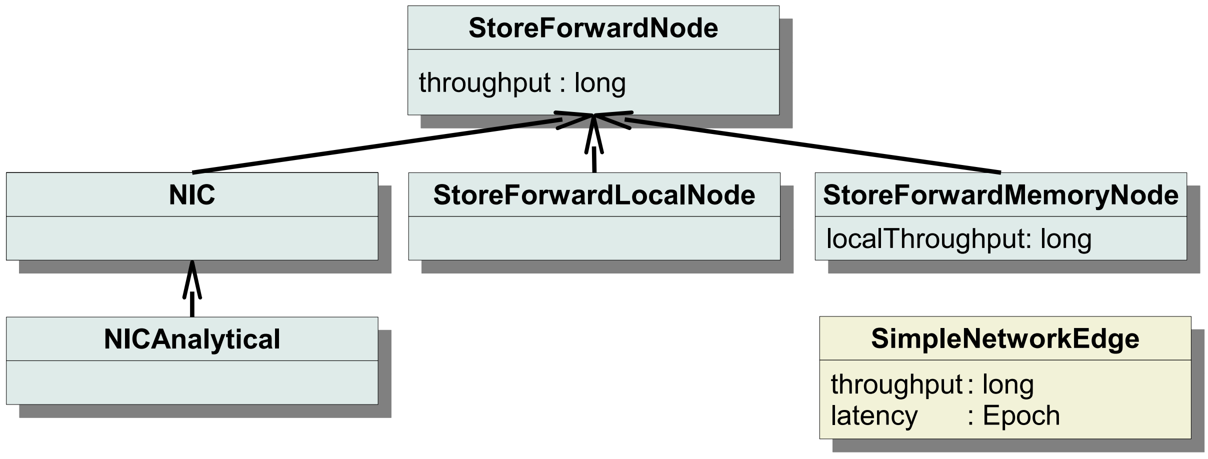


Figure 5.6.: Modeled network components.

implementation is that simulation time is reduced dramatically. However, network congestion is not modelled and parallel I/O cannot be simulated since transport is not pipelined. With the latter model network congestion on sending and receiving nodes are simulated explicitly, but it is limited to these nodes; congestion on intermediate nodes is not covered. Further, parallel I/O is supported. Therefore, the model for transferring a full message at once is deprecated.

$$t = \left(\sum_{c \in \text{route}} \mathit{latency(c)} \right) + \frac{\text{jobSize}}{\min_{c \in \text{route}} \mathit{throughput(c)}} \tag{5.4}$$

Simulation of memory access for data exchange Memory throughput limits communication performance to some extent because data must be copied between memory and network adapter[5]. The concept of network nodes can be stretched to simulate memory access time due to communication: By setting the maximum throughput of a network node to the available memory performance it limits network performance as memory does. Consequently, traffic is routed over the node that represents memory and, therewith, limits performance.

This is especially useful for intra-node communication – shared-memory is used when two processes within the same node communicate. Typically, this is faster than communicating data over the network interconnecting nodes. However, with intra-node communication data is read from memory and might be written to the same memory subsystem; consequently memory throughput is shared between those two processes.

The *StoreForwardLocalNode* model considers this fact and simply halves available throughput when data is exchanged by two processes within one node. The *StoreForwardMemoryNode* has an additional parameter, the *localThroughput*, which restricts throughput if data is exchanged between two direct neighboring components of a network node. By connecting NICs of processes with this node, communication between NICs of these share the memory system, this approximates the memory architecture of the Intel Nehalem architecture[6].

5.3. Transport Layer Communication Model

Similar to existing communication concepts the developed transport layer communication model requires that messages are fragmented into packets. Packets are created by a NIC, which additionally provides

[5]Refer to Page 2.2.4 for a discussion of the memory usage involved in communication.
[6]Example models for our cluster are discussed in Section 7.3 and illustrated in Figure 7.3.

a high-level message interface. Thus, the basis for transferring messages between two processes is data transport of single packets and a stream of multiple packets. The reliable packet transfer and flow control is discussed in this section.

Packets are forwarded by all *network components* (nodes and edges in a network topology) from the source to the target. Incoming data is buffered on each intermediate node; it performs *store-and-forward* switching on the input port. Multiple network topologies can be created with an individual routing algorithm[7]. The routing of individual packets is explained in Section 5.3.1.

The development of the packet transport layer for the simulation has mainly been guided by two requirements: 1) Bottlenecks should cause understandable and realistic behavior. 2) Available resources should be utilized to provide estimates for possible performance. An illustrating example of a bottleneck in the network topology and potential throughput is given in Section 5.3.2.

Several transport algorithms have been evaluated for the simulator. Mechanisms like wormhole routing or store-and-forward switching with dropping, are not capable to saturate congested networks completely. Common protocols and technology, such as for TCP and Infiniband, are also sensitive to certain parameters (such as the TCP window size) that depend on the hardware configuration. In the literature many cases are provided in which TCP or Infiniband deliver suboptimal performance (for example, see [LM94, FLS+06]). Therefore, for this thesis another network flow model is designed that is based on streams. In Section 5.3.3 the principle of this data flow concept is explained. Our example is then adjusted in Section 5.3.4 to demonstrate the packet transport of the flow model.

The packet flow model covers physical layer, data link layer, network layer and transport layer of the *OSI model* [Zim80]. The application layer that allows message matching similar to MPI is discussed separately in Section 5.4.2.

5.3.1. Packet Routing

The NIC on the sender-side fragments messages into chunks called *packets* of a maximum size which is specified in the global parameter *network granularity*. In the network graph packets must be transported between the *source* and the target of the packet (*destination*). The source and destination are NICs, while the intermediate network nodes can be instances of a *StoreAndForwardNode* subclass.

Individual packets are processed with *store-and-forward* switching[8] on the input port of a component: When a single packet arrives on an input port, the component queues the packet. Basically, a packet is processed when the required outgoing port is ready – this decision is made by the packet flow model (see below). The term *processing* means the component actively relays a packet to another component; this symbolizes copying data from input port to output port.

The model applies store-and-forward switching to nodes and edges. Processing of a packet utilizes a network component completely. The time needed to forward a packet is determined by the component's throughput. Additionally, when an edge finishes to transmit a packet towards the next node, it arrives there later – determined by the latency of the edge. This approach is different from the processing model of a block device as the block device is considered to be busy while seeking on the drive.

Since an edge is connected to just two endpoints, a packet that is submitted to one end of the edge is transported directly to the other end. A node can be connected to multiple edges and thus multiple routes can exist in the network graph. Therefore, every node maintains a routing table that is created upon startup of the simulator. A routing algorithm decides which route to use: While a packet is on its path through the network graph, each intermediate node decides which outgoing edge to relay the packet to and thus each

[7]See Page 30 for an introduction to routing strategies.

[8]In an earlier stage of the simulation, a cut-through network model was implemented as well. However, cut-through switching does not improve performance when an outgoing link is congested, and since this project does not focus on network simulation the model has not been maintained in newer versions of the simulator.

packet is routed individually. This algorithm can be selected in the system model. When multiple routes exist, the algorithm can use them to improve performance and to balance workloads.

Currently, packets can be either routed statically on the shortest path, or they can be transferred in a round-robin fashion to neighbors that have the same distance to the target. The latter algorithm is useful to improve performance in regular structures such as N-dimensional grids.

An efficient routing algorithm must take the network flow model into account: It can happen that a communication partner blocks data transfer because a component is saturated. To achieve optimal performance in a congested network an algorithm must take this into account, otherwise it achieves suboptimal performance. A routing algorithm that is aware of the flow status checks whether a target node has capacity to receive the packet before it is relayed to the component. Since a component has a global view on all components in PIOsimHD, good load-balancing schemes can be evaluated.

5.3.2. Congestion and Bottlenecks

The concept and potential problems of simulating packet transport are clarified in three thought experiments. A very simple experiment illustrates prioritization of slower data streams by overloading an intermediate link. In a more involved example, the propagation of congestion is assessed. This experiment is also used later in the section to demonstrate how a flow is processed. In the last example the interaction between prioritization of data data streams and propagation of congestion is discussed.

Prioritization of data streams Assume two source nodes (A and B) send a very large message to one target node (node Y), one intermediate node interconnects the sources with the target. This topology is shown in Figure 5.7; node A has a faster connection than B, latency is not considered. In this setup the link of the central node towards node Y is the bottleneck because the maximum performance of the incoming edges is 120 MiB/s while the outgoing edge can only transfer packets with a speed of 100 MiB/s.

A well designed flow algorithm achieves the maximum flow of 100 MiB/s on the edge towards Y by throttling the incoming data flow to the node. There are several ways to distribute the available bandwidth among the incoming edges. Available throughput on all edges can be throttled by the percentage the incoming data flow exceeds the possible outgoing flow; i.e., performance of all streams is reduced to a percentage of the requested demand, in the example to 100/120 = 83%. In the steady state this would lead to the effective throughput illustrated in Figure 5.7b.

One disadvantage of this solution is that even the slow performance of the edge from node B is throttled further to 17 MiB/s. Another solution is to prioritize the slower data stream, this solution is depicted in Figure 5.7b. While this is not completely fair to node A, the performance degradation of its data stream is minimal. Such a configuration may lead to a better overall behavior because the communication of node B finishes earlier[9].

Now imagine the throughput of the edge from node B is improved to 50 MiB/s. In this case, a total of 150 MiB/s could flow into the center node. With a fair sharing, the steady state would transfer 66 MiB/s and 34 MiB/s from node A and B, respectively. With the priority scheme, both edges would achieve 50 MiB/s. In both cases the transfer of A is delayed by B.

Attention must be paid to the priority algorithm to avoid unfair treating of the faster edge: Assume performance of the edge is increased further to 60 MiB/s. If an algorithm that prioritizes data flow grant the 60 MiB/s to the slower edge, the performance of the edge would degrade to 40 MiB/s. Thus, the faster edge could actually be throttled to be slower. Now imagine the slower edge achieves 99 MiB/s – this would effectively starve the faster edge. Therefore, when an edge has a speed of more than half of the faster edge,

[9] In general, to achieve optimal performance, the communication that is on the critical path of the execution should be performed as quickly as possible. Since a priority scheme does not know about the critical path, it is impossible to prioritize optimally. A prioritization of slower data streams tries to balance communication time under all streams – especially if the amount of communicated data is similar, this scheme improves performance.

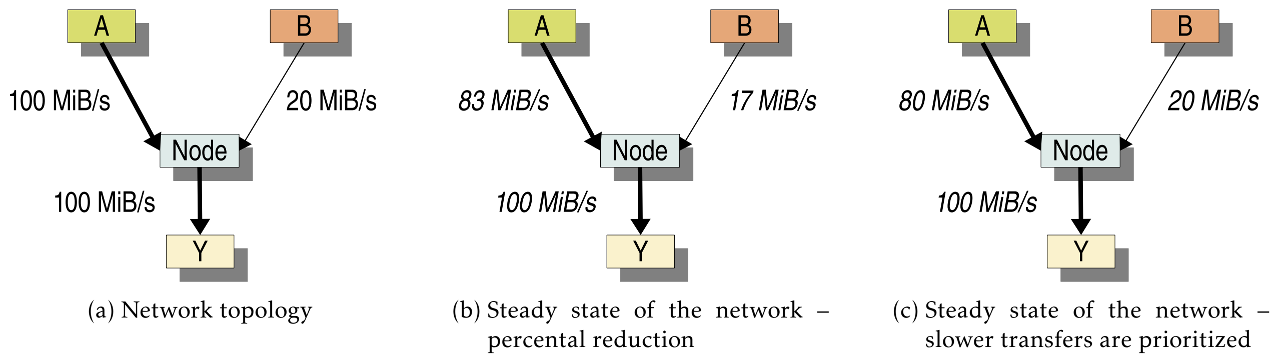

A
B
100 MiB/s
20 MiB/s
Node
100 MiB/s
Y
(a) Network topology
83 MiB/s
17 MiB/s
100 MiB/s
(b) Steady state of the network – percental reduction
80 MiB/s
20 MiB/s
100 MiB/s
(c) Steady state of the network – slower transfers are prioritized

Figure 5.7.: Network congestion and prioritization of data streams based on a simple example. One intermediate node (switch) connects the two sources (A, B) to the destination node (Y). Speed of the links is 100 MiB/s (thick line) or 20 MiB/s (thin line). In the steady state the effective throughput is annotated.

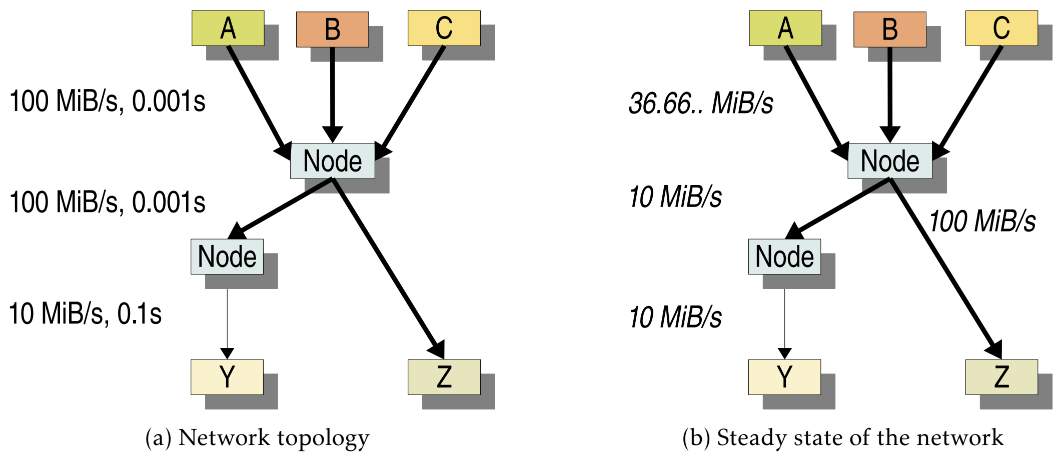


Figure 5.8.: Propagation of network congestion. Two intermediate switches connect the sources (A, B, C) to the destination nodes (Y, Z). Speed of the links is 100 MiB/s (thick line) or 10 MiB/s (thin line). In the steady state the effective throughput is annotated.

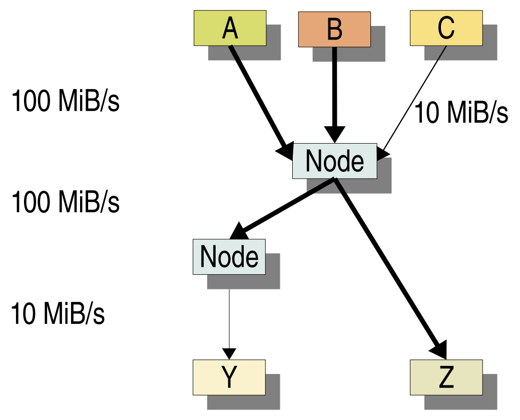
the algorithm must distribute performance equally – leading to a performance of 50 MiB/s for both edges. This is actually done by the developed flow model.

Propagation of network congestion Assume three source nodes (A, B and C) transmit data to two destination nodes (Y and Z); all sources try to send a very high number of packets to both targets at the same time. All edges except the one to Y have a characteristic throughput of 100 MiB/s and a latency of 0.001 s; the edge to node Y has less throughput and a higher latency. For the moment the processing speed of all intermediate nodes is considered to be infinite. This configuration is shown in Figure 5.8a.

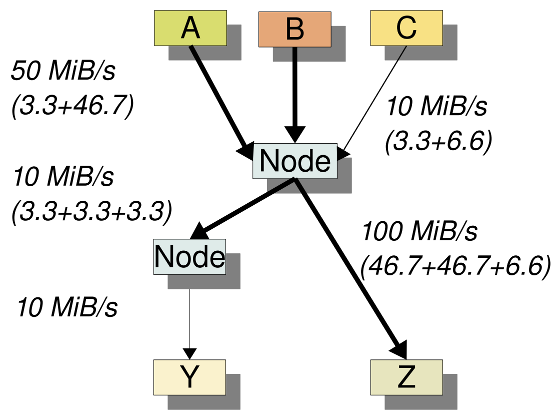
The steady state of the network is illustrated in Figure 5.8b[10]. The illustrated transfer speed is optimal and fair for all streams. Further, the picture looks realistic to the observer.

A network model that provides unlimited buffers, leads to a configuration in which all sources send data at 100 MiB/s; packets would queue up on the intermediate nodes. With a limited buffer size some throughput would be wasted due to retransmission of lost packets. Further, such a model must implement complex schemes to detect and to treat packet loss.

[10]Note that the flow for steady networks can be computed by using max flow algorithms.



(a) Network topology



(b) Steady state of the network – slower transfers are prioritized

Figure 5.9.: Propagation of network congestion with the flow concept. Two intermediate switches connect the sources (A, B, C) to the target nodes (Y, Z). Speed of the links is 100 MiB/s (thick line) or 10 MiB/s (thin line). In the steady state the effective throughput is annotated.

Propagation of network congestion with multiple bottlenecks In this thought experiment the topology illustrated in Figure 5.8a is slightly modified – one edge of a source node now transfers data at a speed of 10 MiB/s. The modified topology is shown in Figure 5.9a.

A steady state with prioritized slower transfers is given in Figure 5.9b. In the figure a possible throughput distribution for the individual data transfers is provided, for example node C distributes the maximum throughput for transfers to node Y and Z into $3\frac{1}{3}$ MiB/s and $6\frac{2}{3}$ MiB/s, respectively. While the maximum throughput can be determined with a maximum flow algorithm, the distribution of capacity to the individual data transfers is undetermined and a complex policy decision.

The advantage of the priority scheme over a percental sharing will become clear, if we assume all nodes just transfer data to node Y. In such a configuration the available performance of the intermediate edge (10 MiB/s) is largely overcommited because 210 MiB/s can be delivered by the incoming edges. Thus, the edge is utilized by 2100% and, therefore, with a percental sharing, each node is assigned 4.76% of the potential link performance. Thus, performance of node A and B would be throttled to 4.76 MiB/s and the performance of node C to 0.476 MiB/s. By increasing the potential throughput of the edges for node A and B further (e.g., to 1 GiB/s), the ratio for the slow connection becomes even smaller. This almost starves communication of node C. With the priority scheme for slower edges all edges would operate at 3.3 MiB/s, which allows a continuous data transfer.

5.3.3. Flow Model

The designed flow model results in the steady configurations for prioritized data transfer as indicated in the examples. For this, every component in the data flow model maintains a local status of all *streams* to control data flow in an optimal way. A stream is the set of packets with the same sender and receiver, i.e., every pair of *source* and *destination* starts exactly one stream. Basically, a stream owns a *virtual channel* through which data is transferred. Packets in a stream are processed with an FCFS strategy, thus they arrive in the order they are sent[11]. This concept works without packet loss (and without re-transmission of packets).

The maximum number of packets of a stream allowed to be simultaneously in flight is limited by the maximum amount of data which can be send until the target component receives it; a network link should

[11]If alternative network paths exist in the graph and load balancing is done by the routing algorithm, then this condition might be violated.

be saturated with the packets if possible. In essence the amount of data that could be transferred over a single link is defined by the *bandwidth-delay-product*. Also, to fully utilize a link, the time for a component to transmit a local packet to the next component, that is, its processing time for the packet, must be considered.

Therefore, for every stream a component maintains the sum of the processing time of packets that are currently in flight to the next component. This time covers the latency of the connected edge and the processing time of the sending component. When a job is about the be started, the currently pending latency of the stream is compared with the component's bandwidth-delay-product: $latency + \text{maxProcessingTime}$. If the pending latency is higher than the bandwidth-delay-product then the current job is processed but further transfer to the target is stalled [12]. The maximum processing time is estimated using the maximum packet size (which equals the global network granularity) and the component's throughput. The processing time of a component is bound by the *maximum processing time* of that component. Overestimation of the maximum processing time causes more packets in flight. Note, that the size of the packets is limited by the *network granularity*.

When a component cannot process incoming packets quickly, the amount of data in flight increases on the source component and all connected components stop data transfer to the saturated component. The component still receives the data, thus it must buffer these packets on the input port. The amount of data received per stream is in the order of the bandwidth-delay-product of the incoming link that is connected to the input port. At most one stream is established for every pair of source and destination[13].

While packets travel from source to target, such slower components on the communication path delay packet acceptance, which propagates towards the source. Ultimately, the congestion reduces network transmission upstream as suggested in Figure 5.8b. Whenever data is processed on a saturated component, the transmission of one blocked incoming data flow is continued. If multiple incoming edges deliver data to a node which is currently saturated, this node will multiplex its capabilities among all incoming streams (the flow to continue is selected in round-robin fashion).

Packets received on a NIC are removed from the network, which usually means the data is stored somewhere in memory. Network cards can be instructed to stall reception of further data. This is used, when the cache of an I/O server is filled up because further data could not be stored in memory. Thereby, I/O servers participate in the network flow protocol.

5.3.4. Illustration of Data Flow

Several facets of the flow model are discussed for the example of propagating network congestion that is illustrated in Figure 5.8. In this example the bandwidth-delay-product is 100 KiB for the fast network links and 1 MiB for the slower network link. A network granularity of 100 KiB is assumed in the following.

Steady-state in the example A snapshot of the links and buffers for the steady state of the network is visualized in Figure 5.10.

The behavior of node A, node B, node C is the same, for simplicity the discussion focuses on node A: Two packets, one for target Z and one for Y, would be queued on node A. One packet from node A would be on the outgoing edge from node A – this could be either a packet to target Z or one for target Y. The central node buffers most packets, for each target one queue is maintained that contains one packet from each source node. Both outgoing edges transfer one packet at a time, the packet is from any of the sources; packets of all sources are transferred in round-robin. The intermediate node on the path to node Y buffers just one packet. Due to the higher capacity of the link to Y exactly 10 packets are transferred at a given time to the destination. They can be thought of to be in-flight. Due to the fair strategy, three are from each

[12]The job must be started to ensure that the packets in flight fill the "pipe" towards the next component completely.

[13]Therefore, in a real system the packet flow scheme would require a large, but limited buffer space – in the order of $O(N^2)$, where N is the number of nodes in the network graph. The coefficient depends on the latency and packet size.

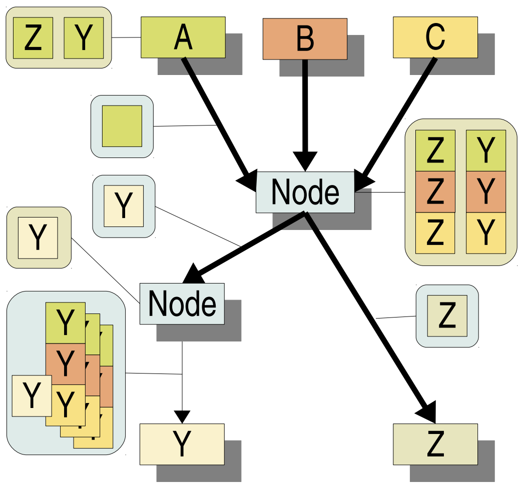


Figure 5.10.: Steady state of the network that is computed with the flow scheme. Buffered packets on the nodes and on the wires are indicated, colors encode the source node and letters the target. In case the source node is unknown, the color of the target node is taken.

source and the last could be from any of them. As soon as one packet is received by a component another one is transferred towards the sink; this propagates upwards the stream.

The example assumed unlimited fast processing on intermediate nodes, to account for the processing time on the components. One additional packet is buffered on every component in the implementation.

Component utilization The processing of the packets on the edges is illustrated in Figure 5.11 – the left column displays the edges and each row highlights the packet transport over this edge. Processing inside nodes is omitted in this figure.

In this visualization packets with the same source and destination are depicted with the same color, for instance the blue color indicates packets flowing from node A to node Y. Bottleneck of this configuration are edges towards Y and Z; they process packets all the time. Due to the lower throughput, Y takes 10 times longer to relay a packet. A steady flow of packets occurs, yet, there is plenty of idle time for the central edge that connects the intermediate nodes. Multiplexing between the two targets causes the source nodes to send one packet to Y for every 10 packets transferred to Z. Behavior of this figure matches the throughput observed in Figure 5.8b.

The wakeup and propagation mechanism can be illustrated with this figure, too. Look at the completion of a packet on the edge to Y: Once a new packet starts, to be processed the edge that leads to the node transmits another packet. Once this process starts the center node determines one of the three streams in a round-robin manner, and a packet of that stream is transferred towards Y.

Flow control in a single stream To demonstrate how the steady state of the network is achieved, the data flow of a single stream is discussed in this thought experiment: Assume only node A transfers packets to node Y. The startup phase of a transmission of 10 packets (100 KiB each) is illustrated in Figure 5.12. In this figure the processing on intermediate nodes, which process data quickly, is included, too.

At the beginning of the transfer, node A manages to transmit data at full speed. It is throttled only for the last two packets. This late throttling is caused by the fact that transfer is resumed once the next component starts to process the packet and the edges have a latency equal to the processing time of one packet. To ensure seamless packet transfer, that means packet transfer without interrupts that utilizes the full network performance, processing time must be taken into account. Consequently, in this experiment edges are actually permitted to transfer one packet to cover for the latency, and one additional packet can be processed on a component to cover the processing time of the next component towards the destination.

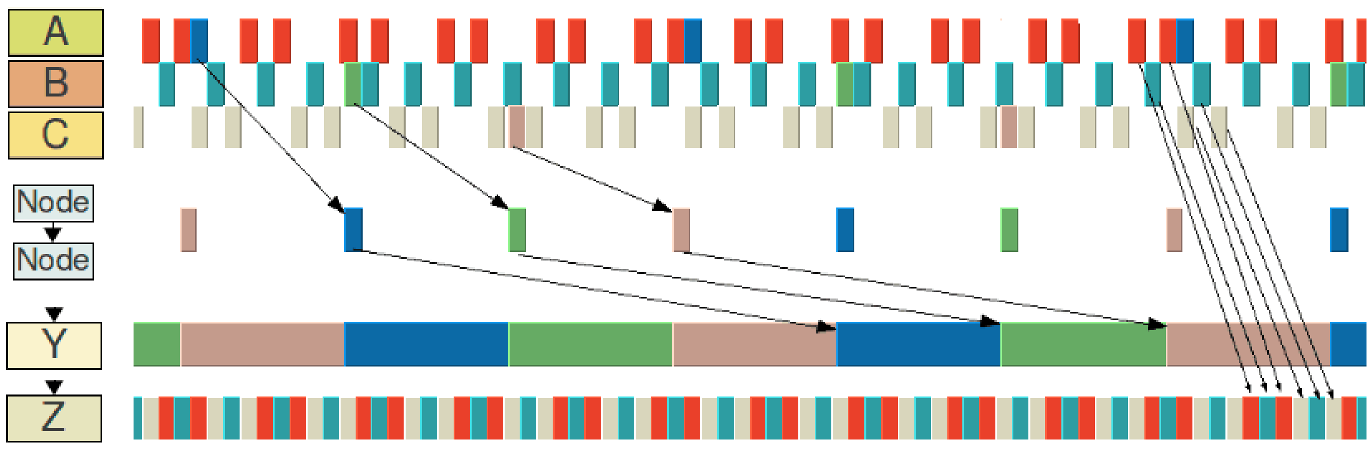


Figure 5.11.: Steady state of the network – the processing of packets on the edges from and to the nodes is indicated. Arrows illustrate the transport of individual packets.

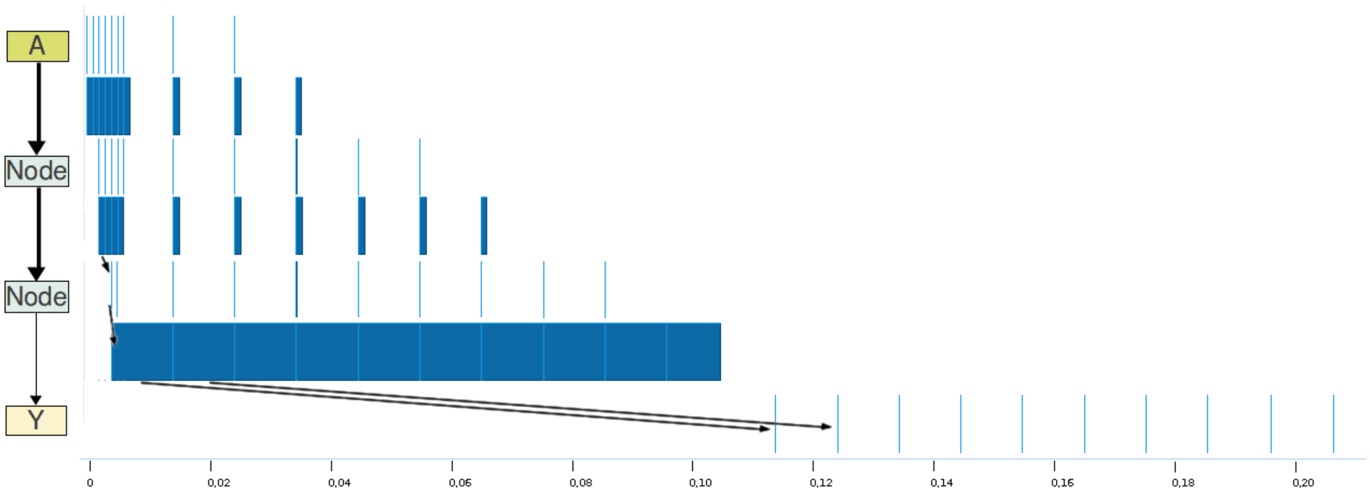


Figure 5.12.: Startup phase of a single transmission from node A to node Y. The processing of packets on nodes and edges is illustrated – latency defers reception on node Y. Arrows demonstrate the flow of packets.

In contrast to the last figure, in this display the latency of the slow link to node Y can be seen – the first packet arrives on node Y after the incoming edge finished to transfer all packets. At time 0.11 s all packets are in-flight.

5.4. Software Model

Important aspects of models for software components are described in this section: In Section 5.4.1 the execution model of parallel applications is introduced and demonstrated on an `MPI_Allreduce()` implementation. The inter-process communication with message exchange is presented in Section 5.4.2. Processing of the parallel file system and the client-server communication protocol is the focus of Section 5.4.3. An abstract description of the interplay between server, cache layer and block device is provided in Section 5.4.4. Currently implemented cache layers are introduced in Section 5.4.5. The implementation and processing of these models is clarified with small examples.

5.4.1. Execution of Parallel Applications

PIOsimHD is designed to simulate programs utilizing the MPI standard for communication. This includes support for asynchronous communication and collectives defined by MPI-3.

The *client* process, which is hosted on a *node*, executes a predefined sequence of *commands*. A command is of a certain type, for example, an MPI function call or any other imaginable individual or collective operation. Commands further contain parameters for the execution: For each command type a class provided in PIOsimHD-Model defines the parameters which are supported during the later execution. An instance of a command type will just be referred to as a *command*. When the model is built the sequence of commands and their parameters are fixed. During the simulation this sequence is processed sequentially without branching or other control structures. To simulate execution of a command at least one implementation must be provided for a command within the simulator – currently the most important MPI functions are implemented.

Internally, the implementation of a potential client *command* is programmed as a state machine. The discussed logic and processing of state machines is performed by the implementation of client processes – refer to Section 5.2.1. In each step of the command one of the following operations can be triggered by the state: Blocking network operations can be initiated, a child command can be started – that is, another MPI alike operation, or the state machine is blocked until another process restarts it manually.

Once all issued operations of the state complete, the client state machine proceeds to the next step in the state machine until the command is finished. To simulate calculation and CPU utilization a computation job is executed before the operations of a step are executed; the number of instructions to process is set by the state machine. Note, that the simulator requires that a step of a state machine must take some time – with the current implementation of Epoch 1 ns must pass. This is important for the visualization, as immediate completion of states is hard to see in the timeline, especially when a sequence of multiple operations completes immediately.

By invoking child commands existing commands can be reused. Imagine a dynamic broadcast implementation which chooses existing MPI implementations for the broadcast based on the given parameters and the cluster model. States can also spawn multiple child state machines, that means other commands. With this capability, for example, a command can initiate concurrent data transport between a process and multiple endpoints.

Upon execution the simulator keeps only one instance for each implemented client command, all invocations of the command are executed by this instance. Thus, a particular MPI function has the *global view* of all clients calling the same operation in the simulation. This global world view, for instance, allows implementing `MPI_Barrier()` without network communication at all: Clients block when they reach the barrier; when the last client invokes the barrier it activates processing of all pending callees.

Multiple implementations for a given MPI function can be programmed and selected in the model specification upon simulator execution. Thus, available MPI implementations can be evaluated against each other, or the implementation can be selected which matches the characteristics of the machine best.

When an operation is implemented in the simulator it is important to ensure proper operation. As the simulator does not send real data, the verification of the communication algorithm itself must be done by the modeler.

Processing of state machines illustrated for an `MPI_Allreduce()` implementation To illustrate the execution of state machines consider a naive implementation for `MPI_Allreduce()` in which processes transfer their data to a root process which reduces the result and transfers it back to the other processes.

A regular MPI implementation cannot allow sending of an arbitrary amount of data before the receiver is ready, because memory and, therewith, buffer space is limited. If a communication partner is ready, data can be stored in the final memory region. Hence, in a realistic implementation the receiving process sends

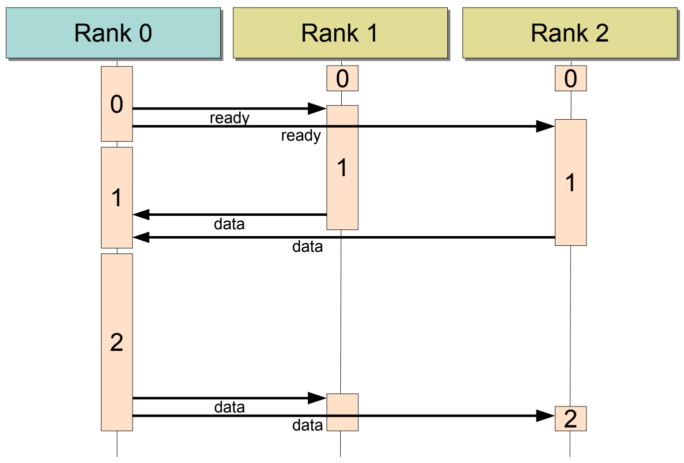


Figure 5.13.: Interaction for an `MPI_Allreduce()` implementation in which processes communicate with the root process – illustrated for three processes.

a message to the sender to indicate that it is ready, before a large data transfer is initiated. This scheme is included in the implementation model which is discussed in the following.

The interaction diagram in Figure 5.13 visualizes the states and interaction between three processes. In State 0 the siblings announce to receive the ready message. Rank 0 announces to send the ready messages in its first state. A process is suspended until all initiated network operations are completed, then it transits to the next state that is programmed. Between states programmed operations are executed. In this case, all processes will transit to State 1 when communication is finished: Once the ready message arrives the siblings proceed to State 1 in which data is transferred to the root process, also a job to receive the reduction result is posted at the same time. The root process computes the reduction result in State 2, and initiates sending of the result to both siblings. They transit to State 2 which indicates that data has been received. The Java implementation of the reduce operation is described in detail on Page 268.

A simulation run of this allreduce implementation is visualized in Sunshot in Figure 5.14. In this figure client activity is traced on three clients and the processing time on Rank 0. Concurrent network jobs are shown in separate timelines due to the relation concept of HDTrace. Send and receive operations contain the name of the client component and the unique identifier. The topology viewer shown on the left lists the modeled cluster components and nodes. More details about assessing simulation results are given in Section 5.5.3.

The `MPI_Allreduce()` could also be implemented by invoking an `MPI_Reduce()` followed by an `MPI_Bcast()`. By starting nested MPI calls existing implementations are re-used. The simulator permits a developer to select a specific implementation for a command, or to rely on the default implementation. In the default implementation the one picked by the run-time configuration is chosen (see Section 6.3 for more information).

5.4.2. Inter-process Communication

In short, the model of the communication is similar to buffered point-to-point message exchange in MPI. Inter-process communication is initiated by the state machine on a NIC. A *network job* either sends a message from the local NIC to a remote NIC, or it starts receiving of a message. Data flow of the created packets is controlled by the NIC as described in Section 5.3.

The typical addressing is done in the fashion of MPI: an envelope characterizes the posted messages, and the receiver waits for a message with an envelope that matches certain criteria; these *message matching criteria* describe the communication partner and matching conditions. Most criteria assist in matching of messages required for proper MPI communication. A receive operation completes, when the last packet of

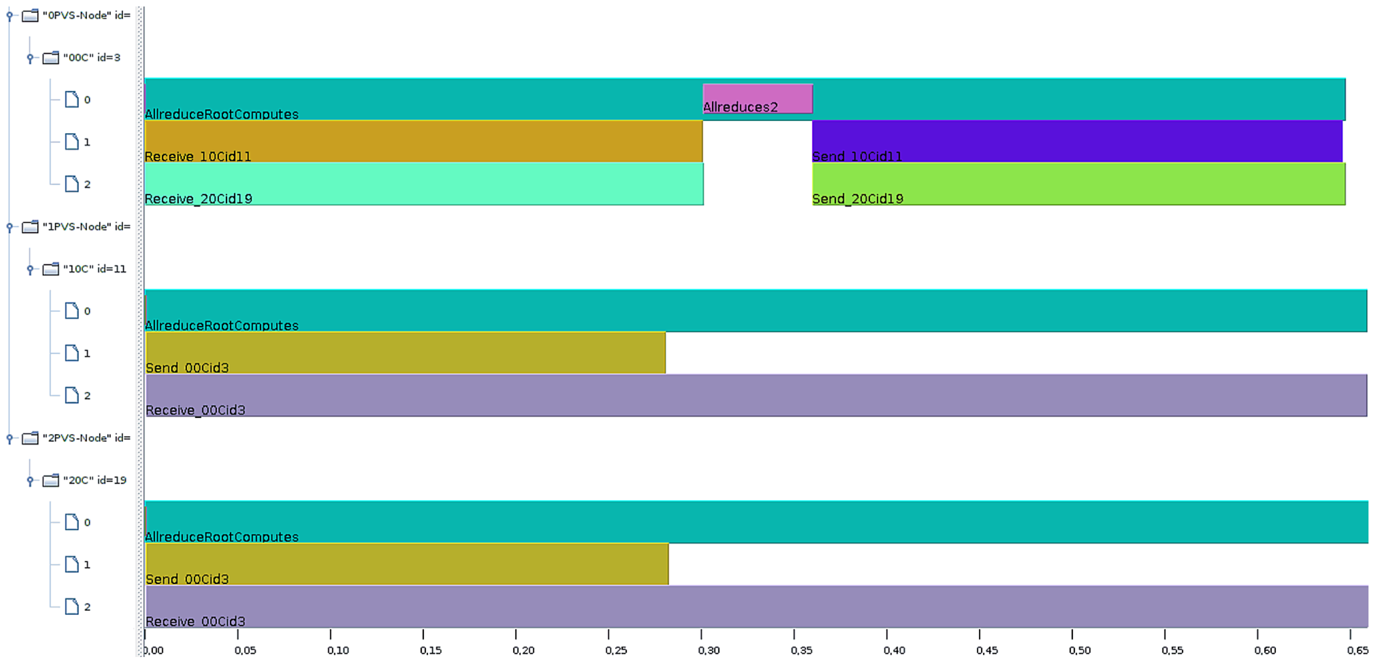


Figure 5.14.: Simulated client activity for an `MPI_Allreduce()` implementation that transfers 10 MiB of data. In this naive implementation all processes transfer their data to the root process which computes the results and broadcasts it to all other processes.

a matching message arrives. If no receive is posted when a message arrives, the message is buffered on the receiver-side until a receive with a matching criteria is posted. When a receive is posted that matches the criteria of a previously received message, the receive will complete immediately.

Sends complete when the NIC transmitted all data packets to the wire. Consequently, in a real application it would be safe to reuse potential data buffers. Since fragmentation of the NIC interleaves packets of independent jobs, concurrent transmission of all messages is guaranteed. Therefore, large messages do not block small jobs.

Messages can piggyback arbitrary user data, they have a total size (number of bytes on the wire) and contain the source and target node. In a real system, protocols add a header that describes the packet content, and which controls network parameters. PIOsimHD can add an overhead representing packet or message headers. This overhead can be added per packet or per message. By default, the overhead is 40 bytes which is the minimal TCP header (20 bytes) and IP header (20 bytes). Currently, the additional MPI header is not considered explicitly, but it could be incorporated into the simulator easily.

Usually, all data is ready when a message is initiated and can be transferred automatically by the NIC. However, PIOsimHD can also create an empty message and append data on the fly until all data is transmitted. In this approach available data is transferred on demand, similar to a TCP stream. This capability is used in the I/O path to start a message before all data is read or written[14].

Message matching Message matching criteria rely on the following attributes contained in the message envelope:

- **Source**: The *NodeHostedProcess* data is send from. On the receiver side the source indicates from which process data shall be received from. If it is not set, data is accepted from any source.

[14]With this concept it is possible to evaluate concepts of MPI communication which can be set up before all computed data is available. Currently, the MPI standard just permits non-blocking I/O of memory regions; these cannot be modified while the communication happens. Theoretically, a process could initiate many small messages to allow fine grained parallelism. However, invoking an MPI call is time-consuming. Therefore, overlapping of communication and computation is typically coarse grained. With the simulation the benefit of fine grained communication can be evaluated.

- **Communicator**: An instance of an MPI communicator – this field can be considered as a communicator context.
- **Tag**: A numeric identification which distinguishes messages from one another. A receiving job can also permit to receive from messages with any tag.
- **CommandImplementation**: To avoid false matching of messages between independent command implementations, the command implementation that invoked the network operation is contained in the envelope. Since operations can be nested, the root command implementation is stored in the envelope, too. The CommandImplementation can be considered to act like an MPI communicator context.

To permit reception, the envelope of the received message must match the conditions specified in the posted receive. It is also possible to receive all messages from the network – this capability is used by I/O servers to receive the initial requests (see the description of the client-server communication below).

If the network topology offers a single route between sender and destination, the nature of the flow protocol will ensure that messages which are transferred arrive in the order they are started[15].

Preventing false matching of messages The two *CommandImplementation* fields are automatically filled with the implementation of an operation to avoid wrong matching. An example will clarify this issue, let us take our `MPI_Allreduce()` implementation which invokes `MPI_Reduce()` on the root process and then `MPI_Bcast()` to transfer the outcome to all siblings.

A wrong matching is illustrated based on the code snippet in Listing 5.1. Executed with more than one process this code should produce a deadlock because none of the collective calls can terminate. Since the implementation of `MPI_Allreduce()` in the simulator relies on nested functions that invoke the two collective calls that are called from all other processes, the C code snippet will execute and terminate. In this case, the computation of the correct result will happen, however this is pure luck and works only when the `MPI_Allreduce()` call is implemented that way.

In general, without the CommandImplementation criterion, messages from one collective operation could be received by another operation, hence, the outcome would be unpredictable and probably wrong. Therefore, PIOsimHD requires the root implementation to match allowing nested operations to match if and only if the parent implementation is identical. In most cases it is desired that an implementation of an operation exchanges data only with itself and thus the current implementation must match. Consequently, this criterion ensures that non-blocking collective operations will behave correctly as long as at most one operation of a given collective is started with the same communicator.

There are a few exception to the strict matching requirement: Client-server communication and matching of messages between `MPI_Send()`, `MPI_Recv()` and `MPI_Sendrecv()`. To allow this, the *CommandImplementation* can be overridden when a network job is submitted.

Listing 5.1: Code snippet which illustrates a wrong matching of messages from two MPI collectives by the `MPI_Allreduce()` implementation. On a correct MPI implementation, this code should deadlock.

```
Initialize MPI and buffers correctly

if( rank == 0 ){
  MPI_Allreduce(& sum, & sumTotal, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
}

if ( rank != 0 ){
  # With the MPI_Allreduce() implementation that invokes MPI_Reduce()
  #  and MPI_Bcast() the code will terminate.
  MPI_Reduce(& sum, & sumTotal, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

[15]When multiple routes are used by the routing algorithm, the order might change, though.

```
    MPI_Bcast(        & sumTotal, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  }
```

5.4.3. Parallel I/O

An abstract parallel file system defines how client-server interaction takes place. While inspired by the operation of the PVFS client-server protocol, the abstraction layer of this interface is generic enough to approximate the I/O path behavior of many parallel file systems: File data is partitioned among all servers as defined by a selectable distribution function. In the default configuration data is distributed in round-robin fashion across all servers. Fault-tolerance mechanisms and data replication are not considered in the model.

Metadata operations are not considered in the client-server communication as the communication protocol for metadata is heavily dependent on the underlying parallel file system. However, file sizes are maintained as a global attribute for a file. This global attribute is adjusted (enlarged) when a write beyond the current end-of-file completes.

Client-server communication protocol In the model clients and servers interact similar to PVFS, which is an archetype for accessing raw data. The server provides an implementation (a so-called processor) for each type of request, which is either: read, write or flush. A server listens to new requests and forwards them to the processor responsible for this type of request, allowing new operations to be added easily. Non-contiguous I/O requests in form of ListIO are explicitly supported. Network transfer and I/O are pipelined for reads and writes. Thus, data transfer and I/O can happen concurrently.

Client-server interaction, the internal processing on their NIC's, and the operations performed on server-side are shown in Figure 5.15 and in Figure 5.16.

In the read path a client requests data from all servers holding necessary data and waits until data has been received. In detail, the following steps are performed[16]:

1. An `MPI_File_*read*()` invocation first checks the file size and removes accesses to non-existing file areas, that means extents bigger than the file size are not transferred to a server.
2. Supported by the distribution function, the client translates all read accesses to the physical addresses and the servers. For every server that holds parts of the required data one request is created.
3. For each participating server, the request and the receive for the expected response (the data) are posted. All receives are processed concurrently. The client is stalled until all operations complete. When data has been received, the read call will complete (see Step 10).
4. The network forwards a request to an involved server, which invokes the `process()` method of the read processor.
5. The request processor invokes `initiateSend()` on the NIC to create a response message that will hold all data. At beginning this message is empty, but it will grow continuously while data is read from disk.
6. Arrival of the I/O request is also announced to the cache layer which causes the block device to fetch the required data.
7. Once some data has been fetched from the block device, the cache layer calls `readPartitialData()` on the request processor to announce this fact. The processor, in turn, appends the fetched data to the pre-created message.

[16]Due to the complexity of the field, caching strategies are discussed on Page 242.

8. The network interface can transmit parts of the received data before all reads are completed. Data is partitioned into packets according to the network granularity; once a packet is full it is sent as soon as possible.
9. Whenever a packet leaves the NIC, a callback of the request processor (`messagePartSendCB()`) is invoked. Basically, this tells the cache layer that buffering of this data is not necessary any more.
10. Upon completion of one pending server request, the NIC invokes the callback `recvCompletedCB()`. The client process implementation ensures that the client state machine defers further execution until all outstanding receives are completed.

In the write path, a client sends requests to all involved server, and waits for acknowledgements. In detail, the client-server interaction works like this:

1. An invocation of an `MPI_File_*write*()` derivate creates requests to all the involved servers. Data is partitioned with the distribution function – similar to the read operation.
2. Requests are transferred to the servers, which start the appropriate request processor.
3. The request processor announces the write request to the cache layer and starts a job to receive data from the client.
4. The client sends the I/O data to the servers and waits for an acknowledge from the server that the data has been received completely (see Step 10).
5. Data is partitioned into packets (according to the network granularity) by the NIC and transferred to the target servers.
6. Upon reception of a packet the server informs the write processor about this condition by invoking `messagePartReceivedCB()`. The write processor asks the cache layer if the data fits in the cache, if it does, the data is written into the cache.
7. Later in the process it might happen that the block device cannot store data quick enough and thus the cache fills up. If that happens, then arrival of a new message part causes the write processor to call `blockFurtherRecv()` on the network card. This call forces the network flow protocol to deny reception of further packets, due to the flow protocol, ultimately the network transport to the server stalls. Note, that the NIC does not distinguish between data or other requests and thus no new requests are received on the server as long as the NIC is blocked.
8. Upon completion of a write operation the cache layer calls `writePartialData()`. This checks if one of the pending and already received data fragments fit into the cache. If so, it will be written to the cache. If all pending write operations completed, packet transfer is enabled again by calling `unblockFurtherRecv()`. This check is omitted if data transfer is not stalled.
9. Submission of the last packet of a data transfer into the cache causes the request processor to forge an acknowledge message for the client. Consequently, from the client perspective, the write request may complete before all data is actually persisted. In case the client wants to enforce persistent data, it must submit a flush operation after the write.
10. When the client has received the acknowledgements from all involved servers, it updates the size of the file if required.

The flush operation is a simple remote procedure call: The client creates a flush request which is executed by the flush request processor. By calling `announceIOrequest()`, the operation is given to the cache layer. When the cache layer completes processing of the flush, it notifies the processor. The processor, in turn, posts an acknowledge message for the client.

I/O forwarding It is possible to add I/O forwarders to the communication path. On the client, a forwarding process can be defined that relays communication for at least one server. All data between this particular client and server is then routed via the forwarder. In detail, a forwarder accepts incoming messages. Upon arrival of a (request) message a new message is created; for every packet that arrives, data is

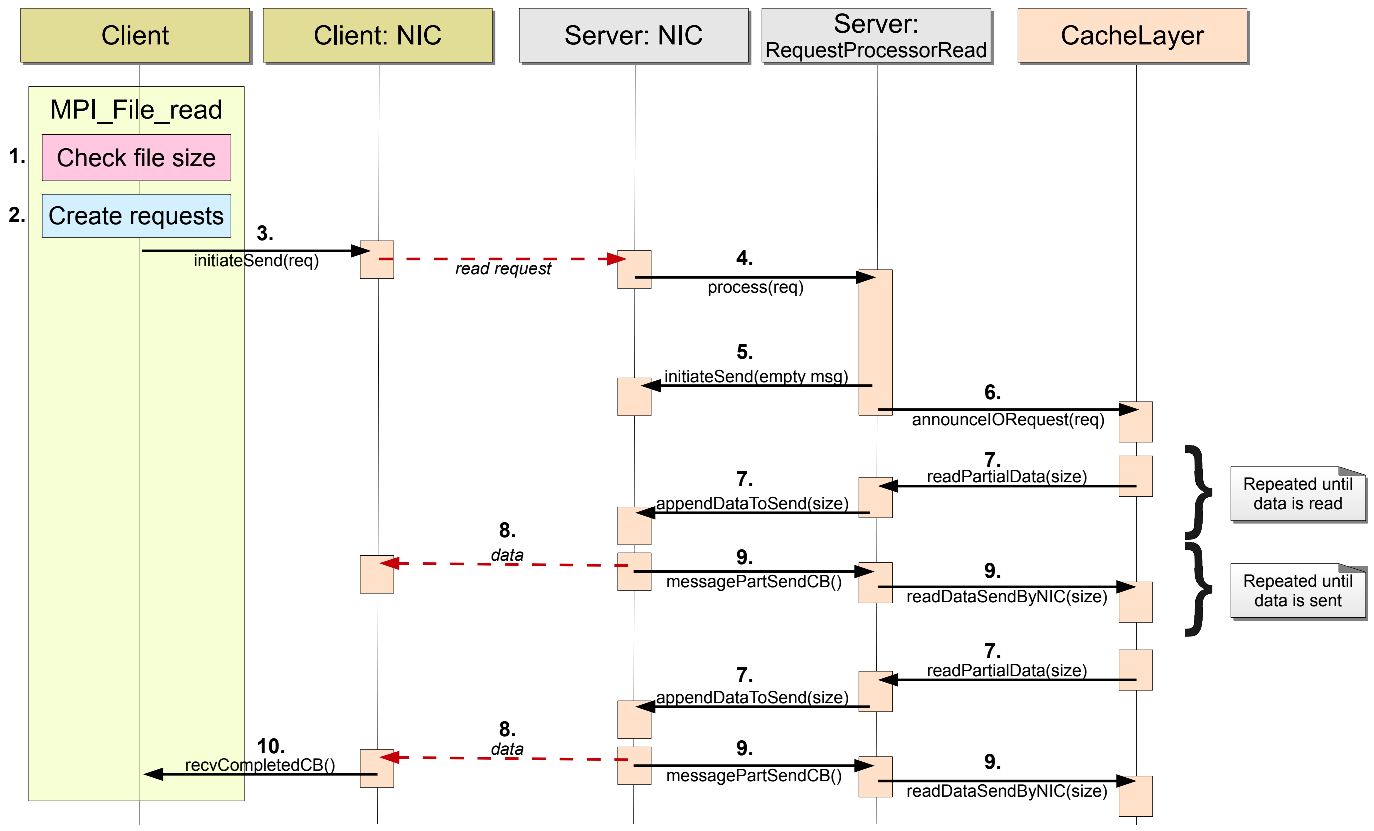


Figure 5.15.: Illustration of the client-server communication protocol – read path.

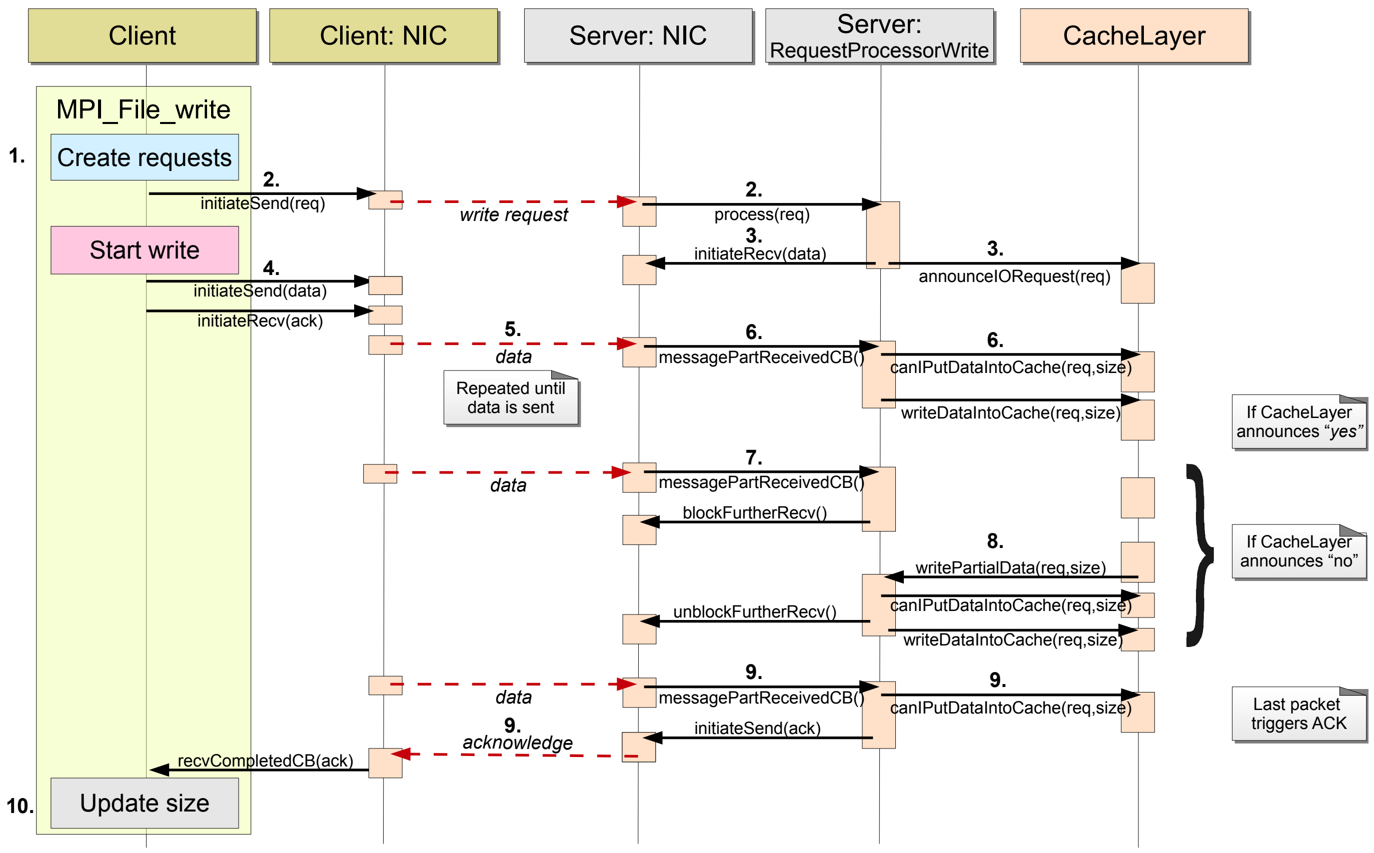

Client
Client: NIC
Server: NIC
Server: RequestProcessorWrite
CacheLayer
MPI_File_write
1. Create requests
2. initiateSend(req)
write request
2. process(req)
3. initiateRecv(data)
3. announceIORequest(req)
Start write
4. initiateSend(data)
initiateRecv(ack)
5. data
Repeated until data is sent
6. messagePartReceivedCB()
6. canIPutDataIntoCache(req,size)
writeDataIntoCache(req,size)
If CacheLayer announces "yes"
data
7. messagePartReceivedCB()
blockFurtherRecv()
8. writePartialData(req,size)
canIPutDataIntoCache(req,size)
unblockFurtherRecv()
writeDataIntoCache(req,size)
If CacheLayer announces "no"
data
9. messagePartReceivedCB()
9. canIPutDataIntoCache(req,size)
initiateSend(ack)
9. acknowledge
Last packet triggers ACK
recvCompletedCB(ack)
10. Update size

Figure 5.16.: Illustration of the client-server communication protocol – write path.

appended to this message until data transfer completes. Forwarders can form a hierarchy to relay data via additional forwarders, if desired. The current implementation of the I/O forwarder does not offer sophisticated optimization techniques, instead each request and response is simply forwarded between client and server.

Modeling MPI calls Variants of the `MPI_File_open()` and `MPI_File_close()` commands are implemented which simulate the fact that on a real system, clients may exchange file information. Both implementations call `MPI_Barrier()` to synchronize the clients. The *BroadcastOpen* implementation simulates the behavior in which one client has gathered the metadata for the file and broadcasts them to all other processes which reduces load on metadata servers – such an implementation can be found in PVFS, for example. One implementation of the `MPI_File_close()` operation flushes the file data when it is invoked. Processing of this implementation in detail: All processes synchronize by calling `MPI_Barrier()`, then Rank 0 sends a synchronize call to all servers; finally, all processes synchronize with a barrier – now all data is flushed to disk.

Client-side collective I/O with the two-phase protocol is modeled, too[17]. By default the implementation uses 16 MiB of buffer size per client (like MPICH2). The two-phase implementation and a variant of it is compared to server-side optimizations in [KKL12].

5.4.4. Interaction of Server Cache and Block Device

Network communication between client and server transfers requests and data to the cache layer, which steers the block device and cache usage in order to optimize performance[18]. First, the orchestration between cache layer, network and block device is described, then the implemented models of the cache layers are clarified.

Typical file systems do not announce all non-contiguous requests to the block device. This effectively prevents certain optimizations, especially aggregation within the cache layer. Therefore, with the designed abstract file system, all information is propagated as soon as possible.

The read path is illustrated in Figure 5.17 and consists of the following steps:

1. The read processor of the server announces the complete non-contiguous request, that is, the ListIO, to the cache layer.
2. Extents which have to be read are added to a *job queue* that manages accesses for all read operations across requests and files.
3. Whenever the block device is not fully utilized, that means the number of currently scheduled requests is below *maxConcurrentIOOps*, another job is initiated. The job to schedule is selected from the list of pending jobs in the job queue and it is transferred to the block device by invoking `startNewIO()`. Read jobs are preferred over write jobs because an application waiting for data cannot proceed[19]. However, to perform an operation enough cache space must be available: It is checked whether the node has enough free memory to buffer the data to fetch. If so, the memory is reserved and the read job is executed[20]. Otherwise, an available write job is scheduled to free more memory.
4. Completion of an I/O job fires a callback in the cache layer; upon invocation the cache layer notifies all requests for which data has been read, this in turn appends the data to their messages.
5. Whenever read data has been sent by the NIC, used memory on the node is freed.

[17] Refer to Page 55 for a description of two-phase.

[18] Refer to the performance factors mentioned in Section 2.2.5.

[19] The policy to prefer read over write operations is driven by the fact that a process usually requires the data read to continue its operation. However, writes can be deferred with caching strategies. The local file system ZFS prioritizes operations in this manner, too.

[20] Since the simulator is executed sequentially, a correct accounting of the memory is possible.

In contrast to the read path, in the write path a cache layer must wait for data to arrive on the NIC, hence, not all operations can be started immediately. As a consequence, read operations are not symmetric to write operations. The write path is visualized in Figure 5.18. It processes operations with the following steps:

1. The write processor of the server announces the complete ListIO to the cache layer – with this knowledge the cache layer can defer pending operations and might even wait for the data to build a better access pattern.
2. Whenever data arrives on the server, the request processor invokes the `canIPutDataIntoCache()` function to ask the cache layer to accept the incoming data, which usually queries the free memory on the node and tries to allocate free memory on the node. If there is not enough memory available, the server blocks reception of further packets[21].
3. Every packet fitting into the cache is submitted to the cache layer by the write processor. Extents that have been received are still undecided – this permits the cache layer to pick the best selection and assume the right data has been transmitted by some kind of hint from the server. Normally, file systems transmit data in the order the bytes are written by the non-contiguous access pattern, i.e., in sequential manner – the first extent, then the second extent and so forth.
   The currently implemented cache layers decide at this point which data they actually received and add I/O jobs to the job queue. While the cache layer can change the order, current implements do not exploit that feature. In most cases a reordering of write operations is possible later, too, due to the caching in memory on the server. When data is written to the cache layer, it also allocates the required buffer space on the node.
4. It is checked if another I/O job can be submitted to the block device with the same method as in the read path (see Step 3).
5. Upon completion of a write job, a callback in the cache layer is invoked. Similar to reads, for every request of which some data has been written by the I/O job this fact is propagated to the write processor. Memory space to buffer the data is immediately freed once the data has been written to the block device.

This general cache layer behavior is designed to permit aggressive (write) caching; a cache implementation might aggregate multiple high-level I/O operations into fewer operations on the block device. Data which is read is not cached for future reuse currently. Applications should avoid this kind of accesses and reuse already fetched data. Moreover, behavior of a read cache can be unpredictable due to interference of concurrent applications and, further, mechanisms such as read-ahead can even degrade performance.

Aggregation of multiple operations is limited by the *I/O granularity* which is the maximum size of data that can be accessed in a single block-device job. It defaults to 10 MiB and can be set at simulation time. Since data sent across the network is split into smaller chunks to accommodate the network granularity (default 100 KiB), network transfer and I/O access can be pipelined. To illustrate the pipelining scheme consider reading of 20 MiB data. Once the first 10 MiB have been read from block device, the fetched data can be sent to the requesting client.

A consequence of the implementations for the read and write path is that large accesses (bigger than I/O granularity) cause the cache layer to create read jobs in the size of the I/O granularity, but write jobs are limited to the size of network granularity. Consequently, if the cache layer does not buffer or defer write operations, the effective I/O granularity of write requests is limited by the network granularity.

5.4.5. Implemented Cache Layers

Implemented models of cache layers build on the abstract interaction between network, block device and cache layer. The class hierarchy of the currently available cache layers is illustrated in Figure 5.19.

[21]This has been discussed in Section 5.4.3

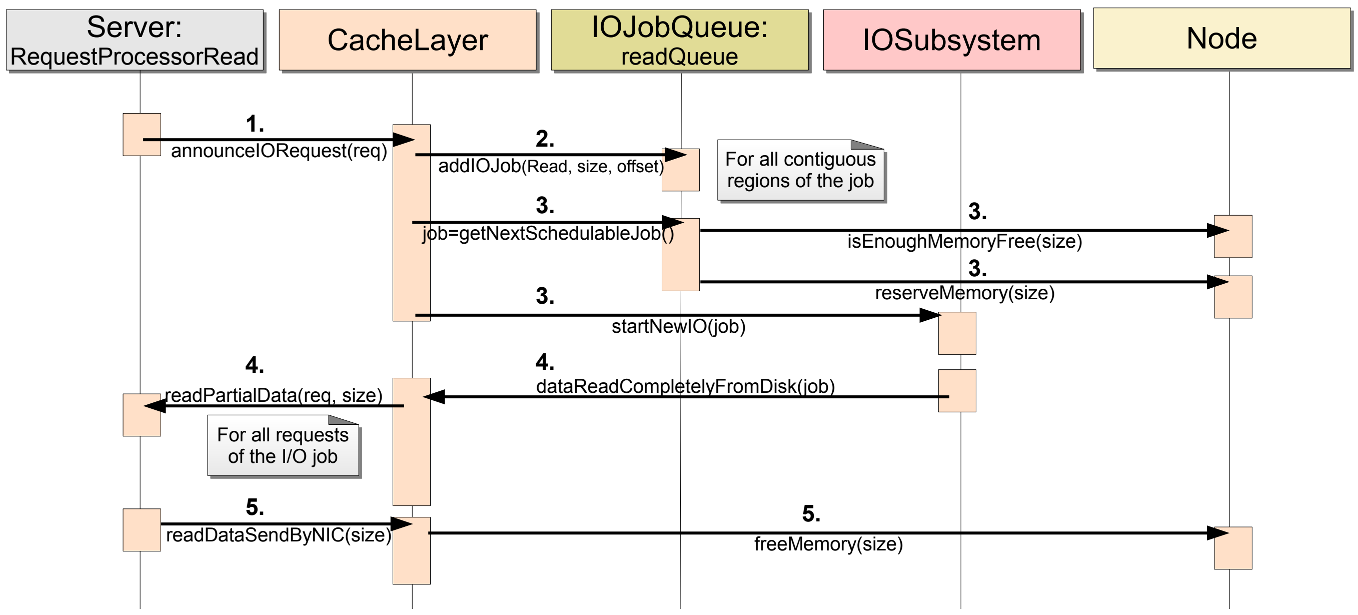


Figure 5.17.: Read path – illustration of the interaction between request processor, cache layer and the block device.

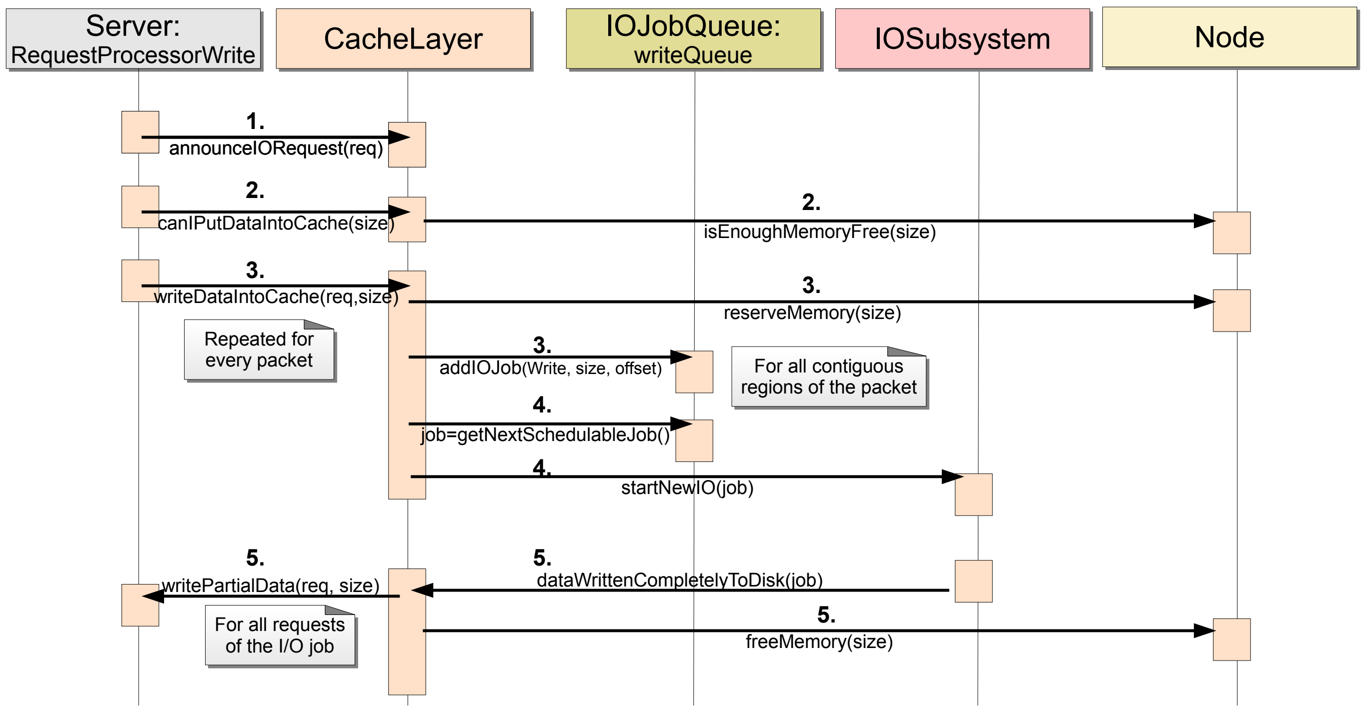

Server:
RequestProcessorWrite
CacheLayer
IOJobQueue:
writeQueue
IOSubsystem
Node
1.
announceIORequest(req)
2.
canIPutDataIntoCache(size)
2.
isEnoughMemoryFree(size)
3.
writeDataIntoCache(req,size)
3.
reserveMemory(size)
Repeated for every packet
3.
addIOJob(Write, size, offset)
For all contiguous regions of the packet
4.
job=getNextSchedulableJob()
4.
startNewIO(job)
5.
writePartialData(req, size)
5.
dataWrittenCompletelyToDisk(job)
For all requests of the I/O job
5.
freeMemory(size)

Figure 5.18.: Write path – illustration of the interaction between request processor, cache layer and the block device.

All cache layers are derived from *ServerCacheLayer*. The *maxConcurrentIOOps* attribute limits the maximum number of operations issued to the block device. To allow application of scheduling algorithms inside the block device this parameter must be set to a value larger than 1.

NoCache The *NoCache* implementation issues one I/O job operation after another on the block device with an FCFS strategy. Read operations are preferred over write operations – as long as any read operation is available write operations are stalled[22]. At most one write operation can be pending inside the cache. Consequently, write-behind is not possible with NoCache. Internally, this is done by returning false from `canIPutDataIntoCache()` when one write operation is queued or already executed by the cache layer (and block device). I/O operations are not aggregated in NoCache, therefore, the maximum operation size of write operations equals the network granularity. For read operations it is limited by the I/O granularity.

SimpleWriteBehindCache *SimpleWriteBehindCache* is a cache layer which permits write-behind by caching pending write operations in the memory of the node – as long as memory is available. The class extends the NoCache implementation slightly as most functionality of the cache layer is embedded in NoCache. Only the `canIPutDataIntoCache()` function is overridden to check for free memory on the node. It does not aggregate operations, thus the maximum I/O size for write operations is still bound by the network granularity.

AggregationCache The *AggregationCache* performs write-behind but also aggregates pending I/O operations into extents up to the size of the I/O granularity. Reads are still preferred over write operations. A fair mechanism processes the earliest issued operation (the oldest operation in cache). Thereby, it combines all operations of the same type which overlap with the extent of the selected operation. This step is repeated until no further combination is possible, or if the aggregated operation reaches the I/O granularity. Hence, a single block device operation can contain data from multiple requests – even from multiple clients. Overlapping areas in the accessed extent are handled by reading data once and distributing it to all requesters. Data from multiple writers to a region is discarded because only the newest data must be written. Compared to the mentioned *NoCache* and *SimpleWriteBehindCache*, the aggregation allows write operations with larger chunks than the network granularity.

However, the aggregation process can be quite time-consuming; the current implementation iterates over all pending operations and repeats this process when one of the operations can be aggregated. Therefore, the implementation has a worst-case complexity of $O(N^2)$, where N is the number of currently pending operations.

AggregationReorderCache The *AggregationReorderCache* extends the policy of the *AggregationCache* by allowing the cache to change the order in which operations are executed. Scheduling of pending operations is performed in the following way: Read operations and write operations are scheduled in phases, first all read operations are scheduled. After they complete, all write operations are processed and vice versa – new operations are queued up in the meantime.

When a type of operation is selected, files are processed sequentially, that means all operations of the selected type are executed for the first file, then for the next file, until all files are processed. Operations for each file are executed in the order of their offset and thus with an algorithm similar to the SCAN algorithm of disk drives.

Flush operations are combined in this cache and performed after write operations of the file completed and thus a flush operation could be deferred for later execution – still the synchronizing character is preserved.

[22]Since read operations are always synchronizing, i.e. they require the client to wait until data is fetched from the I/O subsystem, but written data can be buffered, this strategy usually improves performance.

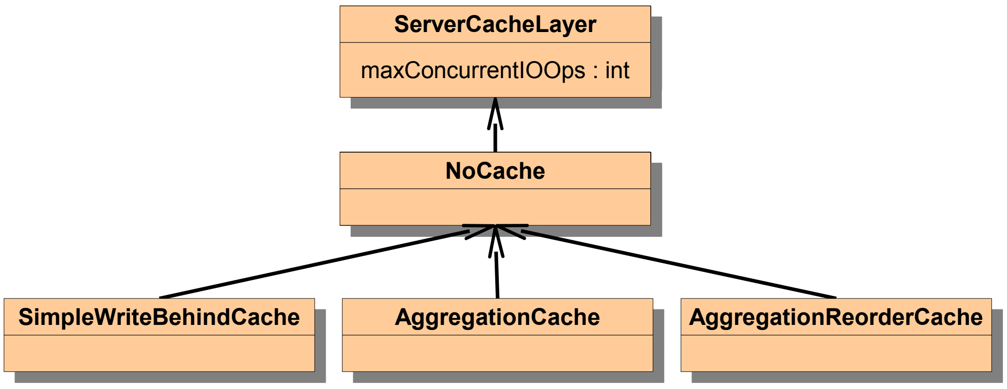


Figure 5.19.: Implemented cache layers.

While internal implementation is more complicated than the elevator algorithm of the *RefinedDiskModel*, internal data structures and behavior are similar[23].

Right now, all implemented cache layers process received data of non-contiguous requests in the order of the ListIO. An extension of the concept would add the appropriate I/O job on demand – thus, enabling the cache layer to virtually decide in which order data is received. With this extensions an assessment of disk-directed methods for write operations would be possible. Disk-directed methods for read operations are already implemented in the *AggregationReorderCache*; these have been evaluated in [KKL12].

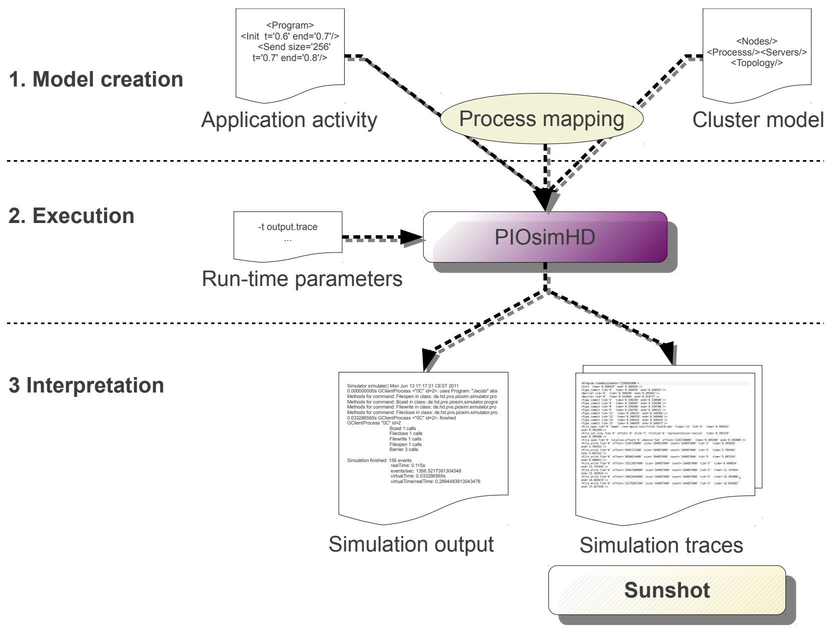
Discussion of the I/O path All these cache layers are abstractions for mechanisms in real systems. To qualitatively assess them, they are compared to optimizations available in Linux for accessing the local file system, and with mechanisms in PVFS. Linux aggressively caches data, it supports write-behind as well as read-ahead. The cache can grow to fill all available memory, this is similar to modeled caches that may use all available memory. When data of the cache is modified it is not immediately written to disk, instead Linux defers it until a timer fires or if the amount of dirty pages is too high. Furthermore, Linux stalls writing processes if more than a given percentage of available page cache is dirty, and thus defers the writing process[24]. In the write path modified cache pages are aggregated similar to the AggregationCache in many cases.

Compared to PVFS, the described I/O model permits additional optimizations. By default PVFS uses a *buffer size* of 256 KiB per I/O operation (according to the flow protocol). Effectively, accesses are fragmented into accesses with a granularity of the buffer size. This limits the number and size of reads which can be dispatched to the block device. Unfortunately, in this case the aggregation and scheduling offered by Linux do not help, as only up to 8 read operations are issued to the OS. When multiple clients access the file system concurrently, this leads to random access patterns on the block device, which, in turn, causes performance degradation[25].

Therefore, with large data sets the PVFS read performance is expected to be comparable to the *NoCache* implementation. As caching and aggregation of I/O operations is done by the OS, observable write performance should be comparable to the one achieved with the simulated *AggregationReorderCache*. Short term fluctuation due to updates of file system metadata and the complex interplay between multiple caches of a real I/O subsystem are not covered by any of the models.

Consequently, while there are similarities between the modeled I/O path and a real system, the modeled I/O behavior is much simpler and depends on only two variables (the granularity of network and I/O).

[23]Therefore, refer to Figure 5.5 on Page 225 for internal processing details.
[24]Refer to Section 3.6.1 for additional information about the behavior of the Linux cache.
[25]Refer to [Kun07] for an additional description of this issue.



