Dynamically Adaptable I/O Semantics for High Performance Computing

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

Michael Kuhn

aus Esslingen

Hamburg, November 2014

Gutachter:
Prof. Dr. Thomas Ludwig
Prof. Dr. Norbert Ritter

Datum der Disputation: 2015-04-08

Abstract

File systems as well as libraries for input/output (I/O) offer interfaces that are used to interact with them, albeit on different levels of abstraction. While an interface's syntax simply describes the available operations, its semantics determines how these operations behave and which assumptions developers can make about them. There are several different interface standards in existence, some of them dating back decades and having been designed for local file systems; one such representative is POSIX.

Many parallel distributed file systems implement a POSIX-compliant interface to improve portability. Its strict semantics is often relaxed to reach maximum performance which can lead to subtly different behavior on different file systems. This, in turn, can cause application misbehavior that is hard to track down. All currently available interfaces follow a fixed approach regarding semantics, making them only suitable for a subset of use cases and workloads. While the interfaces do not allow application developers to influence the I/O semantics, applications could benefit greatly from the possibility of being able to adapt them to their requirements.

The work presented in this thesis includes the design of a novel I/O interface called JULEA. It offers support for dynamically adaptable semantics and is suited specifically for HPC applications. The introduced concept allows applications to adapt the file system behavior to their exact I/O requirements instead of the other way around. The general goal is an interface that allows developers to specify *what* operations should do and *how* they should behave – leaving the actual realization and possible optimizations to the underlying file system. Due to the unique requirements of the proposed interface, a prototypical file system is designed and developed from scratch.

The new I/O interface and file system prototype are evaluated using both synthetic benchmarks and real-world applications. This ensures covering both specific optimizations made possible by the file system's additional knowledge as well as the applicability for existing software. Overall, JULEA provides data and metadata performance comparable to that of other established parallel distributed file systems. However, in contrast to the existing solutions, its flexible semantics allows it to cover a wider range of use cases in an efficient way.

The results demonstrate that there is need for I/O interfaces that can adapt to the requirements of applications. Even though POSIX facilitates portability, it does not seem to be suited for contemporary HPC demands. JULEA presents a first approach of how application-provided semantical information can be used to dynamically adapt the file system's behavior to the applications' I/O requirements.

Kurzfassung

Dateisysteme und Bibliotheken für Ein-/Ausgabe (E/A) stellen Schnittstellen für den Zugriff auf unterschiedlichen Abstraktionsebenen bereit. Während die Syntax einer Schnittstelle lediglich deren Operationen festlegt, beschreibt ihre Semantik das Verhalten der Operationen. Es existieren mehrere Standards für E/A-Schnittstellen, die teilweise mehrere Jahrzehnte alt sind und für lokale Dateisysteme entwickelt wurden; ein solcher Vertreter ist POSIX.

Viele parallele verteilte Dateisysteme implementieren eine POSIX-konforme Schnittstelle, um die Portabilität zu erhöhen. Ihre strikte Semantik wird oft relaxiert, um die maximal mögliche Leistung erreichen zu können, was aber zu subtil unterschiedlichem Verhalten führen kann. Dies kann wiederum zu schwer nachzuvollziehenden Fehlverhalten der Anwendungen führen. Alle momentan verfügbaren Schnittstellen verfolgen einen statischen Semantikansatz, wodurch sie nur für bestimmte Anwendungsfälle geeignet sind. Während die Schnittstellen keine Möglichkeit für Anwendungsentwickler bieten, die Semantik zu beeinflussen, wäre ein solcher Ansatz hilfreich für Anwendungen, um die Dateisysteme an ihre Anforderungen anpassen zu können.

Die vorliegende Dissertation beschäftigt sich mit dem Entwurf und der Entwicklung einer neuartigen E/A-Schnittstelle namens JULEA. Sie erlaubt es, die Semantik dynamisch anzupassen und ist speziell für HPC-Anwendungen optimiert. Das entwickelte Konzept erlaubt es Anwendungen, das Verhalten des Dateisystems an die eigenen E/A-Bedürfnisse anzupassen. Das Ziel ist eine Schnittstelle, die es Entwicklern gestattet zu spezifizieren, *was* Operationen tun sollen und *wie* sie sich verhalten sollen; die eigentliche Umsetzung und mögliche Optimierungen werden dabei dem Dateisystem überlassen. Aufgrund der einzigartigen Anforderungen der Schnittstelle wird außerdem ein prototypisches Dateisystem entworfen und entwickelt.

Das Dateisystem und die Schnittstelle werden mit Hilfe von synthetischen Benchmarks und praxisnahen Anwendungen evaluiert. Dadurch wird sichergestellt, dass sowohl spezifische Optimierungen als auch die Tauglichkeit für existierende Software überprüft werden. JULEA erreicht eine vergleichbare Daten- und Metadatenleistung wie etablierte parallele verteilte Dateisysteme, kann durch seine flexible Architektur aber einen größeren Teil von Anwendungsfällen effizient abdecken.

Die Resultate zeigen, dass E/A-Schnittstellen benötigt werden, die sich an die Anforderungen von Anwendungen anpassen. Obwohl der POSIX-Standard Vorteile bezüglich der Portabilität von Anwendungen bietet, ist seine Semantik nicht mehr für heutige HPC-Anforderungen geeignet. JULEA stellt einen ersten Ansatz dar, der es erlaubt, das Dateisystemverhalten an die Anwendungsanforderungen anzupassen.

Acknowledgments

First of all, I would like to thank my advisor Prof. Dr. Thomas Ludwig for supporting and guiding me in this endeavor. I first came into contact with high performance computing and file systems during his advanced software lab about the evaluation of parallel distributed file systems in the winter semester of 2005/2006 and have been interested in this topic ever since.

I am grateful for the many fruitful discussions, collaborations and fun times with my friends and colleagues from the research group and the DKRZ. In addition to my family, I also want to thank my wife Manuela who readily relocated to Hamburg with me. Special thanks to Konstantinos Chasapis, Manuela Kuhn and Thomas Ludwig for proofreading my thesis and giving me valuable feedback.

Last but not least, I would also like to thank everyone that has contributed to JULEA or this thesis in one way or another: Anna Fuchs for creating JULEA's correctness and performance regression framework and helping with the partdiff benchmarks, Sandra Schröder for building LEXOS and JULEA's corresponding storage backend, and Alexis Engelke for implementing the reordering logic for the ordering semantics.

"There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened."

Douglas Adams – The Restaurant at the End of the Universe

Contents

Chapter 1.

Introduction

In this chapter, basic background information from the fields of high performance computing and parallel distributed file systems will be introduced. This includes common use cases and key architectural features. Additionally, the concepts of I/O interfaces and semantics will be briefly explained. A special focus lies on the deficiencies of today's interfaces which do not allow applications to modify the semantics of I/O operations according to their needs.

1.1. High Performance Computing

High performance computing (HPC) is a branch of informatics that is concerned with the use of supercomputers and has become an increasingly important tool for computational science. Supercomputers combine the power of hundreds to thousands of central processing units (CPUs) to provide enough computational power to tackle especially complex scientific problems.[1] They are used to conduct large-scale computations and simulations of complex systems from basically all branches of the natural and technical sciences, such as meteorology, climatology, particle physics, biology, medicine and computational fluid dynamics. Recently, other fields such as economics and social sciences have also started to make use of supercomputers.

As these simulations have become more and more accurate and thus realistic over the last years, their demands for computational power have also increased. Because CPU clock rates are no longer increasing [Ros08] and the number of CPUs per computer is limited, it has become necessary to distribute the computational work across multiple CPUs and computers. Therefore, these computations and simulations are usually realized in the form of parallel applications. While all CPUs in the same computer can make use of threads, it is common to employ message passing between different computers. Large-scale applications typically use a combination of both to distribute work across a supercomputer.

Due to the heavy dependency of scientific applications on floating-point arithmetic operations, floating-point operations per second (FLOPS) are used to designate a

[1] CPUs typically contain multiple cores and the fastest supercomputers incorporate millions of cores.

supercomputer's computational power and have replaced the simpler instructions per second (IPS) metric. The performance development can be most easily observed using the so-called TOP500 list. It ranks the world's most powerful supercomputers according to their computational performance in terms of FLOPS as measured by the HPL benchmark [DLP03]. The performance of the supercomputers ranked number 1 and 500 as well as the sum of all 500 systems during 1993–2014 is shown in Figure 1.1 using a logarithmic y-axis. As can be seen, throughout the history of the TOP500 list, the computational power of supercomputers has been increasing exponentially, doubling roughly every 14 months. The currently fastest supercomputers reach rates of several PetaFLOPS – that is, more than 10^{15} FLOPS.