```
<Program>
<Init  t='0.6' end='0.7'/>
<Send size='256'
t='0.7' end='0.8'/>
```

```
<Nodes/>
<Processs/><Servers/>
<Topology/>
```

```
Simulator simulate() Mon Jun 13 17:17:31 CEST 2011
0.000000000s GClientProcess <"0C" id=2>: uses Program: "Jacobi" alia
Methods for command: Fileopen in class: de.hd.pvs.piosim.simulator.pro
Methods for command: Bcast in class: de.hd.pvs.piosim.simulator.progra
Methods for command: Filewrite in class: de.hd.pvs.piosim.simulator.pro
Methods for command: Fileclose in class: de.hd.pvs.piosim.simulator.pro
0.033286565s GClientProcess <"0C" id=2>: finished
GClientProcess "0C" id=2
                    Bcast 1 calls
                    Fileclose 1 calls
                    Filewrite 1 calls
                    Fileopen 1 calls
                    Barrier 3 calls

Simulation finished: 156 events
                    realTime: 0.115s
                    events/sec: 1356.5217391304348
                    virtualTime: 0.033286565s
                    virtualTime/realTime: 0.2894483913043478
```

```
<Program timeAdjustment='1258362808'>
<Init  time='0.268425' end='0.268426'/>
<Type_commit tid='0'  time='0.268435' end='0.268597'/>
<Barrier cid='0'  time='0.268599' end='0.285692'/>
<Barrier cid='0'  time='0.314468' end='0.314777'/>
<Type_commit tid='4'  time='0.338258' end='0.338988'/>
<Type_commit tid='6'  time='0.338993' end='0.339280'/>
<Type_commit tid='8'  time='0.339288' end='0.339790'/>
<Type_commit tid='9'  time='0.339795' end='0.340225'/>
<Type_commit tid='11'  time='0.340232' end='0.340366'/>
<Type_commit tid='12'  time='0.340370' end='0.340409'/>
<Type_commit tid='14'  time='0.340415' end='0.340435'/>
<Type_commit tid='15'  time='0.340439' end='0.340475'/>
<File_open cid='0' name='./env-mpiio-test/file1-level0.dat' flags='12' fid='0'  time='0.350413'
end='0.395368'/>
<File_set_view fid='0' offset='0' etid='5' filetid='6' representation='native'  time='0.395370'
end='0.395588'/>
<File_seek fid='0' relative-offset='0' whence='Set' offset='3145728000'  time='0.395590' end='0.395808'/>
<File_write fid='0' offset='3145728000' size='104857600' count='104857600' tid='5'  time='0.395810'
end='2.784152'/>
<File_write fid='0' offset='6501171200' size='104857600' count='104857600' tid='5'  time='2.784165'
end='5.007522'/>
<File_write fid='0' offset='9856614400' size='104857600' count='104857600' tid='5'  time='5.007534'
end='8.488822'/>
<File_write fid='0' offset='13212057600' size='104857600' count='104857600' tid='5'  time='8.488834'
end='11.747644'/>
<File_write fid='0' offset='16567500800' size='104857600' count='104857600' tid='5'  time='11.747654'
end='15.383924'/>
<File_write fid='0' offset='19922944000' size='104857600' count='104857600' tid='5'  time='15.383988'
end='16.663674'/>
<File_write fid='0' offset='23278387200' size='104857600' count='104857600' tid='5'  time='16.663683'
end='17.817536'/>
```

Figure 5.20.: Simulation workflow with required input and generated output.

5.5. Simulation Workflow

The workflow to run a simulation experiment with PIOsimHD consists of three phases: model creation, execution of the simulator and interpretation of simulator output. These phases are separated from each other, which provides modularity. With this approach, for example, it is possible to change simulation parameters without modifying a created model.

The overall workflow for conducting a simulation study is illustrated in Figure 5.20. The following steps are required:

1. Model creation
   a) Instantiate a cluster model
   b) Describe or load application activity
   c) Map applications and ranks to available processes
2. Executing PIOsimHD
   a) Prepare run-time parameters
   b) Simulation model initialization: bind model and parameters
   c) Start simulation
3. Interpretation of the results
   a) Analyze simulation output
   b) Inspect traces (with Sunshot)

These steps are described further in this section.

5.5.1. Model Creation

The cluster is described in a system model that covers hardware and software behavior: These models are constructed by using classes provided in *PIOsimHD-Model*. Currently available model classes including their parameters, have been introduced in this section. For example, a SimpleDisk model has a parameter for average seek time.

Theoretically, it would be possible to create system and software models independently from each other. However, in many cases software and hardware are tightly coupled – software needs the resources offered by hardware, and must be placed on a system that provides it. Therefore, it was decided that individual software parameters are part of the model that needs them during simulation. There are also several global model parameters, such as the network granularity and I/O granularity that are set for all hardware and software components that need them.

Application behavior can be defined independently of a system model. However, there is a slight dependency between system model and application activity: a *process* (and its placement) is part of the system model – it contains the name of the application that should be executed and the MPI rank. Clearly, the number of processes that are referenced by an application should match the provided hardware resources. Note that PIOsimHD permits concurrent processing of multiple applications and thus the influence of activity that stresses network and I/O infrastructure can be investigated in silico.

PIOsimHD-Model offers the alternative to either define system model and application behavior in Java, or to read them from XML files. Helper classes simplify creation of cluster model and application behavior in a Java program. This is especially useful to perform small tests of I/O systems or MPI internal communication. It can also be used to prototype parallel applications and to assess communication and I/O performance before the application is actually implemented. Classes are provided that can read (and write) a system model from a file, behavior of existing applications can directly be read from trace files[26]. Trace files can be loaded into memory at simulator initialization, or, since they can be rather large, the simulator can load commands on demand. It is also possible to instantiate a system model in Java and bind applications to project files. This allows users to evaluate a variety of system models, while activity of a real application is fed into the simulator.

Reading commands from trace files The HDTrace MPI-wrapper records MPI calls and all required parameters, therefore, PIOsimHD can just instantiate commands and load the contained information[27]. Computation is not recorded explicitly, however, this information can be deducted – the time between two subsequent calls determines the computation time of jobs for the simulator. In the header of each trace and relation file the current processor speed and model information is recorded[28]. This information is used to compute the required cycles for compute jobs.

To change the computation time, the *instructionsPerSecond* parameter of a node can be changed. Therewith, the overall processing speed can be altered; scaling of computation time enables simulation of arbitrary computation speeds. However, the relation between computation phases will stay the same. By manipulating activities in the trace file certain regions can be sped up individually. Some existing simulators use this technique to change the behavior of parts of trace files (see for example BigSim in Section 2.6). Since the trace files are just XML files, modification is quite simple.

By itself, the currently used metric (instructions per second) is not very representative for the complexity of a CPU. Furthermore, the initially measured CPU speed may change at run-time because CPU frequency could be adjusted by an operating system at runtime to improve energy efficiency. The MPI-Wrapper can explicitly store computation phases and several hardware counters can be measured during the computation, for which it utilizes Likwid. Recorded counters include the actually observed *instructions per cycle* metric and Flop/s. In the future, those hardware counters could be used by a refined computation model. Thus, by using the hardware counters processing time on a target machine could be estimated better; theoretically, cache performance and memory footprint could be approximated by the percentage of memory

[26]These trace files are created by applications instrumented with HDTrace.
[27]The simulator maps trace entries to command types, this is described in Section 6.3.3.
[28]See Listing 4.1 on Page 191 for an example.

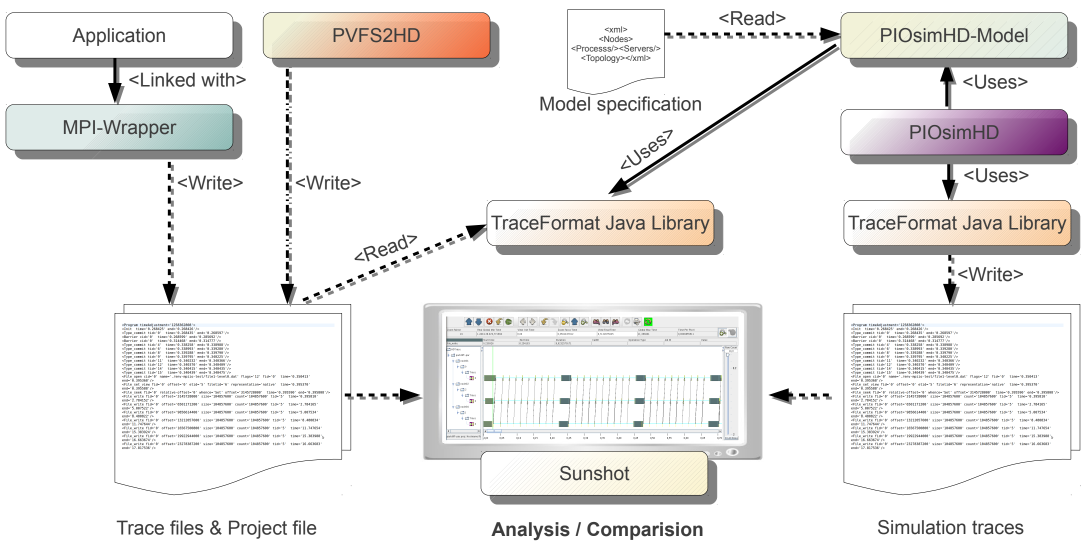


Figure 5.21.: Analysis workflow for existing applications including relevant components of HDTrace.

access and an expected cache hit/miss ratio. Since the application of Likwid is limited to some platforms such an extended model is not implemented in the simulator, yet[29].

5.5.2. Executing PIOsimHD

Run-time parameters control the behavior of PIOsimHD – the output of simulation results and information for debugging. When the parameters and the model are ready, the simulator can be initialized with them. Once initialized, further modification of parameters is not allowed.

The model classes provided by PIOsimHD-Model represent components and commands, and encapsulate necessary parameters. The simulator uses this information to instantiate required simulation classes. By binding the model, PIOsimHD prepares the simulation model which means instantiating *simulation classes* representing the model classes. A simulation class encapsulates behavior of the model class and contains all functionality required for simulation. An implementation uses parameters supplied by the model class for parameterization; the simulator must provide an implementation for every model. Once all objects are initialized, the routing algorithm is invoked on every network to determine potential routes.

The simulator core offers capabilities to change the implementation of a model, and of each command type[30]. Therefore, it is possible to evaluate alternative implementations for a model class without changing the model. Once the model is created for each network topology, the selected routing algorithm is executed.

The simulator is also shipped with a *command line interface* that executes models which are defined in a *model specification file*. This XML file is basically the serialized version of the hardware model: The mapping from application names to the sequence of commands that should be executed is done by specifying HDTrace project files. A model in PIOsimHD-Model can be serialized to such a specification file. Therewith, users can exchange models that are encoded in Java because the generated model specification file can be loaded with the command line interface that is part of PIOsimHD.

Simulating behavior of existing applications The workflow of simulating execution of an existing MPI program in a virtual cluster environment is depicted in Figure 5.21. Components of *HDTrace* that are involved in the different steps of the workflow are included in the figure: To trace the application activity, the MPI-wrapper is linked with the application, then all MPI(-IO) activities are intercepted and events are

[29]Refer to Page 197 for more discussion of this feature in the context of the wrapper.
[30]This is discussed in Section 6.3.

recorded by using the *TraceWriting C Library*. If PVFS is used as the underlying file system, additional traces for client and server I/O activities can be included. While events of PVFS are not used during simulation, they allow us to compare simulated and real I/O behavior.

The system model of the target machine architecture must be set up by the user in a model specification file. This model also contains references to the project files of applications and a mapping of processes to available nodes. To model a real system, the characteristics of the system must be determined and specified accordingly.

A user starts the command line interface and specifies the model specification file. *PIOsimHD-Model* reads the model from the file and loads required application trace files. In this process, the raw trace information is read and converted into a sequence of commands which can be understood by the simulator. The mapping of recorded activity to commands that are executed by the simulator is defined by a configuration file. Since the simulator should perform similar activity, the recorded parameters are read into the commands, but timestamps are explicitly computed during the simulation run and thus the timestamps from the trace files are discarded. Compute jobs are an exception, the simulator creates compute jobs by determining the time between two neighboring events – the number of cycles for a job are determined by using the compute times and CPU speed recorded in the trace file. A trace file is assigned to a single process; during the simulation this process replays the activities of the recorded application process. The simulator is executed with additional parameters provided on the command line, which control output behavior and verbosity, for example.

While *PIOsimHD* performs the discrete-event simulation, it can record the simulated activity by using the *TraceFormat Java Library*. Results of simulation can be visualized by *Sunshot*. Thus, simulation run and recorded application (and PVFS) activities can be compared visually by the user. By modeling the hardware on which the application has been executed, the models can be validated and inefficiencies of the system can be analyzed. This brings us to the reporting features of the simulator.

5.5.3. Interpretation of Simulation Results

There are two approaches to assess simulation results: For a first overview the simulator outputs internal information such as component usage profiles to the console. Also, activity of the simulated components can be written to trace files for a comparison with the original run. Supported output formats for the traces are HDTrace and TAU. The latter can be converted to SLOG2 format and then visualized by Jumpshot. With the evolving of HDTrace, the TAU trace writer became deprecated because HDTrace offers more capabilities.

By recording activity it is possible to look into behavior of simulated components. Thus, analyzing network flow or low-level I/O access is possible, which is unfortunately impossible with HPC tracing tools, yet. Parameters can be set to define the project name and the type of information which should be traced: On the client-side MPI activity, nested operations and state machine states can be selected individually. Further, server activity – which is the processing of state machines and block-device activity – and internal activity (packet transport) can be enabled separately. During simulation of an experiment, relations and regular trace files are written depending on this configuration. So far, the capability offered by statistics is not used, but in the future it could be used to record metrics similar to *libRUT*: CPU, network and disk utilization.

Visualization of client activity and packet transport The Sunshot visualization of the cluster configuration for the flow-example in which only node A sends data is given in Figure 5.22[31]. The topology viewer on the left shows the traced components: The node with the name "NA" (and internal id 4) hosts client process "A", the activity of the client process is rendered in a relation. Here, the client performs a

[31]See Page 230 for a description of the experiment configuration.

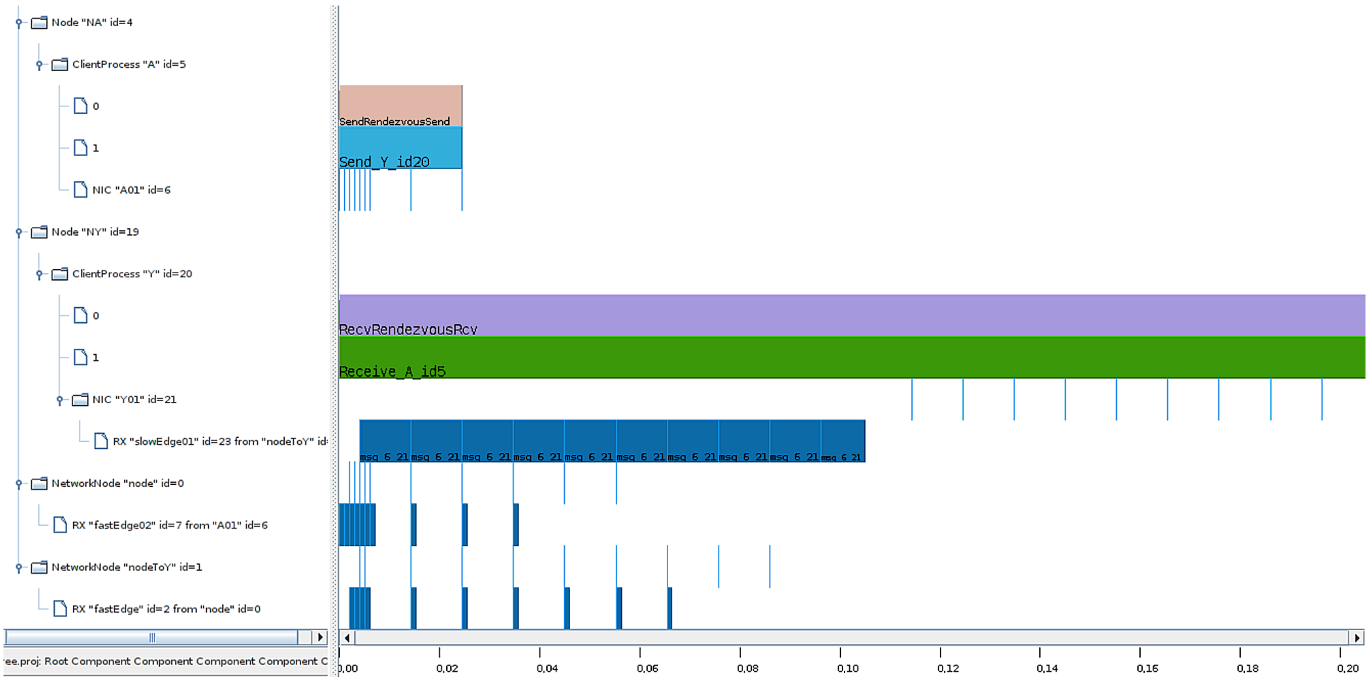


Figure 5.22.: Visualization of simulated client and network activity. The cluster configuration of the flow example in Figure 5.8 in which only node A sends data is simulated.

SendRendezvousSend operation; a command is suffixed with the implementation name used during simulation. Internally, the state machine of this implementation triggers a network send operation to the client process Y (rendered below). Again, the target name and its id are encoded in the name of the activity category (here “Y id20”).

Normally, computation time required for processing of a state machine is rendered inside the command trace entry as a nested state; as the *SendRendezvousSend* implementation does not perform extra computation it is not visibile in this example[32].

A network card belongs to its process and is therefore found underneath the process in the topology tree. Timelines for node NY and the process Y are rendered in a similar fashion. Edges are grouped under their target’s node. Labels on edges contain the name of the source network node. Therewith, the interconnection and flow of packets can be understood; in the example, packets sent by “nodeToY” are received on the NIC of node NY.

In the example the high network latency of the slowEdge becomes apparent. Intermediate network nodes and all their incoming edges are grouped in the same way, therefore, the two nodes (“nodeToY” and “node”) are visible.

To track the flow of packets, relation arrows can be computed in Sunshot; a screenshot that includes all arrows is shown in Figure 5.23. Note that this small example already shows a vast amount of visual information. For this reason, the advanced filter mechanisms and the color-coding feature were implemented in Sunshot to help users concentrating on relevant information.

Visualization of I/O activity I/O components are structured in the topology node in a straight forward manner: A server is rendered as a child of a node, a server hosts a block device, its activity is listed underneath the server. Execution of concurrent requests on the server is visualized by the relation concept like client activity. A screenshot of one process is given in Figure 5.24. In this example the process opens a file, writes 1 MB of data and then closes the file. The startup phase of this example is provided in Figure 5.25.

[32]In fact, every state performs at least one computation cycle, thus computation is rendered, but very small.

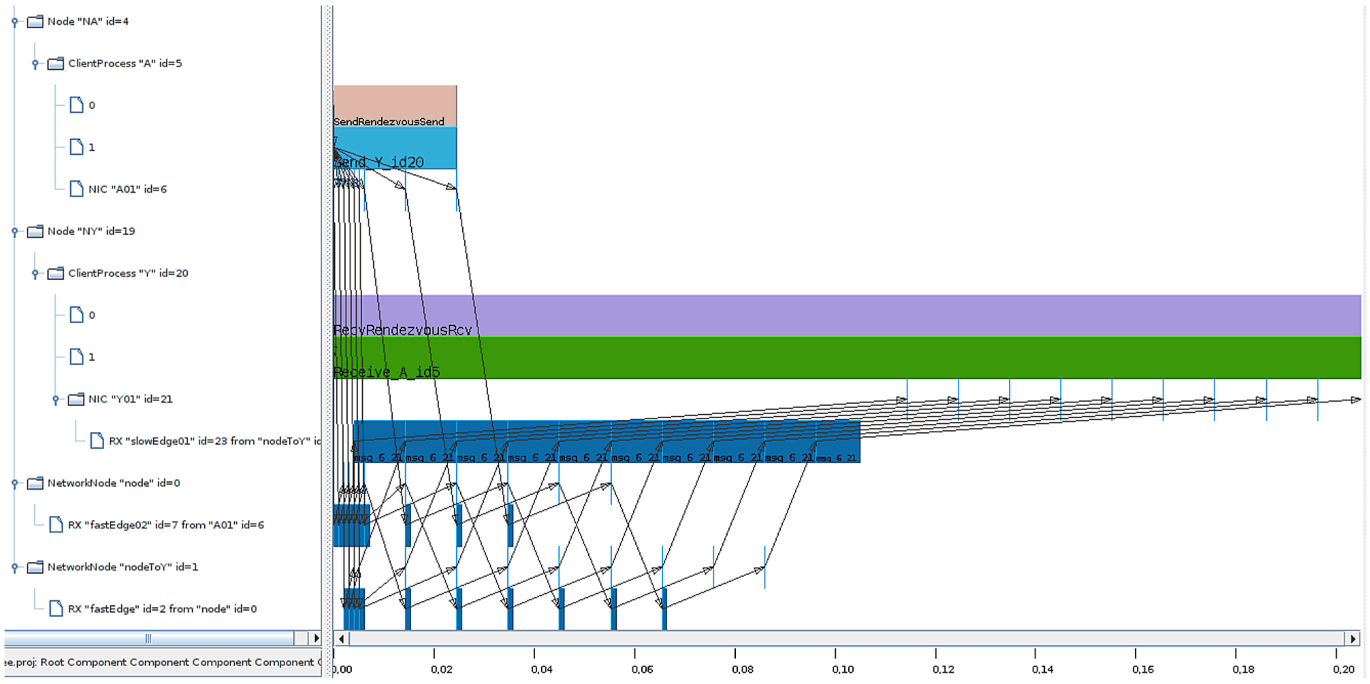


Figure 5.23.: Visualization of simulated client and network activity including relation arrows. The cluster configuration of the flow example in Figure 5.8 in which only node A sends data is simulated.

Since the aggregation cache is used, write operation are cached quickly on the server; the client performs a flush operation when the file is closed. The block device performs two write operations: the first operation writes 100 KiB with an average seek; 100 KiB is the maximum size of a message due to the selected network granularity. Then data is aggregated and, due to the sequential nature of the access pattern, subsequent data can be written without seeks.

Detailed information about the accessed extents of I/O requests and the block device operations are provided in the *info box window*[33].

Output of internal information In addition to the tracing, the simulator can output internal information and statistics that have been acquired during the simulation. At run-time, the simulator prints the mapping of applications to processes and the implementations which are used for simulating particular commands. When a client terminates the model time is written to the console. This type of output allows tracking of processing inside the simulator, it can be disabled.

Upon termination, basic information is provided: Current model time – that is the virtual program's run-time including deferred I/O activity, number of processed events and wall-clock time. Further, uncompleted client activity and network operations are reported because any unfinished activity indicates a problem: Either the application specification is erroneous and caused a deadlock, or the implementation of a command in the simulator has a bug.

Additionally, individual components manage their own *internal information* which are available after the simulation terminates. This information is provided in container classes – it can be serialized for output on the console, or automatically analyzed. Currently, implemented components maintain statistical information about the number of total operations and partition them into efficient and inefficient operations. By providing an overview, a first assessment of the simulation run is possible without looking at simulation traces.

An example output is provided in Listing 5.2. Collected internal information is just serialized to the console in this example: Clients track the numbers of commands executed, the refined disk model maintains

[33]See Page 199 for more information about the *info box window*.

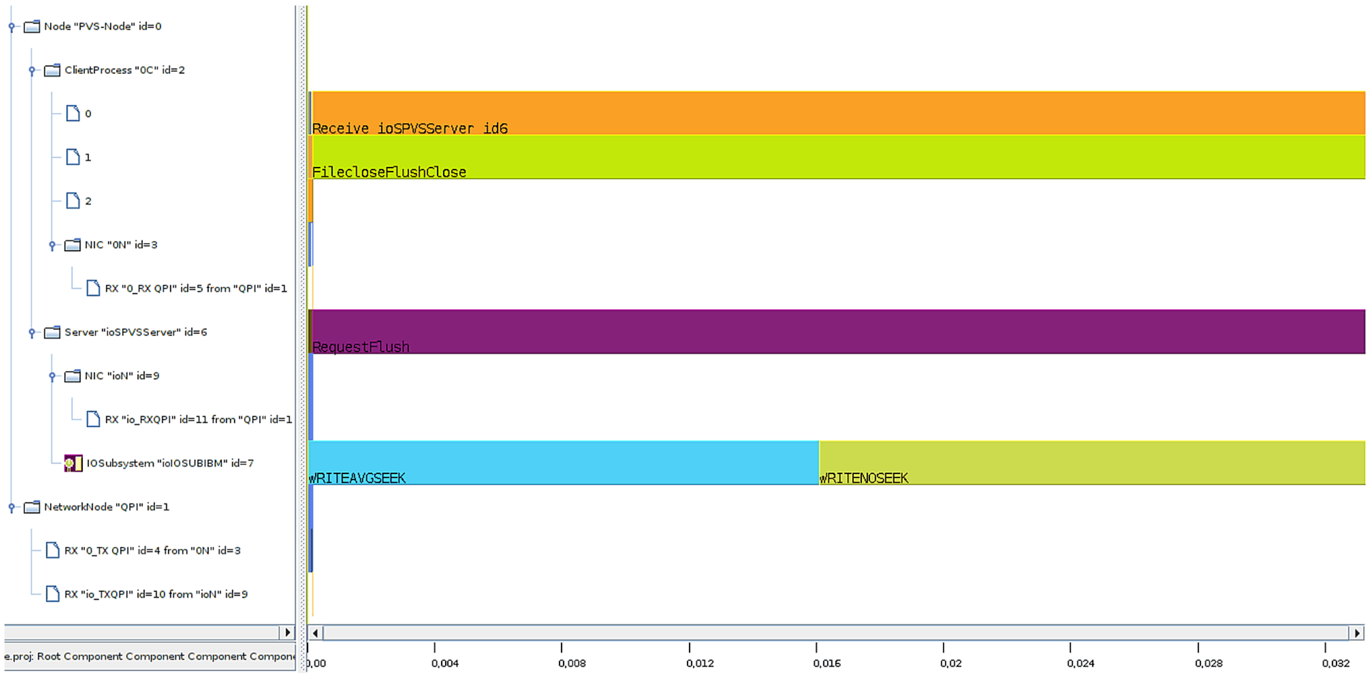


Figure 5.24.: Visualization of simulated I/O activity – one client writes 1 MB of data to one server. Due to the write-behind mechanisms most data is written to disk while the client waits for the completion of the flush operation.

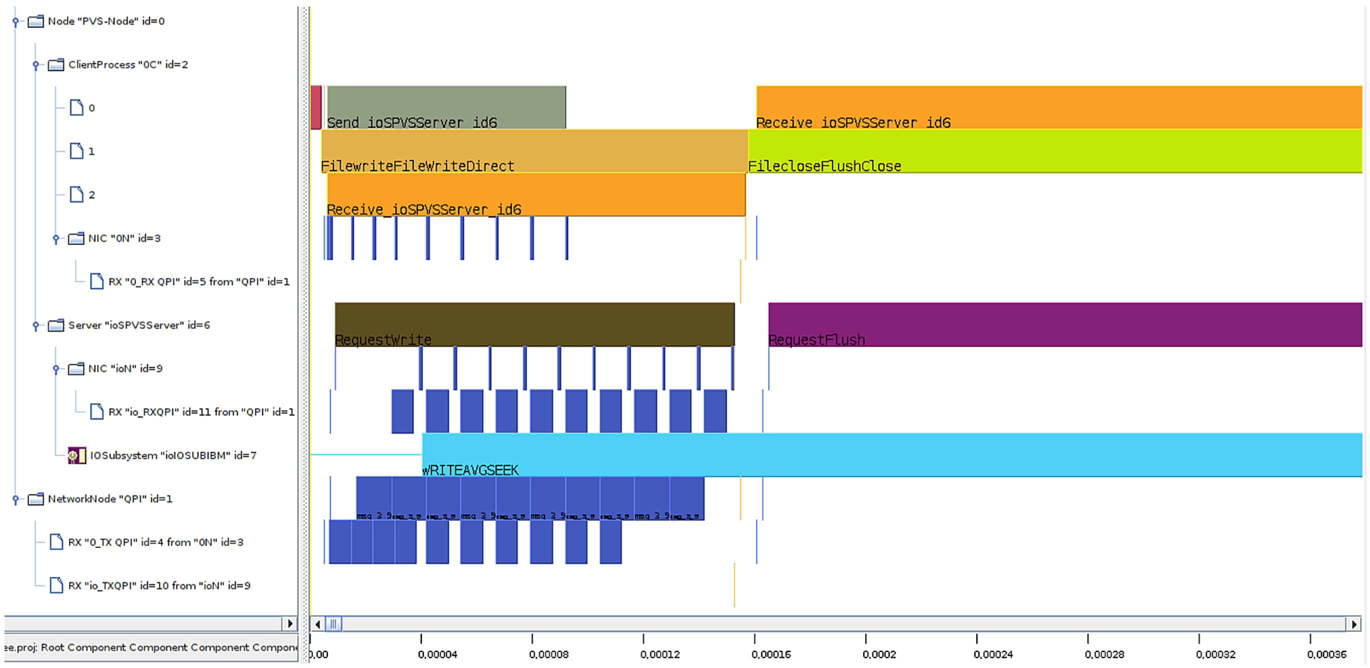


Figure 5.25.: A zoom into the start phase of Figure 5.24.

the number of performed operation, seeks performed and the total amount of accessed data. On the block device three operations have been executed, one fast operation and two slow operations – a total of 1 MB of data has been written. Simulation completed in 0.115 s (realTime) and execution of this experiment takes 0.0333 s on the virtual cluster (virtualTime).

Listing 5.2: Simulation output for the simple I/O experiment including internal information collected during simulation.

```
Simulator simulate() Mon Jun 13 17:17:31 CEST 2011
0.000000000s GClientProcess <"0C" id=2>: uses Program: "Jacobi" alias: "Jacobi" rank 0
Methods for command: Fileopen in class: de.hd.pvs.piosim.simulator.program.FileOpen.BroadcastOpen
Methods for command: Bcast in class: de.hd.pvs.piosim.simulator.program.Bcast.PipedBlockwise
Methods for command: Filewrite in class: de.hd.pvs.piosim.simulator.program.Filewrite.FileWriteDirect
Methods for command: Fileclose in class: de.hd.pvs.piosim.simulator.program.FileClose.FlushClose
0.033286565s GClientProcess <"0C" id=2>: finished
GClientProcess "0C" id=2
        Bcast 1 calls
        Fileclose 1 calls
        Filewrite 1 calls
        Fileopen 1 calls
        Barrier 3 calls
GRefinedDiskModel "ioIOSUBIBM" id=7 <#ops, noSeekAccesses, fastAccesses, slowAccesses, dataAccessed> =
  →<3, 1, 0, 2, 1000000>
Simulation finished: 156 events
         realTime: 0.115s
         events/sec: 1356.5217391304348
         virtualTime: 0.033286565s
         virtualTime/realTime: 0.2894483913043478
```

5.6. Chapter Summary

In this chapter PIOsimHD, the Java discrete-event simulator for cluster systems, is introduced. Goal of PIOsimHD is to assist researching MPI-IO for arbitrary systems, and to foster understanding of performance factors in HPC.

Experiments with simulated hardware and application behavior can either be conducted by tracing existing MPI-IO applications or by coding the actions directly in Java. Providing the appropriate level of detail for the simulation of application and system behavior is difficult. Therefore, PIOsimHD allows users to choose a level of abstraction for hardware and software layers at execution time. Basic, simple to understand, models of network, disk and CPU represent most relevant performance factors. Sophisticated models of hardware and software components can be implemented as well. Model classes offered by PIOsimHD-Model represent a certain behavior and hold the parameters for components and commands. Parameters of a component are used in a simulation class that represents and implements behavior of the model class.

Simulation with PIOsimHD provides the freedom of constructing arbitrary system and application behavior: Hardware components are interconnected with user-defined network topologies; the characteristics of components can be parameterized at will. For inter-process communication, messages are partitioned into packets by the NIC. Packet travel is controlled by a flow algorithm that ensures optimal utilization of the network. To utilize the links, the flow model fills the bandwidth-delay-product of the wires per data stream and packets are never lost. Several examples demonstrated how a realistic maximum network throughput is obtained.

A software model describes the execution of a parallel application: A process executes a sequence of commands; each command is represented by a state machine which may trigger network operations and nested state machines. The I/O model for the abstract parallel file system is analogous to PVFS client-server processing but focuses on data transfer. A file system server contains a block device which is controlled by a cache layer. Several implementations of the cache layer are introduced, they incorporate optimization strategies such as write-behind, aggregation and scheduling of requests.

The current approach of assessing simulated behavior is discussed. To give a first impression of the simulation result to the user, PIOsimHD outputs basic information on the command line and provides statistics of component *activity. However, experiments conducted with the simulator can be also recorded for later analysis with Sunshot. Depending on the desired level of detail, several types of client, server, and network activity can be recorded. With this feature existing traces can be compared with the simulated behavior.*

The abstract knowledge gained in this chapter is complemented by interesting internals of the simulator provided in the next chapter.

Bibliography

[FLS$^+$06] J. Flich, P. López, J. Sancho, A. Robles, and J. Duato. Improving InfiniBand Routing through Multiple Virtual Networks. In Hans Zima, Kazuki Joe, Mitsuhisa Sato, Yoshiki Seo, and Masaaki Shimasaki, editors, *High Performance Computing*, volume 2327 of *Lecture Notes in Computer Science*, pages 363–368. Springer Berlin / Heidelberg, 2006.

[KKL12] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Simulation-Aided Performance Evaluation of Server-Side Input/Output Optimizations. In *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP. Munich Network Management Team, IEEE, 2012.

[Kun07] Julian Martin Kunkel. Towards Automatic Load Balancing of a Parallel File System with Subfile Based Migration. Master's thesis, Ruprecht-Karls-Universität Heidelberg, 08 2007.

[LM94] T. V. Lakshman and Upamanyu Madhow. Performance Analysis of Window-based Flow Control Using TCP/IP: Effect of High Bandwidth-Delay Products and Random Loss. In *Proceedings of the IFIP TC6/WG6.4 Fifth International Conference on High Performance Networking V*, pages 135–149, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[Zim80] H. Zimmermann. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980.

Chapter 6 VI

Simulator Implementation

This chapter highlights selected aspects of the implementation and brings them into the context of the design. Illustrating examples give an impression of model development and implementation, and of processes involved in model selection and parameterization for an experiment. An exhausting documentation of all involved aspects and interfaces has been omitted; it is documented by and inside the source code.

Model classes offer the parameters for the implementations of components and commands, this is demonstrated for the RefinedDiskModel in Section 6.1. The implementation of a model is explained in Section 6.2 for the SimpleDiskModel.

Implementations for the models are can be selected in configuration files. This allows the user to extend the simulator and to evaluate various modules for the same model (see Section 6.3). Since selection of a model is very important, the selection process and underlying file formats are discussed extensively.

In Section 6.4 code is listed that illustrates how to set up a cluster model and to configure the simulator to execute a small experiment. This example demonstrates how to encode a model directly in Java.

Implementation and execution of commands on client-side is another aspect that demonstrates the extensibility of the simulator. Section 6.5 documents the state machine implementations for two implementations of `MPI_Bcast()`.

6.1. Model Classes

Model classes are relatively simple containers for the data required during the simulation. Java annotations constrain the content of fields. Correct usage is checked once the model is provided to the simulator. Furthermore, the annotations are used for automatic serialization and deserialization of the model – Java objects are converted to XML files.

The introspection and analysis of the field attributes is done in the implemented (de)serializing classes via the *Reflection API*. Existing classes or frameworks such as *java.beans.XMLEncoder* or *XStream*[1] have not been used. The reason is that new attributes are not only used to serialize the models, but also to perform elementary consistency checks. Furthermore, the attributes have been used in the former simulation GUI to automatically display available parameters and restrict possible values.

A model implementation is exemplified for the *RefinedDiskModel*. The implementation is provided in Listing 6.1. For convenience the base class `BlockDevice` is listed as well – the `BlockDevice` class inherits from the `BasicComponent` class. The template parameter (`Server`) indicates that the parent component of an I/O subsystem is of a `Server` class. The BasicComponent implements `getObjectType()` – this function is required during the serialization process to know the component type. A model class (or its supertype) must implement this function to return the name of the component type – such as BlockDevice.

The implementation of the RefinedDiskModel class is very simple. Parameters of the model are defined as variables[2]. These variables are annotated with `@Attribute` to indicate that they are parameters for the model. Additionally, values can be restricted with annotations. For example, the Epoch of trackToTrack-SeekTime must defined and cannot be Null, therefore, it is annotated with `@NotNull`. `RPM` is constrained by `@NotNegativeOrZero`, which checks that the value conforms to this condition. The class provides accessors for its parameters. To make sure that all values are assigned during the parameterization, values are initialized with an invalid state.

[1] `http://xstream.codehaus.org/`
[2] Refer to Section 5.2.2 for a discussion of this model.

Listing 6.1: Excerpt of the model class for the *RefinedDiskModel* and its superclass

```
abstract public class BlockDevice extends BasicComponent<Server> {
        public final String getObjectType() {
                return BlockDevice.class.getSimpleName();
        }
}

public class RefinedDiskModel extends BlockDevice {
        /**
         * Time needed to position the access arm to a neighbouring track:
         * This is the minimum seek time.
         */
        @Attribute
        @NotNull
        private Epoch  trackToTrackSeekTime = Epoch.ZERO;

        /**
         * The average time needed to move the access arm to another track.
         */
        @Attribute
        @NotNull
        private Epoch  averageSeekTime = Epoch.ZERO;

        /**
         * Rotations per minute. It is used to calculate rotational latency.
         */
        @Attribute
        @NotNegativeOrZero
        private int RPM = -1;

        /**
         * Sustained transfer rate of the disk, once the access arm is placed.
         */
        @Attribute
        @NotNegativeOrZero
        private long sequentialTransferRate = -1;

        [...]

        /* Example accessors: get() and set() method of the RPM. */

        public void setRPM(int rpm) {
                RPM = rpm;
        }

        public int getRPM() {
                return RPM;
        }

        ...
```

6.2. Implementation of a Component

An implementation of a component accesses data stored in the corresponding model class. This is used to parameterize the model. Since discrete-event simulation is used, the component must implement methods to process events. Several superclasses are provided that provide an execution scheme, which simplifies development of new component implementations. For example, classes derived from a *SSequentialBlockingComponent* issue one job after another and schedule pending operations with FCFS strategy.

For a discussion of the concept, the implementation of the simple disk model is provided in Listing 6.2, this class relies on the SSequentialBlockingComponent supperclass to dispatch I/O jobs. The implementation of the SimpleDisk model is directly documented in the source code.

It is important to note that the SSequentialBlockingComponent class drives the dispatching of the events and thus eases the implementation of components that execute jobs sequentially. Internally, the superclass queues pending operations, and tries to dispatch a pending job whenever it is idle. The execution model of a blocking component works as follows: When a job is dispatched for execution, the `jobStarted()` method is called. Duration of the job is determined by the `getProcessingTimeOfScheduledJob()` method. A blocking component queues an internal event when the job is considered to be finished and suspends further operations. Once the model time matches the completion time of the job, this event reactivates the component. The SSequentialBlockingComponent invokes the `jobCompleted()` function. After that, the next pending operation is executed.

This little example just indicates the processing of very simple components. The processing schemes of superclasses and, especially, the implementation of node component and client processes are complex. Therefore, a description is out of the scope for this thesis.

Listing 6.2: Implementation of the simple disk model

```
// Implementation of a SimpleDisk model.

// The superclass defines a sequential execution model for pending operations.
// Template parameters for the superclass indicate the model (here SimpleDisk) and
// the type of work which is scheduled (here an IOJob).
// The IGIOBlockDevice interface must be implemented because it is used by the
  →cache
// layer to communicate with the I/O subsystem

public class GSimpleDisk
extends SSequentialBlockingComponent<SimpleDisk, IOJob>
implements IGIOBlockDevice<SimpleDisk>
{
  // Reference to the component which uses this disk.
  // Whenever an I/O operation completes on the simulated block device, a method
  // of the callback is invoked to inform the using component about the completion.
  IBlockDeviceCaller callback;

  // Some internal statistics about execution of operations
  // are maintained during simulation:
  // The total number of operations that have been dispatched.
  int totalOperations = 0;
  // The total amount of data that has been accessed.
  long totalAmountOfData = 0;

  // Set the callback, this method is declared in the IGBlockDevice interface.
  // It is invoked when components are instantiated to represent the model.
  @Override public void setIOCallback(IBlockDeviceCaller callback) {
      this.callback = callback;
  }

  // Determine the processing time of an I/O job; declared (and used) in the
  // SSequentialBlockingComponent class.
  @Override protected Epoch getProcessingTimeOfScheduledJob(IOJob job) {

    if(job.getOperationType() == IOOperationType.FLUSH){
      // A flush finishes immediately.
      return Epoch.ZERO;
    }

    // Compute the processing time with the model parameters for throughput and
    // average access time from the SimpleDisk model. The size of the processed I/O
    // job is needed.
    final long accessedData = ((StreamIOOperation)
      →job.getOperationData()).getSize();
    return getModelComponent().getAvgAccessTime().add(
        accessedData / (float) getModelComponent().getMaxThroughput()
        );
  }

  // This method is invoked whenever a job is dispatched by the superclass.
  @Override protected void jobStarted(Event<IOJob> event, Epoch startTime) {
    // Start tracing of the I/O job; use a helper class which eases tracing
    // for arbitrary block device implementations.
    IOSubsytemHelper.traceIOStart(this, event.getEventData());
  }

  // This method is invoked by the SSequentialBlockingComponent class,
  // when a job completes.
  @Override protected void jobCompleted(Event<IOJob> event, Epoch endTime) {
    // Access the actual IOJob from the event.
    IOJob job = event.getEventData();

    // End the tracing of the I/O job.
    IOSubsytemHelper.traceIOEnd(this, job);

    // Update the statistics.
    totalOperations++;
    switch(job.getOperationType()){
    case READ:
    case WRITE:
       totalAmountOfData += ((StreamIOOperation) job.getOperationData()).getSize();
    }

    // Notify the callee of the I/O subsystem about the completion of the I/O job.
    callback.IOComplete(endTime, job);
  }

  // A cache layer invokes this method to submit a new IOJob.
  // This method is declared in the IGIOBlockDevice interface.
  @Override public void startNewIO(IOJob job) {
    // The current model time.
    Epoch time = getSimulator().getVirtualTime();
    // Add a new event to this component which should be executed ASAP.
    // The SSequentialBlockingComponent will queue this event for future execution
    // if it is currently busy.
    addNewEvent(new Event<IOJob>(this, this, time, job, null));

    // Try to actually start a pending operation;
    // this call is implemented in the SSequentialBlockingComponent superclass.
    startNextPendingEventIfPossible(time);
  }

  // This method is called by the simulator, when simulation finishes.
  // It is declared in the base class for a simulated component.
  // Normally, the method checks that the internal state of this component is
    →clean,
  // that means no further operations are pending, which would indicate a bug in
  // the simulator. But here this method is stretched to output data.
  @Override public void simulationFinished() {
    // Some statistics are output for the purpose of demonstration.
    // In general, this should be done by implementing getComponentInformation()
    // because that allows automatic processing and analysis of the collected data.
    System.out.println("BlockDevice␣" + getIdentifier() +
      "␣<#ops,dataAccessed>␣=␣<" + totalOperations + ",␣" + totalAmountOfData+">");
  }
}
```

6.3. Dynamic Selection of Implementations

The simulator core offers capabilities to select an implementation for a component model, such as the SimpleDisk model, and for each command type. By using the Java Reflection API, the simulator instantiates selected classes at run-time (on demand). Therewith, an easy extension of currently implemented models is possible. An alternative implementation can be provided outside of the packet namespace used in this dissertation[3]. Also, intercomparison of implementations can be done without changing the model. In Section 6.3.1 the selection of an alternative component implementation is discussed. Choosing a command implementation is detailed in Section 6.3.2.

Additionally, a configuration file defines the mapping of HDTrace information to commands. Therewith, trace entries can be excluded from a simulation run by modifying the mapping. This is described in Section 6.3.3.

6.3.1. Adjusting Model Implementations

In the cluster model each component can have its own model class. For instance, one I/O server can use the SimpleDisk model class, while another uses the RefinedDiskModel. The implementing class of a model uses the data provided in the model class as parameters.

The implementations for model classes can be chosen before the simulator is run. The mapping of a model class to an implementation is encoded in a simple text file which is read during simulator startup. One simulation class can be chosen per model class; the selected mapping is fixed during simulation. At run-time the simulator instantiates the chosen implementations for all components that use the implementation. In our example, one implementation can be chosen for all SimpleDisk models and one for all RefinedDiskModel models. Consequently, existing models and contained parameters can be re-used and evaluated with alternative implementations without the need to replicate or modify the model.

An excerpt of the file is provided in Listing 6.3: For each of the component types such as a Server or a BlockDevice, a section starting with "+" contains a list of all model classes implementing the component type, and the mapping to the implementation. Elements are listed with the *canonical class name* [4], followed by the canonical class name of the implementation that is used for simulation. For example, it can be seen that the Server component is currently implemented by the class GSimpleServer. There are several cache layers available; an implementation is provided for each of them.

Listing 6.3: Mapping from model components to simulator implementations. Excerpt of the file `ModelToSimulationMapper.txt`.

```
+Server
de.hd.pvs.piosim.model.components.Server.Server =
  →de.hd.pvs.piosim.simulator.components.Server.GSimpleServer
```

[3]The packet namespace that has been used for all developed components is *de.hd.pvs.piosim*.
[4]The canonical class name includes the fully qualified packet name and class name.

```
+Router
de.hd.pvs.piosim.model.components.Router.Router =
  →de.hd.pvs.piosim.simulator.components.Router.GRouter

+ServerCacheLayer
de.hd.pvs.piosim.model.components.ServerCacheLayer.NoCache =
  →de.hd.pvs.piosim.simulator.components.ServerCacheLayer.GNoCache
de.hd.pvs.piosim.model.components.ServerCacheLayer.SimpleWriteBehindCache =
  →de.hd.pvs.piosim.simulator.components.ServerCacheLayer.GSimpleWriteBehind
de.hd.pvs.piosim.model.components.ServerCacheLayer.AggregationCache =
  →de.hd.pvs.piosim.simulator.components.ServerCacheLayer.GAggregationCache

[...]
```

6.3.2. Changing Command Implementations

Similar to model classes of hardware components, the parameters of a command are specified in a model as well, also XML serialization follows the same rules. Mapping from the model of a given command to an implementation is selected in another text file. Additionally, the implementations of the command types can be set in Java, too, making it possible to evaluate alternative command implementations automatically.

An excerpt of the mapping by the file is given in Listing 6.4. For each command, such as an `MPI_Barrier()`, the file contains a list with all available implementations. The goal of this list is to allow a user to query available implementations.

The list for a command is started with the canonical class name of the command model prefixed by a "+". All the consecutive lines are potential implementations. For example, a barrier command could do nothing, or it could be implemented by doing a virtual barrier[5]. Currently, two other implementations are provided.

When multiple commands are related to each other thus requiring matching implementations, a group has to be defined. For example, `MPI_Send()`, `MPI_Recv()` and `MPI_Sendrecv()` depend on each other. Consequently, they are defined as a group, ensuring matching implementations.

By default the simulator takes the last implementation in the list to implement the command, however, a user can override this behavior in the model XML, or upon startup of the simulator. It is very easy to integrate new commands or implementations for existing ones by adding the appropriate lines to the file.

Usually, it does not make sense to pick different implementations for a command on different processes since processing and communication in distinct implementations follow their own protocol. Imagine combining a dummy `MPI_Barrier()` implementation which does not send any message with another implementation sending messages for synchronization – obviously, this will lead to a deadlock. Therefore, for each command one implementation is used during the whole simulation and across all client processes. There is one exception to this rule – a command can invoke arbitrary implementations in nested commands.

`NoOperation` and `Compute` operations are internal operations. The first does not do anything; all unknown trace entries are mapped to this command type. The latter deals with the computational time in the application: When an XML trace is loaded, computation jobs are forged in the simulator to fill the gaps between consecutive trace entries. This is necessary because the MPIWrapper does not record compute jobs explicitly. The number of cycles of a job depends on the time passed between the two trace jobs and the speed of CPU that created the trace; this information is provided in the trace header. For example, if

[5]This is a barrier which continues when all clients started it without triggering any network activity.

the first trace entry ends at t=1.0 s and the second trace entry starts at 2.0 s, then a compute job is created that needs exactly one second.

With the extension introduced on Page 197, an explicit recording of compute jobs is possible. This could include passed cycles accurately. Hence, computation could be simulated better. Yet, these extensions are not utilized in the simulator [6].

Listing 6.4: Mapping from model commands to simulator implementations. Excerpt of the file `CommandToSimulationMapper.txt`.

```
# Define command groups and mapping from models to simulator implementations.
# Each section defines a command (or a set of commands), which is then
# implemented by several following classes.

# All undefined operations are mapped to NoOperation, which also needs an
  →implementation.
+NoOperation
de.hd.pvs.piosim.simulator.program.Global.NoOperation

# Implementation for a compute job, a job can do nothing or just pass time
  →(cycles).
+Compute
de.hd.pvs.piosim.simulator.program.Global.NoOperation
de.hd.pvs.piosim.simulator.program.Compute.Time

# Define a group for MPI_Send, MPI_Recv and MPI_Sendrecv.
+de.hd.pvs.piosim.model.program.commands.Send,
  →de.hd.pvs.piosim.model.program.commands.Recv,
  →de.hd.pvs.piosim.model.program.commands.Sendrecv
# Either implement all three with NoOperation.
de.hd.pvs.piosim.simulator.program.Global.NoOperation,
  →de.hd.pvs.piosim.simulator.program.Global.NoOperation,
  →de.hd.pvs.piosim.simulator.program.Global.NoOperation
# Or with a rendezvous implementation.
de.hd.pvs.piosim.simulator.program.SendReceive.Rendezvous.RendezvousSend,
  →de.hd.pvs.piosim.simulator.program.SendReceive.Rendezvous.RendezvousRcv,
  →de.hd.pvs.piosim.simulator.program.SendReceive.Rendezvous.RendezvousSendrecv

# The MPI_Barrier command is implemented with several alternatives.
+de.hd.pvs.piosim.model.program.commands.Barrier
de.hd.pvs.piosim.simulator.program.Global.NoOperation
de.hd.pvs.piosim.simulator.program.Global.VirtualSync
de.hd.pvs.piosim.simulator.program.Barrier.Direct
de.hd.pvs.piosim.simulator.program.Barrier.BinaryTree

[...]

+de.hd.pvs.piosim.model.program.commands.Filereadall
de.hd.pvs.piosim.simulator.program.Global.NoOperation
de.hd.pvs.piosim.simulator.program.Filereadall.Direct
de.hd.pvs.piosim.simulator.program.Filereadall.TwoPhase
de.hd.pvs.piosim.simulator.program.Filereadall.ContiguousTwoPhase

[...]

+de.hd.pvs.piosim.model.program.commands.Bcast
de.hd.pvs.piosim.simulator.program.Global.NoOperation
de.hd.pvs.piosim.simulator.program.Global.VirtualSync
de.hd.pvs.piosim.simulator.program.Bcast.BroadcastScatterGatherallMPICH2
```

[6]Refer to Section 197 for a discussion of the issues with the implementation.

```
de.hd.pvs.piosim.simulator.program.Bcast.BinaryTreeSimple
de.hd.pvs.piosim.simulator.program.Bcast.BinaryTree
de.hd.pvs.piosim.simulator.program.Bcast.BinaryTreeMultiplex
de.hd.pvs.piosim.simulator.program.Bcast.BroadcastScatterGatherall
de.hd.pvs.piosim.simulator.program.Bcast.BroadcastScatterBarrierGatherall
de.hd.pvs.piosim.simulator.program.Bcast.BinaryTreeSimpleBlockwise
de.hd.pvs.piosim.simulator.program.Bcast.PipedBlockwise
```

6.3.3. Mapping Trace Entries to Command Models

When trace files are read, individual trace entries (state names) of the trace file must be mapped to command models that should keep the information (these are defined in the `CommandToSimulationMapper.txt`). A text file defines the interpretation of a state name; an excerpt of this mapping is listed in 6.5. A comma separated list specifies XML state names which are mapped to the command model following the colon. For example, an `MPI_Init()` is mapped to the *GlobalSync* implementation, which is basically a barrier across `MPI_COMM_WORLD`. If the name of the state name and the implementation are identical, the state name can be omitted. Most operations follow this one to one mapping. However, I/O operations are renamed in the simulator, also variants of one operation such as `MPI_File_write()` and `MPI_File_write_at()` are mapped to one model class. This mapping also shows the currently supported MPI functions.

Listing 6.5: Mapping from trace states to command models. Excerpt of the file `TraceEntryNameToCommandMapper.txt`.

```
Init,Finalize:GlobalSync
:Send
:Recv
:Sendrecv
:Wait
:Allreduce
:Allgather
:Barrier
:Reduce
:Gather
:Bcast
:ReduceScatter
:Scatter
File_set_view:Filesetview
File_open:Fileopen
File_close:Fileclose
File_read,File_read_at:Fileread
File_write,File_write_at:Filewrite
File_read_all,File_read_all_at:Filereadall
File_write_all,File_write_all_at:Filewriteall
```

6.4. Defining an Application and System Model in Java

PIOsimHD supports conducting experiments with recorded trace files, but also building a model directly in Java[7]. Actually, it is also possible to create the system model in Java and then load trace files for some, or all applications.

Several builder classes assist in building a model on the fly. This is supported by a template library which can duplicate components, and classes which can create primitive SMP and cluster topologies with a cen-

[7]For further information refer to the description of the workflow in Section 5.5.

tral switch. Those classes could also be adjusted to build arbitrary systems. After a model is built it can also be exported to XML.

Example code to build and run the experiment is demonstrated for the flow experiment that is discussed in Section 5.3.2 on Page 230. The code to set up a sequence of commands in the application model and to run the simulation is provided in Listing 6.6. Code to build the underlying cluster model is given in Listing 6.7. To ease model creation, templates are defined and instantiated in the code. Since the example is well documented, further code discussion is omitted.

When the simulator completes execution, the internal run-time information of the components is printed on the command line. Additionally, the code activates tracing, so traces including a project file, are written at run-time. A visualization of executing this program is provided in Section 5.5.3 (Page 250); the output of the internal run-time information is shown on Page 252.

Listing 6.6: Java code to construct a simulation experiment in Java by using builder classes. In this example one process sends data to another process.

```
public class ModelTest{

public void runTest(){
// Create the model builder and a single network topology.
ModelBuilder mb = new ModelBuilder();
INetworkTopology topology = mb.createTopology("LAN");

// Chose a routing algorithm for the network topology.
PaketRoutingAlgorithm routingAlgorithm = new PaketFirstRoute();
topology.setRoutingAlgorithm(routingAlgorithm);

// Create the system model here, see the code snippet in Listing 6.7
createSystemModel(mb);

// Helper class to ease building of applications.
// Each ApplicationBuilder is responsible for a single application.
// Prepare an application with three processes.
ApplicationBuilder aB = new ApplicationBuilder("flowExample", "Application␣
  →illustrating␣data␣flow", 3, 1);

// Create a program builder which assists in application model building.
ProgramBuilder pb = new ProgramBuilder(aB.getApplication());

// Use the created application in the system model.
// The mapping is defined in the system model:
// Every client provides a String which sets the application and the rank to use.
mb.setApplication("flowExample", aB.getApplication());

// Set up the sequence of commands to execute:
// Rank 0 sends a message with 1000 KiB of data to process 3.
pb.addSend(world, 0, 3, 1000*KiB, 0);
// Process 3 receives data from process 0 that matches tag 0.
pb.addRecv(world, 0, 3, 0);
// Add a barrier at the end of the processing.
pb.addBarrier(world);

// Configure the global model.
// Adjust the eager size of MPI send/receive.
mb.getGlobalSettings().setMaxEagerSendSize(1000 * MiB);

// Set up parameters for the execution.
RunParameters parameters = new RunParameters();
parameters.setTraceEnabled(true);
parameters.setTraceInternals(true); // Trace communication details.
parameters.setTraceFile("/tmp/three");

Simulator sim = new Simulator();

// Initialize the model; this creates entities based on system model and
  →configuration.
sim.initModel(mb.getModel(), parameters);

// Run the simulation. Results are stored in the simRes object.
SimulationResults simRes = sim.simulate();

// Write the internal run-time information of all components to standard out.
// For this purpose, a serializer is provided that writes all information to a
  →String.
SimulationResultSerializer serializer = new SimulationResultSerializer();
System.out.println(serializer.serializeResults(simRes));
// An automatic analysis of the output could be implemented here, too...
}
```

Listing 6.7: Java code to set up a cluster model with the builder classes. In this example a cluster model for the flow example of Page 5.3.2 is created.

```
public void createSystemModel(Modelbuilder mb){
// Instantiate objects that are later used as templates.
final Node node = new Node();
final NIC nic = new NIC();
final StoreForwardNode fastNode = new StoreForwardNode();
final SimpleNetworkEdge fastEdge = new SimpleNetworkEdge();
final SimpleNetworkEdge slowEdge = new SimpleNetworkEdge();

// Set model properties.
nic.setTotalBandwidth(100*GBYTE);
nic.setName("nic");

node.setName("node");
node.setInstructionsPerSecond(1000000000);
node.setCPUs(1);
node.setMemorySize(GBYTE);

fastNode.setName("fastNode");
fastNode.setTotalBandwidth(100*GBYTE);
fastEdge.setName("fastEdge");
fastEdge.setLatency(new Epoch(0.001));
fastEdge.setBandwidth(100*MiB);

slowEdge.setName("slowEdge");
slowEdge.setLatency(new Epoch(0.1));
slowEdge.setBandwidth(10*MiB);

// Add the instances to the template library.
mb.addTemplateIf(node);
mb.addTemplateIf(nic);
mb.addTemplateIf(fastNode);
mb.addTemplateIf(fastEdge);
mb.addTemplateIf(slowEdge);


// Create the central node by cloning the template.
// Therewith, instances of the model objects are created that have the same
```

```
  →parameters.
// New instances can be created by referring to the template object, or
// by using the name of a template.

final NetworkNode nodeInner = mb.cloneFromTemplate(fastNode);
nodeInner.setName("node");
// Add the node to the system model.
mb.addNetworkNode(nodeInner);

// Create nodeToY from the same template.
final NetworkNode nodeToY = mb.cloneFromTemplate(fastNode);
nodeToY.setName("nodeToY");
mb.addNetworkNode(nodeToY);

// Create the interconnect between the inner nodes.
// Instantiate edges for incoming and outgoing data transfer.
NetworkEdge e1 = mb.cloneFromTemplate(fastEdge);
NetworkEdge e2 = mb.cloneFromTemplate(fastEdge);

// Connect the edges with the nodes in the topology.
mb.connect(topology, nodeInner, e1, nodeToY);
mb.connect(topology, nodeToY, e2, nodeInner);

// Add all clients and interconnect them.
// An array encodes the processes' names.
final String [] names = {"A", "B", "C", "Y", "Z"};

for(int i=0; i < names.length; i++){
  NIC clientNIC = mb.cloneFromTemplate(nic);
  clientNIC.setName(names[i]);

  final ClientProcess clientProc = new ClientProcess();
  clientProc.setName("C" + names[i]);
  // Set the rank and application.
  clientProc.setRank(i);
  clientProc.setApplication("flowExample");
  // Assign the NIC.
  clientProc.setNetworkInterface(clientNIC);

  // Instantiate the node.
  final Node node = mb.cloneFromTemplate(node);
  node.setName("N" + names[i]);
  mb.addNode(node);
  // Add the client to the node.
  mb.addClient(node, clientProc);

  // Interconnect the nodes, node Y is treated differently.
  if(i != 3){
    // Interconnect via the central network node.
    e1 = mb.cloneFromTemplate(fastEdge);
    e2 = mb.cloneFromTemplate(fastEdge);
    mb.connect(topology, clientNIC, e1, nodeA);
    mb.connect(topology, nodeA, e2, clientNIC);
  }else{
    // Interconnect node Y.
    e1 = mb.cloneFromTemplate(slowEdge);
    e2 = mb.cloneFromTemplate(slowEdge);
    mb.connect(topology, clientNIC, e1, nodeToY);
    mb.connect(topology, nodeToY, e2, clientNIC);
  }
}
}
```

6.5. Implementation and Execution of Commands

The implementation of commands is demonstrated with a few examples. Additionally, this illustrates the concept of state machine execution[8].

First, the implementation of `MPI_Allreduce()` that calls `MPI_Reduce()` followed by an `MPI_Bcast()` is shown in Listing 6.8[9]. Implementations are sublasses of `CommandImplementation` – a template parameter specifies the model of the command realized.

The processing of an operation using the state machine concept is encoded in the `process()` method. Arguments to the command, e.g., communicator and data size, are part of the model instance (here `cmd`). Instructions on how to proceed with the state machine must be stored in the `ICommandProcessing` object passed to `process()`.

The step encodes the current state of the state machine. Our example state machine has two states which are executed on all clients. An implementation can define arbitrary operations for each state and the next step to proceed. When a new command is executed, the `step` number is set to `CommandProcessing.STEP_START`. A command is considered to be finished when the state `CommandProcessing.STEP_COMPLETED` is reached.

In the example, nested operations are started by creating a new command and then calling `invokeChildOperation()`. The first parameter to this function is the command to submit, the next is the state to which the state machine should transit when the nested operation completes. With the last parameter, a specific implementation for the command can be selected; even if another one is chosen in the global description of the experiment it can be overridden for the execution of this single command.

Before a state is executed, the number of CPU cycles for processing this state are fetched by calling `getInstructionCount()`. Since no extra computation has to be simulated in the example, the default value of 1 cycle is returned.

Listing 6.8: Java code of the `MPI_Allreduce()` implementation which calls `MPI_Reduce()` followed by `MPI_Bcast()`

```
// This implementation performs a reduce operation followed by a broadcast.
public class ReduceBroadcast extends CommandImplementation<Allreduce>{
    @Override public void process(Allreduce cmd, ICommandProcessing OUTresults,
                        GClientProcess client, long step, NetworkJobs compNetJobs){
        if (cmd.getCommunicator().getSize() == 1){
            // Only one process in the communicator.
            // Consequently, we have no data to broadcast and we can complete.
            return;
        }

        // We need two states: The first state is STEP_START,
        // the second state is called BROADCAST.
        // A step with this value indicates that the previous reduce operation has
          →completed.
        final int BROADCAST = 2;

        // Distinguish the two states of this state machine.
        if (step == CommandProcessing.STEP_START){
```

[8] Details about the abstract execution concept of client operations is provided in Section 5.4.1.
[9] Background information about this naive implementation is provided on Page 235.

```
            // Create a reduce operation to reduce data to root of the
              →communicator.
            Reduce reduce = new Reduce();
            reduce.setCommunicator(cmd.getCommunicator());
            reduce.setSize(cmd.getSize());
            reduce.setRootRank(0);

            // When the reduce completed, transit to the state BROADCAST.
            // The simulator will use the global broadcast implemenation.
            OUTresults.invokeChildOperation(reduce, BROADCAST, null);

        }else if(step == BROADCAST){

            // Initiate a broadcast from root.
            Bcast bc = new Bcast();
            bc.setCommunicator(cmd.getCommunicator());
            bc.setSize(cmd.getSize());
            bc.setRootRank(0);

            // Once the operation completes, finish the state machine.
            OUTresults.invokeChildOperation(bc, CommandProcessing.STEP_COMPLETED,
              →null);
        }
    }

    // This function provides the number of instructions required to process each
      →step
    // of the state machine. Since the actual computation is encoded in
      →MPI_Reduce, no
    // additional computation is required.
    @Override
    public long getInstructionCount(Allreduce cmd, GClientProcess client, long
      →step) {
        return 1;
    }
}
```

Pipelined broadcast implementation Now, a more sophisticated implementation of `MPI_Bcast()` is discussed. Actually, this broadcast might be invoked as a child operation from the previously discussed *ReduceBroadcast* implementation.

The *PipedBlockwise* broadcast implementation builds a pipeline between processes of the communicator; the root process sends data to Rank 1 which sends data to Rank 2 and so on. Data is fragmented into smaller packets to allow pipelining, transfer on a node is started once the first packet is received. The implementation assumes the first packet can be buffered on the receiver side. Before the second packet of data is sent, a process waits for notification from its successor indicating that it is ready to receive further data; this happens when the successor actually invokes `MPI_Bcast()` with the same communicator. This implementation prevents that a *late starter* is overwhelmed by incoming data.

An estimate for the time costs of this implementation is given in equation 6.1. Throughput and latency characterize the network components that process a packet, and the time packets travel across connections. In the equation the variable *path* contains all network components of the route which the packets travel, i.e., the network edges and nodes involved while the packet is sent from Rank 0 to Rank 1, then from Rank 1 to Rank 2 and so forth. The count function defines how often a component is used in the pipeline's path as the throughput of a component is multiplexed among all streams. If no congestion happens and all processes start at the same time, then this formal estimate should be close to practice.

$$t = \sum_{c \in \text{path}} \text{latency(c)} + \frac{\text{splitSize}}{\min_{c \in \text{path}} \text{throughput(c)/count(c)}} \quad (6.1)$$

Code implementing the pipelined variant is listed in Listing 6.9. Since the simulator and the tracing primarily operate with the global rank, the extended class `CommandImplementationWithCommunicator-LocalRanksRemapRoot` remaps a global rank into local ranks, simplifying implementations of collective commands which follow a one-to-many or many-to-one operation. In those schemes, one process is special, this class translates the ranks in such way that the implementation can always assume Rank 0 refers to the dedicated source or target rank. Currently, the method `getLocalRankExchangeRoot()` is used when network operations are issued, it maps a potentially remapped rank back to the real rank[10].

This state machine is more complicated than the previous one, it encodes the current progress of the pipeline in the step number; the step number represents the number of blocks which have been transferred. Three types of processes can be distinguished: The root process transmits data to the first process, the last process just receives data and intermediate processes receive and transmit data. To communicate with other processes, the methods `addNetSend()` or `addNetReceive()` are invoked which queue a request on the NIC. Note that no real data is sent, the network message is just characterized by its size. Messages received from other clients are provided in the `compNetJobs` data structure and can be inspected. In this implementation, data of the first receive is sent to the next process in the pipeline. An MPI header is attached to each message to to simulate additional MPI overhead. The state machine will not proceed to the next step before all message jobs are completed.

Listing 6.9: Java code of the pipelined `MPI_Bcast()` implementation. Data is transferred in blocks of `splitSize`.

```
public class PipedBlockwise extends
 →CommandImplementationWithCommunicatorLocalRanksRemapRoot<Bcast>{
  // Define the packet size for fragmenting messages.
  final long splitSize = 1 * 1024*1024;

  // Size of the packet message headers.
  final int msgHeader = 20;

  // Tags used in communication. They are normally used to distinguish ready
  // messages and data, in this case they are just used to increase readability.
  final int READY_TAG = 5;
  final int DATA_TAG = 30;

  // Return the special rank which will be mapped to Rank 0 by the superclass.
  @Override public int getSingleTargetWorldRank(Bcast cmd) {
          return cmd.getRootRank();
  }


  // This method is a variant of the previously mentioned process() method.
  // In addition to process(), it takes the remapped rank in the communicator and
  // the rank of the root process as arguments.
  // The activity of each process and its steps is listed separately; while this
  // approach leads to more code, each of the states can be coded independently.
  @Override public void processWithLocalRanks(Bcast cmd, ICommandProcessingMapped
   →OUTresults, Communicator comm, int clientRankInComm, int rootRank,
   →GClientProcess client, long step, NetworkJobs compNetJobs){

    if (cmd.getCommunicator().getSize() == 1){
      // No peer to broadcast data to.
```

[10] It would be possible to overload the `addNetSend()` and `addNetReceive()` calls to get rid of this requirement – this is future work.

```
      return;
    }

    if(clientRankInComm == 0){
      // Rank 0 starts the pipeline.
      // Amount of data which remains to be sent.
      final long dataRemains = cmd.getSize() - (splitSize * step);

      // Variable that keeps the amount of data to send in this step.
      long amountToTransfer;

      // Determine amount of data to send
      if (dataRemains > splitSize){
        // Another block of data must be transmitted.
        OUTresults.setNextStep(step + 1);
        amountToTransfer = splitSize;
      }else{
        // This is the last block to transmit.
        OUTresults.setNextStep(CommandProcessing.STEP_COMPLETED);
        amountToTransfer = dataRemains;
      }

      // Each message adds a small message header.
      IMessageUserData data = new NetworkSimpleData(amountToTransfer + msgHeader);

      // The rank to send data to, which is the remapped rank 1.
      int targetRank = getLocalRankExchangeRoot(rootRank, 1);

      // Send the first packet without waiting for a notification.
      OUTresults.addNetSend(targetRank, data, DATA_TAG, cmd.getCommunicator());

      if(step == 0){
        // Wait for an acceptance notification from the other rank to prevent
        // sending of messages which are not yet ready to be received.
        OUTresults.addNetReceive(targetRank, READY_TAG, comm);
      }

    }else if(clientRankInComm == (comm.getSize() - 1) ){

      // The last rank only accepts packets.
      if(step == 0){
        // Announce to be ready to the previous rank.
        OUTresults.addNetSend(
          getLocalRankExchangeRoot(rootRank, clientRankInComm-1),
          new NetworkSimpleData(msgHeader), READY_TAG, cmd.getCommunicator());
      }
      // The number of iterations to run before all data is received.
      // The number of steps is equal to the number of blocks to transmit.
      final int stepsToPerfom = (int)((cmd.getSize() - 1)/splitSize);

      // Decide if more data must be accepted.
      if(step < stepsToPerfom){
        OUTresults.setNextStep(step + 1);
      }else{
        // This is the last iteration.
        OUTresults.setNextStep(CommandProcessing.STEP_COMPLETED);
      }

      // Post a receive operation to receive data from the previous rank.
      OUTresults.addNetReceive(
        getLocalRankExchangeRoot(rootRank, clientRankInComm-1),
        DATA_TAG, cmd.getCommunicator());

    }else {  ////////////////////////////////

      // A rank which receives and forwards data.

      if(step == 0){
        OUTresults.setNextStep(1);

        // Post an operation to receive the first packet from the previous rank.
        OUTresults.addNetReceive(
          getLocalRankExchangeRoot(rootRank, clientRankInComm-1),
          DATA_TAG, cmd.getCommunicator());

        // Announce to be ready to the previous rank.
        OUTresults.addNetSend(
          getLocalRankExchangeRoot(rootRank, clientRankInComm-1),
          new NetworkSimpleData(msgHeader), READY_TAG, cmd.getCommunicator());

        // Wait for the next process to be ready before starting data transfer.
        OUTresults.addNetReceive(
          getLocalRankExchangeRoot(rootRank,clientRankInComm+1),
          READY_TAG, cmd.getCommunicator());
        return;
      }

      assert(step != 0);

      // The number of steps to perform adds one step for the initial receive.
      final int stepsToPerfom = (int)((cmd.getSize() - 1)/splitSize) + 1;

      // Only receive data from the previous rank.
      if( step < stepsToPerfom){
        OUTresults.setNextStep(step + 1); // Receive again.

        // Post a receive operation to access data from the previous rank.
        OUTresults.addNetReceive(
            getLocalRankExchangeRoot(rootRank, clientRankInComm - 1),
            DATA_TAG, cmd.getCommunicator());
      }else{
        OUTresults.setNextStep(CommandProcessing.STEP_COMPLETED);
      }

      // Pass received data to the next process by just forwarding
      // received message content.
      final IMessageUserData data = compNetJobs.getResponses()[0].getJobData();

      int targetRank = clientRankInComm + 1 ;
      OUTresults.addNetSend( getLocalRankExchangeRoot(rootRank, targetRank), data,
                             DATA_TAG, cmd.getCommunicator());
    } // End of the block for an intermediate rank.
  } // End of process().
}
```

From the perspective of a client, this pipeline involves one sending and one receiving process, so one stream of data is processed. Assume a developer intends to write an algorithm in which data streams are processed independently at the same time – for example, to allow an independent communication with different file servers. To implement such a scheme, first it is necessary to implement a streamed point-to-point connection (like in the example above). Another state machine could instantiate this command for every communication partner and all commands can be spawned as nested operations. Then the state machine executes every nested state machine independently, once all children completed, the parent state machine proceeds.

Visualization of the pipelined broadcast A screenshot showing the execution of a pipelined broadcast on a homogeneous cluster configuration is shown in Figure 6.1: Three processes join a broadcast of 10 MiB of data, that means the pipelined broadcast transfers 10 blocks of data. Activity on the switch and the processes illustrate the pipelined character; pipeline startup happens quickly and outgoing edges of the switch are busy transferring data.

Interestingly, Rank 1 needs more time to receive a packet from Rank 0 than to pass it on to Rank 2. A similar fast transmission is observable for the first packet of Rank 0. This is no mistake, the behavior is caused by the flow protocol: The model of a compute node adds several internal network nodes to permit faster intra-node communication. Since each network component buffers at least one network packet, the network topology can hold multiple packets. A packet that is submitted to the NIC requires additional time to arrive at its target because of the latencies along the way. Therefore, forwarding of previously received data happens a bit quicker than receiving data for the next step. In a real system such a behavior could manifest as well because in many cases network streams are buffered by the OS (or by the NIC). Therefore, it is realistic that completion of a data transfer is announced before data is completely received[11].

By inspecting the behavior in Sunshot the correct operation of the implementation could be verified. For this example, communication pattern and the amount of data transferred matches the expected behavior of the pipelined implementation.

6.6. Chapter Summary

To provide insight into the implementation of the simulator, selected aspects of the implementation are discussed in this chapter. This includes creation and implementation of the cluster and application model that are executed.

Model classes represent components and commands; they are containers that encapsulate necessary parameters. The simulator provides an implementation class for every model and uses parameters supplied by the model class. This is illustrated by the class for the RefinedDiskModel model. Java annotations support automatic (de)serialization of parameters to/from XML. Additionally, these annotations are used to perform an automatic model check. Code for the SimpleDisk implementation demonstrates creation of a new component by using such a model. This process is simplified by using a superclass that processes issued operations with the FCFS strategy.

At run-time, the simulator loads a mapping that defines the implementations responsible for the corresponding component models. Implementations for commands can be adjusted similarly and, further, the mapping from trace entries to commands can be changed. Selection of the implementations is done by adjusting text files. Therewith, different (MPI) command implementations can be compared without changing the model. Command implementations can be changed directly in the model or Java as well.

Preparation and execution of a model are demonstrated for the flow example; in this process system model and application commands to execute are defined. An alternative way to coding applications in Java is the instrumentation of an MPI program to automatically create traces that can be loaded by the simulator.

Two examples for an implementation of a command illustrate internal processing and state machine concept: An `MPI_Allreduce()` implementation is realized with a nested call to `MPI_Reduce()` and `MPI_Bcast()`. An implementation of a broadcast exemplifies the programming of complex algorithms and their state machines.

In the next chapter, the model for our cluster is built, validated and experiments with the simulator are conducted.

[11]The amount of data that can be buffered is expected to be different, though.

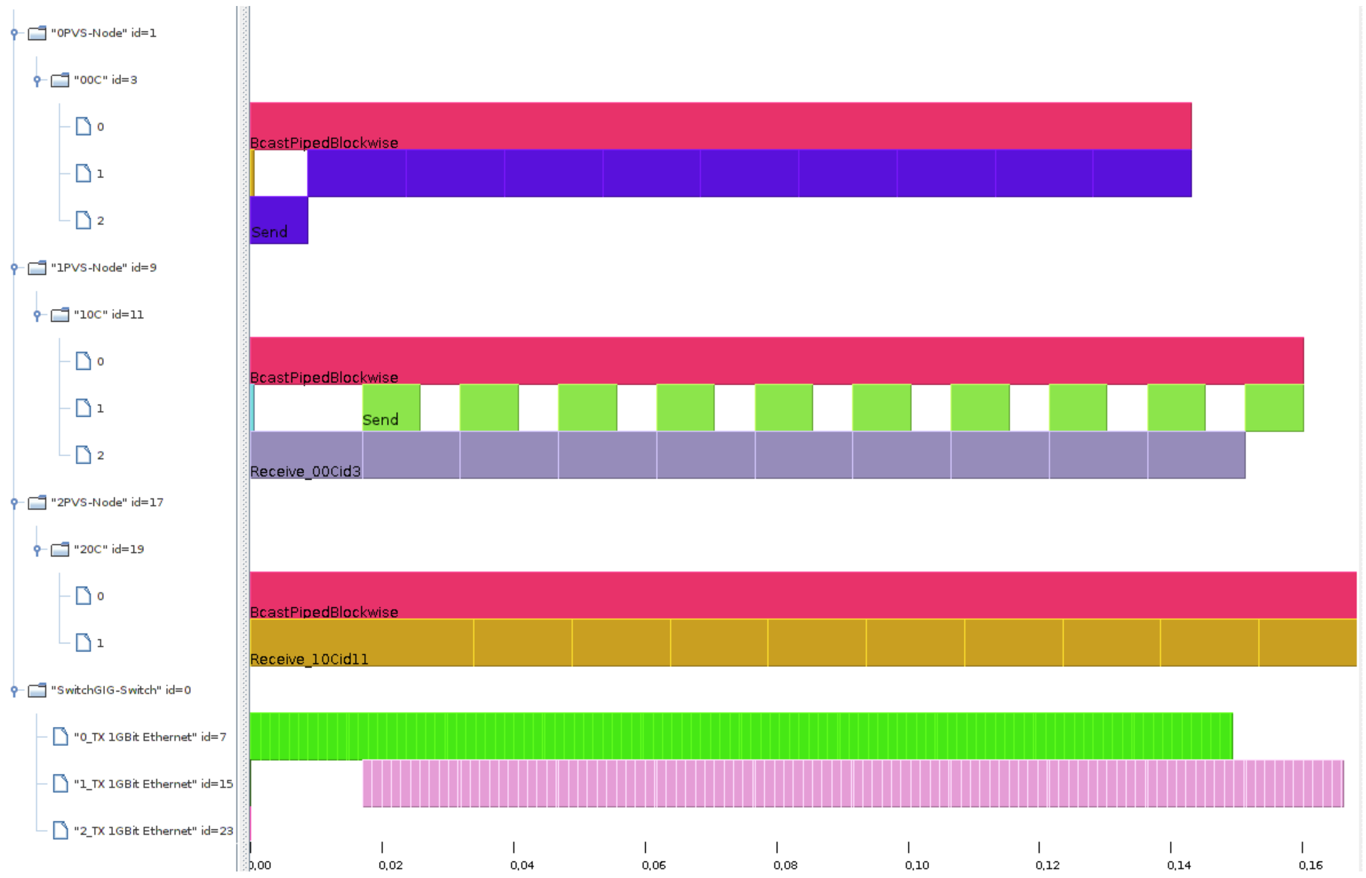


Figure 6.1.: Visualization of the pipelined broadcast implementation. 10 MiB of data is broadcasted to two processes resulting in 10 transfers of 1 MiB each. The upper process is the root process sending blocks of data, the center process forwards data to the last process which just receives data.

Chapter 7 VII

Evaluation

A thorough evaluation of the HDTrace environment requires an assessment of the non-functional requirement of performant code. Therefore, the overhead of HDTrace and the suitability of PIOsimHD to estimate performance for large cluster systems is evaluated. Overhead in terms of computation time and storage space is briefly discussed in Section 7.1. Performance and scalability of PIOsimHD is assessed in Section 7.2.

Validation of the model and a correct implementation of it is very important to provide sufficient simulation accuracy. To validate the results, model parameters must be identified. Compared to the real system the models introduced in Section 5 are rather simple and can be parameterized with just a few characteristics. Still, these representative characteristics must be determined. The conceptual model is adjusted and parameterized in Section 7.3 to represent the experimental cluster that has been introduced in Chapter 3.

The characterization of our system presented in Chapter 3 has multiple roles during the evaluation. First, the average performance measurements are used to parameterize the model. Second, several performance results act as a reference during the validation process. Finally, the measurements put expectations for exact simulation into perspective: The complex performance behavior of individual operations showed the difficulty in analyzing, characterizing and infeasibility of capturing all relevant aspects in a model. A consequence of the revealed behavior is the need for a detailed analysis of the implemented models. Also, it shows that it is insufficient to compare the average performance of simulation and observation. To assess overall correctness and behavior of individual operations, a visual inspection and comparison of trace files and simulation results is performed during the validation. This also helps localizing reasons for diverging simulation results.

After the parameterization, in Section 7.4 a qualification of analytical model is performed by comparing measured results for simple experiments with estimated performance. Therefore, the simulator's model is simplified to analytical models, which can estimate performance for a subset of experiments. The simulator cannot do better than the underlying model, therefore, this process fosters understanding the inaccuracy that is inherent to the implemented model. Then, the primary network and I/O model of the simulator are verified against the conceptual analytical model in Section 7.5. This proves that the basic behavior of the analytical model is captured by the implementation.

All collective calls of MPICH2 use more or less complex point-to-point communication patterns underneath. In Section 7.6, measured performance of collective calls is compared to the times obtained by replaying the underlying point-to-point communication patterns. This does not only validate the communication model, it also shows potential performance loss of MPICH2 on our cluster – with this methodology the achieved performance of MPICH2 can be validated as well. Simulated and measured performance of a parallel I/O benchmark is assessed in Section 7.7. Collective operations are implemented inside the simulator, the approach of their implementation is discussed in Section 7.8.

Complex application behavior is evaluated with PIOsimHD in Section 7.9; this demonstrates its applicability to scientific programs. To this purpose, traces of the MPI calls of a scientific application are recorded and replayed in the simulation. To show the capabilities of PIOsimHD to evaluate alternative collective calls, several alternative implementations of the `MPI_Bcast()` *function are evaluated in Section 7.10.*

The additional supplementary experiments utilize the simulator to localize bottlenecks in existing systems. Altogether, this scientific methodology shows that PIOsimHD can be applied to analyze behavior of MPI-IO applications, and that the underlying hardware and software model is capable to predict performance of a cluster system sufficiently. Furthermore, several phenomenons that could be revealed with post-mortem tracing can be observed in recorded activity of the simulation as well. Considering the rather simple system model, it is astonishing to see the high level of accuracy in predicting performance and the capability to recreate many real world phenomenons.

7.1. Overhead of HDTrace

Tracing with *HDTrace* intercepts regular function calls and records the activity, hence, the activation of tracing influences the application performance. A qualitative analysis of this overhead is important to verify that the influence is neglectable for the results that are presented in this thesis. Further, in a small experiment, the overhead of HDTrace is compared with VampirTrace.

For this test, the VampirTrace distribution which is part of Open MPI has been used in combination with OTF output. A single process is started on a cluster node. This process loops 1 million times over an `MPI_Barrier()`. A barrier has been selected since it involves almost no parameter checking and avoids copying of memory regions. The source code of this program is provided in Listing 7.1. While this simple test does not evaluate performance of complex operations it shows the behavior of the tracing infrastructure.

Listing 7.1: Source code used to evaluate the tracing overhead

```c
#include <mpi.h>
#include <stdio.h>
#define COUNT 1000000

int main (int argc, char *argv[])
{
  MPI_Init (&argc, &argv);

  // In this benchmark the MPI timer is used. Thus, accuracy is limited.
  double start_time = MPI_Wtime();

  for(int i=0; i < COUNT; i++){
    MPI_Barrier(MPI_COMM_WORLD);
  }

  double end_time = MPI_Wtime();

  printf("time:␣%fs␣per␣Iter:␣%fs\n", end_time - start_time,(end_time - start_time)
    → / COUNT);

  MPI_Finalize();
  return 0;
}
```

Computation overhead Measured times are shown in Table 7.1. The wall-clock time gives the total run-time of the program. When tracing is enabled, initialization and finalization takes some time. During the finalization, trace data must be written to NFS, which is time-consuming. This additional overhead is ignored, the time per barrier is computed by dividing the run-time of the loop by the number of iterations.

Running the code just with *Open MPI* and *MPICH2* yields a run-time in the order of 20 ns per barrier. *VampirTrace* and *HDTrace* add an overhead of 1.8 μs and 1.4 μs, respectively. Further, trace-initialization and finalization increase wall-clock time by a few seconds. Judged by the collected information, overhead of both tracing methods is comparable. In this configuration neither of them includes tracing of the performance counters. If the Likwid feature is enabled in HDTrace, additional time will be spent to set and query the performance counters between two MPI calls. When the Likwid extension is used to record performance counters in the trace, the total overhead of 95μs is very high[1]. The overhead per call is in the order of the round-trip time for MPI on our cluster. Therefore, communication of small messages is notably deferred by this extension.

[1] The overhead is caused by the current implementation, which requires re-initialization of performance counters between two subsequent (`MPI_Barrier()`) calls.

Configuration	Wall-clock time	Time for the loop	Time/barrier
Open MPI	0.197 s	0.011 s	11 ns
MPICH2	0.038 s	0.022 s	22 ns
HDTrace	3.12 s	1.37 s	1.37 μs
VampirTrace	5.46 s	1.82 s	1.82 μs
HDTrace-Likwid	117 s	95.00 s	95.00 μs

Table 7.1.: Time overhead of HDTrace and VampirTrace.

Configuration	Size of the trace files	Size of bzip2 compressed trace files
HDTrace	49 MiB	3.7 MiB
OTF	53 MiB	7.1 MiB
HDTrace-Likwid	300 MiB	17 MiB

Table 7.2.: Size of unpacked and packed trace files.

A tracing environment must output trace information to a file, to reduce this overhead both libraries buffer activity. When the buffer capacity is reached, it is written out to the file system, this is time-consuming and defers further activity of the process. Since the benchmark only measures the time of the whole loop, a potential flush that happens in the loop is accounted to all events. The additional overhead to finalize the library is not measured by this benchmark.

Storage requirements The size of the recorded trace files is given in Table 7.2. Roughly 50 bytes of data is stored in the traces per barrier. Compression with `bzip2` noticeably reduces the amount of stored data, this works even better for the XML file. Note that OTF natively offers the capability to compress files on the fly with *zlib*[2]. If this is enabled, then the OTF trace file is compressed to 8.2 MiB. HDTrace does not compress recorded data.

In general, HDTrace files are expected to be bigger than OTF files, because all parameters of the MPI calls are recorded and the length of the timestamp increases in longer running programs. However, in this simple benchmark *HDTrace*'s overhead – in terms of space and time, is comparable to *VampirTrace*.

Summary and Conclusions

The overhead of HDTrace is assessed by measuring system behavior upon invoking `MPI_Barrier()`. Since the collective call is invoked with just a single process, the time spent inside MPI is very low – about 10-20 ns, and the overhead is high; on average, HDTrace adds 1.3 μs to each call. The overhead is expected to be similar for other MPI calls. To put it into context, this value is about three times the duration for an intra-socket message transfer of an empty message (0.4 μs) but much smaller than inter-node latency (about 0.4 ms). Thus, the overhead matters only for measuring local inter-process communication of small messages. Measurements with VampirTrace lead to a similar overhead, proving that the chosen design is not inferior to state-of-the-art tracing tools. During the later experiments, tracing is disabled for experiments in which small messages are exchanged, therefore, the overhead is negligible during the validation experiments.

7.2. Performance of PIOsimHD

Performance of the simulator is important, because a simulation study could involve 100,000 of processors each running 1,000 of commands. Each simulation run consists of three phases: Model creation, the

[2]`http://zlib.net/`

initialization of the simulation model, and the event processing by the simulator. The time for initialization and model creation depends on the complexity of the network topology, the components the model consists of, and the parsing of commands if commands are loaded into memory.

The time needed to create an application model with multiple commands increases with the number of commands linearly. Since the performance of running a sequence of multiple operations is expected to be similar to running each command in isolation, it suffices to determine performance for a single command. To assess simulation performance, the raw event processing speed and the scalability of the simulator phases is evaluated. Experiments measure the performance of inter-node communication, which is the basis for I/O as well. To reduce the number of events to process, two available alternative NIC models are evaluated, too. All experiments are encoded in Java and use builder classes to set up the system model; they rely on the introduced system model for our Westmere cluster that uses a StoreForwardMemoryNode to model processor sockets.

Performance of I/O simulation is not analyzed explicitly because measurable speed of simulated I/O depends on the chosen cache layer and its implementation. For instance, picking the next operation out of N pending operations has currently a worst-case complexity of $O(N^2)$ when using the *AggregationCache* implementation – this cache tries to merge the first pending operation with all other pending operations in a linear fashion. The actual duration of picking the next operation is proportional to the number of fused operations, but the complexity to process all N operations yields this bound. Therefore, execution typically slows down when choosing this implementation. However, during the whole processing of the operations the other cache layers yield an average-case complexity of $O(N)$ (for NoCache) or $O(N \cdot \log(N))$ (for AggregationReorderCache). The additional complexity of $\log(N)$ of the *AggregationReorderCache* is caused by managing operations in a heap data structure in order to fuse them.

Event processing performance is measured on a laptop equipped with an Intel i7-640M (2.8 GHz) and the scalability on an AMD Opteron system. The Opteron system is equipped with four 6168 processors running at 1900 MHz and 128 GByte main memory. Both systems run under Ubuntu 11.10 (64-bit), and OpenJDK 1.6_23 (IcedTea6) with the server VM[3]. Since the Laptop supports Intel's Turbo Boost technology, which could affect shorter experiments, the CPU governor is set to userspace and the frequency is fixed to 2.8 GHz.

7.2.1. Event processing speed

To measure the speed of the event processing, a simple experiment is conducted: One process sends 100 GiB of data to another one; both are hosted on a single node. In the experiments the NIC model fragments data into chunks of 100 KiB resulting in more than 10 million processed events. The implementation of the simulator uses asserts to check the correctness of parameters and it offers debugging capabilities of the internal states. These features have an impact to performance and are therefore evaluated, too.

Table 7.3 provides the measured times – every test is repeated three times and the average time is listed in the table. Reported time concentrates on the actual processing of the events, that means time to build a model is excluded. The compute time variance is small, less than 2 % for long runs. For runs without debugging calls, a variation of 5 % can be observed. The variability of wall-clock time increases to more than 10% with TurboBoost enabled.

If debugging calls are commented out, on average 1.5 million events are processed by the simulator per second. The impact of removing debugging calls is tremendous – with enabled debugging only about 45,000 events/s can be processed, which is a factor of 30. With instrumentation it has been found that the string processing in Java slows down execution: Whenever two strings are concatenated a *StringBuffer* object is instantiated. Processing of asserts needs some time, too (even if they are disabled). Therefore, scripts are available that comment out asserts and the invocations of debugging messages. To evaluate the impact, those scripts have been applied to clean the code.

[3]The times on the Opteron system were measured with a slightly newer Java version.

Assertion calls	Debugging calls	Wall-clock time	Events per second
Enabled	Disabled	426 s	44,200
Disabled	Disabled	423 s	44,600
Commented out	Disabled	418 s	45,200
Disabled	Commented out	12.5 s	1,500,000
Commented out	Commented out	12.5 s	1,500,000

Table 7.3.: Event processing speed of PIOsimHD.

Therefore, once the model and simulator is verified it is recommended to remove debugging calls from the simulator. The impact of asserts is not measurable with OpenJDK and those should be kept[4]. Preliminary tests with the JDK from Sun showed similar improvements, but with this JDK disabling of assertions improved performance, too.

Note that the speed of the simulator, as measured by the number of events processed per second, is in the order of state-of-the-art simulators such as the *LogGOPSim* [HSL10].

7.2.2. Scalability

To assess scalability, an `MPI_Bcast()`[5] with 100 MiB data is simulated with an increasing number of nodes; each node in the cluster model hosts exactly one process[6]. All three phases of simulation are measured: The time to create the cluster model with one socket per node, the time to instantiate the simulation classes, and the time to execute the simulation.

Memory consumption (resident set size) of PIOsimHD has been determined after pausing execution once the simulation model is instantiated and it is ready for simulation[7]. Measured values for 1024, 2048, 4096 and 65536 processes are 198 MiB, 256 MiB, 322 MiB and 2232 MiB, respectively. Therefore, the amount of memory per simulation process decreases to roughly 30 KiB of memory. During the execution the amount increases because internal states must be kept, especially packet fragmentation over the whole network topology requires additional intermediate objects. For example, the high-watermark for used memory during the whole simulation run is 254 MiB, 345 MiB, 1239 MiB and 4763 MiB for 1024, 2048, 4096 and 65536 processes, respectively. There is a high fluctuation in the maximum amount of memory occupied, for example, for 2048 processes three measurements resulted in: 303 MiB, 345 MiB and 397 MiB. Probably the irregular execution of the garbage collection causes this variation, objects that are eligible for garbage collection and not referenced further can still exist and the garbage collection can decide to allocate new memory. Since it is not possible to force the garbage collector to clean all unused memory objects, the observed memory usage is above the actually needed amount of memory. For that reason the observed values must be taken with a grain of salt.

The times obtained from the Opteron system are presented in Figure 7.1a. With 65536 processes, building the model takes 6.3 minutes, initialization of the simulation model finishes after 5 minutes and the simulation of the single `MPI_Bcast()` needs 85 minutes. To assess scalability, the relative run-time per process is given in Figure 7.1b; with perfect linear scaling the times would produce horizontal lines in the diagram. The relative times of the simulation line are determined dividing the measured run-time by the number of receiving processes, which is one less than the total number of processes because the root process broadcasts its data.

[4]Coded assertions can be enabled in Java by specifying `-ea` to the *Java application launcher* (`java`).

[5]The implementation of the broadcast uses a binary tree algorithm to distribute data among all processes; if a node must send data to multiple receivers, it will send these messages concurrently. Therefore, the number of transferred messages is one less than the total number of processes.

[6]Internally, the congestion model is used, therefore, the NIC fragments data into 100 KiB chunks.

[7]Data has been fetched from `/proc/<PID>/status`.

Model creation becomes faster with an increasing number of clients and starting with 4096 processes it saturates at roughly 5 ms per receiving process (look at Figure 7.1b). Simulation performance improves up to 128 processes, then time to simulate a single process doubles between 128 processes and 65536 processes which is not perfectly linear but almost.

For model creation and simulation of this experiment, a linear scaling is desired and expected: The total amount of data that is transferred scales linearly with the number of processes that receive data, which is the total number of processes minus one (the initial root process). Thus, the number of processed network packets and created events is linear as well[8]. Also, efficient data structures are used in the implementations, for example, heaps manage events and pending operations on the components. In combination with a low number of pending events per component, performance of these data structures behave linearly. The reason for the "superlinear" scaling with lower number of processes is unknown.

Time to initialize the simulation model shows a completely different behavior (the red lines in the figures): Up to 128 processes time decreases like with the other phases; starting with 8196 processes, the times per process becomes quadratic. This indicates that during this process quadratic complexity is involved. This is a bit surprising, because mainly, the simulation model initialization performs operations that scale linear with the number of model components (including network edges and nodes). The reason for performance degradation with high number of processes is not completely clear. One reason might be that the routing algorithm does not scale linearly (the routing algorithm is invoked after all components are initialized). Therefore, the routing algorithm is briefly discussed.

Two alternative routing implementations have been evaluated:*first route* and the *hierarchical route*. The times of both are included in Figure 7.1a. The *first route* algorithm is implemented naively with a *breadth-first-search* (BFS). For every process the BFS is started; it updates the routing information to this process. Therefore, it yields an average complexity of $\Theta(N^2)$ (where N is the number of processes). This results in a very high runtime, simulation initialization needs 9000 s for 8196 processes. Therefore, more efficient routing algorithms had to be deployed.

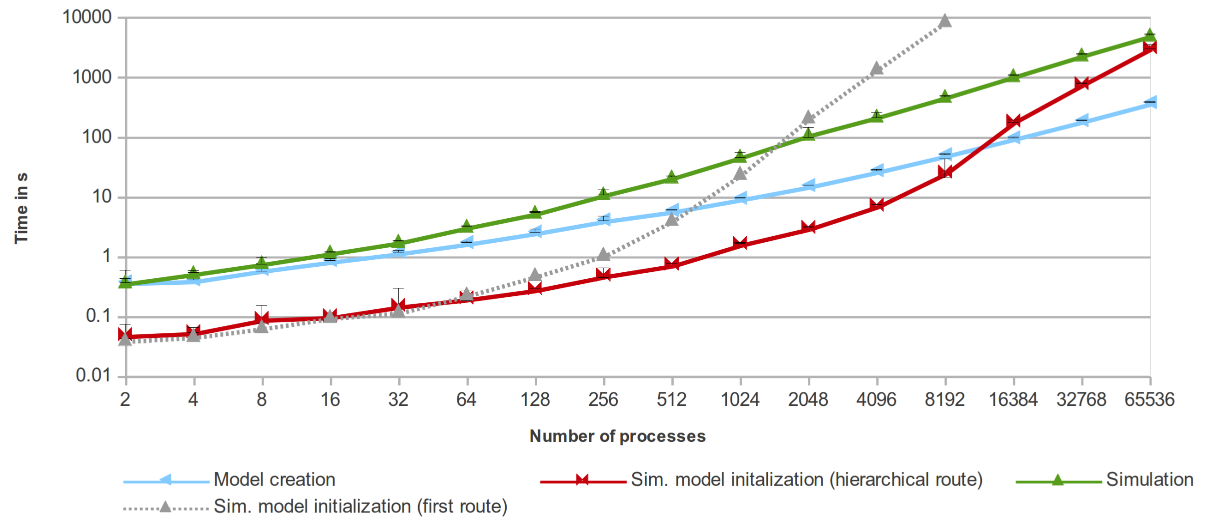
Theoretically, a routing algorithm could be used that matches the given topology to avoid creation of routing tables at all. However, this restricts the topology that can be created by the user. Since the network topology in this experiment does not provide alternative routes, a variant of the general purpose routing algorithm has been implemented that exploits all kind of hierarchical topologies. The *hierarchical route* algorithm creates default routes from the processes towards the central switch. Compared to *first route* this algorithm conserves computation time and memory. Internally, the hierarchy is formed by determining the distance from a network node to all processes with a single BFS. For every inner node a default route is set towards the node that is more distant (towards the center); the routing table of this node is the last table that must be adjusted. Routes to other nodes than the default route are determined with the approach of *first route*. Theoretically, in a switched topology this algorithm yields a complexity of $\Theta(N)$.

However, starting with 8196 processes, the runtime becomes quadratic, again. Further performance analysis is required to identify the cause. While scalability of the model initialization is currently limited, the author claims a scalable implementation is possible with small modifications of the current code base.

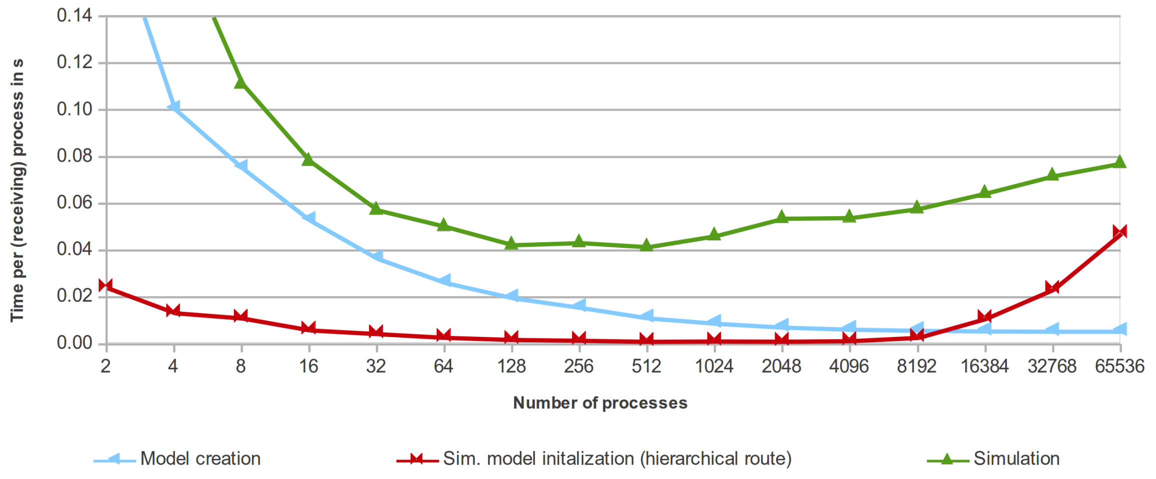
7.2.3. Alternative NIC models

Since processing of the broadcast operation takes a long time on large configurations, alternative communication models are investigated that reduce the number of events to process; the simulation of the network traffic is evaluated for three different models: The congestion model, the analytical NIC, and transfer of just a single packet per message with the analytical NIC. Note, that the time to build a model is independent of the chosen NIC model.

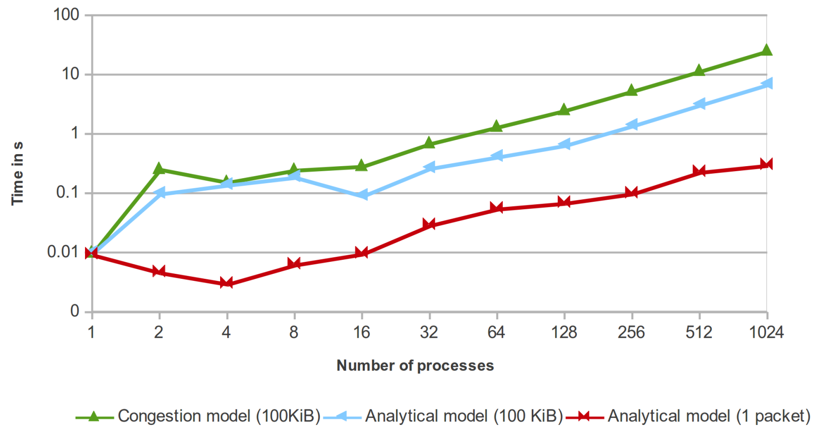
[8] The total number of created events with $2^2 = 4$ processes is 80,047 events and with 2^{16} processes 1,748,539,339 events, therefore, the number of events per receiving process is constant at about 26,680.



(a) Times measured on the Opteron system including error bars for minimum and maximum. Two different routing algorithms are evaluated during initialization of the simulator.



(b) Scalability – time per process measured on the Opteron system. For the simulation line, the time is relative to the number of receiving processes ($N - 1$).



(c) Simulation time of different NIC models measured on the Laptop.

Figure 7.1.: Performance of the simulator executing a binary broadcast operation.

The simulation of a single point-to-point message exchange depends on the chosen NIC model. The default NIC model fragments data into packets with a size of network granularity that are transmitted with the flow protocol. When this congestion model is applied, the number of events depends on the message size and the network granularity. The analytical NIC model processes packets only on sender and receiver side, therefore, intermediate congestion is not covered[9]. This model reduces the number of events because intermediate nodes do not process the packet. Many other simulators use analytical models to estimate communication time. Therefore, by using a large transfer granularity and the analytical NIC model a similar setup is generated.

In this experiment times from the laptop are presented that are measured for up to 1024 processes. As expected, the total number of executed events roughly doubles with the number of processes and so does the execution time (see Figure 7.1c). Performance of the analytical model is better by approximately a factor of 5. When just a single packet is sent, the execution time is much faster because the number of packets is decreased by a factor of 1,000 (compared to using a transfer granularity of 100 KiB). As this trend is expected to continue for larger configurations, the alternative NIC models can be used to reduce simulation time – by reducing accuracy.

7.2.4. Extrapolation of performance

Runtime of larger systems is extrapolated and discussed based on measured results. First, by extrapolation a memory usage of 5 GiB to simulate `MPI_Bcast()` on 65,536 processes, 1.5 million processes could be simulated on the working groups Opteron system. Additional commands, representing an application program, that are kept in memory increase the occupied space. In order to use the feature of the simulator to load trace files on demand, the maximum number of open files must suffice, which is unlikely for 1.5 million processes. Since simulating that large number of processes would take a prohibitively large amount of time, the memory utilization is ignored in the following discussion.

The time to build a cluster model and to instantiate the simulation model are neglected in this consideration because theoretically model creation and instantiation could scale linearly with the number of created objects. If the measured 0.08 s per process were valid for 1.5 million processes, then it would take 33 hours to simulate the single `MPI_Bcast()`. Therefore, simulation of programs that execute thousand or millions of MPI commands is infeasible. Program execution requires simulation of multiple commands and thus a fast execution of individual operations is important. By applying an analytical model, the time to simulate the single command decreases by a factor of 1000 – to 2 minutes and thus the behavior of small programs can be analyzed.

For a medium scale of 1,000 processes, simulation on the Opteron node takes 40 s with the network granularity of 100 KiB. With this model network congestion can be approximated well and interaction of small programs can be investigated. Since hybrid applications spawn less processes and use threads for the computation, it suffices to create less processes per socket. This also reduces the number of MPI processes to simulate and thus complexity by an order of magnitude. Thus, the demonstrated scalability of the congestion model suffices to simulate runs with 1,000 processes.

An easy way to achieve better scalability is to rely on homogeneous hardware characteristics and fixed topologies as implemented by most existing cluster simulators. Since PIOsimHD is able to model arbitrary network topologies and heterogeneous components, this optimization potential cannot be exploited. While a parallel simulation is desirable, achieving scalability is difficult because modeling of network congestion and I/O scheduling algorithms imply data dependencies. These dependencies, in turn, restrict expected benefit of conservative synchronization algorithms and the overhead of optimistic approaches. Also, one design goal of the simulator is to allow inspection of the global system state in order to evaluate best-case selection of collective algorithms and I/O algorithms. Since a global state requires synchronization, this approach becomes prohibitive in a parallelized simulator.

[9]The NIC models are described in Section 5.2.3.

7.2.5. Summary and Conclusions

The performance and scalability of the simulator is evaluated demonstrating that the simulator is able to process about 1.5 million events per second. This raw sequential performance is comparable to other simulators such as LogGOPSim, which is written in C. To evaluate scalability, the time for simulating an `MPI_Bcast()` operation with a variable number of processes between 2 and 65536 is measured. The time needed for model creation and simulation are almost proportional to the number of processes. This is expected because the number of generated events is proportional to the number of processes and the creation of the model depends on the number of components. Therefore, the selected data structures and the implementation are appropriate. However, the initialization of the simulation model takes more time. Partly, this is due to the setup of the routing table – further optimization of this process is necessary.

With 16384 processes, it takes about 1000 s to simulate the single broadcast operation, about 60 ms for a single process. In the experiment, the `MPI_Bcast()` transfers 100 MiB of data per process; message data is fragmented into 100 KiB packets. These packets are transferred with the network flow model from source to sink. When the analytical NIC model is used instead of the congestion model, many events can be saved because intermediate network components do not actively participate – this reduces simulation time by a factor of 4. By sending a single packet with the analytical model, the number of events is reduced by a factor of 1,000 and the simulation time by two orders of magnitude. This performance analysis shows that although PIOsimHD is a sequential program, it is suitable for simulating applications running on medium sized clusters – a simulation of thousands of processes with network congestion is feasible.

7.3. Parameterization

For our Westmere cluster, a representing model is derived from the introduced domain model for cluster supercomputers by parameterizing the network topology, network edges and nodes, compute performance, memory and the block device. Most model parameters are directly derived from the characterization experiments described in Chapter 3, the others are determined with the help of product datasheets offered by vendors.

7.3.1. Network Topology

A model of our clusters' network topology is shown in Figure 7.2. In this star topology, one central switch interconnects all available nodes with Gigabit Ethernet. Within a node, all hosted processes and their network interfaces are connected via a fast connection with a specification according to *Intel's QuickPath Interconnect* [Int09]. Besides the QPI node in a compute node, all other network nodes are modeled by a *StoreForwardNode*.

While there is no direct simulation of memory access, it is an important factor to estimate performance of intra-node communication. In the system architecture of a compute node, all PCI(-Express) components, such as the NIC, are connected to the *I/O Hub* (IOH) via QuickPath. The *Nehalem* microarchitecture actually offers up to three memory-channels per socket. Thus, in our cluster, the local node communication can be represented by one "memory" node per socket; this QPI node simulates memory access of communication, and also shares the memory bandwidth when local data transfer happens – hence it is implemented by the *StoreForwardLocalNode* model. Refer to Page 227 for a description of this model. Consequently, the network topology shown in Figure 7.3 represents the system architecture better and is used for simulation experiments. In this model, memory is shared just between processors of a socket; memory nodes connect to the I/O Hub which is represented by the QPI node.

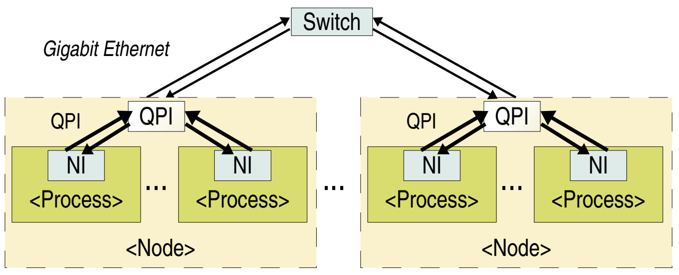


Figure 7.2.: Simple model of the cluster's star topology with a fast node-internal communication and slower inter-node communication. The number of nodes and processes per node can vary.

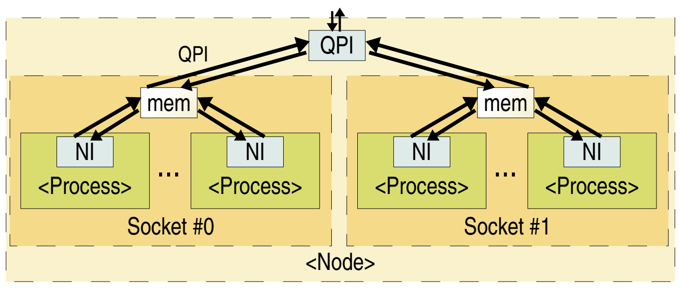
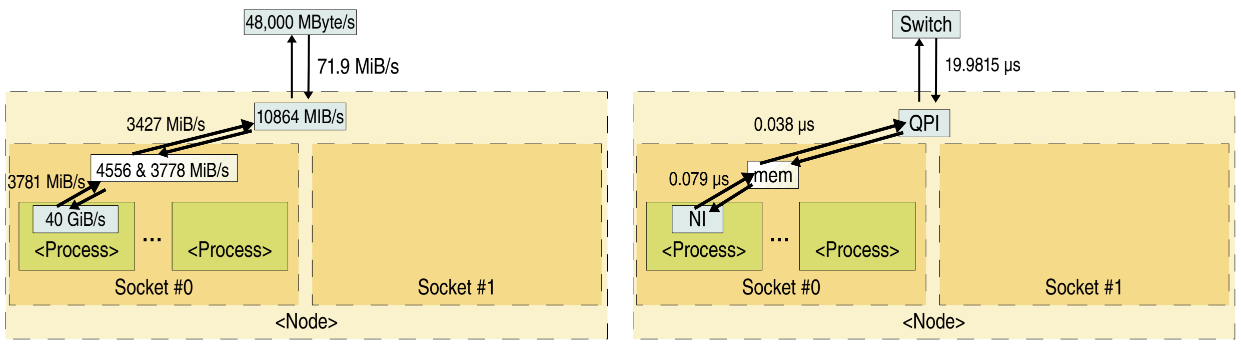


Figure 7.3.: Refined network topology of a node to represents the memory limitations in a dual-socket system better. The *mem* nodes limit the actual communication performance by memory bandwidth and latency. The QPI node represents the I/O-Hub.



(a) Throughput of edges and nodes; The memory node transfers messages faster between processes of the local socket.

(b) Latency of edges

Figure 7.4.: Schematic network topology of the cluster model with annotated characteristics.

7.3.2. Network Edges and Nodes

Throughput must be selected for all nodes and edges in the model's network topology (Figure 7.3) and latency for all edges. To derive the required parameters, mainly characteristics measured for MPICH2 are consulted: By picking average latency and throughput as observed in the experiments in Section 3.5, the model should resemble behavior of the MPI implementation.

Throughput In Figure 7.4a the selected throughput for nodes and edges is annotated next to them. Throughput is directly taken from the experiments measuring point-to-point throughput between two processes with the PingPong kernel. Three configurations have been analyzed: inter-node communication, inter-socket communication and intra-socket communication. According to Table 3.4, achieved throughput for this process placement is 71.9 MiB/s, 3427 MiB/s and 3781 MiB/s, respectively. However, the bi-directional intra-socket performance is better than uni-directional performance. Since the *mem* node represent the memory bottleneck, the larger bi-directional throughput of 3778 MiB/ and 4556 MiB/s is chosen to represent the memory node for inter-process and intra-process communication, respectively.

NICs have a very high throughput of 40 GiB/s to avoid interference with the other models. Edges are annotated with the uni-directional throughput. For the model the throughput of the QPI node interconnecting both sockets is limited by the theoretical QPI performance of roughly 10 GiB/s – any very high value would work, because node edges have a lower limit.

Determining latency The latency of the network edges can be reconstructed from the experimentally determined average latency measured by `mpi-network-behavior`. The observed MPI latency is 42.198 μs, 0.4849275 μs and 0.2154 μs for inter-node, inter-socket and intra-socket communication, respectively (refer to Section 3.5.3). To determine the latency for edges, observed network latency must be attributed to all links of a communication path. Time to process an empty message (at least 40 bytes) with the throughput of intermediate nodes and edges must be taken into account because data is copied between the network nodes.

First, the latency of NIC edges is computed. Due to the store-and-forward scheme, two edges and a memory node are involved in the data transport and the processing time of them must be subtracted: NIC_EdgeLatency = $0.2154\mu s/2 - 2 \cdot 40$ bytes/3781 MiB/s – 40 bytes/4556 MiB/s = $0.1077\mu s - 0.00837\mu s - 0.02\mu s = 0.0791488\mu s$. Processing time on the model's NIC nodes is much faster and thus can be neglected.

With the same scheme, latency of the other edges can be computed; latency between two sockets on one node is: SocketEdgeLatency = $0.4849275\mu s/2 - 2 \cdot$ NICEdgeLatency $- 2 \cdot 40$ bytes/3778 MiB/s $- 2 \cdot 40$ bytes/3427 MiB/s – 40 bytes/10864 MiB/s = $0.038198\mu s$, and the latency of the Ethernet edge is given by: EthernetEdgeLatency = $42.198\mu s/2 - 2 \cdot$ NICEdgeLatency $- 2 \cdot$ SocketEdgeLatency $- 2 \cdot 40$ bytes/3778 MiB/s $- 2 \cdot 40$ bytes/3427 MiB/s $- 2 \cdot 40$ bytes/10864 – 40 bytes/48,000 MByte/s = $19.9815\mu s$. Since the simulator operates at an accuracy of nanoseconds, times are rounded accordingly. An overview of the model's latencies is given in Figure 7.4b.

7.3.3. Compute Performance

During the execution of commands in the simulator, their run-time is approximated by the processor speed and the number of instruction performed[10]. In the model, the actual speed is set to 2.66 GHz, although *dynamic voltage and frequency scaling* (DVFS) of the CPU is activated on our cluster. Since this setup reproduces the exact times of compute jobs, that is, the measured run-time and the simulated time are identical, the impact of DVFS on the simulation results can be ignored for the conducted experiments.

[10]Refer to Page 5.5.1 for more details.

7.3.4. Memory

Each node is equipped with 12 GByte DDR-3 main memory. On a real system this memory is used to run applications, to manage administrative structures in the operating system, and to cache data on storage nodes. In the system model of the simulator, available memory is currently used only for caching I/O on servers. In order to reduce the total amount of data that must be written until data is actually flushed to disk, in most test cases the available memory is restricted to 1 GiB. Memory throughput and latency is already discussed since those parameters influence the point-to-point intra-node communication.

7.3.5. Hard Disk Drive

To parameterize the *RefinedDiskModel*, five values must been chosen: RPM, short seek distance, sequential transfer rate, track-to-track seek time and average seek time.

Most information is available in the datasheet [Sea10][11]. While RPM specification can be directly taken from the datasheet (7200 revolutions per minute), the other values are subject to additional overhead of the local file system. These values are discussed and assessed together with experimentally determined performance in the following.

Estimating short-seek distance Short-seek distance of the model is considered to represent the amount of data at which a seek to a neighboring track is performed with the track-to-track seek time. Data per track is the only physical value which could not be measured and must be derived based on the vendor's information instead. To estimate the average amount of data which can be stored per track, information from the data sheet is used: The *recording density* (1413 kb/in) specifies the maximum number of bits which are recorded in one inch of a track, the *track density* (236,000 tracks/in) indicates the number of tracks per inch of the radius, the *areal density* (329 Gb/in^2), which is the number of information bits per area, and *form factor* (3.5 in) indicates the size of the platter [12]. Only one head and one platter is used. For simplicity, it is assumed that data is stored to the outer border (up to 1.75" distance from the center of the platter). However, the innermost circle cannot be used due to the spindle and the read/write head. Note, that a hard disk platter might a bit smaller than the form factor. The size of the inner circle is unknown but can be approximated with the areal density and the used space from the data sheet with the following equation: usedAreal $= \frac{250\,\text{GB}\cdot 8\,\text{bit/bytes}}{329\,\text{Gb/in}^2} = 6.08\,\text{in}^2 = \pi \cdot (1.75^2 - r_{inner}^2)$. Hence, the radius of the unused inner circle is about 1.08 in, which seems very high.

The average track length is about $(3.5\,\text{in} + 2\,\text{in})/2 \cdot \pi = 8.64\,\text{in}$. Therefore, the average amount of data in a single track is $7.64\,\text{in} \cdot \frac{1413\,\text{kb/in}}{8\,\text{bit}} = 1.5\,\text{MiB}$. The outermost track (at 1.75 in) could hold at most 1.9 MiB, since the recording density specifies the maximum value.

The amount of data per track can be also calculated by using track density and device capacity: $\frac{250\,\text{GB}\cdot 1000\,\text{MB/GB}}{236000\,\text{tracks/in}\cdot(3.5-2)/2\,\text{in}} = 1.4\,\text{MB/track}$. These calculations are just approximating the amount of data per track. Since the identical specifications are also used for a HDD with a capacity of 160 GB, it is likely that some sections of the disk are not accessible by the OS – it might be reserved for spare sectors and/or to improve the production yield, the sections might account for defects during the manufacturing process. If data would be stored across the whole radius, then a density of about 0.6 MB/track could be achieved. For simplicity, a compromise of 1 MiB/track is chosen for the disk model.

Throughput Under the assumption that all data of a track can be accessed in one revolution of the 7200 RPM drive, and by assuming 1 MiB/track, the sequential read rate would be 120 MiB/s. The data

[11] The values of the datasheet are introduced in Section 3.6.1.

[12] The values provided by the manufacturer are expected to include the overhead of the data-encoding scheme to store bits in magnetic fluxes. Actually, inner tracks carry less data than outer tracks; since the average short-seek distance is used this is ignored.

sheet [Sea10] mentions 125 MByte/s, which is about 119 MiB/s and thus the result indicates that the basic considerations about the capacity per track match.

The experiments presented in Section 3.6 revealed a rich fluctuation of I/O throughput with some unintuitive results. For example, a sequential read and write throughput of 96 MiB/s is observable with direct I/O for small record sizes up to 128 KiB, but for larger records, performance decreased to 75 MiB/s. An excerpt of the measurements is given in Table 7.8 and in Table 7.9 for IOZone and `posix-io-timing`, respectively. Sequential reads with IOZone from a re-mounted file system achieved a performance of about 96 MiB/s, independent of the record size and the file size (see Table 3.8). Several configurations which actually hit the available disk, are evaluated in this table. In experimental results, performance of PVFS is determined, which uses cached I/O. Thus, the IOZone results are considered to be representative. Therefore, a sequential transfer rate of 96 MiB/s is assumed for the disk drive.

Actually, with the I/O model, the maximum throughput can be achieved iff data is accessed sequentially. In case file offsets do not match, an additional latency is added as determined by the RPM and seek times.

Seek times The datasheet of the HDD lists a typical track-to-track seek time for reads “below” 1 ms, and an average seek time of around 8.5 ms, the values for writes are 0.2 ms and 1 ms higher, respectively. Since the *refinedDiskModel* takes only one value, the average of read and write is computed and results in a track-to-track seek time of 1.1 ms and an average seek time of 9 ms.

To cross-validate model values with the datasheet, assume that the disks head must move after accessing a track of 1 MiB with a seek time of 1.1 ms, this decreases the theoretical sequential throughput of 119 MiB/s to 106 MiB/s (computation is shown in Equation 7.6). This still matches with our observations, because the model’s sequential transfer rate of 96 MiB/s is lower. For sequential access, the simulation does not add these seeks so that the model’s sequential transfer rate is the same as the measured one.

The average seek time is only valid when the accessed data is distributed across the whole hard disk. But, in all experiments, not more than 5 GiB of data is accessed, which is just a fraction of the disk’s capacity of 250 GByte. Therefore, the average seek time of 9 ms is expected to be too high. There are reasons that the exact duration of a seek is unknown: The heads of an HDD are accelerated depending on the distance from the current to the target track, the amount of data on a track depends on the track’s length, and, at last, the file system places data of files on arbitrary tracks. Consequently, without simulating the whole file system and disk in detail, a perfectly accurate model is not possible. Therefore, the *RefinedDiskModel* just takes the character of those two seek times as representative behavior and the values provided by the datasheet are used. The influence of the seek time is assessed in more detail during the following validation process.

7.3.6. Overview of the Parameters for the Experimental Cluster Model

A summary of the parameters used in the cluster’s system model is given in Table 7.4. Characteristics of network edges and nodes are also annotated to the virtual network topology in Figure 7.4 to show where they are applied.

7.3.7. Summary and Conclusions

In the parameterization process, a network topology is created that matches the configuration of our dual socket nodes – it consists of a hierarchy for intra-socket, inter-socket and inter-node communication. For each socket, a memory node models local communication and thus restricts performance of node-internal inter-process communication. By using the characteristics measured during system analysis, the parameters for the simulation model are selected: Throughput is directly annotated to nodes and edges; about 4000 MiB/s for local data access and 72 MiB/s for Ethernet edges. The latency of edges is computed hierarchically starting from intra-socket communication: 0.08 μs, 0.04 μs and 20 μs for intra-socket, inter-socket and Ethernet edges, respectively. During the computation of these values, the throughput of the edges and

Component	Metric	Value
Node	CPUs	12
	instructions per second	$2.66 \cdot 10^9$
	memory size	1 GiB
I/O subsystem	track-to-track seek time	1.1 ms
	average seek time	9 ms
	sequential transfer rate	96 MiB/s
	short seek distance	1 MiB
	RPM	7200
NI node	throughput	40 GiB/s
Memory node	intra-socket throughput	4556 MiB/s
	external throughput	3778 MiB/s
QPI node	throughput	10864 MiB/s
Switch node	throughput	48,000 MiB/s
NIC edge	latency	0.079 μs
	throughput	3781 MiB/s
Socket edge	latency	0.038 μs
	throughput	3427 MiB/s
Ethernet edge	latency	19.982 μs
	throughput	71.9 MiB/s

Table 7.4.: Parameters for the hardware components in the system model of our Westmere cluster.

nodes and the latencies of the edges already computed are taken into account. Computation speed is set to the processor frequency of 2.66 GHz. The amount of available memory is set to a value that matches the available cache in the experiments – memory is limited depending on the experiment configuration. Parameters for the RefinedDiskModel are partly taken from the datasheet: 7200 RPM, track-to-track and average seek times (1.1 ms and 9 ms). In all I/O experiments, a file with several GiB of data is accessed. Therefore, the average seek time is an overestimation of expected seek time; however, it turns out that the values still lead to acceptable behavior. The short seek distance is computed as the amount of data stored on a single track (1 MiB). For throughput, the measured value of 96 MiB/s is used.

7.4. Qualification of the Domain Model

In this section the hardware model for point-to-point network communication and the hard disk drive, which is introduced in Section 5.2, are qualified against the obtained system characteristics from Section 3.6.

To this end, the underlying analytical models of the simulator are extracted and compared to observations. This allows assessing the accuracy of the introduced models and to derive expectations to the simulator implementing those models[13].

Since the analytical models are simplifications of the inner processing of the simulator, they do not cover certain dynamics which are implemented by the simulator, such as network congestion or disk caching. Instead, in this qualification, underlying mathematical models for network and disk behavior are validated against the measurements for basic behavior of point-to-point communication and disk I/O.

[13]Remember that a perfect matching of reality and a model is not possible because a model abstracts from reality.

Configuration	Latency	Throughput in MiB/s	
		Uni-directional	Bi-directional
Inter-node	42.198 μs	71.9	71.9
Inter-socket	0.485 μs	3427	3778
Intra-socket	0.2154 μs	3781	4556

Table 7.5.: Characteristics for the analytical network model.

7.4.1. Network Behavior

To validate the network behavior, point-to-point communication of the model is compared with inter-node, inter-socket and intra-socket message transfer of MPICH2 for the PingPong and Sendrecv (bi-directional) kernels.

First, the analytical model is recapitulated: The network model implemented in the simulator uses latency and bandwidth to characterize individual links and intermediate nodes. In a simplified model, such as in the LogP family (Page 107), the time for communication of a given message size is estimated by a single homogeneous latency and throughput as shown in Equation 7.1 – all communications are treated identically and congestion is not covered. For a concurrent bi-directional communication the equation can still be applied because both channels operate at the given speed. An estimation of the PingPong kernel doubles the obtained time (see Equation 7.2).

Network nodes representing memory share their performance among all concurrent operations, therefore, the amount of data transferred is twice as much as for a uni-directional link. Thus, for intra-node communication with the Sendrecv kernel, Equation 7.3 is applied.

The analytical model discussed here does not cover the distinction between the eager and the rendezvous protocol since it is not too important on our cluster – this will become clear when the results are compared. However, the simulator supports both communication schemes and it includes a parameter that controls when the rendezvous protocol should be used.

Remember, the computed latency of the simulator model is corrected by the size of an empty message (40 bytes) and the link topology for our cluster (refer to Table 7.4 to see the performance characteristics of the network). With the time, the throughput can be estimated as listed in Equation 7.4. The analytical model represented by the equation uses the original values obtained by the hardware characterization – these values are recapitulated in Table 7.5.

$$\begin{aligned}
\text{estimated Sendrecv time} &= \frac{\text{message size}}{\text{latency} + \text{message size/maximum throughput}} && (7.1)\\
\text{estimated PingPong time} &= 2 \cdot \textit{estimated Sendrecv time} && (7.2)\\
\text{estimated Sendrecv time (memory)} &= \frac{\text{message size}}{\text{latency} + 2 \cdot \text{message size/maximum throughput}} && (7.3)\\
\text{estimated throughput} &= \frac{2 \cdot \text{message size} + 40\,\text{bytes}}{\text{estimated time}} && (7.4)
\end{aligned}$$

For every kernel and placement of the processes, a graph that contains throughput of the measurement and the model estimate is generated by using Equation 7.4. Graphs for inter-node, inter-socket or intra-socket are shown in Figure 7.5, 7.6 and 7.7, respectively. In intra-node communication, a comparison of results between memory throughput – as measured with the uni-directional PingPong kernel, and the bi-directional kernel is performed. The average and minimum times of the first run measured with MPICH2 are used for comparison[14].

[14]The first run is used for comparison, because the average value across all runs is expected to be closer to the model, which is built with the average latency and throughput across all runs. Therefore, by comparing model and measurements for a single run, deviation can be assessed better. The variation between independent runs is discussed in Section 3.5.5.

Configuration	Kernel	Relative accuracy in %	
		Empty message	128 MiB message
Inter-node	PingPong	128 (average 99.6%)	100.3
	Sendrecv	102	85 (average 112%)
Inter-socket	PingPong	109	113
	PingPong	109*	103*
	Sendrecv	93	103
Intra-socket	PingPong	109	129
	PingPong	108*	107*
	Sendrecv	86	107

Table 7.6.: Relative model accuracy of the analytical network model compared to the minimum measured duration of the first run with MPICH2. The values marked with * are estimated by applying the higher bi-directional throughput characteristics

An additional graph is generated showing *relative accuracy* which is defined as estimated time of the model divided by the measured time. Therefore, a value above 100% indicates that the approximation is below measured throughput, i.e., too slow, and a value below 100% underestimates the runtime, that means the model is too fast.

The relative accuracy of the two kernels is shown for empty messages and for 128 MiB messages in Table 7.6. For inter-node communication, the relative accuracy of the average throughput is mentioned, too, because it differs significantly from the best case.

Observations and interpretation

a) Qualitatively, the throughput diagrams highlight that the simple network model roughly recreates the measured behavior (for example look at Figure 7.5a).

Especially for empty and 128 MiB messages, the estimation is close to the observed average; the error is typically within 10% (look at Table 7.6). This is expected because the latency and throughput has been chosen by looking at the performance of these cases. Consequently, all relative diagrams for the PingPong kernel should match well for empty and large message payloads.

b) Besides the range between 1 KiB and 16 KiB, the estimate of the PingPong kernel is close to 100% for inter-node communication (see Figure 7.5b). For small and large payloads some messages arrive earlier than the average, pushing the error relative to the MPICH2 minimum above 100%.

One reason for the overestimation between 1 KiB and 16 KiB could be the Ethernet technology and the interaction with the TCP protocol. The *maximum transmission unit* (MTU) of the cluster's Ethernet interfaces on the cluster is 1500 bytes. Thus, larger messages are fragmented into packets of that size and are transmitted with store-and-forward switching. Smaller messages are fragmented in just a few packets and cannot utilize all components through the network route. Therefore, the time of such messages depends on the throughput of all intermediate components. Consequently, the model predicts a higher throughput than measurable.

For larger messages, this effect can be ignored. Transfer of very small messages is dominated by latency and, therefore, this effect plays a minor role for messages smaller than 1024 bytes.

c) The time of the bi-directional kernel is also matched quite well for very small and very large messages (Figure 7.5d). Message sizes between 1024 bytes and 128 KiB can be received much faster than approximated (about 16 times, the figure is clipped). This is due to an early-receive of the message – the process with Rank 1 might start data transfer before Rank 0 and thus it completes transfer of the data to Rank 0 before this rank starts. Message sizes handled by the eager-protocol (slightly below 128 KiB of data) are influenced by this behavior.

d) In local communication the approximation for the PingPong kernel underestimates performance by up to 20% for messages below 32 KiB (see Figure 7.7d and Figure 7.6d).

e) Qualitatively, the intra-node estimates are on a comparable level – the relative accuracy is appropriate for many cases (see Figure 7.6 and Figure 7.7). However, there are regions in which predicted throughput is just half of the measured throughput.

f) In the range between 64 KiB and 8 MiB the approximation of inter-socket communication is 40-100% slower than the real system. This is probably caused by the cache hierarchy: the L1, L2 and L3 cache have a capacity of 32 KiB, 256 KIB and 12 MiB, respectively. In the experiment data is available in the cache, therefore, the cache helps. But in general, messages can contain data which is currently not in a CPU cache. Caches are not covered by the communication model and thus this kind of data error is inherent to the model.

g) By using the higher bi-directional characteristics, a better approximation is achieved for large messages – above 8 MiB (compare Figure 7.6a with Figure 7.6c). But with the slower uni-directional throughput, results match better for smaller messages.

The throughput has been obtained with the modified `mpi-network-behavior` benchmark which transfers 100 times 1 GiB of data (see Section 3.5.2). Somehow the benchmark underestimates the memory throughput for the smaller accesses of the test cases.

Concluding this section, the model can be applied to approximate performance for inter-node, inter-socket and intra-socket communication on our test system. Cached data and architectural details lead to a better communication performance in intra-node communication than the model predicts.

However, building a model which addresses the cache hierarchy requires simulating the processor microarchitecture in detail: An exact bookkeeping where data is currently cached, and the processor communication to ensure cache consistency between cores (for example performed by the *MESI* protocol). It does not suffice to expect that data is cached when it is small enough, since the user can start a point-to-point communication any time. Simulation on that level of detail is out of the scope for a cluster-wide simulator.

Scientific applications are expected to communicate larger amounts of data which do not fit into the processor caches. Thus, the necessity of such a detailed cache model for this project is questionable.

7.4.2. Hard Disk Drive

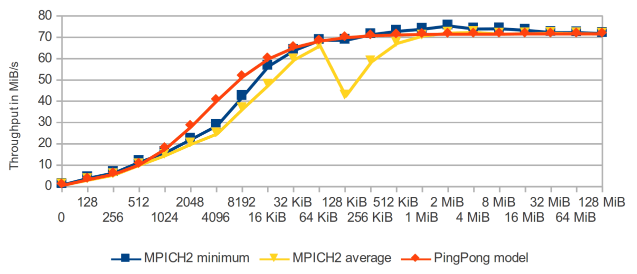
To validate the general approach of the *RefinedDiskModel*[15], estimated throughput is compared with measurements of Section 3.6.2 for a variable record size. The model matches performance of observed sequential access perfectly (96 MiB/s) because seeks are ignored in the model and it is parameterized with the maximum throughput. Random access in contrast, must be assessed in detail to show the appropriateness of the model.

First, the behavior of the basic model which estimates performance is briefly recapitulated and analyzed: The RefinedDiskModel depends on the last accessed offset, therefore, an exact computation is only possible in the simulator. However, a qualitative analysis of the underlying analytical model for random access can be performed by computing potential throughput with Equation 7.6.

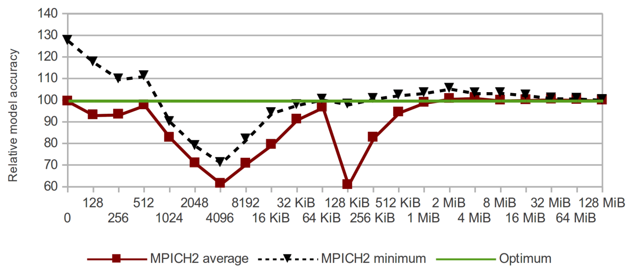
$$\text{rotLat} = \frac{1}{2} \cdot \text{time for one rotation} = \frac{1}{2} \cdot \frac{60s}{\text{revolutions per minute}} = \frac{30s}{RPM} \tag{7.5}$$

$$\text{throughput} = \frac{\text{record size}}{\text{latency} + \text{record size}/\mathit{sequentialTransferRate}} \tag{7.6}$$

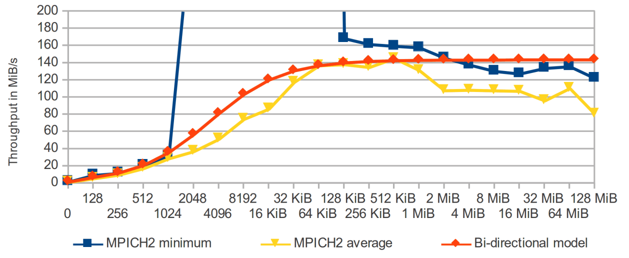
[15]Refer to Section 5.2.2 for an introduction of this model.



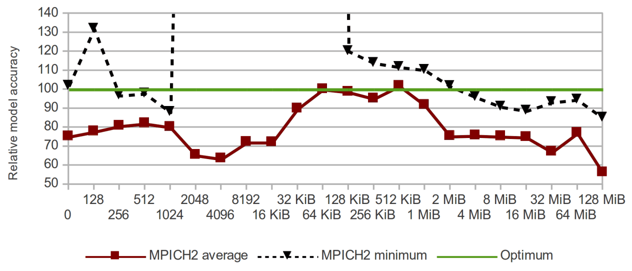
(a) PingPong kernel throughput.



(b) PingPong kernel – relative accuracy.



(c) Sendrecv kernel throughput.



(d) Sendrecv kernel – relative accuracy.

Figure 7.5.: Inter-node communication performance – comparison of determined `mpi-network-behavior` performance of one run with the network model.

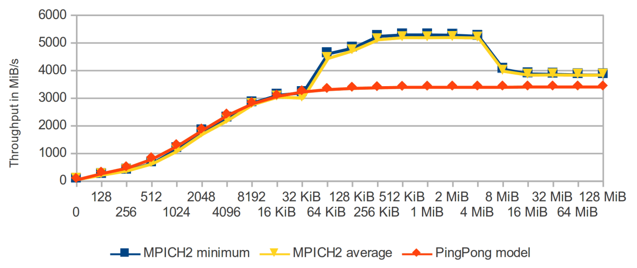
In this model, throughput and latency can be modified or omitted. To show a corridor of potential throughput, several latency values are compared: the rotational latency (given by the spinning speed), track-to-track seek time and average seek time. As noted above, the values of the disk model are: an RPM of 7200 ($rotLat = 4.166$ ms), track-to-track seek time of 1.1 ms, average seek time of 9 ms and a maximal sequential transfer rate of 96 MiB/s. The expected throughput with this model is visualized in Figure 7.8. Computed values of the analytical modes are listed in Table 7.7. The rotational latency is included in all models, while the throughput limitation (maximal sequential transfer) is included in a few (it is abbreviated with TP). To model data aggregation, the *Aggregate2* model assumes two records can be aggregated – that is, two random 16 KiB records actually manifest as one 32 KiB record. For cached write calls, such kind of aggregation could actually be done by the OS or the disk's NCQ algorithm.

Most of the conducted measurements do not directly reveal the real characteristics of an HDD because all I/O is passed through the file system and operating system. Therefore, measurements are chosen which are actually related to the disk's behavior. Two tables list a selection of throughputs measured by IOZone (see Table 7.8) and `posix-io-timing` (Table 7.9). As the model is not valid for cached I/O, data points which hit disk are taken – these are mostly results obtained with 1 GiB available memory, and the experiment with remount. Several file sizes are taken and listed as a configuration. Direct I/O is compared for `posix-io-timing` for synchronous re-write and the random read performance. The random re-write test is picked because data blocks are already allocated by the file system and thus little update of file system structure is necessary.

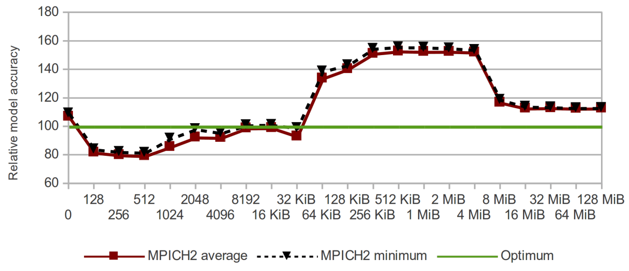
A diagram containing model performance and measured performance is given in Figure 7.9. With this graph, a qualitative first analysis can be done.

Observations and interpretation

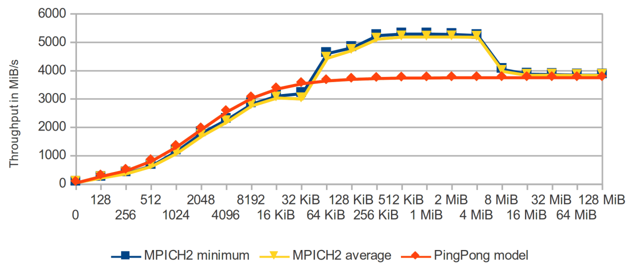
a) Seek time is the dominating factor for small record sizes (below 128 KiB), which is intuitive. See Figure 7.8a and compare the performance of the models without limited sequential transfer rate with those with limitation. Keep in mind that larger records could be fragmented on disk, which causes



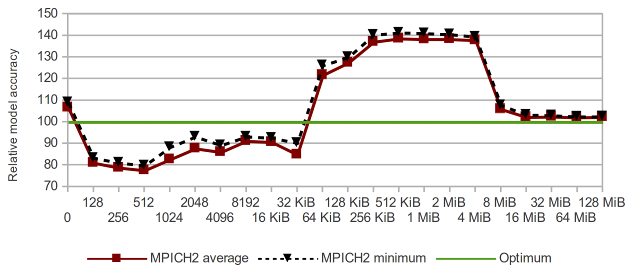
(a) PingPong kernel throughput.



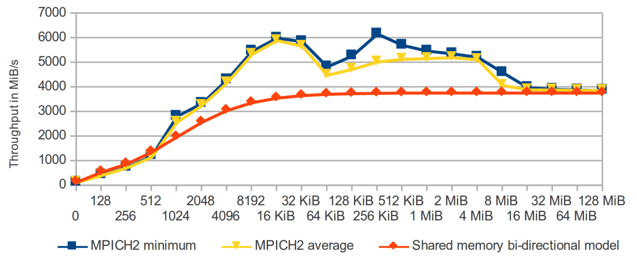
(b) PingPong kernel – relative accuracy.



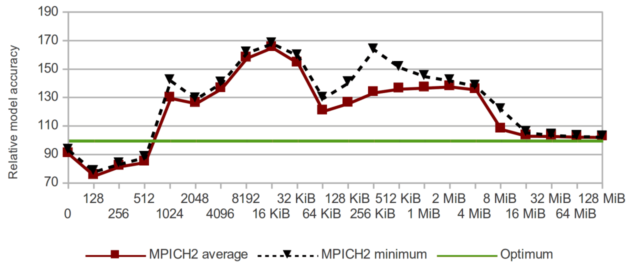
(c) PingPong kernel throughput with the higher bi-directional throughput.



(d) PingPong kernel with bi-directional characteristics – relative accuracy.

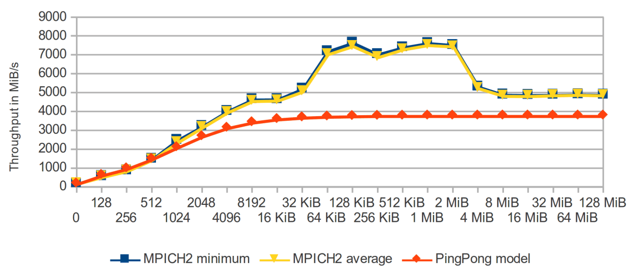


(e) Sendrecv kernel throughput.

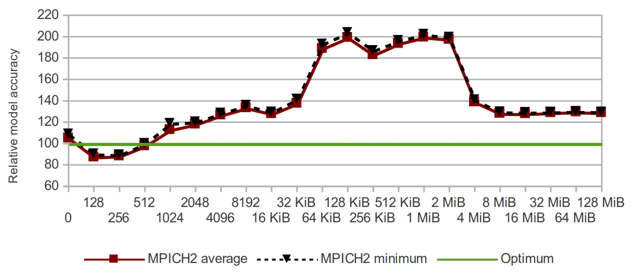


(f) Sendrecv kernel – relative accuracy.

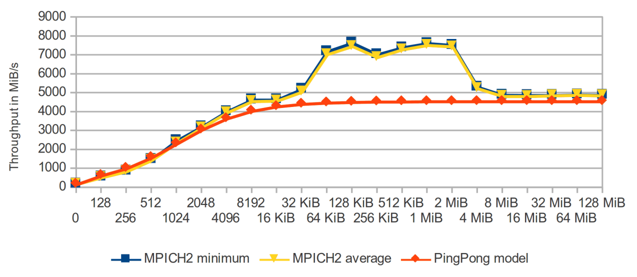
Figure 7.6.: Inter-socket communication performance – comparison of determined `mpi-network-behavior` performance of one run with the network model. The PingPong kernel is assessed by using two different characteristics for throughput.



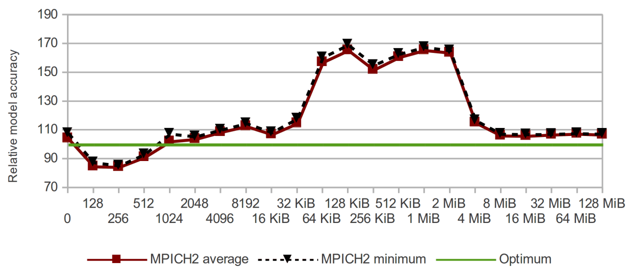
(a) PingPong kernel throughput.



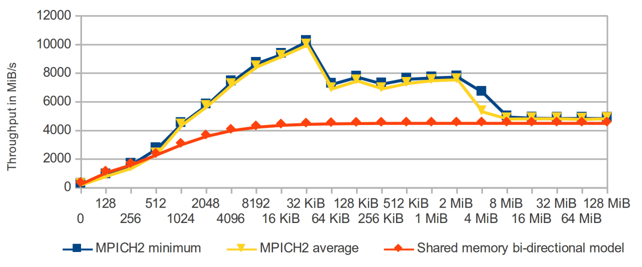
(b) PingPong kernel – relative accuracy.



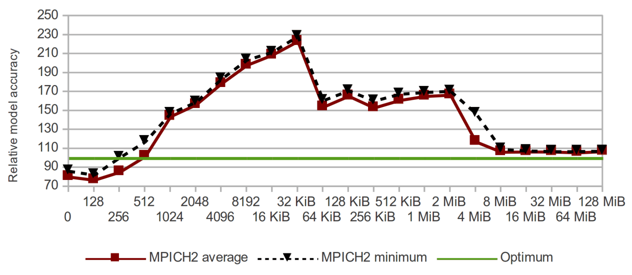
(c) PingPong kernel throughput with bi-directional characteristics.



(d) PingPong kernel with bi-directional characteristics – relative accuracy.

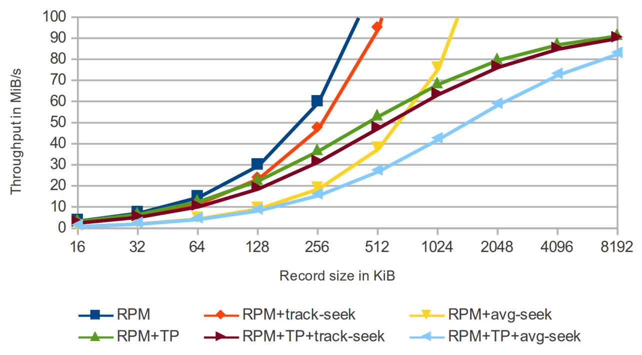


(e) Sendrecv kernel throughput.

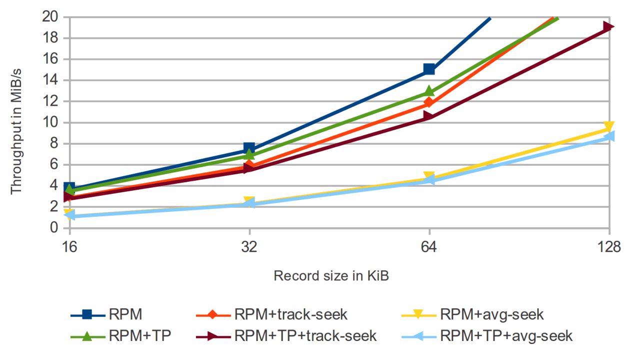


(f) Sendrecv kernel – relative accuracy.

Figure 7.7.: Intra-socket communication performance – comparison of determined `mpi-network-behavior` performance of one run with the network model. The PingPong kernel is assessed by using two different characteristics for throughput.



(a) All record sizes.



(b) Small records.

Figure 7.8.: Behavior of the disk model with different characteristics. The models incorporate revolutions-per-minute (RPM), throughput (TP), track-to-track and average seek time.

Configuration	Record size in KiB									
	16	32	64	128	256	512	1024	2048	4096	8192
RPM	3.75	7.5	15	30	60	120	240	480	960	1920
RPM+track-seek	2.97	5.93	11.87	23.73	47.47	94.94	189.87	379.75	759.49	1518.99
RPM+avg-seek	1.19	2.37	4.75	9.49	18.99	37.97	75.95	151.90	303.80	607.59
RPM+TP	3.61	6.96	12.97	22.86	36.92	53.33	68.57	80.00	87.27	91.43
RPM+TP+track-seek	2.88	5.59	10.56	19.03	31.76	47.73	63.76	76.63	85.23	90.29
RPM+TP+avg-seek	1.17	2.32	4.52	8.64	15.85	27.21	42.40	58.82	72.95	82.90
RPM+TP+Aggegate2	6.96	12.97	22.86	36.92	53.33	68.57	80.00	87.27	91.43	93.66

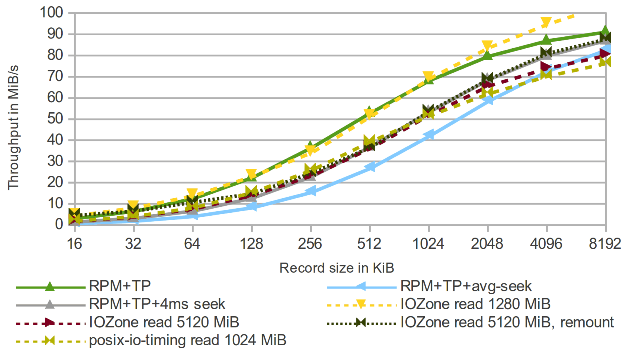
Table 7.7.: I/O throughput computed with the analytical model (in MiB/s).

Configuration	Record size in KiB									
	16	32	64	128	256	512	1024	2048	4096	8192
Read 1280 MiB	4.86	8.56	14.24	23.40	34.51	51.60	69.46	83.96	95.04	104.26
Read 2560 MiB	2.74	5.00	6.52	16.33	26.39	40.62	57.33	74.41	85.35	91.25
Read 5120 MiB	2.27	4.34	8.06	14.48	23.65	36.61	52.31	65.96	74.31	80.35
Read 10 MiB, remount	8.98	14.17	18.23	26.67	32.99	47.58	58.79	74.88	85.58	83.67
Read 1280 MiB, remount	5.61	8.84	12.64	18.20	28.27	40.60	57.49	71.71	82.49	89.43
Read 2560 MiB, remount	5.24	8.20	11.93	17.30	27.00	39.46	56.03	71.23	82.05	88.66
Read 5120 MiB, remount	5.00	7.05	11.22	15.31	24.97	37.11	54.14	69.13	81.30	88.17
Write 2560 MiB	6.80	12.77	21.08	36.28	53.12	65.90	78.73	85.80	90.29	91.22
Write 5120 MiB	5.98	11.52	19.88	31.79	46.47	58.30	66.75	74.63	72.66	78.82

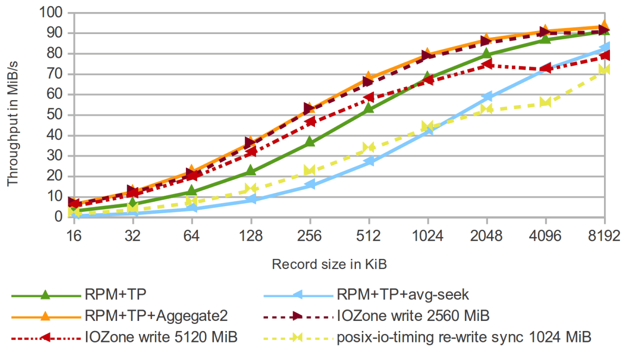
Table 7.8.: Average random I/O throughput measured by IOZone (with 1 GiB of available memory; in MiB/s).

Access pattern	Record size in KiB									
	16	32	64	128	256	512	1024	2048	4096	8192
Sequential write	95.6	95.3	95.4	95.0	81.2	75.0	75.6	74.9	74.9	74.9
Sequential read	96.7	95.5	96.6	96.8	96.7	76.4	76.3	76.2	76.2	76.4
Random write	5.3	10.4	18.6	29.1	38.3	51.3	54.9	56.1	60.3	68.9
Random write sync	0.6	1.3	2.5	4.8	8.8	15.4	24.4	33.9	40.7	55.0
Random re-write sync	2.1	4.1	7.6	13.7	22.7	33.8	44.3	52.7	56.0	72.0
Random read	2.3	4.6	8.5	15.8	25.9	39.7	52.0	61.0	67.7	73.7

Table 7.9.: Average throughput in MiB/s of direct I/O by using *O_DIRECT* (and *O_SYNC*) to access 1 GiB of data – measured using `posix-io-timing` with a variable access granularity.



(a) Random read throughput.



(b) Random write throughput.

Figure 7.9.: Comparison of the throughput of the analytical I/O models with the average random performance obtained with IOZone and `posix-io-timing`.

additional seeks. These are not covered by the model because data layout on disk is not tracked.

b) Measured read throughput lies between the RPM+TP characteristics and results which include average seek time (see Figure 7.9a). The actual values for the observations with the large file are just in between these two models.

c) Actually, the average seek time is too pessimistic for the experiment, because the file accessed is just roughly 5 GiB of size. Under the assumption that file bytes are stored close together on disk, the access arm does not have to move the average distance between two random positions on the 250 GByte disk – which would be covered by the average seek time. An average seek time of 4 ms covers the behavior of the HDD for the 5 GiB file much better (see the gray line in the center of the figure).

d) In the experiment with the 1280 MiB file parts of the file are still available in the page cache from previous execution of the write kernel. Therefore, the measurements outperforms the measured sequential transfer rate for the largest block size (see Figure 7.9a). Interestingly, the measurement matches many data points with the estimated performance of RPM+TP characteristics.

e) Measured performance of the file with a size of 10 MiB for small block sizes is higher than any of the models (refer to Table 7.8). This is probably due to the behavior of the Seagate HDD, which also offers an 8 MiB cache and uses read-ahead to fetch records.

f) Interestingly, the write experiment of the 2560 MiB file behaves like the RPM+TP+Aggregate2 model (see Figure 7.9a). Theoretically, it is possible that on average two operations in the OS cache are fused into a single operation of twice the size. To illustrate the high probability for merging multiple blocks, consider the file with the size of 2560 MiB and 1 GiB of main memory. Now almost every other block can be kept in memory before data must be written out. Every new block that should be written can be merged with at least one other block[16] For small record sizes this assumption might also be true for the 5120 MiB file; however, for larger record sizes performance drops.

Adjusting I/O characteristics to match observed behavior While it has been shown that the behavior of the 5 GiB file could be matched for larger record sizes by choosing a latency of 8.16 ms (rotational latency + 4 ms), the model still does not match all observations. One explanation is the variation in the file size which influences seek time – blocks of smaller files are located close together on disk and thus seek time is lower. Also, write performance of the disk drive is slightly lower than read performance.

[16]While this result suggests that aggregation is actually performed, this has not been verified. A detailed monitoring and analysis of block I/O is possible, but out of scope for this thesis.

However, since the measured behavior is complex, the question arises if the general I/O model is appropriate to explain observable throughput. Potentially, the RefinedDiskModel might not capture the behavior well, for instance the file system could scatter data blocks across the block device leading to additional latency while accessing these blocks.

To better understand the model, a small parameter study is conducted: the total latency (including rotational latency) and sequential transfer rate are adjusted to predict each of the presented throughput curves. By modifying latency, performance estimation of small records can be adjusted and by changing throughput, large record sizes. The relative model accuracy of the differently parameterized model is visualized in Figure 7.10 by plotting the quotient of model throughput and measurement – that means a value above 100% shows that the model is faster than the observation. As a reference, the throughput of the model which takes sequential transfer rate and RPM into account, is given in Figure 7.10a.

Observations and interpretation

a) RPM+TP predicts better performance than measured by `posix-io-timing`; the model achieves a stable 120% and 150% of measured throughput for reads and writes, respectively (see Figure 7.10a). A higher predicted performance is expected because both operations require synchronization with the HDD and additional seeks. However, the relative performance is stable across record sizes which is surprising. IOZone write performance is much better because the OS can defer and aggregate operations which improves throughput.

b) Model accuracy of the reference model RPM+TP+4 ms is between 30 and 150% (see Figure 7.10b). The additional latency leads to better matching for larger read operations. However, the random read performance of IOZone cannot be fitted with the two parameters of the model. One reason for the discrepancy is that the kernel offers read-ahead to fetch more data than required. Read-ahead is enabled for larger record sizes and the additional time for operation can be considered as additional latency to regular disk reads.

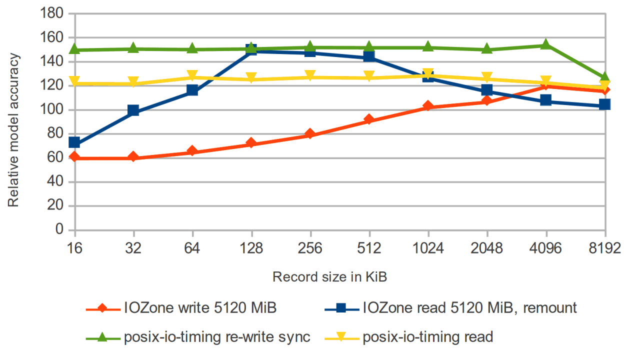
c) By adjusting the sequential transfer rate and latency, three of the curves could be fitted well (see Figure 7.10e, Figure 7.10c and Figure 7.10d). However, optimal sequential transfer rate and latency vary.

A consequence of these observations is, that the RefinedDiskModel can be applied to estimate random I/O throughput for disk accesses, if it is parameterized appropriately. Choosing appropriate parameters depends on access type and file size. The source of latency, may it be rotational latency, overhead of read-ahead, additional latency of the OS scheduler, or just seek time can be fit into the model. For instance, direct I/O is slower than cached I/O, because the operating system and the disk cannot optimize the access patterns in order to hide additional seeks to access and manipulate file system structures. The actual observed behavior is hard to understand and to predict due to many involved factors.

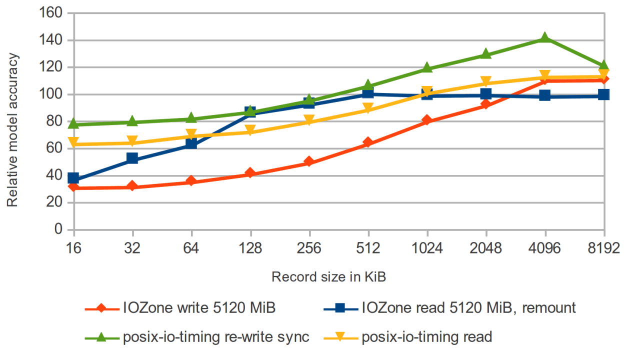
While read and write operations could easily be distinguished in the implementation and thus handled differently by adding additional characteristics to the RefinedDiskModel, the current model is kept. The model follows the basic behavior of hard disk drives and can be used to assess common optimization strategies. For example, in order to simulate write-behind and optimization by OS and HDD, several caching algorithms have been implemented in PIOsimHD. Their performance impact is assessed in the simulation experiments.

7.4.3. Summary and Conclusions

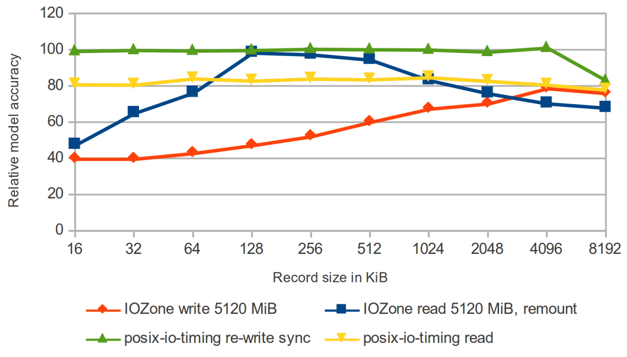
In the qualification for network and hard disk drive, the estimated performance for a variable message size and record size is compared to measured performance. The network model that is based on the characteristics of latency and throughput matches performance in many cases quite well (relative accuracy ±10%). Prediction of the bi-directional kernel is worse, because the observations fluctuate much more and measured performance is lower for large messages. In intra-node communication, the CPU cache improves



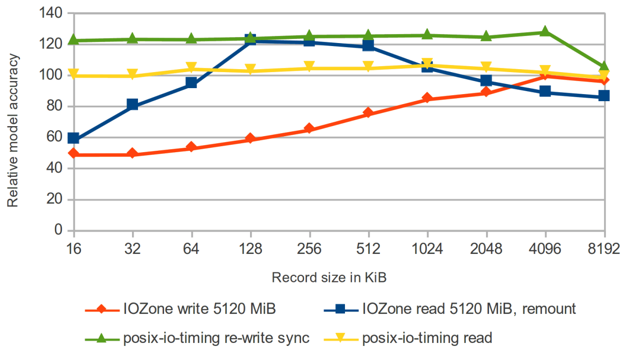
(a) RPM+TP model – sequential transfer rate: 96 MiB/s, latency: 4.166 ms.



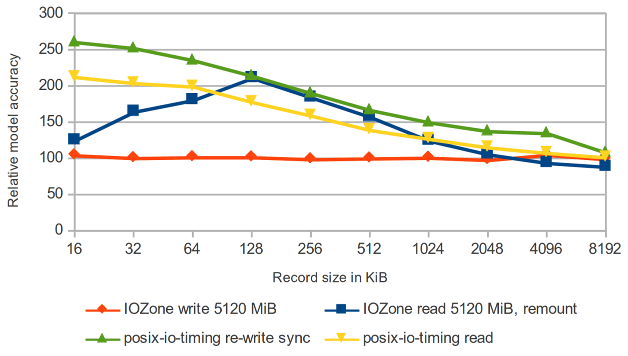
(b) RPM+TP+4 ms. Fitting the curve of IOZone read – sequential transfer rate: 96 MiB/s, latency: 8.166 ms.



(c) Fitting the curve of `posix-io-timing` re-write – sequential transfer rate: 63 MiB/s, latency: 6.3 ms.



(d) Fitting the curve of `posix-io-timing` read – sequential transfer rate: 80 MiB/s, latency: 5.1 ms.



(e) Fitting the curve of IOZone write – sequential transfer rate: 80 MiB/s, latency: 2.3 ms.

Figure 7.10.: Relative performance of several parameterizations of disk latency and transfer rate.

throughput significantly in the measurements; especially message sizes that fit into the L3 cache are about 50% faster than predicted. Since CPU cache is not modeled, the simulation is biased for these message sizes by the same extent. However, while data of the benchmark is available in the CPU cache, a real application might also transfer data that is stored in main memory. In this case, performance will be lower and expected to be more accurately approximated by the model. Without modeling the CPU cache, it is impossible to determine the locality of data; also, such a model is complex and interwoven with the model for computational activity because that involves memory activity as well.

The sequential performance of the RefinedDiskModel model matches observations perfectly, because the value is chosen according to the measurements. The qualification of the model also assesses random access performance: Qualitatively, most measurements are described by a model based on the characteristics of throughput and latency – basically, the measurements lie between the model that incorporates average seek time and the one that relies on the rotational latency. However, since the determined random performance depends on the chosen benchmark and its configuration, an accurate match with all results is impossible. A parameter study for latency and throughput shows that all benchmark results can almost perfectly be matched by picking the right characteristics – but these characteristics depend on the benchmark, the access type and the configuration. Therefore, latency and throughput suffice to describe random access on an HDD. However, these characteristics depend on many optimizations and hardware features provided by the real system: the physical block layout of the file accessed, the cache behavior and the efficiency of aggregating smaller requests into bigger ones, for example. Basically, by improving the performance for certain use cases, these optimizations make the system behave as if its characteristics are different.

7.5. Verification of Network Behavior and Hard Disk Model

In this section, network and I/O performance of simple test cases is computed with the simulator and compared with the analytical models for our cluster, which serves as an experimental environment. During this process, important and tunable simulation specific parameters are discussed.

7.5.1. Network Behavior

In contrast to the analytical model, the simulator has an implementation specific handling of messages. Messages are fragmented into packets which are transferred via store-and-forward switching between the components of the network topology (see Section 5.2.3). A larger network granularity reduces the concurrent processing on the simulated devices and bigger messages may defer short messages due to FCFS scheduling. However, an advantage of increasing the network granularity is that it makes simulation faster.

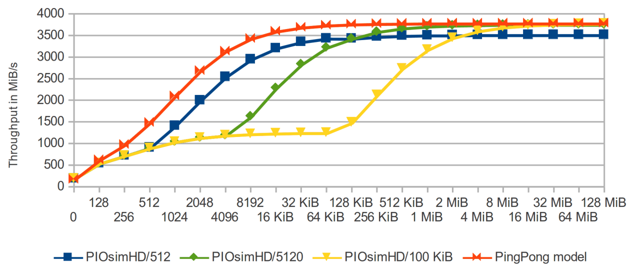
Overhead to represent communication protocol and message headers (currently 40 bytes) can be added either per message, or per packet. The later is influenced by the fragmentation rate and adds much more overhead for smaller network granularities. The *network granularity* parameter and protocol header overhead are assessed in the following.

The figures in Figure 7.11, Figure 7.12 and Figure 7.13 compare network granularities of 512 bytes, 5 KiB and 100 KiB with the reference provided by the analytical model – computed throughput and relative accuracy[17] of the PingPong and SendRecv kernel is provided. The first diagram shows performance with protocol overhead per packet, the other two account protocol overhead once per message.

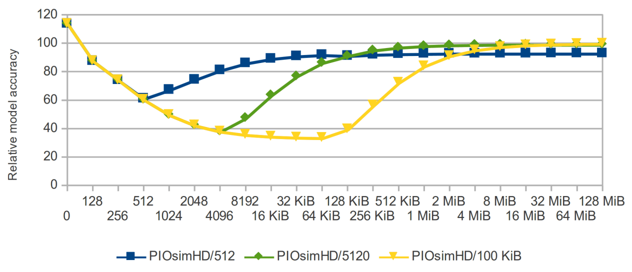
Observations and interpretation

a) The performance of intra-socket communication with 40 bytes overhead per packet reduces performance for large messages when a packet size of 512 bytes is used. Throughput of a smaller network

[17]Relative accuracy is defined as simulation throughput divided by the analytical model's throughput.



(a) Throughput.



(b) Relative performance.

Figure 7.11.: Intra-socket communication for the PingPong kernel with protocol overhead per packet.

granularity might be below the throughput of larger message sizes (see Figure 7.11b). This is expected because packet header is about 7.8% of a 512 byte packet. However, for most users performance degradation due to smaller network granularity is not intuitive. Therefore, the simulator adds overhead once per message by default to cover the message's header.

b) The relative performance of analytical model and simulation matches well for a network granularity of 512 bytes, at least 65 % accuracy for messages of that size in intra-node communication and more than 90% in inter-node communication (see Figure 7.12).

For messages smaller than network granularity, achieved performance is lower than predicted by the analytical model. The reason is the store-and-forward switching: the whole message must be received by all components on the path between sender and receiver; in the cluster model up to 13 components are involved in inter-node communication, every edge and node of the topology forwards the packet. This takes more time than anticipated by an analytical model, because the analytical model assumes optimal concurrent processing in intermediate nodes. With increasing message size (or decreasing network granularity), data can be processed concurrently on intermediate nodes, improving overall performance.

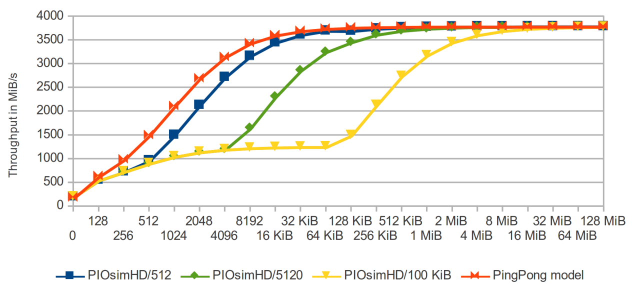
c) On our cluster system store-and-forward switching is performed in inter-node communication, the MTU is 1500 bytes. It has been observed in Section 7.4.1, that the analytical model overestimates performance slightly for message sizes between 1 KiB and 16 KiB. The difference between analytical model and the store-and-forward simulation can be assessed in Figure 7.12f; for transferring a packet of 512 bytes the discrepancy is 10% and decreases with the increase of the payload.

d) Starting with messages 32 times as large as the network granularity, almost optimal performance is achieved for local communication (refer to Figure 7.12b). For example, with fragments of 100 KiB, a good performance of 3500 MiB/s is estimated for 4 MiB messages. The ratio is slightly lower for inter-node communication, although the communication path is longer (see Figure 7.12f).

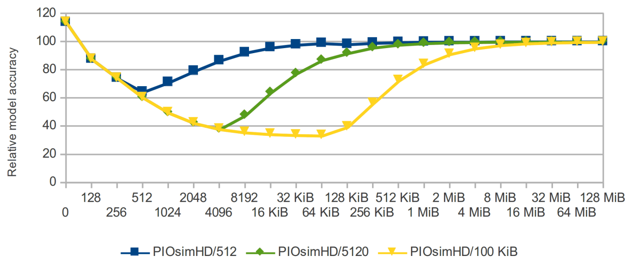
e) A small performance dip appears for messages with a payload of 128 KiB and 256 KiB (look at Figure 7.13d). This is caused by the rendezvous protocol which is supported by the simulator, but not incorporated into the simple analytical model. Since performance degradation is very low, the default eager-size seems appropriate for our testbed.

f) The simulation of bi-directional communication behaves similar to uni-directional communication (compare Figure 7.13 to Figure 7.12).

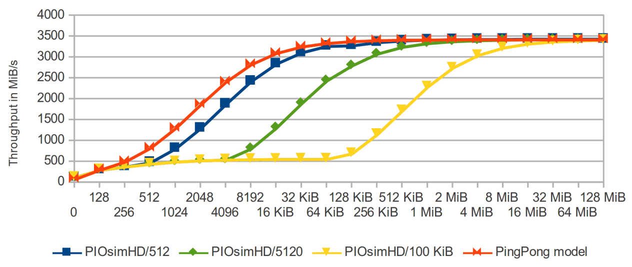
The conducted verification demonstrates that the simulation model approximates the point-to-point communication of the analytical model well. The analytical model, in turn, is close to measured performance of our testbed, in many cases. Since a network granularity of 100 KiB suffices to achieve a high throughput (for messages larger than 4 MiB), this granularity is used for most further testing.



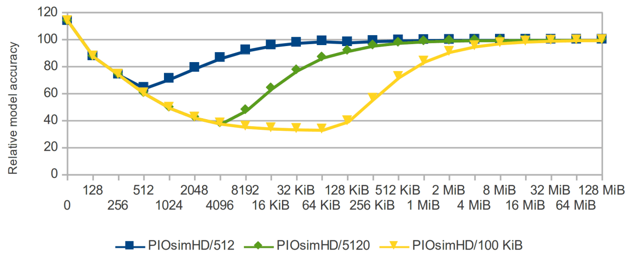
(a) Intra-socket throughput.



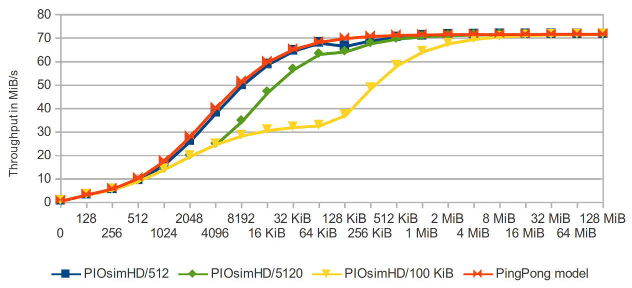
(b) Intra-socket communication – relative performance.



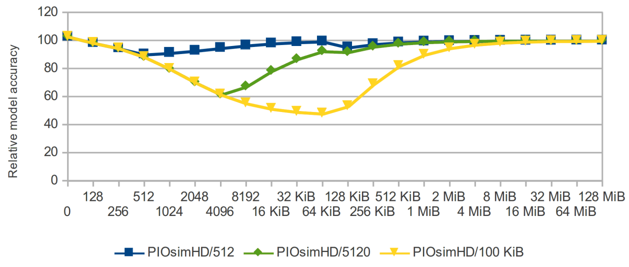
(c) Inter-socket throughput.



(d) Inter-socket communication – relative performance.

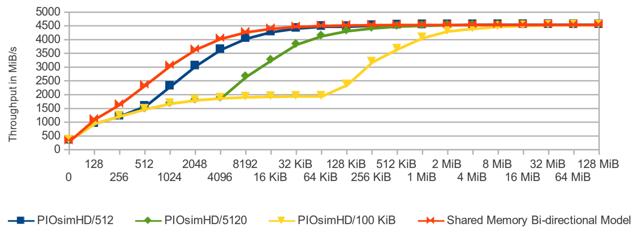


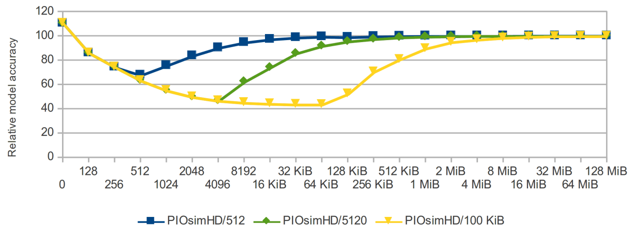
(e) Inter-node throughput.

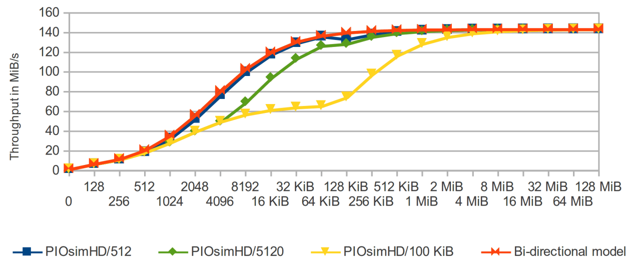


(f) Inter-node communication – relative performance.

Figure 7.12.: Comparison of the network model with simulated performance of the PingPong kernel.


(a) Intra-socket throughput.


(b) Intra-socket communication – relative performance.


(c) Inter-node throughput.

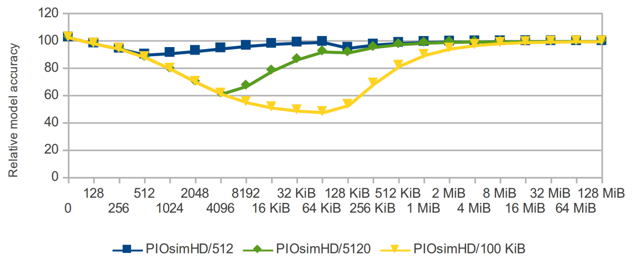

(d) Inter-node communication – relative performance.

Figure 7.13.: Comparison of the network model with simulated output for the Sendrecv kernel.

7.5.2. I/O Subsystem

To verify the simulation of a single I/O subsystem, performance of the NoCache, WriteBehind and the AggregationReorderCache implementation is compared to the analytical model. A system model is built that consists of one client, one server and a very fast interconnect (zero latency, throughput of 1000 GiB/s). Network granularity is set to 100 KiB and I/O granularity is set to 10 MiB. Either a sequential or a random pattern is created by the client, accessing records of a fixed size until 1280 MiB of data have been touched.

Results for random I/O are given in Figure 7.14 – other results are quite similar and obvious and, therefore, these figures have been left out. As noted in Section 5.4.5, caching of read operations is currently not modeled, therefore, it cannot be assessed.

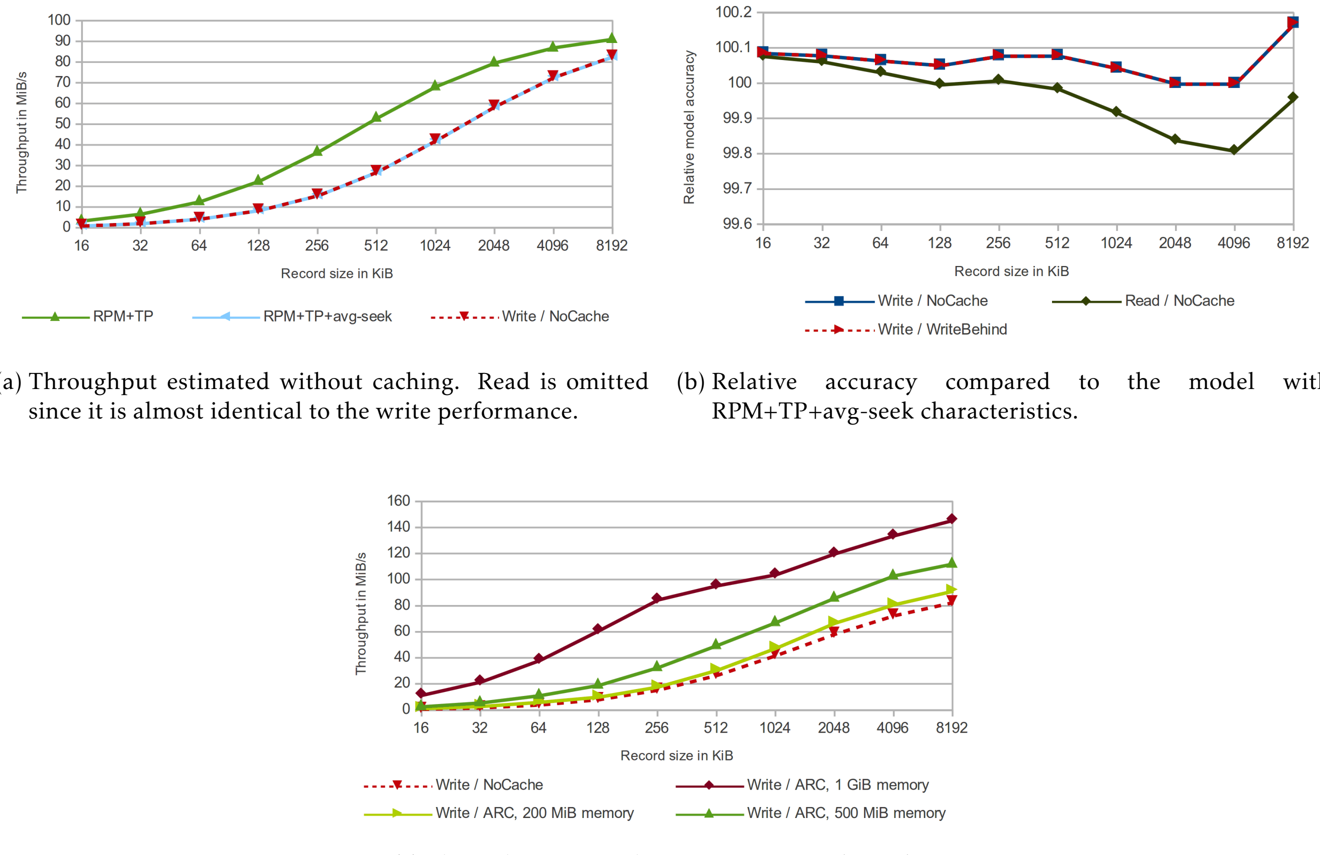
Observations and interpretation

a) Sequential performance is 95.9 MiB/s and above 95.2 MiB/s for writes and reads, respectively. Note that the performance is independent of the record size, it only depends on the characteristics of the block device, therefore, the figures are omitted. Write is a bit faster because of the pipelining scheme: Whenever a data block is received, it is put into the cache and thus, at the beginning of a larger request, data is written in chunks of the network granularity (100 KiB). Therefore, network and disk can operate concurrently.

A single read request can fetch up to 10 MiB of data from the disk. Once data is read, it is transferred to the client. Therefore, the network and disk activity happen sequentially, although the network is very fast, this reduces performance slightly.

b) The simulation of write operations match the analytical model quite well for all record sizes (see Figure 7.14a). Read operations achieve similar performance, therefore, they are omitted in the figure. The relative accuracy of the simulation compared to the analytical model is shown in Figure 7.14b; implementation in the simulator and theoretical model match well. A minimal drop in read performance to 99.8% for larger record sizes is caused by the pipelining scheme.

Actually, a single I/O request is transported from the client to the server of the modeled file system.



(a) Throughput estimated without caching. Read is omitted since it is almost identical to the write performance.

(b) Relative accuracy compared to the model with RPM+TP+avg-seek characteristics.

(c) Throughput using the *AggregationReorderCache*.

Figure 7.14.: Random I/O performance – comparing simulation with the analytical model.

At this point the behavior of the cache layers could differ, because the write path can be pipelined and the cache layer schedules operations. Hence, performance of read and write operations could be different. In this experiment there is no noticeable effect because the interconnect is very fast. However, when performance of the simulated parallel file system is assessed, this effect will become visible.

c) When using the *AggregationCache* or the *AggregationReorderCache* (ARC), write operations can be deferred and aggregated, which is expected to increase performance. The impact of using 1 GiB, 500 MiB or 200 MiB of memory to cache and optimize operations is shown in Figure 7.14c. With an increase in cache size performance of random writes increases as well.

d) When more cache is available, performance of the AggregationReorderCache improves to a larger extend. Thus, the improvement is not proportional to available memory size. For example, for 16 KiB records simulated performance is 1.2 MiB/s, 1.7 MiB/s, 3.1 MiB/s and 12 MiB/s for NoCache, and ARC with 200 MiB, 500 MiB and 1 GiB of main memory, respectively.

On average roughly 50% of the blocks are re-written during the run. Therefore, a performance higher than the transfer rate of 96 MiB/s is possible because blocks can be overwritten in the cache before actual I/O is performed. Also, consecutive blocks can be aggregated to improve performance. The memory with a capacity of 1 GiB can hold all data; when a small cache is used, it is unlikely that random blocks are neighbors, therefore, they cannot be aggregated into a larger request. The bigger the cache the more likely a cached block is overwritten before it is stored on the virtual block device. This behavior, in combination with the characteristics of the HDD, leads to the observed behavior.

e) Overall, observed behavior of the cache layers matches the analytical model. A comprehensible, yet complex caching behavior is revealed by the AggregationReorderCache.

7.5.3. Summary and Conclusions

During the verification of the network behavior, the network granularity is varied and the communication overhead of transferring a header per packet is compared to one header per message. As the mathematical model does not fragment messages, it outperforms the simulation results. A small network granularity increases the concurrent transfers within the network, because intermediate network components can already start to transfer packets while the sender is still emitting packets of the message. Up to 13 network components are involved in inter-process communication. In the worst case, only one packet is transferred – in this case, the simulation achieves only 40% of theoretical performance. Achieving about 90% of theoretical performance requires that the network granularity suffices to allow concurrent processing on each component – the message size must be larger than the network granularity by at least an order of magnitude. Model performance is achieved perfectly when the message size is more than two orders of magnitude larger than the network granularity. Fragmentation is realistic for inter-node communication; the Ethernet protocol, for example, fragments data into frames with a size of at most 1,500 byte (the MTU on our system). On a real system, fragmentation does not apply to intra-node communication, therefore, the simulator systematically underestimates performance for small message sizes.

The overhead per packet reduces potential throughput for larger messages as more data must be transferred. This leads to an improved performance of larger network granularities, which would be counterintuitive for users; therefore, overhead is accounted only once per message in subsequent experiments.

To verify the performance of the I/O subsystem, a two node cluster setup is used, one node acts as a client that performs random or sequential accesses, and one acts as a server. Client and server are connected with a very fast network without latency. Without write cache, the simulation reproduces the performance of the mathematical model perfectly (accuracy of 99.8%). Additionally, the benefit of the AggregationReorderCache is evaluated for the random workload with three different cache sizes. As expected, it shows that the benefit increases with the cache size, but in a non-linear fashion.

7.6. Evaluation of the Network Model with Complex Communication Patterns

With the comparison of simulation and observation for more sophisticated communication patterns, the cluster model is validated. Additionally, by comparing performance of MPICH2 with the simulated estimate existing discrepancy is analyzed and discussed. This allows identification of suboptimal communication behavior.

While basic point-to-point communication between a single pair of processes has already been assessed in Section 7.5, several point-to-point communication patterns involving multiple processes are discussed. Therefore, a new benchmark has been written – `mpi-bench`, which measures times for several patterns and message sizes – this benchmark is introduced in Section 7.6.1. Since all collective calls are implemented with basic point-to-point communication, those experiments provide additional insight into simulation accuracy.

The methodology of the validation are discussed in Section 7.6.2. Results of the validation is discussed in subsequent sections of this chapter: `MPI_Barrier()` is used for time results in the benchmark, therefore, it is assessed first in Section 7.6.3. Additionally, the impact of computation time on measurement and simulation is sketched. Point-to-point communication of 10 KiB messages evaluated in Section 7.6.4 and collective communication in Section 7.6.5. Larger message payloads are assessed in Section 7.6.6 and in Section 7.6.7 for point-to-point communication and collective calls, respectively.

7.6.1. The mpi-bench Benchmark

The mpi-bench benchmark measures the following MPI collective calls: MPI_Barrier(), MPI_Reduce(), MPI_Allreduce(), MPI_Bcast(), MPI_Gather(), MPI_Allgather() and MPI_Scatter(). Further, the following point-to-point communication schemes are supported:

- *Ring* – every process sends data to its right neighbor and receives from the left neighbor.
- *Paired* – every even process exchanges data with the next odd process by using MPI_Sendrecv().
- *SendToRoot* – all processes send data to the root process, which receives data in order.
- *SendrecvRoot* – processes use MPI_Sendrecv() to exchange data with root.

A few point-to-point patterns require an even number of processes, invalid configurations are excluded from the measurement.

The benchmark iterates over all collective and point-to-point communication kernels and message sizes. Kernels are repeated a number of times (the trivial pseudo code is given in Listing 7.2). Evaluated message sizes are 10 KiB[18], 1 MiB, 10 MiB, 100 MiB and 1000 MiB. Note that the overall amount of communicated data depends on the collective operation. The payload is the amount of data for the invocation of the MPI function (e.g., Scatter() transfers this amount of data to every process). A few collective calls require additional buffer space and could not be benchmarked for some configurations, because memory of the nodes does not suffice.

Listing 7.2: Pseudo code of mpi-bench – main loop

```
for (i = 0; i < options.count ; i++):
  for each kernel:
    execute kernel

if rank == 0:
  print time
```

Time for executing a kernel is measured with MPI_Wtime() on all processes. Timing is protected by a barrier to ensure that initialization and cleanup of kernel relevant data structures are not measured. The benchmark supports aggregating minimum and maximum times across all processes and runs. This is interesting because the amount of work done by a collective call can vary with the rank of the process; it is nice to see the variability between slowest and fastest process.

The time of the call including the final barrier is output for Rank 0, too. It is a good estimate for the execution time of the whole kernel on all processes and determined by the slowest process. Listing 7.3 shows the pseudo code for kernel execution and time. The MPI_Barrier() function is an exception to this rule, it does not require the second barrier.

Listing 7.3: Pseudo code of mpi-bench – archetype of a communication kernel

```
Initalize buffers and test-specific buffers

MPI_Barrier()
startTime = getTime()

Execute the collective operation or point-to-point communication scheme
callTime = getTime() - startTime