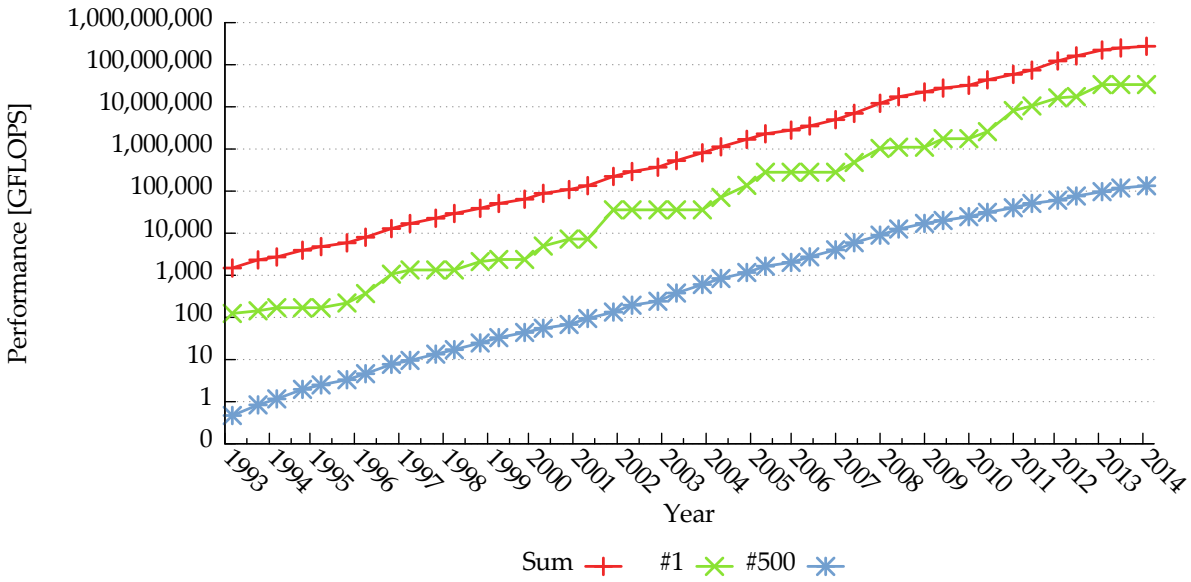


Figure 1.1.: TOP500 performance development from 1993–2014 [The14c]

While the increasing computational power has allowed more accurate simulations to be performed, this has also caused the simulation results to grow in size. Even though data about the supercomputers' storage systems is often not as readily available, the highest ranking supercomputers currently have storage systems that are around 10–60 petabytes (PB) in size and have throughputs in the range of terabytes (TB)/s. The main memory of such systems usually already exceeds 1 PB. Consequently, simply dumping the supercomputer's complete main memory to the storage system – which is a very common process called *checkpointing* – can already take several minutes in the best case.[2] However, it is also possible for checkpointing to take up to several hours for imbalanced system configurations. Many HPC applications frequently write checkpoints to be able to restart in case of errors due to their long runtimes. Additionally, job execution is typically coordinated by so-called *job schedulers*; these

[2] Storing 1 PB with 1 TB/s takes 1,000 s, which equals 16:40 min.

schedulers generally only allow allocations of up to several hours, that is, long-running applications have to write checkpoints in order to split up their runtime into several chunks. Due to the large amounts of data produced by parallel applications, the tools to perform analysis and other post-processing often have to be parallel applications themselves. Therefore, high performance input/output (I/O) is an important aspect because storing and retrieving such large amounts of data can greatly affect the overall performance of these applications.

For example, a parallel application may perform iterative calculations on a large matrix that is distributed across a number of processes on different computers. To be able to comprehend the application's calculations, it is often necessary to output the matrix every given number of iterations. This obviously influences the application's runtime since the matrix has to be completely written before the program can continue to run. A common access pattern produced by these applications involves many parallel processes, each performing non-overlapping access to a shared file. Because each process is responsible exclusively for a part of the matrix, each process only accesses the part of the file related to the data this specific process holds.

However, the I/O requirements of parallel applications can vary widely: While some applications process large amounts of input data and produce relatively small results, others might work using a small set of input data and output large amounts of data; additionally, the aforementioned data can be spread across many small files or be concentrated into few large files. Naturally, any combination thereof is also possible. Additionally, the source of data is diverse: Detectors and sensors might deliver data at high rates or parallel applications may produce huge amounts of in-silico data. As can be seen, the different requirements of parallel applications can make high demands on supercomputers' storage systems.

1.2. Parallel Distributed File Systems

The storage system used by the parallel applications is usually made available by a parallel distributed file system. File systems provide an abstraction layer between the applications and the actual storage hardware, such that application developers do not have to worry about the organizational layout or technology of the underlying storage hardware. Additionally, file systems usually offer standardized access interfaces to reduce portability issues. To meet the high demands of current HPC applications, parallel distributed file systems offer efficient parallel access and distribute data across multiple storage devices. On the one hand, parallel access allows multiple clients to cooperatively work with the same data concurrently, which is a key requirement for the parallel applications used today. On the other hand, the distribution of data allows to use both the combined storage capacity as well as the combined throughput of the underlying storage devices. This is necessary to be able to build the huge storage

systems with capacities of several PB and throughputs in the range of TB/s described above. Figure 1.2 illustrates these two concepts: Multiple clients access a single file concurrently while the file's data is distributed across several storage devices.

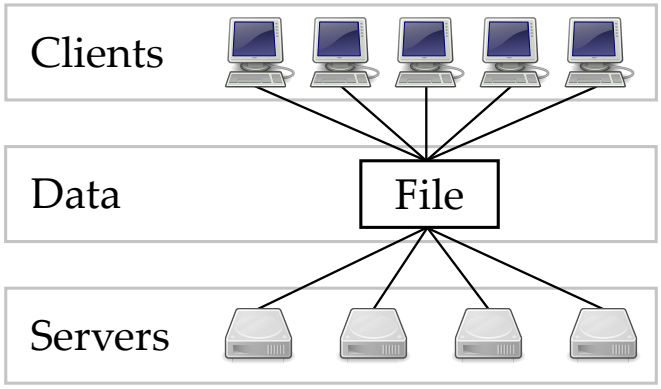


Figure 1.2.: Parallel access from multiple clients and distribution of data

While home directories and smaller amounts of data are sometimes still stored on non-parallel file systems such as NFS[3], large-scale I/O is almost always backed by a parallel distributed file system. Two of the most widely used parallel distributed file systems today are Lustre [Clu02] and GPFS [SH02], which power most of the TOP500's supercomputers [The14c].

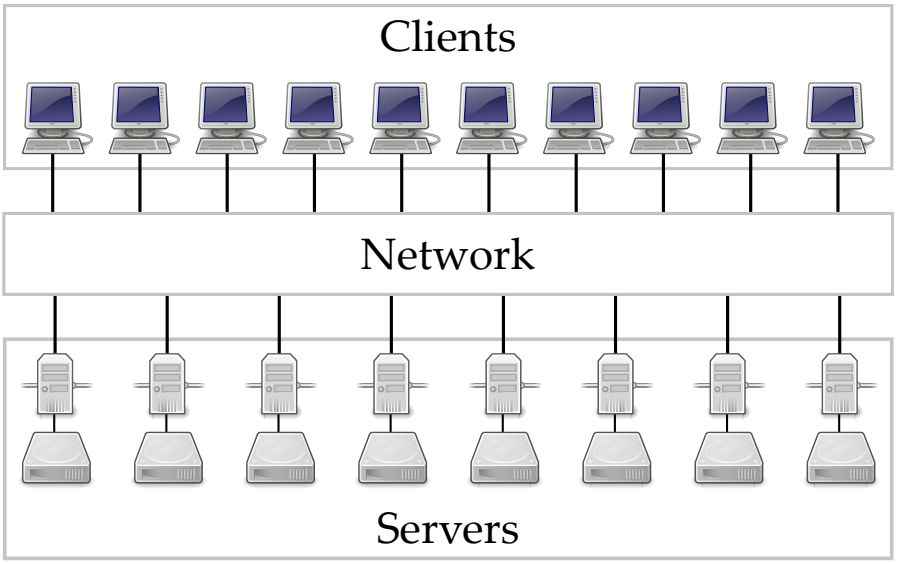


Figure 1.3.: Parallel distributed file system

Figure 1.3 shows the general architecture of an exemplary parallel distributed file system. Machines can be divided into two groups: clients and servers. The clients

[3] Network File System

have access to the parallel distributed file system and are used to execute the parallel applications. They usually do not have storage attached locally and have to perform all I/O by sending requests to the server machines via the network. The servers are attached to the actual file system storage and process client requests. These servers can be full-fledged computers or simpler storage controllers.