MPI_Barrier()
barrierTime = getTime() - startTime

Cleanup
```

[18]The benchmark for 10 KiB messages is slightly different, because it requires much more repetitions, this is mentioned in the methodology.

7.6.2. Methodology

Conducted experiments The benchmark by itself iterates over several message sizes: 1 MiB, 100 MiB and 1000 MiB. Additionally, a modified benchmark version assesses 10 KiB messages. Short messages help assessing the latency model while larger payloads are increasingly affected by the network throughput characteristics – for 100 MiB, the influence of the latency is expected to be rather low on our network. It turned out that experiments with the largest payload do not offer more insight – results are comparable to the tests with 100 MiB and, therefore, omitted.

The benchmark is run for many different configurations: Configurations are pairs of node count and process count. Node numbers range from 1 to 10, and up to 20 processes are placed on the nodes. MPICH2 is run to distribute the processes in round-robin fashion on the available nodes and the sockets within. To illustrate the placement scheme, consider the configuration 2-5, here 5 processes are placed on the two nodes (and the four sockets) as follows: ((0,4),(2)),((1),(3)), that is, the first socket on the first node hosts Rank 0 and 4, the second socket on the same node just Process 2, and so forth. Simulated process placement mimics this process mapping, that means ranks are placed in the same fashion among the nodes.

The performance of Open MPI is measured for larger payloads, too. Results are put into the context of MPICH2 and the simulation. This also fosters the discussion of the MPI implementations' efficiency; one goal of the simulator is to evaluate new implementations to achieve optimal performance on clusters in the long run.

Measuring performance For larger payloads, three runs are performed in a single program execution. To cover the variability between independent runs, `mpi-bench` is restarted four times; the cluster scheduler might select other nodes for each run. Therefore, at least 12 measurements are made per configuration and collective call. A variant of the benchmark measures payloads of 10 KiB. In this case, communication time is much lower, therefore, more runs must be performed. For the small payload (and for assessing `MPI_Barrier()`), 1.000.000 and 10.000 repetitions are executed for intra-node and inter-node communication, respectively. In order to support analysis of the distribution of the times, short operations are timed individually and analyzed with the statistics tool R. Since node-local operations complete quickly, batches of 100 operations are timed together.

Simulating communication patterns By using MPICH2HD, the benchmark is instrumented with HDTrace to record MPI-internal communication for all collective operations – remember, with HDTrace, the point-to-point communication of collective operations can be traced. The real communication patterns are extracted from the trace files and then fed into the simulator. Due to the simplicity of the Point-to-point communication patterns, they are coded directly in Java. Thus, the simulated communication patterns are identical to the communication performed within MPICH2.

The benchmark's smallest payload is not instrumented, because tracing overhead disturbs the time – roughly 1.4 μs are required to trace an MPI command[19]. However, the simulation of collective operations includes this computational overhead if computation time is simulated (which is the case for most results presented). Therefore, the result of the simulator is biased by the tracing overhead.

To cover for the variability of computation time during kernel execution, the pattern with the minimum total runtime is extracted from the available trace files. With this methodology, the suitability of the models to simulate the behavior of MPICH2 is shown. Furthermore, the discrepancy between simulation and observation is discussed.

Since the measurements of local communication with 10 KiB messages measures time for batches of multiple operations, the simulation of a single operation and/or subtraction of the time for the additional barrier is not accurate because repeated invocation of `MPI_Sendrecv()` can cause the first process to finish much quicker, for example. Consequently, to represent the effect of early starters in the collective operations

[19]This is discussed in Section 7.1.

correctly, the simulation must recreate this behavior – the simulator loads the extracted point-to-point communication pattern 100 times and adds the barrier pattern (as recorded in the collective pattern).

For larger operations, the extracted pattern of the barrier is appended to all executions. Therefore, the exactly same operations are performed in the simulation as they are by the benchmark.

Computation time is taken into account by issuing compute jobs for the time between two MPI function invocations in the extracted communication pattern – for example, inside the collective call, some computation can happen after an `MPI_Recv()` before the data is transmitted to another process. This is especially true for `MPI_Reduce()`.

In the simulation, a network granularity of 512 bytes is used for 10 KiB messages and for 1 MiB messages, in order to allow pipelining of message packets. For larger payloads, 100 KiB packets are used.

Comparing observation and simulation In the comparison, absolute execution times for a command are shown in separate diagrams for intra-node communication, and for inter-node communication. Two diagrams are provided per kernel, because inter-node communication takes much longer than intra-node communication; scales are adjusted in the diagrams. The time is given per operation, that means time for batches is divided by the number of runs. The configuration (nodes, processes) is provided on the x-axis, while the time is given on the y-axis.

For small payloads, the mean, first and third quartile are computed with R and included in the diagrams. A diagram might include minimum and maximum; the maximum is omitted iff it is much worse and thus far away from the mean value.

Simulation results are put into relation to the measured results by computing the relative value: virtual time of the model is divided by the time for the metrics given in the legend, values larger than 100% indicate a slower execution in the virtual environment than in reality.

7.6.3. Assessing `MPI_Barrier()` and computation time

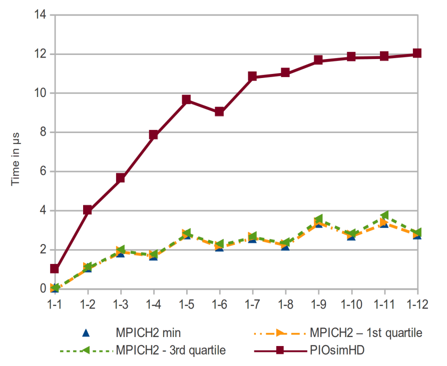
Assessing behavior of `MPI_Barrier()` is the basis for understanding further collectives, because reported times are protected by a barrier. The measurements and the simulator results that include computation are presented in Figure 7.15. Without simulating computation time, variability decreases; results are given in Figure 7.16.

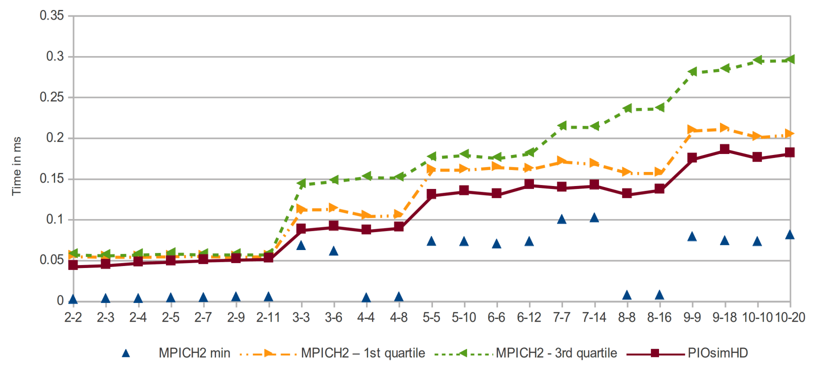
Observations and interpretation

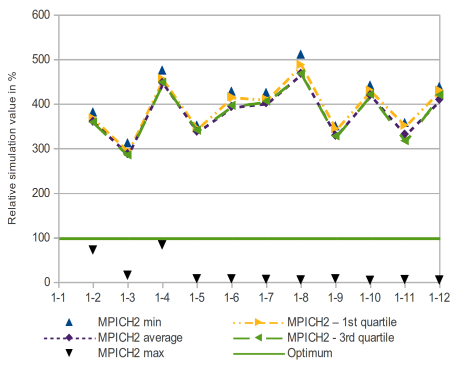
a) The time of MPICH2 increases with the number of nodes in inter-node communication, but is robust against increasing process counts (look at Figure 7.15b, compare the configurations n-n to the configurations in which twice as many processes are deployed on the same nodes (n-2n) and to configurations on two nodes). For example, the time for 8 processes on 4 nodes is better than deploying the processes on 8 nodes. The reason is that the barrier algorithm in MPICH2 is SMP-aware – thus, it tries to avoid inter-node communication. Internally, MPICH2 uses a binary tree algorithm, which explains the increasing time for 3, 5 and 9 nodes.

b) In local communication, the maximum observed execution time is much higher than the third quartile (see Figure 7.15c). The simulated time is three to four times the measured time of the first quartile (look at Figure 7.15a and Figure 7.15c). Still, the model is much faster than the maximum observation (triangles on the bottom). By looking at this figure, it seems that the simulation does not represent the average value for intra-node communication well.

c) All the times measured are between the values from the respective PIOsimHD runs that simulate computation, and the runs that do not model computation: When computation time is not simulated, the simulator produces much better estimates (compare Figure 7.15a to Figure 7.16a). The simulation


(a) Execution time for local communication.


(b) Execution time for inter-node communication.


(c) Relative model accuracy – local.

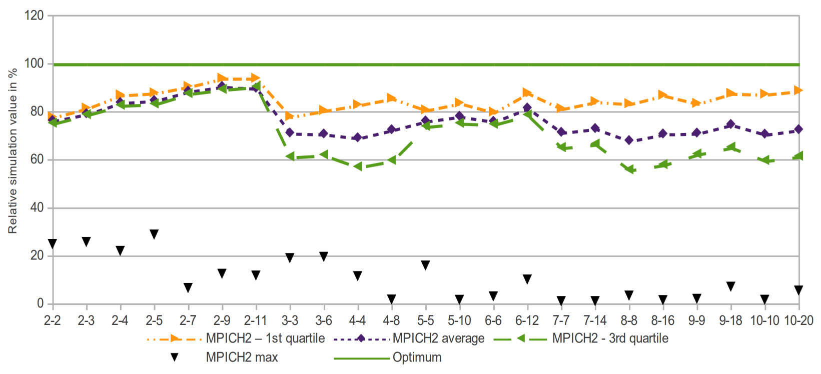
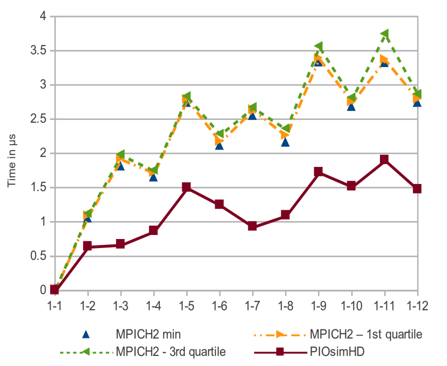

(d) Relative model accuracy.

Figure 7.15.: Performance of `MPI_Barrier()`.


(a) Execution time for local communication.

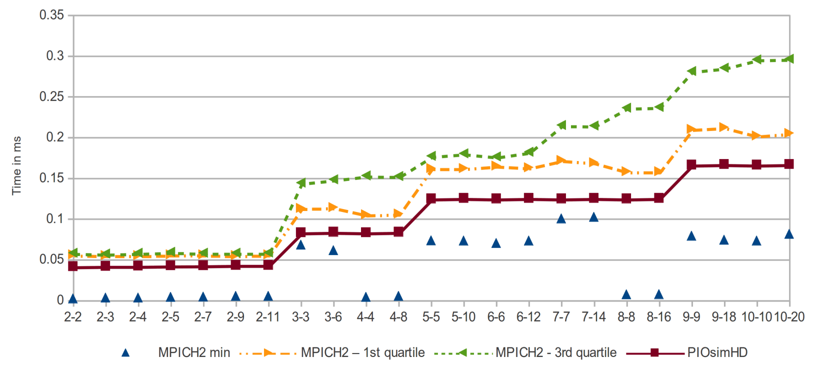

(b) Execution time for inter-node communication.

Figure 7.16.: Performance of `MPI_Barrier()` without simulating computation.

that includes computation time estimates performance much more accurately; it does not reproduce all maxima and minima of the measurement accurately, but many of them.

In fact, the overhead of tracing an MPI call (approximately 1.4 *μ*s) is in the order of the time for executing a single barrier. Since the barrier is implemented as a sequence of MPI commands, the recorded computation time (including overhead) is much higher than the actual communication time for the barrier. This explains the deviation in the estimate.

d) Without simulating computation, the duration of an inter-node barrier is monotonically increasing with the configuration, and of course lower than by performing computation (see Figure 7.16b). Behavior of intra-node communication is not monotonically increasing (see Figure 7.16a). However, the general trend is.

e) The overall behavior of simulated inter-node communication resembles the behavior of the first quartile quite well (refer to Figure 7.15b and Figure 7.15d); the accuracy of the simulation is above 80%.

f) Small variations are caused by the variation of computation and communication times. By comparing the results of simulating computation time or ignoring it, the simulator permits us to quantify the effect of MPI-internal computation on inter-node communication. For a barrier, the effect is in the order of 10% (compare Figure 7.15b to Figure 7.16b).

Communication happens within several *μ*s, which is in the order of the tracing overhead. However, the edge in the network model for the socket has a latency of 0.038 *μ*s, which is a very small fraction of 1 *μ*s. Since intra-node communication is biased by the tracing overhead, an accurate simulation of intra-node communication that includes computation is not possible. When computation time is not taken into account, time to prepare and cleanup the communication MPI is not simulated. Therefore, simulation without computation is expected to be faster than the observation.

Consequently, this effect must be considered when assessing intra-node communication of small messages with traced collective calls – therefore, it is important for 10 KiB point-to-point and collective messages. Inter-node communication is not affected by the tracing overhead because inter-node latency is much higher.

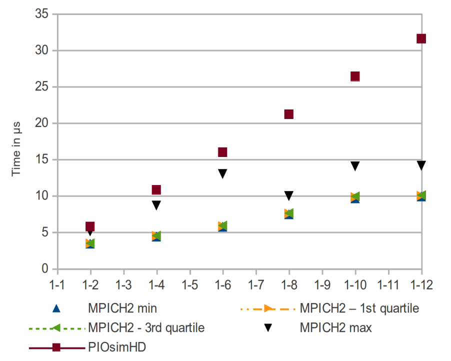
7.6.4. Point-to-point Communication of 10 KiB of Data

Results of the four evaluated point-to-point communication patterns are shown in Figure 7.17, Figure 7.18, Figure 7.19 and Figure 7.20, respectively. In any case the sender transmits 10 KiB of data to the receiver which is either the next process or root. Due to the simplicity of these communication patterns, the simulation is directly encoded in Java. Similar to the simulation of collective calls, the point-to-point scheme is repeated to match the measurement conditions, and protected by a barrier. Configurations which use an odd number of processes cannot form pairs of processes. Thus, the reported duration is zero and the data points are missing in the diagrams for relative model accuracy.

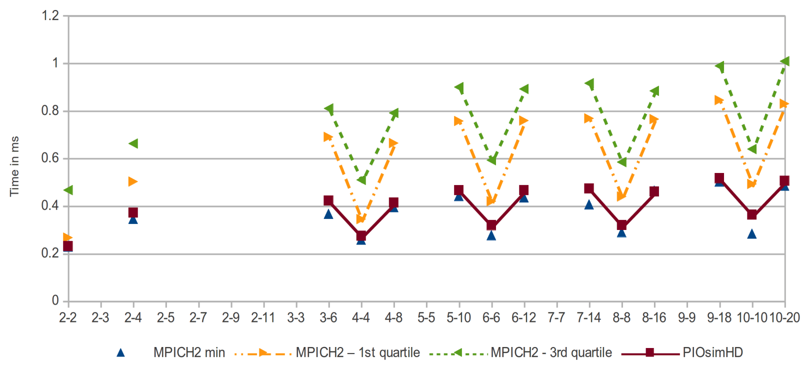
Observations and interpretation

a) In intra-node communication the time to transfer small amounts of data increases with the number of processes. While the kernel in which the processes transmit data to root matches the observation well (see Figure 7.19a and Figure 7.20a), the model overestimates the increase for the paired and the ring pattern by up to 300%. The reason for the high accuracy of data transfer to the root process is simple – the benchmark repeats the point-to-point communication by which the model has been calibrated (refer to Section 7.3).

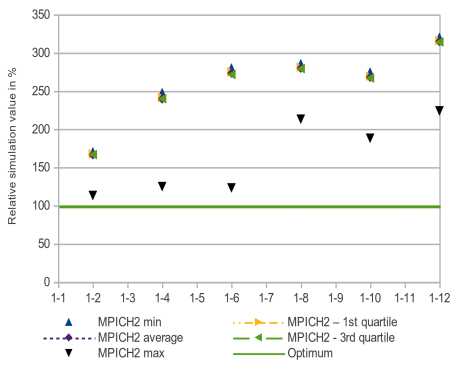
With the paired and the ring communication patterns, in the measurements, time increases linearly by about 1 *μ*s per process when more than two processes are used, but there is an offset for two processes of about 4 *μ*s(see Figure 7.17a and Figure 7.18a). This is an indicator that startup time for small message transfer is quite high and that data processing happens concurrently. Due to the small



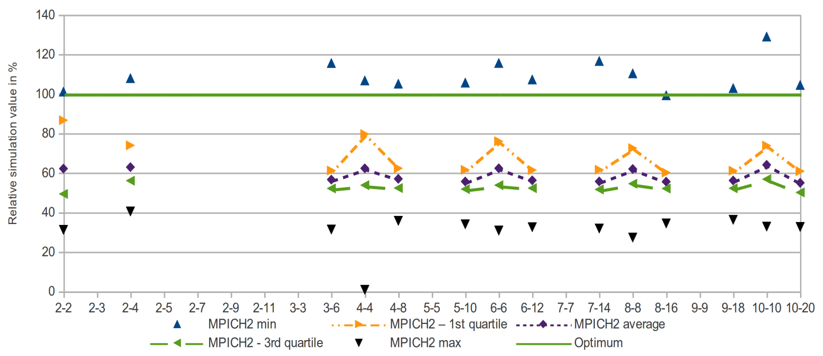
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

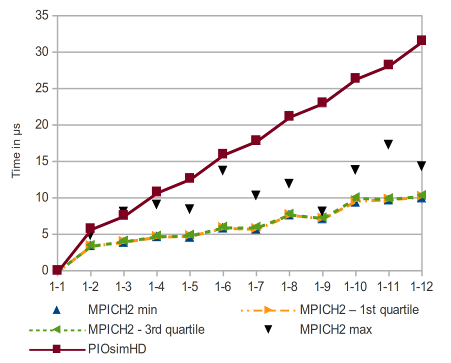


(c) Relative model accuracy – local.

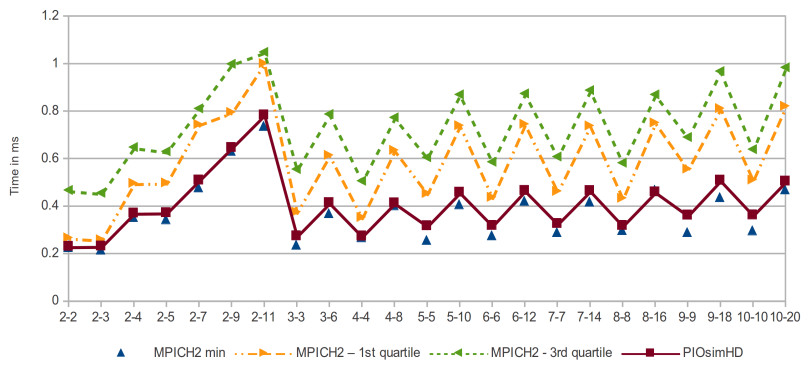


(d) Relative model accuracy – inter-node communication.

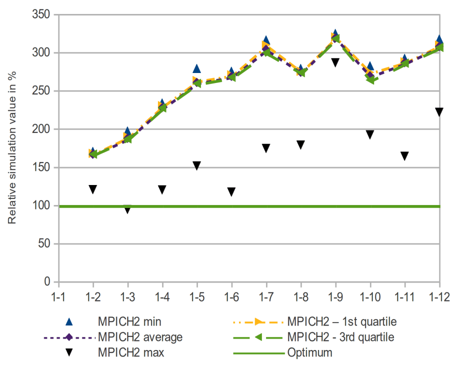
Figure 7.17.: Paired point-to-point communication and 10 KiB of data.



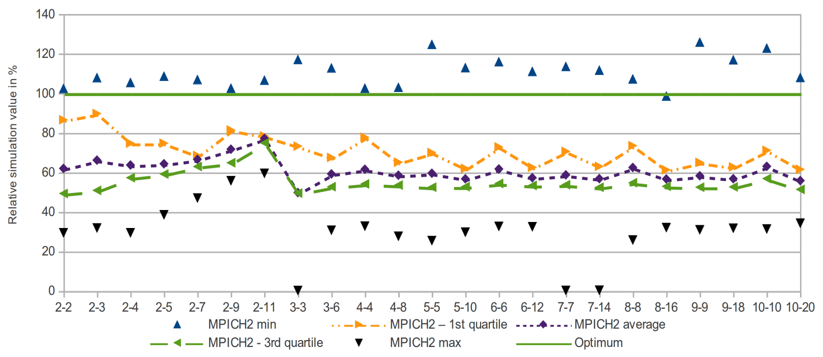
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

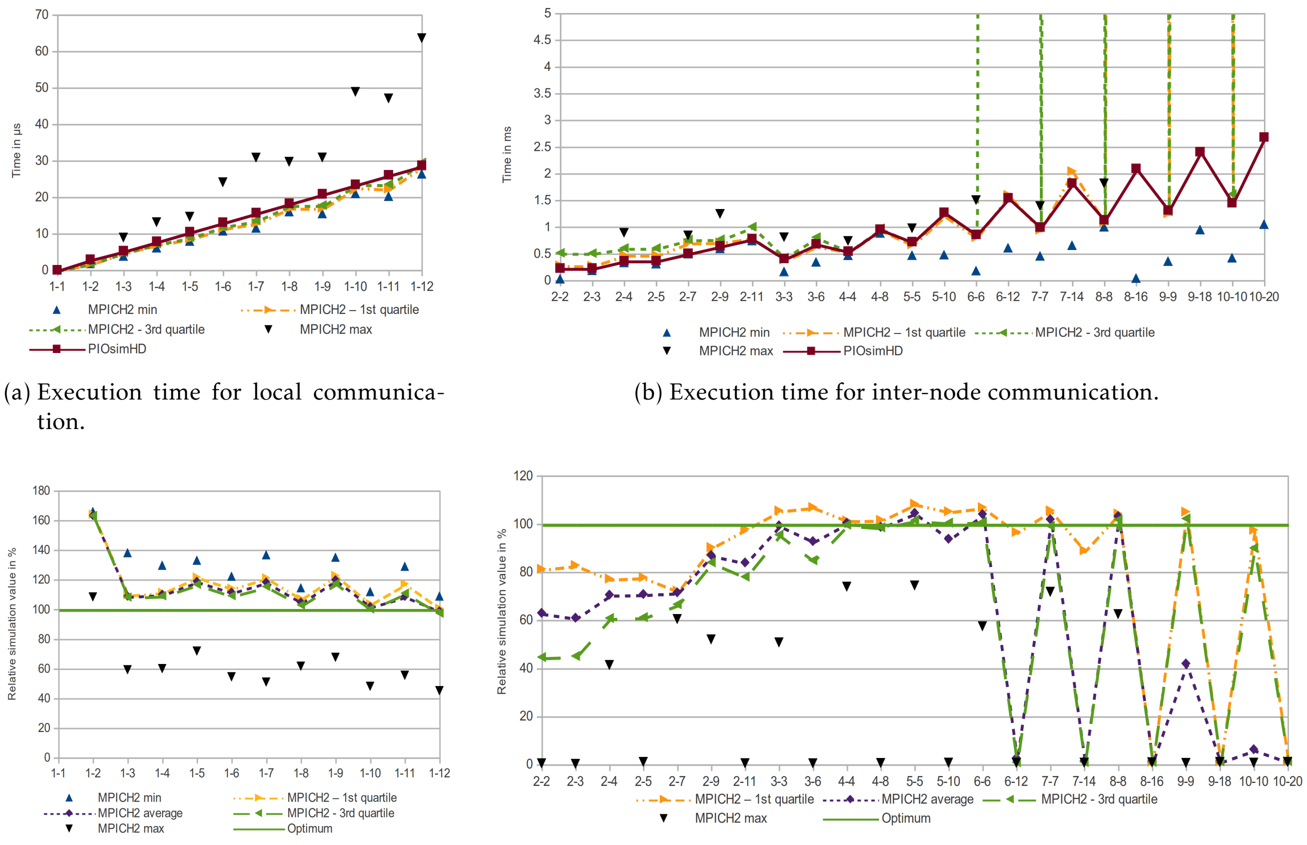


(c) Relative model accuracy – local.



(d) Relative model accuracy – inter-node communication.

Figure 7.18.: Ring point-to-point communication and 10 KiB of data.



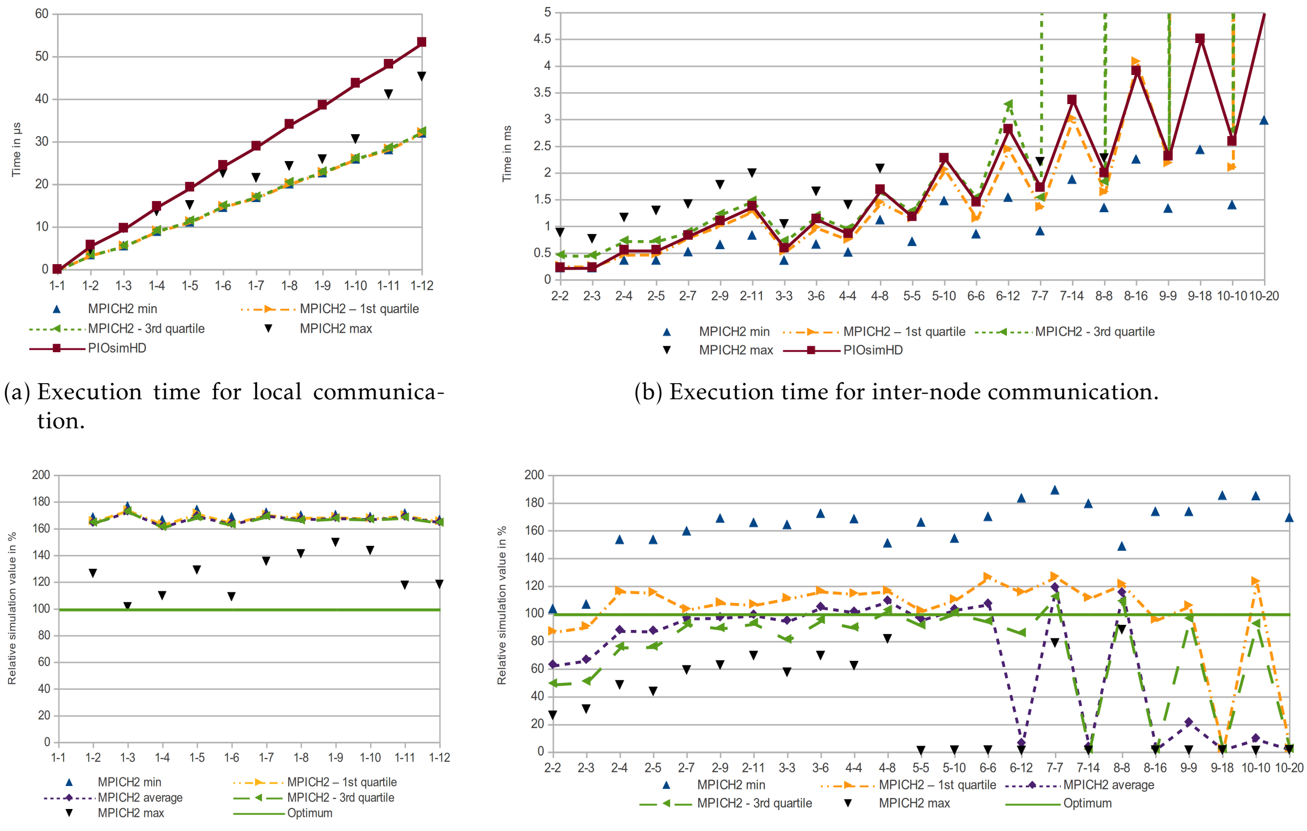
(a) Execution time for local communication.

(b) Execution time for inter-node communication.

(c) Relative model accuracy – local.

(d) Relative model accuracy.

Figure 7.19.: SendToRoot point-to-point communication and 10 KiB of data.



(a) Execution time for local communication.

(b) Execution time for inter-node communication.

(c) Relative model accuracy – local.

(d) Relative model accuracy – inter-node communication.

Figure 7.20.: SendrecvRoot point-to-point communication and 10 KiB of data.

payload, caching effects of the processors have an important impact on performance (as reported in Section 7.4.1). The current system model handles concurrent processing suboptimally, but overall, the behavior and the maximal observed time are estimated reasonable well for up to 6 processes. Since PIOsimHD does not simulate computation at all in this test, higher intra-node communication is not caused by tracing overhead.

b) Basic behavior of the inter-node communication is simulated quite accurately. However, results match the minimum execution time (refer to Figure 7.17b, Figure 7.18b, Figure 7.19b and Figure 7.20b). This is a bit surprising, because during parameterization, the average latency has been used as a reference. Therefore, intuitively, it is expected that the average observation matches the estimation. The SendToRoot and the SendrecvToRoot kernels (Figure 7.20b) are an exception; here the estimation matches the average time.

c) For many communication patterns, the communication time increases with the number of processes that are placed on a fixed number of nodes (this can be seen in all figures, for example, refer to the configurations with 2 nodes in Figure 7.18b and Figure 7.19b). The degradation is caused by the process placement – a single network interface is shared among all processes of a node. If the communication neighbor of a process is placed on the other node then additional communication over the slow Ethernet interface occurs – which is the case for most communication patterns and configurations. Due to the congestion model offered by PIOsimHD, this behavior can be simulated well.

d) When data is sent to the root process by multiple nodes, many observations take much longer than expected – the third quartile is above 200 ms which is too much comparing to a single paired communication which needs about 1–2 ms (the y-axis is cropped for these kernels, but almost vertical lines indicate the fluctuation, see Figure 7.19b). This happens if intra-node and inter-node communication occur at the same time for larger configurations starting with 6 – 12. The cause of this effect is unknown to the author. But due to the big discrepancy of simulation and conceptual model, it is likely to be a bug in the MPI implementation. To back up the simulation results, estimate the time to transmit data from 12 processes to Process 0 with 12 times 1 ms (according to the paired communication pattern). Since the result achieved by MPI is far slower than expected, the model cannot predict performance accurately (see Figure 7.19d).

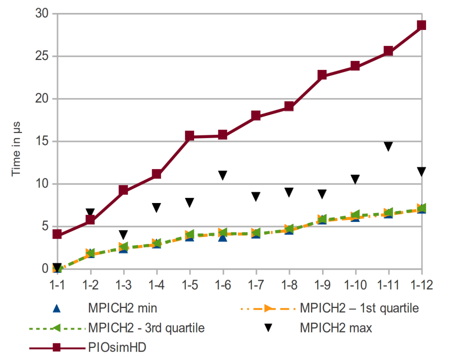
7.6.5. Collective Communication of 10 KiB of Data

Now collective calls are invoked for the small 10 KiB payload. Figures 7.21, 7.22, 7.23,7.24, 7.25 and 7.26 show the measurements and compare the simulated results of the collective calls for `MPI_Bcast()`, `MPI_Gather()`, `MPI_Allgather()`, `MPI_Scatter()`, `MPI_Reduce()` and `MPI_Allreduce()`, respectively.

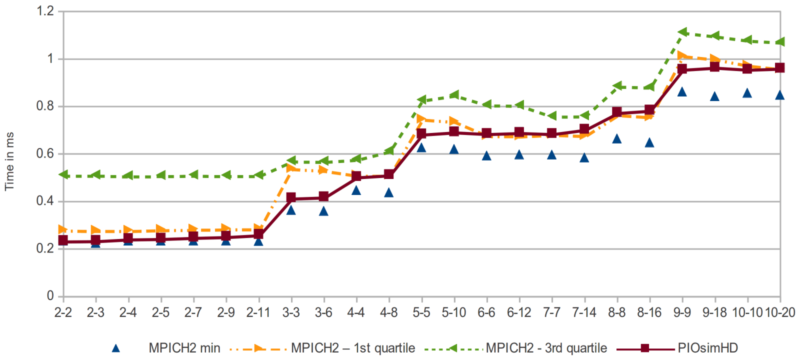
Observations and interpretation

a) Before simulation results are compared, a few selected considerations about observed MPICH2 performance: By looking at 7.21b, a monotone behavior with increasing node numbers, yet invariant to increasing numbers of processes per node, becomes visible – the implementation of `MPI_Bcast()` is SMP-aware.

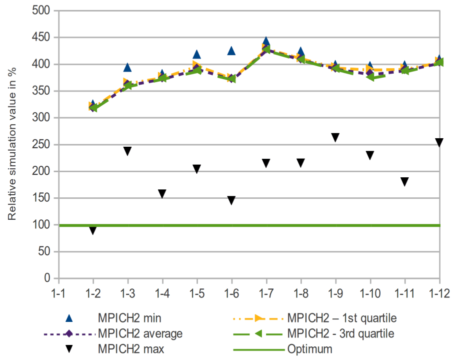
The time for an Allgather of configuration 2-11 is similar to the ones measured for configurations 10-10 and 10-20 (see Figure 7.23b). However, `MPI_Gather()` behaves differently; remember, with `MPI_Gather()`, all processes transfer data to the root process. This is due to the algorithm used in MPICH2. A faster algorithm can be easily envisioned and approximated: consider 11 processes on two nodes, doing an `MPI_Gather()` (roughly 1 ms) and an SMP-aware broadcast (10 times 0.25 ms), leading to 3.25 ms rather than the 7 ms that have been observed. In the same diagram, faster performance is achieved for configurations 4-8 and 8-16.



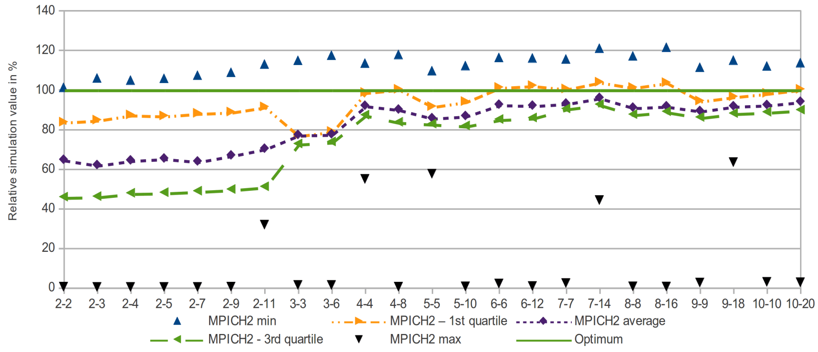
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

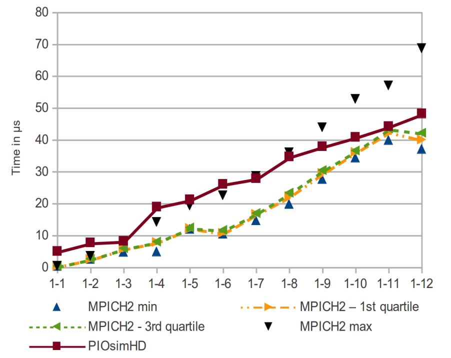


(c) Relative model accuracy – local.

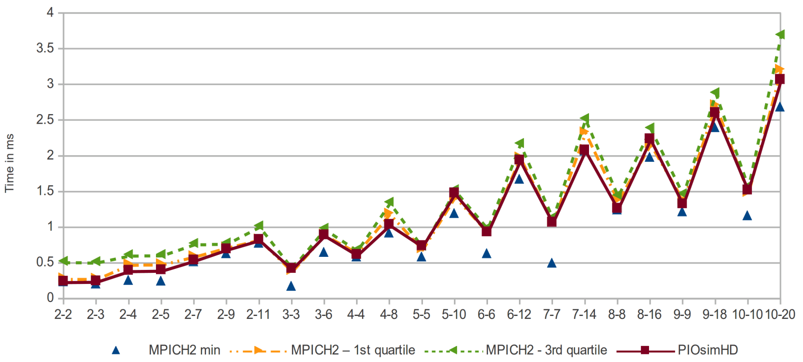


(d) Relative model accuracy – inter-node communication.

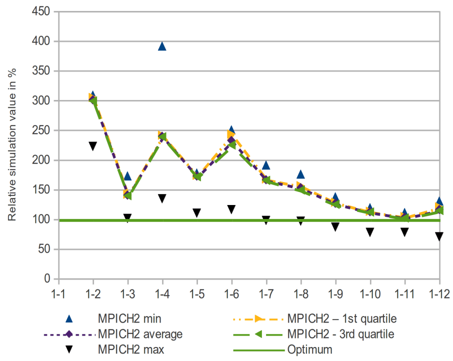
Figure 7.21.: `MPI_Bcast()` and 10 KiB of data.



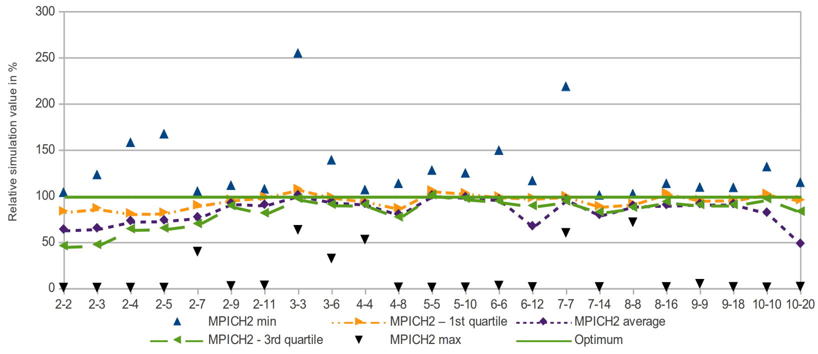
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

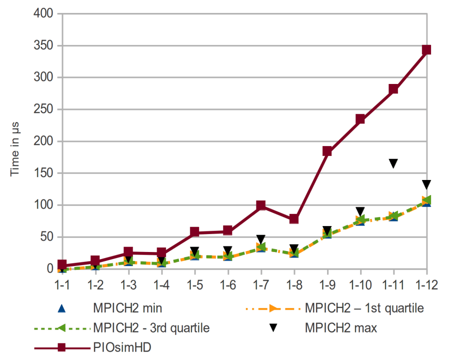


(c) Relative model accuracy – local.

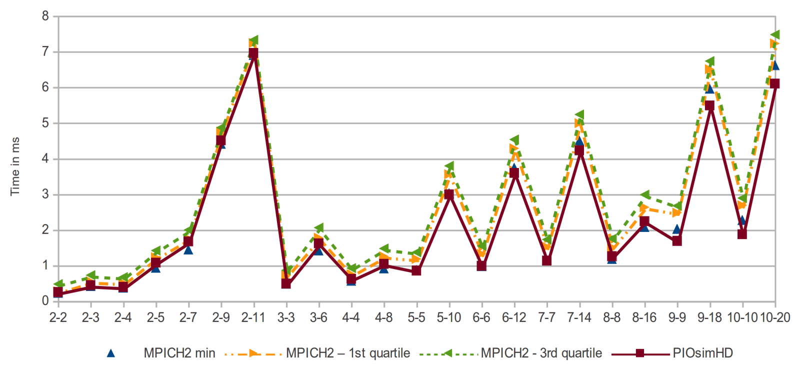


(d) Relative model accuracy – inter-node communication.

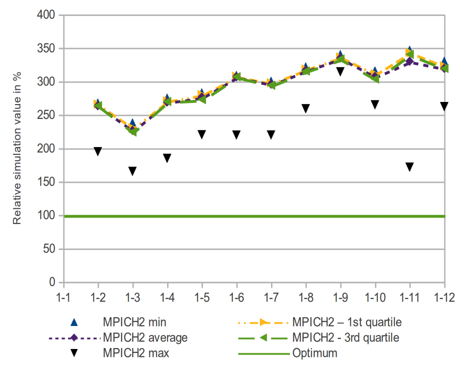
Figure 7.22.: `MPI_Gather()` and 10 KiB of data.



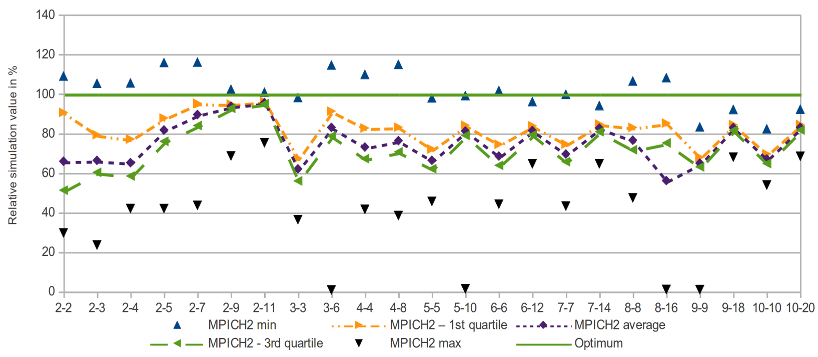
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

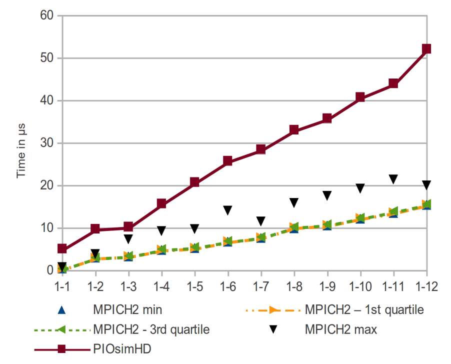


(c) Relative model accuracy – local.

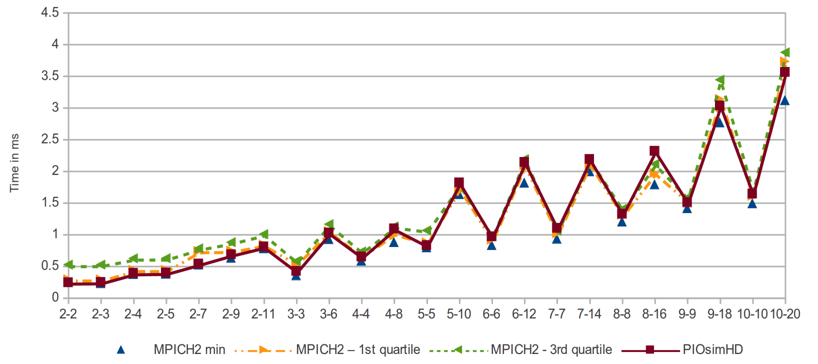


(d) Relative model accuracy – inter-node communication.

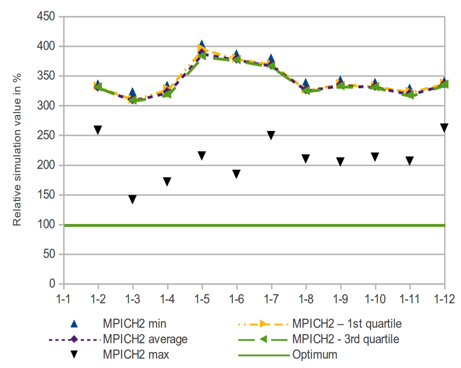
Figure 7.23.: `MPI_Allgather()` and 10 KiB of data.



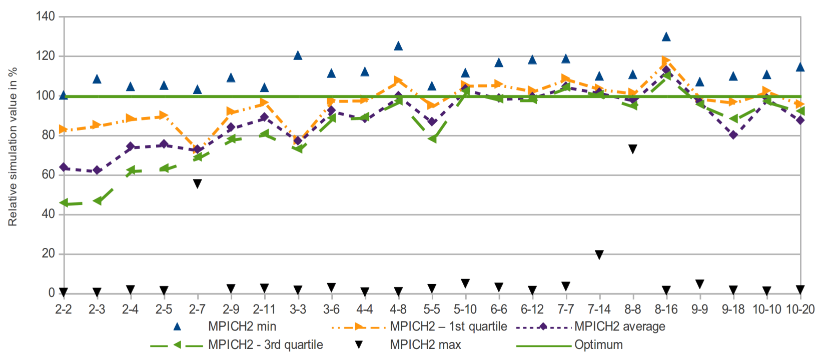
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

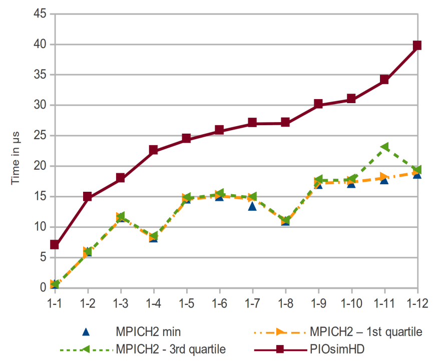


(c) Relative model accuracy – local.

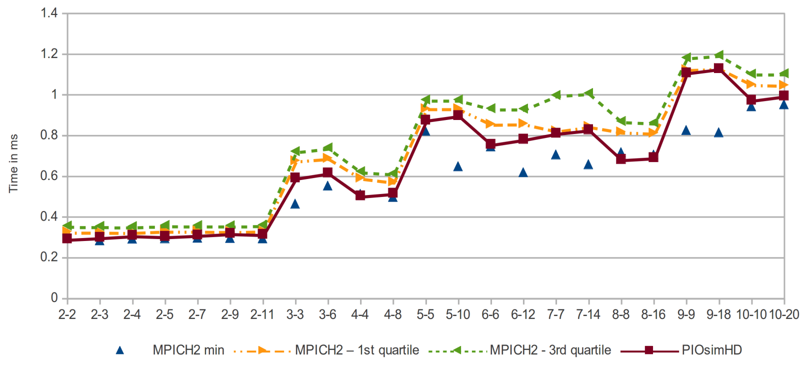


(d) Relative model accuracy – inter-node communication.

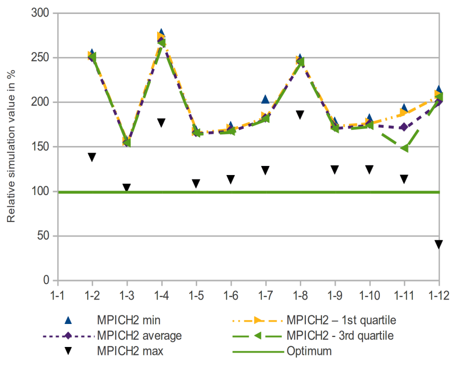
Figure 7.24.: `MPI_Scatter()` and 10 KiB of data.



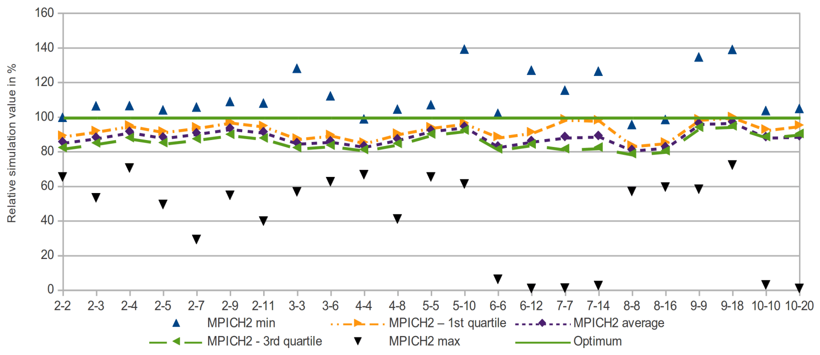
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

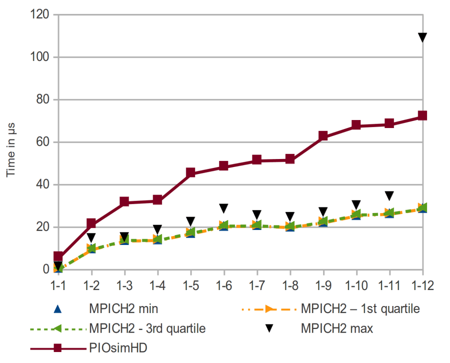


(c) Relative model accuracy – local.

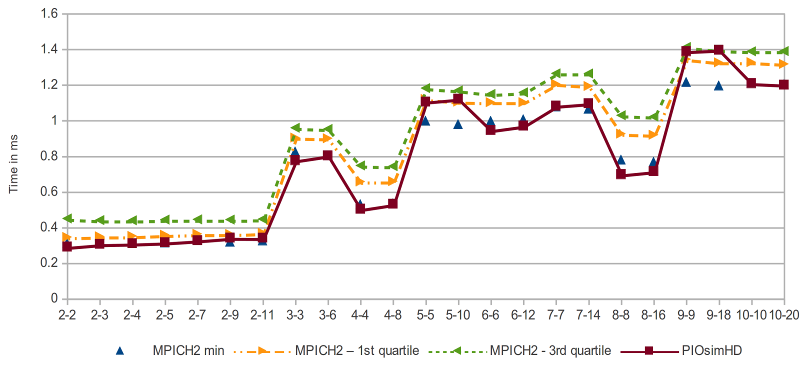


(d) Relative model accuracy – inter-node communication.

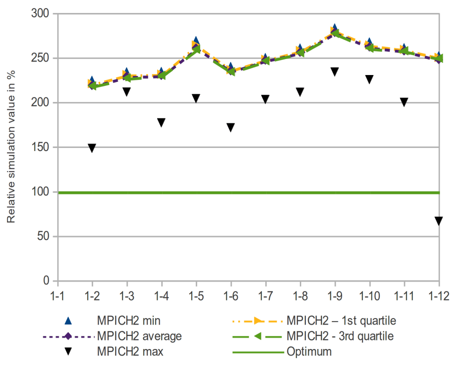
Figure 7.25.: `MPI_Reduce()` and 10 KiB of data.



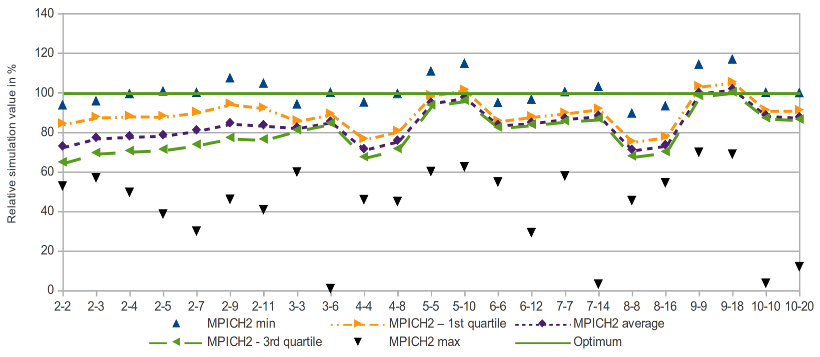
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

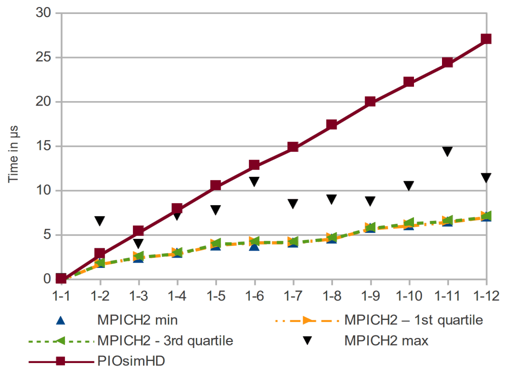


(c) Relative model accuracy – local.

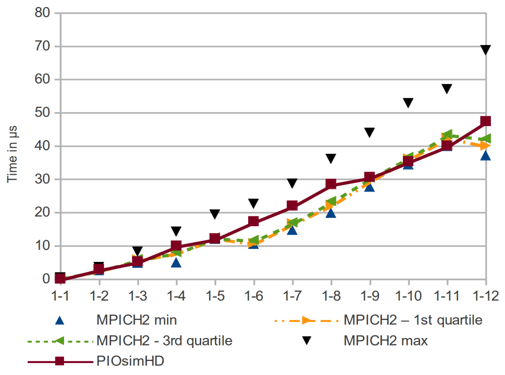


(d) Relative model accuracy – inter-node communication.

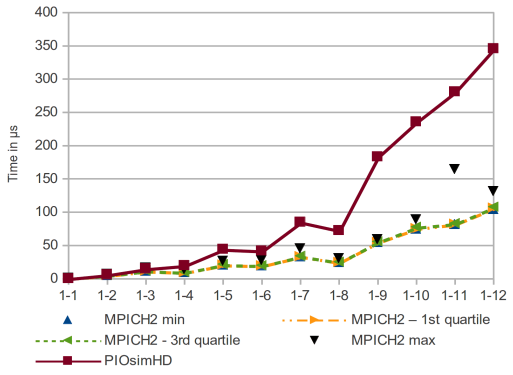
Figure 7.26.: `MPI_Allreduce()` and 10 KiB of data.



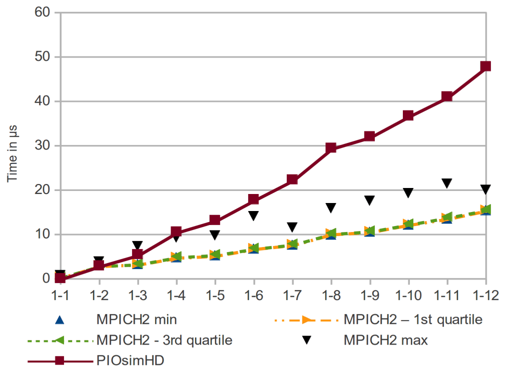
(a) `MPI_Bcast()`.



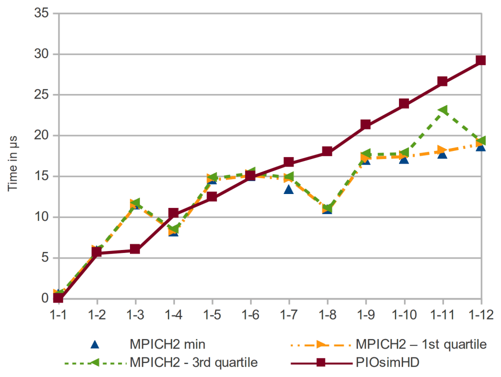
(b) `MPI_Gather()`.



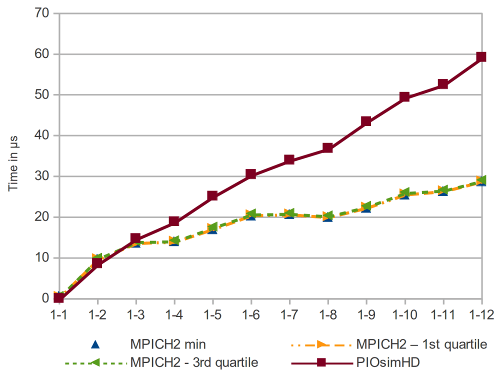
(c) `MPI_Allgather()`.



(d) `MPI_Scatter()`.



(e) `MPI_Reduce()`.



(f) `MPI_Allreduce()`.

Figure 7.27.: Execution time for local collective communication of 10 KiB of data – without simulating computation.

Performance of MPI_Reduce() and MPI_Allreduce() do not monotonically increase with the number of nodes (see Figure 7.25b and Figure 7.26b). On the one hand, this is due to the semantics of MPI_Reduce() (refer to page 46 for a discussion). On the other hand, it is inherent to the algorithm implemented in MPICH2, which is realized by basically doing an MPI_Reduce_scatter() with the recursive halving algorithm followed by the code to gather (or allgather) the distributed sums of the vector.

Consequently, with algorithmic changes, the performance could probably be improved. Further discussion and more examples of suboptimal behavior are provided in [KTML09].

b) For inter-node communication, the measured minimums (and often the first-quartiles) are approximated well by the simulator. Further, the parameterized model typically approximates the first quartile with a relative value between 80% and 110%.

c) Performance of local communication is underestimated by the model for Bcast, Allgather, Scatter, Reduce and Allreduce. The relative time is invariant with the number of processes, for most configurations it takes around 300% of the measured duration. As discussed in the point-to-point communication, this could be an indicator that the general local communication model or the parameterization has some bias. However, this is a bit counterintuitive to the fact that the estimation of MPI_Gather() has different characteristics (see Figure 7.22c) – the discrepancy starts at 300% and decreases with an increasing number of processes to match the observation quite well.

d) Unfortunately, as discussed in Section 7.6.3, tracing a single operation involves an average overhead of 1.4 μs per operation; the simulator replays traces for experiments, but compares the times to application runs without tracing. For example, the total duration recorded in the trace file to perform an MPI_Bcast() with 10 Processes is about 18 μs– but the third quartile of the measurement without tracing is 6 μs. The granularity of recorded timestamps is 1 μs. The simulator estimated 23.5 μs for the same execution. Therefore, an offset is expected.

Without simulating the computation contained in the traces, the tracing overhead could be ignored. However, then the MPI internal computation is not addressed which has an impact on the results as well. Either way, the accuracy cannot be determined accurately for small messages.

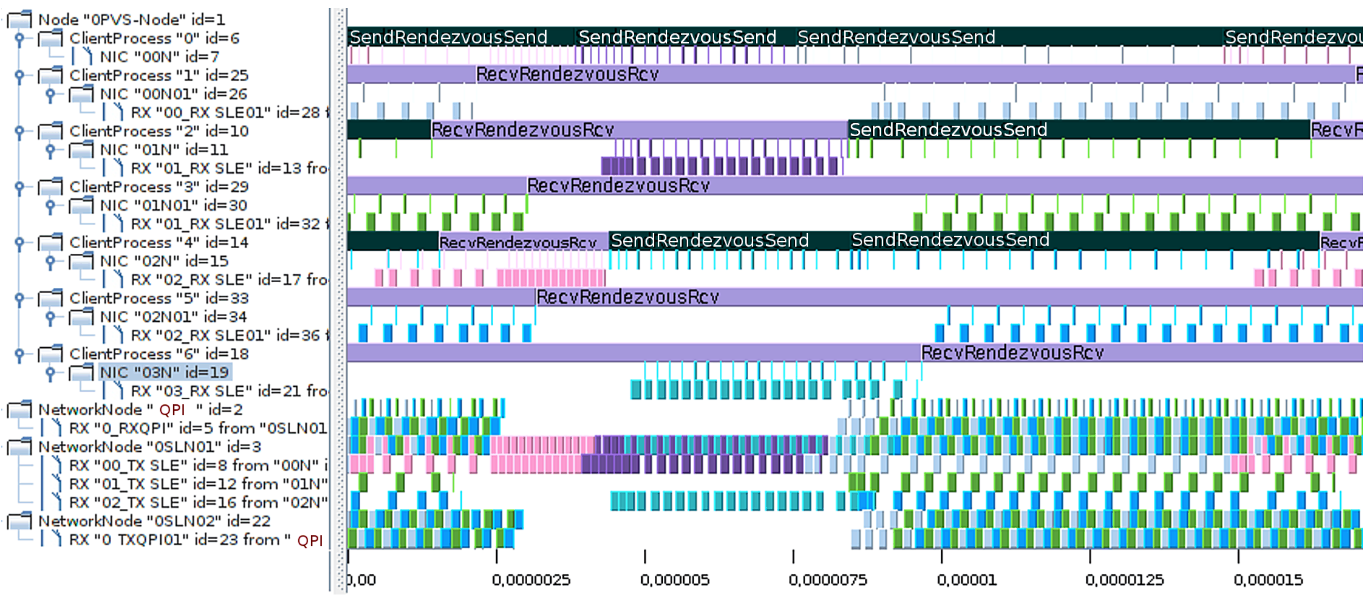
e) To assess the influence of the computation, the simulation is restarted without dealing with computation; the generated additional diagrams for local communication allow assessing its influence (compare the figures with computation with the simulation results without in Figure 7.27). Without computation, the gather operation and the reduce are simulated well, also, for the first few configurations, most other operations match well.

The broadcast operation is an exception to this rule – while it shows a linear increase in time with the number of processes, the gradient of the simulation is much steeper and similar to the results obtained without computation (see Figure 7.27a). As a side note, with 7 μs the measured intra-node broadcast is also much faster than the SendToRoot pattern, which can be considered to be an inverse pattern (it took 30 μs for 12 processes).

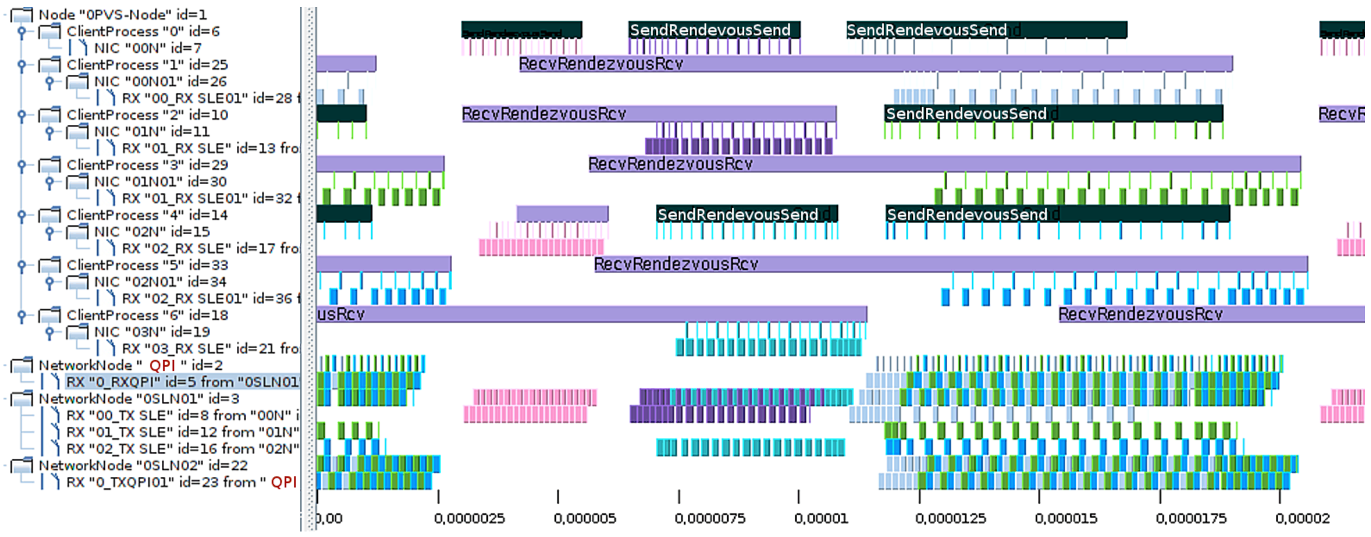
f) To assess the simulated behavior of MPI_Bcast() in intra-node communication, network activity has been recorded by the simulator. The excerpt in Figure 7.28 shows activity of a single broadcast operation – remember, overall, 100 individual operations are executed to emulate overlapping occurring in a real system.

In the simulation, the bottleneck is the modeled QPI interconnect, which interconnects both sockets – the second socket is busy to receive data from the first socket (look at the legend on the left, the QPI node is marked in red, the last time line shows the edge). Since the first socket is almost continually busy transferring messages in the case with computation, not much speedup can be achieved by skipping computation; at a certain point, the simulated memory channel of the socket is busy (as seen on the NetworkNode below QPI in Figure 7.28b). During the computation phases of a process, other processes still transfer data, hence the modeled memory channel is utilized to some extent.

The tree communication pattern used in MPICH2 communicates data through QPI (i.e., Rank 0 sends data to Rank 8, Rank 4, Rank 2 and Rank 1), for the given placement. Instead of communicating data to the second socket once, the observed pattern transfers almost always data between the two sockets; since the model uses a lower throughput between the two sockets, performance is lower. While the simulation model is rather simple and does not address cache usage, it is likely that an extension of the SMP-aware `MPI_Bcast()` implementation to make it aware of node-internal topology and memory channels could improve performance by transmitting data locally. The difference in the simulation and the observation might be caused by the processors' internal caches.

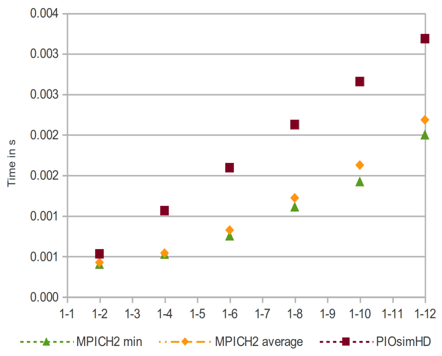


(a) Without simulating computation.

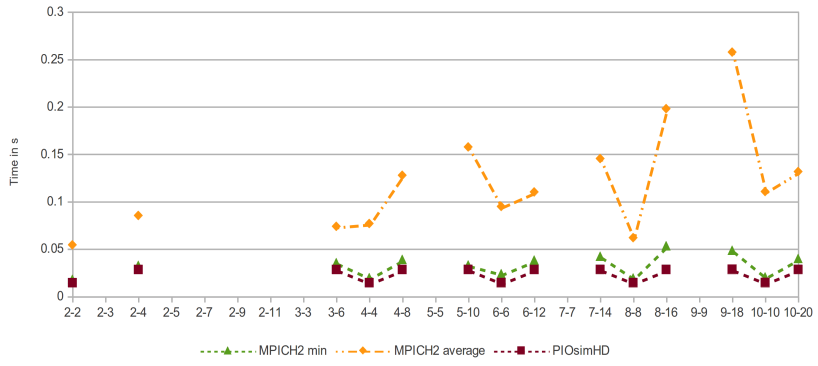


(b) With simulated computation.

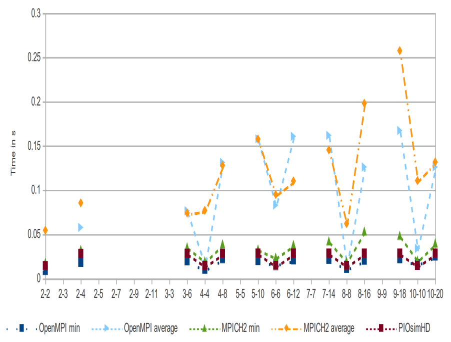
Figure 7.28.: Simulated activity of the `MPI_Bcast()` and 10 KiB of data for local communication. The screenshots of Sunshot show the activity of the client processes and all modeled hardware components over time; colors encode the type of the activity – for messages one color is assigned to each pair of sender and receiver. Empty space on a client represents computation.



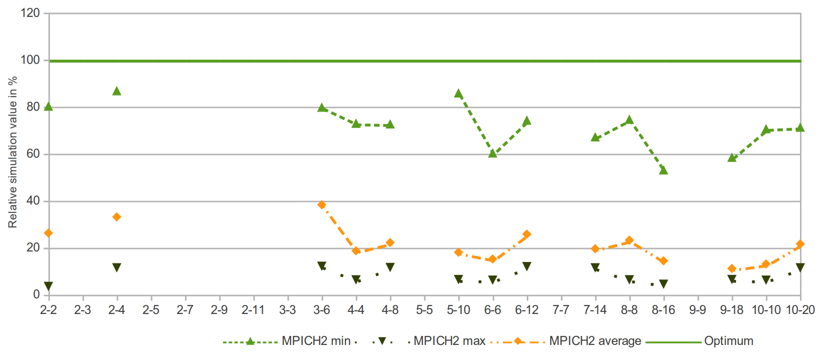
(a) Execution time for local communication.



(b) Execution time for inter-node communication.



(c) Relative model accuracy – local.



(d) Relative model accuracy – inter-node communication.

Figure 7.29.: Paired communication and 1 MiB of data.

7.6.6. Point-to-point Communication of Large Messages

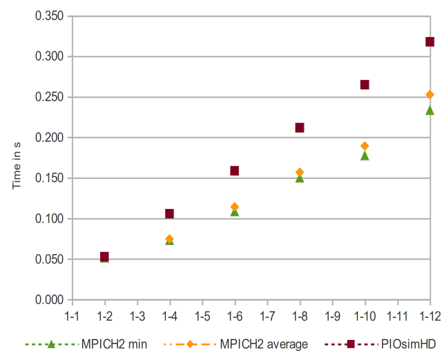
The results for message transport of the four point-to-point communication schemes are shown for 1 MiB and 100 MiB messages in Figures 7.29, 7.30, 7.31, 7.32, 7.33, 7.34, 7.35 and 7.36. Times measured with Open MPI are included, allowing a comparison to the results obtained with MPICH2 and the simulated results. Minimum, average and maximum lines are drawn if possible; if values are too far above the graph, then the line is removed. Frequently, the maximum value is removed because the outliers need much more time than the average value.

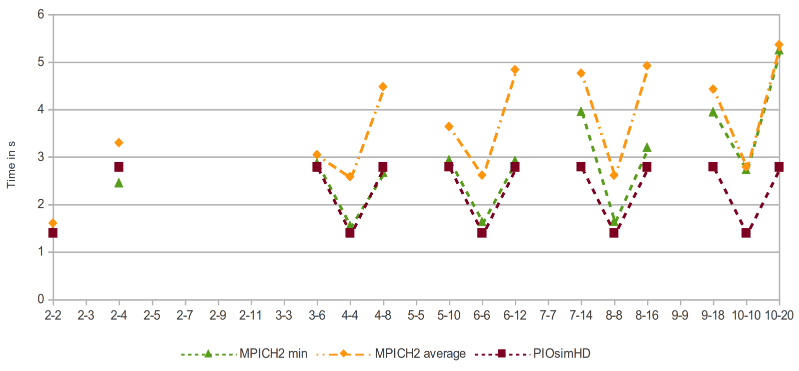
Observations and interpretation

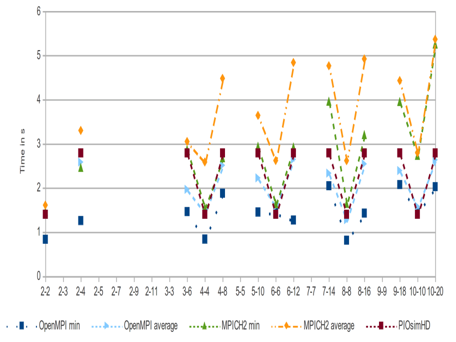
a) The time for the SendToRoot kernel is well approximated by the simulator for local communication and intra-node communication (look at Figure 7.33, and Figure 7.34).

In general, the simulation represents the minimum duration of inter-node point-to-point communication patterns for MPICH2 well (look at the relative model accuracy of the figures). With many kernels and local communication, estimated performance is slower by around 50%, but this is often similar to the visible discrepancy between MPICH2 and Open MPI (for example, look at Figure 7.32). However, high fluctuations in inter-node throughput increase the average measured value thus leading to an optimistic estimation by PIOsimHD.

b) In inter-node communication, the average and maximum measured transfer times are typically much higher than the minimum when transferring 1 MiB of data, which shows again that measurements vary to a large extent. In the paired or the ring communication scheme, average transfer of 1 MiB takes up to 4 times longer than the fastest transfer, and about 10% longer for patterns which exchange data with root only. Minimum and average values are closer together when 100 MiB of data are transferred. This observation is valid for MPICH2 and for Open MPI.


(a) Execution time for local communication.


(b) Execution time for inter-node communication.


(c) Relative model accuracy – local.

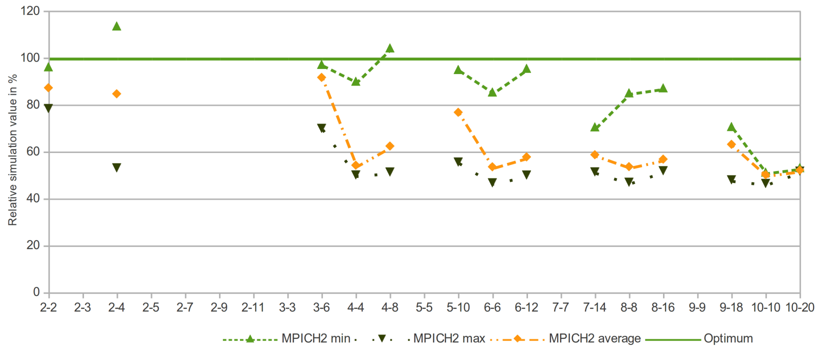
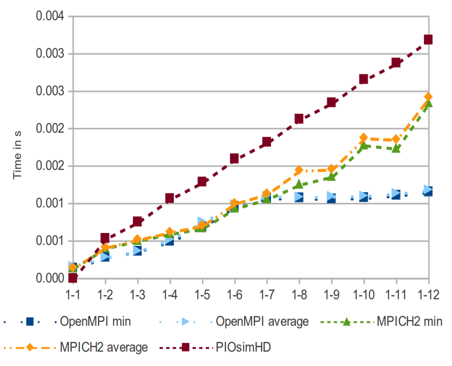
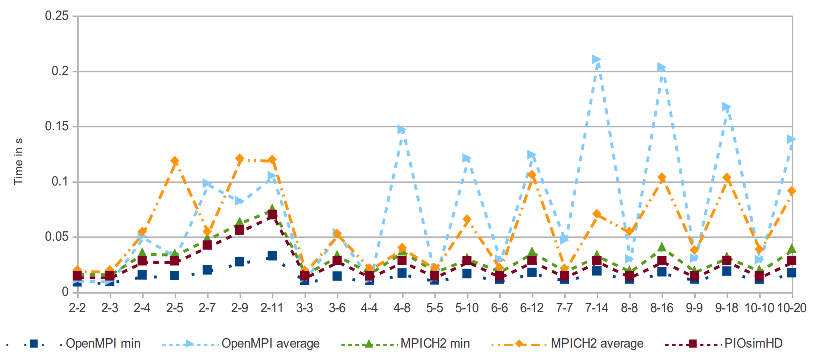
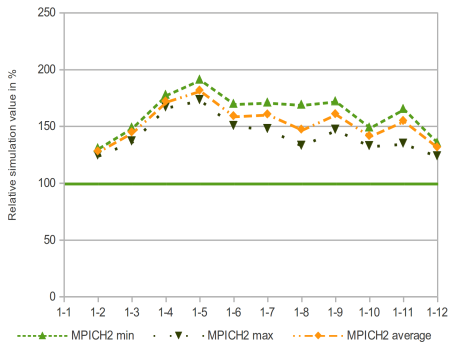

(d) Relative model accuracy – inter-node communication.

Figure 7.30.: Paired communication and 100 MiB of data.


(a) Execution time for local communication.


(b) Execution time for inter-node communication.


(c) Relative model accuracy – local.

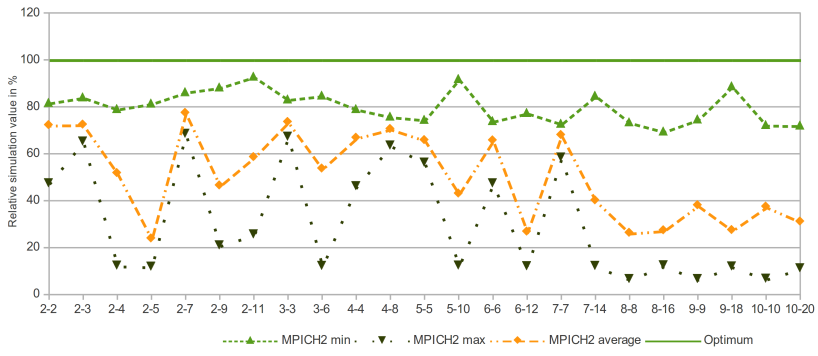
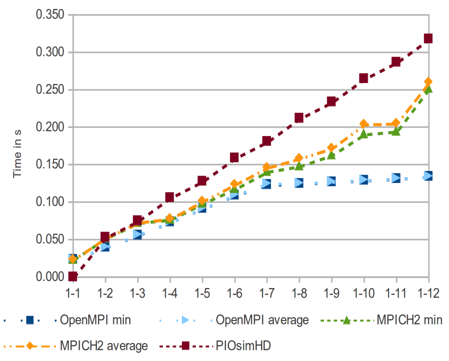
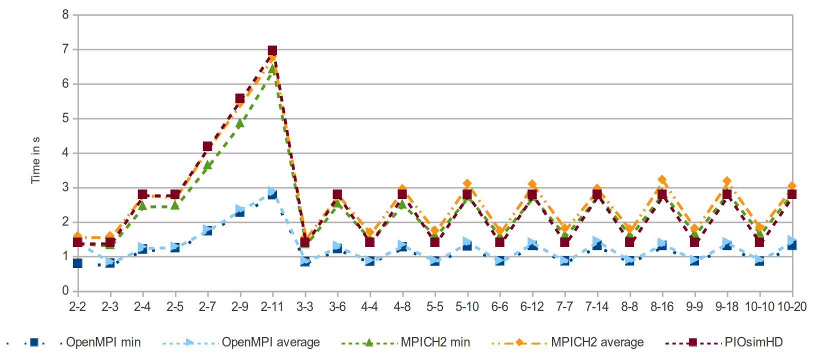

(d) Relative model accuracy – inter-node communication.

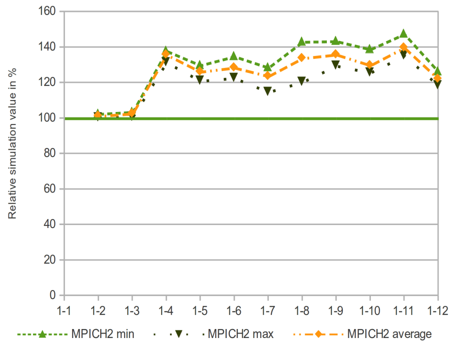
Figure 7.31.: Ring communication and 1 MiB of data.



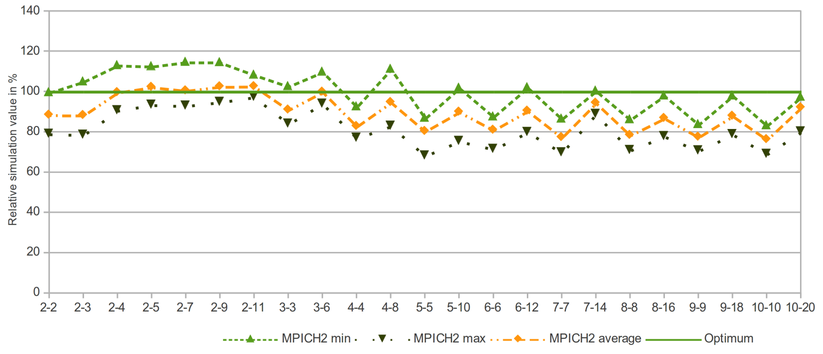
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

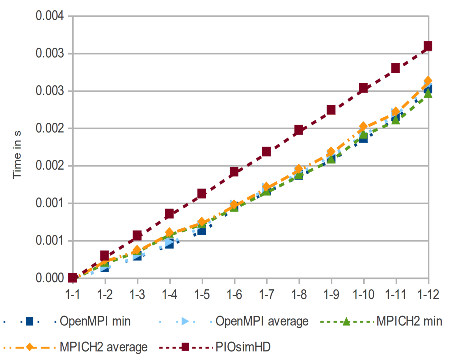


(c) Relative model accuracy – local.

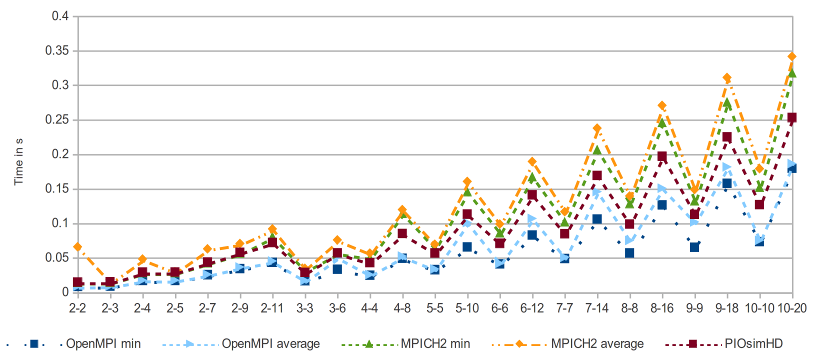


(d) Relative model accuracy – inter-node communication.

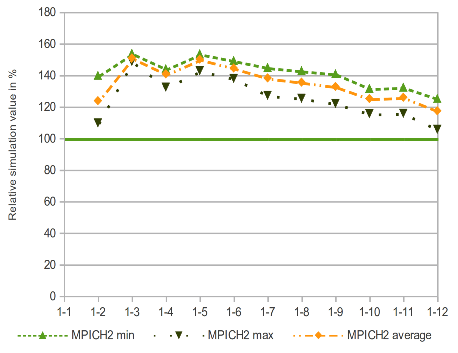
Figure 7.32.: Ring communication and 100 MiB of data.



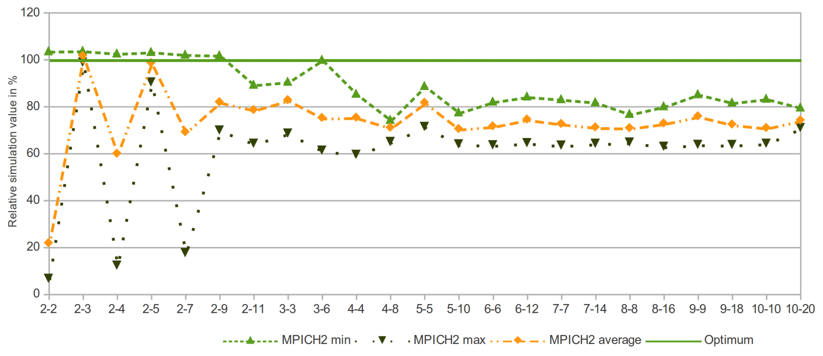
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

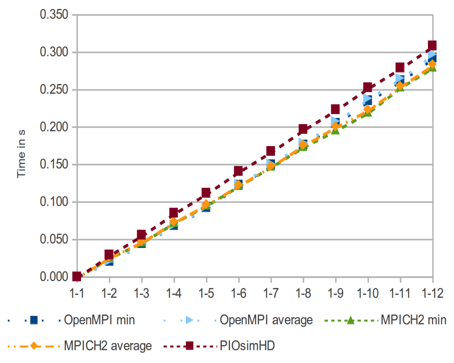


(c) Relative model accuracy – local.

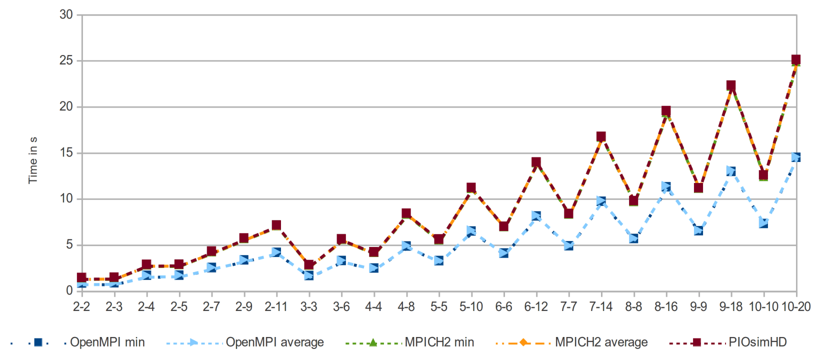


(d) Relative model accuracy – inter-node communication.

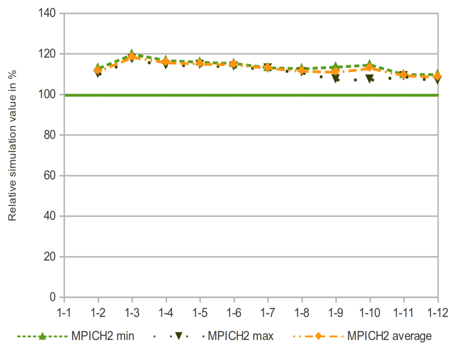
Figure 7.33.: SendToRoot communication and 1 MiB of data.



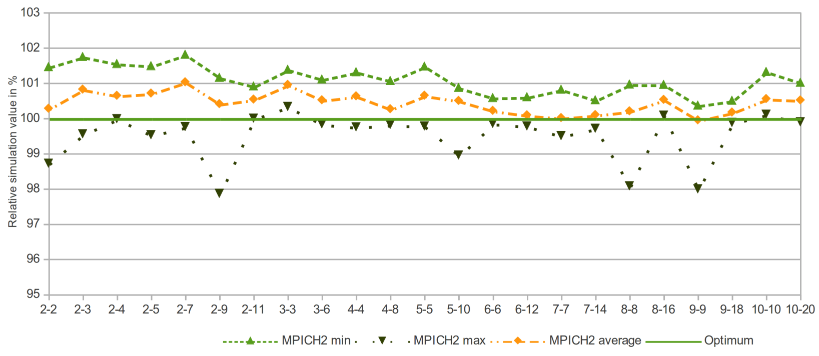
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

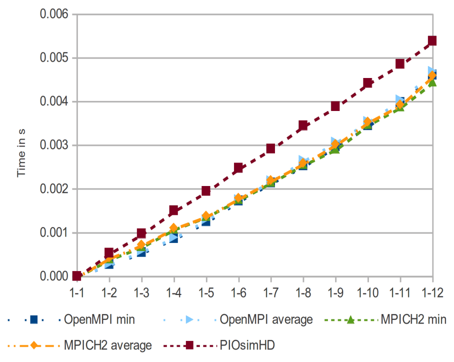


(c) Relative model accuracy – local.

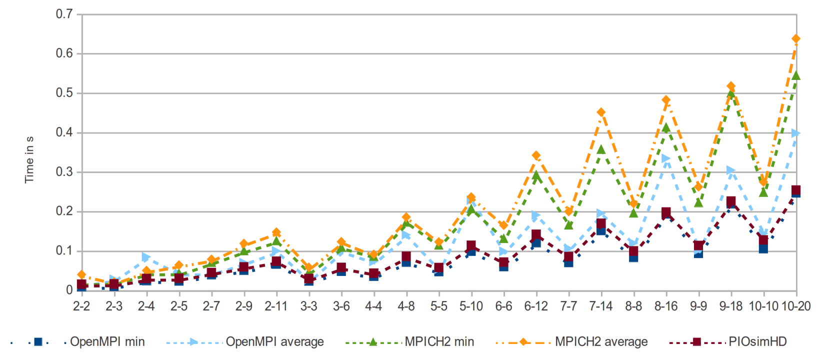


(d) Relative model accuracy – inter-node communication.

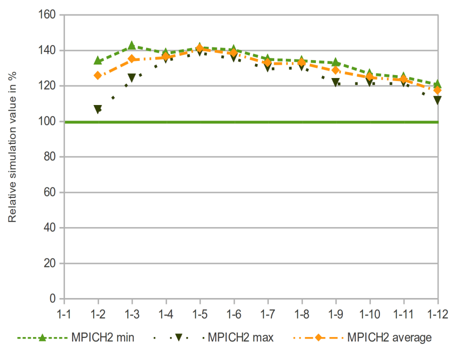
Figure 7.34.: SendToRoot communication and 100 MiB of data.



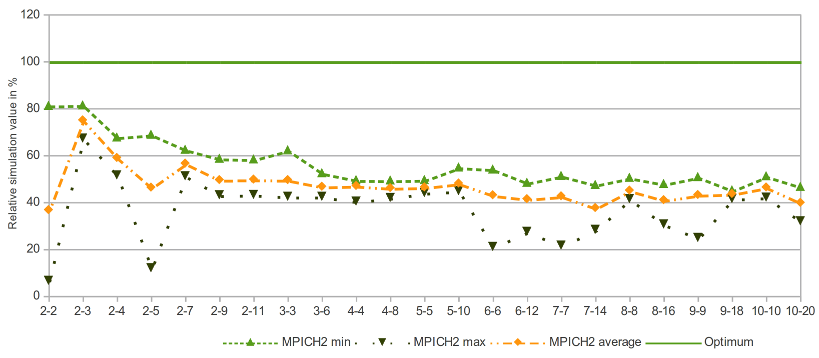
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

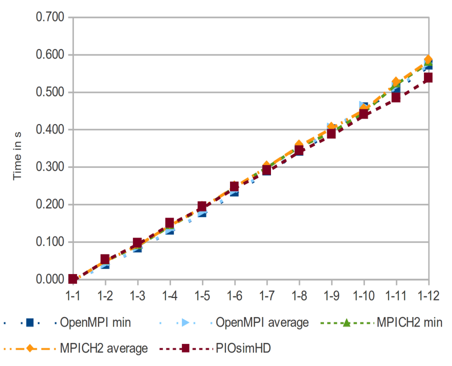


(c) Relative model accuracy – local.

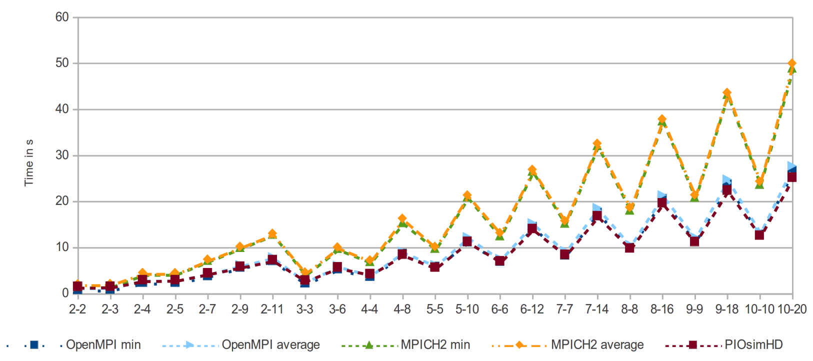


(d) Relative model accuracy – inter-node communication.

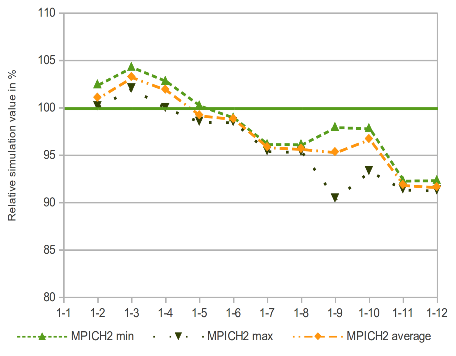
Figure 7.35.: SendrecvRoot communication and 1 MiB of data.



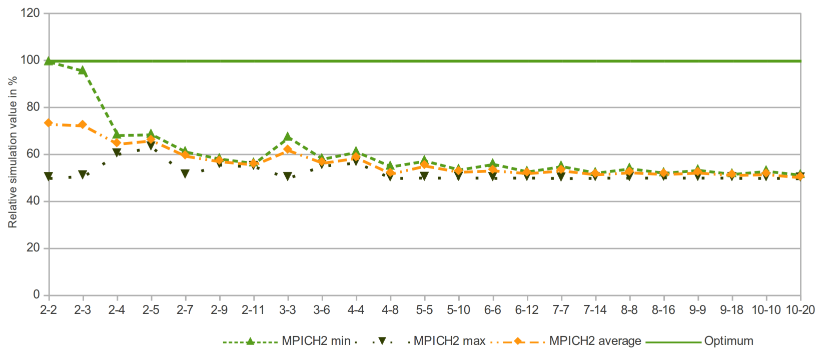
(a) Execution time for local communication.



(b) Execution time for inter-node communication.



(c) Relative model accuracy – local.



(d) Relative model accuracy – inter-node communication.

Figure 7.36.: SendrecvRoot communication and 100 MiB of data.

c) In intra-node communication, Open MPI is typically about twice as fast as MPICH2. As discussed before, the reason is the automatic utilization of both Gigabit Ethernet network adapters.

Interestingly, in intra-node communication Open MPI, is faster for the ring and the paired communication scheme, too. This contradicts the results presented in Section 3.5.4 on page 149; it has been slower by 6% for messages sizes of 1 MiB and slower by 40% for smaller messages.

d) In the ring communication, with Open MPI kernel time increases linearly up to 7 processes, then the transfer time increases only slowly (refer to Figure 7.32a). Thus, starting with 7 Processes, the communication throughput increases from roughly 5 GiB/s to 9 GiB/s (100 MiB of data is transferred per process in 0.14 s). Therefore, the observed memory access pattern of Open MPI utilizes available memory bandwidth better than MPICH2, for which the simulator has been calibrated.

e) Interestingly, for inter-node communication and exchanging data with `MPI_Sendrecv()` (and for larger configurations with a paired kernel), the simulator predicts half the measured time observed with MPICH2 (see Figure 7.32 and Figure 7.30). However, the results measured with Open MPI match the simulation quite well.

A brief theoretic consideration helps understanding the observations; Process 0 exchanges (sends and receives) 100 MiB of data with all other processes. Due to the bi-directional network, a performance is expected that is similar to the case where all processes just send data to root (see Figure 7.34). However, that is not the case, instead the time of the measurement doubles, it takes about 25 s to transfer data for 10 processes (1000 MiB) which is a throughput of 40 MiB/s (and 70 MiB/s for Open MPI). Therefore, it seems that just uni-directional communication is performed. Since for two processes, the network performance of 71 MiB/s can be achieved (the simulation results match perfectly), there is a problem in the MPI implementation and/or the network stack of our cluster.

7.6.7. Collective Communication of Large Messages

So far, point-to-point communication and collective operations with small amounts of data have been discussed. For a complete discussion, medium-sized and large messages must be assessed. Unfortunately, the amount of memory available per node does not suffice to perform calls with 100 MiB payloads in all configurations (where multiple processes are placed on one node). This is because both MPI implementations use internal buffers to hold intermediate data; configurations which failed are 5 to 12 processes on one node and 7 to 11 processes on two nodes. For these configurations in which memory suffices, the 100 MiB messages show a similar behavior than the 10 MiB messages. Therefore, only the collectives with 10 MiB messages are assessed.

Results for 1 MiB and for large 10 MiB collective `MPI_Bcast()`, `MPI_Gather()`, `MPI_Allgather()`, `MPI_Reduce()` and `MPI_Allreduce()` are provided in Figures 7.37- 7.48.

Observations and interpretation

a) With the large payload, PIOsimHD approximates local communication well; not only behavior – in many cases estimated runtime is at most 20-40% faster than observation (see Figure 7.38c, 7.40c, 7.42c, 7.44c, 7.46c, 7.48c). Since the collective calls rely on point-to-point communication, a small deviation is expected.

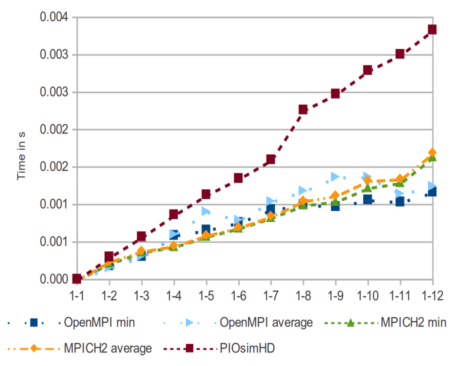
For `MPI_Bcast()` and `MPI_Reduce()`, the 10 MiB payloads are more precisely approximated by PIOsimHD than the smaller ones (e.g., compare Figure 7.38c to Figure 7.37c). Presumably, deviation is caused by the processor cache; 1 MiB of data fits into the L3 cache, and the broadcast and the reduce operation keep at most one copy per process in the buffer – the other calls need more data. Therefore, all configurations fit into the available L3 cache of 12 MByte per processor. However, this assumption cannot answer the question why performance of the other MPI collectives does not vary for configurations which should fit into the cache, such as configuration *1–2*.

`MPI_Allreduce()` is also better approximated for larger payloads. Since it is similar to `MPI_Reduce()` in respect to semantics, it benefits from the same effects. While the simulation for `MPI_Scatter()` matches well, it underestimates performance for 1 MiB of data, but overestimates performance for more than 6 processes on a single node.

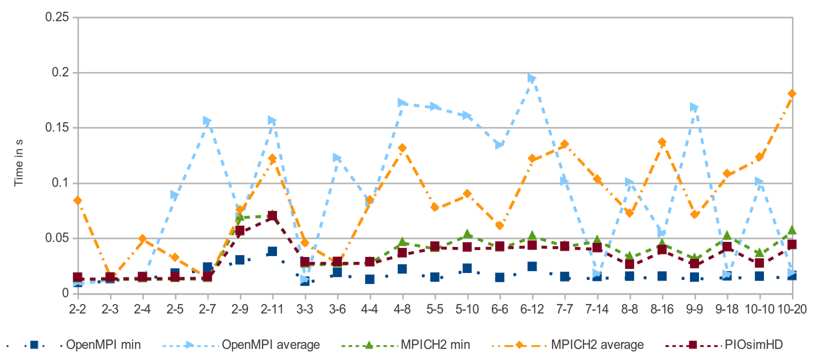
b) For inter-node communication of all collective calls, the variability of 1 MiB messages is much higher than when transferring more data, that means on average much more time is needed for data transfer than in the best case (e.g., compare Figure 7.39b to Figure 7.40b). The maximum is not given in the figures, because it is be much longer than the average and thus could not be rendered into the scale. For a payload size of 10 MiB of data, there is almost no difference between minimum and average for `MPI_Gather()`, `MPI_Allgather()` and `MPI_Scatter()` (see Figure 7.40b, Figure 7.42b and Figure 7.44b). In general, there is more variation with `MPI_Bcast()`, `MPI_Reduce()` and `MPI_Allreduce()` (look at Figure 7.38b, Figure 7.46b and Figure 7.48b).

c) Neither MPICH2 or Open MPI show better performance in all cases; sometimes Open MPI performs better, sometimes MPICH2. To give a few examples: MPICH2 performs better in inter-node configurations of multiple processes per nodes for `MPI_Bcast()` and 10 MiB payload, but for 1 MiB payloads, Open MPI is better (look at Figure 7.38b and Figure 7.37b). Open MPI does a faster `MPI_Allgather()` of 10 MiB of data in configurations with 2 nodes but achieves similar performance for other configurations. Reduce is slower with Open MPI for configurations *2–7* to *2–11* and when more than one process is placed per node, but it is faster for the other configurations (look at Figure 7.46b).

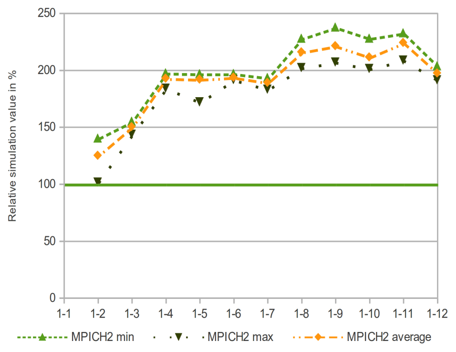
One might claim that there might be an issue on our cluster with the SMP-awareness of Open MPI. However, Open MPI needs roughly half as much the time for `MPI_Scatter()` and `MPI_Gather()`. But by keeping in mind that Open MPI can use both network interfaces, observed performance is disappointing.



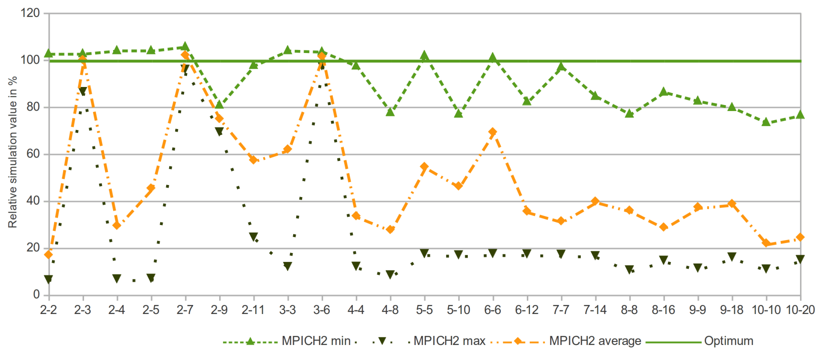
(a) Execution time for local communication.



(b) Execution time for inter-node communication.



(c) Relative model accuracy – local.



(d) Relative model accuracy – inter-node communication.

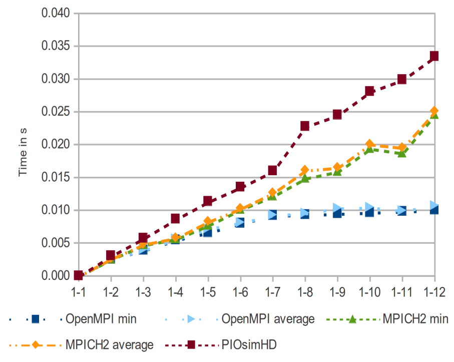
Figure 7.37.: `MPI_Bcast()` and 1 MiB of data.

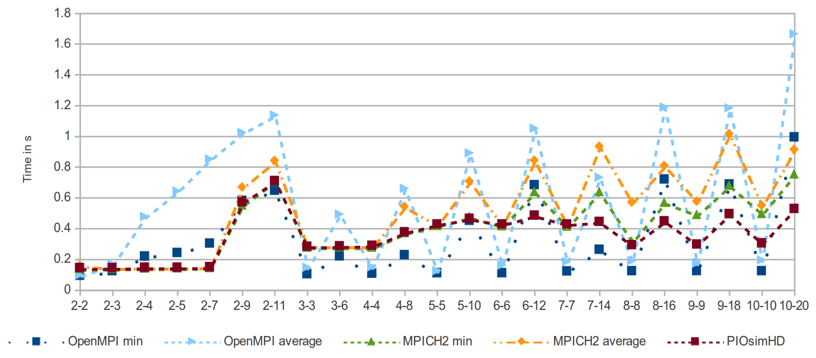
In most cases, Open MPI performs better in intra-node configurations. An exception are a few configurations of broadcasting 1 MiB of data, here MPICH2 is slightly faster (see Figure 7.37a).

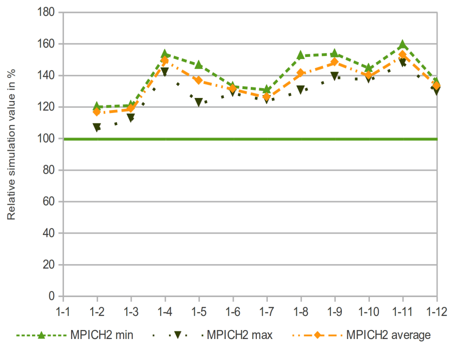
d) A closer look at the local performance achieved by Open MPI shows for scatter, gather and allgather, that the time for execution is proportional to the number of processes. However, for the ring point-to-point communication and the remaining collective calls, starting with 6 or 7 processes, the execution time increases only minimally (e.g., look at Figure 7.31a) – for reduce and 10 MiB of data it starts already at 4 processes (see Figure 7.46a). Maybe this is caused by an increasing cache re-use or better utilization of the memory subsystem. Due to the simulator's coarse grained simulation of memory access, it cannot simulate this behavior, yet.

e) The simulated time for intra-node configurations does not increase linearly in all cases; a large time increase is visible for the configuration with 8 processes and `MPI_Bcast()` (see Figure 7.37a). This can be seen in `MPI_Gather()` and `MPI_Scatter()`, too. Further, there are sometimes small plateaus visible. In many cases, the simulated behavior is visible in the observation as well, although the actual change in values is different. For example, the current model is able to predict a performance increase for MPICH2 in configuration *1–8* with `MPI_Reduce()`; this configuration needs less time than configurations *1–6* and *1–7* (see Figure 7.46a). In this case, the algorithm is responsible for the lower performance.

Sometimes the performance jumps are shifted to the neighboring configuration, but sometimes the gap is not visible in the observed times. In those cases, it is likely that the memory access pattern changes throughput of the memory subsystem. Therefore, the current model cannot capture these effects. However, by comparing model and measured behavior, the impact of the memory subsystem could be better understood.


(a) Execution time for local communication.


(b) Execution time for inter-node communication.


(c) Relative model accuracy – local.

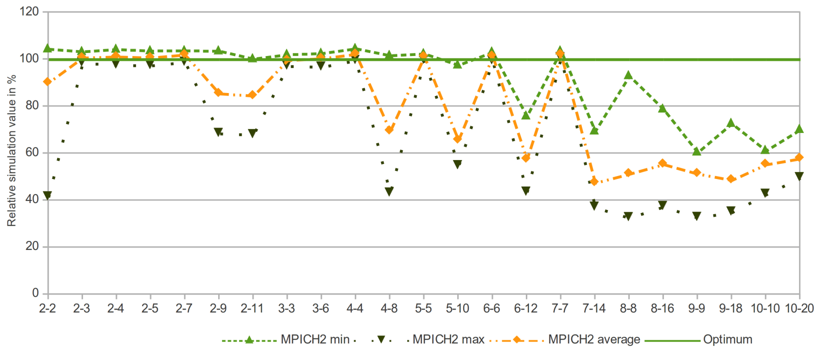
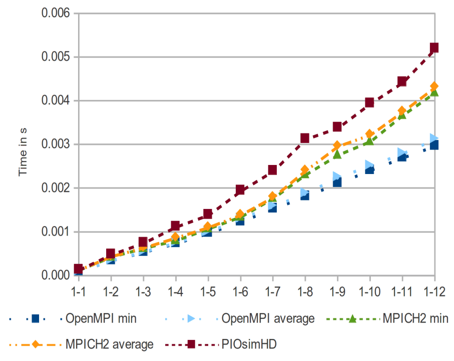
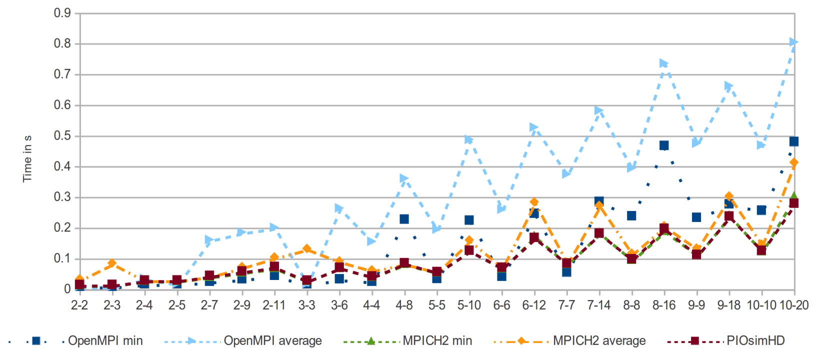
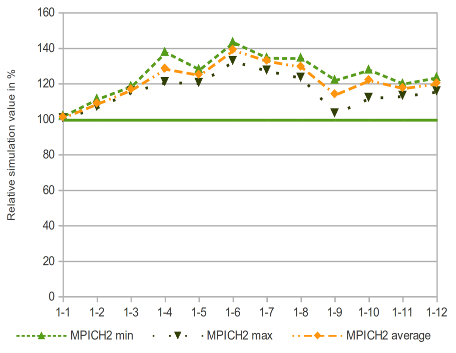

(d) Relative model accuracy – inter-node communication.

Figure 7.38.: `MPI_Bcast()` and 10 MiB of data.


(a) Execution time for local communication.


(b) Execution time for inter-node communication.


(c) Relative model accuracy – local.

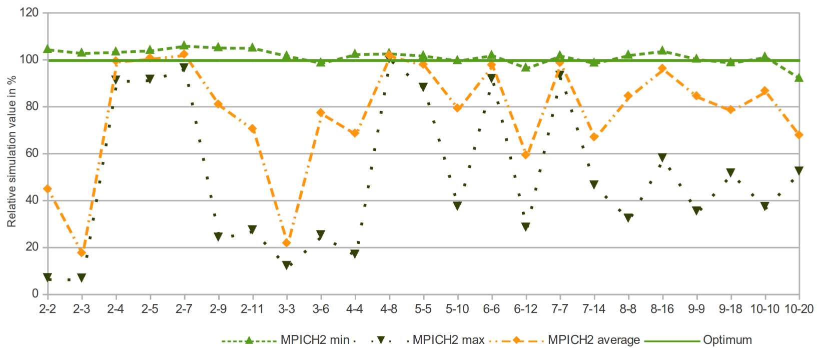
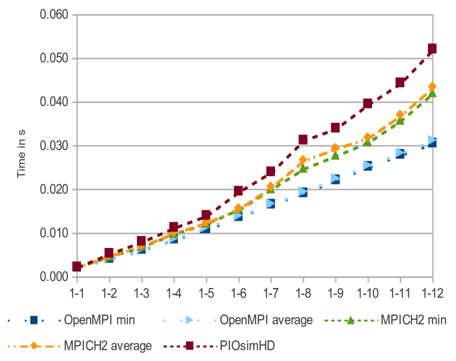
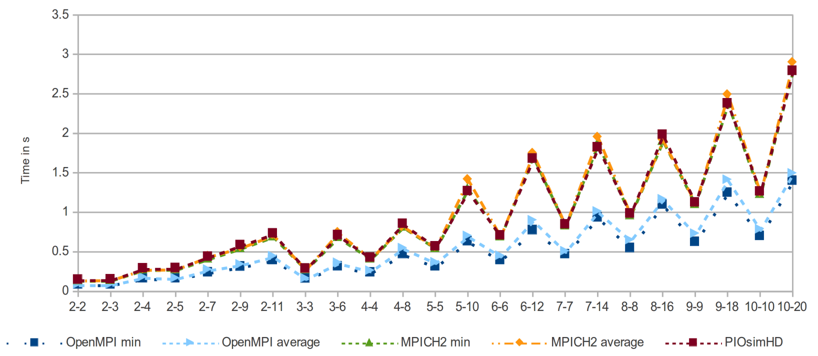

(d) Relative model accuracy – inter-node communication.

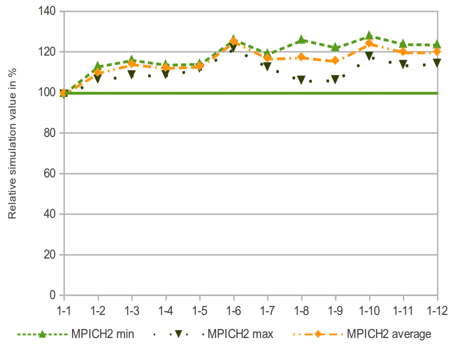
Figure 7.39.: `MPI_Gather()` and 1 MiB of data.



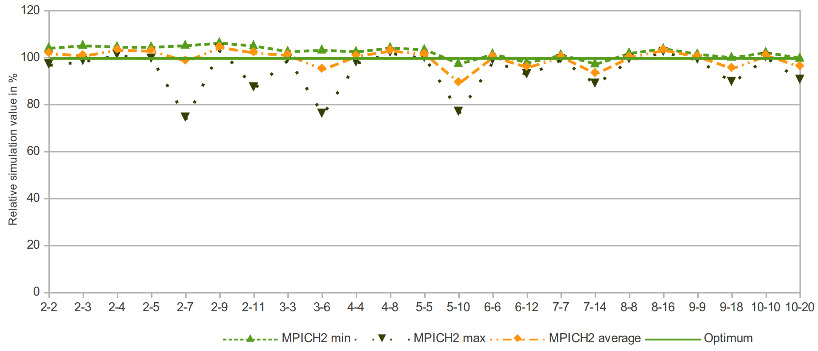
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

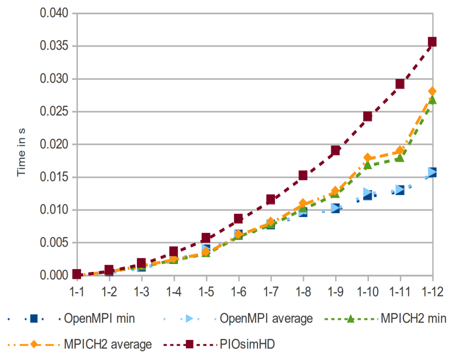


(c) Relative model accuracy – local.

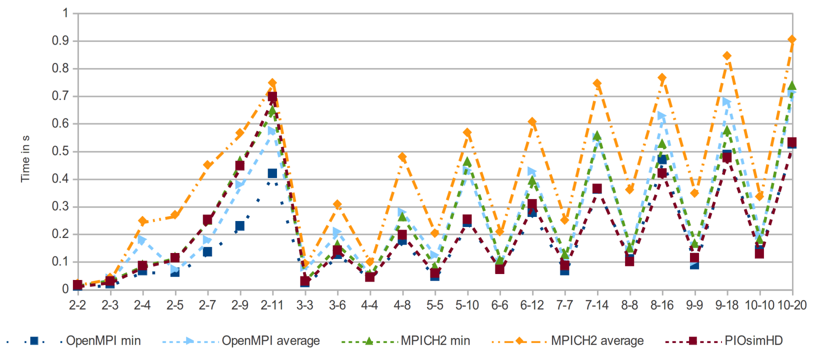


(d) Relative model accuracy – inter-node communication.

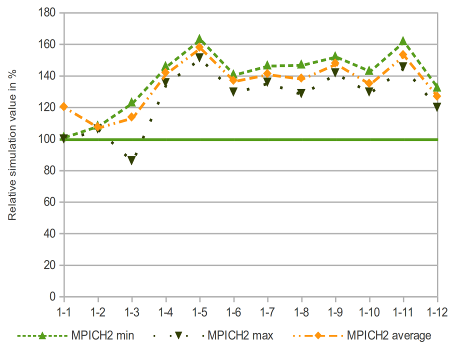
Figure 7.40.: `MPI_Gather()` and 10 MiB of data.



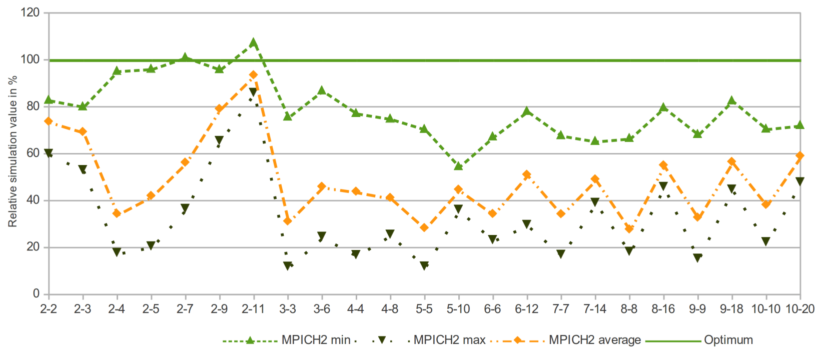
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

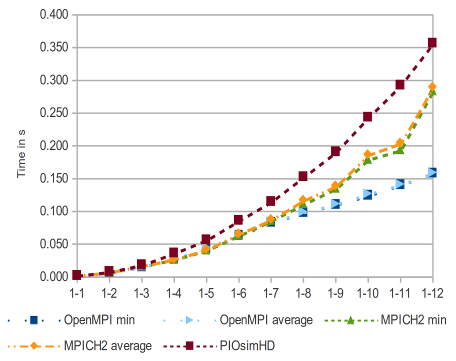


(c) Relative model accuracy – local.

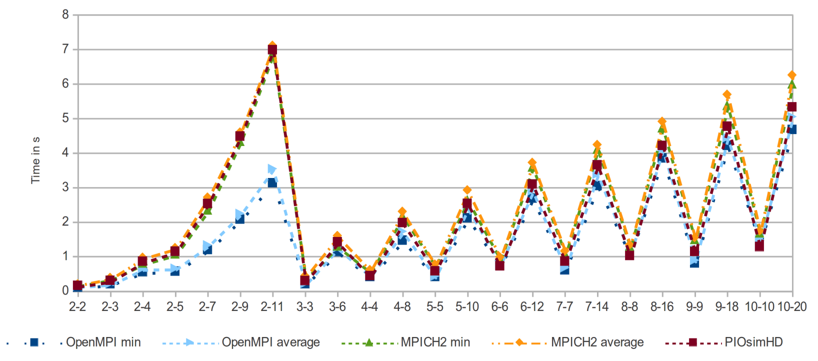


(d) Relative model accuracy – inter-node communication.

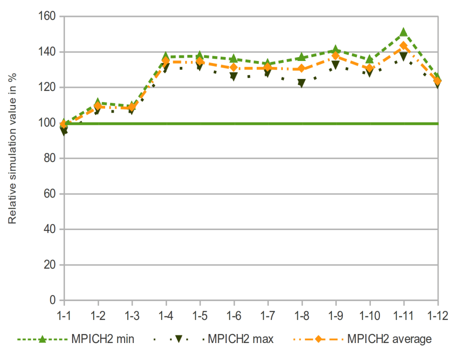
Figure 7.41.: `MPI_Allgather()` and 1 MiB of data.



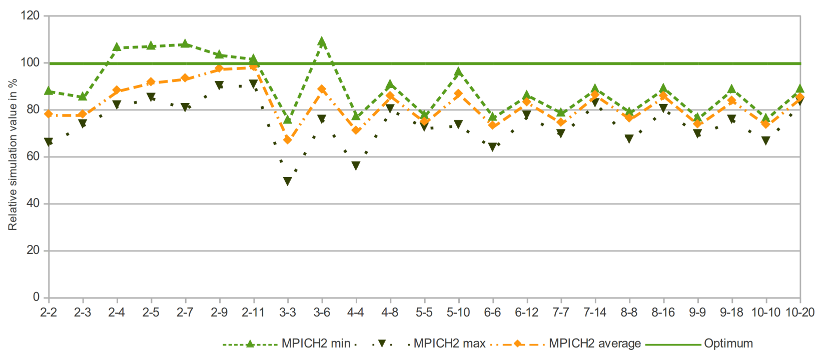
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

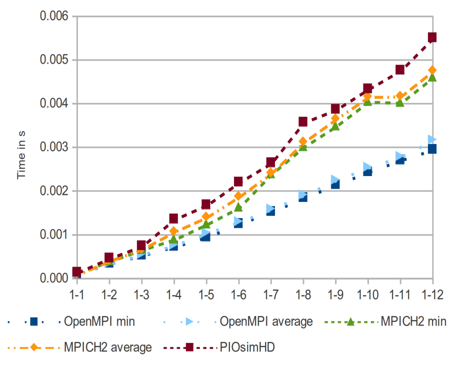


(c) Relative model accuracy – local.

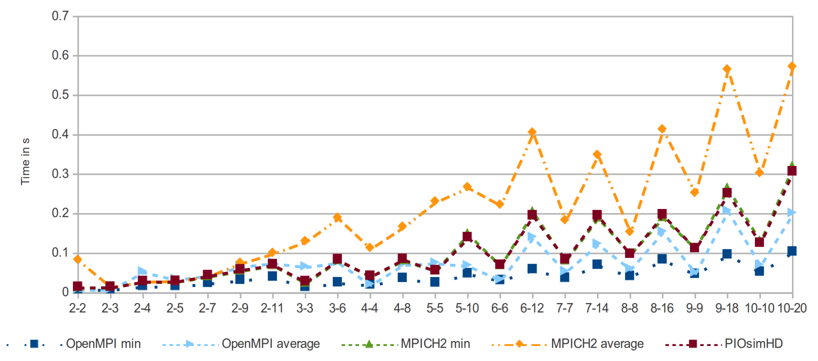


(d) Relative model accuracy – inter-node communication.

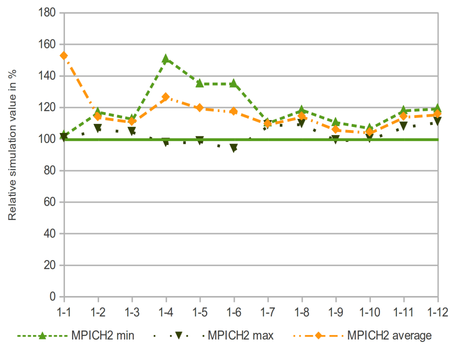
Figure 7.42.: `MPI_Allgather()` and 10 MiB of data.



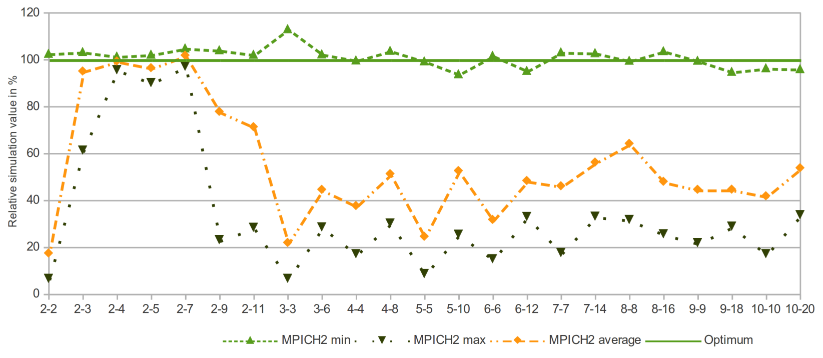
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

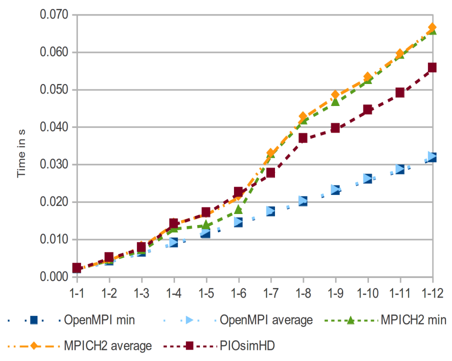


(c) Relative model accuracy – local.

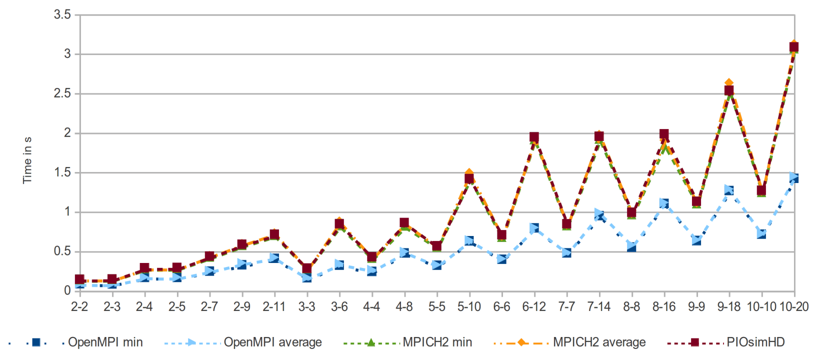


(d) Relative model accuracy – inter-node communication.

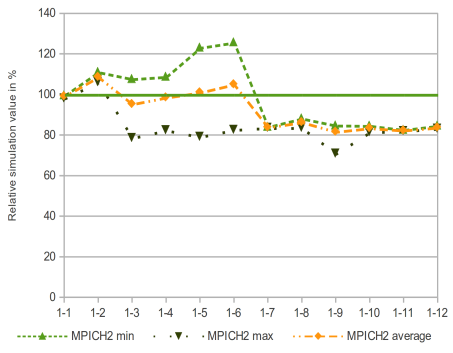
Figure 7.43.: `MPI_Scatter()` and 1 MiB of data.



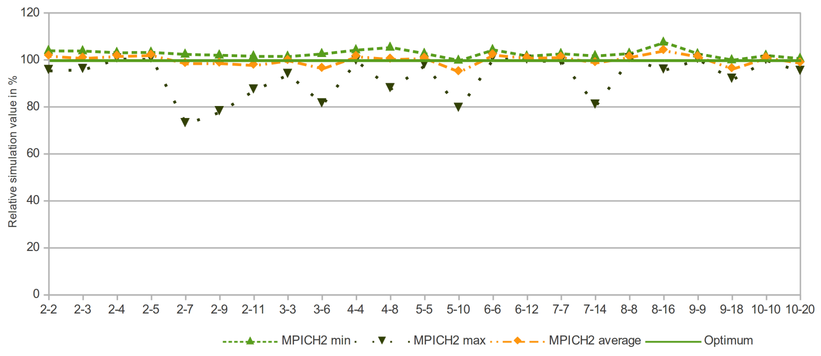
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

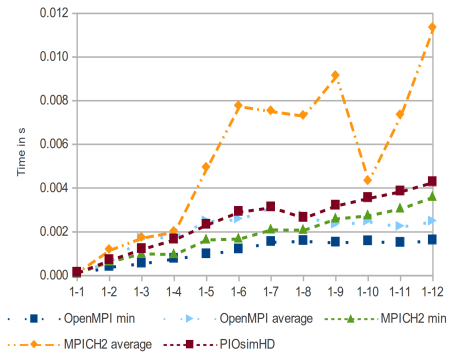


(c) Relative model accuracy – local.

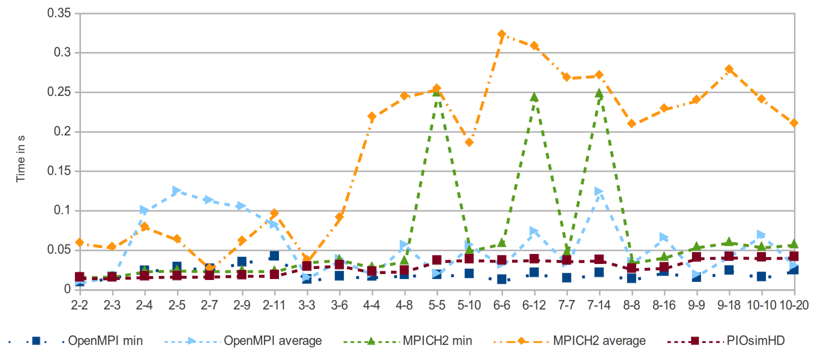


(d) Relative model accuracy – inter-node communication.

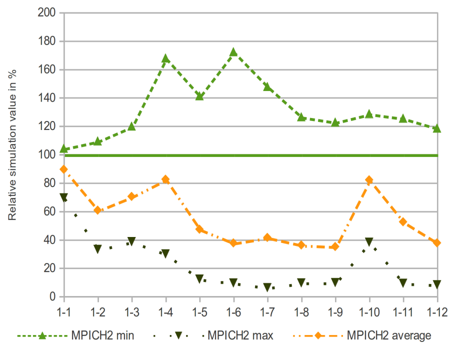
Figure 7.44.: `MPI_Scatter()` and 10 MiB of data.



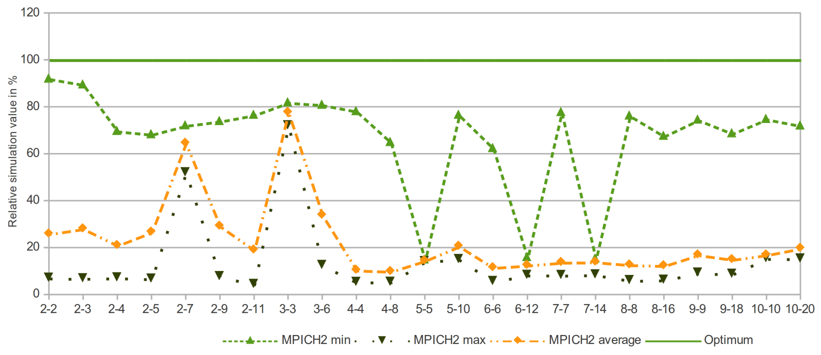
(a) Execution time for local communication.



(b) Execution time for inter-node communication.

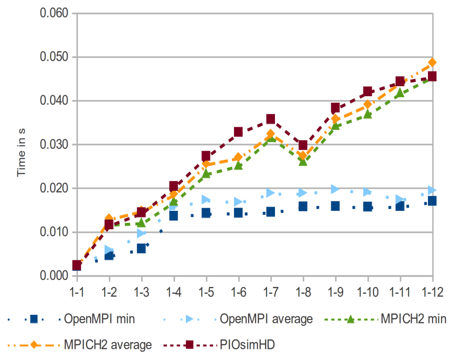


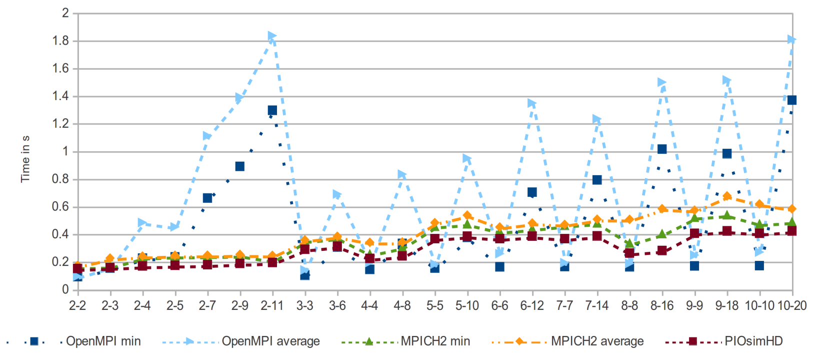
(c) Relative model accuracy – local.

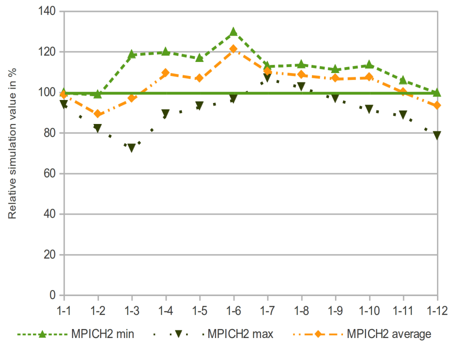


(d) Relative model accuracy – inter-node communication.

Figure 7.45.: `MPI_Reduce()` and 1 MiB of data.


(a) Execution time for local communication.


(b) Execution time for inter-node communication.


(c) Relative model accuracy – local.

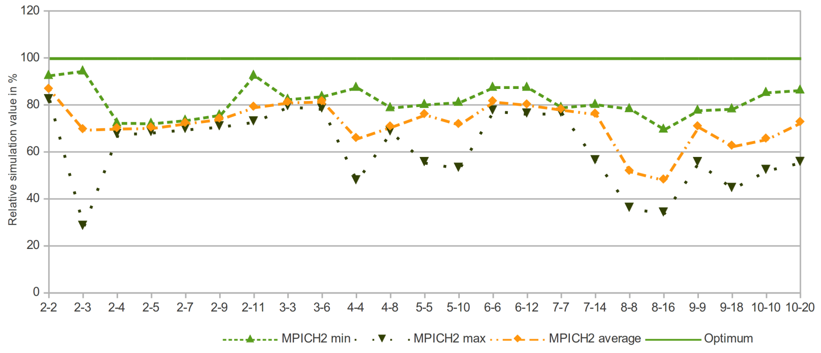
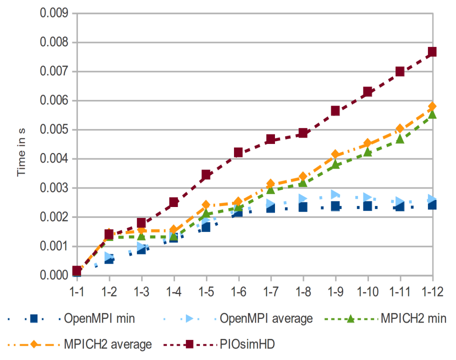
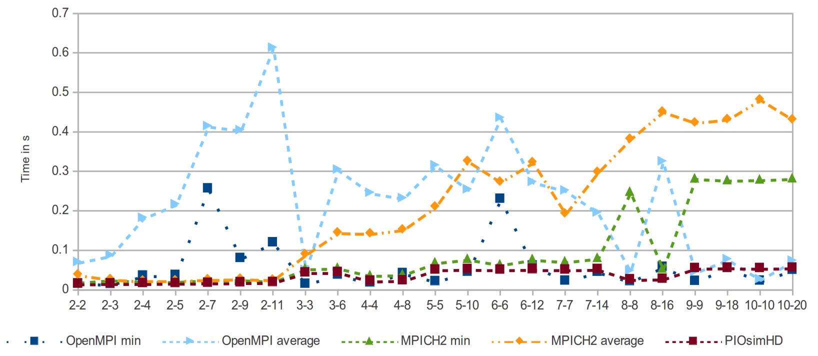
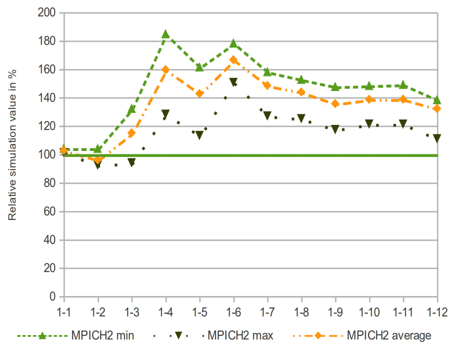

(d) Relative model accuracy – inter-node communication.

Figure 7.46.: `MPI_Reduce()` and 10 MiB of data.


(a) Execution time for local communication.


(b) Execution time for inter-node communication.


(c) Relative model accuracy – local.

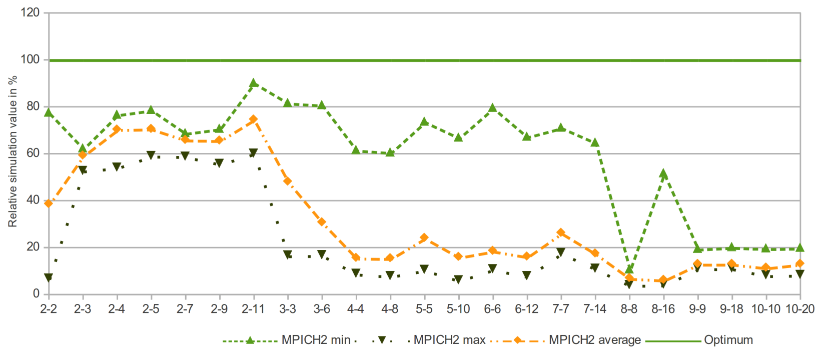
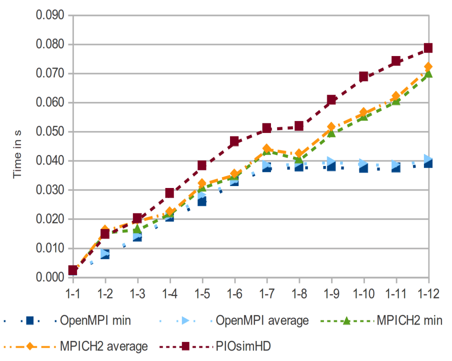
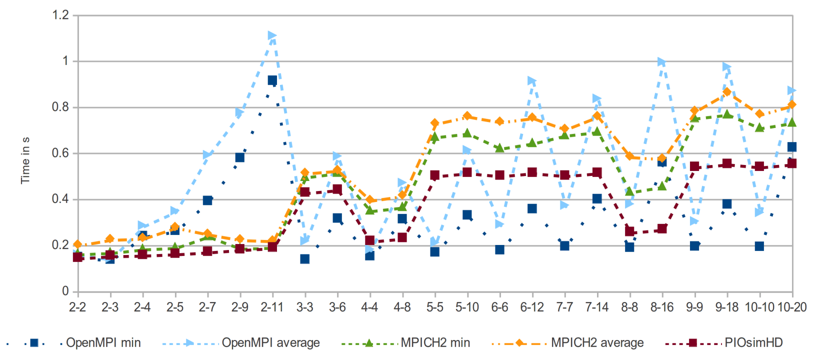

(d) Relative model accuracy – inter-node communication.

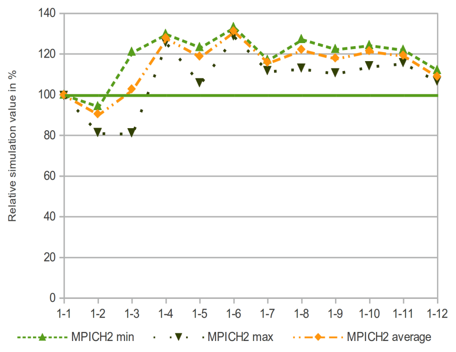
Figure 7.47.: `MPI_Allreduce()` and 1 MiB of data.



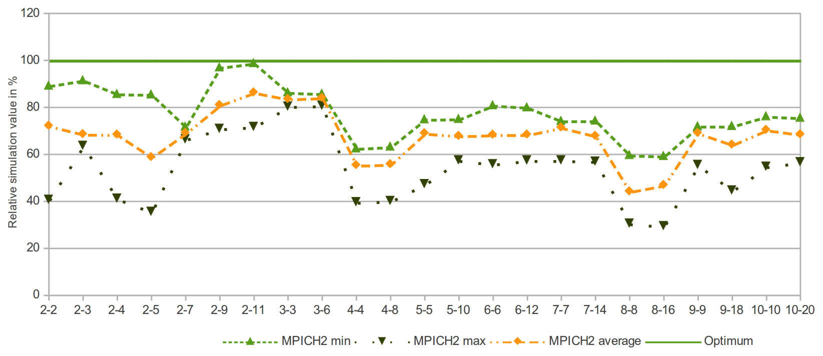
(a) Execution time for local communication.



(b) Execution time for inter-node communication.



(c) Relative model accuracy – local.



(d) Relative model accuracy – inter-node communication.

Figure 7.48.: `MPI_Allreduce()` and 10 MiB of data.

7.6.8. Summary and Conclusions

The network model is validated for a rich variety of point-to-point and collective communication patterns. For this, the MPI-internal activity is traced with HDTrace and the resulting point-to-point activity is loaded by PIOsimHD. The process mapping of the real runs are recreated in the simulation. Although it is very small, the compute time between two MPI calls is also included in the simulation. Thus, the simulator performs the identical operations as observed on the real system.

Performance is measured for 10 KiB, 1 MiB, 10 MiB, 100 MiB and 1000 MiB operations and several process configurations with 1 to 10 nodes, and up to 20 processes are analyzed. Small amounts of data are sensitive to latency and the network granularity while larger are dominated by the throughput characteristics. The simulation resembles the SendToRoot communication pattern for intra-node and inter-node communication very well because the model is parameterized based on the measurements for this pattern. Independent measurements show a high fluctuation of the times, often a factor of two is observed between minimum and third-quartile (or maximum) time even when 100 MiB of data is transferred. Therefore, for individual configurations, the number of at least 12 measurements might not suffice to capture the real minimum sufficiently well, the given minimum might still be much higher than the true minimum. Clearly, the variability is caused by the interaction between communication pattern, MPI and our system hardware. However, some patterns are rather robust and vary only slightly, especially `MPI_Scatter()` and `MPI_Gather()` are stable to variation. Overall, the simulation matches many results with a relative accuracy of about ±20% compared to the first quartile (or the minimum) of the measurement.

There are several cases in which the estimation and the measurement differ by more than 50%. Some of these are due to the discrepancy between hardware model and real system, and several are caused by bottlenecks in the system. The model for intra-node communication is far less accurate than the modeled inter-node communication, for small messages the predicted time is 4 times slower than the observed time, however, several communication schemes also match quite well (e.g. SendToRoot and `MPI_Gather()`).

Partly, this could be caused by caching effects. In general, local communication is better approximated for larger message sizes overestimating execution time by about 40%. Also, several patterns are almost perfectly estimated, for example the `MPI_Reduce()` with 10 MiB of data, here a performance drop occurs on configuration 1-8 that is also visible in the simulation. A bottleneck of the MPICH2 implementation is exposed by the SendToRoot and SendrecvRoot kernels, for some data points the expected time is much lower than the observation. In many cases, the bi-directional communication behaves just like the execution of two uni-directional communications. Another example for unexpected behavior is `MPI_Scatter()` with 10 MiB of data per process. With this communication pattern the time of configuration 1-6 is half the time for configuration 1-7. An abnormal increase is also visible in the simulation from configuration 1-7 to 1-8 which might indicate that it is due to the communication pattern. Reduce and Allreduce are predicted to have a 30% better performance than observed, further investigation is necessary to determine the reasons – presumably, additional internal buffering is done by the implementation that slows down execution.

In many cases, the simulated curves resemble the measurements well, proving that the model and the identified characteristics sufficiently describe the system. Additionally, a close match between estimate and observed performance proves that the analyzed call does not include additional overhead – it is implemented well and, according to simulation, free of not modeled bottlenecks. However, the communication algorithm can still be suboptimal, which is proven when the estimated performance matches but theoretical considerations lead to a better algorithm. It turns out that several communication patterns of MPICH2 are suboptimal. For example, `MPI_Allgather()` is just partly SMP aware; for a small message, the configuration 4-8 behaves as expected – it is slightly slower than 4-4, but configuration 5-10 takes longer than 10-10. Another suboptimal algorithm is `MPI_Reduce()`, its time does not monotonically increase with the number of processes, instead some configurations with fewer processes take longer. With large messages, `MPI_Allgather()` is slower in configuration 2-9 than with 10-10. Due to the estimation done by simulation, the algorithmic suboptimality can clearly be distinguished from unexpected bottlenecks inside library and system.

For local communication, the overhead of HDTrace is investigated because intercepting MPI calls takes time that is accounted to the computation time of MPI. By comparing the results with and without executing recorded compute jobs, the time for many local communication patterns improved but not for `MPI_Bcast()`. An in-depth analysis shows that a bottleneck is predicted in the interconnection of the two sockets, the connection is already busy when computation jobs are simulated. Therefore, performance cannot be improved significantly by removing waiting times – thus the bottleneck manifests also in the run without computation. Although the performance of the real system is better than that of the modeled system, the simulation still demonstrates expected system behavior for some communication patterns.

Open MPI is also benchmarked for large messages. In many cases, it shows a different behavior than MPICH2, even in local communication times. To a large extent this is caused by different algorithms realized in Open MPI, and to the fact that Open MPI tries to utilize both network interfaces. There are some interesting patterns, for example in local communication Open MPI scales better – for several communication patterns the run-time increases almost linear up to 4 to 6 processes and then increases just slightly with an increasing number of processes. This is surprising as it is also visible for larger message exchanges, e.g., for the ring-communication of 100 MiB of data and `MPI_Reduce()`. Presumably, the implemented optimizations utilize the CPU cache more efficiently.

7.7. Evaluation of Parallel I/O

Similar to collective operations, this evaluation validates the simulation model; also it allows identification of unexpected behavior and fosters a discussion about the potential discrepancy between simulation and observation. To validate performance of parallel I/O, the MPI benchmark `parabench`[MRKL10] is used to execute I/O patterns which are then compared with simulation results.

Many different experiments have been executed for this thesis, but due to space limitations, a specific subset has been chosen for presentation and further assessment. The methodology used to measure system performance, the conducted experiments and the approach of comparing measurements with simulation results are described in Section 7.7.1. Estimated and observed performance are discussed based on the following classes of experiments: Accessing data on tmpfs checks whether the abstract client/server communication model is correct (see Section 7.7.2). The performance of accessing cached data with two different cache sizes is assessed in Section 7.7.3. Furthermore, results for different amounts of free memory and for tmpfs are compared.

An experiment with a variable number of clients on 5 nodes is presented in Section 7.7.4. At last, an experiment in with clients and servers are placed on the same nodes is shown in Section 7.7.5. The simulator generates several patters which occur in the real system, and allows analysis beyond the possibilities in vivo. A discussion of emerging complex patterns is given in Section 7.7.6.

7.7.1. Methodology

Conducted experiments The following orthogonal parameters show the diversity of I/O validation experiments conducted[20]:

- Access pattern: All processes access a single shared file sequentially in several iterations. Either a client performs 10 iterations with 100 MiB records (a total of 1 GiB), 10240 iterations with 100 KiB records (a total of 1 GiB), or 100 iterations with 100 MiB records (a total of 10 GiB). Records of the file are partitioned round-robin among the clients, that means that Rank 0 accesses the first record, Rank 1 the second record and so on.
- Level of access: The access pattern is also influenced by the level of access: contiguous vs. non-contiguous I/O and collective vs. independent I/O. These two parameters are orthogonal and the benchmark supports all four patterns: independent contiguous (Ind-C), collective contiguous (Coll-C), independent non-contiguous (Ind-NC) and collective non-contiguous (Coll-NC).
- Storage backend: PVFS is configured to put its storage space either on tmpfs or on the local hard disk drive.
- Memory limitation: Without restricting memory about 10 GiB of free memory is available for caching. To assess the impact of caching, memory is limited to either 400 MiB or 1 GiB by using the `mem-eater` program (see Section A.4 for further information).
- Configuration of client and server processes: Several classes of configurations are possible on our cluster system; the number of clients and servers and the actual placement can vary. To test scalability, PVFS clients and servers are placed on disjoint nodes, that means every process is placed on its own node. In this configuration, a variable but identical number of servers and clients is placed (between 1 to 5 each). Since these configurations do not necessarily lead to a bottleneck in the network, one additional configuration with 3 clients and 2 servers is evaluated. This configuration enforces network congestion on the servers' network links.

  To test concurrent access, 5 server nodes provide storage for an increasing process count located on 5 client nodes. In the last type of experiment, client and server processes are placed on the same nodes, and thus clients and servers use overlapping nodes.
- System configuration: On the real system, many parameters are adjustable. For simplicity, the default values of our Ubuntu servers are kept and not varied. Therefore, PVFS uses a flow buffer size of 256 KiB – data is accessed on disk with that granularity and network transport and I/O are performed concurrently[21]. The *deadline* I/O scheduler is used. The maximum transmission unit of our Ethernet connections is 1500 bytes. MPICH2 offers parameters to control I/O optimizations: By default the

[20]There are several simulation parameters that can be changed, the configuration is part of the experiment – these parameters are described in a minute.

[21]Refer to Section 2.1.4 for a description of the I/O protocol.

two-phase protocol of MPICH2 uses a collective buffer of 16 MiB and data sieving uses a read buffer of 4 MiB.

Measuring performance For a single measurement, a client and server configuration, the memory limitation and the storage backend are chosen. Then parabench is started to execute all levels of access sequentially. For each level of access, data is written and then read. Between tests processes are synchronized with a barrier and the file is deleted after it has been read.

The definition of the described access scheme is given in the PBL language [MRKL10] in Listing 7.4. The access patterns that precede the PBL description are shown for 100 MiB records and for 100 KiB records in Listing 7.6 and Listing 7.5, respectively. An example trace illustrates the temporal pattern – for a configuration with three clients and two servers, the client-sided trace is given in Figure 7.51a.

Listing 7.4: Parabench description to measure the performance of all 4 levels of access

```
$env = "pvfs2:/pvfs2"; # Setup a global variable

# Time all commands in the block and label the timer "MPI-IO␣test".
time["MPI-IO␣test"] {
        # Time an individual execution of the pattern with the name pattern0.
        # (independent, contiguous I/O)
        time["pwrite-lvl0"] pwrite("$env/file.dat", "pattern0");
        # Synchronize all processes with an MPI_Barrier();
        barrier;

        time["pread-lvl0"] pread("$env/file.dat", "pattern0", "world");
        barrier;
        # Delete the file.
        pdelete("$env/file.dat");
        barrier;

        # Repeat the sequence with pattern1 (collective, contiguous I/O).
        time["pwrite-lvl1"] pwrite("$env/file.dat", "pattern1");
        barrier;

        time["pread-lvl1"] pread("$env/file.dat", "pattern1", "world");
        barrier;
        pdelete("$env/file.dat");
        barrier;

        # Independent, non-contiguous I/O
        time["pwrite-lvl2"] pwrite("$env/file.dat", "pattern2");
        barrier;

        time["pread-lvl2"] pread("$env/file.dat", "pattern2", "world");
        barrier;
        pdelete("$env/file.dat");
        barrier;

        # Collective, non-contiguous I/O
        time["pwrite-lvl3"] pwrite("$env/file.dat", "pattern3");
        barrier;

        time["pread-lvl3"] pread("$env/file.dat", "pattern3", "world");
        barrier;
        pdelete("$env/file.dat");
        barrier;

}
```

Listing 7.5: Parabench description of accessing 1 GiB of data with 100 KiB records

```
# A pattern is defined by a name, the interleaving type (always 2),
# the number of iterations, the record size per iteration and the access level.
# Independent, contiguous (level 0):
define pattern {"pattern0", 2, (10 * 1024), (100 * 1024), 0};
# Collective, contiguous:
define pattern {"pattern1", 2, (10 * 1024), (100 * 1024), 1};
# Independent, non-contiguous:
define pattern {"pattern2", 2, (10 * 1024), (100 * 1024), 2};
# Collective, non-contiguous:
define pattern {"pattern3", 2, (10 * 1024), (100 * 1024), 3};
```

Listing 7.6: Parabench description of accessing 1 GiB of data with 10 accesses of 100 MiB records