Because all I/O operations have to pass the network, they can be expensive to perform in such an architecture. This is due to the fact that the network introduces additional latency and throughput constraints. However, newer concepts such as burst buffers are increasingly used to improve this situation [LCC+12]. Details about the architecture of common parallel distributed file systems – including the different kinds of servers and the distribution of data – will be given in Chapter 2.

1.3. Input/Output Interfaces and Semantics

Parallel distributed file systems provide one or more I/O interfaces that can be used to access data within the file system. Usually at least one of them is standardized, while additional proprietary interfaces might offer improved performance at the cost of portability. Additionally, higher-level I/O interfaces are provided by I/O libraries and offer additional features usually not found in file systems. Popular interface choices include POSIX[4], MPI-IO, NetCDF[5] and HDF[6]. Almost all the I/O interfaces found in HPC today offer simple byte- or element-oriented access to data and thus do not have any a priori information about what kind of data the applications access and how the high-level access patterns look like. However, this information can be very beneficial for optimizing the performance of I/O operations.

There are notable exceptions, though: For instance, ADIOS[7] outsources the I/O configuration into an external XML[8] file that can be used to describe which data structures should be accessed and how the data is organized. On the one hand, this additional information enables the I/O library to provide more sophisticated access possibilities for developers and users. On the other hand, the knowledge can be used by the library to efficiently handle I/O requests.

However, even these more advanced I/O interfaces do not offer support to specify additional semantical information about the applications' behavior and requirements. Due to this lack of knowledge about application behavior, optimizations are often based on heuristic assumptions which may or may not reflect the actual behavior.

The I/O stack is realized in the form of layers, with the typical view of a developer being shown in Figure 1.4 on the next page. The parallel application uses a high-level

[4] Portable Operating System Interface
[5] Network Common Data Form
[6] Hierarchical Data Format
[7] Adaptable IO System
[8] Extensible Markup Language

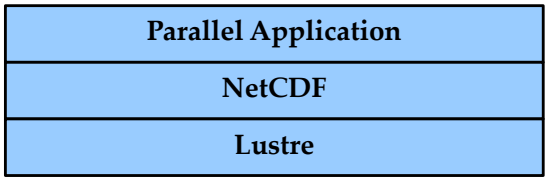


Figure 1.4.: Simplified view of the I/O stack

I/O interface – in this case, NetCDF – that writes its data to a file system – in this case, Lustre. The underlying idea of this concept is that developers only have to care about the uppermost layer and can safely ignore the other ones; in fact, it should be possible to exchange the lower layers without any behavioral change. However, in reality, the I/O stack is much more complex and features a multitude of intermediate layers that have subtle influences on the I/O system's behavior. Additionally, it is necessary to take all layers into account to obtain optimal performance. This can lead to I/O behavior that is very hard to predict, let alone explain and understand. Consequently, it is a difficult task to improve potential performance problems. The whole I/O stack and its problems will be described in more detail in Chapter 2.

While the I/O interface defines which I/O operations are available, the I/O semantics describes and defines the behavior of these operations. Usually each I/O interface is accompanied by a set of I/O semantics, tailored to this specific interface. The POSIX I/O semantics is probably both the oldest and the most widely used semantics, even in HPC. However, due to being designed for traditional local file systems, it imposes unnecessary restrictions on today's parallel distributed file systems. POSIX's very strict consistency requirements are one of these restrictions and can lead to performance bottlenecks in distributed environments.

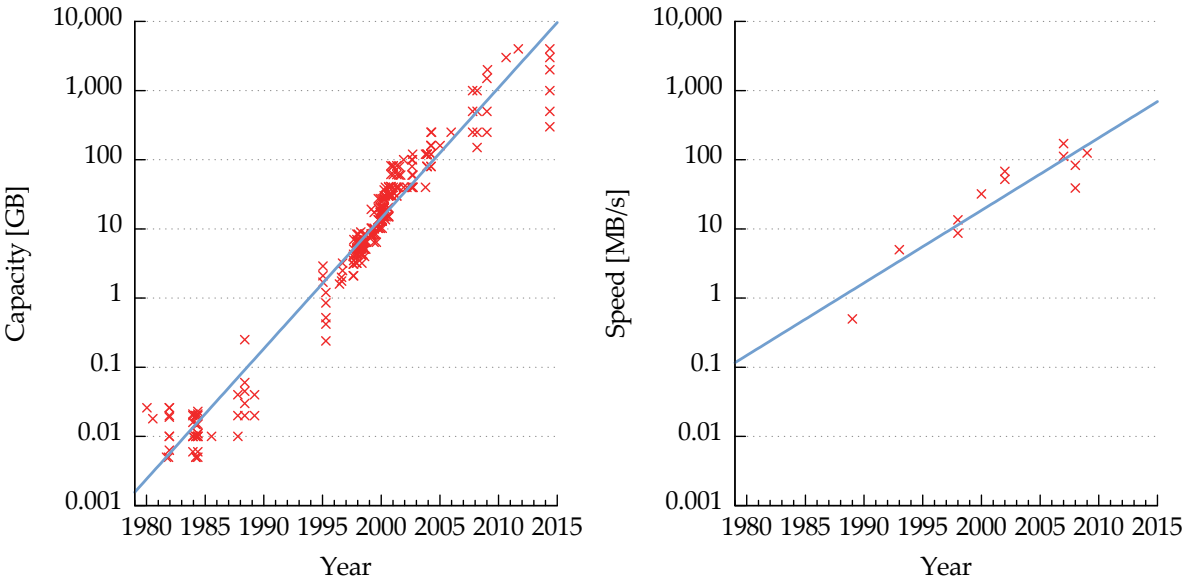
Parallel distributed file systems often implement the strictest I/O semantics – that is, the POSIX I/O semantics – to accommodate applications that require it or simply expect it to be available for portability reasons. However, this can lead to suboptimal behavior for many use cases because its strictness is often not necessary. Even though application developers usually know their applications' requirements and could easily specify them for improved performance, current I/O interfaces and file systems do not provide appropriate facilities for this task.

1.4. Motivation

Performing I/O efficiently is becoming an increasingly important problem. CPU speed and hard disk drive (HDD) capacity have roughly increased by factors of 500 and 100 every 10 years, respectively [The14c, Wik14e]. The speed of HDDs, however, grows more slowly: Early HDDs in 1989 delivered about 0.5 megabytes (MB)/s, while

current HDDs manage around 150 MB/s [Wik14b]. This corresponds to a 300-fold increase of throughput over the last almost 25 years. Even newer technologies such as SSDs only offer throughputs of around 600 MB/s, resulting in a total speedup of 1,200. For comparison, over the same period of time, the computational power increased by a factor of more than 1,000,000 due to increasing investments. While this problem can not be easily solved without major breakthroughs in hardware technology, it is necessary to use the storage hardware as efficiently as possible to alleviate its effects.

To make the problem worse, the growth rate of HDD capacity has recently also started to slow down. While the same is true for CPU clock rate, this particular problem is being compensated for by growing numbers of increasingly cheap cores. However, the price of storage has been staying more or less constant for the last several years, requiring additional investment to keep up with the advancing processing power.



(a) HDD capacities from 1980–2014

(b) HDD speeds from 1989–2009

Figure 1.5.: Development of HDD capacities and speeds [Wik14a, Wik14b]

Figures 1.5a and 1.5b show the increase in HDD capacity and speed from roughly the same period of time. As can be seen, HDD capacity is growing much faster than their speed, which leads to various problems even outside of HPC. For example, simply rebuilding a replaced HDD in a redundant array of independent disks (RAID) took around 30 minutes in 2004[9], while the same operation takes more than seven hours today[10].