```
define pattern {"pattern0", 2, 10, (100 * 1024 * 1024), 0};
define pattern {"pattern1", 2, 10, (100 * 1024 * 1024), 1};
define pattern {"pattern2", 2, 10, (100 * 1024 * 1024), 2};
define pattern {"pattern3", 2, 10, (100 * 1024 * 1024), 3};
```

Simulation model Because all patterns are rather simple, they are encoded directly in PIOsimHD. The system model for our cluster is used. Server nodes are supplied with the same amount of free memory as in the measurement. However, several parameters define the simulation behavior, for example, cache layer and network granularity. The simulator offers multiple caching algorithms, if not mentioned explicitly, then the AggregationReorderCache is used. For message transport, a transfer granularity of 100 KiB is chosen. However, this granularity impacts network performance due to the store-and-forward switching. Therefore, for 100 KiB records, an additional simulation with an access granularity of 10 KiB is performed.

To simulate collective I/O, the model for the original two-phase-protocol is used. Note that the simulator offers a variant of two-phase as well (refer to [Kuh09] for further information). Data sieving that is applied for independent, non-contiguous I/O on the real system is not simulated.

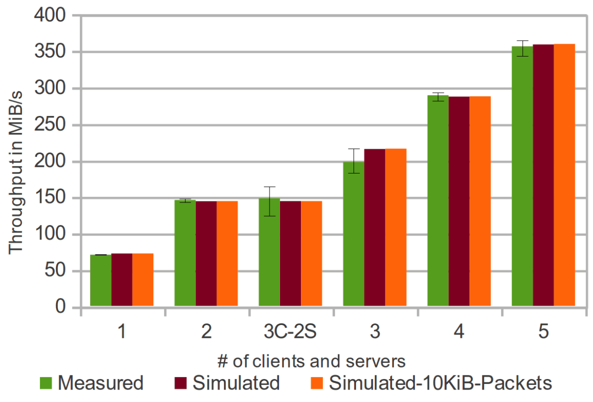
Comparing observation and simulation A single measurement is repeated three times, which allows evaluation of deviation. Generated diagrams provide the average value and error bars for the minimum and maximum. Sometimes execution was not successful because execution on clients stalls while accessing data. Those experiments have been restarted several times to achieve a successful run. However, there are reproducible crashes for a few experiments, for example when 100 KiB records are accessed from multiple clients per node. Therefore, these values could not be measured and these missing values are omitted from the diagrams.

In all cases, client activity is recorded with HDTrace to allow a later comparison with the simulation results. Additionally, a few traces are generated in which PVFS server activity is recorded. However, since tracing of the servers slows down activity slightly, those measurements are not shown, but they have been used to identify issues in performance differences.

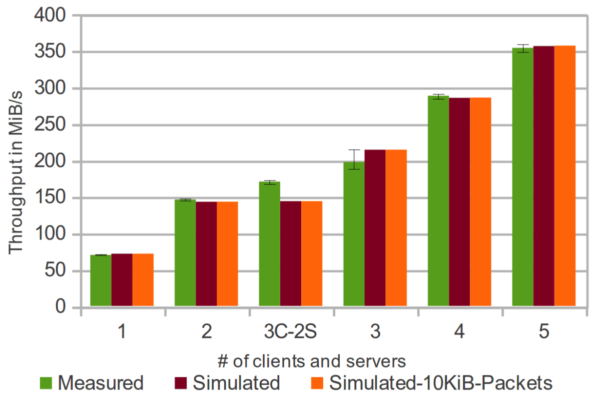
With PIOsimHD, the performance is estimated (simulated) and compared with the measurements. Due to the slight difference in the client and server communication performance, and the improved scheduling of PIOsimHD, it is not expected that simulation results match perfectly.

7.7.2. Accessing Data on tmpfs

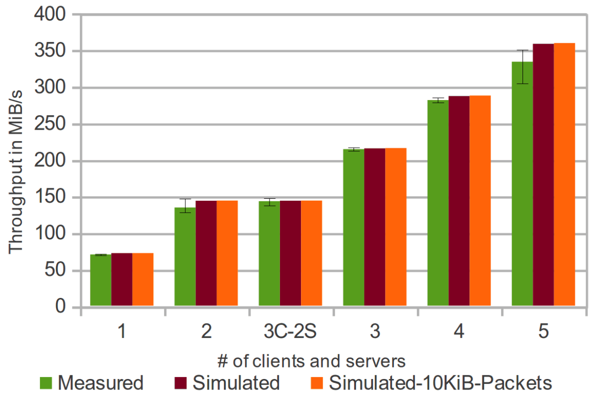
In this validation experiment, the network model of parallel I/O is compared with measured activity. By putting the data on a *tmpfs*, the influence of the I/O subsystem is ignored because data on tmpfs can be accessed with 2,000 MiB/s, according to the characterization in Section 3.4.2.



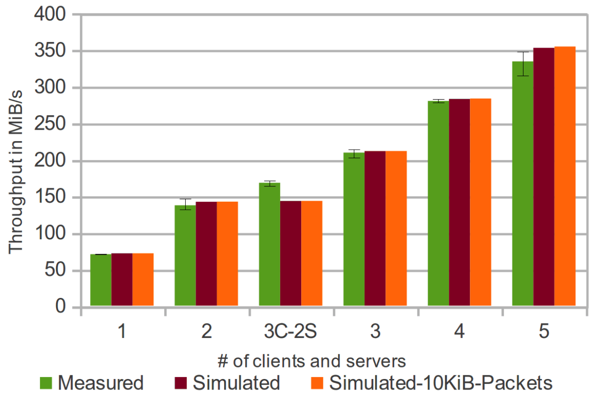
(a) Write independent, contiguous.



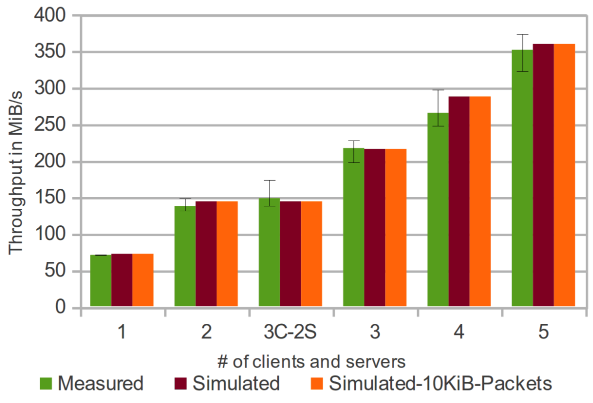
(b) Read independent, contiguous.



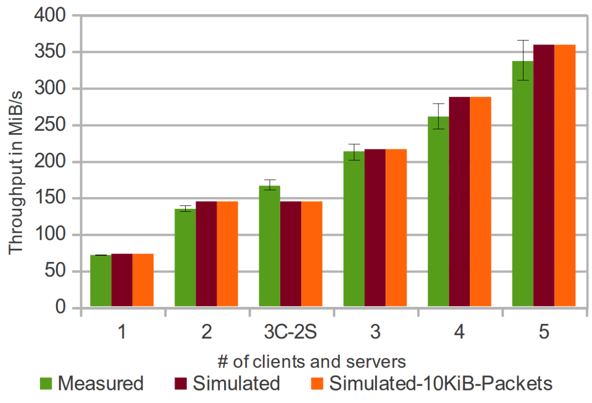
(c) Write collective, contiguous.



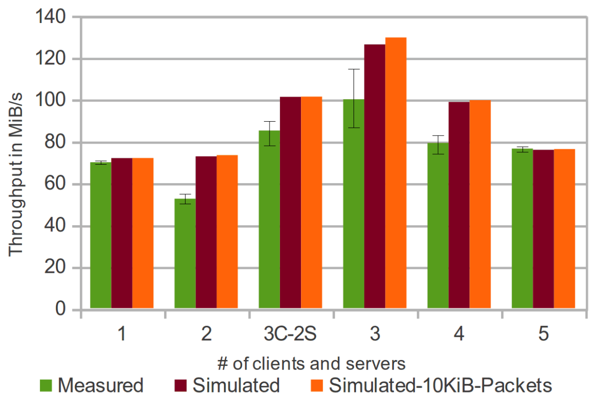
(d) Read collective, contiguous.



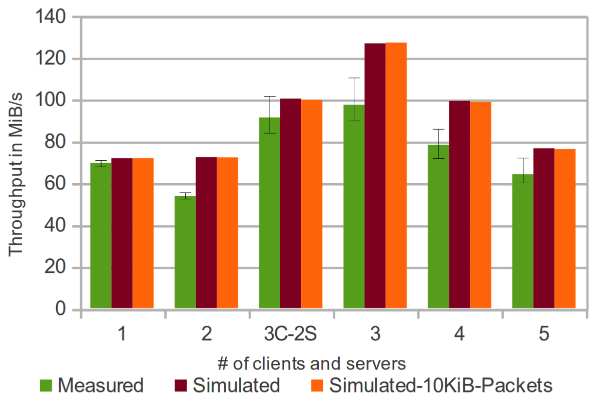
(e) Write independent, non-contiguous.



(f) Read independent, non-contiguous.

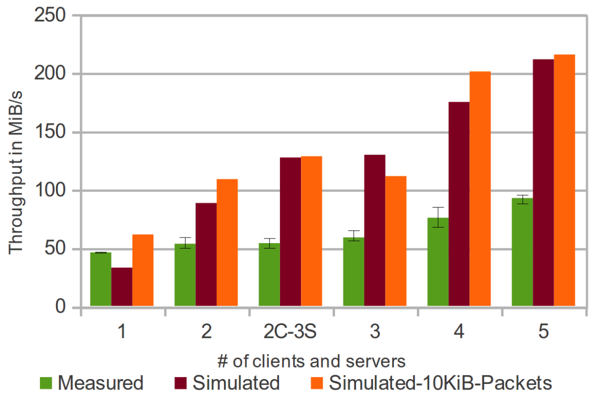


(g) Write collective, non-contiguous.

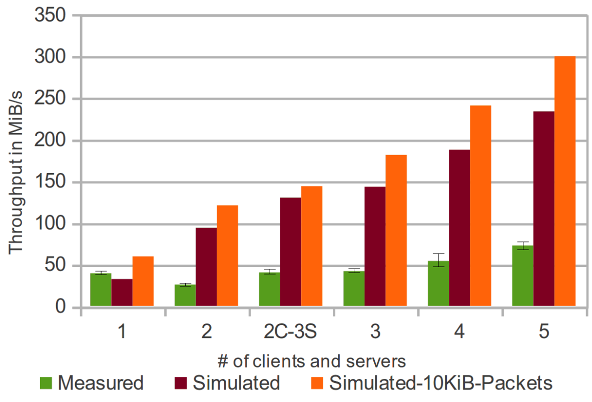


(h) Read collective, non-contiguous.

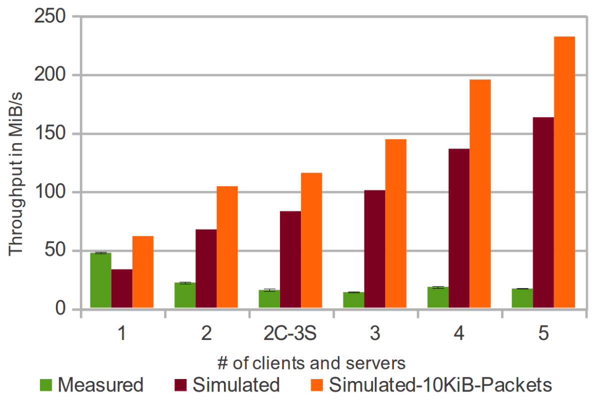
Figure 7.49.: Performance of accessing data on tmpfs by using 100 MiB records – configuration with a variable number of disjoint client and server nodes.



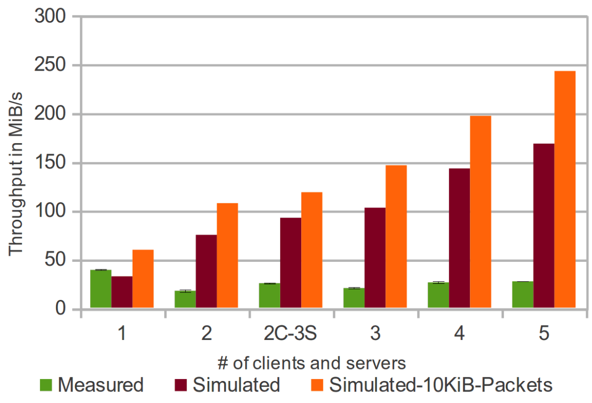
(a) Write independent, contiguous.



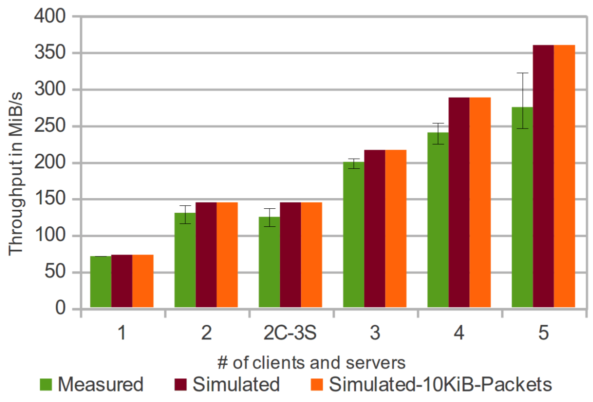
(b) Read independent, contiguous.



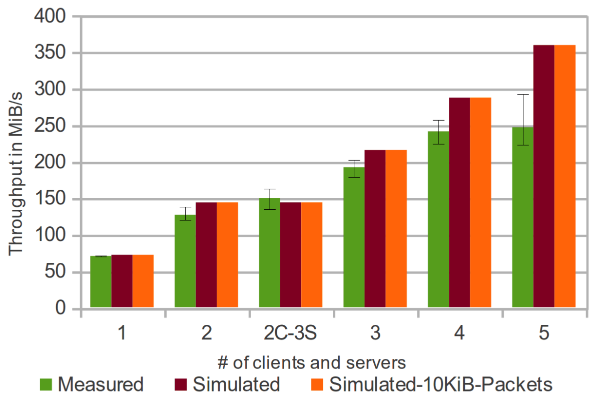
(c) Write collective, contiguous.



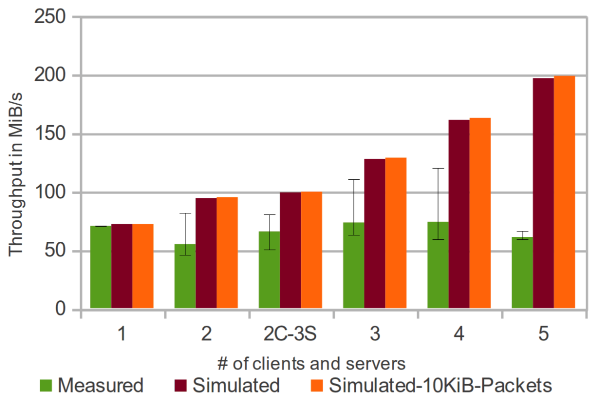
(d) Read collective, contiguous.



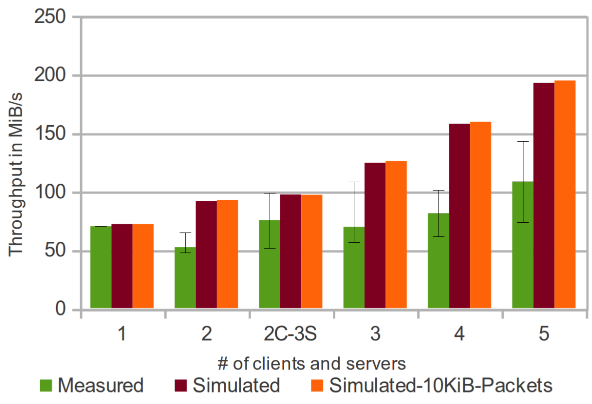
(e) Write independent, non-contiguous.



(f) Read independent, non-contiguous.



(g) Write collective, non-contiguous.



(h) Read collective, non-contiguous.

Figure 7.50.: Performance of accessing data on tmpfs by using 100 KiB records.

For simulation, the disk model is adjusted to a sequential transfer rate of 2,000 MiB/s and without latency. To assess the impact of the transfer granularity on results, the default granularity of 100 KiB and a granularity of 10 KiB are compared for both record sizes.

The performance for accessing 100 MiB records and for accessing 100 KiB records is shown in Figure 7.49 and in Figure 7.50, respectively. Further, an overview of the client-side processing of the benchmark is visualized in Figure 7.51.

Observations and interpretation

a) First, the results with 100 MiB records are assessed (see Figure 7.49). In many configurations, the observed minimum and maximum differ by about 10% from the average value. With an increasing number of clients and servers, the performance increases; about 70 MiB/s of data can be accessed per client and server except for collective non-contiguous access. Note that 70 MiB/s is the observed performance limit of our TCP/IP network connection. Interestingly, the configuration *3C–2S* has a slightly better performance of 170 MiB/s which is about 85 MiB/s per server[22]. The simulator estimates performance very well in all cases. Especially the fastest observation and the simulation are alike.

In a few cases, the measured performance is slightly better than the simulation results. Mostly it happens for independent and non-contiguous I/O, see Figure 7.49e and Figure 7.49f).

b) Collective non-contiguous access is much lower; in the best case, an aggregated performance of about 120 MiB/s is achieved. Read and write achieve almost identical performance values. Performance drops from 70 MiB/s to 50 MiB/s when using two instead of one client and server. PIOsimHD shows a similar performance behavior. However, it slightly overestimates performance by at most 20 MiB/s or roughly 20% (compared to the average observation). This is due to the improved scheduling of the simulated parallel file system in PIOsimHD. In general, the low performance is due to the two-phase protocol.

c) The change to the smaller access granularity has no noticeable effect for 100 MiB records; only the performance for three clients and servers increases slightly for collective and non-contiguous writes (see Figure 7.49g).

d) Next, look at the results for 100 KiB records in Figure 7.50. In the measured performance, the deviation between minimum and maximum is low for contiguous access but increases to +-25% for non-contiguous access.

e) For contiguous I/O, PIOsimHD predicts an almost linear scaling with the number of servers (see Figure 7.50). While the simulation matches the expectations for a parallel file system, the measured performance stays behind – the measurement of independent writes shows a slow increase from 50 MiB/s to 100 MiB/s, for collective writes performance is reduced and stays on a low level.

Although the model matches expectations for a parallel file system at this point, it seems that the simulation and the real system differ too much. An explanation could be a wrong model for client/server communication or systematic mistakes in modeling the system or application behavior. Another explanation for degraded performance in the real system could be the variance of network and I/O performance. However, with the discussion of cached I/O in Section 7.7.3 it will be shown that those kind of effects can be seen in the simulation as well: Minimal timing effects can cause long-term delays and congestions in the network which slow down the whole processing.

f) With the exception of non-contiguous access, choosing a network granularity of 10 KiB increases performance. This behavior is expected: For independent I/O, the network granularity of 100 KiB

[22]Basically, one might claim that this result suggests that multiple streams perform better; however, with a growth in servers, the number of concurrent data streams increases as well but does not improve over the 70 MiB/s per server. Presumably, network performance is limited from client-side, although it is unclear how.

fragments a single I/O into just two packets, thus the store-and-forward switching increases communication time. This is especially true for a single client. Data is exchanged in collective I/O between the clients; therefore, the overhead is even higher in this case. When multiple clients are connected, the latency is not so important any more, therefore, the experiments with configuration *2C–3S* show less improvement with a smaller granularity.

In non-contiguous I/O all data is transferred with a single server request. Consequently, in this case, a lower network granularity does not change the situation much.

g) Non-contiguous I/O offers much better performance on the real system; also, the model estimates the observed throughput well. With the simulation, the quality of the four levels of access can be predicted; for five servers and clients, the four levels of access achieve roughly 220 MiB/s, 220 MiB/s, 350 MiB/s and 200 MiB/s, respectively. Thus, on our system, the additional network communication of collective I/O in combination with the modified access pattern degrades performance.

h) The trace of the client activity also indicates that collective I/O is a bottleneck for both configurations – non-contiguous collectives are slower for large records, as are contiguous collectives for small records (refer to Figure 7.51).

By inspecting the traces, many dynamic effects can be observed. One example is the difference in processing speed of the clients when independent I/O is performed. To visualize this activity a zoom into independent-contiguous I/O is given in the screenshots.

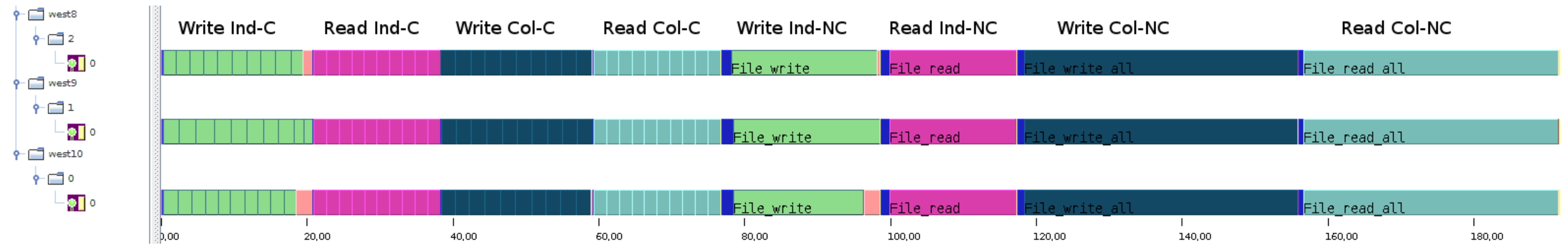
Another effect is the aggregation of requests on a single server: When multiple clients access data located on one server, it might be a bit slower. Since the server has to multiplex its speed among all requests, subsequent operations could queue up. Therefore, a single server might be overloaded. A further discussion of emerging patterns is provided in Section 7.7.6.

i) An interesting simulation result emerges for the configuration with three servers and clients each (see Figure 7.50a). By writing contiguous blocks independently with a transfer granularity of 10 KiB, the aggregated throughput is reduced from 132 MiB/s to 111 MiB/s.

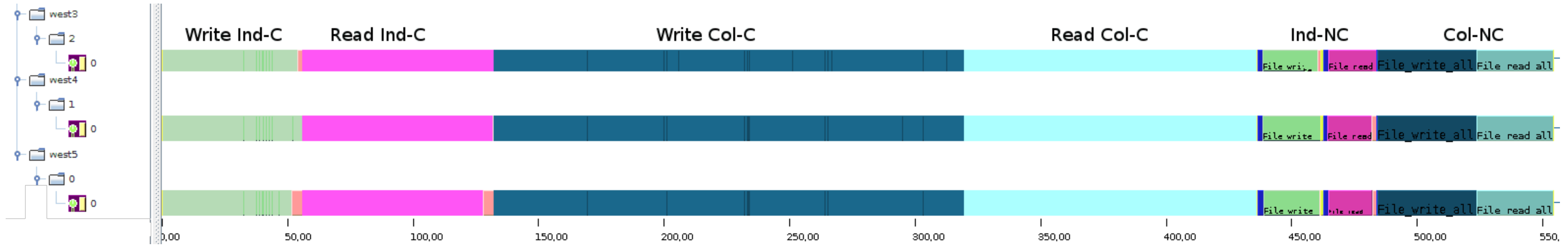
To assess the simulation results better, traces and profiles for both network granularities are generated and shown in Figure 7.52. Profiles show the inclusive time; a run takes about 23 seconds. In the profile, all three clients need almost the same time to perform the file write (orange colored entries next to the clients). However, with 100 KiB network granularity, the second client needs more time to receive data from the second server. Due to the fast I/O device almost no time is spent to perform I/O (the write activity next to the I/O-subsystems is close to 0). Surprisingly, in all cases the first server needs less time to process requests, although the amount of data written per server is identical.

An excerpt of the timeline for 100 KiB granularity is given in Figure 7.52c. The figure shows an excerpt of client and server activity; on the client-side, every job which receives data from a particular server is encoded by another color. In Figure 7.52d, the first few activities are shown for 10 KiB granularity. In this figure, the requests from the client and server side are visualized, too. Further, the network activity on Ethernet edges to the clients is provided: Every combination of client and server is drawn in its own color.

By looking at the timelines, a complex communication pattern can be observed; this is caused by the data partitioning (round-robin partitioning with chunks of 64 KiB). In Figure 7.52c, it becomes apparent that the first server processes requests of the clients sequentially, while the second and third process requests in a concurrent fashion. Since the NIC multiplexes all requests, it takes longer for these servers to transfer data back to the clients. With a smaller network granularity the pattern changes, now the first server needs almost as much time as the other servers (compare the profiles). While all clients need the same amount of time, the total aggregated processing time on the servers increases from 41 s to 45 s. Thereby, the concurrency on the servers increases, and although a smaller network granularity is used, all requests take longer due to the shared network resource. This leads to the performance degradation for 10 KiB access granularity. By comparing the speed of the individual

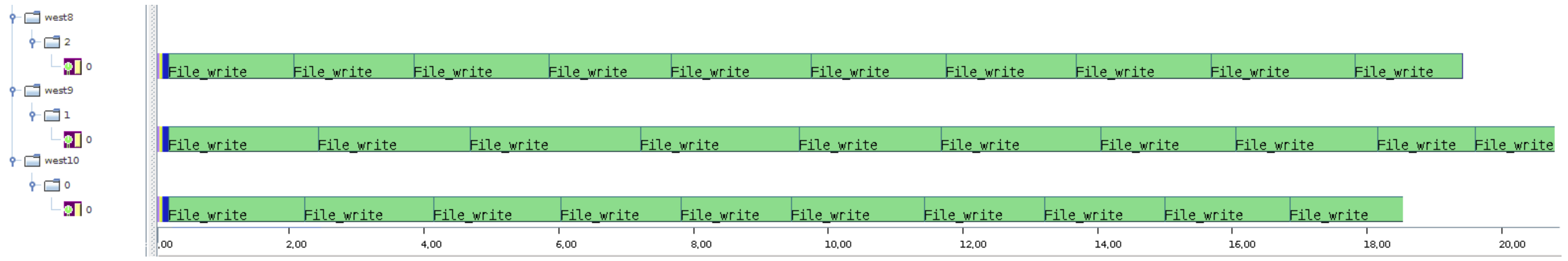


(a) Record size of 100 MiB.

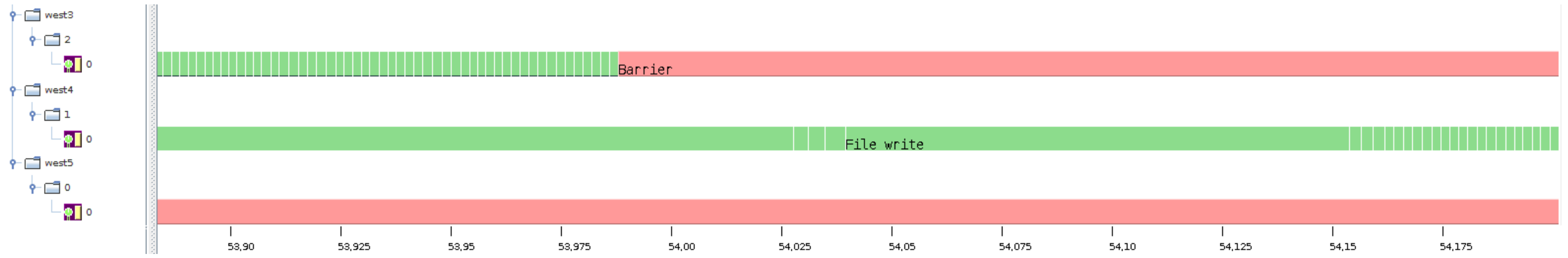


(b) Record size of 100 KiB.

Figure 7.51.: Client-sided trace for three clients accessing data on two servers – data is stored on tmpfs. The barrier between runs is colored in orange and the MPI_File_set_view() is colored in blue.

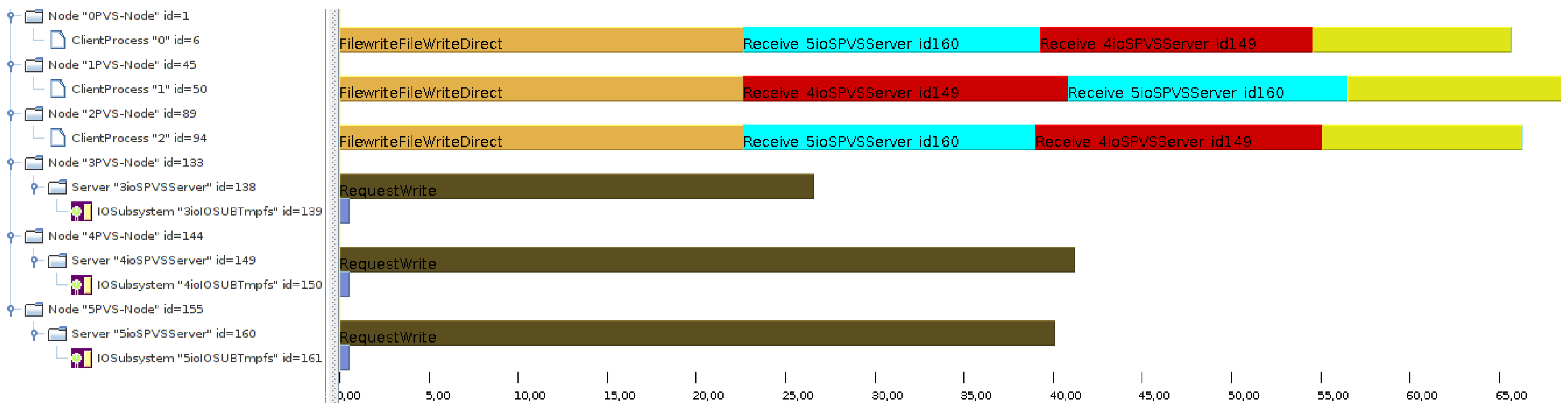


(c) Record size of 100 MiB – zoom into the independent contiguous I/O phase.

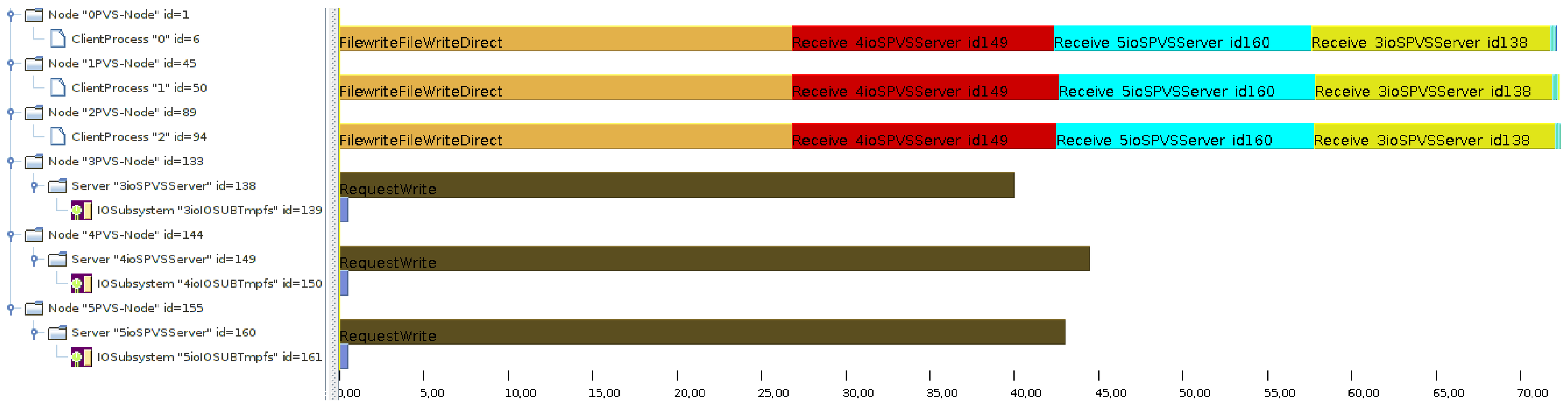


(d) Record size of 100 KiB – zoom into the end of the independent contiguous I/O phase.

Figure 7.51.: Client-sided trace for three clients accessing data on two servers – data is stored on tmpfs. The barrier between runs is colored in orange and the `MPI_File_set_view()` is colored in blue.

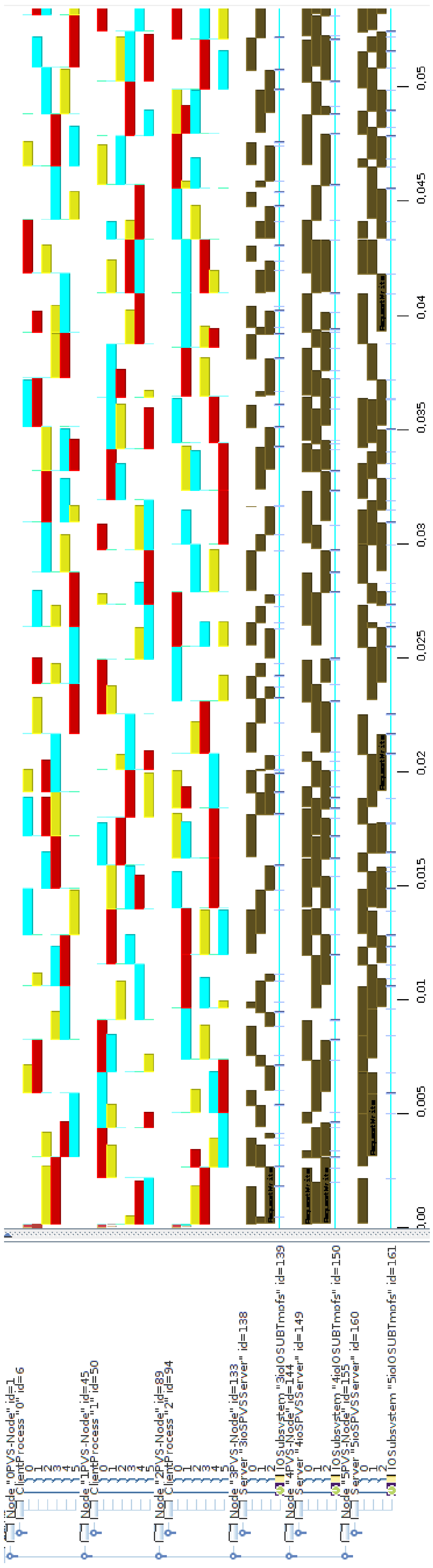


(a) Profile for a network granularity of 100 KiB showing the inclusive time.



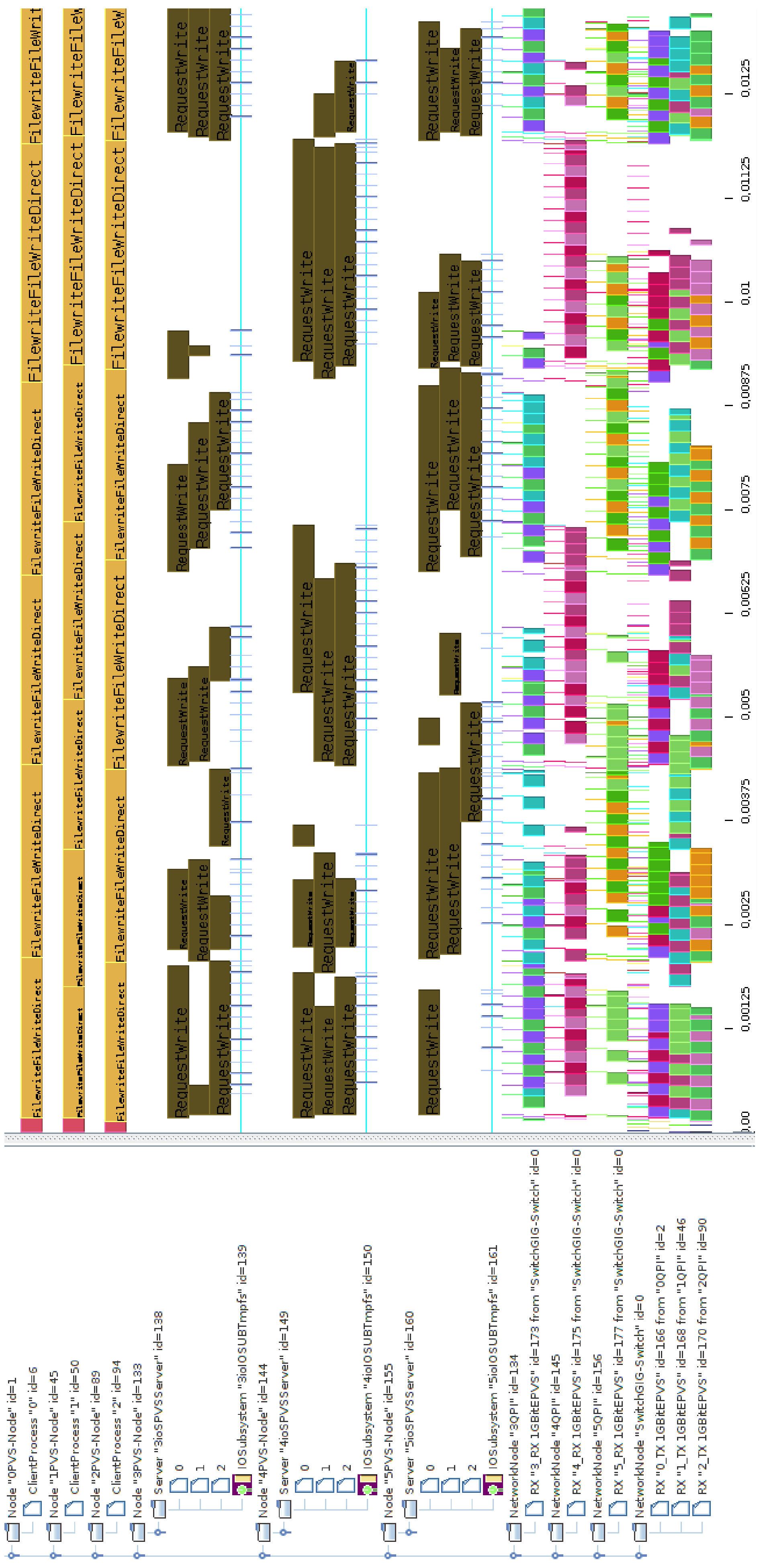
(b) Profile for a network granularity of 10 KiB showing the inclusive time.

Figure 7.52.: Simulation of the processing for three clients accessing data on three servers – data is stored on tmpfs.



(c) Excerpt of the timelines for a network granularity of 100 KiB.

Figure 7.52.: Simulation of the processing for three clients accessing data on three servers – data is stored on tmpfs.


Node "0PVS-Node" id=1
ClientProcess "0" id=6
Node "1PVS-Node" id=45
ClientProcess "1" id=50
Node "2PVS-Node" id=89
ClientProcess "2" id=94
Node "3PVS-Node" id=133
Server "3ioSPVSServer" id=138
IOSubsystem "3ioIOSUBTmpfs" id=139
Node "4PVS-Node" id=144
Server "4ioSPVSServer" id=149
IOSubsystem "4ioIOSUBTmpfs" id=150
Node "5PVS-Node" id=155
Server "5ioSPVSServer" id=160
IOSubsystem "5ioIOSUBTmpfs" id=161
NetworkNode "3QPI" id=134
RX "3_RX 1GBitEPVS" id=173 from "SwitchGIG-Switch" id=0
NetworkNode "4QPI" id=145
RX "4_RX 1GBitEPVS" id=175 from "SwitchGIG-Switch" id=0
NetworkNode "5QPI" id=156
RX "5_RX 1GBitEPVS" id=177 from "SwitchGIG-Switch" id=0
NetworkNode "SwitchGIG-Switch" id=0
RX "0_TX 1GBitEPVS" id=166 from "0QPI" id=2
RX "1_TX 1GBitEPVS" id=168 from "1QPI" id=46
RX "2_TX 1GBitEPVS" id=170 from "2QPI" id=90
FilewriteFileWriteDirect
RequestWrite
0,00
0,00125
0,0025
0,00375
0,005
0,00625
0,0075
0,00875
0,01
0,01125
0,0125

(d) Excerpt of the timelines including network activity for a network granularity of 10 KiB.

Figure 7.52.: Simulation of the processing for three clients accessing data on three servers – data is stored on tmpfs.

clients it can be seen that the second client completes 5 requests while the other clients process 4 requests (look at Figure 7.52d).

As can be seen already, this simple access patterns leads to complex behavior, which, in turn, causes a variation in server load. A modification of the network granularity changes timings and thus can lead to a variation in behavior. This is realistic: Such parameter sensitivity has been reported for real systems [Kun07]. Additional examples for insights into server behavior with the help of simulation and a discussion of this issue is given in Section 7.7.6.

7.7.3. Cached Data

To assess the impact of caching, I/O is conducted on the local disk and the benchmark is run with a variable amount of free memory. Originally, it was intended to measure performance for a cache size of 400 MiB, 1 GiB and 10 GiB, and to compare the results of the hard disk drive with tmpfs. However, PVFS crashed reproducibly for 100 KiB accesses. Since the times of 1 GiB and 10 GiB are identical for accessing records with 100 MiB size, those values are omitted. Due to the influence of network granularity, simulation for 100 KiB accesses rely on a granularity of 10 KiB. To evaluate the influence of the access granularity, 100 KiB accesses with 1 GiB of memory are also simulated with a 100 KiB granularity. The results are shown in Figure 7.53 and in Figure 7.54 for a record size of 100 MiB and 100 KiB, respectively.

Observations and interpretation

a) In the measurement conducted with MPICH2 and PVFS, the observable performance increases with additional free memory, also performance of the disk I/O is below the one of tmpfs. This is intuitively expected. A few runs are slightly faster with less memory available; presumably, this is caused by a variation in the timing of operations.

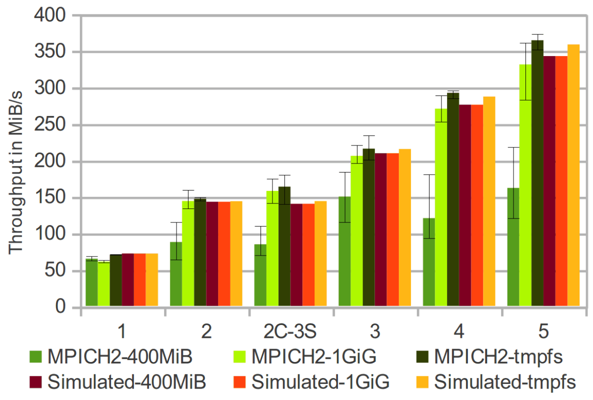
In general, simulation provides an upper bound to throughput, or it is close to the observed performance. There are some exceptions to this observation, which are discussed in detail in the following.

b) In several diagrams the read performance from disk is faster than the one estimated by PIOsimHD (look at Figure 7.53d, Figure 7.54b and Figure 7.54h). The reason is the page cache of the Linux kernel: Data is written shortly before it is read and Linux caches as much I/O as possible. Since about 1 GiB of data is written per client, almost all data is still available during the read phase of the benchmark. Consequently, the performance of measured reads can be on a similar level as the measurements conducted on tmpfs. For a 400 MiB cache which is benchmarked for 100 MiB records, this effect is not so big but still visible.

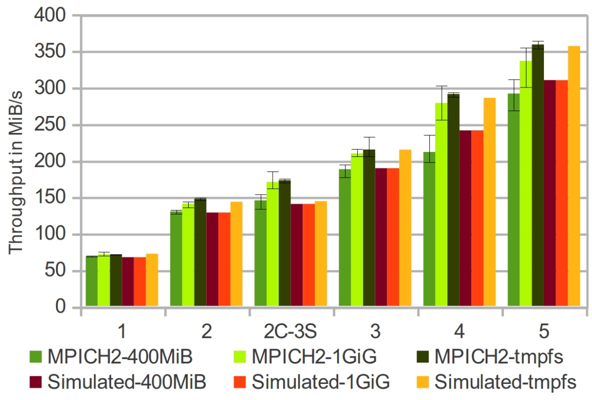
The simulator does not keep data in a cache, instead it fetches data from the disk drive model. Thus, read performance is not influenced by the memory size in the experiments and might be below the measurement. However, measured read performance with a smaller cache size is on a similar level as the estimated performance. It is questionable if HPC algorithms read data which has been written recently – typically, with an appropriate design such a pattern can be avoided.

c) There are a few cases in which the measurement outperforms simulated estimations. Mainly in the *2C–3C* configuration for 100 MiB records a better performance of up to 200 MiB/s can be observed – amounting to about 100 MiB/s per client (look at Figure 7.53e). This is higher than expected; the benchmarked network throughput of a single data stream between two nodes is 71 MiB/s.

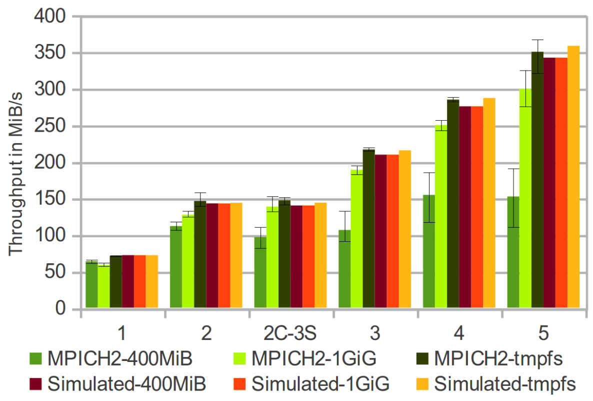
With the system model used in the simulation, performance is limited. Therefore, the observed discrepancy is inherent to the parameters of the model. That effect also causes a higher maximum measured performance for collective I/O and 100 KiB records than predicted by PIOsimHD even for tmpfs; for instance, for two clients and three servers a maximum read performance of 130 MiB is observable, while the simulation roughly estimates 100 MiB/s (see Figure 7.54h). A simple theoretic consideration confirms this theory of too fast communication: On average, both clients must forward



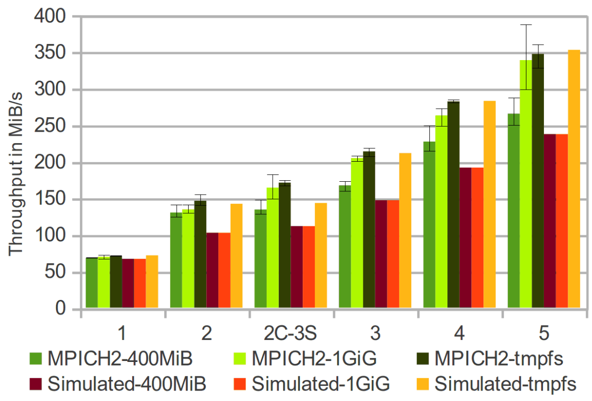
(a) Write independent, contiguous.



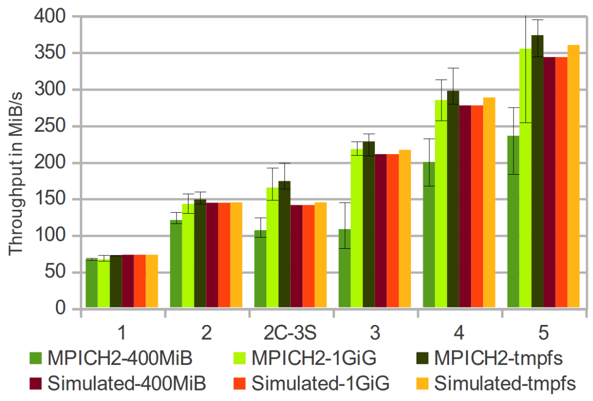
(b) Read independent, contiguous.



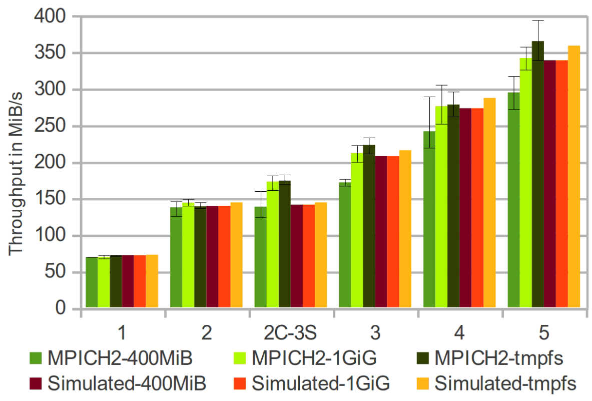
(c) Write collective, contiguous.



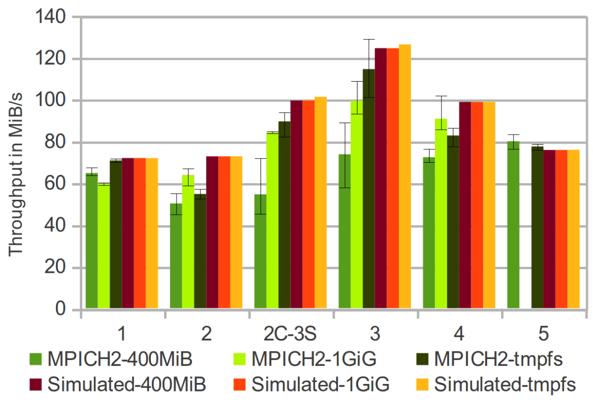
(d) Read collective, contiguous.



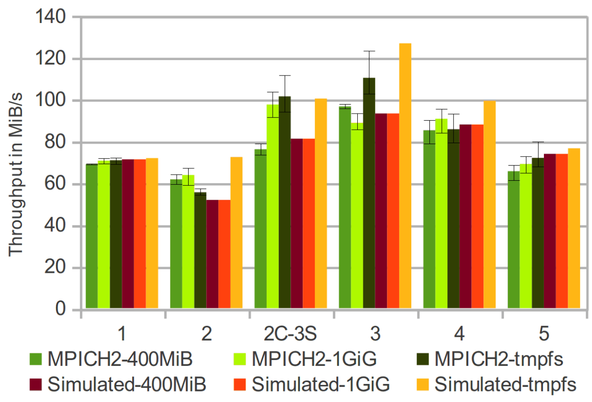
(e) Write independent, non-contiguous.



(f) Read independent, non-contiguous.

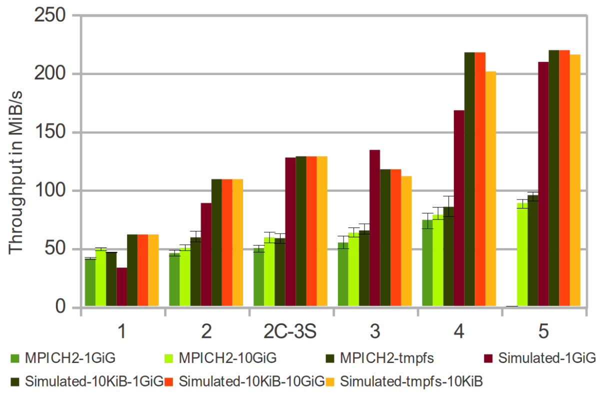


(g) Write collective, non-contiguous.

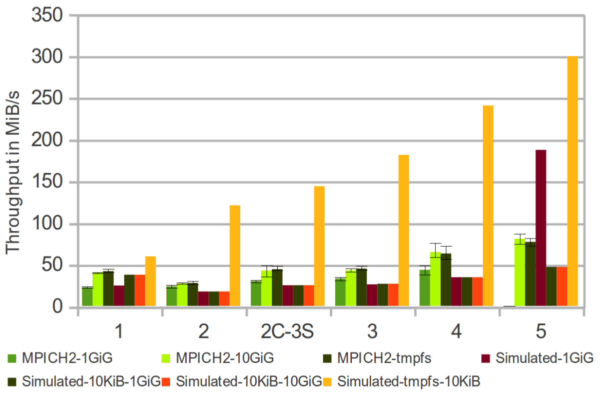


(h) Read collective, non-contiguous.

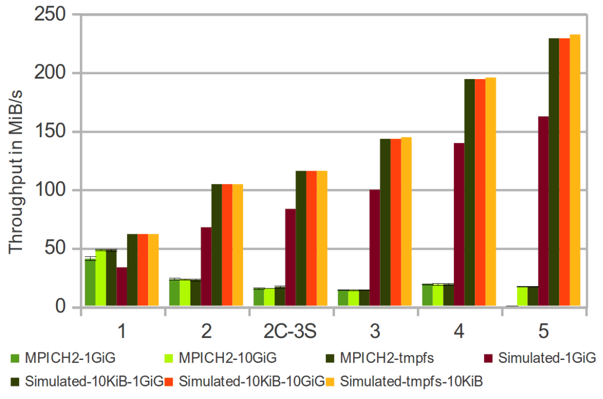
Figure 7.53.: Performance of accessing data by using 100 MiB records. The amount of available memory for disk I/O is suffixed in the legend.



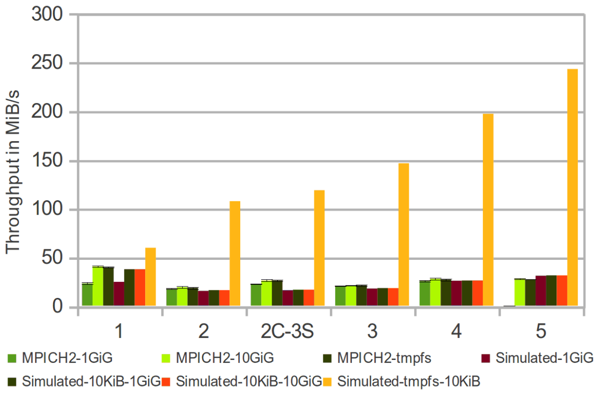
(a) Write independent, contiguous.



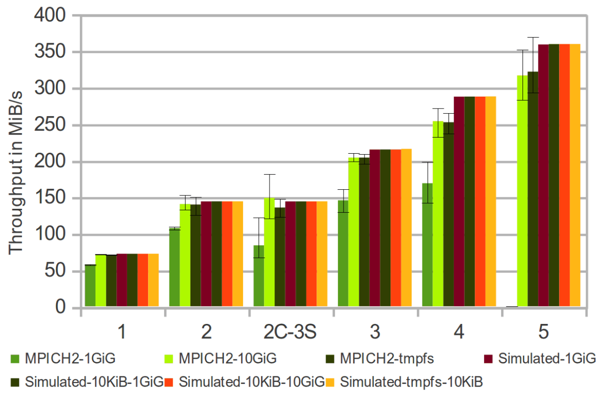
(b) Read independent, contiguous.



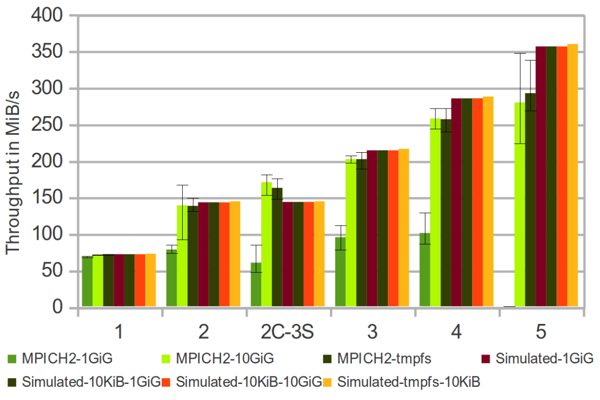
(c) Write collective, contiguous.



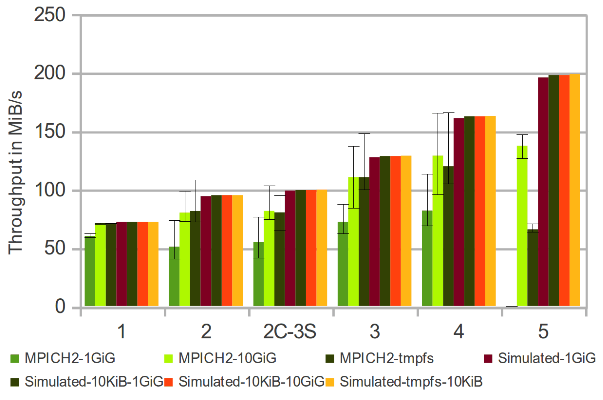
(d) Read collective, contiguous.



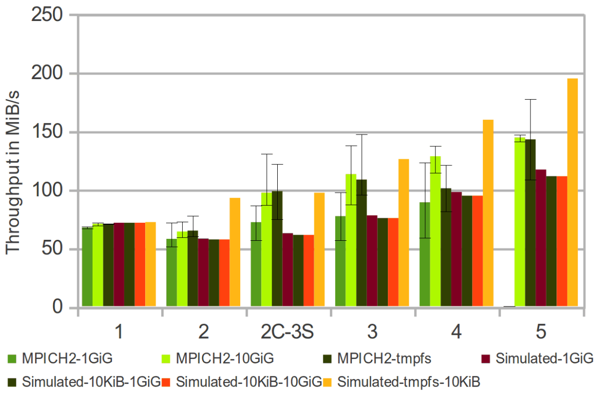
(e) Write independent, non-contiguous.



(f) Read independent, non-contiguous.



(g) Write collective, non-contiguous.



(h) Read collective, non-contiguous.

Figure 7.54.: Performance of accessing data by using 100 KiB records. The amount of available memory for disk I/O is suffixed in the legend.

50% of data to the other process. Since reading of data and data exchange are not pipelined, they are processed sequentially. Thus, the NIC which achieved 71 MiB/s throughput must transport 1.5 bytes per byte of data read from disk. Therefore, an effective throughput of about 47 MiB/s is possible leading to an aggregated performance of 95 MiB/s. Consequently, the network communication works better in the experimental run than during the characterization of the system.

Therefore, it seems that the network limitation of our cluster is partly resolved when congestions happen only on sender or receiver side. Since every client contacts every server, there are already multiple streams in that case. Thus, as expected, the 71 MiB/s restriction does not look like a hardware specific network limitation. Instead it seems like a software (or kernel) issue but the exact reason could not be resolved, yet.

d) The simulation of the I/O-subsystem achieves a write performance similar to the simulation with tmpfs. This is due to the write-behind caching strategy and the performed data aggregation.

Reads are not cached in the simulation. And since they must be requested from the disk drive, the operation must wait for the slower drive. Consequently, estimated read performance of the disk drive model must be lower than the one on tmpfs. For large (or non-contiguous) accesses, the modeled scheduler can optimize access very well leading to an almost optimal throughput.

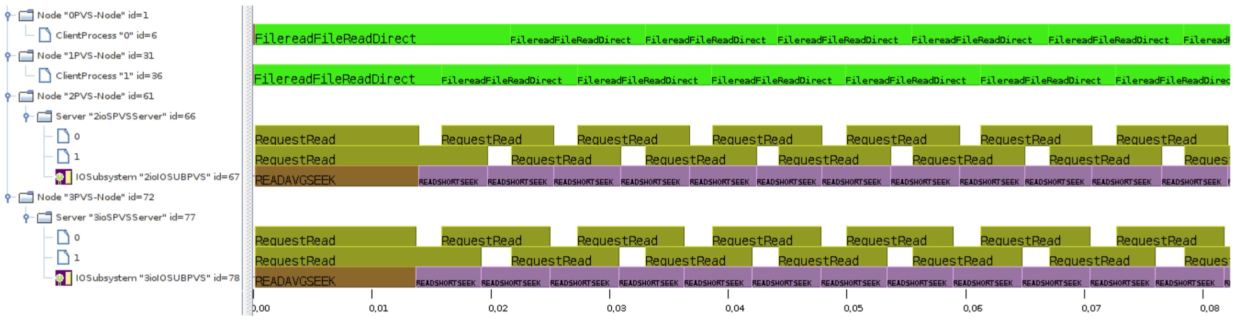
e) When 100 KiB records are read, the observed performance is very low, even for tmpfs (see Figure 7.54b). This has been mentioned for the measurements on tmpfs already. However, now the slow performance becomes visible for simulation of the disk drive; even when assuming 10 GiB of main memory, performance of the simulated tmpfs is much faster than the simulated disk drive. Interestingly, when using a larger network granularity, the simulator predicts a good performance for the run with five clients and servers (roughly 190 MiB/s).

To assess this behavior, a screenshots of the simulation traces are provided for two and five processes in Figure 7.55. With two processes, I/O scheduling does not help: The read request that arrives first is dispatched, in the meantime the second request arrives and is processed shortly after. Since disk and network speed are similar to I/O transfer rate, a pattern emerges in which the disk must skip the 100 KiB region read from the other process. An alternative to this seek-dominated pattern would be that the file is read sequentially; with a slightly different timing, the read pattern could look like sequential access.

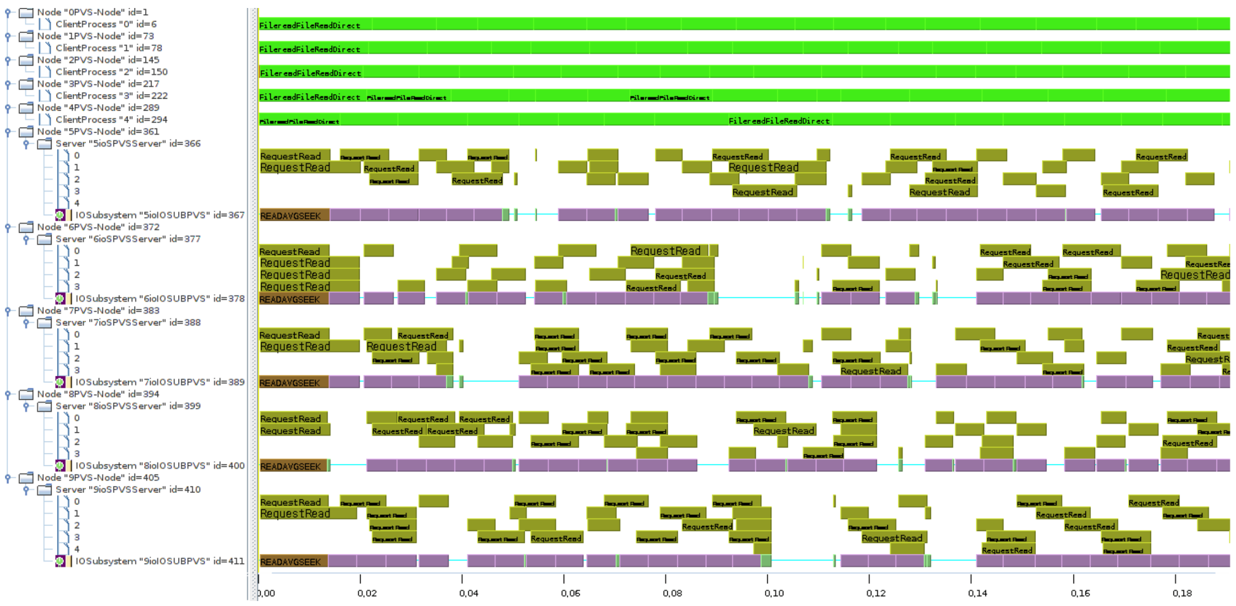
The pattern for five clients is already more complicated (look at Figure 7.55b). Although the load is balanced, idle times on individual servers show up. It can be also seen that the duration for individual client I/O operations varies, some clients proceed faster than others. During the processing, it happens that multiple operations are queued on a single server. The scheduler of the server can then aggregate these operations and read data faster; after some time, a sequential pattern emerges (see Figure 7.55c). This sequential read improves throughput and eliminates idle phases and, consequently, aggregated performance increases.

However, since low performance is observed for MPICH2 on tmpfs as well, slow disk I/O cannot be the only explanation. Due to network congestion, even accessing data on a fast storage system could cause imbalance on the servers (this has been seen for tmpfs in Figure 7.52). As reported before, the reason is that a server multiplexes its NIC among all connected clients, thus performance of the NIC is shared. Once requests of multiple clients are pending on a single server, all clients must wait for data stored on this server. Since the server multiplexes the NIC, data transfers of all clients take longer. This problem does not arises for collective I/O because in this mode another operation can be started only if the current operation has been completed. Another example of this issue is provided in Section 7.7.6

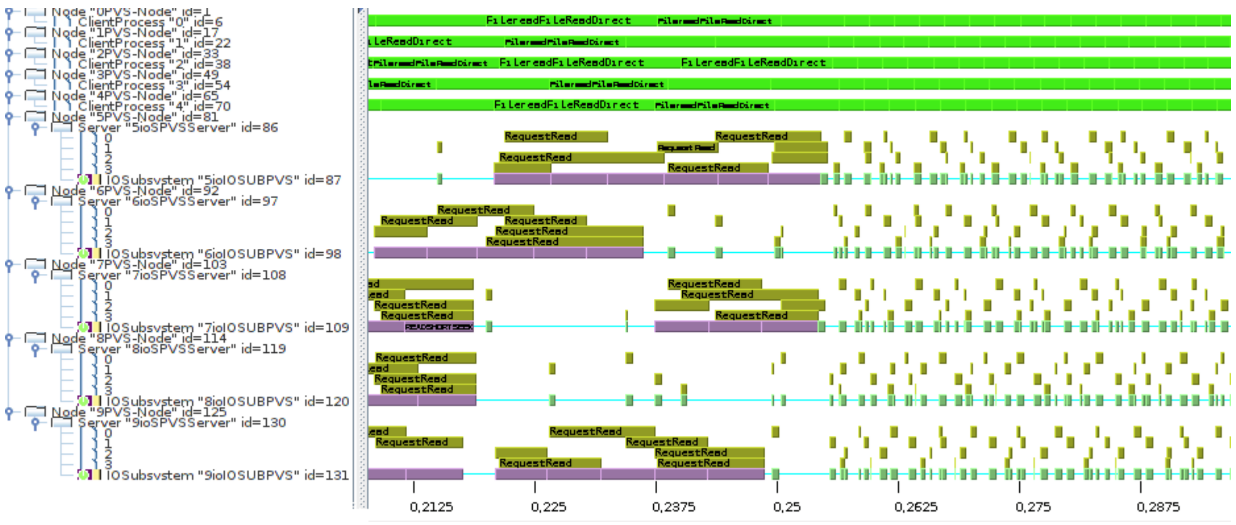
f) In the simulation, it looks like that the amount of free memory does not influence the observable performance – results for 400 MiB, 1 GiB and 10 GiB of memory behave alike. However, in the measurements a noticeable performance increase shows up. The reasons are the simulated caching algorithms. The write scheduler dispatches operations efficiently on the disk model: in the simula-



(a) Timeline for 2 clients/servers.



(b) Timeline for 5 clients/servers – startup phase.



(c) For 5 clients/servers – transition to efficient sequential I/O.

Figure 7.55.: Simulation of reading data from the hard disk drive using a record size of 100 KiB – excerpt.

tion, 400 MiB of memory suffice to hold enough data to schedule large sequential disk operations. Although no caching of written data is done for later reads, the efficient scheduling improves performance there too. In contrast to these results, the current behavior of our Linux servers seems suboptimal – the measurements vary with the cache size. An example which demonstrates the impact of cache size for simulated writes is given in Section 7.7.6.

g) Predicted performance for tmpfs, accessed by a small record size and independent writes, is worse than the one obtained with the hard disk drive model (see Figure 7.54a). Further, for most experiments with 100 KiB accesses, a smaller network granularity improves estimated time due to the pipelining of the store-and-forward switching. An exception to this observation is the result of independent reads for 5 clients/servers and of independent writes for 3 clients/servers (see Figure 7.54b and Figure 7.54a). As reported before, in those cases, minimal timing effects change the I/O behavior on the virtual servers leading to congestion and potentially less efficient disk access.

7.7.4. Hosting Multiple Processes per Node

To assess the behavior of concurrent operations, an experiment with five nodes hosting between 5 and 30 processes and five servers is discussed. Quantitative results are given in Figure 7.56.

Observations and interpretation

a) For writes, estimated and measured performance are alike. For reads, observed performance decreases with an increasing number of clients from more than 300 MiB/s to 250 MiB/s. The simulator stays at almost the original performance level.

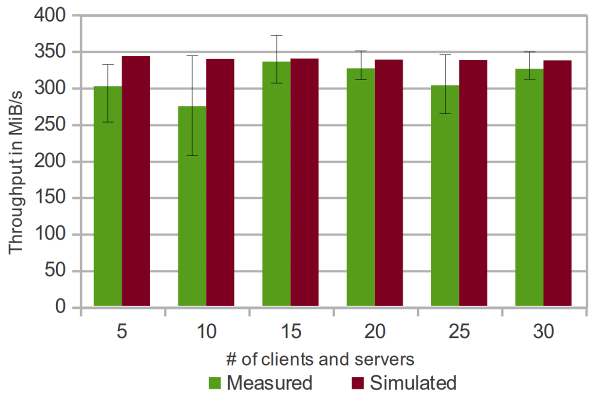
The causes for this behavior are the flow protocol in PVFS and the I/O-scheduling; a server multiplexes the block I/O among all pending requests, and for every read-request up to 8 operations of 256 KiB are issued to the Linux kernel. Since multiple clients issue requests, the disk must seek between the partitioned file regions. Thus, the observed I/O pattern is not sequential – depending on the timings, it can look more or less like random I/O of 256 KiB records. A longer explanation is given in [Kun07]. In the simulation, scheduling is done differently; therefore, the pattern looks like sequential I/O to the disk.

b) The maximum observed and simulated performance are about 350 MiB/s. This is also the theoretical limit of the network communication on our cluster (71 MiB/s for 5 clients). Simulation estimates this performance with 30 clients for all levels of access, except for collective non-contiguous I/O and for collective reads. Typically writes perform better – already with only five clients, the best performance is observed.

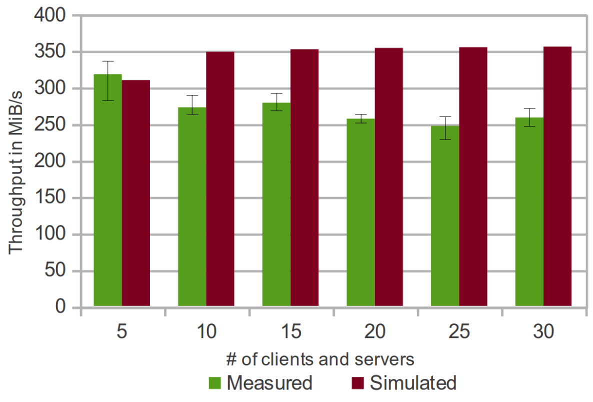
The reason for slower collective performance is that these operations synchronize the clients. Therefore, for collective I/O, the slowest client (and server) limits the speed of the group. While writes can be cached with write-behind, read performance is limited by the disk. Thus, collective contiguous read performance is lower than collective write (see Figure 7.56c and Figure 7.56d).

c) Even with an increasing number of clients, the performance of two-phase I/O is very low (look at Figure 7.56g and Figure 7.56h). Measured performance is below 80 MiB/s and even decreases slightly with the number of clients. This observation is caused by the suboptimal access pattern: By default all clients participate as I/O aggregators. However, the clients placed on a single node share the NIC, thus all their data must be communicated over the single NIC of the node. This is caused by PVFS flow protocol – with an increasing number of clients, the sequential access pattern degrades into a random read pattern.

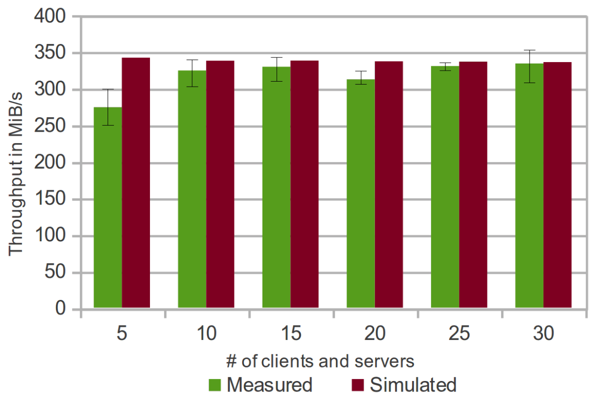
In PIOsimHD, the I/O scheduler can improve the order of operations; the more operations are pending, the better they can be aggregated and ordered – this leads to sequential patterns. Thus, disk I/O is not the bottleneck in the simulation. This can also be seen in traces for the simulated processing



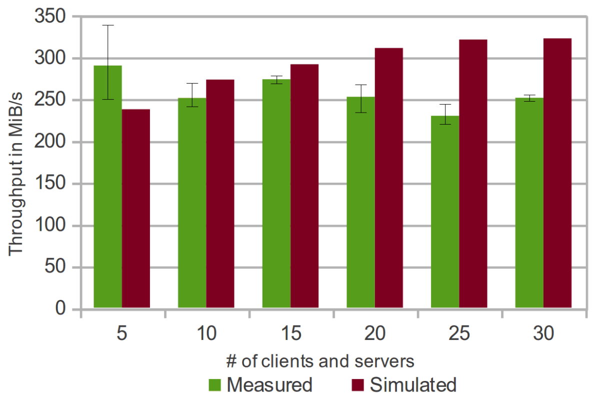
(a) Write independent, contiguous.



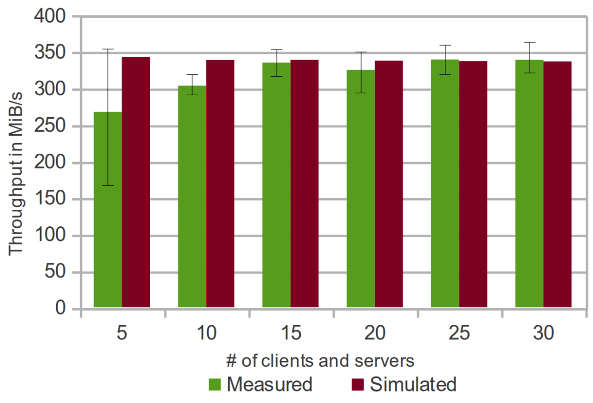
(b) Read independent, contiguous.



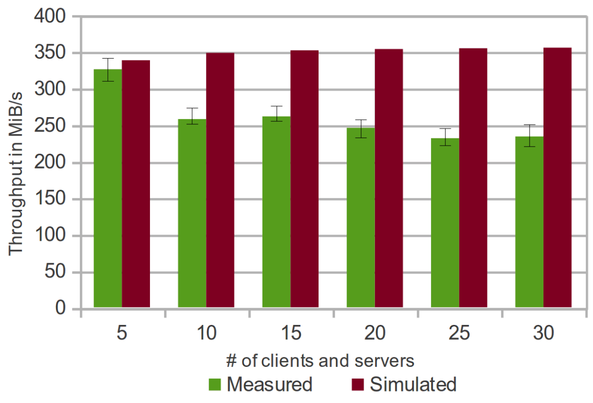
(c) Write collective, contiguous.



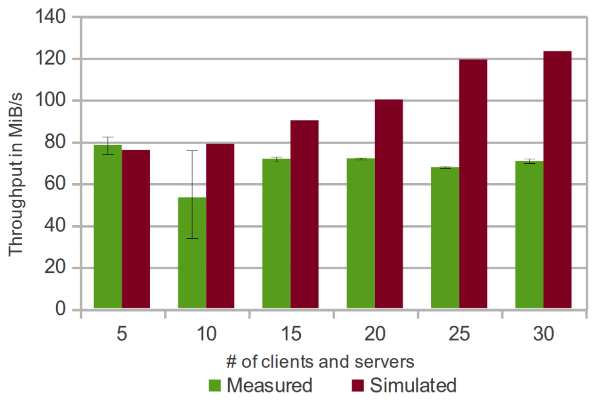
(d) Read collective, contiguous.



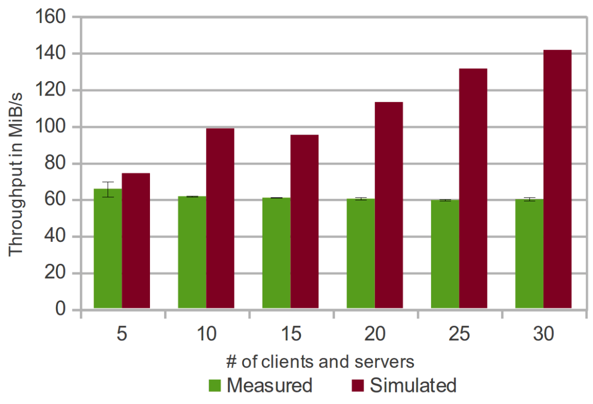
(e) Write independent, non-contiguous.



(f) Read independent, non-contiguous.

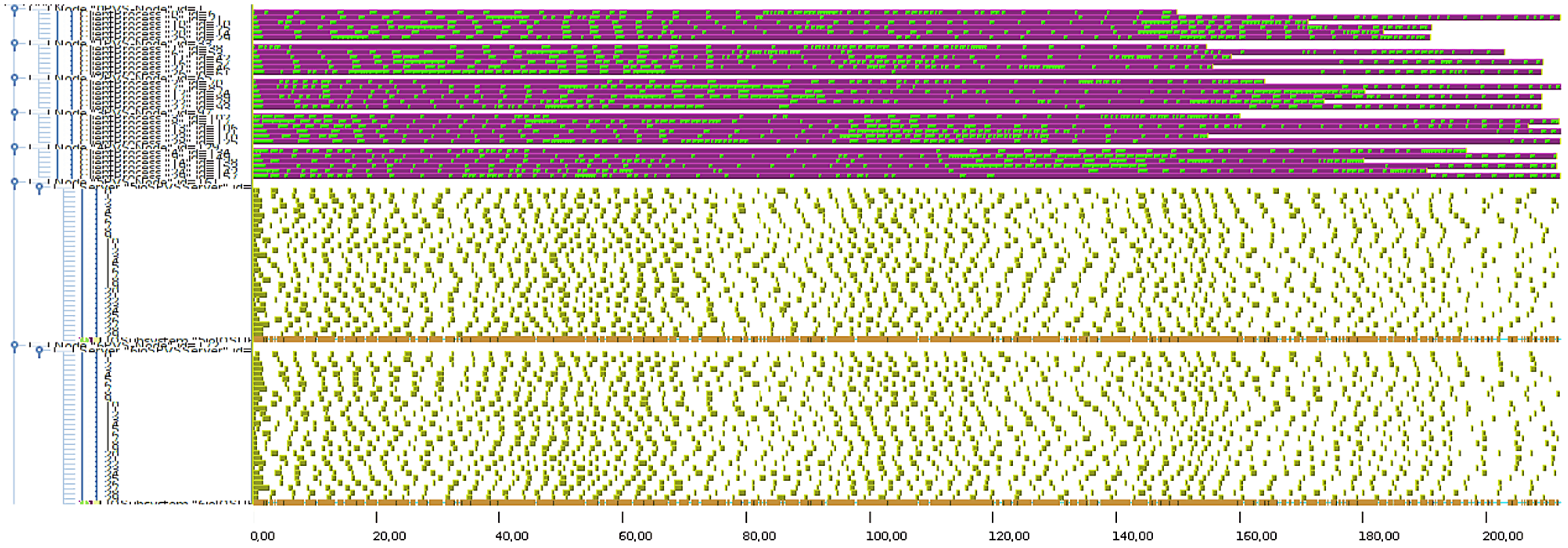


(g) Write collective, non-contiguous.



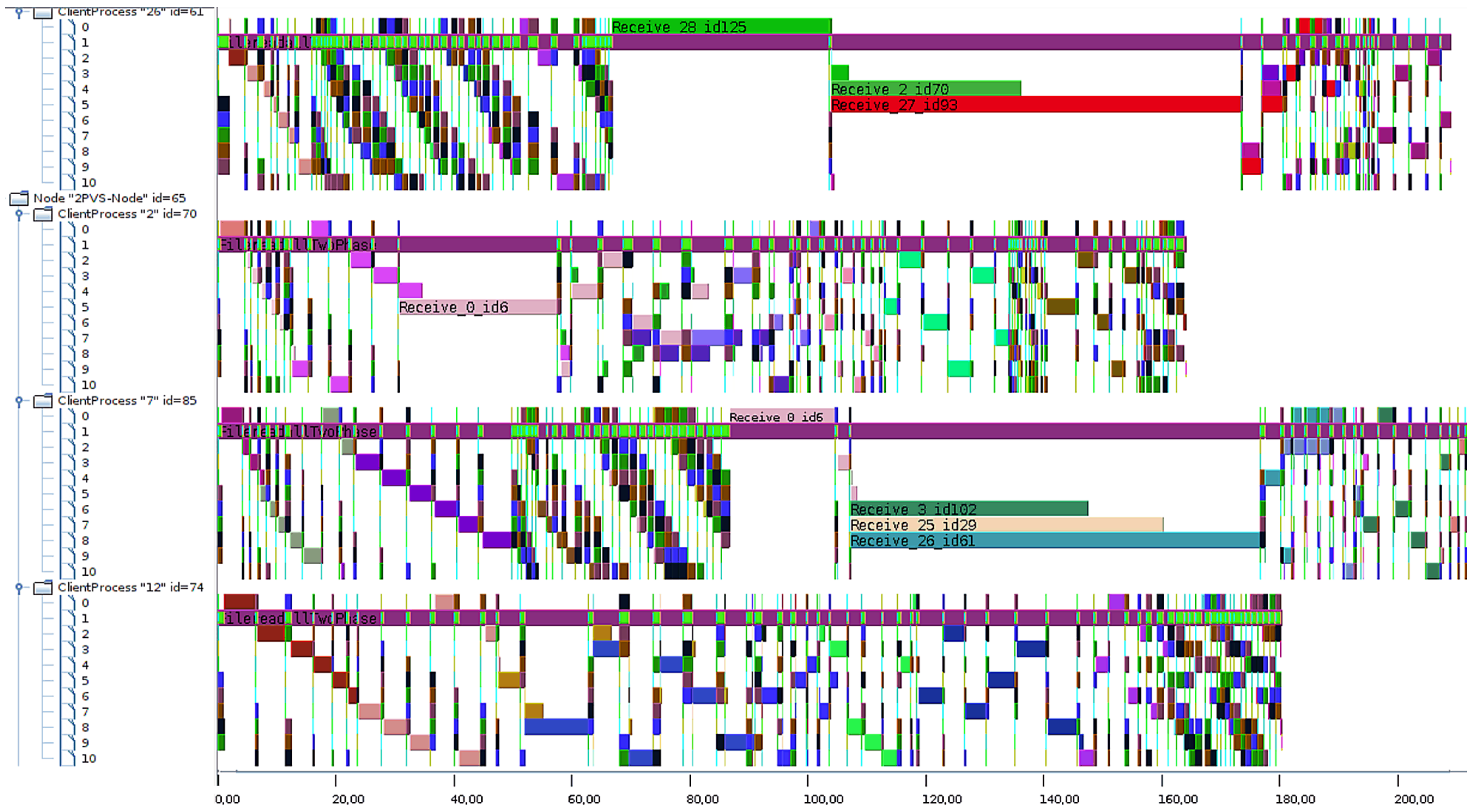
(h) Read collective, non-contiguous.

Figure 7.56.: Performance of accessing 100 MiB records with collective non-contiguous I/O – configuration with 5 nodes hosting a variable number of processes and 5 servers with 1000 MiB cache.



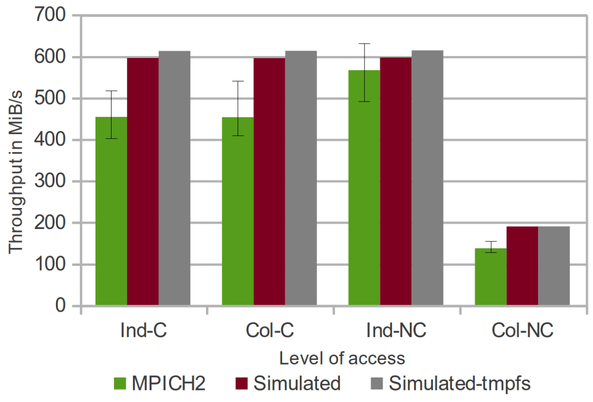
(a) Timeline overview. Timelines for all 30 processes are shown on top, timelines for two servers are given below. For each server, the lowest timeline shows activity of the I/O-subsystem.

Figure 7.57.: Simulation of 30 clients reading data from the hard disk drive using a record size of 100 MiB stored on five servers. Green rectangles on the client side indicate I/O requests, the other (purple) operations are caused by the data exchange.

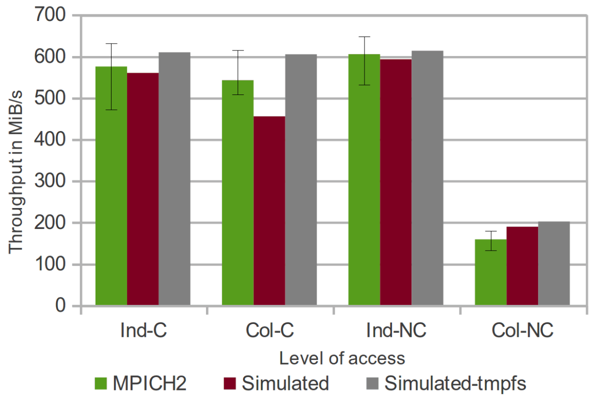


(b) Timelines including client-side operations – excerpt for 4 processes.

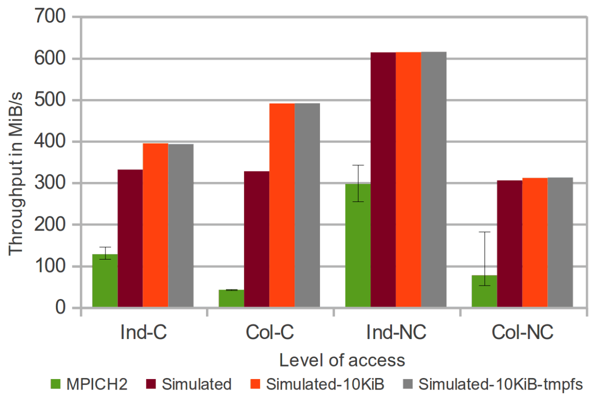
Figure 7.57.: Simulation of 30 clients reading data from the hard disk drive using a record size of 100 MiB stored on five servers. Green rectangles on the client side indicate I/O requests, the other (purple) operations are caused by the data exchange.



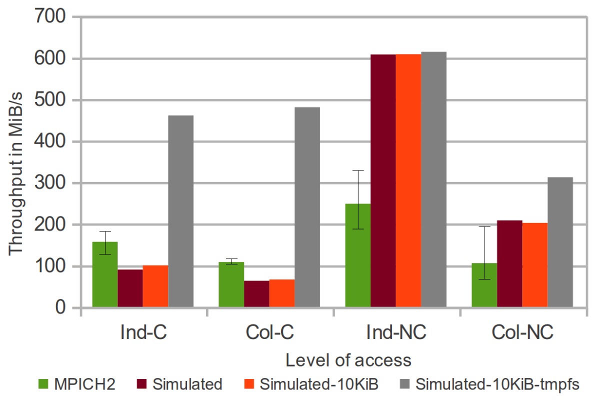
(a) Writing 100 MiB records.



(b) Reading 100 MiB records.



(c) Writing 100 KiB records.



(d) Reading 100 KiB records.

Figure 7.58.: Overlapping 8 clients and servers transferring 1 GiB of data per process, 2 GiB main memory per node.

of this experiment (look at Figure 7.57). It is interesting to see that clients terminate at a different timestamp. Whenever a client requests data which is just needed for itself, it does not invoke point-to-point operations for data exchange. Thus, it can skip the communication phase and proceed to the next I/O phase[23]. In a later phase, communication to another participant might be necessary, thus the client might stall until the other client is ready. Data which is read for another client is sent immediately to the peer[24]. The synchronization and different pace of the clients causes the interesting pattern.

Probably the answer for the increase in performance of the simulation result is the improved concurrent usage of the network by the clients. Unfortunately, due to the complexity of the algorithm, a detailed evaluation is out of the scope for this thesis.

7.7.5. Overlapping Client and Servers

On a cluster system, servers can be placed on the same nodes which host the clients. In this overlapping configuration, clients can exchange data quickly with the local server, but the network interface is shared between application and the file system. Results for 8 nodes each hosting one client and one server are shown in Figure 7.58. Every diagram shows the measured performance for all levels of access; tmpfs is simulated to estimate theoretical peak performance.

[23]This implementation is slightly different from the real two-phase implementation that requests synchronization for all point-to-point communications.

[24]This could be achieved with a real implementation, too. The buffer in which data read is to be stored could be set up during the initialization of the collective call.

Observations and interpretation

a) The maximum observed and the estimated performance are about 600 MiB/s. This is equal to the performance delivered by 7 links each with 71 MiB/s, and one hard disk drive (96 MiB/s). This is correct because we expect that, from the perspective of every client, $\frac{1}{8}$ of data is written locally and the remaining parts are communicated to remote servers. Thereby, the maximum performance of this configuration is higher on our cluster than the one that could be achieved with disjoint clients and servers (it would be approximately 570 MiB/s if we had that number of nodes).

b) With PIOsimHD, the maximum performance can be predicted well for 100 MiB records. In this case, estimated performance of collective contiguous I/O is slightly lower than measured.

It seems that a higher write throughput could be achieved on the real system. Especially, performance of many access patterns and 100 KiB records is below 50% of the value estimated by PIOsimHD. With that record size, the collective I/O on the real system performs almost like independent-contiguous I/O. For writes, the improved scheduling in PIOsimHD offers much better performance than observed in the experiments.

c) Similar to previous results, a transfer granularity of 10 KiB leads to a higher contiguous I/O performance. Measured read performance of contiguous accesses is higher than the estimate due to the caching of Linux – during reads, data is still available in the page cache, thus performance of reads is higher than for writes (as discussed before).

d) Contiguous reading small records achieves low performance because it creates a random pattern on the disk (as reported for cached I/O). However, this is not only visible on the real system – the simulated system achieves similar performance levels. Therefore, performance stays behind the theoretical possible throughput with tmpfs. The actual timing of the incoming requests influences the pattern, which explains the deviation in the measurements.

7.7.6. Simulating Server Behavior

In the previous sections the validation of parallel I/O showed that overall simulation with the simple model matches observable performance well. However, for writes PIOsimHD frequently outperforms the measurement, this is due to the improved cache scheduler in PIOsimHD.

In some cases, there is a discrepancy between estimation and observed behavior that is due to the influence of minimal timing effects on the server: A slight variation has a macroscopic effect. Consequently, the efficiency of the simulation depends on timings and race conditions which fluctuate on a real system as well.

A simple example will demonstrate the macroscopic impact of a slight variation: Assume two clients requesting 10 MiB of data from two servers, the request of the first client arrives first on the first server, but later than the request of the second client on the other server. Thus, when the first 10 MiB of data are read on the servers, the first server transfers the data read to the first client, while the second server transfers data to the second client. Then the second data block is read and sent to the other client. Thus, both clients must wait for completion of both operations. If the request of the first client was executed on both servers before that of the other client, then this client would have completed its task sooner. This timing issue can be seen for some simulation results, for example in Figure 7.60b (look at the first operations for the two clients and servers).

Complex patterns formed by minimal timing variations have been introduced in Figure 7.52, Figure 7.55 and in the figure for collective read from disk (Figure 7.60b). Further, it has been observed that individual clients or servers proceed faster than other clients; for example compare the trace in Figure 7.51d with Figure 7.52; in this simulation experiment, individual operations take longer, also the total time spent on the server side requests differs.

Several reasons for timing effects have been identified in the simulation and on the cluster system:

- Concurrency: The network interface or the block device of a server might process multiple requests concurrently. Therewith, the available resource (and performance) is multiplexed among all requests. The disk drive has an additional disadvantage; since its operation speed depends on the previously executed I/O operation, the observed access pattern is important.

  Performance of the network does not depend on previous operation. But a network congestion causes multiplexing of available network bandwidth among all streams. Therefore, it slows down the speed of all individual streams. In our setup, this can be caused by the fact that a single network interface needs to transfer data to multiple end-points, or multiple senders might send data to a single receiver. This can happen during client/server communication, or two-phase collective I/O.

  An example for network congestion is illustrated in Figure 7.52d.

- Random I/O: The disk drive could see a random-like pattern which increases seek-time and slows down processing. This happens for PVFS, since it processes operations with a granularity of the flow buffer size (256 KiB). In many cases, the simulator showed better performance because it aggregates operations to chunks of 10 MiB; this, in turn, reduces the costs for disk seeks. PVFS just relies on the capabilities of the Linux kernel and does not re-arrange read operations at all. Therefore, its performance is worse than that of the improved scheduling in PIOsimHD.

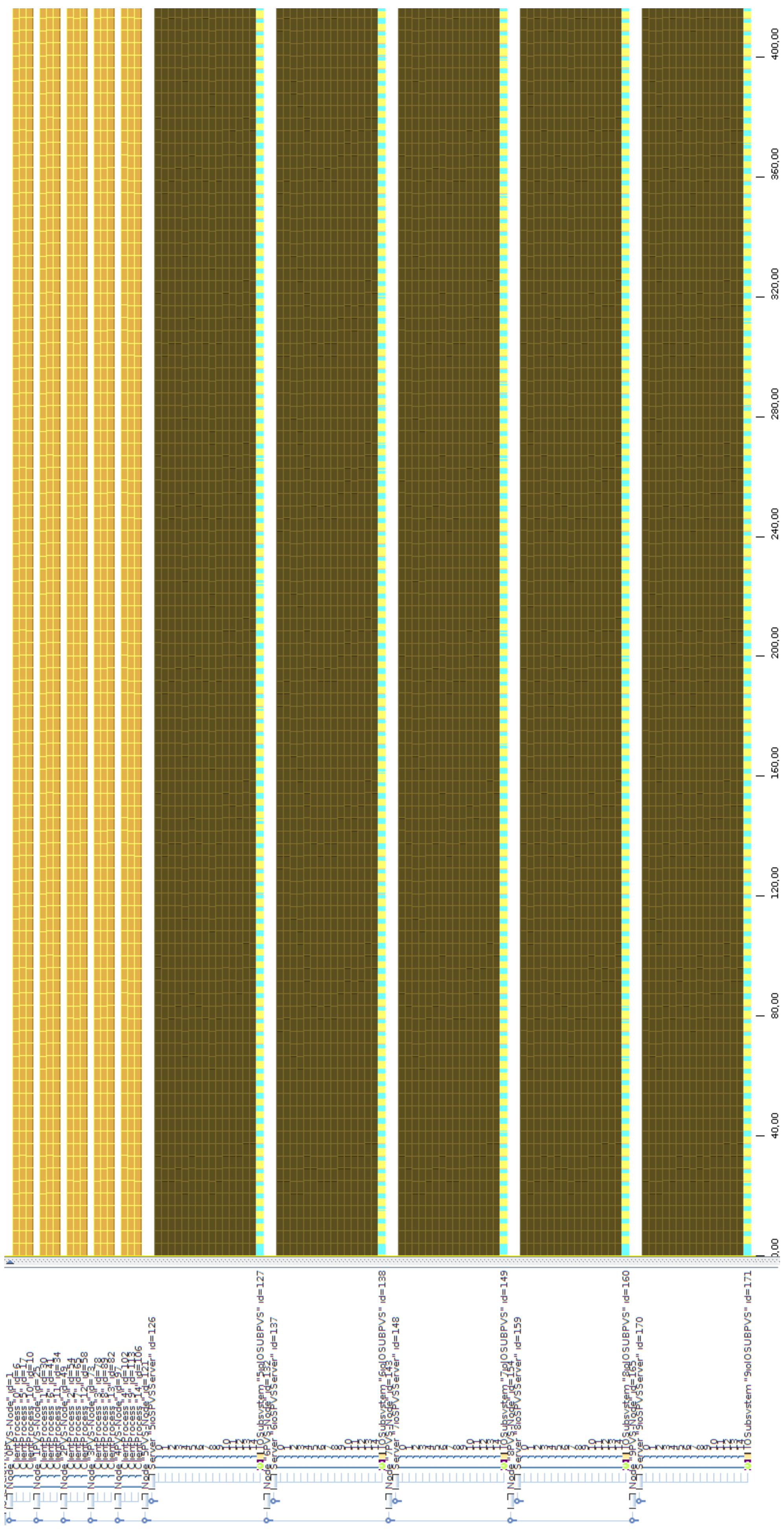
  Still, macroscopic behavior caused by minimal timing variations that can be observed in a real system can be found in the simulator as well. To give an example for an inefficient I/O pattern: In one experiment, 10 GiB of data is written per client, 15 clients access data on five servers. In Figure 7.59, two screenshots show the behavior for 1 GiB of memory and for 100 MiB of memory. With 100 MiB of memory, the amount of memory does not suffice to cache enough data to enable efficient aggregation and I/O scheduling. Therefore, time increases from 410 s to 700 s. In the screenshot in Figure 7.59b, it can be observed that the first server is a bottleneck; all requests queue up on the first server, while the other servers are able to process operations quickly. Such overloading of a single server has been reported for existing systems in [Kun07].

- Processing order of pending operations: When multiple operations are pending, one must be picked for further processing; this choice has an performance impact. In the simulator, the order in which operations are executed mainly depends on the timing by which events arrive. The I/O scheduler is an exception because it tries to optimize the access pattern for the disk drive. It does not consider the client; the client might wait for a single operation that is deferred by the I/O scheduler. When the block device accesses data or a network component executes an operation, it might be that the completion of that operation could still be blocked. For example, the network could be congested or the client might wait for completion of another operation. However, picking the right operation could enable issuing the next operation on the client side, thus processing could be theoretically faster.

  At the time an operation must be picked for execution, the system does not know which one is optimal, because no information about future processing is available. To pick the right operation, an oracle would be necessary. However, there could be rules which steer behavior in the hope to improve performance, for example, by preferring smaller accesses over larger accesses. Unfortunately, any rule could be suboptimal for other instances and the author claims that pathological cases can be found for all rules.

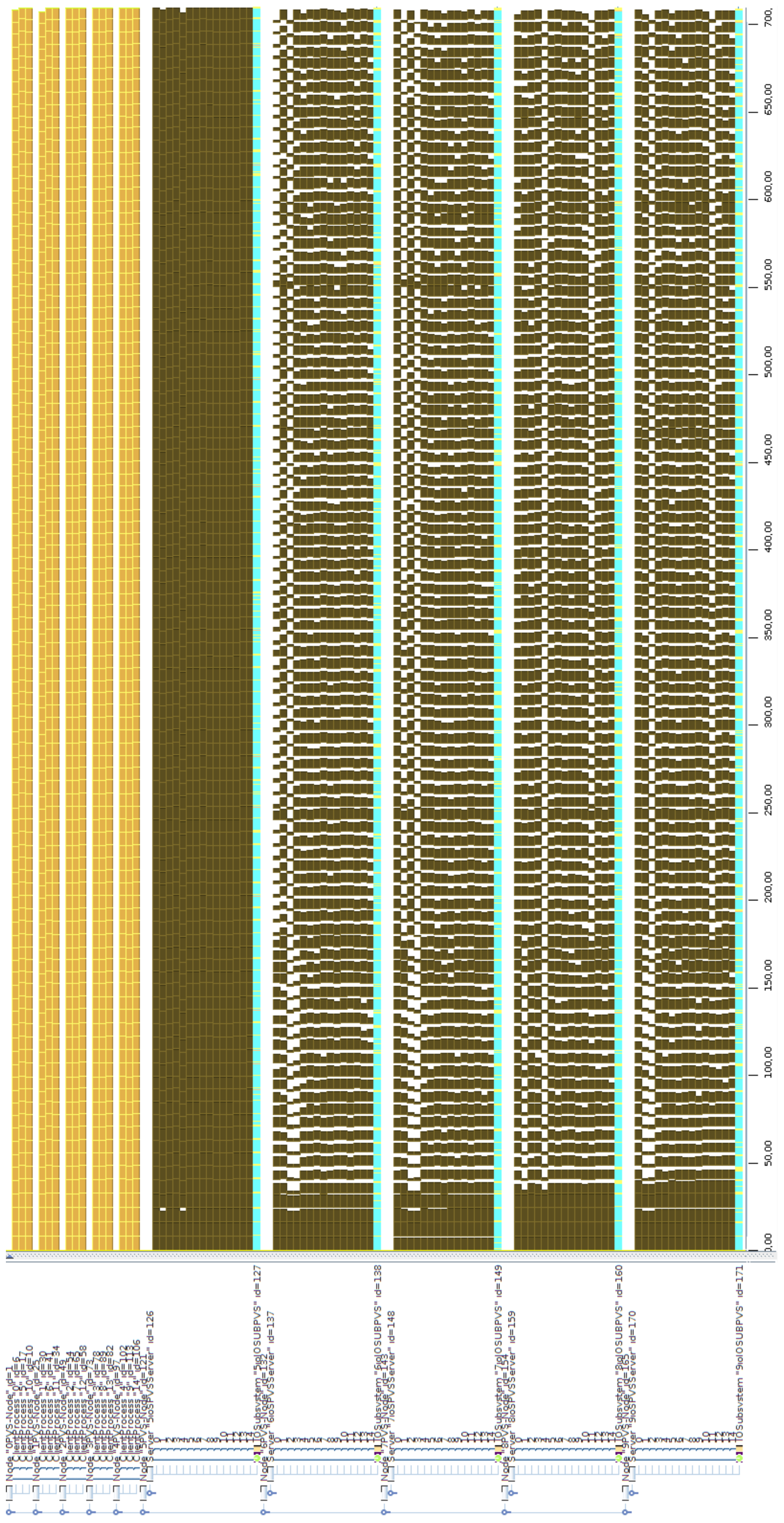
- OS noise and hardware effects: On the real system, random-like effects are caused by the complex interplay of the system components (refer to Section 3.6). An example of traced server behavior for collective, non-contiguous I/O shows the deviation observed in real runs; even on tmpfs, performance of individual operations can vary – a few I/O operations take much longer than on average (look at Figure 7.60). In the figure, activity on the two servers is shown; every request and every I/O call on the server-side is given in another timeline.

  Currently, PIOsimHD implements fixed characteristics and does not add further noise. This can lead to simpler and/or repeated patterns such as shown in Figure 7.55a and Figure 7.60b. The former


0,00
40,00
80,00
120,00
160,00
200,00
240,00
280,00
320,00
360,00
400,00
Server "5ioSPVSServer" id=126
1IOSubsystem "5ioIOSUBPVS" id=127
Server "6ioSPVSServer" id=137
IOSubsystem "6ioIOSUBPVS" id=138
Server "7ioSPVSServer" id=148
IOSubsystem "7ioIOSUBPVS" id=149
Server "8ioSPVSServer" id=159
IOSubsystem "8ioIOSUBPVS" id=160
Server "9ioSPVSServer" id=170
IOSubsystem "9ioIOSUBPVS" id=171

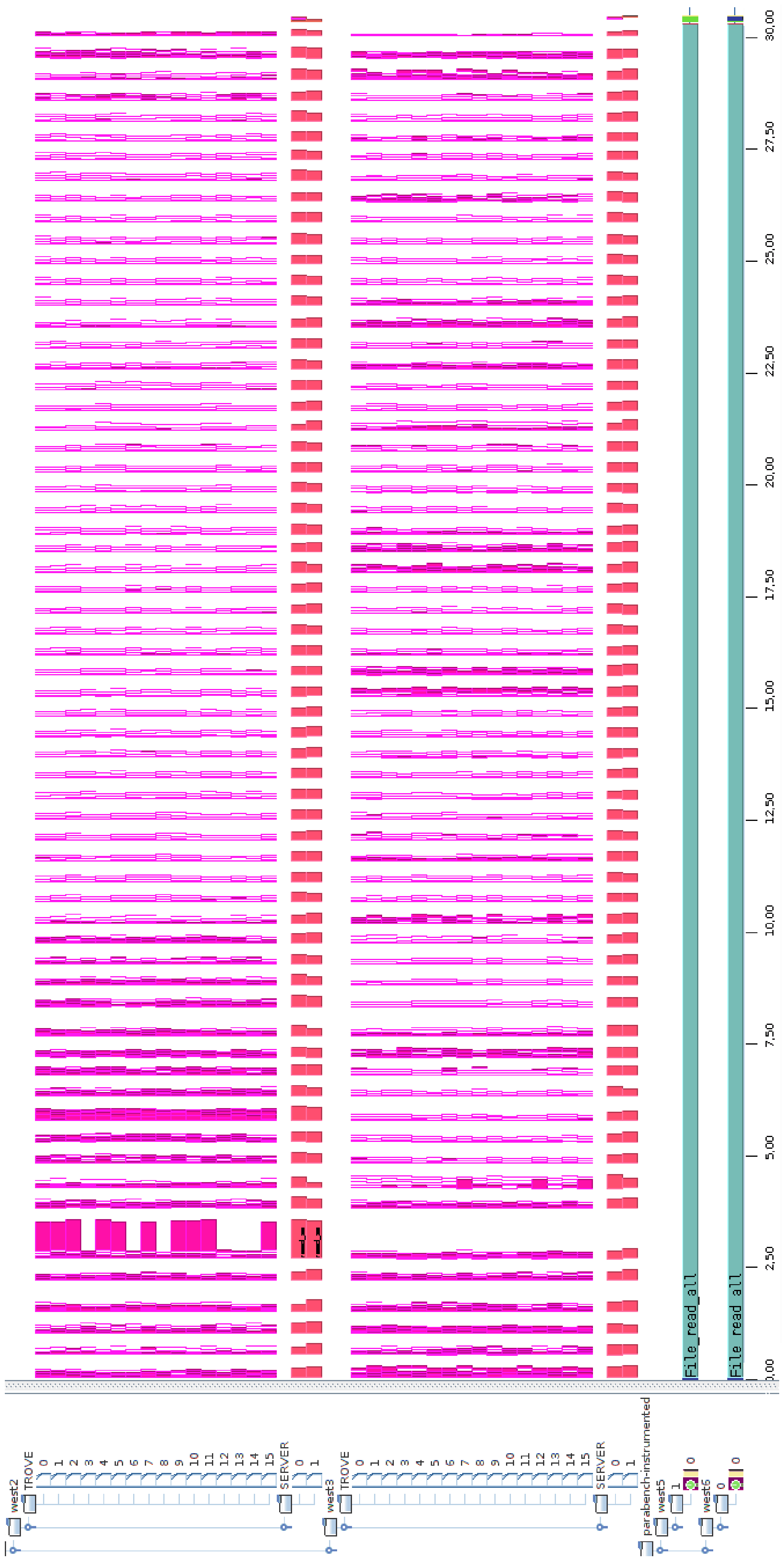
(a) 1 GiB main memory.

Figure 7.59.: Simulation of 15 clients and 5 servers, each client writes 10 GiB of data.



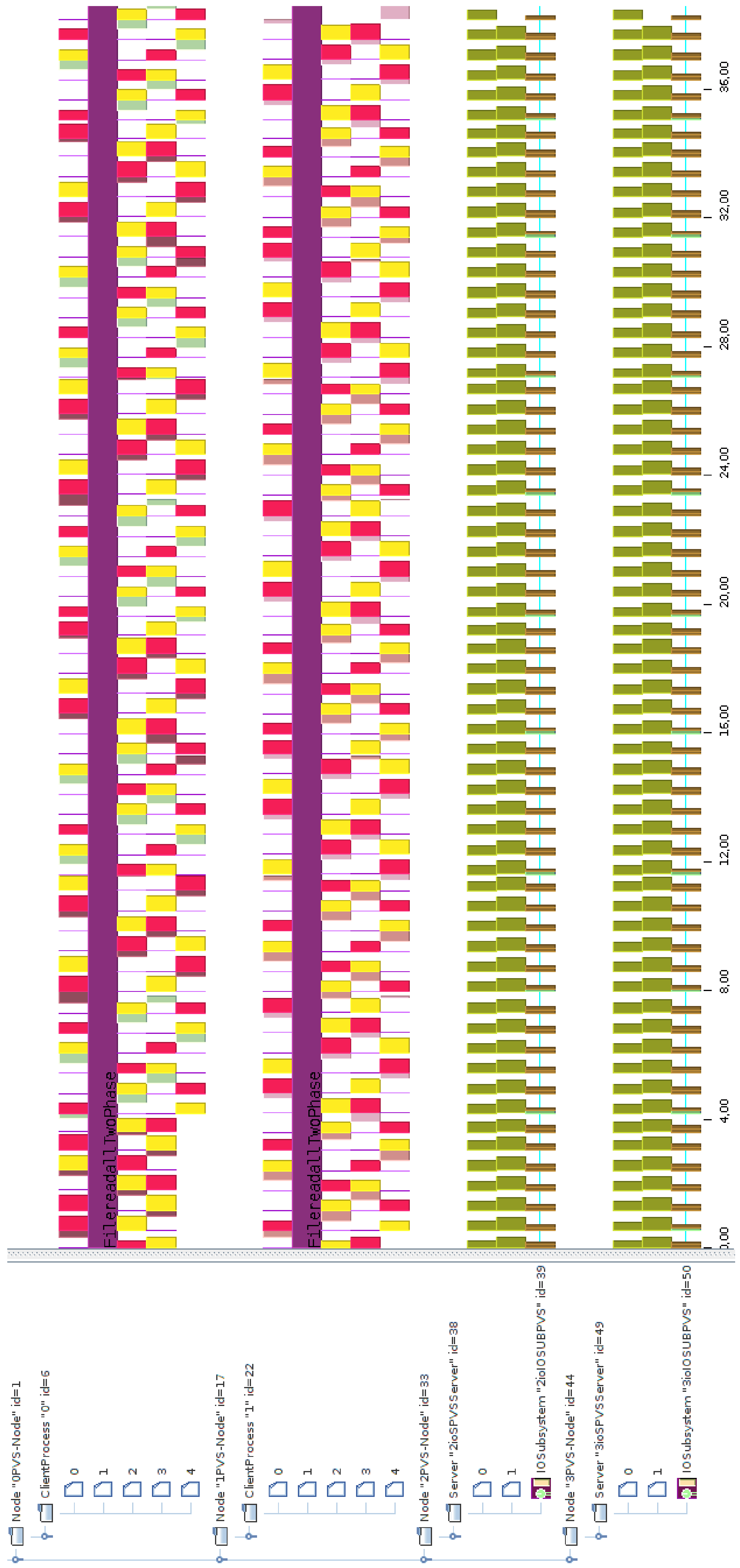
(b) 100 MiB main memory.

Figure 7.59.: Simulation of 15 clients and 5 servers, each client writes 10 GiB of data.



(a) Read behavior.

Figure 7.60.: Visualization of measured collective non-contiguous I/O for two clients/servers and tmpfs.



(b) Read behavior.

Figure 7.60.: Visualization of simulated collective non-contiguous I/O for two clients/servers using tmpfs.

figure has been discussed already. The latter figure visualizes the simulation results for tmpfs; the pattern is repeated after a while.

All those effects can be observed for a real system, however, analysis is usually difficult due to the lack of introspection. This kind of sensitivity to timing effects is inherent to the way parallel I/O is conducted. On a real system, experiments which are sensitive for timing variations lead to a higher deviation of the timing between repeated measurements. Since the models in the simulation are characterized by a fixed timing, a run results in a deterministic result. However, despite the simple models in PIOsimHD, the simulated behavior could be very complex by itself and reproduce many effects in silico.

The modifications in HDTrace to visualize internal processing of PVFS allow introspecting client and server activity (refer to [Tie09] for details about the capabilities). With the help of Sunshot, those imbalances could be spotted on traced data and in the simulation. If network congestions degrade performance, for example as seen in Figure 7.52, then a trace of the system cannot determine the exact cause, because existing tracing tools are not able to capture network traffic. Further discussion of performance analysis with trace-based tools and trace results showing similar patterns to the simulation are provided in [KTML09, KL08, Kun07].

7.7.7. Summary and Conclusions

Parallel I/O is evaluated with the programmable benchmark parabench for a number of experiments that vary orthogonal parameters: the access type, record size (100 KiB or 100 MiB), level of access, storage backend (tmpfs or HDD), amount of cache, number of clients and servers and whether clients and servers are placed on the same or physically separated nodes.

In many cases, performance of PVFS is well approximated by the simulation; especially, cacheable accesses and non-contiguous I/O demonstrate a good match. In most cases simulation provides upper bounds to PVFS performance. Examples in which a much better performance is expected, are independent I/O and small record sizes. Collective non-contiguous requests are usually slightly better in the simulation. In general, simulation results are better because the simulator uses an improved server-sided I/O scheduler. It can be observed that PVFS performance is much lower in many cases and degrades with an increase of clients and servers for independent operations because it schedules operations of all clients concurrently leading to a random-like pattern. For independent non-contiguous access, PVFS achieves its best performance. In all cases, contiguous collective calls are typically slower, because they lead to random accesses and involve additional communication.

In some cases, performance of the simulation is slightly lower (mostly by 20% and in a few cases by up to 50%). It turns out that the modeled system is sensitive to timing issues. The timing issues are investigated by inspecting generated traces – the simulation allows detailed assessment of results. These effects are also visible on the real systems, but, due to different data flow and variability in the system characteristics, they usually manifest on other configurations. An example in which timing on real system and simulation behave very similar are the collective reads with 100 KiB records. In this case, simulation and observation lead to a performance of about 25 MiB/s, however, with tmpfs a performance of 250 MiB/s is possible.

The influence of the network granularity is investigated, showing that it influences timing and improves performance for small record sizes but not for large record sizes, because data transfer through the network can be pipelined. While the model does not incorporate short term variability explicitly, the individual I/O operations lead to similar behavior as in the real system. For example, simulation traces reveal a transition from random to efficient sequential I/O. Also, an overloading of individual servers is demonstrated that degrades performance, i.e., one server is busy to schedule operations and causes random access on disks, while the others have idle time. The revealed timing effects show that PIOsimHD is a valuable tool for investigating real world behavior. In a master thesis the simulator has already been used for evaluating collective I/O and disk-directed I/O.

7.8. Verification of the Implemented Collective Communication

During the execution of the parallel program, every command and thus every MPI operation must explicitly be programmed in PIOsimHD. Currently, the following collective calls of MPI are implemented: `MPI_Barrier()`, `MPI_Reduce()`, `MPI_Allreduce()`, `MPI_Bcast()`, `MPI_Gather()`, `MPI_Allgather()` and `MPI_Scatter()`. The implementation makes use of the concept of state machines as discussed in Section 6.5.

The simulator provides at least two alternative implementations for these collective calls: A simple realization that meets the semantics of MPI, and a re-implementation of MPICH2 algorithms. For every call, the simple implementation transfers all data through the root process. Further, several more sophisticated algorithms have been implemented. With the help of PIOsimHD, the efficiency of those algorithms can be assessed. An example of obtained performance for alternative `MPI_Bcast()` implementations is given in Section 7.10.

In order to assess simulation results and observation, the simulated collective call and the real call should behave alike. Thus, the real communication behavior should be encoded in the simulator. However, it is tedious to replicate existing codes, and as we have observed, current behavior is suboptimal for MPICH2 and Open MPI in many cases. Currently, one algorithm provided by MPICH2 is implemented in the simulator for every collective call, although MPICH2 offers multiple algorithms for many collective calls.

Before details about the implemented collective are given recall the optimization potential within MPICH2: the implementation tries to optimize performance by selecting an appropriate algorithm for the specified parameters[25]. Typically, in MPICH2, the algorithm is selected depending on the size of the data to be communicated, and the number of processes. In most collective calls, one algorithm is provided which is optimized for latency and another one that is optimized for throughput. Environment variables can be set to adjust the message size at which the algorithm is changed[26]. Several algorithms implemented in MPICH2 are also SMP-aware (performance and SMP behavior has been discussed in Section 7.6).

In a software lab, a student re-implemented MPICH2 algorithms in PIOsimHD and assessed the correctness of the implemented algorithms [Thi12]. In detail, first the existing algorithms have been analyzed by inspecting the MPICH2 source code and by analyzing the trace files which have been used to validate the collective patterns (see Section 7.6). Second, existing code in PIOsimHD has been extended to mimic one of the used algorithm for every collective call. For the validation, the traces have been compared with the simulation results obtained by using the hardware model of our cluster. For every collective call screenshots for two configurations are also provided. The implemented broadcast algorithm is also verified against the extracted point-to-point communication patterns in Section 7.10.

Note that the implementation in the simulation may differ for certain observations. This is due to the fact that most implementations in the simulator are not SMP-aware. Further, in order to optimize throughput, MPICH2 uses multiple algorithms for a single MPI command. This is not realized in the simulator, yet. However, the software lab proved the practicability of the modular design for implementing and evaluating new MPI implementations.

7.9. Simulating Behavior of Scientific Applications

With the help of the introduced HDTrace environment, communication and I/O behavior of existing applications can be traced and their behavior can be replayed on arbitrary virtual cluster systems[27]. While several applications have been traced with HDTrace and replayed with the simulator already, the results obtained with the simple Jacobi *partial differential equation* (PDE) solver `partdiff-par` are evaluated and

[25]For further details about the optimization potential refer to Section 2.3.4.

[26]The variables are documented in `README.envvar` which is distributed with the source tarball. For instance, the selected `MPI_Allreduce()` implementation depends on the environment variable `MPIR_PARAM_ALLREDUCE_SHORT_MSG_SIZE`.

[27]Refer to Section 4.3.1 for a detailed description of the workflow.

discussed[28]. The reason for this choice is that the PDE application can easily be parameterized for arbitrary problem sizes. This allows us to select an arbitrary communication to computation ratio and, furthermore, to specify the I/O workload.

Many of the experiments conducted behave similar. Therefore, a subset of the experiments has been chosen for discussion that represents the parameter space and showed interesting behavior. The methodology of the comparison including experiments conducted is described in Section 7.9.2.

The experiments are classified and discussed depending on the bottleneck on the real system: Network-bound experiments are assessed in Section 7.9.3, computation-bound experiments in Section 7.9.5, I/O-bound experiments storing data on tmpfs in Section 7.9.6 and I/O-bound experiments storing data on disk drives in Section 7.9.7. In Section 7.9.8, a mixed unbalanced workload is evaluated which stresses CPU and file system, implying a synchronization overhead.

The semantics of MPI functions implies certain synchronization of processes, early arriving processes have to wait for their communication partners to be ready. With the simulator, these waiting times can be studied. This problem is related to the *critical path* analysis. A definition of the critical path according to Schulz:

> *"It identifies the longest execution sequence without wait delays throughout the code. In other words, the critical path is the global execution path that inflict wait operations on other nodes without itself being stalled. Hence, it dictates the overall runtime and knowing it is important to understand an application's runtime and message behavior and to target optimizations."* [Sch05]

A slight modification to this definition is applied for this thesis: The critical path is the global execution path on which data dependency of communication inflict wait time on other processes without being stalled by not-ready communication partners. In other words, whenever a MPI call is performed on the critical path, all other processes that participate due to the semantics of the MPI call are ready. Analyzing waiting time throughout the execution path is demonstrated with HDTrace for `partdiff-par` in Section 7.9.4.

7.9.1. The PDE Solver `partdiff-par`

The partial differential equation solver `partdiff-par` implements the Jacobi method for a quadric 2D matrix with fixed values on the borders. It iterates over all matrix elements and computes a new matrix, which serves as input for the next iteration. For every element the new value is computed by applying a four-point stencil – the four neighboring values are needed. This program does not read an input matrix, instead the matrix is initialized.

The program is parallelized by distributing consecutive rows of the matrix among the processes. With 7 rows and 2 processes that means, for example, that row 1 to 4 is assigned to the first process, and row 5 to 7 to the other process. With this domain decomposition, every process can compute the local submatrix for the next iteration. Due to the data dependency of the stencil, two halo rows are required: one for the process above and one for the process below. After the new matrix is computed, the boundary rows are exchanged with the neighboring processes using `MPI_Sendrecv()`. Then every process can start with its next iteration.

The PDE terminates when a number of iterations has been computed, it outputs a few values of the final matrix on the terminal and writes them into a file using POSIX I/O – these values can be used to check the correctness of the computation.

Parallel I/O is done between two iterations, the frequency can be parameterized on the command line. The PDE supports writing out the matrix diagonal or the full matrix; collective and individual MPI-IO calls are supported. The matrix diagonal is written to allow assessing of the program's convergence behavior. In this process, data is appended to a file; this file can be inspected online – while the program still updates

[28]The program itself is introduced in Section 7.9.1.

the data. In the program terminology, this process is called *visualization*. The full matrix is stored in a file to *checkpoint* the state of processing; two checkpoint files are kept and alternately overwritten every time a checkpoint is written. This ensures that a crash during the checkpoint procedure does not corrupt the previous checkpoint.

In all cases, data is written with a single MPI call and every process writes the data it manages. Actually, due to the limitations of MPI-IO, at most 2 GiB of data can be written with a single MPI call. Therefore, if more than 2 GiB of data have to be written for a checkpoint, the data will be partitioned into chunks of 1.9 GiB.

7.9.2. Methodology

Experiments conducted Due to the flexibility of the program, there are several parameters that define an experiment. They can be classified into runtime parameters for `partdiff-par` and the system configuration. Many different sets of experiments have been executed on our cluster. Conducted experiments are listed in Table 7.10. Table entries and the experimental variability are explained in the following:

- Runtime parameters define the size of the matrix and the number of iterations performed by the program. Further, the frequencies of appending visualization data and of checkpointing are considered as runtime parameters. Data is written when the current iteration is a multiple of the specified number, e.g., with a frequency of 501, the operation is performed every time 501 iterations are completed.

- Matrix size: This value gives the size of the matrix which helps assessing runtime – for example, the amount of data written in a checkpoint corresponds to the matrix size. The problem size depends on the number of *interlines*, which is a runtime parameter. For a quick validation of the result, a subset of the matrix is output (9 times 9 points). Interlines define the number of non-visible rows and columns between these points and thus the full matrix size. The matrix is square with $N = 8 \cdot \text{interlines} + 9$ rows, thus the required memory is in the order of $O(\text{interlines}^2)$ – each matrix element is a double precision floating point number. In the table, the size of the matrix is encoded in the experiment name: extra-small (XS), small (S), medium (M), large (L) and extra large (XL).

- Configuration: A single configuration is defined by the number of processes and the resources processes run on. In detail: the total number of nodes, the number of nodes for the PDE and the number of application processes and server processes (at most one server process is started per node). In most experiments the numbers of processes etc. are varied, variable numbers are given with N in the table. N is used as a single configuration parameter within an experiment, so, for example, in experiment XS-0S the number of nodes and processes is always identical.

  Experiments either use disjoint nodes to execute PDE and file system servers, or PDE and server processes are executed on the same nodes. If the number of nodes is larger than the number of PDE nodes, server processes are placed on these additional nodes. Such a configuration with disjoint servers is indicated in the experiment name with a D. If server processes are run on the same nodes and thus they overlap with the application this is indicated in the name with an O.

- Memory restriction: To enforce real disk I/O, the memory of the server nodes is limited with the `mem-eater` utility for some experiments. The memory footprint of the PDE solver must be taken into account for overlapping configurations – memory occupied by the PDE is not available for caching of the server. Approximately twice the matrix size is required for the PDE. Further, approximately 100 MiB is needed for the Linux kernel and temporary server buffers. The memory limit of an experiment can also depend on the number of clients and servers. Matrix size and memory limitation is provided in the table.

  For example, in the configurations *M-O-1000M* effectively about 1000 MiB of cache is available per server process (and node). In configuration *M-O-s1000M*, an aggregated 1000 MiB of cache is available for all servers, thus at most 50% of a single matrix can be cached on the servers.

Experiment	Matrix size	Configuration					Runtime parameters			
	in MiB	Nodes	Servers	PDE nodes	PDE procs	Memory limit	Interlines	Iter.	Freq chk	Freq vis
XS-0S	4.9	N	0	N	N	Unlimited	100	1000	No	No
S-2S-1CN-150M	488	3	2	1	N	150	1000	1000	501	10
S-2S-tmpfs	488	N+2	2	N	N	Unlimited	1000	1000	501	10
S-2S-150M	488	N+2	2	N	N	150	1000	1000	501	10
M-D	1953	$2 \cdot N$	N	N	N	Unlimited	2000	100	5	1
M-O	1953	N	N	N	N	Unlimited	2000	100	5	1
M-D-1000M	1953	$2 \cdot N$	N	N	N	1000	2000	100	5	1
M-D-s900M	1953	$2 \cdot N$	N	N	N	100+900/N	2000	100	5	1
M-O-1000M	1953	N	N	N	N	1000+3900/N	2000	100	5	1
M-O-s1000M	1953	N	N	N	N	100+4800/N	2000	100	5	1
L-D	4394	$2 \cdot N$	N	N	N	Unlimited	3000	100	5	1
L-O	4394	N	N	N	N	Unlimited	3000	100	5	1
XL-0S	23,926	N	0	N	N	Unlimited	7000	40	No	No
XL-0S-5CN	23,926	5	0	5	$N \cdot 5$	Unlimited	7000	40	No	No
XL-5C	23,926	N+5	N	5	5	Unlimited	7000	40	5	1
XL-O	23,926	N	N	N	N	Unlimited	7000	40	5	1

Table 7.10.: Experiments conducted with the Jacobi PDE solver – configurations and runtime parameters. A variable number of clients and servers is indicated with N, i.e., these experiments are run for different N. The name of the experiment encodes the size of the matrix (as defined by the interlines), whether the configuration is overlapping (O) or disjoint (D) and further configuration options of interest.

Depending on the experiment, CPU[29], network and file system are stressed differently and run-time is dominated by one hardware component: The XS experiment is network-bound, the M, L and XL experiments are I/O-bound with the exception of the computation-bound XL-0S experiments. The S experiments are computation bound but stress network and I/O-subsystem as well.

Simulating application behavior In order to replay the application behavior, the cluster model is used, but in a few experiments multiple processes are placed per core. Since wall-clock time of the recorded trace is used as a reference for simulating computation time, a virtual processor is created with a number of cores equal to the placed processes – this model guarantees an accurate replay of the computation time in the simulator.

The amount of memory per server is set according to the memory limit in the configuration. For experiments in which client and servers are placed on the same node, the amount of memory is adjusted to account for the matrices of the PDE. The available memory is reduced by two times the matrix size divided by the number of processes placed per node, and additional 100 MiB to account for the Linux system. On the real system, the available memory varies slightly and thus the value is not exact. However, the cluster model offers at least a similar amount of cache.

In PIOsimHD, a network granularity of 100 KiB is set and implementations for collective operations are chosen that mimic algorithms of MPICH2[30]. A single row has a size of 6 KiB for 100 interlines and about 440 KiB for the large matrix, so due to the store-and-forward switching, the selected network granularity seems suboptimal. However, as we will see the granularity suffices to mimic the application behavior. Note that the sequential (POSIX) output of some matrix values at the end of the application is not simulated – the duration of unknown calls are converted to a compute job. Thus, the time spend in the sequential output is simulated perfectly.

Since computation time can vary in the observations, all traces for an experiment that have been measured are also simulated; every configuration is measured three times to capture this variance. Minimum and maximum are visualized as an error bar in generated graphs.

[29]Since memory access is included in the computation time, it is not mentioned separately. In fact, with the chosen functions, the PDE algorithm is memory-bound for larger matrix sizes and not computation-bound.

[30] Currently, only one collective algorithm of MPICH2 is supported (per MPI function), but MPICH2 might use several algorithms. Refer to Section 7.8 for a description of implemented algorithms.

To analyze the traces better, PIOsimHD is re-run with different internal implementations for the traced MPI commands. For example, the implementation for opening and closing a file can be altered to broadcast metadata[31]. The following simulations are run with the recorded trace files:

- *Validation*: To validate the simulation the default settings are applied and the results are compared to the measurement.
- Determination of the *compute time*: By replaying the traces and skipping all MPI calls the maximum compute time of any processes is computed. The computed time will serve as baseline to estimate the accuracy for the simulation, since execution cannot be faster than the maximum compute time.

  While the value could be determined by inspecting traces or profiles, it was simpler to automatically compute the values with the already existing code that runs the PDE traces in the simulator.
- In *SyncVirtually* an infinite fast network and IO subsystem is simulated. With this configuration the load-balance between the processes is checked. Also, it approximates performance of deploying the fastest imaginable network technology.

  Internally, all MPI calls are replaced with implementations that ensure causality of the call, but do not use the network at all. Collective operations will synchronize the processes – the last process which joins a collective operation activates all other processes. Opening a file is treated as a collective operation, independent I/O and closing a file does nothing. An alternative implementation of point-to-point communication matches messages directly in the commands without using network communication. Thus, a receive will complete immediately if the message has been send; still the rendezvous protocol will block a sender of a large message until the receiver is ready.

  This experiment effectively computes the critical path and allows visualization of waiting times in Sunshot; a small experiment in Section 7.9.4 shows how.
- In *No-IO*, the simulator is configured to ignore I/O calls. Thus, it replays just communication and the *impact of IO* to the total run-time can be assessed. This value could not be determined from the traces alone. While an executed I/O operation defers future operations of a process, the application runtime increases iff it is on the critical path.
- *Flush-on-close*. In this experiment an alternative persistency semantics is evaluated. By default writes can be deferred, thus a crash of a data server might cause data loss. To avoid data loss, a close could invoke a flush on the storage. Internally, the `MPI_File_close()` is replaced with an implementation that flushes cached data on all data servers. In detail, this implementation synchronizes all processes by using a barrier, then Rank 0 sends flush requests to all servers, and finally, all processes synchronize with another barrier.

Comparing observation and simulation Every configuration is measured and re-run three times to give an impression on the variability of network, disk and computation performance. Runtime of Rank 0 is extracted from the trace file and taken as a reference. The average value of all runs is computed and diagrams are created which show the average and an error bar for the minimum and maximum.

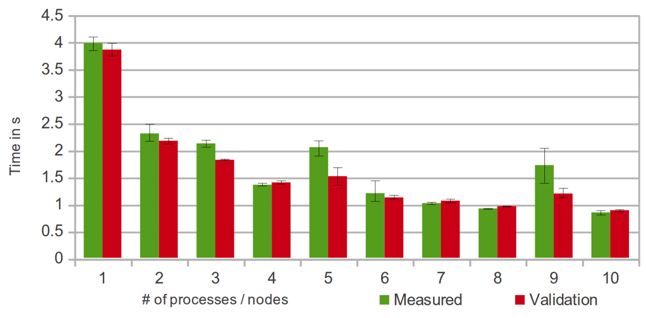
It is tempting to compare only execution time and estimated run-time for the application. However, since compute time can be simulated accurately and the application performs a good fraction of computation, a relative measure is necessary. Otherwise any level of accuracy can be obtained by increasing the fraction of computation.

To provide a quantitative comparison of the simulation with the observation the following metrics are defined:

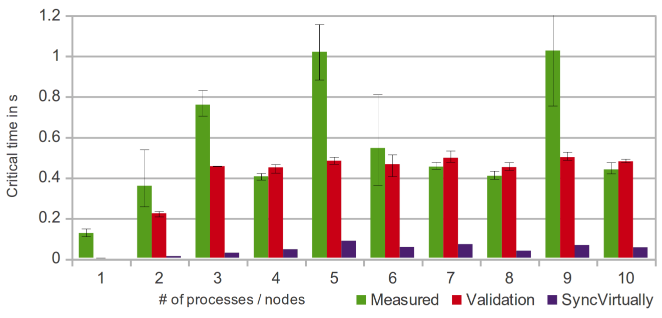
$$\text{critical time} = \text{execution time} - \text{compute time} \tag{7.7}$$

$$\text{relative accuracy} = \frac{\text{critical time of the the simulation}}{\text{critical time of the observation}} \cdot 100 \tag{7.8}$$

[31]The selection process of implementations is described in Section 5.4.



(a) Wall-clock time.



(b) Critical time due to communication.

Figure 7.61.: Times of the network-bound experiment XS-0S – a variable number of processes each hosted on its own node.

Execution time is the observed run-time of the program, it can be computed for simulation runs as well as for the measurement. The *critical time* is equal to the amount of time lost by doing I/O and communication; to be more precise, it is the amount of time spend in communication and I/O on the critical path of a parallel application.

A relative accuracy can be computed by comparing the critical time of the simulation with the observation. Originally, it was intended to compare simulation and observation with this relative measure, because the accuracy of the simulated network and I/O behavior could be quantitatively assessed. However, it turned out that such a relative comparison is not appropriate, because the PDE relies on many different calls and the timings of individual operations can compensate. Instead of using the relative accuracy, recorded and simulated profiles and traces are directly compared for selected experiments.

An additional advantage of critical time is that it provides a quantitative absolute view, this on the one hand, allows to assess deviation in measurements and, on the other hand, allows comparison of simulation results with different simulation configurations. Therefore, the critical time is used solely for assessing measurement and the various simulation results quantitatively while visual inspection with Sunshot verifies accurate simulation of individual operations.

7.9.3. Network-Bound Workload

In experiment XS-0S a small matrix and a high number of iterations is chosen to assess network-bound behavior; time is measured for a configuration with between 1 and 10 processes and nodes. Observed and estimated wall-clock time is shown in Figure 7.61a. The critical time which is needed to perform the communication and the final output is given in see Figure 7.61b. This experiment will also demonstrate several timing aspects not mentioned so far.

Observations and interpretation

a) The total run-time decreases from 4 s down to 1 s (look at Figure 7.61a). Measured times vary by up to 0.5 s. In many cases, estimated wall-clock time and the measured timing are alike. The critical time of the configuration takes much less time – it is between 0.1 s and 1 s.

b) The critical time increases from one to three processes (look at Figure 7.61b). This is due to the exchange of halo rows, one process does not have to exchange any halo rows, with two processes one exchange happens, and with more processes processes in the middle exchange data with both neighbors. With more than three processes, the trend of the communication time stays on the same level around 0.45 s. There are several longer running configurations, such as for 3, 5 and 9 processes; time is spent inside the communication. For those configurations the simulator estimates a shorter runtime. Therefore, these configurations must be assessed in detail.

c) Time lost due to synchronization is roughly 0.05 s, which is just a fraction of the total time lost by communication (look at the results of SyncVirtually in Figure 7.61b). Therefore, the impact of late senders or a late receivers can be ignored. The computation load is well balanced and load-imbalance is not the reason for pending communications.

Further, the variance of the critical time of these configurations is much higher than for the other configurations. The measured results vary very much, but simulated time shows little deviation. Thus, in principle computation time of the three runs are alike and they should behave similar. Since the simulator uses a fixed time for the communication and mimics computation time, the additional variability of the measurement must be rooted in the communication or I/O path.

d) A detailed assessment is possible by comparing measured behavior with simulated behavior. In Figure 7.63 and Figure 7.62 screenshots of the runs are provided for 5 clients and 7 clients, respectively. The run for seven clients is well estimated (the critical time of the simulation matches in Figure 7.61b).

Profile and excerpts of the timeline for one run and its replay with PIOsimHD are provided to give evidence to this conclusion: The simulated profile matches the behavior of the measured profile well. Communication is just slightly slower than on the real run (compare Figure 7.62a and Figure 7.62b).

In Figure 7.62e the startup phase of the run is given. This figure also shows the communication pattern of the PDE. It can be seen that the first few `MPI_Sendrecv()` calls take a long time, this is due to the fact that the processes are not started at the same global time by MPICH2 – Rank 6 starts 24 ms later than Rank 1. The simulator starts all processes at the identical time. Although the startup time is treated slightly different, the offset of 24 ms does not have much impact (look at the profiles).

Excerpts of the emerging communication patterns are given in Figure 7.62c and in Figure 7.62d. Not only does a similar communication pattern emerge in the simulation, this pattern also mimics application behavior surprisingly accurately.

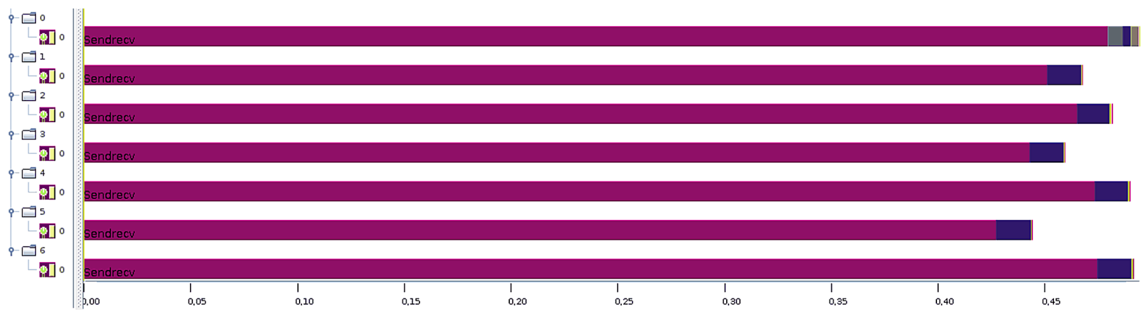
e) With five processes the estimated time is much faster, a trace is visualized in Figure 7.63[32]. In the observation the `MPI_Init()` of Rank 0 happens 0.5 s later than for Rank 1 (see Figure 7.63a). Since the whole run takes only about 2.65 s, the late start of 0.5 s is significant; the simulator starts all processes at the same time (see Figure 7.63b). Consequently, that late initialization explains a large part of the offset seen in Figure 7.63a.

Several other operations take much more time than expected: The Sendrecv operation on the late starter (Rank 0) takes about 0.25 s. It can be also seen that the final output, which are 81 double values of the matrix that are converted to ASCII and written to a file and to standard-out, takes about 0.5 s (look at `IO_fopen()` and `IO_fclose()` at the end of Figure 7.63a). POSIX-I/O is not simulated, but the time of these calls is converted to a regular compute job. Therefore, simulation time of the POSIX I/O matches perfectly.

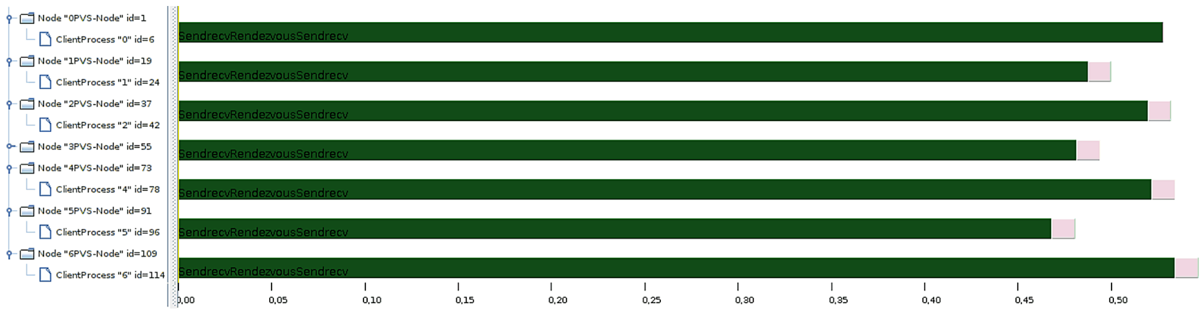
The final barrier takes also at least 0.25 s (look at the `MPI_Barrier()` on the right of Figure 7.63a, Rank 0 is the last process joining the barrier). In the simulation the barrier is not even visible at that zoom level. It seems that network communication of the measurement is somehow slow during startup and finish phase, but the reason is unknown. Without this slowdown, the intermediate communication and computation is well simulated, the main part of the program is estimated to take about 1.2 s.

This example already shows that simulation results are valuable to estimate runtime. Even observed communication patterns and emerging behavior can match very well. However, due to variability of network and I/O, simulated patterns can also differ from the measured pattern. With visual inspection, the discrepancy between simulation and observation can be identified and the cause of the behavior can be understood.

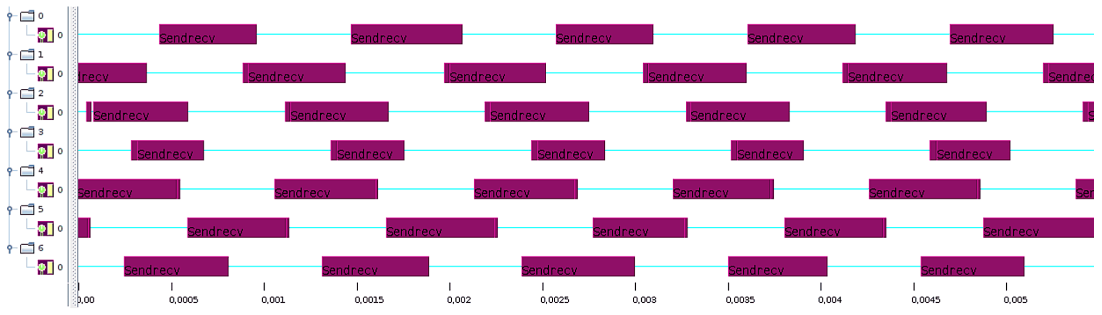
[32]The visualized run is not part of the diagrams in Figure 7.61, to assess the variance further three additional runs have been made. Those runs showed a similar behavior and thus validate the correctness of the presented runs.



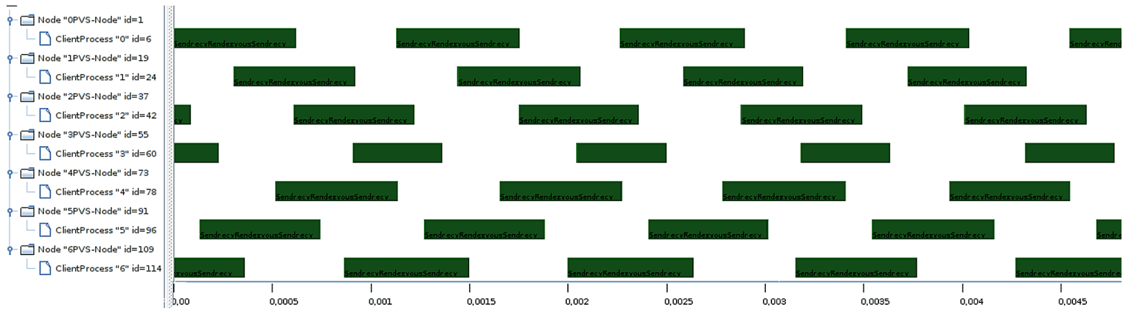
(a) Measured profile.



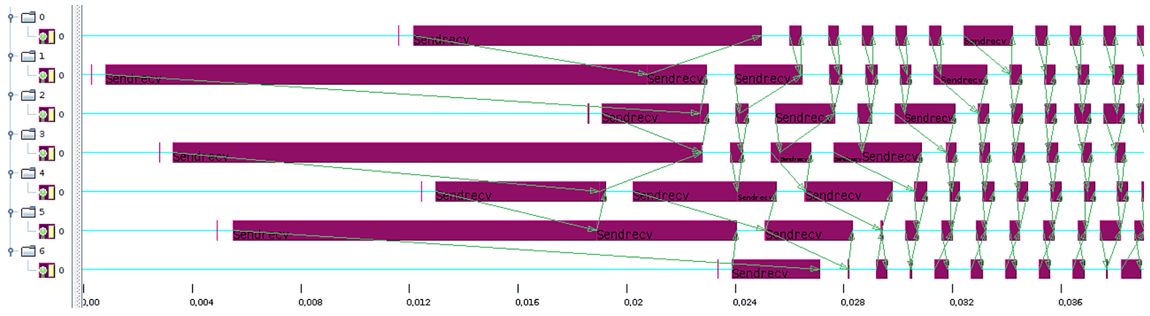
(b) Simulation profile.



(c) Measured timeline – excerpt.

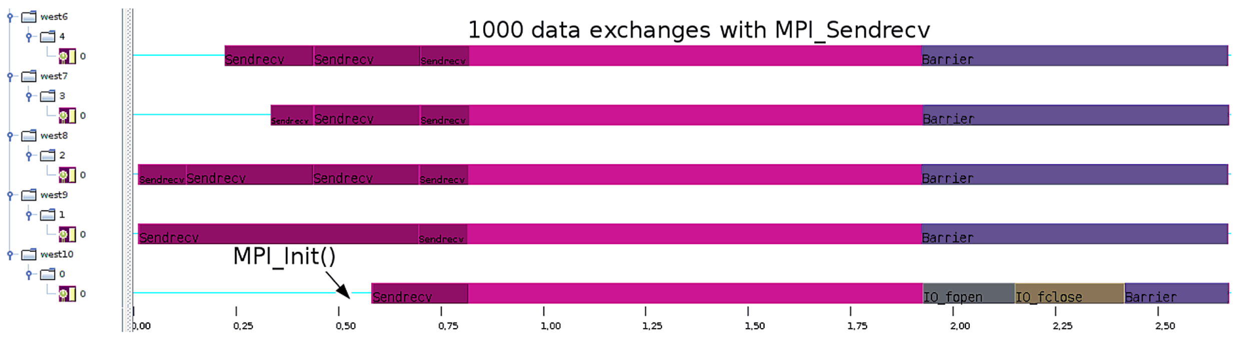


(d) Simulated timeline – excerpt.

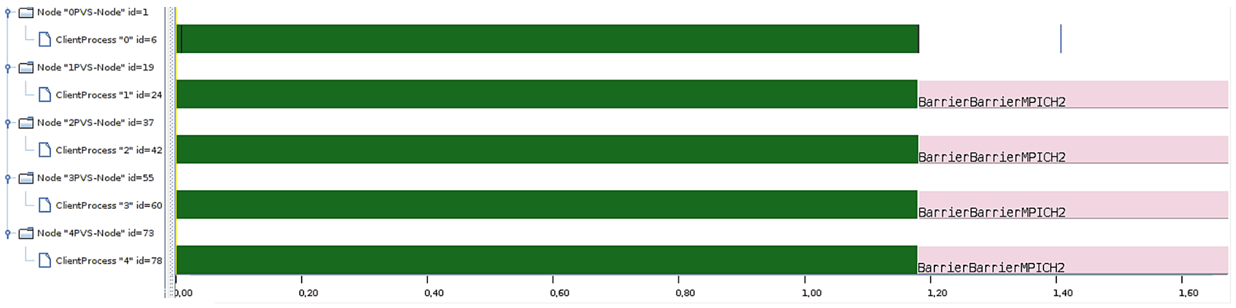


(e) Measured timeline – startup phase.

Figure 7.62.: Visualization of a PDE run for experiment XS-0S – 7 clients. `MPI_Sendrecv()` calls are colored in red for the observation and green for the simulation results.



(a) Measured overview.



(b) Simulated overview.

Figure 7.63.: Visualization of a PDE run for experiment XS-0S – 5 clients.

Although the lazy startup of MPI processes is not modeled, the behavior of these short-running experiments is simulated well. To reduce the impact of the startup phase, the experiments could be designed to run longer. So, while the simulator is capable to assess short runs, further experiments are designed to run several 10s of seconds.

7.9.4. Critical Path Analysis

In this little experiment, the capability to analyze the critical path of an MPI application is discussed. Analysis is done for the XS-0S configuration and the run with seven clients (original traces are visualized in Figure 7.62).

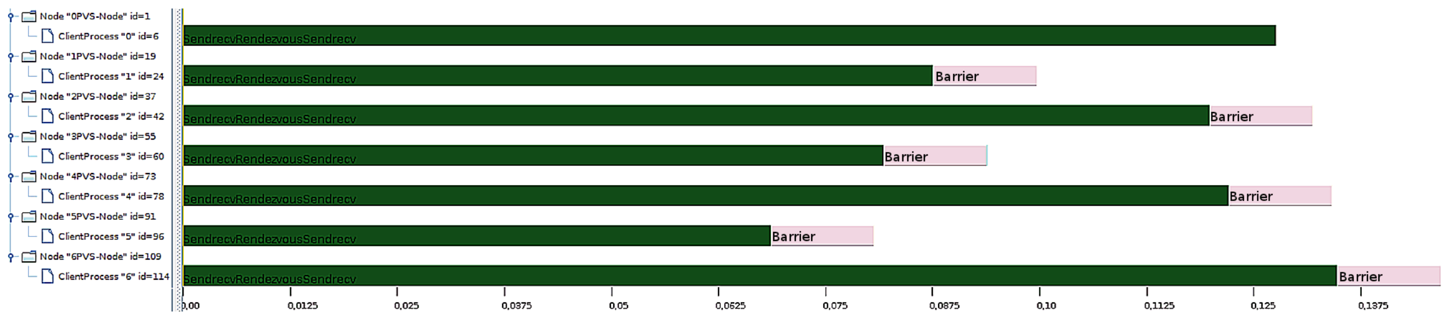
To analyze the critical path, the implementations are chosen from the *SyncVirtually* configuration: An infinite fast network and I/O subsystem allows immediate completion of communication, but enforces restrictions of data dependency. For example, the last process joining a collective operation triggers all other processes to continue immediately. Basically it simulated execution as if operations on the critical path are executed immediately (since all data dependencies are met) and shows the waiting times of all processes under this assumption.

In Figure 7.64 the timeline and profile windows are provided. Communication still need at least one CPU cycle in the simulation, with the cluster model that corresponds to a processing time of 4 ns[33]. With the help of Sunshot all events that do not depend on another event, that means they need 4 ns, are removed by applying a filter. The filtered output is given in Figure 7.64c. Additionally, a zoom into selected iterations of the PDE is shown in Figure 7.65.

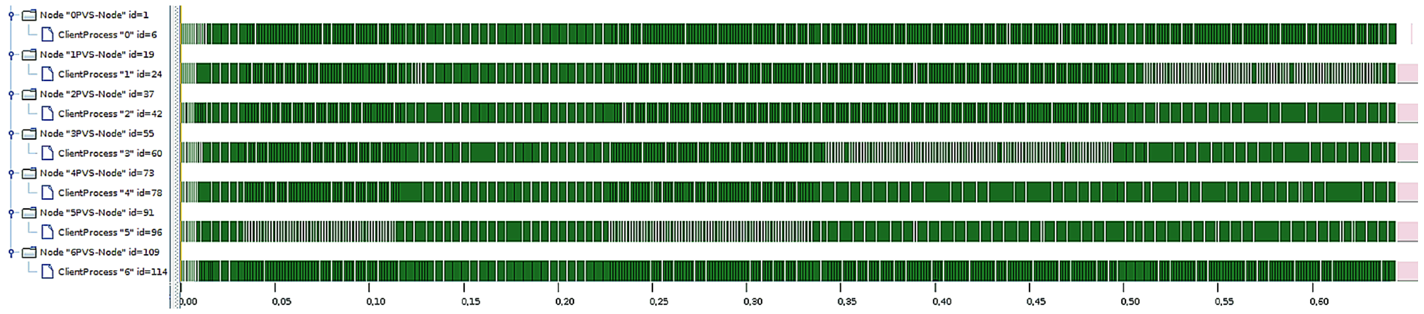
Observations and interpretation

a) The profile in Figure 7.64a reveals how much time would have been spend in stalled communication, if the network were infinitely fast. All times shown are caused by data dependencies – the commu-

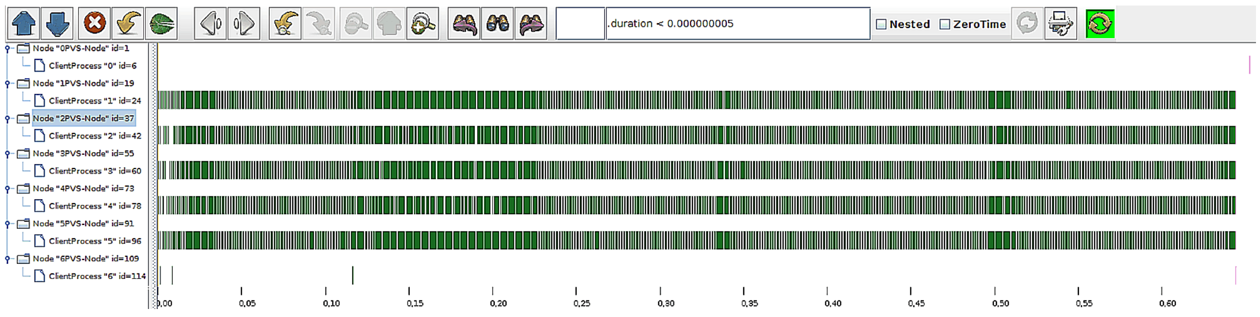
[33]The processors are configured to process instructions effectively at 250 MHz, which is an efficiency of 10% for a 2.5 GHz processor.



(a) Profile.



(b) Timeline.



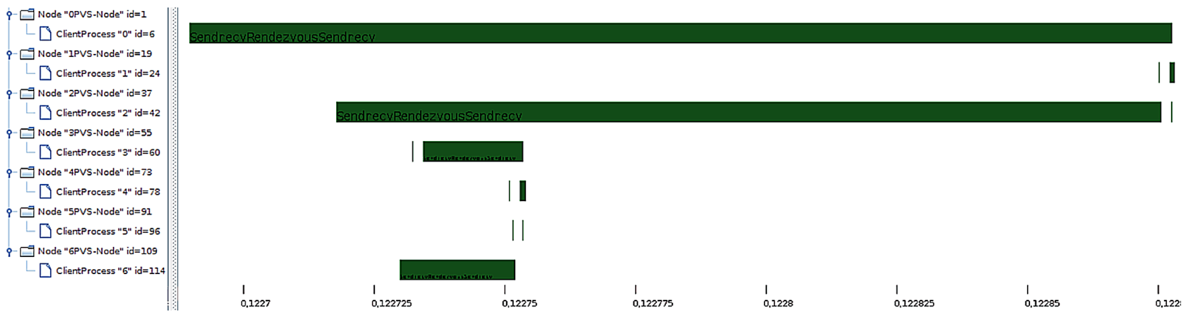
(c) Timeline showing critical activity – non-critical operations are filtered.

Figure 7.64.: Analyzing the critical path and caused waiting times for the experiment XS-0S with seven clients. The original traces and simulation results are provided in Figure 7.62.

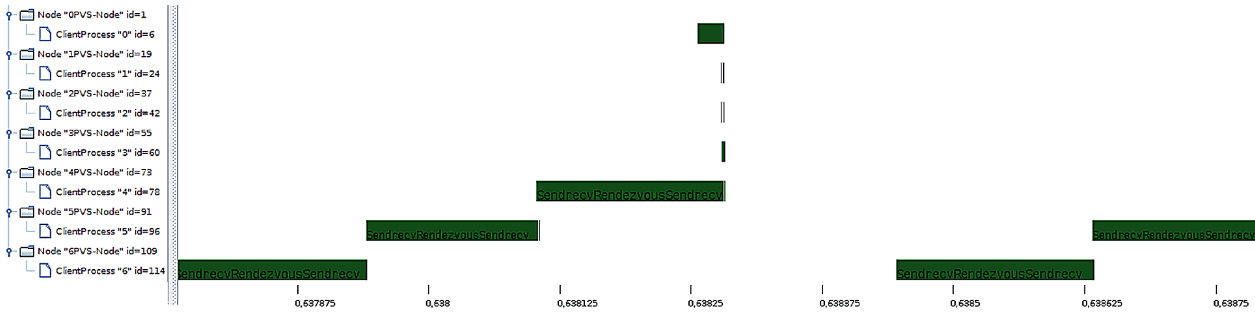
nication partners do not reach the MPI calls at the exact same time. Therefore, a slight imbalance must exist in computation time. The total time lost by inaccurate synchronization due to OS jitter or load-imbalance of the processes is 0.14 s for Rank 6. Theoretically, the last process could have used this amount of time for computing the solution. The total amount of computation time wasted is the sum of the time for all processes. It can be seen that waiting times are not evenly distributed among the processes – Rank 6 spends 60 ms more time in communication than Rank 5, waiting twice as long for its communication partner on overage (which is Rank 5 for the main part of the program).

b) The complete timeline by itself does not look very interesting, besides the long execution time of the barrier the other calls complete quickly. By applying a filter in Sunshot, immediately completing operation become visible – those might be on the critical path. Almost all events from the first and the last timeline vanish but the ones of the intermediate processes observe waiting times (see Figure 7.64c).

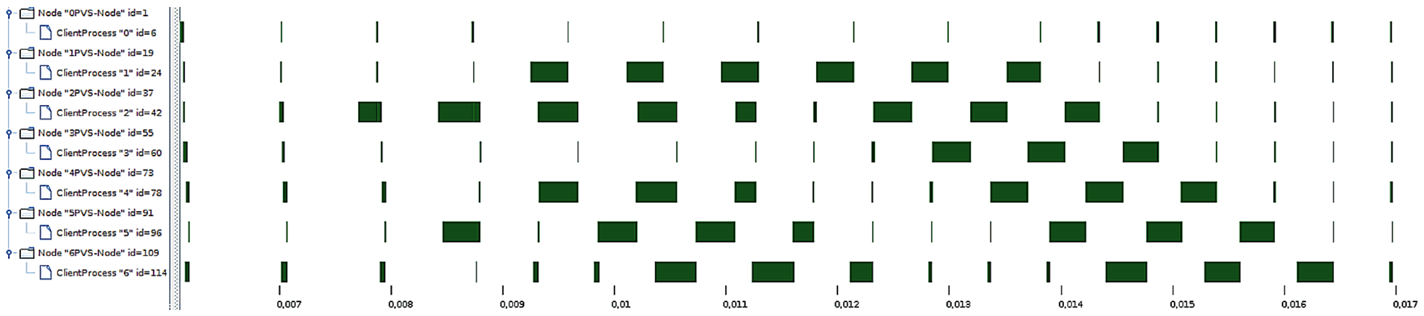
c) To understand the behavior better, zooms of the timelines are inspected (see Figure 7.65). In these screenshots, immediately completing activities are not filtered. Communication of a single iteration results in several types of patterns. In Figure 7.65a the first process waits for Rank 1 which computes for a longer time. Therefore, as long as Rank 1 does not initiate the communication, Rank 0 must wait.



(a) Iteration type A.



(b) Iteration type B.



(c) Multiple iterations at application start.

Figure 7.65.: Analyzing the critical path for the experiment XS-0S with seven clients by zooming into the traces. The original traces and simulation results are provided in Figure 7.62.

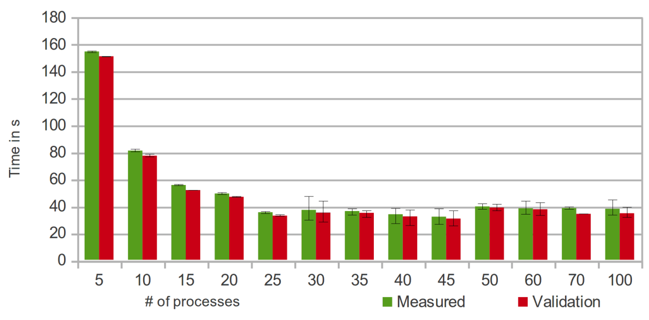
Then Rank 1 exchanges the halo row with Rank 2, here the latter rank is late[34]. Any of the events which complete immediately may be on the critical path (these can be found for Rank 1, Rank 5 and one of Rank 2 and Rank 3). The further execution determines which are on the critical path.

Due to the domain decomposition, the first and the last rank exchange data with just one neighbor, while the other processes exchange halo rows with the previous and the next rank. Between the data exchange of the two rows is a short computation time which varies slightly and thus one of the inner processes synchronizes late. That leads to the pattern observed in Figure 7.64c – intermediate processes are almost always on the critical path.

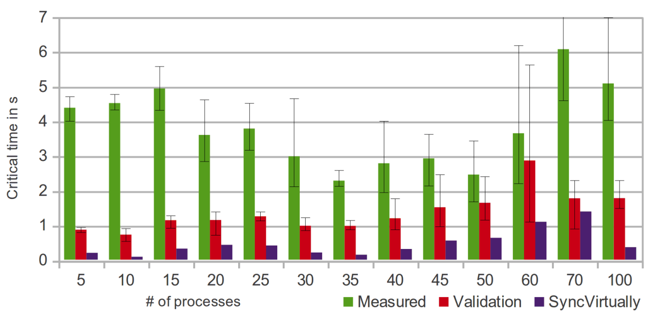
Another pattern is provided in Figure 7.65b. Here a "staircase" pattern emerges due to a late synchronization of three processes. Also, three processes participate in the critical path of the left iteration, longer computation on one of the three processes increases the wait time for all other processes. The events of Rank 3 participate in the critical path of the right iteration.

Due to OS noise, computation time varies slightly and thus the pattern changes over time. For `partdiff-par` the critical path wanders between processes and interesting patterns emerge (see Figure 7.65c).

[34]Note that the selected implementation for `MPI_Sendrecv()` uses the eager protocol for the small amount of data to transfer.



(a) Wall-clock time.



(b) Critical time due to communication.

Figure 7.66.: Times of the computation-bound experiment XL-0S-5CN – a variable number of processes hosted on 5 nodes.

With the help of the filter function, the tolerance level to noise can be adjusted – the amount of waiting time that is tolerable. Those areas are candidates for optimization since more work could be done. As the domain decomposition is rather simple for the PDE solver and the fluctuations are mainly caused by OS noise, there is not much room for improvement of the PDE.

7.9.5. Computation-bound Workload

This section discusses the results obtained for computation-bound workloads – no I/O is performed, yet. Two similar scenarios are discussed: in the *XL-0S-5CN* experiment, the behavior of increasing the number of processes on a fixed number of nodes is evaluated – up to 100 processes are placed on just 5 nodes. In this discussing a way of pretending a "false" accuracy is described, which can be achieved by just comparing the total runtime with the estimated time. In the second *XL-0S* scenario, a variable number of processes is measured in which every process is placed on a dedicated node. The measured times are given in Figure 7.66 and Figure 7.67.

Observations and interpretation

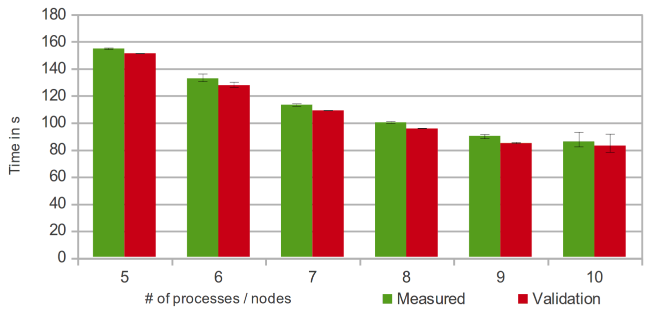
a) The measured wall-clock time and simulated time match very well. As mentioned before, with the current model, simulation of computing time is accurate. Consequently, a computation-bound experiment is expected to be well approximated. For this reason a comparison of total runtime can be used to achieve an arbitrary simulation accuracy and thus fool readers and users.

Using critical time, results can be assessed much better. The critical time reveals that at most 7 s of the total run-time are inflicted by communication overhead (see Figure 7.66b and Figure 7.67b). Therefore, even ignoring communication time would lead to a similar total run-time.

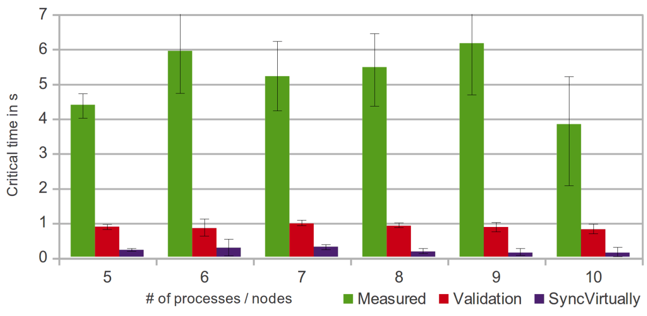
Simulated critical time and observed critical time differ – especially for smaller process numbers the simulator predicts a faster communication, between 1 to 3 s is expected to be spent in communication.

b) For 10 processes both experiments behave similar (compare the wall-clock time and the critical time). Almost linear scaling is achieved for experiment *XL-0S*, the runtime decreases from 155 s to 85 s by doubling the processes (see Figure 7.67). Scalability of the *XL-0S-5CN* scenario is limited – performance does not improve further by using more than 25 processes.

c) Since the critical time is low for both experiments most time is spent inside the computation and thus the experiments are compute-bound. Similar effects can be seen for both scenarios. Therefore, in the following the focus is put on experiment *XL-0S-5C*. By analyzing this experiment the critical time of the other experiment is discussed, too.



(a) Wall-clock time.



(b) Critical time.

Figure 7.67.: Times of computation-bound experiment XL-0S – a variable number of processes each hosted on its own node.

d) In the *XL-0S-5CN* scenario wall-clock time does not improve further when using more than 25 processes (see Figure 7.66a). A node has 12 processing elements (on two processors), thus a performance increase up to 60 processes is expected.

There are several potential reasons, of which a few can be investigated directly by inspecting the simulation results. One theory could be that network overhead increases with the number of processes. However, by inspecting the critical time of 45 and 50 processes, it can be seen that it stays on the similar level while the wall-clock time increases. Consequently, the time for computation must have increased. This could be caused, for example, by a suboptimal usage of the memory hierarchy or utilization of the memory controllers.

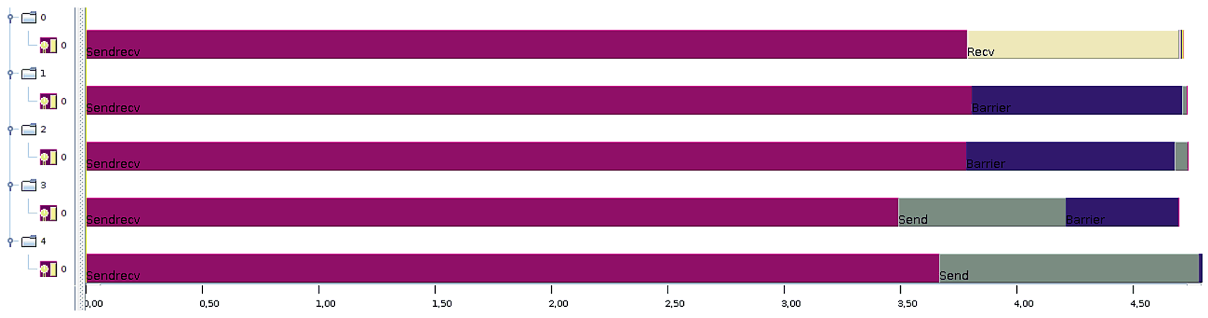
Theoretically, another reason could be that processes perform additional computation, which is not true due to the domain decomposition. However, each process executes certain identical parts of code (some parts of the initialization and finalization, for example); thus, according to *Amdahl's law* speedup is limited by this sequential fraction.

Also, for larger processes number, the physical cores do not suffice to permit exclusive dispatching of a process to a core. Since 60 physical cores are available, the processor must be multiplexed for larger configurations. The observed "additional" computation is inherent to the way computation time is currently determined: HDTrace generates timestamps from the wall-clock timer. Thus, a potential multiplexing of the processor is hidden – computation time is added even if a process is not dispatched. This blends with algorithm specific effects, such as an increase in computation time due to replicated computation and memory utilization.

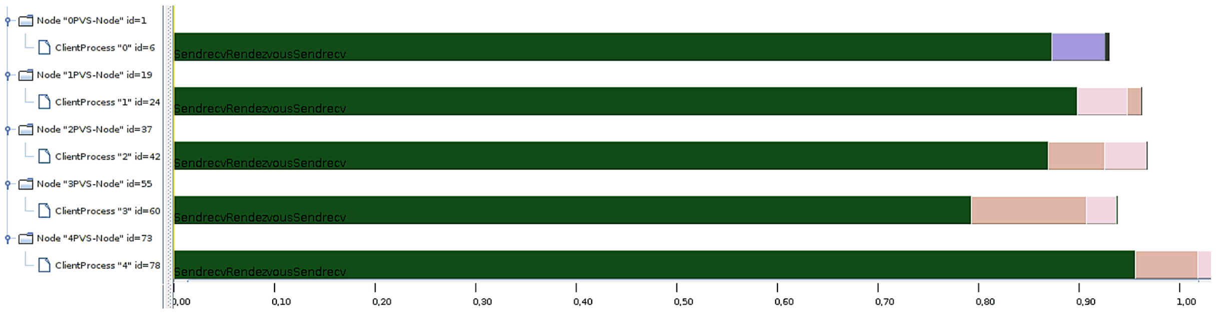
Probably, the amount of required compute time is virtually increased: Spin-locks inside MPICH2 contribute to the overall amount of recorded "compute" time as follows: When more than 60 processes are scheduled they might "compute", i.e., wait for communication of a neighbor which is currently not dispatched by the Linux process scheduler. Since the scheduler is not aware that a process waits for another, a process that is currently stuck in `MPI_Recv()` waits for a process that issues the matching `MPI_Send()`. If the communication partner is currently not dispatched on any of the CPUs, the process burns cycles in the Spin-lock – this processing time is just wasted.

The impact of the reduced performance due to multiplexing the physical cores among the logical processes can be seen by comparing the critical time between 25 and 100 processes; the maximum network overhead increases only by 3 seconds while the wall-clock time stays on the same level. If the algorithm and system would scale, then the wall-clock time for 100 processes would have been approximately one forth of the time for 25 processes and thus about 10 s.

However, since wall-clock time does not improve by increasing process counts from 25 to 45 processes additional computation time must also be spent inside the program – either due to redundant computation – which is unlikely, or by a fully utilized memory controller.



(a) Measured profile.



(b) Simulated profile.

Figure 7.68.: Visualization of a PDE run for experiment XL-0S(-5CN) – profile for 5 client processes.

e) By looking at *SyncVirtually*, the load imbalance between the processes can be assessed. While lower process numbers lead to a small additional waiting time on the critical path, for 60 processes imbalance slows down execution by 1 s. Consequently, one second of runtime is wasted due to synchronization costs, which is only a small fraction of the overall runtime of 30-40 seconds.

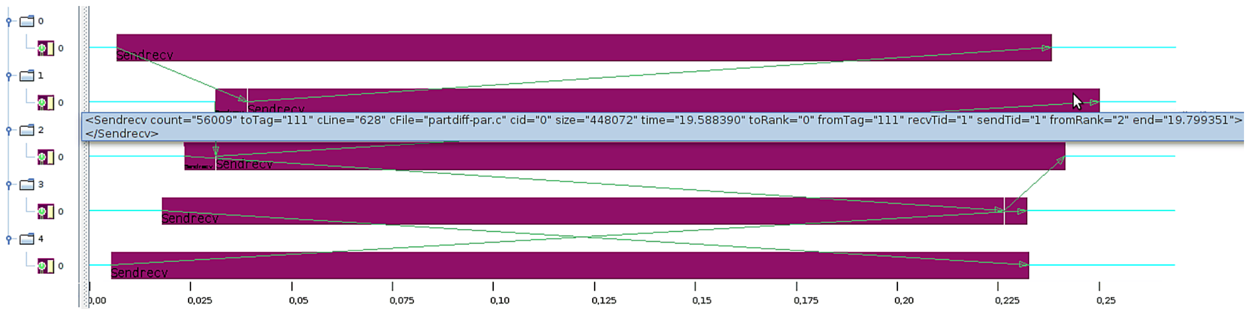
f) The loss due to communication can be derived by subtracting critical time of *SyncVirtually* from *Validation*. It can be seen that at the beginning less than 1 s is needed for communication. Due to the shared memory subsystem that time increases slightly with the number of processes.

g) For several process numbers, the measured runs deviate more than the *Validation* configuration (see the critical time diagram). Deviation of the *Validation* configuration is due to variance in the computation times – this increases when the process count exceeds the number of available cores, because the processor must be multiplexed among all processes. Remember, the communication time and synchronization behavior is identical for simulation and measurement. Consequently, it can be concluded that the difference between the variance of simulation and observed critical time must be caused by fluctuations in the network communication speed. Still, the absolute difference between critical time of simulation and observation must be explained.

h) To understand the divergence between simulation and observation better, profiles are given in Figure 7.68. These diagrams show data for 5 clients, thus they belong to both experiments. It can be seen that on the real system most communication time is spent in `MPI_Sendrecv()`, some time is spent in `MPI_Recv()` and `MPI_Send()`. The latter two point-to-point communication calls are only used to collect the 9 lines of matrix output.

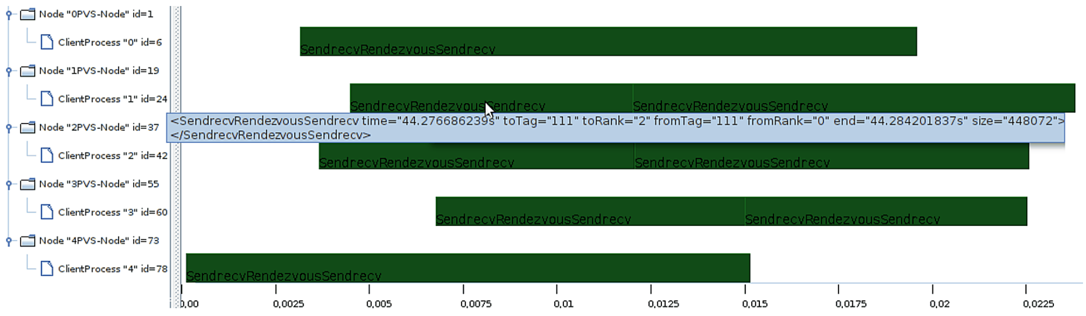
On average simulated calls complete in 20% of the time. Either the simulation model is wrong, or there is an issue on our cluster (or the PDE application) which degrades performance. This calls for more in-depth investigation.

i) Screenshots of a communication phase are provided in Figure 7.69. In the upper two figures, the data exchange of the halo rows is shown for a single iteration. The observed and simulated pattern is different and the time for the data exchange is very different – the simulator predicts about 0.02 s but the measurement takes about 0.2 s. If 40 iterations of this configuration are multiplied by 0.2 s, then it results in the 8 s as shown in the profile. Other instances of `MPI_Sendrecv()` can be found in the



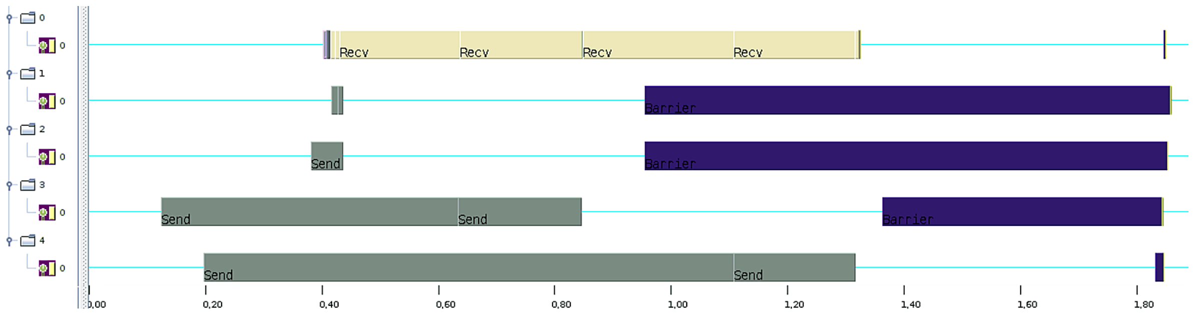
```
<Sendrecv count="56009" toTag="111" cLine="628" cFile="partdiff-par.c" cid="0" size="448072" time="19.588390" toRank="0" fromTag="111" recvTid="1" sendTid="1" fromRank="2" end="19.799351">
</Sendrecv>
```

(a) Measured timeline – single data exchange.

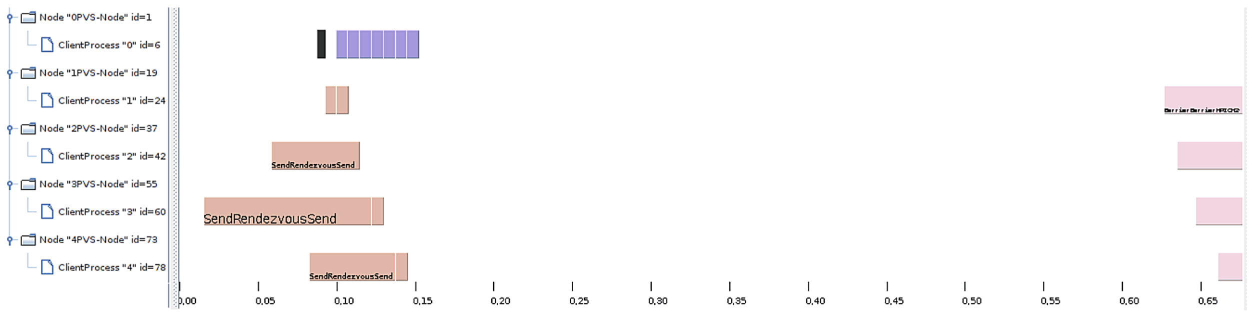