Although it is theoretically possible to compensate for this fact in the short term by simply buying more storage hardware, the ever increasing gap between the ex-

[9] Assuming a 160 gigabytes (GB) HDD with a throughput of 75 MB/s.
[10] Assuming a 4 TB HDD with a throughput of 150 MB/s.

ponentially growing processing power on the one hand and the stagnating storage capacity and throughput on the other hand, requires new approaches to use the storage infrastructure as efficiently as possible.

Usage	Component	Size	Speed
Desktop	CPU	8 cores	100 GFLOPS
	Main Memory	16 GiB	50 GiB/s
	Storage	4 TB	600 MB/s
TOP500, rank 1	CPU	3,120,000 cores	33.9 PFLOPS
	Main Memory	1.3 PiB	No data
	Storage	12.4 PB	No data
TOP500, rank 2	CPU	560,640 cores	17.6 PFLOPS
	Main Memory	694 TiB	No data
	Storage	40 PB	1.4 TB/s

Table 1.1.: Comparison of important components in different types of computers

To properly assess a system's performance, it is not only necessary to take the absolute sizes and speeds into account, but also to consider their relationship with each other. Interesting quantities include the amount of main memory per core, the proportion of main memory and storage, as well as the main memory and storage throughput per core. Table 1.1 contains typical sizes and speeds of the most important computer components for different usage scenarios.[11]

Based on the given numbers, typical desktop computers are equipped with 2 gibibytes (GiB) of main memory per core and offer 250 GB of storage per 1 GiB of main memory; additionally, the storage can be accessed with 600 MB/s. Assuming a fair distribution among all cores, this provides per-core throughputs of 6.25 GiB/s to the main memory and 75 MB/s to the storage.

The numbers change drastically when looking at supercomputers: The TOP500 system ranked number 1 is equipped with a very large number of cores, a reasonable amount of main memory and a relatively small storage system. It offers 0.44 GiB of main memory per core and 9.5 GB of storage per 1 GiB of main memory; this equals 22 % and 3.8 % of the desktop computer's main memory per core and storage per main memory, respectively. While the moderate amount of main memory per core is usually sufficient for the very compute-intensive HPC applications, the small amount of storage dramatically limits the amount of data that can be stored. As mentioned earlier, HPC applications often write checkpoints to storage. Using this configuration, it is only possible to dump the main memory contents eight times before the storage

[11] The components for the desktop usage represent a reasonably powerful desktop computer in 2014; the data for the TOP500 systems ranked number 1 and 2 can be found in the 2014-06 list [The14c].

is filled up.[12] This is a stark contrast to the 232 possible main memory dumps on a typical desktop computer.

The TOP500 system ranked number 2 is equipped with much less cores, half the amount of main memory, but a much larger storage system. It offers 1.27 GiB of main memory per core and 57.6 GB of storage per 1 GiB of main memory. This corresponds to 63.5 % and 23 % of the desktop computer's main memory per core and storage per main memory, respectively. While this amount of storage offers more freedom for storing large checkpoints and application output, the actual storage throughput warrants a closer look. Assuming that all cores access the storage in a fair manner, the system offers a throughput of 2.5 MB/s per core; this merely corresponds to 3.3 % of the desktop computer's per-core storage throughput.

Due to the reasons outlined above, it is necessary to use the storage systems of supercomputers as efficiently as possible, both in terms of capacity as well as performance. Many of the parallel distributed file systems in use today do not allow applications to exhaust their potential. This is due to the fact that these file systems are optimized for specific use cases and do not offer enough opportunities for application developers to optimize them according to their applications' needs.

1.5. Contribution

The goal of this thesis is to explore the usefulness of additional semantical information in the I/O interface. The JULEA[13] framework introduces a newly designed I/O interface featuring dynamically adaptable semantics that is suited specifically for HPC applications. It allows applications developers to specify the semantics of I/O operations at runtime and supports batch operations to increase performance. The overall goal is to allow the application developer to specify the desired behavior and leave the actual realization to the I/O system. This should allow applications to make the most of the available storage hardware and thus increase the overall efficiency of I/O in HPC systems. This approach is expected to improve the current situation because existing solutions simply do not allow such fine-grained control over so many different aspects of file system operations.

1.6. Structure

This thesis is structured as follows: Chapter 2 contains an overview of the current state of the art; all important concepts related to file systems, object stores, I/O interfaces and I/O semantics are introduced and explained. The design of the JULEA I/O

[12] The count of eight stems from the fact that main memory and storage are counted using GiB and GB, respectively. While GiB use a base of two, GB use a base of ten.

[13] JULEA is not an acronym.

interface is elaborated in Chapter 3, focusing on the differences to traditional I/O interfaces and file systems. Chapter 4 covers related work and compares JULEA's design with existing approaches. Select parts of the implementation are presented in-depth in Chapter 5. Chapter 6 contains an analysis of the behavior of different file systems using both synthetic benchmarks as well as real-world applications. A conclusion and future work are given in Chapter 7.

Summary

This chapter has introduced the I/O problems found in today's HPC systems that are caused by the ever increasing gap between computational speed on the one hand and storage capacity and speed on the other hand. It has also given an overview of parallel distributed file systems as well as I/O interfaces and semantics, and their impact on overall performance. Because current supercomputers show a trend of neglecting their storage systems in favor of computation, new approaches are necessary to make the most of the available storage hardware.

Chapter 2.

State of the Art and Technical Background

In this chapter, an in-depth overview of existing technologies related to I/O interfaces and semantics will be provided. Today's I/O interfaces and semantics will be analyzed regarding their suitability and adaptability for high performance computing applications. Additionally, different approaches for managing the file system namespace will be compared.

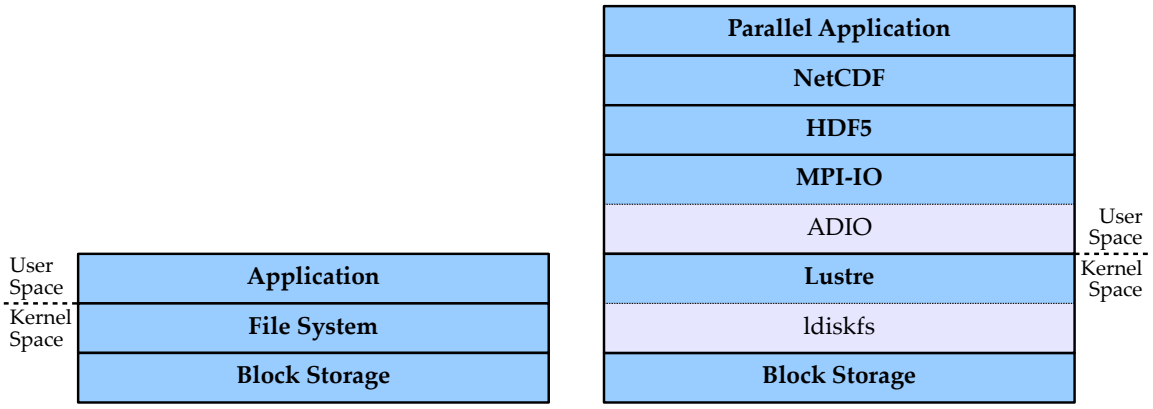
2.1. Input/Output Stack

Input/output (I/O) stacks usually feature a strongly layered architecture. Traditionally, this has been a major advantage because the clear separation between the different layers provides benefits regarding portability and interchangeability of individual layers. Figure 2.1a on the following page shows the relatively simple I/O stack of a traditional application that directly uses the underlying file system's I/O interface. Since all layers interact using standardized interfaces, it is easily possible to exchange the underlying storage device or even file system without adapting the application.

However, the I/O stack used by current high performance computing (HPC) applications is much more complex due to the more advanced requirements. This has led to the situation visualized in Figure 2.1b on the next page, which illustrates all the different layers involved in common scenarios. The different layers will be briefly explained below; more detailed information can be found in the following sections.

Parallel Application This can be an arbitrary parallel program executed on a supercomputer. For instance, this could be an earth system model using the de-facto standard MPI[1] for communication. It uses NetCDF[2] to read and write its data, which is a popular choice because it allows easy exchange of data.