```
<SendrecvRendezvousSendrecv time="44.276686239s" toTag="111" toRank="2" fromTag="111" fromRank="0" end="44.284201837s" size="448072">
</SendrecvRendezvousSendrecv>
```

(b) Simulated timeline – single data exchange.

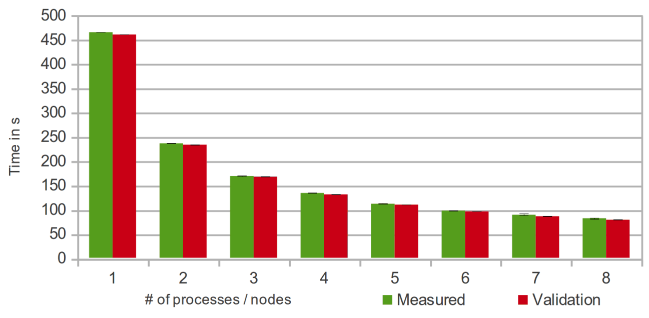


(c) Measured timeline – matrix output phase.

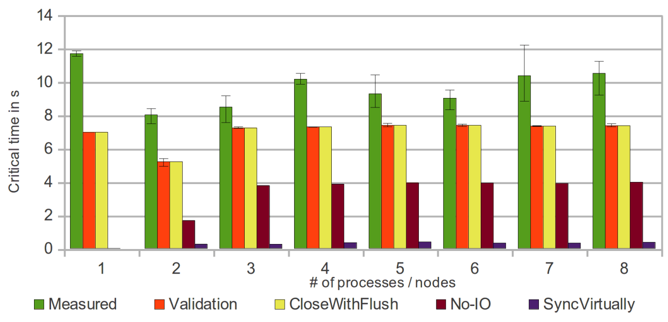


(d) Simulated timeline – matrix output phase.

Figure 7.69.: Visualization of a PDE run for experiment XL-0S(-5CN) – timeline for 5 client processes.



(a) Wall-clock time.



(b) Critical time.

Figure 7.70.: Times of the experiment S-2S-tmpfs – a variable number of processes each hosted on its own node, two dedicated servers store data on tmpfs .

trace in which the receive part takes also 0.2 s after the sender is ready. A row has a size of 400 KiB, consequently on the real system a performance of roughly 2 MiB/s is observable. This is much less than expected, although with this size the rendezvous protocol is active.

The mentioned behavior is also visible during the end phase in which the 9 rows are gathered (see Figure 7.69c). In this case, the simulation and the observation differs to a large extent as well. While a single send operation can take up to 0.2 s in the traced run, the simulation processes them much faster (see Figure 7.69d). In the observation it can also be seen that the sends from Rank 1 proceed quickly, others from Rank 2 and Rank 3 do not. By inspecting the trace it can be checked that all sends are of equal size, and thus the message transfer should require a similar amount of time because all processes are on disjoint nodes – this is also the result of the simulation. However, matching receives of the later rows are already posted by Rank 0 when the sender becomes ready; thus the later transfers of Rank 2 and Rank 3 are expected to be done as fast as the first two receives. Since the other processes are computing there shouldn't be network collisions that degrade performance. The exact reason is unknown.

The degraded performance on the real system explains the difference in the critical time diagram. Although, the critical time diagrams do not match, the simulation results seem to be correct for a healthy system: A few data exchanges such as the one between Rank 1 and Rank 0 in Figure 7.69c happen in 0.01 s which matches the time predicted by the simulation – this time corresponds to an observable network throughput of about 40 MiB/s.

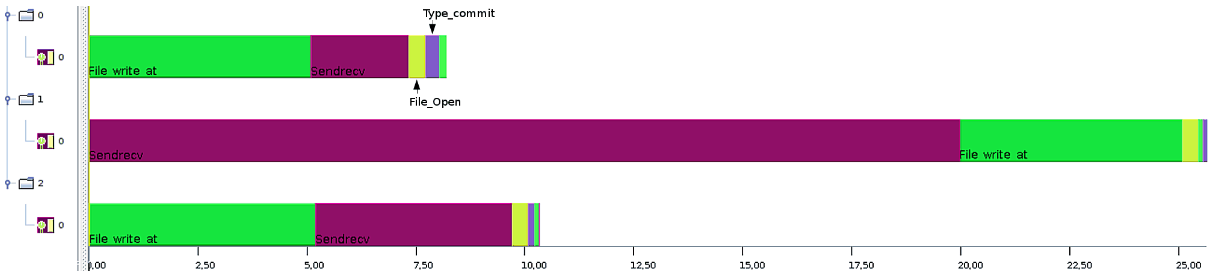
7.9.6. In-memory I/O

To assess the general simulated I/O behavior, servers store data on tmpfs in this experiments, and thus no overhead due to the block device are experienced. In experiment *S-2S-tmpfs*, the number of processes and nodes is varied while the number of servers is fixed to 2 – results are provided in Figure 7.70. The critical time for various simulated I/O variants is also given.

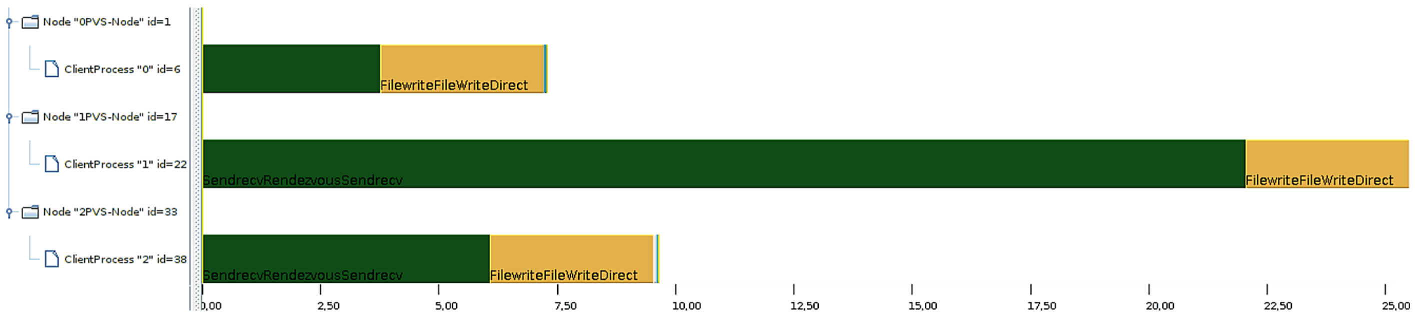
Observations and interpretation

a) The measured critical time is at most 12 s, which is just a fraction of total run-time (look at the measured data). Critical time varies only slightly between the configurations, this means the communication and I/O overhead is independent of the number of participating processes.

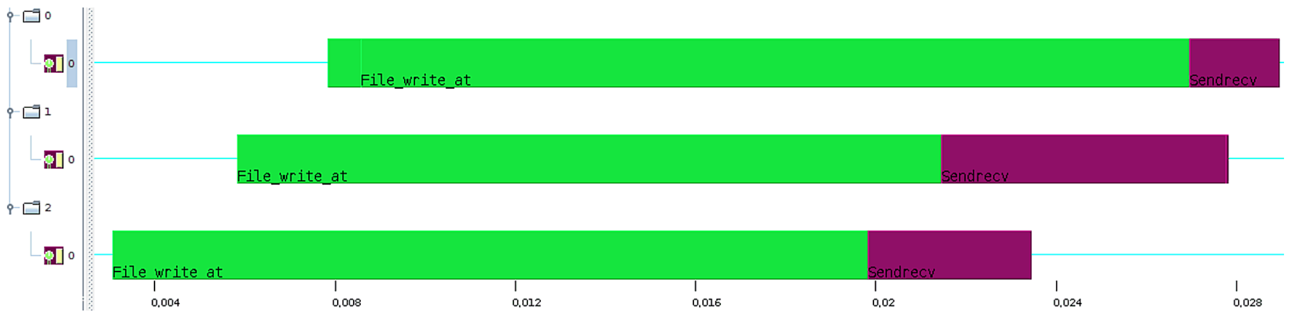
b) Time of the validation replay underestimates the measured critical time by about 2 s. However, there is little noticeable fluctuation in the simulation, while the measurement varies by 3 s. Overall the load is well balanced (*SyncVirtually* is low in Figure 7.70b). The time loss due to communication (delta between *No-IO* and *SyncVirtually*) is (obviously) zero for a single process, 2 s for 2 processes



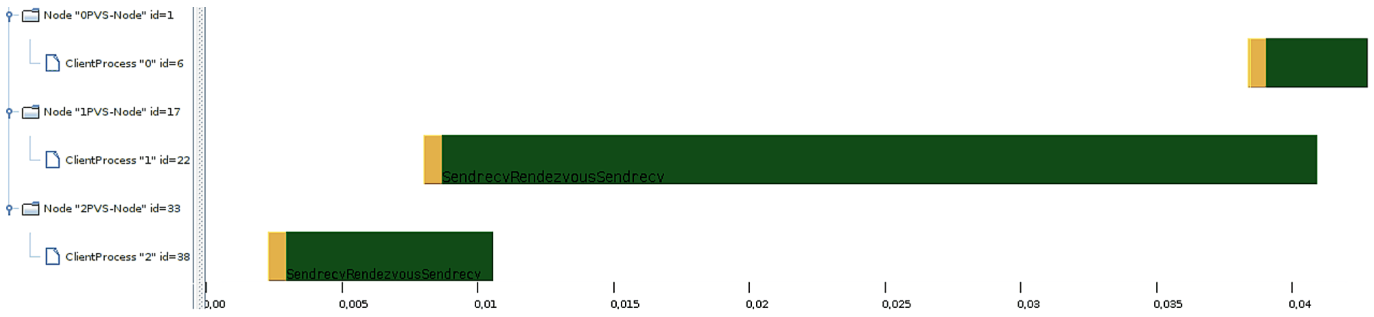
(a) Measured profile.



(b) Simulated profile.



(c) Measured timeline – writing out visualization output.



(d) Simulated timeline – writing out visualization output.

Figure 7.71.: Visualization of a PDE run for experiment S-2S-tmpfs – three clients.

and 4 s for all other configurations. For three processes, the intermediate process must exchange halos with both processes, which explains the additional costs for configurations with more than two processes.

c) The amount of time lost by I/O is about 3.5 s for more than one process, but 7 s for a single process (look at the difference between *Validation* and *No-IO*). The simple explanation is that a single client cannot utilize the available network bandwidth of the two servers. Thus, for a single client I/O takes roughly twice the time. In the measurement the difference is not so big. This is caused by the suboptimal performance of writing visualization data – which is explained later.

d) With the variation in the semantics of `MPI_File_close()`, the critical time does not increase noticeably. Since data is written to a model of tmpfs, the flush should not take much time, which is observed. The only difference is due to the fact that in the current flush implementation all processes synchronize with a barrier. Thus, the slowest client determines pace of the collective operation. In the experiment, all clients have to synchronize at the end of an iteration anyway, therefore, the difference is marginal.

e) To assess the discrepancy of up to 3 s between simulation and measurement, the runs are evaluated with Sunshot – generated screenshots are shown in Figure 7.71. The measured and the simulated profile show a similar behavior (compare the first two figures). However, in the measurement the `MPI_Type_commit()` needs a noticeable amount of time (0.3 s for three calls), since it is not simulated it does not appear in the simulated profile. In the simulation the command is just mapped to a compute job, thus its synchronizing behavior is lost. A simple simulation could be to invoke an `MPI_Barrier()` and thus at least processes would be synchronized in simulation (but this is not done, yet).

Opening a file takes also a considerable amount of time (0.37 s for two calls). In the simulation, opening a file takes almost no time because it is simulated by a broadcast operation from the first process. This latter process is also done by PVFS, but additional metadata operations are involved in a real system. The simulated profile shows slightly more time for point-to-point communication, however, file I/O is quicker in the simulation.

f) As it turns out, writing the matrix diagonal takes much longer on the real system than anticipated (compare Figure 7.71c and Figure 7.71d).

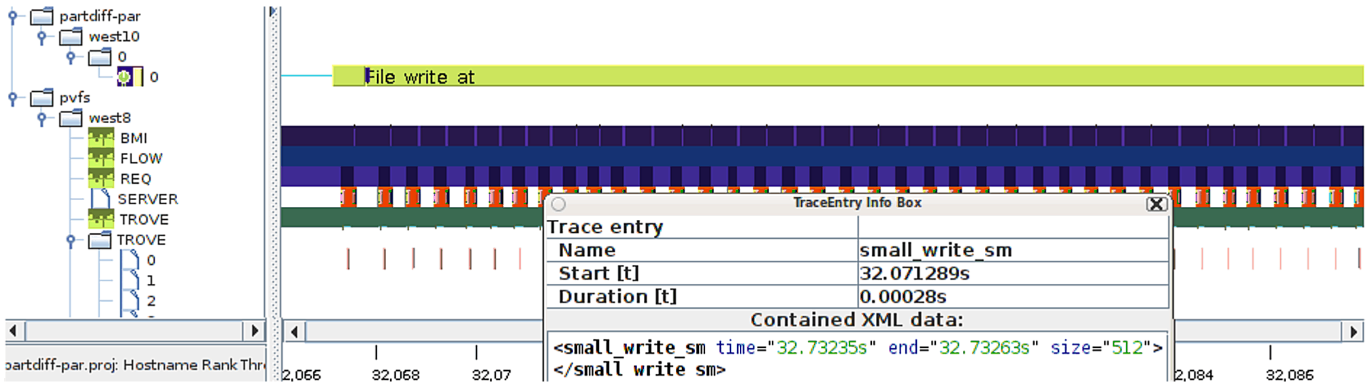
I/O also leads to additional waiting time during halo exchange; while the time spent in I/O is well balanced among all processes, waiting time for Rank 1 is much higher because its communication partners are not ready. Since this process synchronizes with both other processes fluctuations in any of them lead to additional waiting time.

g) To debug the issue with slow I/O further, a run for a single client and server is traced with HDTrace. Since this issue happens for all configurations, discussion can be reduced to this configuration. The recorded times for output during the "visualization" phase are visualized in Figure 7.72a. Output of such a single phase, that means the matrix diagonal, corresponds to 64 KiB of data.

In the figure the server-side activity can be assessed in detail – every layer has events and/or statistics associated with the activities performed by the layer. The number of concurrent operations of every particular PVFS layer is given as a statistic – at each point in time the value accurately represents the number of pending operations. In brief, the figure shows[35]: *BMI* – the network activity, *FLOW* – the regular I/O operations (very small operations are not included), *REQ* are the number of outstanding requests. The *SERVER* timeline shows the pending statemachines and the processing of each statemachine. *TROVE* indicates the number of concurrent I/O operations. Each individual I/O operation which is issued by Trove is recorded including offset and size.

If we look at the process of writing the diagonal, the statistics timelines for BMI and REQ reveal that many operations are performed. By inspecting these requests on the *Server* timeline, each operation

[35]For further information about the layers of PVFS refer to Section 2.1.3.



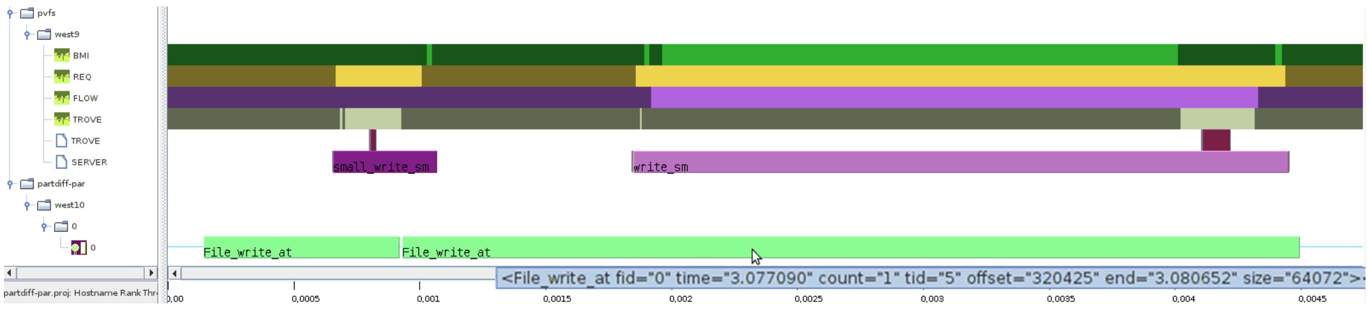
TraceEntry Info Box

Trace entry	
Name	small_write_sm
Start [t]	32.071289s
Duration [t]	0.00028s

Contained XML data:

```
<small_write_sm time="32.73235s" end="32.73263s" size="512">
</small_write_sm>
```

(a) Data exchange.