[1] Message Passing Interface
[2] Network Common Data Form



(a) Traditional I/O stack (b) HPC I/O stack

Figure 2.1.: I/O stacks used in traditional and HPC applications

NetCDF This high-level I/O library provides a convenient interface to interact with self-describing data. This allows storing additional meta information together with the data, which is widely used in the natural sciences. However, it does not define its own file format and thus does not directly store the data itself. Instead, it delegates this task to yet another I/O library called HDF[3].

HDF This high-level I/O library also provides an interface to interact with self-describing data, similar to NetCDF. Additionally, it defines file formats to actually store and access this data. It can use different storage backends such as POSIX[4] for serial I/O and MPI-IO for parallel I/O.

MPI-IO The so-called I/O middleware provides a portable interface for data access that abstracts from the underlying file system. It usually includes optimizations for different file systems to enable efficient I/O. Common implementations of MPI include MPICH[5] and OpenMPI; both use ROMIO to provide MPI-IO support. ROMIO implements file-system-specific optimizations in the so-called ADIO[6] layer [HK04]. This middleware then accesses a parallel distributed file system.

Lustre The parallel distributed file system provides common file system functionalities such as metadata management, path lookup and striping to the upper layers. Parallel distributed file systems often provide a POSIX interface for portable access;

[3] Hierarchical Data Format
[4] Portable Operating System Interface
[5] MPICH acts as the base for many other popular MPI implementations such as MVAPICH, IBM MPI, Intel MPI and Cray MPT.
[6] Abstract-Device Interface for I/O

proprietary interfaces are also possible, however. Lustre uses another underlying POSIX file system to store both its data and metadata. In this case, ldiskfs provides a full-featured file system based on ext4.

Block Storage The low-level storage hardware provides storage capacity for the file system. It is often supplied in a block-oriented fashion in the form of locally attached hard disk drives (HDDs) or solid state drives (SSDs). However, it also possible to use more complex architectures like a storage area network (SAN).

2.1.1. Problems

All I/O initiated by the application has to pass all the different layers and is potentially copied and transformed multiple times along its way. While the clear separation between layers provides advantages in terms of portability and interchangeability, it can prove counterproductive for performance. The different interfaces are often inappropriate to transport information necessary for high performance across layer boundaries. Additionally, no reliable information might be available about the other involved layers, making it hard – if not impossible – to adapt to the software environment at hand. This can have a negative impact on overall I/O performance.

The fact that the layers do not have any information about the the other layers often implies that each layer has to perform its own optimizations to be able to use the I/O system's full potential. For instance, almost all layers implement their own caching to reduce the number of I/O accesses that have to be performed. However, these optimizations can also conflict and actually reduce the achieved performance.

While the upper layers usually provide more comfort and abstraction, the performance yield might be lower. They often provide interfaces that are more suited for handling data types actually found in parallel applications and it would therefore be favorable to be able to use them. Because performance is often more important than convenience, the difficult-to-use byte-oriented lower layers are often used directly to harness the I/O system's full potential. This has led to a multitude of different libraries written around the low-level I/O interfaces.

One problem that drives developers to the low-level interfaces is the fact that the high-level I/O libraries often do not offer fine-grained control over the actual I/O and instead hide this complexity from the user. For instance, while it is easily possible to align the I/O operations for optimal performance with POSIX and HDF, NetCDF does not offer such functionality [Bar14].

Additional semantical information could help reducing the need for fine-grained control by providing the I/O system with enough information to make meaningful decisions by itself. Presently, support for modifying the I/O semantics is very limited at best. While some layers provide basic support, it is currently not possible to pass semantical information down through the I/O stack. To ease the development of codes

in need of high performance I/O, it would be very beneficial to provide easy-to-use interfaces that are still able to provide adequate performance.

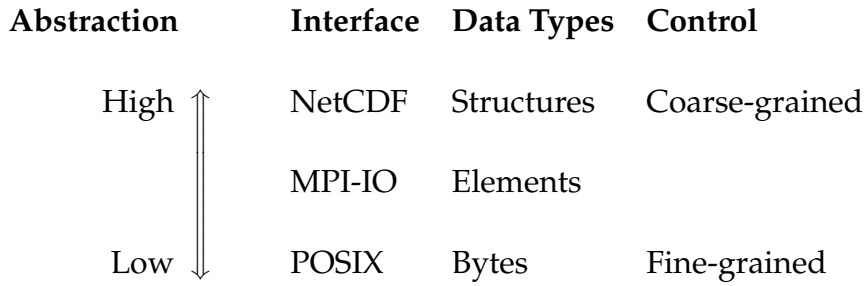


Figure 2.2.: Levels of abstraction found in the HPC I/O stack

Figure 2.2 gives an overview of the different levels of abstraction found in the I/O stack. Higher levels of abstraction such as those provided by NetCDF allow convenient access to data structures but only coarse-grained control over the I/O interface's behavior. The I/O middleware is provided by MPI-IO, which provides an element-based interface and some degree of control over internal functionalities. The lowest level of abstraction is provided by the POSIX interface; access is only possible in the form of a byte stream but I/O can be manually tuned for optimal performance due to the fine-grained control.

The remaining chapter will give an in-depth overview of the complete I/O stack with the exception of block storage, which will only be mentioned briefly. To convey how the different components build and improve upon each other, the overview will be given bottom to top.

2.2. File Systems

File systems store, manage and make data available for later reuse in an organized fashion. Without them, developers and users would have to interface directly with the storage hardware – for example, HDDs and SSDs.

Traditionally, file systems expose two basic data structures called *files* and *directories*. While files contain actual data, directories are used for organizational purposes. Directories can contain files as well as other directories, usually providing a hierarchical file system namespace. Virtually all file systems distinguish these two concepts, even if they are not necessarily called the same. Files and directories are usually accessed by their name – called a *path*; more information about path traversal is available in Section 2.7 on page 46. Examples for traditional file system include Windows's NTFS[7], OS X's HFS+[8] or Linux's ext4 [MCB+07].

[7] New Technology File System
[8] Hierarchical File System Plus

While files and directories represent the bare minimum in terms of file system functionality, many file systems provide additional features such as so-called *named pipes* and *forks*: Named pipes provide a first in, first out (FIFO) data structure for inter-process communication (IPC) within the file system. Forks allow storing several different data streams within a single file or directory [Wik14c, Mea03]. For instance, image files could have an additional data stream storing thumbnails of the image to make their on-the-fly generation redundant.

The explanations and observations in this thesis will focus on Linux because it is the de-facto standard operation system used in the HPC field. It is therefore important to note that almost all Linux file systems are POSIX-compliant. Consequently, the POSIX standard plays an important role regarding file systems; more information about it and its implications will be provided in Section 2.5.1 on page 35.

Linux kernel file systems are also generally implemented using the so-called *virtual file system (VFS)* layer. This layer provides a standardized interface – in this case, as defined by POSIX – that file systems can implement. The VFS then provides uniform access for user space applications: Independent of the actual file system implementation, applications can use the POSIX interface to perform I/O. The VFS layer then takes care to forward the I/O operations to the appropriate file system implementation. On the one hand, this provides benefits regarding portability because applications do not have to be aware of the underlying file system. On the other hand, it only allows file systems to provide a POSIX interface, making it more complicated to experiment with alternative interface approaches.

2.2.1. File System Metadata

Files stored within a file system consist of data as well as metadata. While the file's data represents the actual content of the file (for example, an image or a movie), metadata translates to "data about data" and refers to structural information in the context of file systems. This information is required for data management and traditionally stored in so-called *inodes* – or index nodes. File system metadata should not be confused with metadata in the context of self-describing data formats; the latter refers to additional information about the data and will be explained later.

File data can vary extremely in size, ranging from configuration files occupying only some bytes to videos and simulation results that can easily use several gigabytes (GB) or even terabytes (TB). Metadata usually only occupies several bytes – for example, ext4's default inode size is 256 bytes. Larger sizes of a few kilobytes (KB) are also possible, but relatively uncommon. Inodes usually have a fixed size and a fixed format with fields for permissions, ownership, different timestamps, flags and much more.

Figure 2.3 on the next page shows an excerpt of an inode as found in the ext4 file system. The inode contains fields of fixed size for the different types of metadata; these can be roughly separated into three main areas:

Size	Content
2 bytes	Permissions
2 bytes	User identifier (ID)
4 bytes	Size
4 bytes	Access time
4 bytes	Inode change time
4 bytes	Data modification time
4 bytes	Deletion time
2 bytes	Group ID
⋮	⋮
60 bytes	Block map, extent tree or inline data
⋮	⋮
4 bytes	Version number
100 bytes	Free space

Figure 2.3.: Structure of a 256 bytes inode (`struct ext4_inode`) [Won14]

1. The first fields are used to store access permissions, user and group ownership, the file's size, and different timestamps.