```
<File_write_at fid="0" time="3.077090" count="1" tid="5" offset="320425" end="3.080652" size="64072">
```

(b) Data exchange with supplied hints.

Figure 7.72.: Visualization of a PDE run for one client and one server.

turns out to be a small I/O operations. In total, the visualization output is about 64 KiB of data; writing generates 125 small requests with an aggregated size of 512 bytes and one with 72 bytes. By checking the executed operations on Trove, a sequential access pattern can be seen. This large number of small requests is the cause of the degraded performance in the measurement – in the simulation a single request is started for the whole data, thereby simulation achieves much better performance.

The reason for the observed pattern is the handling of non-contiguous datatypes by ROMIO. Since ROMIO does not use an additional buffer to store data, every non-contiguous region in memory is normally accessed with an individual operation. PVFS allows encapsulating a list of up to 64 non-contiguous operations with one request, this feature is enabled in the measurement to transfer the matrix diagonals. Therefore, about $64 \cdot 8 = 512$ byte of data is transferred per request which matches our observation.

h) PVFS is also aware of memory and file datatypes. However, this feature must be explicitly enabled with undocumented hints like *romio_pvfs2_listio_write*. When this hint is enabled, only one I/O request is issued to the server (look at Figure 7.72b). This in turn reduces the time to write out the matrix diagonal in the configuration for a single client from an average of 69 ms down to 3.4 ms.

If the 20 ms for writing the diagonal collapses to a fraction such as 2 ms, then the executed 100 visualization iterations save roughly 2 s of time. This is approximately the difference between simulation and measurement. Therefore, the simulator approximates the intended behavior of I/O well.

Furthermore, with the help of the simulator, the performance potential could be identified. By inspecting traces of PVFS client and server, the reasons could be revealed and, ultimately, this knowledge allowed us to debug this issue and achieve the theoretically possible performance.

7.9.7. I/O-Bound Workload

Three I/O-bound experiments are discussed: *XL-5C*, *XL-O* and *M-O-s1000M*.

In the first experiment, a fixed number of clients store data on a variable number of servers, the memory is not limited (the measured times are shown in Figure 7.73). *XL-O* is a variant in which clients and servers are placed on the same nodes. Due to the shared memory, the amount of cache increases with the number of servers.

In the XL experiments a single matrix has a size of 24 GiB, and the matrix diagonal roughly 440 KiB. Two matrices are kept in memory by the PDE and each node has about 12 GiB of memory, therefore, for the configuration of 5 clients/servers about 2 GiB of cache is available per server. The matrix is written out 8 times and the diagonal every iteration, but only 16 MiB of data is stored for the matrix diagonal in total. Times of this experiment are given in Figure 7.74.

In the experiment *M-O-s1000M*, clients and servers are overlapped as well. However, memory is limited for all servers to provide in total approximately 1 GiB of cache. Thus, with a matrix size of 1950 MiB about 50% of the matrix can be kept in the aggregated server caches. Results of this experiment are presented in Figure 7.75.

Observations and interpretation

a) The critical time of all three experiments is in the order of the wall-clock time. Further, if the I/O is not simulated, then the critical time decreases to a very low value. Consequently, all experiments are limited by I/O and computation time plays a minor role. The simulation configuration which just synchronizes (*SyncVirtually*) is not given in the diagrams, because its critical time is even less than the results of *No-I/O*, which is almost 0.

b) Overall, the run-time is well estimated by the simulator. In general, the *Validation* configuration achieves a run-time and critical time that is 10-20% lower than measured. A fluctuation is barely visible in the simulated results (compare the wall-clock times). Since data cannot be cached completely in those experiments, the implemented caching strategy suffices to represent the real system – although the strategy differs from the caching behavior on the real system.

c) The implementations selected in *CloseWithFlush* lead to a higher critical time than the validation runs. Thus, the costs of flushing the file becomes apparent. Predicted critical time for flushing is similar to the measured times for experiment *XL-5C* and higher for experiment *XL-O*. Due to the caching behavior of Linux, server-sided I/O could be stalled leading to a synchronous behavior similar to the one estimate of the CloseWithFlush simulation[36].

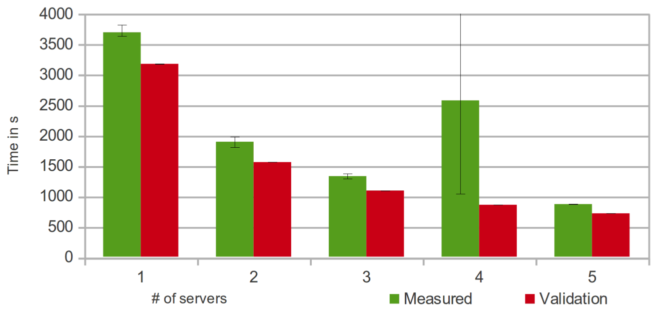
d) In the experiment with a smaller matrix, the simulator predicts a faster execution than measured (look at Figure 7.75). Further, the simulation in which data is flushed to disk is faster than the measurement. A single client is an exception, here the simulated flush achieves exactly the runtime of the observed performance. This discrepancy seems to high, therefore, it must be analyzed.

e) To validate the simulation run, traces are generated and evaluated for experiment *XL-O*. Screenshots of the profiles and timelines for a run with 5 clients are given in Figure 7.76.

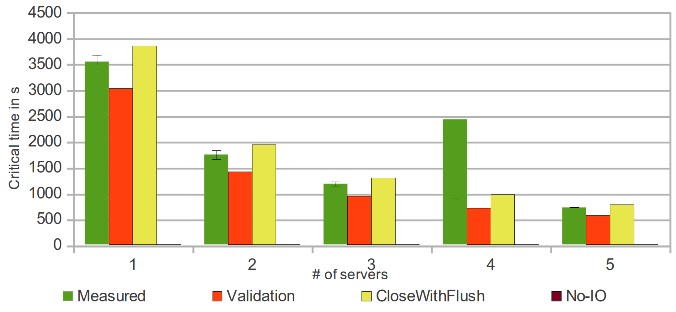
In the measured profile, the last process takes about 520 s while the other processes need 700 s (see Figure 7.76a). Since all processes write the same amount of data, the I/O scheduling of the servers preferred operations from the last process. The simulation predicts 540 s for all processes (look at Figure 7.76b). This imbalance could lead to the difference in the graph.

f) The timeline in Figure 7.76c shows the processing in detail. Processes start at the same time to write their checkpoint data, but the last process finishes earlier. This stalls data exchange between Rank 4 and Rank 3, because the latter is not ready to transfer matrix data. The amount of data written per process is about 4.8 GiB which is larger than the I/O limit of MPICH2 of 2 GiB, therefore, three `MPI_File_write_at()` calls are needed.

[36]This is just true for this experiment, because the checkpoint is closed directly after it has been written.

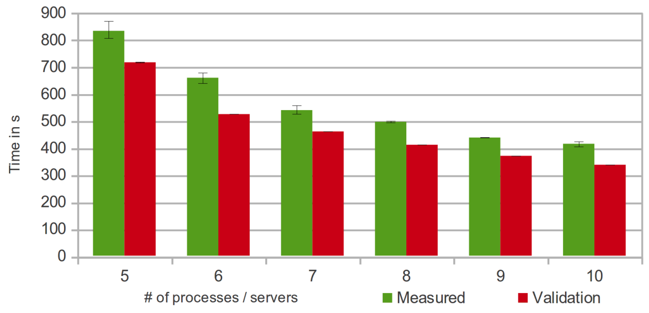


(a) Wall-clock time.

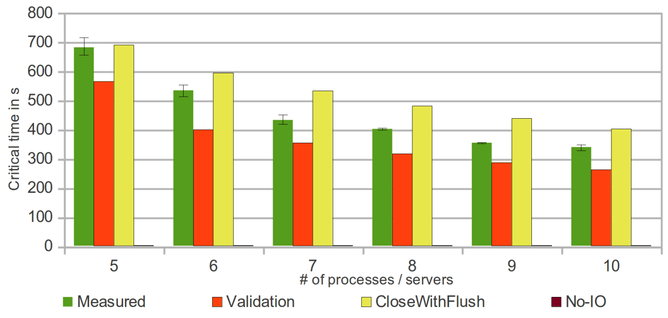


(b) Critical time.

Figure 7.73.: Times of the I/O-bound experiment XL-5C – a variable number of servers storing data for 5 clients hosted on 5 nodes.

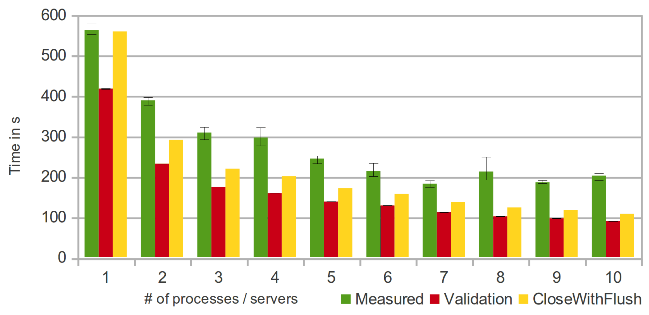


(a) Wall-clock time.

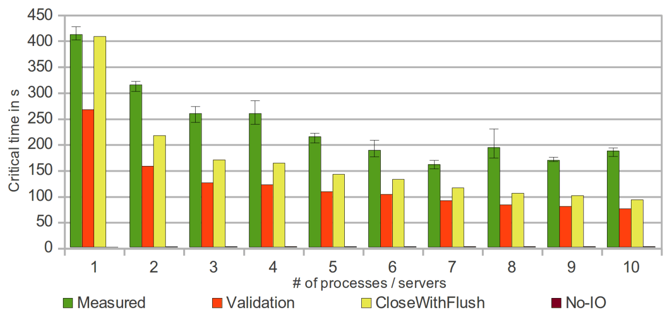


(b) Critical time.

Figure 7.74.: Times of the I/O-bound experiment XL-O – a variable number of clients and servers.

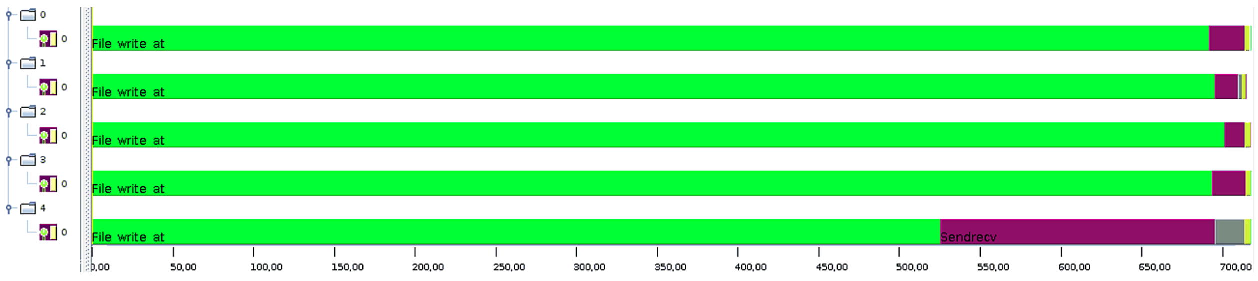


(a) Wall-clock time.

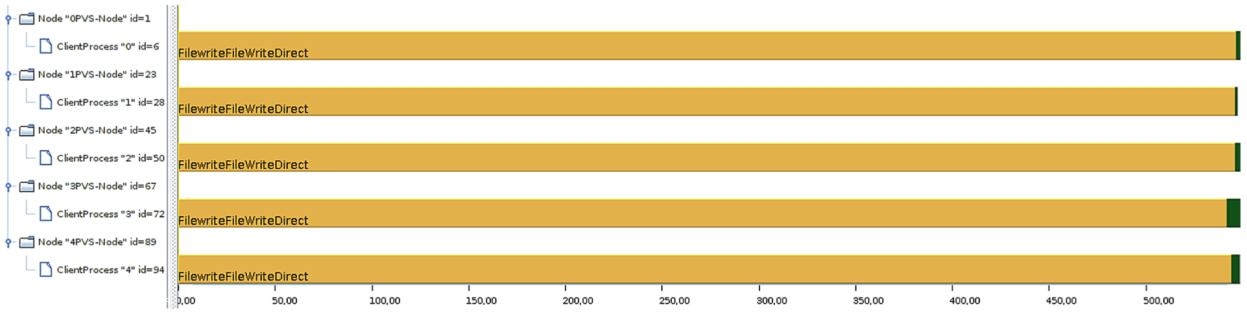


(b) Critical time.

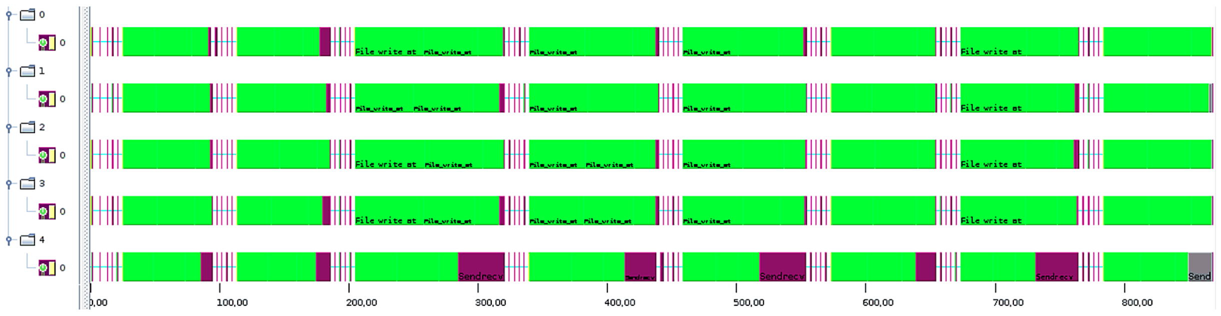
Figure 7.75.: Times of the I/O-bound experiment M-O-s1000M – a variable number of client and servers, approximately 1 GiB of cache is available for all servers.



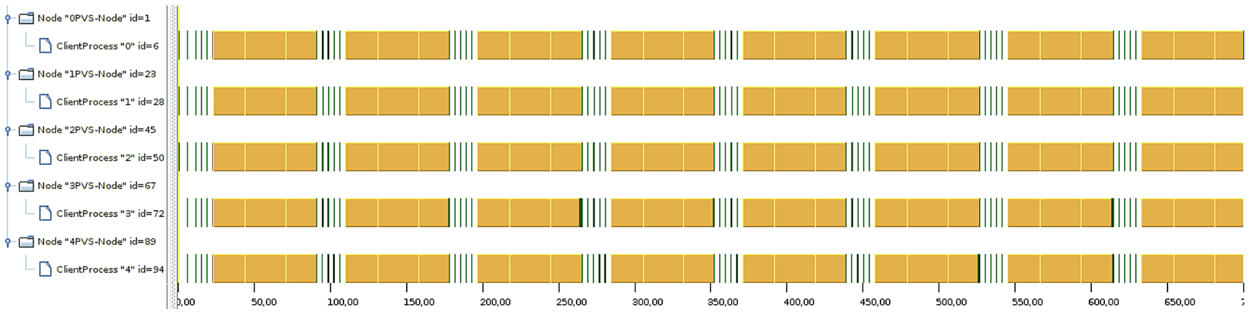
(a) Measured profile.



(b) Simulated profile.



(c) Measured timeline.



(d) Simulated timeline.

Figure 7.76.: Visualization of a PDE run for experiment XL-O – five clients.

Over the whole application run the time needed to checkpoint the matrix varies. At the beginning about 70 s are needed, later up to 110 s can be required. This is probably due to the Linux scheduler and the way PVFS schedules operations; since the cache does not suffice to hold all data, the requests of all clients compete for the shared disk resource. This could degrade the sequential access pattern to a random pattern.

In the simulation this fluctuation is not visible, every checkpoints needs about 68 s. On the one hand, data can be written back in the computation phases between the checkpointing. On the other hand, the simulated I/O scheduler works differently and optimizes for throughput. Surprisingly, this leads to a fair selection of the I/O jobs and almost identical completion times of the individual I/O.

g) To understand the access pattern better, the simulated server processing is visualized for the flush-on-close experiment (see Figure 7.77). The figure contains timelines for the client requests and the disk activity; the purple color is used for flush requests. On the timeline for disk activity blue encodes an average seek and yellow indicates that no seek is required, i.e., the file is written sequentially. In the overview figure the additional overhead caused by flushing the file to the disk becomes visible – the lime green activity after the write operations is the collective close operation which enforces a data flush on all servers.

On every server, one request is processed quickly, this is the request from the client which is located on the same node. Although the local operation is quicker, this process must wait for completion of the requests to other servers.

h) An excerpt of a single checkpoint iteration is given in Figure 7.77b. A variability in efficiency of the disk scheduler becomes visible between the servers – the disk of some servers shows more yellow (no seeks required) than others. Also, the first two servers finish earlier than the last server.

Every client writes about 4.8 GiB of data. Therefore, the 86 s measured for writing data to disk corresponds to an average throughput of 57 MiB/s per server. Consequently, with the interleaving of processes a complex pattern emerged. This complex pattern and the resulting local access pattern for the servers is probably the reason for suboptimal performance of PVFS. By tracing PVFS client and server activity, this could be analyzed in detail, but this is out of the scope for this thesis.

7.9.8. Mixed and Synchronization-bound Workload

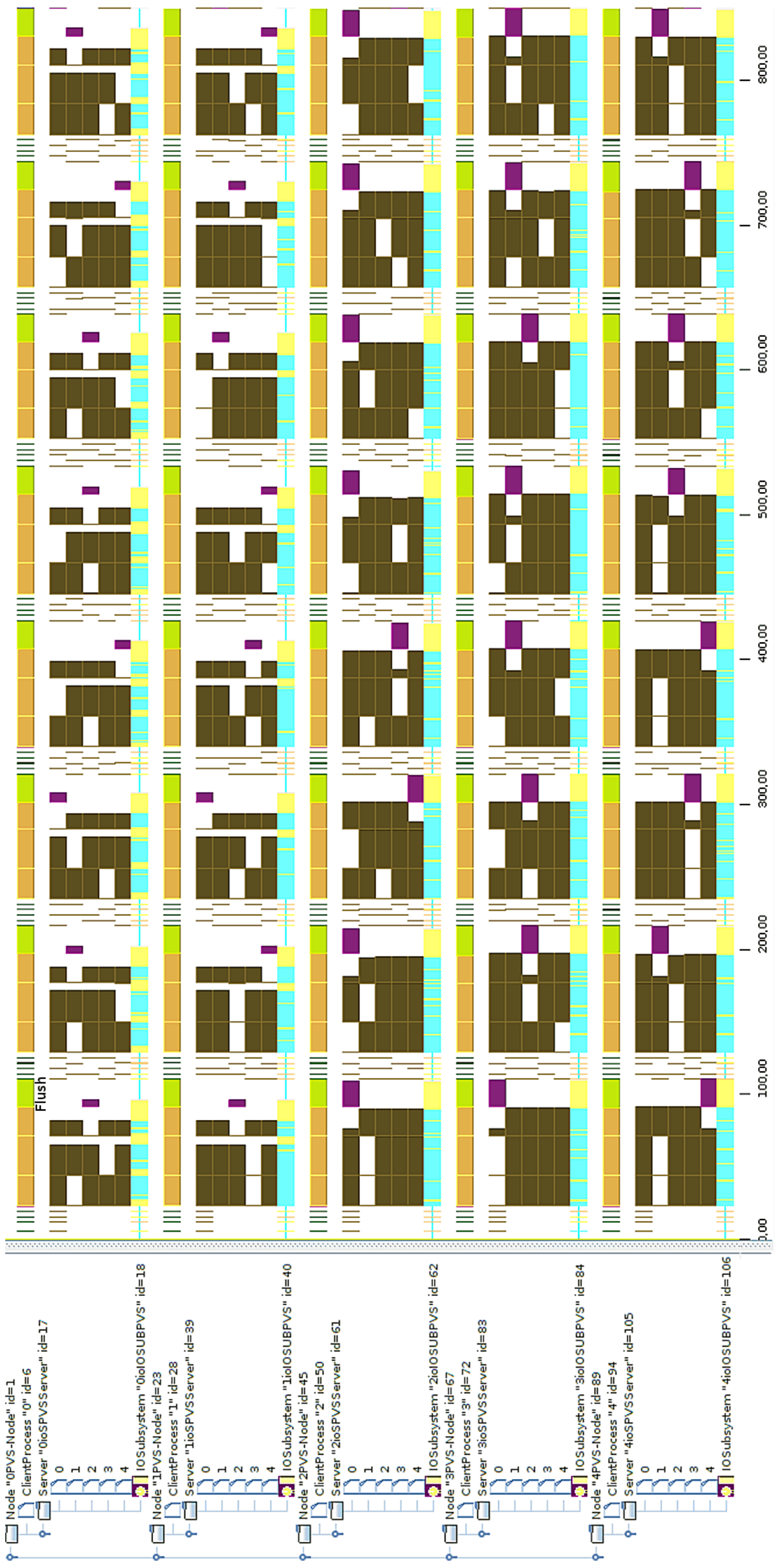
So far performance of the experiment was limited by a single resource, either by computation, communication or I/O. With this workload a larger fraction of time is spent in computation, I/O and in load imbalance between the processes. In the experiment *S-2S-1CN-150M* up to 26 clients are placed on a single node; two servers store data.

The matrix has a size of 488 MiB, and one checkpoint and 100 visualizations are written out. Each visualization output has a size of 64 KiB. Server memory is limited to 150 MiB, and thus checkpoint data must be written to disk. The times are given in Figure 7.78.

Observations and interpretation

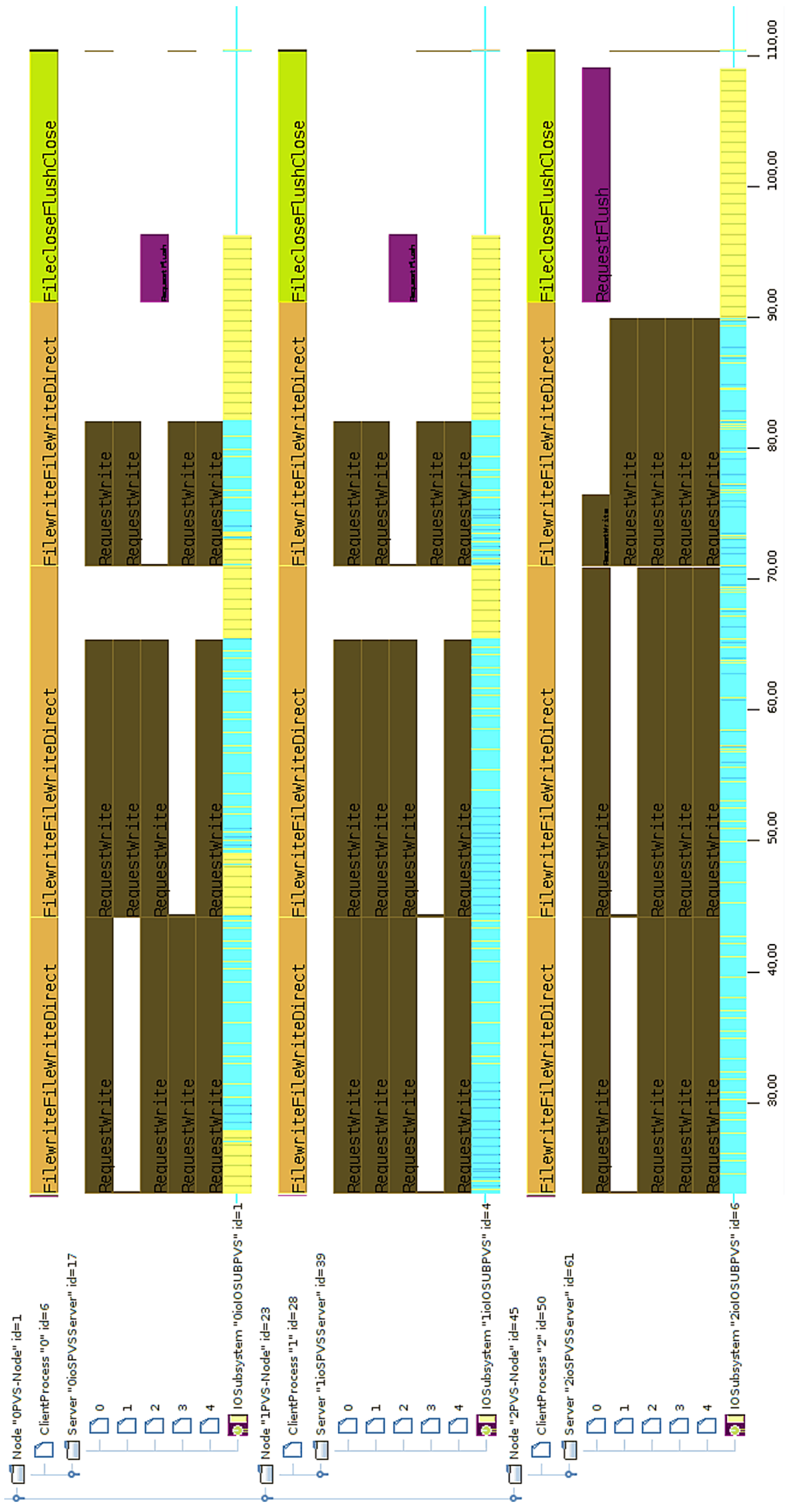
a) For individual runs the critical time varies between 12 s and 50 s with an average of about 30 s. Scaling of this small problem is very limited, minimum observed run-time is about 100 s, which is already reached by 6 processes – 12 cores are available in our dual socket servers. Therefore, starting with 6 processes the critical time increases to about 30% of the total run-time (look at Figure 7.78b). Thus, run-time is distributed into time for computation, synchronization and I/O (see Figure 7.78c).

b) For a single process, the I/O operations take 12 s which is the critical-time (no communication is required for a single process). The simulation estimates just 7 s. For other process numbers the simulated time is below, too.



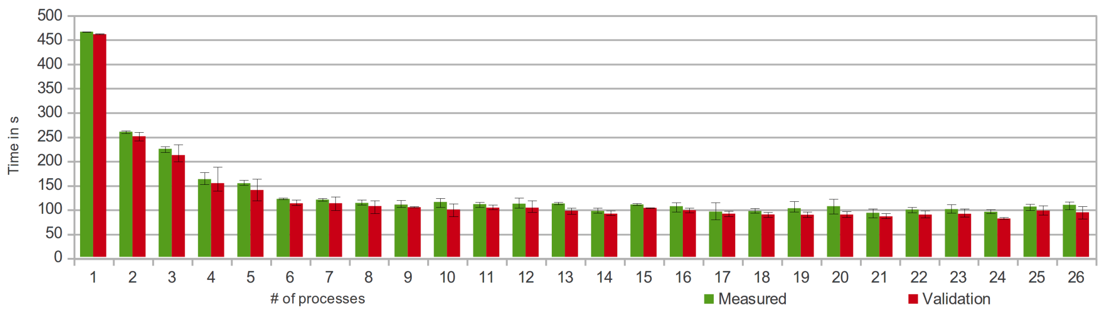
(a) Overview.

Figure 7.77.: Simulated client and server activity for experiment XL-O with flush-on-close semantics – five clients.

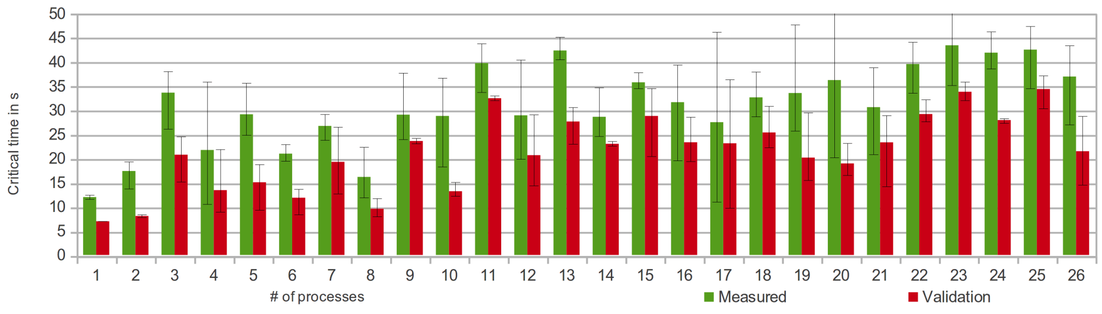


(b) Excerpt of one iteration.

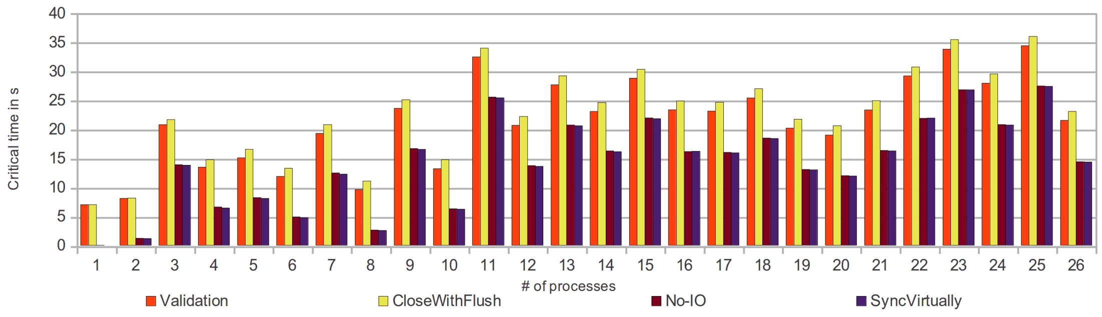
Figure 7.77.: Simulated client and server activity for experiment XL-O with flush-on-close semantics – five clients.



(a) Wall-clock time.



(b) Critical time.



(c) Critical time – comparison of several simulator configurations.

Figure 7.78.: Times of experiment S-2S-1CN-150M – a variable number of processes hosted on one node.

The time for the I/O is determined by looking at the trace files: For a single process, the checkpointing needs about 5.5 s and writing the data for the visualization takes 6.2 s for all 100 iterations. In the simulated results, 6.8 s and 0.2 s are needed for checkpointing and visualization, respectively. Thus, on average, writing the matrix diagonal on the real system is about 62 ms which is much longer than estimated by the simulator. The discrepancy is caused by the large number of requests created in the real system as reported in Section 7.9.6. Therefore, simulated results for a single process are close together. In fact the slow visualization steps reduce observable performance for all number of processes.

c) In Figure 7.78c the critical time for the other simulator configurations is provided which allows assessing these experiments. PIOsimHD estimates that the amount of time spend in I/O is invariant with the number of processes (subtract the duration of *No-IO* from *Validation*). The changed semantics of *CloseWithFlush* implies a small overhead, and with more than two processes it behaves similar and costs about 1.5 s.

Since the system is shared, the scaling is limited – with larger number of processes, the memory controller is utilized and communication overhead increases due to additional halo exchanges. Additionally, the available 12 cores limit the number of concurrently running processes (see Section 7.9.5 for a description of the reasons). While this effect is caused by different factors, it cannot be distinguished from the constant runtime between 6 and 12 processes.

d) The actual communication time is very small, as the low difference between *No-IO* and *SyncVirtually* indicates. This is expected because processes communicate just within a single node. The *SyncVirtually*, however, demonstrates an imbalance in the workload. It turns out that up to 20% of the runtime is spent to wait for synchronization between processes.

e) Up to 16 processes the critical time (and in many cases the run-time) is lower for configurations in which an even number of processes is placed, than for the configuration with one process less. Theoretically, the cause could be the communication or the I/O. By eliminating the network and I/O noise this behavior can still be seen (look at *SyncVirtually* in Figure 7.78c).

A possible explanation is the nature of the dual-socket system and its memory system which is shared among all processes of a single socket. With an odd number of processes the workload on the two memory systems and the L3 cache is not balanced any more.

7.9.9. Summary and Conclusions

To demonstrate the ability of PIOsimHD to mimic existing parallel programs, several configurations of the working groups Jacobi PDE solver are simulated. This PDE solver iterates over a 2D matrix and updates matrix entries by applying a stencil. The runtime parameters are highly configurable which allows arbitrary utilization of CPU, network and I/O subsystem.

A relative comparison of measured and simulated runtime is not appropriate, because the perfect replay of compute jobs execution time would allow achieving an arbitrary accuracy. Therefore, critical time is defined which is the amount of time spent in communication and I/O – all the aspects that are explicitly simulated. However, waiting time as part of the synchronization time is identical in simulation and observation. But since its quantity is determined, the actual time spent in I/O and communication can be assessed. The analysis investigates graphs plotting wall-clock time and critical time. With the help of PIOsimHD alternative hardware configurations are evaluated. This serves to validate the simulation results, and to analyze the impact of synchronization, communication and I/O quantitatively.

Conducted experiments are grouped into network bound, I/O bound, and mixed workloads. In many cases, the simulation behaves similar to actual measurements. This is also demonstrated in a comparison of generated profiles and timelines – in many cases the emerging activity closely mimics observed behavior. Only the startup time of MPICH2 is not well simulated – on the real system processes start time-delayed while the simulation starts all at the same time. Sometimes there is a discrepancy between simulation and measurements – the simulation estimates a better time than observed. Overall, the simulation proved to be a valuable tool for identifying bottlenecks because it provides a good estimate for performance: By comparing trace files of simulation and observations, bottlenecks in the MPI implementation (and PVFS) are identified. In one communication bound experiment, for example, certain send and receive operation take much longer than anticipated. Another example is non-contiguous data that should be written – it leads to inefficient handling in PVFS that degrades performance; this aspect is directly investigated in the traces that are obtained from PVFS with the HDTrace extensions.

A demonstration during the analysis applies PIOsimHD and Sunshot to critical path analysis. With the correct setting (very fast network and I/O subsystems) the duration of events corresponds to synchronization time; the filter feature of Sunshot allows analyzing waiting times of an optimal execution and indicates candidates for the critical path. In the Jacobi PDE, the row-wise domain decomposition and resulting communication pattern causes the critical path to avoid the two outer processes since they only have one communication partner, the other processes have two and synchronization of them leads to waiting times.

Additionally, an alternative I/O semantic (flush on close) is investigated. While flush-on-close does not increase run-time for small I/O because data is already written out, it reduces performance for large configurations. Similar to the previous experiments, complex I/O patterns emerge.

7.10. Alternative `MPI_Bcast()` Implementations

With the help of PIOsimHD, alternative implementations for collective calls can be evaluated and the benefit of a new algorithm can be quantified. To demonstrate the capabilities, alternative broadcast algorithms are implemented and evaluated.

In the simulated experiment, the setup previously used to validate the collective patterns in Section 7.6 is used. Performance of various algorithms is simulated for broadcasting 10 KiB, 1 MiB, 10 MiB and 100 MiB. The measured performance of MPICH2 acts as a reference to assess the following algorithms:

a) *P2P-Replay*. This is not a real implementation in the simulator. Instead, it is the experiment in which the extracted communication patterns from the traces are replayed to mimic behavior of MPICH2 (see Section 7.6 for a description).

b) *BroadcastScatterGatherall*: distributes pieces of the data equally among the processes by invoking `MPI_Scatter()`, then it calls `MPI_Gatherall()` to transfer all chunks to all clients.

c) *Direct*: The root process directly sends data to all other processes. To do so, it iterates over all ranks starting with Rank 1 and transfers the complete message.

d) *Pipelined*: The root process partitions data in chunks of 1 MiB of data and sends it to Rank 1 – once a chunk is received by another rank it forwards it to the next rank and thus a pipelined data transport is achieved. This algorithm is described further in Section 6.5.

e) *Pipelined-SMP*: This implementation makes the Pipelined implementation SMP-aware. The pipeline is built in a manner which avoids inter-node communication – data is pipelined through all local processes before it is transferred to the next process.

f) *Pipelined-SMP512* is a slightly modified Pipelined-SMP implementation that uses a chunk size of 512 bytes.

g) *BinaryTree* builds a spanning tree among the processes. The binary nature of the tree is derived from the fact that the number of processes which have received the data to broadcast doubles in each step. The processing scheme is illustrated in Figure 7.79. For example, for a communicator with a size between 4 and 7, three steps are needed. In the first step, the root rank (we assume Rank 0 is the root) sends data to Rank 4. In the next step, Rank 4 sends data to Rank 6 while Rank 0 sends data

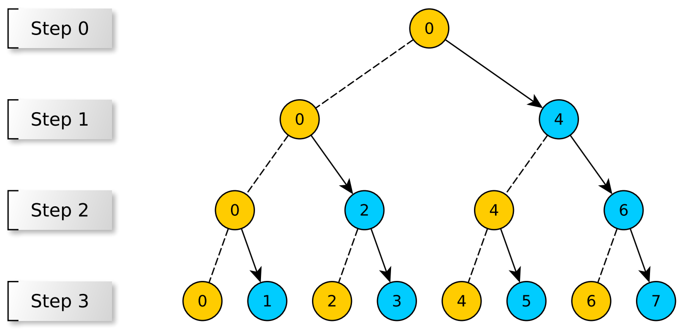


Figure 7.79.: Processing scheme of the BinaryTree implementation for MPI_Bcast(). The root node of the graph is the source of the data to broadcast. In each step, a process which has the data, forwards it to another process (an arrow indicates data transfer between source and target). Consequently, the number of processes which have received the data doubles in each step.

to Rank 2, and so forth. During the process root rank sends data to Rank 4, Rank 2, and, at last, to Rank 1.

h) *BinaryTreeMultiplexed*: This implementation uses the same communication pattern as BinaryTree. However, in contrast to BinaryTree a process sends data to all its children at the same time, which effectively multiplexes the available NIC among all receivers.

i) *BinaryTreeSimpleBlockwise* uses the communication pattern of BinaryTreeMultiplexed. But instead of transferring all data in a single message, data is partitioned in chunks with a size of 1 MiB. This allows for a pipelined transfer.

The measured time and the simulated estimates are shown in Figure 7.80 and in Figure 7.81, for a payload of 10 KiB and 100 MiB, respectively. Performance of the other sizes is between the these extremes, and, therefore, omitted.

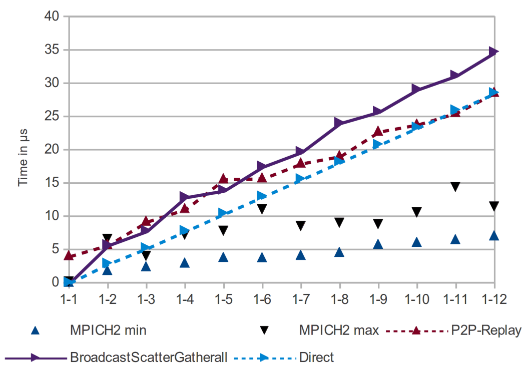
Observations and interpretation

a) For local communication, all simulated results are close together (see Figure 7.80a and 7.80c). This is expected because the characteristics of the components of the internal topology are close together. A slight performance difference is caused by the fact that intra-socket communication is modeled with a higher throughput. Another difference is the amount of data which must be transferred for the broadcast; all pipelined versions need more messages which increases the overhead of packet headers. The results for transferring 100 MiB of data are alike and, therefore, the intra-node times are not provided.

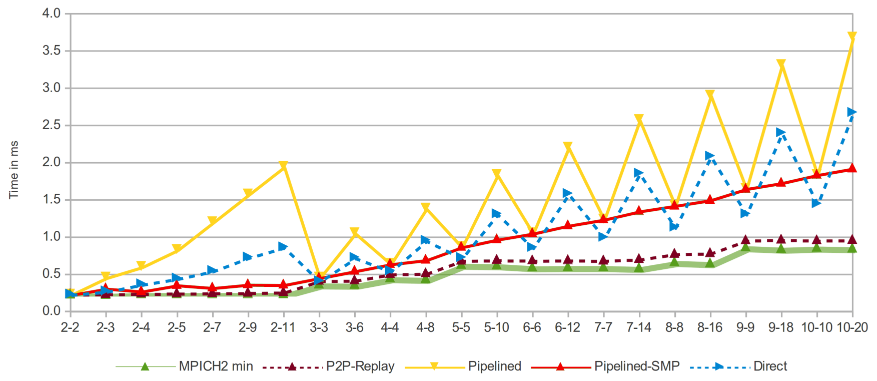
b) In intra-node communication of 10 KiB of data, the BinaryTreeMultiplexed implementation is slightly faster than the other algorithms. This effect is caused by the fact that intra-socket communication and intra-socket communication can be slightly overlapped by multiplexing data access (intra-socket communication is a bit faster in the model). The performance of all simulated algorithms are close together because the characteristics of intra-node communication is similar and cache effects are not modeled. In practice the question is, whether cache reuse makes the algorithm superior to the sequential processing – simulation cannot answer this question.

c) In inter-node communication of 10 KiB messages, the Pipelined implementation is slower then Direct (Figure 7.80b) – this is caused by the high Ethernet latency. The Pipelined-SMP algorithm behaves monotonically increasing. It matches the performance of Pipelined if only a single process is placed per node – this is the expected behavior. For the small payload the pipeline does not work, because data is chunked in fragments of 1 MiB.

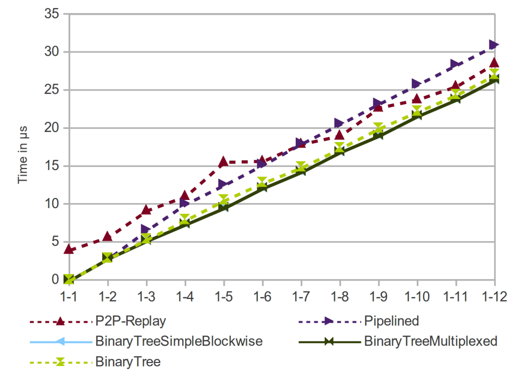
The communication pattern of MPICH2 is faster than most other algorithms – it uses an SMP-aware



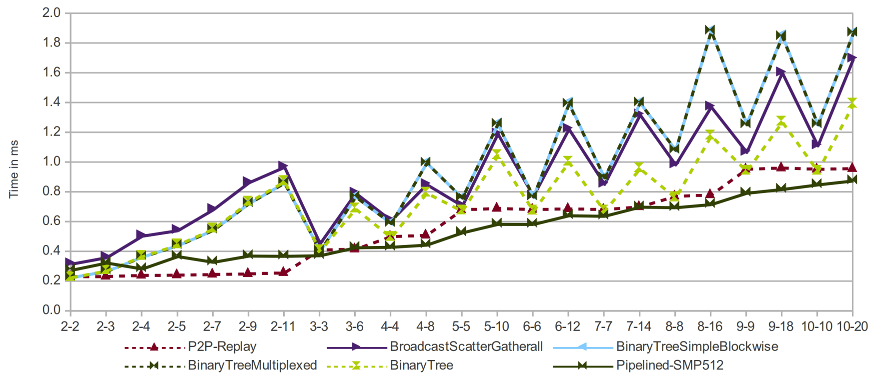
(a) Local communication algorithms (1).



(b) Inter-node communication algorithms (1).

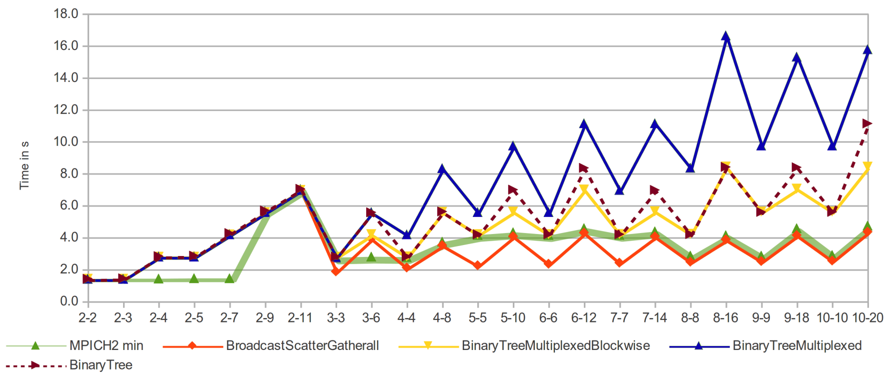


(c) Local communication algorithms (2).

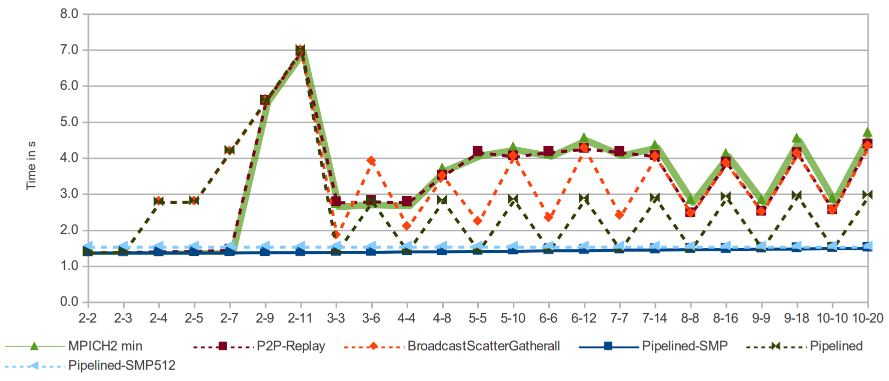


(d) Inter-node communication algorithms (2).

Figure 7.80.: Performance comparison of several `MPI_Bcast()` algorithms to transfer 10 KiB of data.



(a) Inter-node communication algorithms (1).



(b) Inter-node communication algorithms (2).

Figure 7.81.: Performance comparison of several `MPI_Bcast()` algorithms to transfer 100 MiB of data.

binary tree. However, the performance of the Pipelined-SMP512 implementation is slightly better for all configurations, except for two nodes (see Figure 7.80d). There is still room for improvement in the Pipelined-SMP version: This algorithm is not aware of the processor topology inside a node. Therefore, in this case all local processes transfer data between the sockets, which is suboptimal. Thus, overall performance could be improved slightly by knowing the node-internal topology. Also, the implementation could transfer data in a pipeline across all nodes – only one process per node participates, once all data is available on a node data is pipelined through the local processes.

d) In case every process is a dedicated node, the BinaryTree implementation matches the time for the replayed communication patterns well, because MPICH2 uses this algorithm for broadcasting small amounts of data (see Figure 7.80d). However, unlike MPICH2, the implemented algorithm in the simulator is not SMP-aware. All the other tree algorithms behave worse than the simple algorithm. Especially, the multiplexed version is much slower, because performance of the NIC is shared among all receivers which in turn increases the time until the next process can forward the data.

e) For large payloads and starting with eight processes, MPICH2 uses the BroadcastScatterGatherall algorithm. Therefore, the time of the simulation matches the replayed pattern (look at Figure 7.81a). For, for the configurations between 4 to 7 nodes the BroadcastScatterGatherall would also be better. The SMP-awareness does not work for the configurations *2–9* and *2–11*; here the measured time is similar to the one of several simulated implementations.

f) The pipelined implementation is estimated to need about 1.5 s which is at least two times faster than the current default implementation (look at Figure 7.81b). For the large amount of data, a slight overhead of transferring 512 bytes messages becomes visible (up to 5 processes).

Summary and Conclusions

The evaluation of a variety of `MPI_Bcast()` implementations demonstrates how the simulator can be used to investigate performance of alternative MPI algorithms. In this evaluation, a good match between original measurement and replayed point-to-point communication is demonstrated for small amounts of data, thus validating the appropriateness of the implemented algorithm. This experiment also illustrates the importance of selecting appropriate communication algorithms – a wrong choice significantly increases runtime. No single algorithm achieves best performance for all payload sizes.

For example, it turns out that for intermediate node counts with BroadcastScatterGatherall, one of the already implemented algorithms in MPICH2, performance could be improved compared to the actually executed algorithm (a binary tree implementation). In general, on our system binary tree implementations achieve a lower performance. With a pipelined implementation larger amounts of data are split into smaller messages that are transferred individually between the processes – based on the simulation results, an implementation of this algorithm in MPICH2 could improve performance significantly. During the investigation, the importance of SMP-awareness becomes also visible. About half the performance is lost in an SMP-unaware pipelined implementation; and MPICH2 is not SMP aware for configurations 2-9 and 2-11.

Consequently, a dynamic selection of the algorithm that is based on the network topology, the process placement and parameters is required. To achieve maximum performance, it seems also natural to be aware of node-local processor topology and the caching.

7.11. Chapter Summary

In this chapter, the software created for this thesis is evaluated by demonstrating its usefulness for predicting system behavior and by validating its applicability. Due to the length of the conducted evaluation, each section is wrapped up by a short summary by itself.

The systematic evaluation is divided into several stages: First, the overhead of HDTrace is quantified which provides insight into the correctness of times in trace files. Also, the performance and scalability of the sequential simulator is assessed.

Then, a cluster model is developed and parameterized. This model is used for all subsequent experiments. With the qualification process, a theoretical analysis is performed by comparing mathematical models and simulation results. Several basic experiments are conducted to verify the accuracy of network and I/O model before performing a detailed validation of collective operations. For this, the MPI-internal point-to-point communication is recorded and replayed by the simulator. Also, the parallel I/O performance for different levels of access is validated.

After that, collective algorithms implemented in the simulator are briefly described and the behavior of a scientific application is analyzed. The execution of our partial differential equation solver is recorded and replayed by the simulator for many configurations. Finally, alternative `MPI_Bcast()` *algorithms are evaluated to demonstrate the potential of the simulator to help analyzing novel algorithms.*

In the following chapter, the thesis is summarized and concluded.

Bibliography

[HSL10] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim: Simulating Large-Scale Applications in the LogGOPS Model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC, pages 597–604, New York, NY, USA, 2010. ACM.

[Int09] Intel. Intel® 5520 Chipset and Intel® 5500 Chipset Datasheet. Online: `http://www.intel.com/content/www/us/en/chipsets/server-chipsets/server-chipset-5500.html`, March 2009.

[KL08] Julian Kunkel and Thomas Ludwig. Bottleneck Detection in Parallel File Systems with Trace-Based Performance Monitoring. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 212–221, Berlin, Heidelberg, 2008. University of Las Palmas de Gran Canaria, Springer-Verlag.

[KTML09] Julian Kunkel, Yuichi Tsujita, Olga Mordvinova, and Thomas Ludwig. Tracing Internal Communication in MPI and MPI-I/O. In *International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT*, pages 280–286, Washington, DC, USA, 12 2009. Hiroshima University, IEEE Computer Society.

[Kuh09] Michael Kuhn. Simulation-Aided Performance Evaluation of Input/Output Optimizations for Distributed Systems. Master's thesis, Ruprecht-Karls-Universität Heidelberg, 09 2009.

[Kun07] Julian Martin Kunkel. Towards Automatic Load Balancing of a Parallel File System with Subfile Based Migration. Master's thesis, Ruprecht-Karls-Universität Heidelberg, 08 2007.

[MRKL10] Olga Mordvinova, Dennis Runz, Julian Kunkel, and Thomas Ludwig. I/O Performance Evaluation with Parabench – Programmable I/O Benchmark. *Procedia Computer Science*, pages 2119–2128, 2010.

[Sch05] Martin Schulz. Extracting Critical Path Graphs from MPI Applications. In *CLUSTER'05*, pages 1–10, 2005.

[Sea10] Seagate. Product Manual – Barracuda 7200.12 Serial ATA. Online: `http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%207200.12/100529369h.pdf`, 2010.

[Thi12] Artur Thiessen. Simulation von MPI-Collectives in PIOsimHD. Technical report, Universität Hamburg, 04 2012.

[Tie09] Tien Duc Tien. Tracing Internal Behavior in PVFS. Bachelor's thesis, Ruprecht-Karls-Universität Heidelberg, 10 2009.

Chapter 8 VIII

Summary & Conclusions

This chapter provides a summary of the thesis, starting with a brief overview for the contents of each chapter in Section 8.1. Section 8.2 presents a full executive summary of the work, selected resulted and conclusions.

8.1. Summary

This thesis develops and presents tools for an in-depth analysis of parallel applications and the system that runs them in order to simplify the evaluation of cluster systems. Because cluster systems are so difficult to analyze, it is necessary to extend and improve both the tracing processes for MPI and file system internals, and the simulation of parallel programs on system and application levels. This thesis introduces a systematic methodology for increasing insight into complex HPC systems: First some background on relevant topics is given, followed by an exemplary characterization of a cluster system. Then with HDTrace, a comprehensive tracing environment is developed that offers novel capabilities for increasing insight into I/O activity. With PIOsimHD, a tool for simulating parallel programs on application and system level is designed. Using the estimates provided by simulation, it is possible to identify bottlenecks and to gain a deeper understanding of system behavior. A careful validation of PIOsimHD demonstrates an astonishing accuracy of the developed models and further experiments demonstrate the suitability of the environment to design alternative algorithms.

Background and Related Work This section introduces important concepts with respect to parallel file systems, discusses a number of different file system types and describes the suitability of the abstraction level for parallel I/O in the simulator. Further, a discussion of the rich diversity of performance aspects in system and application is provided. Because the complex interactions of hardware characteristics with optimizations implemented on the different layers make application performance hard to predict, it is necessary to develop tools that assist in analyzing and understanding the impact of individual features on overall behavior. The terminology for performance analysis is introduced to provide an overview of this field. Additionally, several existing tools for sequential and parallel execution are briefly discussed using the working group's Jacobi PDE solver as an example with which to distinguish novel features from existing concepts. Also, MPI concepts are introduced that are covered by the simulation model. Since PIOsimHD aims to assist in evaluating optimization potential, existing optimizations for MPI are described. This chapter also introduces the concept of discrete-event simulation, which is the foundation of the developed simulator.

Characterizing the Experimental System In Chapter 3, the hardware of the working group's cluster system is characterized, i.e., the behavior of memory, network and I/O subsystem is assessed. As the behavior is analyzed for a rich variety of configurations, several performance characteristics are determined, a brief introduction to the mathematical background is given and general considerations for characterizing system behavior are discussed. On the one hand, this chapter puts the performance aspects mentioned in the previous chapter into perspective. On the other hand, the obtained characteristics are used as parameters for the model and during later validation experiments. In many experiments, performance fluctuations are visible, for example, independent runs of even long-running benchmarks lead to slightly different performance results. Even more interestingly, although identical hardware is used, memory performance depends on the node on which the benchmark is run.

Many configuration parameters can be set for MPI and the Linux kernel that influence behavior. This chapter further shows that the complex interplay of hardware components cannot easily be captured in a model. To allow an accurate simulation of individual operations, all contributing factors must be known and parameterizable, which is difficult at best. As a consequence, an exact simulation of all hardware and software factors is not feasible.

HDTrace & PIOsimHD With HDTrace, an environment and tool set is developed that allows enhanced recording of MPI-IO behavior including all parameters needed for simulation. This environment also offers novel visualization techniques that ease the analysis of parallel applications and enables comparisons between observations and simulation results.

By replaying recorded traces, the discrete-event simulator PIOsimHD allows simulation of parallel programs on application and system level. Software models for MPI library, and parallel I/O are executed on hardware models for cluster components. While details concerning the models currently being implemented for simulating application and hardware behavior are provided, illustrative examples explain the internal processing. The hardware models provide a level of detail that can be understood, still they rely on the most important characteristics for cluster systems. With the modular implementation of PIOsimHD, alternative models can be provided as well.

Together with PIOsimHD, HDTrace allows replaying application behavior on arbitrary virtual cluster environments and thus serves as an in silico laboratory for investigating the interactions between system, applications and communication library.

Evaluation The capabilities and accuracy of PIOsimHD are demonstrated in Chapter 7. To verify the correctness of the traced timings, the overhead of HDTrace is measured and the performance of PIOsimHD is assessed, showing that it is feasible to simulate clusters with 1,000 processors. To validate the hardware and software models, observations are compared with simulation results. It is necessary to determine the required characteristics of the hardware models, and as an example, a model for our cluster system is developed and parameterized with the characteristics measured in Chapter 3. In the qualification process, measured performance is compared with the basic mathematical models being implemented in the simulator. During verification, the performance of the mathematical network and the HDD model is compared with the simulation results. On the one hand, this indicates that the code implements the expected model. On the other hand, the influence of several run-time parameters, such as the network granularity, is investigated.

Additionally, performance of a variety of complex access patterns based on point-to-point communication and collective communication is evaluated and the simulation results are validated. To show the capabilities of PIOsimHD to mimic existing parallel programs, several configurations of the working group's Jacobi PDE solver are analyzed. Also, a rich set of access patterns for parallel I/O is evaluated.

The series of validation experiments tests the correctness of the models; a large performance difference indicates that the models insufficiently describe the real system. This means that either the relevant hardware aspects are not modeled, the system is described incorrectly, or interactions between software and hardware lead to an unexpected and undesired slowdown. Throughout the validation process, the developed visualization features are used to in order to assess the discrepancy between observation and simulation. With this approach, several bottlenecks within MPI and MPI-IO are localized and the appropriateness of the chosen hardware and software models is determined. Finally, the simulation of MPI calls with PIOsimHD and its ability to evaluate alternative MPI algorithms is demonstrated.

Conclusion The developed software meets the initial design goals as outlined in the following: HDTrace offers an environment which allows tracing and simulating the internal behavior of MPI-IO programs and the parallel file system on a cluster. Observations can be compared with simulation results to analyze the behavior of the existing system and suggest areas for improvement. With these features, inefficiencies can be localized: Many examples illustrate how the developed environment assists in revealing internal behavior and spotting bottlenecks in the existing system. Visual inspection of the traces allows users to quickly isolate areas that behave differently. Further, the infrastructure is useful for teaching performance aspects of cluster systems: Not only it is possible to analyze the behavior of an existing system, the virtual simulation laboratory facilitates the investigation of library-internal behavior and the evaluation of proposed optimizations. While the simplicity of the developed models and implementation makes them easy to understand, the models are powerful enough to reveal complex patterns that are observable on existing cluster systems. In contrast to a real system, PIOsimHD allows accessing internal behavior and manipulating it at will. Thus, it is a powerful tool to conduct research on new MPI-IO algorithms and to evaluate behavior on future systems.

8.2. Executive Summary

The executive summary highlights details about the contents of each chapter and elaborates on selected results.

8.2.1. Background and Related Work

Parallel file systems File systems are the software that enables applications to store a large data volume for a longer time period. A parallel file system allows processes of a parallel application to concurrently access a file at a high performance level. Due to the large number of components in a high-performance cluster, scalability and fault-tolerance are important characteristics of parallel file systems. With Panasas ActiveStor and Isilon's S-Series, two enterprise storage systems are described that permit efficient and parallel access. PVFS, as a representative parallel file system in HPC and its layered architecture, is described in more detail. To illustrate the data flow between client and server, the client-server communication protocol and the involved buffering are described. While all the file systems rely on different concepts and metadata schemes, all of them distribute data among multiple servers and clients must communicate data with the involved servers during an I/O operation.

Performance of parallel applications There is a multitude of hardware and software factors that influence performance of parallel applications. An excerpt of these factors is discussed in this section including optimization strategies that try to mitigate negative characteristics of hardware devices.

Several layers of software are involved in the execution of the application and contribute to observed computation, communication and I/O performance. Of all these, I/O behavior is by far the most complex to understand because I/O requires some computation and communicates data, and not only depends on hardware characteristics but also on the sequence of I/O operations that is performed by an application – the so-called access pattern.

I/O optimization strategies attempt to provide the highest amount of performance for all access patterns, and include caching algorithms for read-ahead and write-behind, aggregation and scheduling of operations, and RAID schemes. With parallel file systems, even more aspects become performance-relevant: Metadata handling, data distribution among servers, data replication, high availability, and synchronization requirements.

The Message Passing Interface The semantical aspects of several collective calls are discussed because they have performance implications, some of which could be potentially mitigated by relaxing the semantics in future standards. By giving programmers a chance to orchestrate parallel I/O, MPI-IO offers capabilities beyond the POSIX standard. A file view with an MPI datatype provides an approach for accessing multiple data regions with a single MPI call. With the sequential consistency data model of MPI-IO, the strict consistency model of POSIX is relaxed. In contrast to POSIX, this data model permits truly parallel access. Performance of an MPI application not only depends on the hardware and software capabilities, but also on the mapping of logical processes to physical resources, which determines the efficiency of inter-process communication. For a collective operation, the communication algorithms that realize it within MPI have a major impact on performance because they define the communication pattern. Depending on hardware, process placement, memory datatype, participating processes, amount of data communicated and state of the program, a different algorithmic implementation might lead to better performance. The rich diversity of the parameter space makes it difficult to utilize a system optimally.

Several optimizations of the MPI library are described: Alternative algorithms for a collective call, automatically choosing the best algorithm from the existing ones, tuning MPI parameters, process placement and topology awareness. Many of these continue to be the focus of current research and there is no solution that leads to full system utilization in all cases. There is a tremendous number of imaginable optimizations for parallel I/O, and in contrast to local I/O, data can be cached, aggregated and scheduled over a hierarchy of components, even within the clients. In several cases, these optimizations are stacked on top of existing solutions to mitigate the restrictions of the layers below. Unfortunately, there is no self-aware library for communication and I/O that takes hardware characteristics directly into account.

Performance analysis and tuning One consequence of the complex system behavior is the difficulty of analyzing and improving the performance of parallel applications. This section introduces several approaches that try to integrate performance as a non-functional requirement during the software development phase. These approaches use hardware models to predict (or simulate) behavior before an application is run on a supercomputer. Unfortunately, however, they are not used in practice. Instead of employing such an integrated approach, developers simply tune their existing program: Performance of the real application is measured, the behavior is analyzed in order to identify bottlenecks of relevant code paths, and then the code is adjusted to hopefully improve performance. This activity is repeated until a satisfactorily speed is achieved, or until it is too time-consuming to proceed with code modification. One drawback of this approach is that it is usually done without knowing the potential speed of the operations; thus it is hard to estimate the potential gains of code modifications. With the concept of co-design, application development and system design is interwoven to embed time-intense aspects of the parallel program directly into the hardware. Therefore, this concept integrates performance as a non-functional requirement into the application development.

This section also introduces the terminology of performance analysis tools. Additionally, several existing tools for sequential and parallel execution are briefly demonstrated by the working group's Jacobi PDE solver in order to distinguish novel features from existing concepts. For tuning an application, these tools measure and visualize the behavior of the parallel application. With most tools, the developer must assess the observations by him/herself, because the systems are not aware of hardware characteristics. However, by recording hardware counters at least the usage of the CPU can be assessed, and thus the efficiency of compute intense code can be estimated. In addition, Scalasca offers an automatic analysis for typical reasons behind MPI-bottlenecks, such as synchronization issues. A section outlining trace formats complements the tools section by mentioning design considerations and introducing an archetype, the Open Trace Format and API.

Discrete event simulation A model is a representation of a specific system and its behavior. In the modeling phase, a conceptual model is developed that covers processes and characteristics of the system well enough to answer specific scientific questions. The meta model that describes many similar systems is referred to as the domain model; models for specific systems are developed by parameterizing the domain model accordingly. This thesis seeks to create a domain model for parallel applications that describes their execution on a cluster computer. Since every model simplifies the system under investigation, the applicability of a model must be shown. The literature describes several approaches to demonstrate the accuracy of a model: In the qualification process, applicability of the conceptual (or domain) model is investigated directly. In regards to validation, the results of a program that implements a model can be compared directly with the observations. The results of the implementation can also be compared with the conceptual model, which is called verification.

A simulator is a program that describes and executes a model. In discrete-event simulation the state of the system is only changed by events; an event is executed at a specific model time. The simulator keeps a list of future events and processes them sequentially: First, the event with the earliest start time is chosen, then the event is executed. An event is able to change the system state or characteristics, and it might generate new events. An overview of existing simulation engines and simulators for parallel applications is provided. Although there are already several simulations for clusters available, none of them offer the capability to simulate MPI-IO applications on application and system level on the same degree as PIOsimHD.

8.2.2. Characterizing the Experimental System

Characterizing system behavior This section describes the general difficulty of characterizing system behavior. A performance metric is a quantitative measure for the system state, and benchmarks are designed in such a manner that they reveal the relevant performance metrics, usually indicating the best case characteristics. Code that implements a particular use case, i.e., that stresses the system in a specific way, is called a kernel. Measurement of the system behavior is subject to observational error, thus a single measurement is not enough to determine the true value of a performance metric.

The measurement results can be displayed with histograms, which is especially useful for large numbers of data points.

Background activity skews the results of a measurement: While short-term activity resembles a random error of the observation, long-term activity looks like a systematic error. Therefore, exclusive use of a resource and repeated runs of a benchmark are important. In this thesis, multiple benchmarks are run to characterize the hardware and their results are compared; this reduces the chance of potential systematic errors. All the experiments are conducted on the working group's 10 node cluster, which offers 20 Intel Westmere processors with a total of 120 physical cores. The hardware specification of components serves as a reference to estimate performance, and by comparing the measurements with the hardware specification, the measurements can be validated.

Memory behavior

With the memory benchmark, the sensitivity of the system to the specific memory access pattern becomes visible: Due to the access granularity of full cache lines, random access is much slower than sequential access, for example. Also, the impact of the 64 KiB level 1, 256 KiB L2 and 12 MiB L3 cache becomes apparent: 128 bit accesses achieve a throughput of up to 40 GiB/s, 25 GiB/s, 20 GiB/s and 12 GiB/s for L1, L2, L3 and memory. The `memory-bandwidth` benchmark achieves a lower performance of about 8 GiB/s for read and write accesses. However, it measures memory access for accessing small memory chunks. With this benchmark, the variability and noise of memory access is investigated. In the initial results, all nodes behave comparably: The distribution of the timings in the plotted histograms look similar. In most cases, the histograms revealed a non-common probability distribution, and spikes are presumably caused by system noise and by hardware factors. The experiment is repeated with the aging system: The obtained behavior of the individual nodes of the write kernel is now different: Most histograms show additional spikes with lower throughput than before, and hence a variation of the hardware that is otherwise expected to be identical now becomes visible. This experiment also demonstrates that repeated runs lead to slightly different behavior: Although each kernel runs for about 120 s, in many cases the resulting histograms show one additional spike with lower performance. While these effects cannot be explained, they are related to the aging of our cluster. The observed effects make an accurate modeling and prediction of performance for memory access difficult. Since all communication and I/O requires memory access, memory access variability generate additional noise to these processes and especially to node-local communication.

Inter-process communication To characterize inter-process communication, benchmarking results for the performance of MPI are provided for intra-socket, inter-socket and inter-node communication. In this process, the characteristics for latency and throughput are determined, and the overall behavior is assessed.

A throughput test shows that inter-node communication is far below the expectation for Gigabit Ethernet. A tuning of the TCP/IP configurations and kernel modifications improves performance only slightly from 67 MiB/s to 72 MiB/s. The reasons for the degraded performance cannot be identified, but it once again illustrates the complexity of the interplay between hardware and software factors, and the tuning. In intra-node communication, the bi-directional data transfer achieves slightly higher performance; intra-socket performance is slightly higher than inter-socket communication – about 3,400 MiB/s are achieved for the uni-directional kernel running on two sockets and about 4,600 MiB/s for the bi-directional intra-socket communication. The quartiles for measured inter-node latency show a high fluctuation – already 50% of the measured data points vary by 10%, and the maximum value is about twice the average value.

In intra-node communication the variability is much lower. However, the maximum value is still much higher than the mean. Further, the observed inter-node latency differs from run to run; one execution of the benchmark even observes a higher minimal latency than the third-quartile of all other runs.

To assess this behavior, an MPI benchmark is executed that measures communication times for a variable amount of transfered data; its results are illustrated in several plots. It turns out that the CPU cache has a large impact on the observable intra-node performance: It improves performance significantly for medium-sized messages that fit into the cache. The characteristics of bi-directional and uni-directional performance are different, and a high fluctuation between independent measurements is observable. In a comparison between Open MPI and MPICH2, these implementations exhibit slightly different performance characteristics.

To assess the variability further, the spread of data points within a single benchmark run is briefly evaluated. Histograms show that timings of a single run are clustered together, i.e., behavior is similar over time; yet, several clusters with similar values can usually be identified. In most cases, they are divided by rather sharp borders. While exact causes are not identified, potential influences are discussed. For example, packet fragmentation of messages, store-and-forward transfer, and the variability of memory accesses add to the observed noise. In timelines, variability that is caused by background activity, i.e., the utilization of a resource, can be distinguished from the random-like effects of hardware and software because it delays a sequence of data points to a certain extent. In addition to the variability within a single run, the average values of independent executions of the benchmark fluctuate by several percentage points.

I/O behavior The behavior of an I/O subsystem is more complex than that of any other hardware device. Besides the hardware technology, the scheduling policy and caching inside the block device, file system and operating system are involved in the I/O path, and they contribute heavily to performance. After these aspects are briefly described, the average performance of cached I/O is analyzed with IOZone. Even for cached accesses the observed performance depends on file size and access granularity. Without an in-depth detailed analysis of the observations for individual operations, understanding the I/O subsystem is impossible.

Similar to the memory and communication benchmarks, a newly written benchmark measures performance of individual operations: Execution times are measured for operations that bypass the Linux cache (O_DIRECT), for synchronous operations, and for regular cached I/O. The performance variation of individual operations is very high: For direct I/O it varies between 10 MiB/s and about 120 MiB/s. When using cached I/O, the fluctuation is even higher, up to 5,000 MiB/s and 2,000 MiB/s for read and write operations are observed, respectively, if data is cached. Multiple horizontal clusters of values can be identified in the created timelines which correspond to hardware and software factors. For example, whenever a metadata block of the file system is updated, performance of the currently executed operation is slower – this becomes visible by looking at the synchronous operations. Sequential I/O is usually much faster than random I/O. However, for synchronous operations, metadata of the file system must be updated before the write call completes. Therefore, the actuator has to move the heads of the disk to the position of the metadata, resulting in random-like behavior. A re-write of already written blocks is usually faster because no additional block allocation is necessary. Similarly, for a cacheable operation, data can already be available in the operating system cache or the disk internal cache. The influence of memory performance and system noise on I/O performance becomes visible by looking at the average results, and at the individual measurements from accessing data on the in-memory file system tmpfs. Overall, by looking at the timelines, many interesting and unexpected patterns are observable. One example is the performance increase with the number of operations when reading data from tmpfs. Another example is the periodic bursts when writing data to the HDD in a random pattern.

HDTrace Environment The tracing library of HDTrace records MPI activities of parallel programs in XML-based trace files. Internal activity of PVFS and MPICH2 are recordable, and with the relation concept, the cause-and-effect chain between parallel program and the triggered activity inside the parallel file system can be assessed. Statistic files record metrics of the operating system such as measured network and block device throughput, but they have also been used to analyze energy consumption of a program. Trace format and API have been designed with high flexibility and simplicity in mind. Using a simple project file, traces of independent event sources can be linked together and displayed in the trace viewer Sunshot.

Sunshot is a major rewrite of the existing Jumpshot viewer to support many new features. It offers many views and features of which several are novel for trace-based tools:

- Many features support assessing statistics: In the timeline display, the scaling of the values can be adjusted to quickly find global or local outliers. Histograms can be plotted to grasp the distribution of the values over the currently zoomed timelines. With derived metrics, a mathematical expression can be applied to aggregate multiple statistics into a single metric.
- It assists analysis by highlighting relevant information: Users can define filters for the events that are shown in timelines and use color-coding of events. These features evaluate the attributes of the activities. Similar to this approach, activities can be colored depending on the value of a mathematical expression.
- Common MPI datatypes are visualized. The hierarchical design of the memory datatypes is displayed to highlight the composition of datatypes. Also, the visualization allows seeing how a datatype is mapped and unrolled to the file offsets, which ultimately shows which bytes of the logical file are addressed. This visualization eases the understanding of user-composed datatypes, while still preserving the nested character of the datatypes to simplify the analysis. With the gained insight into an application's spatial access pattern and its datatype layout, it is possible to localize inefficient access patterns and the communication of wrong data.
- Multiple project files can be loaded at the same time. This allows rendering of parallel application and file system activity in one display.

8.2.3. PIOsimHD – the MPI-IO Simulator

Chapter 5 introduces PIOsimHD, the MPI-IO simulator for application and system level. Also, details of the currently implemented models for simulating application and hardware behavior are provided. The appropriate level of detail for the model is discussed. A detailed model has the disadvantages of being complex and performance-hungry, it may pretend a false accuracy, and it is hard to interpret. The latter aspects have also been observed during the characterization of the system: Interaction between hardware and software is complex, and influenced by the initial state of the system. Consequently, it is difficult to determine all the required parameters and characteristics for simulating the real behavior perfectly.

Hardware model PIOsimHD contains models for cluster nodes, network infrastructure, MPI processes, I/O servers, block devices and a server-side cache layer. For each component, a model can be chosen independently. The implemented models for network communication, parallel I/O and block devices are much simpler than the real hardware behavior; they are just based on a handful of hardware characteristics. However, the chosen level of abstraction describes many aspects of a real compute node's behavior. While the simulator is based on discrete-event processing, network and block devices use higher level concepts; they process one operation, called 'job', after another. The amount of simulated work per job is limited. By dividing complex operations into smaller jobs, it is possible to interleave them. Processing of computation is an exception; with time-sharing, all compute jobs are executed concurrently by multiplexing the available processing capabilities.

A node hosts all kinds of processes: MPI processes, I/O servers and forwarders. It offers a number of CPUs with a given processing speed and memory resources that are shared among all hosted processes. Compute jobs require a certain number of instructions; each CPU processes a defined number of instructions per seconds. Memory is used by the server's cache layer for buffering I/O. There are two models for a block device, the SimpleDisk model and the RefinedDiskModel. The former relies on average access time and the maximum throughput. The latter model incorporates RPM, track-to-track seek time and a short seek distance for which the track-to-track seek time applies. Consequently, it remembers the last access location and it computes the latency of a job based on the position to access, and the previous position. If the offsets differ more than the short seek distance, an average seek is performed; otherwise a small seek is performed, and if the data is accessed sequentially, no seek is necessary at all. A mapping from files to physical blocks is not performed. Files are assumed to be stored sequentially on disk. The implementation of the RefinedDiskModel contains a scheduler for I/O jobs. Similar to NCQ, pending requests are reordered to improve performance, but to use this feature, the cache layer must submit multiple operations.

Network model The network topology is modeled by a graph of nodes and directed edges. By default, messages are fragmented into packets that are transferred with store-and-forward switching between source and target. A routing algorithm determines the path throughout the network. Currently, network nodes and edges are modeled with latency and throughput. The throughput determines the time for processing the packet on the component, i.e., the component is busy during the processing. Latency is added to the arrival time on the target node. To simulate memory access, which is especially important to predict intra-node communication, a memory node adds local throughput as another characteristic that is applied for data transfer between directly connected nodes. Every process is connected to the network topology with its own (virtual) NIC. An *NIC* fragments the message into packets with a size of at most *network granularity*. There is also an analytical NIC model that computes the arrival time on the target node directly. While this alternative model reduces simulation time, it cannot simulate congestion of the intermediate network.

Inter-process communication Data transfer and congestion are simulated with the developed network flow model. In this model, every pair of source and target is assigned to a virtual network channel (a stream), and the data of each stream is transferred independently. Packets of a stream are pushed from source to sink until a congestion occurs. In the model, congestion occurs when more packets of a stream are currently in-flight to the next component than the bandwidth-delay product of the connection to this component allows. A component that encounters a congestion stops data transfer of this stream, until it is notified by the components downstream that it might continue – whenever a component starts to process a packet, it notifies the sender that the packet is now processed and thus received. Streams whose data is available are processed in round-robin which results in a prioritization of streams that are connected with a slower performance. This scheme is illustrated with two scenarios: One scenario shows a bottleneck next to a source and another one shows a bottleneck close to a sink. These scenarios show the performance of the steady state and demonstrate that the flow model is able to utilize network resources while providing a realistic data transfer of larger messages.

Applications Parallel applications are modeled as a sequence of commands; these can be MPI operations, compute jobs or other operations. The simulator provides an implementation for each command; implementations are built on the concept of state machines. Each state defines the next state to execute and it may initiate message exchange or start another command as a nested operation. In this concept, computation is simulated by starting a compute job before the operations of a state are executed. Once the initiated operations are finished, i.e., required messages are sent and received, and all nested operations complete, the state machine proceeds to the specified state. Receives match transferred messages with a semantics similar to MPI communication; the matching rules also ensure that nesting of commands works properly. The concept is illustrated by implementing an algorithm for `MPI_Allreduce()`.

Parallel I/O The model for parallel I/O consists of a client-server communication protocol and I/O forwarding. It is implemented in the MPI-IO calls and the I/O servers. Currently, the model covers the I/O path and no metadata operations. A function partitions data among the available data servers – by default, it is distributed in round-robin. While they differ in their metadata handling, the data path of file systems must at least follow this elementary pattern: In the write path, a client sends data to all servers concurrently, then it waits for an acknowledgement; in the read path, data is requested from all servers at the same time. Each client-side I/O operation results in exactly one request per server of which data is accessed. Collective MPI calls mimic the behavior of the Two-Phase protocol, and therefore cause multiple I/O requests and exchange data between clients.

A server pipelines block I/O and network transfer by fragmenting data into pieces that are transferred between network and cache layer. Data on the block device is accessed in chunks of a size less than *I/O granularity*, while network packets are fragmented into packets with a size of *network granularity*. The interaction between server cache and block device is sophisticated and ensures that enough memory is available to store received (and read) data. The detailed execution path is complex since it involves measures to control network transfer and available cache.

Currently, there are four implementations for the cache layer available. The NoCache implementation processes one operation after another and does not offer a cache. The SimpleWriteBehindCache executes operations with the FCFS strategy but stores incoming data in a cache to permit write-behind. With the AggregationCache, the currently scheduled operation is extended by other pending operations iff these operations overlap with the extent (and file) of the current operation. The AggregationReorderCache increases the capabilities of the AggregationCache by allowing it to pick the next I/O operation – it prefers large operations and executes the operations using an elevator algorithm. All cache layers try to process an operation as soon as one is pending.

A brief comparison of the model parameters with the implementation of PVFS and the Linux kernel shows that similar concepts are covered in many cases: Write-behind, pipelining and the aggregation of operations are implemented; read-ahead or read caching are not yet modeled. However, the current cache layers are much easier to understand than the behavior of a Linux system, which also depends on many tunable run-time parameters. Especially read caches and read-ahead are unpredictable for concurrent access of a large number of clients. Since the introduced model describes the basic operations on the data path, it is applicable to a multitude of parallel file systems.

Workflow To conduct an experiment with PIOsimHD, a hardware and software model is created that is fed into PIOsimHD for simulation. Then the results can be assessed post-mortem. Creation of the model requires the definition of the hardware characteristics of all components, the definition of the process command sequences and the mapping of the available processes to the hardware. The application and system model can be either defined in XML files or created by a Java program. When an application has been traced with HDTrace, these files can be directly loaded by the simulator. To execute PIOsimHD, run-time parameters, such as tracing options, must be set; then the mapping of model components to implementations is performed and the simulation is run. Once the simulation completes, statistics for each component are output, and traces are created (if desired). These traces can be loaded into Sunshot for a post-mortem analysis of behavior. By inspecting trace files, the observed behavior can be compared with the simulated behavior. As opposed to application traces, PIOsimHD supports introspection into internal activity. Thus, individual packet transfer and operations conducted on a block device can be recorded.

8.2.4. Simulator Implementation

Chapter 6 provides more insight about selected implementation details. Characteristics of all hardware and software models are stored in classes of the separate PIOsimHD-Model package. To support automatic conversion between Java classes and XML, the member variables are annotated. The simulator implements a hardware model by accessing the model class with its characteristics. The implementation of a hardware component by the simulation engine is demonstrated based on the SimpleDisk model. The example illustrates the sequential processing of pending jobs and actions when a job is started and completed.

To foster modularity, implementations can be provided and selected dynamically without recompiling the simulator core. The mapping is defined in text files and realized with the Java Reflection API: For each model class, one implementation class is selected that is used during the execution. A single command can be implemented with multiple algorithms. By default, the last implementation is used. However, an algorithm can spawn a nested command and select a specific implementation for a command. As a result of this concept, new and refined models can be incorporated easily.

Example code snippets for defining the application and system model in Java complement the description of the workflow that replays behavior of traced applications in the simulator. Basically, to describe an experiment in silico, components are instantiated, characteristics are defined, and the network components are interconnected. The simulator takes care of loading the corresponding implementations for the selected models.

The implementation of commands with the state machine concept is illustrated for `MPI_Allreduce()` and `MPI_Bcast()`. A simple algorithm for `MPI_Allreduce()` invokes `MPI_Reduce()` and `MPI_Bcast()`. The presented implementation for `MPI_Bcast()` splits the payload into small messages that are transfered in a pipeline through all processes.

8.2.5. Evaluation

Performance The overhead of HDTrace is assessed by measuring the time for invoking `MPI_Barrier()` with a single process. It turns out that the overhead is in the order of one microsecond; therefore, it is relevant only for intra-node communication of small messages. In this experiment, VampirTrace causes a similar overhead. Thus, from the performance perspective, HDTrace is on the same level as state-of-the-art tracing tools.

PIOsimHD demonstrates a processing rate of 1.5 million events per second, which is comparable to other state-of-the-art simulators such as LogGOPSim, although LogGOPSim is written in C. The scalability of the simulator is assessed with an `MPI_Bcast()` call across up to 65536 processes. In the analysis model creation, model initialization and the simulation time is measured demonstrating that the simulator scales almost linearly with the number of simulated processes and events. Alternative NIC models, such as the analytical model, which does not cover congestion, reduce the number of events significantly and therefore need less processing time. With about 1000 s for simulating 16,384 processes, it is feasible to mimic real applications for hundreds of processes and to simulate single MPI calls for even larger numbers of processes.

Parameterization of the model Since the *Westmere* nodes serve as a testbed for all experiments, a system model is built for those nodes. A network topology is created that represents our dual socket compute nodes and the Gigabit Ethernet connection.

In order to represent hardware of our cluster, the model is parameterized. Basic information is queried from available datasheets for individual components, and according to the measured characteristics in Chapter 3. While most characteristics are available directly, network characteristics are derived hierarchically from the measurements.

Qualification of the model In the qualification process, measured performance of point-to-point communication and the I/O subsystem is compared to the mathematical models that are based on latency and theoretical throughput. While in most cases the communication model matches well (±10%), there is a discrepancy for the inter-node SendRecv kernel and for medium sized intra-node communication. For SendRecv these are due to early receives, which is not covered by the timing during the measurements; for medium sized communication, performance is better because it utilizes the CPU cache. Without explicitly tracking data and the CPU cache, the location of the data cannot be considered by any model. However, tracking data is complex and also depends on the computation model because that involves memory activity, too.

Real I/O highly depends on the access pattern and the system configuration. Although observed performance for a variable record size can be fitted into a model with latency and throughput, the actual values for the characteristics depend on the performed experiment because it determines how well optimizations can improve performance. Therefore, the mathematical model that is based on average seek time and maximum sequential throughput underestimates performance. Because the cache layers of the real system and of the simulator try to reorder and aggregate accesses into sequential blocks, the performance of truly random I/O is lower in the mathematical model.

Verification The results of the mathematical models are compared with simulated point-to-point communication and a single I/O subsystem. This checks for correctness of the implemented models for these cases. Additionally, the impact of different network granularities is assessed. Fragmenting messages into packets reveals a discrepancy between simulation and theory because utilization of the network graph requires that the components can process packets concurrently – which is impossible if just a single packet is sent. This shows the necessity of fragmenting messages into smaller packets in the model.

The RefinedDiskModel model matches theoretical performance perfectly, and, as expected, the AggregationReorderCache improves performance of writes with an increasing cache size.

Validation With the verification and qualification process, the most elementary point-to-point and I/O operations are validated indirectly. Several more involved point-to-point and collective communication patterns are used for a direct validation of simulation accuracy: An MPI benchmark measures performance while tracing of internal point-to-point communication is enabled – these traces are fed into the simulator to mimic the communication patterns. With this approach, the discrepancy between model and observation becomes visible. Furthermore, several inefficiencies in MPI are identified: Due to the estimation done by simulation, an algorithmic suboptimality can be clearly distinguished from unexpected bottlenecks inside library and system. Open MPI performance is included in the comparison showing a different performance behavior than the measurements of MPICH2 and the simulation. Both implementations have their strengths and weaknesses – none is optimal, showing the need for tools to systematically analyze and improve communication performance.

In most cases, PIOsimHD resembles the overall communication behavior well; however, in intra-node communication, it usually underestimates performance – presumably, CPU cache effects improve performance on the real system. A high variability in observed communication times becomes apparent, especially in the statistical analysis of small payloads. While node-local communication is typically rather robust, the deviation of minimum, maximum and the quartiles is much higher for inter-node communication. Taking this fluctuation into account and since PIOsimHD should anticipate performance, the relative accuracy of the simulation seems acceptable.

The model of parallel I/O is validated by running a large set of experiments with orthogonal parameters and comparing them with the simulation. In many cases, the simulator approximates performance well. In some cases, it overestimates performance, and in a few cases, its estimates are too low. Several instances in which a discrepancy between simulation and observation occurs are investigated by inspecting traces of the simulator, the recorded traces of the application and the PVFS internal processing. It turns out that the parallel I/O system is very sensitive to timing issues – performance is good if requests can be aggregated into large requests so that sequential access is possible. Bad timing leads to a situation in which blocks on the disk are accessed in random-like fashion, which seriously degrades performance. This is the main reason for the degraded performance in PVFS – the improved scheduling of PIOsimHD leads to more sequential patterns and thus a better utilization of the HDD. Timing effects are assessed by inspecting the trace files; in many cases, effects that are observable on the real system are visible in the simulation as well.

Every command that is executed by a parallel program must be explicitly programmed in PIOsimHD. Currently, the simulator provides at least two alternative implementations for selected collective calls of MPI: a very simple implementation that communicates all data through the root process of the communication, and a re-implementation of an MPICH2 algorithm. By visualizing the internal communication of MPICH2 with Sunshot, the correct implementation of the algorithms in PIOsimHD is verified. Thus, it is possible to simulate applications using similar communication patterns as with MPICH2.

The simulation of the working group's PDE demonstrates PIOsimHD's ability to mimic the execution of existing parallel programs, including the system behavior. A discussion reveals that a relative comparison of measured and simulated runtime is not appropriate, since the accurate replay of timings for compute jobs can lead to an arbitrary accuracy when the amount of computation is increased. Therefore, the critical time metric is defined, which is the amount of time spent in simulated activity – communication and I/O. In many cases, the simulation behaves similar to actual measurements, which is also demonstrated in a comparison of generated profiles and timelines.

Similar to the verification of communication and I/O benchmarks, several bottlenecks of the MPI implementation and PVFS are identified by inspecting trace files. This shows that PIOsimHD and HDTrace are valuable tools for investigating the behavior of parallel applications on a cluster system.

Experimenting with HDTrace/PIOsimHD During the validation experiments, the features of PIOsimHD and HDTrace are also applied to further investigate certain aspects of system and application:

With a modification of the hardware characteristics in PIOsimHD, the replayed activity reveals the critical path of the PDE and the resulting waiting times of individual operations. Sunshot supports this analysis with the filter feature, which eliminates operations to highlight activities with waiting times due to synchronization dependencies.

An evaluation of a variety of `MPI_Bcast()` implementations applies the simulator to investigate performance of alternative MPI algorithms, which is one design goal for PIOsimHD. In this evaluation, a good match between original measurement and replayed point-to-point communication is demonstrated for small amounts of data, thus validating the correctness of the implemented algorithm. With the pipelined implementation that is introduced in this thesis, performance is expected to improve. The performance benefit of an SMP-aware implementation becomes visible as well.

The impact of an alternative I/O semantic is investigated with a modification to the regular `MPI_File_close()`, where the server is forced to flush the write-behind cache to the block device. This alternative semantic slows down execution only for a few experiments, but the effects can be quantitatively assessed, and the emerging complex I/O patterns are also investigated.

In summary, the combination of the enhanced tracing environment and a novel simulator offers unprecedented insights into interactions between application, communication library, file system and hardware. On the one hand, several unique features allow quantitative analysis of measurement which helps localizing bottlenecks. On the other hand, with PIOsimHD applicability for assessing performance of clusters consisting of 1,000 processors, it serves as virtual laboratory for developing alternative algorithms and studying behavior on arbitrary cluster environments.

Chapter 9 IX

Future Works

In this chapter, features of and extensions to the presented work are described that could not be realized during this thesis. Besides the introduced optimizations[1]*, extensions to MPI are imaginable that have yet to be researched thoroughly. A few of those concepts are discussed in Section 9.1. Prior to implementation in a real system, they could be evaluated with PIOsimHD to assess their potential benefit.*

There are three general ideas for extending the presented work:

- Performance evaluation of more cluster environments: The simulator has been applied successfully to the working group's cluster system. PIOsimHD could help in analyzing other supercomputers to identify bottlenecks in the system architecture and it would foster understanding of relevant system characteristics. For example, the Blizzard supercomputer of the DKRZ could be evaluated with a similar methodology. At the same time, the system model of the simulator could be extended to incorporate aspects relevant to this supercomputer, for example it is expected that RDMA has an impact on communication performance.
- Porting to an existing trace format: With HDTrace, an alternative trace format has been developed; its development has been guided by the needs for simulation and tracing of client-server activity that initially could not been recorded with existing trace formats. During this thesis other trace formats have evolved – now it seems that all the required information for the simulation can be included in these formats, although that may not be comfortable. While HDTrace still offers several capabilities beyond other trace formats, it is possible to modify PIOsimHD and Sunshot to rely on trace formats such as OTF. An advantage of this strategy would be that these tools can be applied in typical environments without forcing users to link to another trace environment. Also, it would eliminate the necessity of porting the trace environment to other supercomputers.
- Access pattern repository: A global repository of application behavior might be valuable for the community. Such a repository could contain trace files for relevant scientific applications and it could be open for researchers to provide and to use the available traces. In combination with a replay mechanism this would allow application specific benchmarking – researchers could evaluate application performance without the need for running the application. This is especially valuable for developers of middleware such as communication and I/O libraries. Further, by comparing performance across supercomputers, the system that is best suited for the communication and I/O pattern can be identified. With the student projects Parabench and Paraweb, a first attempt towards this goal has been made.

There are several minor modifications that would improve HDTrace and PIOsimHD:

- Improved build system: The C components mainly rely on the *GNU build system* (also known as *Autotools*). Correct and portable usage of the GNU build system is complicated, also execution of the configuration process is slow. Additionally, the configuration of the experimental intercepting library, e.g., for POSIX calls, is not automatized yet. One task for diligent work is to automatize the whole build process with *Waf*[2]. Waf is much easier to use and maintain, and the configuration process is much faster. The *TraceWriting C Library* has already been ported to Waf.
- Improved analysis of MPI-IO datatypes: Sunshot independently visualizes the MPI datatypes Vector, Contiguous and Struct for every process. More MPI constructors could be supported, also, for collective operations, a view could illustrate the accessed data for each process. It is envisioned to extend the solution by aggregating datatypes of collective calls into one view – showing the accessed file regions of all participants, each encoded with a different color. Thus, the overall activity of the

[1]See Section 2.3.4 and Section 2.3.5.

[2]`http://code.google.com/p/waf/`

collective operation would become visible. Additionally, the visualization tool could be extracted from Sunshot to assist scientists in creating and validating file views.

- Scalable visualization: Right now, Sunshot loads all trace information into memory, rendering tracing of large numbers of processes impossible. There are several potential approaches to increase the applicability of Sunshot. On the one hand, existing strategies could be incorporated: Similar to Vampir, multiple servers could collaborate to overcome this obstacle. On the other hand, due to the open nature of the file format and flexibility, other approaches might be better suited. For example, importing the data into an SQL database would naturally blend well with the filtering and aggregation functions of Sunshot.
- Dynamic data fragmentation: Since messages are fragmented into packets that are transferred with store-and-forward switching between the components of the network topology (see Section 5.2.3), a larger network granularity reduces the concurrent processing on the simulated devices and larger messages may delay short messages due to the FCFS scheduling. However, an advantage of increasing the network granularity is that simulation execution is faster. An adaptive system could be imagined, in which the packets are split depending on the current message size. Such a scheme must be designed with care, since it can become unfair easily, for example, deferring small messages on their route due to larger packets. This must be investigated carefully, because such a model would be unfair because packets of smaller messages would be deferred by the large packets of big messages.
- Compute model: The model of a compute job defines a number of instructions to perform the job; it corresponds directly to CPU frequency and duration of the compute job. Therefore, the variety of CPU architectures and their capabilities to process different workloads are not really reflected by the current compute model – which makes a prediction of a compute phase towards future systems difficult. With the Likwid extension, CPU counters are recorded that could be used by a new CPU model for a more differentiated prediction of the compute phase. For example, the instructions per cycle, Flop/s (single precision and double precision) and the memory bandwidth could be evaluated in an extended compute model (see also page 197). However, since execution time depends on many factors, predictability of any stochastic model is limited and, therefore, it must be assessed critically.
- Random characteristics: All presented hardware models use static characteristics, e.g., the latency of all packets is always identical. While this helps understanding the observations, it is problematic with the current approach to recreate certain dynamic and timing-dependent behavior. It is rather easy to provide additional models that add a small variability to the processing of individual operations – this could even be based on a specified probability distribution. With such an approach, timing effects of the I/O subsystem could be assessed further, while the existing models still provide a reference.

9.1. Conceivable MPI(-IO) Optimizations

With the help of simulation, the effectiveness of novel optimization strategies or potential optimizations could be quantified for arbitrary cluster systems. Further, it would allow assessing all the optimization strategies mentioned with the same application and hardware settings, thus it would make the research transparent by making algorithm evaluation reproducible. By identifying potential optimizations, simulation can help to push relevant modifications into future standards.

A small list of aspects that are not researched thoroughly, but could be assessed in the future:

- Semantic specifications in the standards must be obeyed. Therefore, there are two possible approaches: By relaxing the default MPI semantics, or by offering a choice about the semantics to the user. With an additional hint parameter for each collective function, this information could be exploited. An illustrating example of potential benefit for the library and the user is `MPI_Reduce()`. Semantics require that as long as the identical parameters are given to `MPI_Reduce()`, it should perform internal computations in the same order, independent of process mapping. This implies that locality of data cannot be exploited in all cases because floating point arithmetics is not associative, forcing communication and computation to be independent of the process mapping. With a hint, a user could accept that operations are performed optimally on the given hardware, or even allow nondeterministic execution.

- From a programmer's point of view, an extension to collective calls which permits less data to be received than announced, is more flexible and thus might avoid communication of data amount before the data itself is transferred. An example of how these capabilities can be used in an application is RAxML[3], an program which computes phylogenetic trees by applying optimization heuristics. While the algorithm proceeds with its optimizations, it is unknown how many better configurations are found, yet an upper bound exists. Candidates are transferred from a master process to all peers. In the current version of the algorithm, the full buffer is communicated although it is only partly filled, transferring irrelevant data. Yet, this is faster than broadcasting the number of elements before the actual data is communicated. A consequence of such a flexible communication would be that the executing collective algorithm must be chosen dynamically in an implementation, because the amount of data transferred would depend on the situation.
- Additional buffering of collective data could be done in all-to-one or one-to-all patterns to reduce the impact of early and late starters. With buffering, processes could continue processing even though some may not have invoked the collective function yet. However, careful attention must be paid to the memory usage because memory might be occupied by the application. Partly, this issue is addressed by MPI implementations which use regular point-to-point communication calls to implement collective calls, because these calls use the eager communication protocol if the message is small.
- Alternative client-side collective I/O optimizations are imaginable that improve the access pattern depending on the architecture. For example, right now the topology of the system is not taken into account for collective I/O. A topology-aware MPI implementation that takes multicore aspects into account could also improve performance of collective calls.

[3] `http://www.exelixis-lab.org/`

List of Figures

List of Tables

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the definition; numbers in roman refer to the pages where the entry is used.

Appendix

A.1. OTF API

An excerpt of the low-level OTF API is printed as a reference. The quoted API description is taken from OTF-1.7.0rc1 (2010-03-30). Repeated descriptions of parameters are removed from the API definition. All functions return an integer which is 1 on a successful invocation and 0 on error.

Listing A.1: OTF Writer Interface – quoted from the documentation shipped with OTF version 1.7.0rc1. Repeated descriptions of a parameter have been removed in this description.

```
int OTF_Writer_writeDefinitionComment (OTF_Writer * writer, uint32_t stream, const char * comment)
Write a comment record.
Parameters:
  writer  Initialized OTF_Writer instance.
  stream  Target stream identifier with 0 < stream <= number of streams as defined in OTF_Writer_open().
  comment Arbitrary comment string.

int OTF_Writer_writeDefProcess(OTF_Writer * writer, uint32_tstream, uint32_t process, const char *
  →name, uint32_t parent)
Write a process definition record.
Parameters:
  process Arbitrary but unique process identifier > 0.
  name    Name of the process e.g., "Process␣X".
  parent  Previously declared parent process identifier or 0 if process has no parent.

int OTF_Writer_writeDefProcessGroup(OTF_Writer * writer, uint32_t stream, uint32_t procGroup, const
  →char * name, uint32_t numberOfProcs, const uint32_t *procs)
Write a process group definition record. OTF supports groups of processes. Their main objective is to
  →classify processes depending on arbitrary characteristics. Processes can reside in multiple groups.
  →This record type is optional.
Parameters:
  procGroup       Arbitrary but unique process group identifier > 0.
  name            Name of the process group e.g., "Well␣Balanced".
  numberOfProcs   The number of processes in the process group.
  procs           Vector of process identifiers or previously defined process group identifiers as
    →defined with OTF_Writer_writeDefProcess() resp. OTF_Writer_writeDefProcessGroup.


int OTF_Writer_writeDefFunction (OTF_Writer * writer, uint32_t stream, uint32_t func, const char *
  →name, uint32_t funcGroup, uint32_t source)
Write a function definition record. Defines a function of the given name. Functions can optionally
  →belong to a certain function group to be defined with the OTF_Writer_writeDefFunctionGroup() call. A
  →source code reference can be added to the definition as well.
Parameters:
  func           Arbitrary but unique function identifier > 0.
  name           Name of the function e.g., "DoSomething".
  funcGroup      A function group identifier preliminary defined with
    →OTF_Writer_writeDefFunctionGroup() or 0 for no function group assignment.
  source         Reference to the function's source code location preliminary defined with
    →OTF_Writer_writeDefScl() or 0 for no source code location assignment.


int OTF_Writer_writeDefCollectiveOperation(OTF_Writer * writer, uint32_t stream, uint32_t collOp, const
  →char * name, uint32_t type)
Write a collective operation definition record.
Parameters:
  collOp         An arbitrary but unique collective op. identifier > 0.
  name           Name of the collective operation e.g., "MPI_Bcast".
  type           One of the five supported collective classes: OTF_COLLECTIVE_TYPE_UNKNOWN (default),
    →OTF_COLLECTIVE_TYPE_BARRIER, OTF_COLLECTIVE_TYPE_ONE2ALL, OTF_COLLECTIVE_TYPE_ALL2ONE,
    →OTF_COLLECTIVE_TYPE_ALL2ALL.

int OTF_Writer_writeBeginCollectiveOperation(OTF_Writer * writer, uint64_t time, uint32_t process,
  →uint32_t collOp, uint64_t matchingId, uint32_t procGroup, uint32_t rootProc, uint64_t sent, uint64_t
  →received, uint32_t scltoken)
Write a begin collective operation member record.
Parameters:
  writer       Initialized OTF_Writer instance.
  time         Time when collective operation was entered by member.
  process      Process identifier i.e., collective member.
  collOp       Collective identifier to be defined with OTF_Writer_writeDefCollectiveOperation().
  matchingId   Identifier for finding the associated end collective event record. It must be unique
    →within this procGroup.
  procGroup    Group of processes participating in this collective.
  rootProc     Root process if != 0.
  sent         Data volume sent by member or 0.
  received     Data volume received by member or 0.
  scltoken     Explicit source code location or 0.

int OTF_Writer_writeEnter(OTF_Writer * writer, uint64_t time, uint32_t function, uint32_t process,
  →uint32_t source)
Write a function entry record.
Parameters:
  time              The time when the function entry took place.
  function          Function to be entered as defined with OTF_Writer_defFunction.
  process           Process where action took place.
  sourceOptional    reference to source code.

int OTF_Writer_writeRecvMsg  (OTF_Writer * writer, uint64_t time, uint32_t receiver, uint32_t sender,
  →uint32_t procGroup, uint32_t tag, uint32_t length, uint32_t source)
Write a message retrieval record.
Parameters:
  writer      Initialized OTF_Writer instance.
  time        The time when the message was received.
  receiver    Identifier of receiving process.
  sender      Identifier of sending process.
  procGroup   Optional process-group sender and receiver belong to, '0' for no group.
  tag         Optional message type information.
  length      Optional message length information.
  source      Optional reference to source code.

int OTF_Writer_writeCounter(OTF_Writer * writer, uint64_t time, uint32_t process, uint32_t counter,
  →uint64_t value)
Write a counter measurement record.
Parameters:
  writer      Initialized OTF_Writer instance.
  time        Time when counter was measured.
  process     Process where counter measurment took place.
  counter     Counter which was measured.
  value       Counter value.

int OTF_Writer_writeBeginFileOperation(OTF_Writer * writer, uint64_t time, uint32_t process, uint64_t
  →matchingId, uint32_t scltoken)
Write a begin file operation record
Parameters:
  time         Start time of file operation.
  process      Process identifier > 0.
  matchingId   Operation identifier, used for finding the associated end file operation event record.
  scltoken     Optional reference to source code.

int OTF_Writer_writeEndFileOperation (OTF_Writer * writer, uint64_t  time, uint32_t  process, uint32_t
  →fileid, uint64_t  matchingId, uint64_t  handleId, uint32_t  operation, uint64_t  bytes, uint32_t
  →scltoken)
Write an end file operation record
Parameters:
  time          End time of file operation.
  process       Process identifier > 0.
  fileid        File identifier > 0.
  matchingId    Operation identifier, must match a previous start file operation event record.
  handleId      Unique file open identifier.
  operation     Type of file operation

int OTF_Writer_writeDefKeyValue(OTF_Writer * writer,
uint32_t stream, uint32_t key, OTF_Type type, const char * name, const char * description)
Write a key value definition record
Parameters:
  key             Arbitrary, unique identifier for the key value pair.
  type            Type of the key, e.g., Int32.
  name            Name of the key value pair.
  description     Description of the key value pair.
```

A.2. NTP Accuracy

A NTP client requests the current time from a server and updates (synchronizes) its clock with the response from this reference clock. However, due to communication overhead the received time information is not accurate. NTP provides an algorithm to predict the remote clock well enough for most purposes. A simplified version of the algorithm looks like this: A single request is sent to a server which responds with its current time. The actual time of the client is adjusted to the server's time plus the latency until the message is received on the client-side. Round-trip-time can be determined by the client and leads to a trivial estimation for latency. Thus, the client determines and subtracts this estimated latency from the time sent by the server.

The author excepts that accuracy of time synchronization is at least in the order of the average round-trip time[1]. The reason is that if a client measures the round-trip-time of a request, the bias to the server clock will be in the same order. By keeping a history of previous probes a client can discard responses that needed too much time. Since the average round-trip-time is about 0.08 ms on our cluster, it is expected that observed synchronization accuracy is at least in the order of 0.04 ms.

On the client-side, kernel support is required supporting accurate modification of the clock and reducing OS jitter. There are kernel modules available, however, these are not widely deployed and not installed on our cluster.

While this thesis is conducted the front node of our cluster synchronized time with the servers of the *Physikalisch-Technische Bundesanstalt (PTB)*[2]. The nodes synchronize their time only with the front node, thus the front node is the reference clock for all cluster nodes.

OpenNTPD version 4.2.4p8 is installed on all nodes. A daemon probes the synchronization partner in intervals and adjusts speed of the kernel clock slightly to compensate the time discrepancy between client and server. Since compensation is gradually applied by increasing or decreasing the frequency of the kernel clock slightly, it takes some time until the clocks are synchronized. Upon reboot of a machine, time is stored and read from the hardware clock. Therefore, after rebooting the clock must be re-synchronized by NTP, to speed up the process, `ntpdate` is invoked after a successful restart[3].

To validate the synchronization behavior, the NTP peer statistics of the nodes and the master have been enabled and analyzed. For a period of 200 hours the peer statistics are collected and for each NTP client the estimated time offset to its reference clock (either cluster front-end or the PTB servers) is plotted over time[4]. Obtained statistics are visualized in Figure A.1a. The diagram must be assessed keeping in mind that relative time offset between client and server of NTP is printed. As the nodes synchronize with the front-end, all nodes should behave similarly and try to minimize the time offset to the cluster front-end. It can be seen that in most cases the estimated discrepancy is in the order of the latency or at least the round-trip-time. However, sometimes the drift changes and the time offset increases to an order of several milliseconds which is unexpected.

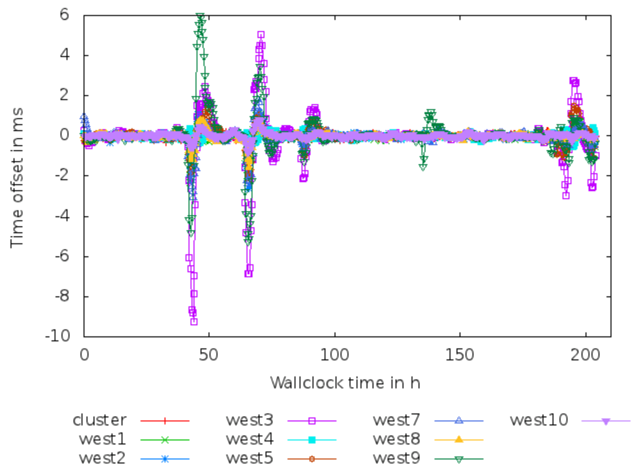
The monitoring of time discrepancy of one node and the time offset from cluster to the PTB servers are visualized in Figure A.1b. The client (west8) track the time discrepancy to the front-end. In most cases, the offset of west8 steers in the opposite direction of the cluster front-end; thus, one hardware clock is faster

[1]The actual NTP algorithm is more sophisticated. For example, to improve the estimation, round-trip-time of multiple packets can be measured. However, transfer time varies between subsequent executions – the average deviation of the latency is called jitter. If the jitter is much lower than the round-trip-time, the estimated latency approximates the true latency very well. Consequently, the deviation of network latency determines quality of time synchronization. By estimating the jitter, the number of requests and synchronization frequency can be dynamically adapted by NTP. Taking these factors into account, the time offset can typically be approximated much better than the latency.

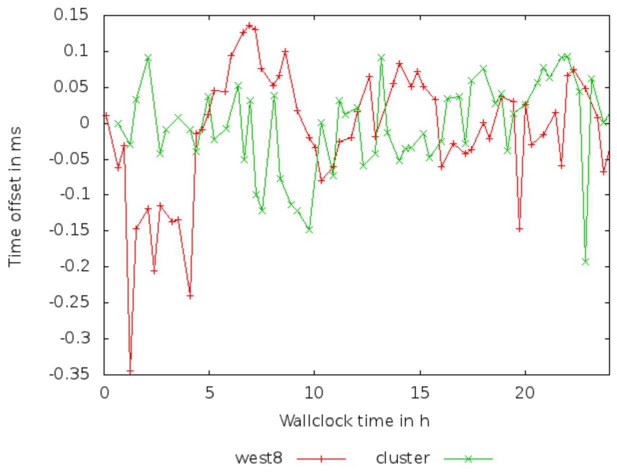
[2]This has been done for testing the accuracy of the protocol. To reduce load of PTB, this should be avoided.

[3]`ntpdate` requests the time information by using NTP and sets the time.

[4]The synchronization frequency is managed by each NTP client independently, thus it fluctuates between the clients. The peerstats file contains a timestamp (based on the local time) and time offset for each synchronization point. Between two consecutive synchronization points, each NTP client adjusts the local clock steadily to compensate for the estimated offset.



(a) All cluster nodes and the cluster front-end



(b) Synchronization one node and cluster front-end

Figure A.1.: Statistics measured by the NTP clients – time offsets between client and server is provided showing synchronization accuracy between front-end and the servers of PTB, and between nodes and the cluster front-end.

than the other. However, sometimes the offset is much higher. Interestingly, the time offset of the front-node to the PTB servers is comparable to the one measured on a compute node, although the front-node synchronizes over the Internet. Theoretically, the cluster front-end could lose contact to the servers at PTB or accuracy is expected to suffer due to jitter of the Internet connection.

To correlate network events, a time discrepancy in the order of the network latency is tolerable. However, sometimes the time discrepancy is much higher. Currently, the reason is unknown. The author tried several NTP options and different configurations, and an alternative NTP implementation, but the accuracy did not improve further. Still, the achieved accuracy is enough for this project. Because, if the time offset between independent devices is too large, then the XML header of the trace files can be manually adjusted. Further, Sunshot offers the capability to automatically adjust offset of individual timelines during visualization.

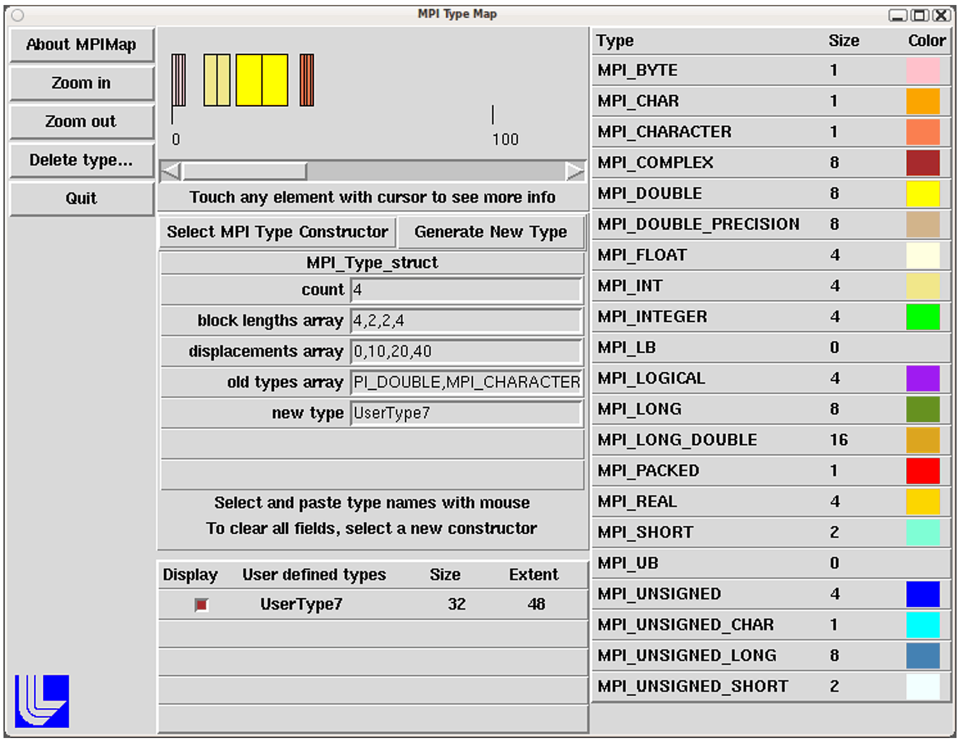
A.3. MPIMAP

MPIMAP [5] is a tool which illustrates the creation of MPI datatypes. It focuses on interactive creation of datatypes and their visualization, existing datatypes are not meant to be loaded by the tool, nor are trace events visualized with it. MPIMAP is built with *TCL/TK* and uses MPI to create and decode datatypes.

Upon execution the program creates one window; a screenshot of MPIMAP is shown in Figure A.2. The window is split into three columns: the buttons on the left permit some interaction with the datatypes, in the center of the window the datatype is visualized in an unrolled fashion at the top, below new datatypes can be created by providing all necessary parameters – new datatypes can refer to already created datatypes and primitive datatypes. User defined datatypes are listed at the bottom, the rightmost column shows the legend of the primitive datatypes and their size in memory.

The visualization engine for datatypes unrolls datatypes – there is no reference provided which indicates the parent datatypes. In the example screenshot, a structure with 4 items is shown with the parameters used for its creation. Several datatypes such as vector datatypes are completely unrolled. Multiple datatypes can be visualized in the drawing area at the same time, this overlapping fosters intercomparison. However, in general the unrolling of datatypes makes it harder for the user to grasp the relevant

[5] `https://computation.llnl.gov/casc/mpimap/UserGuide.html`



Touch any element with cursor to see more info

MPI_Type_struct

count	4
block lengths array	4,2,2,4
displacements array	0,10,20,40
old types array	PI_DOUBLE,MPI_CHARACTER
new type	UserType7

Select and paste type names with mouse
To clear all fields, select a new constructor

Display	User defined types	Size	Extent
🟥	UserType7	32	48

Type	Size	Color
MPI_BYTE	1	🟥
MPI_CHAR	1	🟧
MPI_CHARACTER	1	🟧
MPI_COMPLEX	8	🟫
MPI_DOUBLE	8	🟨
MPI_DOUBLE_PRECISION	8	🟫
MPI_FLOAT	4	⬜
MPI_INT	4	🟨
MPI_INTEGER	4	🟩
MPI_LB	0	
MPI_LOGICAL	4	🟪
MPI_LONG	8	🟩
MPI_LONG_DOUBLE	16	🟧
MPI_PACKED	1	🟥
MPI_REAL	4	🟨
MPI_SHORT	2	🟩
MPI_UB	0	
MPI_UNSIGNED	4	🟦
MPI_UNSIGNED_CHAR	1	🟦
MPI_UNSIGNED_LONG	8	🟦
MPI_UNSIGNED_SHORT	2	⬜

Figure A.2.: MPIMAP main window; the datatype is visualized centered on the top, the interface below allows interactive specification of new datataypes. Colors encode the underlying primitive datatype as listed in the legend on the right.

information – especially for large datatypes. The way datatypes are presented to the user differs from the hierarchical visualization within Sunshot. Furthermore, non-contiguous I/O accesses can be visualized with HDTrace as well.

A.4. Limiting the Amount of Free Memory

The Linux kernel caches data extensively, therefore, writing data into the file system PVFS is cached as well. Since a Westmere node has 12 GByte of main memory, I/O experiments which are designed to hit the disk must access a lot of data, which is time-consuming. Therefore, a small tool has been developed which checks and restricts available memory to a well defined value. The code of this mem-eater tool is printed in Listing A.2. The amount of free memory which should remain is specified as a parameter to the program. By reserving all memory below this limit, further I/O is restricted to the free memory. It also forces the kernel to free page cache and buffer space beyond this limitation.

Listing A.2: Code of the mem-eater tool.

```
/**
 * This tool limits the amount of free memory by allocating memory.
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

/*
 * Return the current amount of memory for a given type from /proc (e.g., size of the page cache).
 * Memory statistics are read from /proc/meminfo, content of the file looks like:
 *   MemTotal:        3915436 kB
 *   MemFree:         1257648 kB
 *   Buffers:          237748 kB
 *   Cached:           1246012 kB
 */
long getMemoryFor(char * what){
        char buff[1024];

        int fd = open("/proc/meminfo", O_RDONLY);
        int ret = read(fd, buff, 1023);
        close(fd);

        buff[ret>1023 ? 1023: ret] = 0;
        // find the particular memory type.
        char * line = strstr(buff, what);

        if (! line){
                printf("Error␣%s␣not␣found␣in␣%s␣\n", what, buff);
                exit(1);
        }

        // Parse the integer after the type.
        line += strlen(what) + 1;
        while(line[0] == '␣'){
                line++;
        }

        int pos = 0;
        while(line[pos] != '␣'){
                pos++;
        }
        line[pos] = 0;

        // Convert the string to integer.
        return atoi(line);
}

/* Page cache and buffer cache can be considered as free memory. */
long getFreeRamKB(){
        return getMemoryFor("\nMemFree:") +getMemoryFor("\nCached:") + getMemoryFor("\nBuffers:");
}

int main(int argc, char ** argv){
        if(argc != 2){
                printf("Syntax:␣[AmountOfRFreeRAM␣in␣MByte]\n");
                exit(1);
        }
        // Parse command line argument.
        long long int maxRAMinKB = atoi(argv[1]) * 1024;

        printf("This␣app␣uses␣enough␣RAM␣until␣at␣most␣%lld␣MByte␣are␣free\n",(long long int) maxRAMinKB
          → / 1024);

        long long int currentRAMinKB = getFreeRamKB();

        printf ("starting␣to␣malloc␣RAM␣currently␣\n␣%lld␣-␣%lld\n", currentRAMinKB, maxRAMinKB);

        // Reserve additional memory blocks in 500 KiB chunks until the amount of free memory matches.
        while(currentRAMinKB > maxRAMinKB){
                long long int delta = currentRAMinKB - maxRAMinKB;
                long long int toMalloc = (delta < 500 ? delta : 500) * 1024;

                char * allocP = malloc(toMalloc);
                if(allocP == 0){
                        printf("could␣not␣allocate␣more␣RAM␣-␣retrying␣-␣free:%lld␣\n", currentRAMinKB);
                        sleep(5);
                }else{
                        memset(allocP, '1', toMalloc);
                }
                currentRAMinKB = getFreeRamKB();
        }

        printf ("Finished␣now␣\n␣%lld␣-␣%lld\n", currentRAMinKB, maxRAMinKB);

        while(1){
                sleep(60*60);
        }
}
```

A.5. Memory Benchmark

The source code of the memory benchmark that is used in Chapter 3 is provided in Listing A.3.

Listing A.3: Code of the `memory-bandwidth` tool.

```c
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/**
 * The benchmark measures the time for accessing a given amount of data in an array.
 * Several access patterns (kernels) are supported.
 */


void getTime(struct timespec * start){
     clock_gettime(CLOCK_MONOTONIC, start);
}

/* Compute a time difference. */
double timeDiffInS(struct timespec *end, struct timespec * start){
       long tmp = (end->tv_sec - start->tv_sec ) * 1000 * 1000 * 1000+ (end->tv_nsec - start->tv_nsec);
       return tmp * 0.001 * 0.001 * 0.001;
}

/* Output all values of the array. */
void printResults(long count, double * time, double size, char const *  name){
  size = size / 1000 / 1000;
  printf("%s␣%f␣size␣timing:", name,  size);

  printf("%.8f", time[0]);
  for(long i=1; i < count; i++){
        printf(",%.8f", time[i]);
  }
  printf("\n");
}

int main (int argc, char *argv[]){
  long size = 1000000000;
  long repeats = 100;
  long numberOfTimingsPerIteration = 1000;
  if (argc > 1){
    size = atol(argv[1]);
  }
  if (argc > 2){
    repeats = atol(argv[2]);
  }
  if (argc > 3){
    numberOfTimingsPerIteration = atol(argv[3]);
  }
  printf("size:␣%ld␣repeats:%ld␣numberOfTimingsPerIteration:%ld␣\n", size, repeats,
    →numberOfTimingsPerIteration);

  long * buffer = malloc(size);

  struct timespec startTime;
  getTime(&startTime);

  // memset aggregated throughput and initialization:
  for(long r=0 ; r < repeats; r++){
        memset(buffer, r, size);
  }
  struct timespec endTime;
  getTime(&endTime);

  double e = timeDiffInS(& endTime,& startTime);
  printf("Aggregated␣time:␣size:␣%ld␣repeats:%ld␣time:%fs␣MB/s:%f\n", size, repeats, e, repeats * size /
    → 1024 / 1024 / e);


  // An array for the timings.
  double * time = malloc(repeats * sizeof(double) * numberOfTimingsPerIteration);

  long bytesPerTiming = (long) size / numberOfTimingsPerIteration;
  long maxIterCountPerTiming = bytesPerTiming / sizeof(long);

  // Write kernel:
  for(long r=0 ; r < repeats; r++){
        volatile register long value = 0;

        for(int t = 0; t < numberOfTimingsPerIteration; t++){
                long * tmp = & buffer[maxIterCountPerTiming * t];

                getTime(&startTime);
                for(long c = 0; c < maxIterCountPerTiming ; c++){
                        tmp[c] = value;
                }
                getTime(&endTime);

                e = timeDiffInS(& endTime,& startTime);

                time[r*numberOfTimingsPerIteration + t] = e;
        }
  }
  printResults(repeats*numberOfTimingsPerIteration, time, bytesPerTiming, "Write␣64␣byte");

  //  Read kernel:
  for(long r=0 ; r < repeats; r++){
        volatile register long value = 0;

        for(int t = 0; t < numberOfTimingsPerIteration; t++){
                long * tmp = & buffer[maxIterCountPerTiming * t];

                getTime(&startTime);
                for(long c = 0; c < maxIterCountPerTiming ; c++){
                        value = tmp[c];
                }
                getTime(&endTime);
                e = timeDiffInS(& endTime,& startTime);

                time[r*numberOfTimingsPerIteration + t] =  e;
        }
  }
  printResults(repeats*numberOfTimingsPerIteration, time, bytesPerTiming, "Read␣64␣byte");

  //  RWW kernel:
  for(long r=0 ; r < repeats; r++){
        volatile register long value = 0;

        for(int t = 0; t < numberOfTimingsPerIteration; t++){
                long * tmp = & buffer[maxIterCountPerTiming * t];

                getTime(&startTime);
                for(long c = 0; c < maxIterCountPerTiming ; c++){
                        value = tmp[c];
                }
                for(long c = 0; c < maxIterCountPerTiming ; c++){
                        tmp[c] = value;
                }
                for(long c = 0; c < maxIterCountPerTiming ; c++){
                        tmp[c] = value;
                }
                getTime(& endTime);
                e = timeDiffInS(& endTime,& startTime);

                time[r*numberOfTimingsPerIteration + t] =  e;
        }
  }
  printResults(repeats*numberOfTimingsPerIteration, time, bytesPerTiming, "Read␣Write␣Write␣64␣byte");
  return 0;
}
```

A.6. MPI Configuration

This section gives more information about the configuration of the MPI implementations that have been used in the experiments. For the instrumented MPICH2 version information is given in Listing A.4 and for Open MPI in Listing A.5 – information has been gathered directly from the binary versions.

Listing A.4: Output of `mpiexec -version`.

```
HYDRA build details:
    Version:                                 1.3.1
    Release Date:                            Wed Nov 17 10:48:28 CST 2010
    CC:                                      gcc  -I.../cur/src/mpl/include -I.../cur/src/mpl/include -I
      →.../cur/src/openpa/src -I.../cur/src/openpa/src -I.../cur/src/mpid/ch3/include -I.../cur/src/mpid
      →/ch3/include -I.../cur/src/mpid/common/datatype -I.../cur/src/mpid/common/datatype -I.../cur/src/
      →mpid/common/locks -I.../cur/src/mpid/common/locks -I.../cur/src/mpid/ch3/channels/nemesis/include
      → -I.../cur/src/mpid/ch3/channels/nemesis/include -I.../cur/src/mpid/ch3/channels/nemesis/nemesis/
      →include -I.../cur/src/mpid/ch3/channels/nemesis/nemesis/include -I.../cur/src/mpid/ch3/channels/
      →nemesis/nemesis/utils/monitor -I.../cur/src/mpid/ch3/channels/nemesis/nemesis/utils/monitor -I
      →.../cur/src/util/wrappers -I.../cur/src/util/wrappers    -O2    -lrt -lpthread
    CXX:
    F77:
    F90:
    Configure options:                       '--prefix=/opt/hdtrace/1.0/MPICH2-normal' '--without-mpe' '
      →--disable-option-checking' 'CC=gcc' 'CFLAGS=␣-O2' 'LDFLAGS=␣' 'LIBS=-lrt␣-lpthread␣' 'CPPFLAGS=␣-
      →I.../cur/src/mpl/include␣-I.../cur/src/mpl/include␣-I.../cur/src/openpa/src␣-I.../cur/src/openpa/
      →src␣-I.../cur/src/mpid/ch3/include␣-I.../cur/src/mpid/ch3/include␣-I.../cur/src/mpid/common/
      →datatype␣-I.../cur/src/mpid/common/datatype␣-I.../cur/src/mpid/common/locks␣-I.../cur/src/mpid/
      →common/locks␣-I.../cur/src/mpid/ch3/channels/nemesis/include␣-I.../cur/src/mpid/ch3/channels/
      →nemesis/include␣-I.../cur/src/mpid/ch3/channels/nemesis/nemesis/include␣-I.../cur/src/mpid/ch3/
      →channels/nemesis/nemesis/include␣-I.../cur/src/mpid/ch3/channels/nemesis/nemesis/utils/monitor␣-I
      →.../cur/src/mpid/ch3/channels/nemesis/nemesis/utils/monitor␣-I.../cur/src/util/wrappers␣-I.../cur
      →/src/util/wrappers'
    Process Manager:                         pmi
    Bootstrap servers available:             ssh rsh fork slurm ll lsf sge persist
    Binding libraries available:             hwloc plpa
    Resource management kernels available:   none pbs
    Checkpointing libraries available:
    Demux engines available:                 poll select
```

Listing A.5: Output of `ompi_info` (excerpt).

```
Package: Open MPI root@cluster.wr.informatik.uni-hamburg.de Distribution
                Open MPI: 1.5.3
   Open MPI SVN revision: r24532
   Open MPI release date: Mar 16, 2011
                Open RTE: 1.5.3
   Open RTE SVN revision: r24532
   Open RTE release date: Mar 16, 2011
                    OPAL: 1.5.3
       OPAL SVN revision: r24532
       OPAL release date: Mar 16, 2011
            Ident string: 1.5.3
                  Prefix: /opt/openmpi/1.5.3
 Configured architecture: x86_64-unknown-linux-gnu
          Configure host: cluster.wr.informatik.uni-hamburg.de
           Configured by: root
           Configured on: Wed May 11 10:13:20 CEST 2011
          Configure host: cluster.wr.informatik.uni-hamburg.de
                Built by: root
                Built on: Mi 11. Mai 10:25:21 CEST 2011
              Built host: cluster.wr.informatik.uni-hamburg.de
              C bindings: yes
            C++ bindings: yes
      Fortran77 bindings: yes (all)
      Fortran90 bindings: yes
 Fortran90 bindings size: small
              C compiler: gcc
     C compiler absolute: /usr/bin/gcc
  C compiler family name: GNU
      C compiler version: 4.4.3
            C++ compiler: g++
   C++ compiler absolute: /usr/bin/g++
      Fortran77 compiler: gfortran
  Fortran77 compiler abs: /usr/bin/gfortran
      Fortran90 compiler: gfortran
  Fortran90 compiler abs: /usr/bin/gfortran
             C profiling: yes
           C++ profiling: yes
     Fortran77 profiling: yes
     Fortran90 profiling: yes
          C++ exceptions: no
          Thread support: posix (mpi: no, progress: no)
           Sparse Groups: no
  Internal debug support: no
  MPI interface warnings: no
     MPI parameter check: runtime
Memory profiling support: no
Memory debugging support: no
         libltdl support: yes
   Heterogeneous support: no
 mpirun default --prefix: no
         MPI I/O support: yes
       MPI_WTIME support: gettimeofday
     Symbol vis. support: yes
          MPI extensions: affinity example
   FT Checkpoint support: no (checkpoint thread: no)
  MPI_MAX_PROCESSOR_NAME: 256
    MPI_MAX_ERROR_STRING: 256
     MPI_MAX_OBJECT_NAME: 64
        MPI_MAX_INFO_KEY: 36
        MPI_MAX_INFO_VAL: 256
       MPI_MAX_PORT_NAME: 1024
  MPI_MAX_DATAREP_STRING: 128
```

A.7. Software Used to Write this Thesis

Several software tools have been used in order to write this thesis in LaTeX. Every artifact is created under under *Ubuntu*. The KDE editor *kile* assisted me in writing the LaTeX. Several graphics are created with *OpenOffice* (LibreOffice), a few diagrams are created with *yEd*[6] and *freemind*[7]. Experimental results are either plotted with *gnuplot* or with *R*[8]. Created screenshots have been edited with *Gimp*. A few cliparts that are licensed under a public domain license have been taken from the *Open Clipart Library*[9]. Some OpenOffice shapes are taken from `http://www.lautman.net/mark/coo/index.html` – these shapes are covered by LGPLv3 license and designed by Mark Lautman.

Last Words of the Author

While I hoped to evaluate the simulator on larger systems and with more applications, running the presented experiments on our cluster consumed far more time than expected.

Since most literature has been accessed on-line, the verification of page numbers and book titles of cited work is hard – often available BibTeX records from the publisher is inconsistent to data offered by other official sources. I tried to verify this data carefully, however, sometimes the information offered by the publisher must be trusted.

Writing down work done in the context of this project was the hardest part for me. Therefore, I appreciate any critics from the reader – feel free to report potential mistakes to me.

[6] `http://www.yworks.com/`
[7] `http://freemind.sourceforge.net/wiki/index.php/Main_Page`
[8] `http://http://www.r-project.org/`
[9] `http://www.openclipart.org/`

Eidesstattliche Erklärung

§

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, wurden als solche kenntlich gemacht.

Diese Versicherung bezieht sich auch auf alle in der Arbeit enthaltenen Grafiken, Zeichnungen und andere bildliche Darstellungen.

Diese Doktorarbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt.

Ort, Datum

Unterschrift