2. The block of 60 bytes in the middle of the inode can contain different kinds of data, depending on the type of object an inode describes. ext4 supports a new extent-based allocation scheme that stores the extent map inside this block. However, if the extent-based allocation scheme is disabled[9], the block is used to store block mappings to direct, indirect, double indirect and triple indirect blocks. If the file's size is below 60 bytes, all file data is inlined into the block; this can be beneficial for overall file system performance by reducing additional read operations for the actual file data: In local file systems, additional read operations usually require costly seek operations on the HDDs. In parallel distributed file systems, the necessity to communicate with additional servers via the network implies even more overhead. For example, the `file` tool has to read the first few bytes of a file to determine its type. Operations such as running `file` on large numbers of files can be sped up significantly by inlining data, because no additional data blocks or extents have to be read. Obviously, this also applies to all other tools which have to read file headers.

3. The 100 bytes of free space at the end of the inode can be used to store extended attributes such as access control lists (ACLs). Should this space not be sufficient to retain all extended attributes, additional entries can be stored in a data block.

[9] This is always the case for ext2 and ext3. ext4 can also be used without extents, but this is uncommon.

Because of their fixed format and size, it is usually not possible to change or even extend the schema of inodes after the file system's creation and without breaking compatibility. Examples of this are ext4's repurposing of the block map field for data inlining and the reservation of free space at the end of the inodes for future extensions.

2.3. Object Stores

In contrast to full-featured file systems, *object stores* provide only a very low level of abstraction on top of storage devices. Strictly speaking, object stores simply offer object-oriented access to data, which can be achieved on any abstraction layer. For example, cloud services usually offer object stores for data storage. However, only low-level object stores will be considered here. Instead of exposing raw block storage to the end user, object stores offer access to so-called *objects* while handling tasks such as block or extent allocation and management of free space internally [ADD+08]. These objects can optionally be organized in so-called *object sets*, which can be used to group related objects.

While object stores are often discussed as a replacement for the low-level block storage, they can also be used as light-weight and low-overhead replacements for file systems when only basic storage management capabilities are required. File systems such as btrfs[10] and ZFS[11] actually use object stores internally. This allows separating the functionality for storage management and advanced file system features, leading to cleaner and more maintainable code.

Objects are usually accessed using unique identifiers such as simple integers or hashes. This can allow very fast access to the objects because no path lookup overhead is incurred; more information about this is provided in Section 2.7 on page 46.

While there are a few object stores available, different technical issues prevent their use by external third-party applications. First, the interfaces of the internal object stores used by file systems are usually not exported for consumption by third parties. Second, even exported interfaces may not be easily usable. For example, ZFS's data management unit (DMU) is largely undocumented and not meant to be used from user space. Last, independently usable object stores are often discontinued in favor of off-the-shelf file systems due to development and maintenance overhead. For instance, Ceph developed and used its own EBOFS[12] [WBM+06], but dropped it back in 2009; it has since been replaced by btrfs.

[10] b-tree file system
[11] Zettabyte File System
[12] Extent and B-tree-based Object File System

2.4. Parallel Distributed File Systems

As presented briefly in Chapter 1, parallel distributed file systems consist of clients and servers that are communicating via a network. The servers can be separated into data and metadata servers. Data servers are usually only used to store the actual file data, while metadata servers hold all information regarding the file system's organizational structure, such as file metadata and directories.

The workload is distributed across all of them to increase capacity as well as performance. While there are always multiple data servers, specific file systems still use centralized metadata management, that is, they support only a single metadata server. However, due to the difficulty of scaling such an approach to higher numbers of clients and files, it is being increasingly replaced by distributed metadata, which uses multiple metadata servers.

Due to this separation, data and metadata servers usually see different access patterns. While file data is meant to be accessed in large chunks, file system metadata is small by design. Consequently, metadata servers are often subject to large numbers of small, random accesses. As HDDs are very bad at handling these kinds of workloads, the problem is often mitigated using HDDs with a high number of revolutions per minute (RPM) or – as is becoming more and more attractive – SSDs, which offer orders of magnitude higher input/output operations per second (IOPS) than HDDs. There have also been approaches using alternative storage technologies such as persistent random access memory (RAM) but these have not been widely adopted [WKRP06].

Technology	Device	IOPS
HDD	7,200 RPM	75–100
	10,000 RPM	125–150
	15,000 RPM	175–210
SSD	Intel X25-M G2	8,600
	OCZ Vertex 4	85,000–90,000

Table 2.1.: IOPS for exemplary HDDs and selected SSDs [Wik14d]

Table 2.1 contains a list of selected storage devices and their respective IOPS for illustrative purposes. As can be seen, a single high-end SSD can easily provide as many IOPS as 450 high-end HDDs.[13] On the one hand, SSDs have a much higher price per GB than HDDs – around 0.8 € per GB for SSDs in comparison to around 0.04 € per GB for HDDs in 2014. On the other hand, metadata only occupies a fraction of the space needed for the actual data – estimations for metadata size are usually in the range of 5 % of the data contained within a file system. Overall, SSDs are an appealing alternative for workloads limited by the number of IOPS such as those commonly

[13] This example uses a modern SSD with 90,000 IOPS and a modern HDD with 200 IOPS.

seen on metadata servers. However, more research is needed to be able to fully take advantage of the improved capabilities of SSDs; currently, the metadata performance of parallel distributed file systems can only be sped up by a factor of 2–4 by using SSDs [AEHH+11].

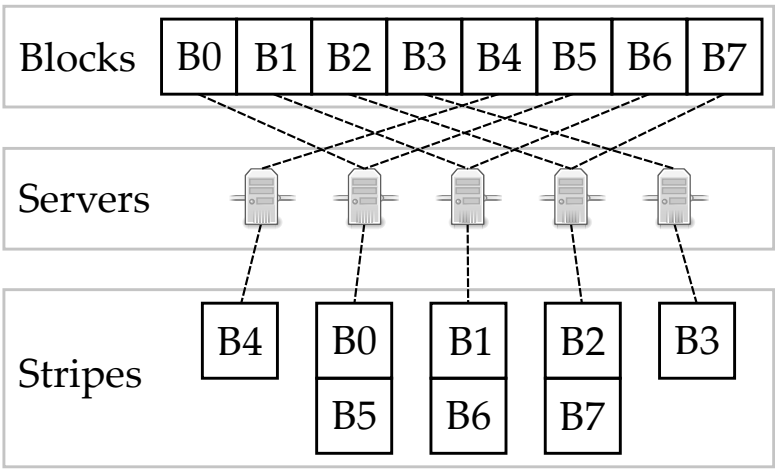


Figure 2.4.: Round-robin data distribution

The file system logic is often implemented in the clients that can usually decide autonomously which servers to contact; the servers do not have to communicate with each other and act as simple data stores. The partitioning of data and metadata is handled using so-called *distributions*.

As illustrated in Figure 2.4, round-robin schemes are a common approach for data distribution; they are used to distribute data in equal chunks across multiple servers in circular order. In this example, eight blocks of data (B0–B7) are distributed across five data servers. The servers hold so-called *stripes* of the data; in this example, the data blocks exactly correspond to the stripes. As can be seen, the round-robin distribution does not necessarily have to start at the first data server; in this case, it starts at the second. The starting server is usually chosen randomly to ensure an even distribution.[14] The data blocks are distributed normally until the last data server is reached; afterwards, the distribution restarts with the first data server.

These round-robin schemes can lead to unbalanced distributions of data, as demonstrated in this example. However, due to the random starting server and large file sizes this problem can usually be ignored.

Metadata is often distributed by means of hashing; using cryptographic hash functions such as the SHA family has the advantage of providing uniform distributions. In these cases, the file name or full path is hashed to decide which metadata server to target. While metadata is usually distributed across multiple metadata servers, it is

[14] Otherwise, multiple clients accessing the beginning of different files would all contact the same data server, which could have negative impacts on performance.

often not striped in any way, that is, the complete metadata for a single file is managed by exactly one metadata server.

Clients and servers are usually hosted on different sets of physical machines to provide more predictable performance characteristics as computational load on the clients should not influence the servers' I/O performance and vice versa [LM10].

Based on this generic explanation, a detailed description of Lustre's architecture and an overview of OrangeFS will be provided in the following sections.

2.4.1. Lustre

Lustre is an open source parallel distributed file system that is widely used on current supercomputers. In contrast to other proprietary solutions such as GPFS[15], it is possible to adapt, extend and improve Lustre due to it being licensed under the GNU General Public License (GPL) (version 2). Lustre powers around half of the TOP100 supercomputers and almost one third of all TOP500 supercomputers [Fel13].

Lustre was started in 1999 by Peter Braam, who founded his own company called Cluster File Systems in 2001 to continue development. Cluster File Systems was acquired by Sun Microsystems in 2007, which started bundling Lustre with its HPC hardware. Oracle Corporation bought Sun Microsystems in 2010 and soon announced that it would cease Lustre development. Today, Lustre is developed and supported by Intel (formerly Whamcloud), Xyratex, OpenSFS[16], EOFS[17] and others.

As is common in parallel distributed file systems, Lustre distinguishes between clients and servers. It is possible to run clients and servers on the same nodes for testing purposes but it is common to distribute them to separate nodes in production environments. While all clients are identical, the servers can have different roles:

- *Object storage servers (OSSs)* manage the file system's data. They provide an object-based interface that clients can use to access byte ranges within the objects. Each OSS is connected to possibly multiple *object storage targets (OSTs)* that store the actual file data.
- *Meta data servers (MDSs)* manage the file system's metadata, such as directories, file names and permissions. MDSs are not involved in the actual I/O but only contacted once when a file is created or opened. The clients are then able to independently contact the appropriate OSSs. Each MDS is connected to possibly multiple *meta data targets (MDTs)* that store the actual metadata.

[15] General Parallel File System
[16] Open Scalable File Systems
[17] European Open File Systems

Lustre does not grant the clients direct access to the storage, but instead delegates this responsibility to the servers. Clients send their requests to the appropriate servers that process them and then in turn send a response to the client. Both MDTs and OSTs use an underlying file system to store their data. Traditionally, an improved version of ext4 called ldiskfs has been used. However, support for ZFS's DMU has been added in Lustre's version 2.4.

Lustre has been implemented as a Linux kernel file system. Its client supports standard Linux kernels, though support for newer Linux versions had to be added manually and was thus not available immediately. However, the client has been merged into the Linux kernel as of version 3.12 and is now available without any further actions.[18] Lustre's server part is only compatible with special enterprise kernels, such as those found in SUSE Linux Enterprise Server and Red Hat Enterprise Linux.[19]

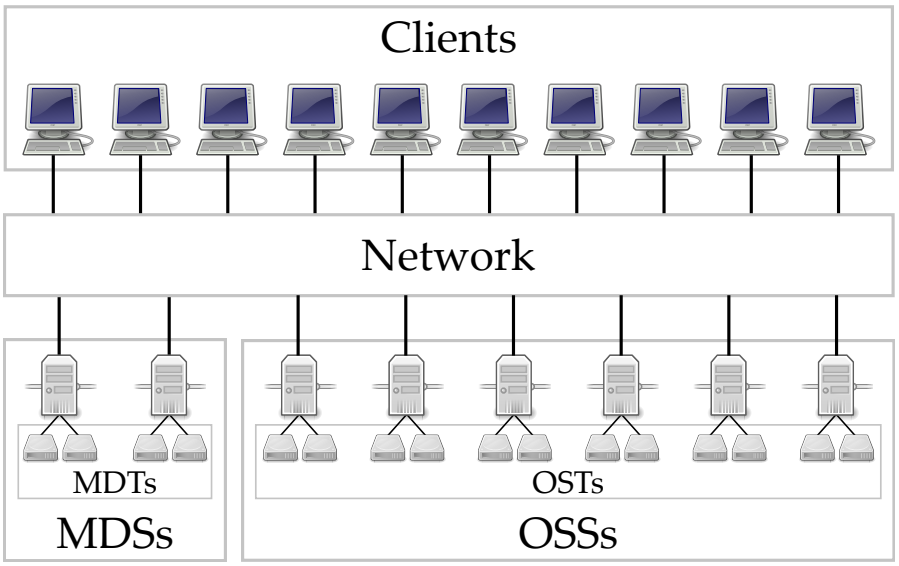


Figure 2.5.: Lustre architecture

Figure 2.5 demonstrates the general architecture of Lustre using a simple example with ten clients and eight servers. There are two MDSs handling all metadata accesses and six OSSs processing all data accesses. Each MDS and OSS has two storage devices attached that represent the MDTs and OSTs, respectively. The Lustre file system can be accessed using a mount point on the clients; they handle all accesses and communicate with the appropriate MDSs and OSSs via the network.

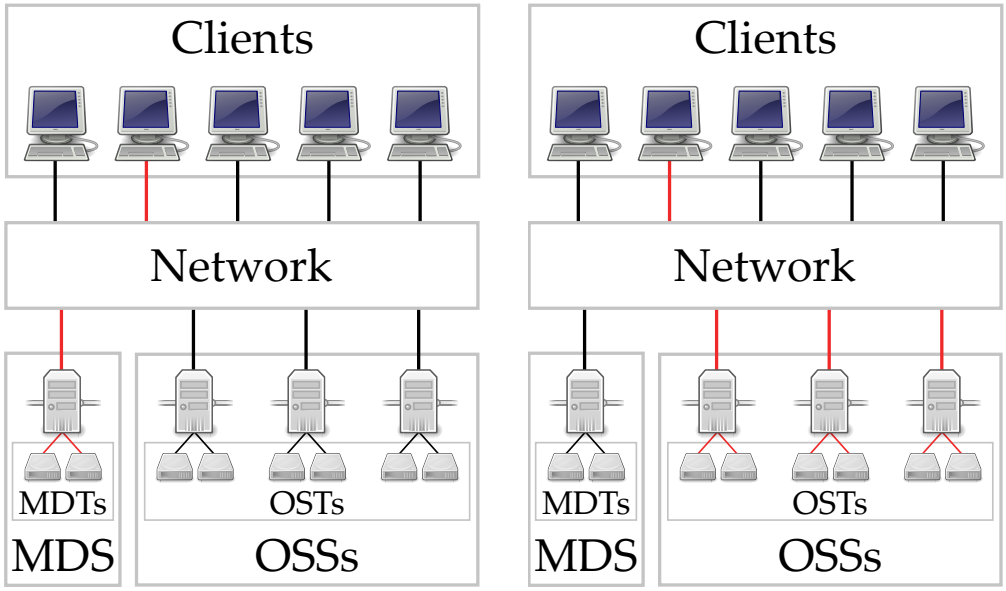
Traditionally, Lustre has only supported a single MDT and consequently one active MDS, optionally complemented by a second failover MDS. With the growing number of clients in today's supercomputers, this posed a serious threat to future scalability

[18] The Lustre client module included in the Linux kernel may occasionally lag behind upstream Lustre development, however.

[19] Free and binary-compatible alternatives such as CentOS and Scientific Linux are also available.

because all metadata was centralized on one server. Lustre 2.4 introduced the so-called *distributed namespace (DNE)* which allows the system administrators to statically distribute the file system namespace across multiple MDTs. While the current implementation only supports a static partitioning, a feature planned for future Lustre releases will offer true distributed metadata with which the directories themselves can be striped across multiple MDTs.

For example, it is common for HPC systems to provide two directories `/home` and `/scratch` that are used to house the users' home directories and scratch data, respectively. DNE can be used to provide appropriate resources depending on the intended usage. `/home` usually contains many – possibly small – files, resulting in high metadata access rates. Less metadata performance might be sufficient for `/scratch` because it usually only contains a small number of large files. This could be solved by using a high-end SSD with a large number of IOPS as `/home`'s MDT and a cheaper but slower solution for `/scratch`'s MDT. The necessary commands to set up Lustre's DNE for this use case can be found in Appendix B.3.1 on page 190.



(a) Step 1: metadata lookup (b) Step 2: data access

Figure 2.6.: One client accessing a file inside a Luste file system

Figure 2.6 shows how a typical access to a file inside a Lustre file system takes place. The example uses five clients, one MDS with two MDTs and three OSSs with two OSTs each. The highlighted connections indicate active communication between the involved machines. The second client wants to access an arbitrary file. In order to do so, it has to contact the MDS to perform a path lookup (see Figure 2.6a). The MDS will return the file's metadata, including its distribution information. Using the distribution information, the client can autonomously determine which OSSs

contain parts of the file's data. Afterwards, the client can contact the appropriate OSSs concurrently (see Figure 2.6b on the preceding page). The fact that the MDS is only involved during the initial opening of the file ensures that possible metadata performance bottlenecks do not influence data performance.

2.4.2. OrangeFS

OrangeFS is another open source parallel distributed file system mainly developed by Clemson University, Argonne National Laboratory and Omnibond. It is the successor of the PVFS[20] project, having started as a development branch of PVFS in 2007. In 2010, OrangeFS became the main branch and replaced PVFS.

OrangeFS supports multiple data and metadata servers in its current version 2.8. Even though it supports multiple metadata servers, a single directory can not be distributed across multiple servers. However, support for distributed directories is scheduled for version 2.9. OrangeFS has excellent MPI-IO support because the widely used MPI-IO implementation ROMIO provides a native backend.

Even though OrangeFS is not as commonly used as Lustre, it still provides interesting features. It can be run completely from user space without the need for any kernel modules: The servers run as normal user space processes, an MPI-IO interface is provided through ROMIO and a POSIX interface is available via a FUSE[21] file system. An additional, optional kernel module is available that allows mounting OrangeFS as any other Linux file system [VRC+04]. Moreover, OrangeFS's code base is much smaller than Lustre's, making it easier to develop modifications and extensions for it.

2.5. Input/Output Interfaces

I/O interfaces define a set of possible operations that can be performed. Additionally, each I/O interface is usually accompanied by its own set of I/O semantics that is tailored specifically to this interface. A description of the most common I/O interfaces and their corresponding semantics follows.

2.5.1. POSIX

The POSIX I/O interface has been originally designed for use in local file systems. Its first formal specification dates back to 1988, when it was included in POSIX.1; specifications for asynchronous and synchronous I/O were added in POSIX.1b from 1993 [IG13]. This interface is very widely used, even in parallel distributed file systems, and thus provides excellent portability [VLR+08].

[20] Parallel Virtual File System
[21] Filesystem in Userspace

```
1 int open (const char *pathname, int flags, mode_t mode);
2 int close (int fd);
3 ssize_t pread (int fd, void* buf, size_t count, off_t offset);
4 ssize_t pwrite (int fd, const void* buf, size_t count, off_t
     ↪ offset);
5 int fstat (int fd, struct stat *buf);
6 int unlink (const char *pathname);
```

Listing 2.1: POSIX I/O interface

To get an overview about POSIX's functionality and usability, Listing 2.1 shows selected functions provided by the POSIX interface:

- The `open` function can be used to create and open existing files (line 1). It accepts a path `pathname` and returns a so-called *file descriptor* that is represented by an integer. Its arguments `flags` and `mode` can be used to specify different file flags and permissions for newly created files, respectively. There are actually three versions of the `open` function: the presented one with three arguments, another version with two arguments and the `creat` function. The version with two arguments omits the `mode` argument and can only be used for already existing files. The `creat` function is equivalent to the `open` function called with `flags` set to `O_CREAT | O_WRONLY | O_TRUNC`, which specifies that the file should be created if it does not exist, opened in write-only mode and truncated to size `0` if it already exists.

- The `close` function simply closes the open file descriptor `fd` (line 2).

- The `pread` function reads data from a file specified by an open file descriptor `fd` (line 3). It reads `count` bytes, starting at byte position `offset`, and stores the read data in the buffer `buf`.

- The `pwrite` function performs the opposite operation (line 4). It writes `count` bytes to the file specified by `fd`, starting at byte position `offset`; the to-be-written data is taken from the buffer `buf`. The traditional `read` and `write` operations work the same as their `p`-prefixed counterparts but do not accept the `offset` argument. Instead, they operate using a file pointer that is advanced automatically after each operation.

- The `fstat` function returns metadata about the open file descriptor `fd` and stores it in `buf` (line 5). There are two more variants of the `fstat` function: The `stat` function works the same but accepts a path instead of an open file descriptor. The `lstat` function is identical to the `stat` function, except that it does not dereference symbolic links. `buf` is a structure containing multiple fields

for the metadata, such as `st_size` for the file size and `st_mtime` for the last modification timestamp.

- The `unlink` function deletes a file given by the path `pathname` (line 6). To be precise, the `unlink` function only removes a link to a file. As files are reference counted objects, they are only deleted if their reference count – that is, their number of links – drops to zero. It is necessary to use the `rmdir` or `remove` functions to delete directories. While the former only removes directories, the latter is able to remove both files and directories.

A longer code example using the functions mentioned above can be found in Appendix C.1 on pages 192–194.

2.5.2. MPI-IO

The MPI-IO interface offers support for parallel I/O and was introduced in the MPI standard's version 2.0 in 1997 [Mes12]. All I/O operations are handled in an analogous fashion to MPI's normal message passing operations. It provides an I/O middleware that abstracts from the actual underlying file system. The popular ROMIO implementation uses the ADIO layer that includes support and optimizations for POSIX, NFS, OrangeFS and many other file systems [HK04]. In contrast to the byte-oriented POSIX interface, the MPI-IO interface is element-oriented and uses the existing MPI infrastructure of MPI datatypes to access data within files. However, the actual I/O functions look very similar to their POSIX counterparts [Seh10].

```
1 int MPI_File_open (MPI_Comm comm, char* filename, int amode,
     ↪ MPI_Info info, MPI_File* fh);
2 int MPI_File_close (MPI_File* fh);
3 int MPI_File_read_at (MPI_File fh, MPI_Offset offset, void* buf,
     ↪ int count, MPI_Datatype datatype, MPI_Status* status);
4 int MPI_File_write_at (MPI_File fh, MPI_Offset offset, void* buf,
     ↪ int count, MPI_Datatype datatype, MPI_Status* status);
5 int MPI_File_get_size (MPI_File fh, MPI_Offset* size);
6 int MPI_File_delete (char* filename, MPI_Info info);
```

Listing 2.2: MPI-IO I/O interface

Listing 2.2 shows selected functions provided by the MPI-IO interface. All functions return an integer that signals whether the operation was successful or not; this can be checked by comparing it with the status code `MPI_SUCCESS` and several error codes.

- Files are created and opened using the `MPI_File_open` function (line 1). It accepts a path `filename` and returns a so-called *file handle* `fh` that can be used

to access the file with the following functions. Different file access modes can be specified using the `amode` argument. MPI-IO provides access modes such as `MPI_MODE_CREATE` and `MPI_MODE_WRONLY` that are identical to their POSIX counterparts. The `info` argument can be used to provide so-called *hints* to the MPI-IO implementation; for instance, this allows modifying internal buffer sizes and timeouts. All processes in the MPI communicator `comm` perform the operation collectively and must provide the same values for the `amode` and `filename` arguments. Opening individual files can be accomplished using the `MPI_COMM_SELF` communicator.

- `MPI_File_close` closes the file handle `fh` again (line 2).
- The `MPI_File_read_at` function reads data from the opened file handle `fh` (line 3). It reads `count` elements of type `datatype`, starting at position `offset`; the data is stored in `buf`. In contrast to POSIX's byte-oriented interface, MPI-IO provides an element-oriented interface. To work with single bytes, it is possible to specify `MPI_BYTE` as the `datatype` argument. The operation's status is stored in `status`. Checking the number of read elements requires an additional step after the operation has finished: The `MPI_Get_count` function can be used to extract this information from the `MPI_Status` object.
- The `MPI_File_write_at` function performs the opposite operation, writing data into the opened file handle `fh` (line 4). It writes `count` elements of type `datatype`, starting at position `offset`; the data is taken from `buf`. Again, the number of written elements can be checked using the `MPI_Get_count` function.
- The `MPI_File_get_size` function retrieves the size of the file opened using the file handle `fh` and stores it in `size` (line 5). In contrast to the POSIX interface, it is not possible to read additional metadata using the MPI-IO interface. For example, it is not possible to get the last modification time.
- The `MPI_File_delete` function deletes the file specified by the path `filename` (line 6). MPI-IO hints can be given using the `info` argument.

A longer example using the above mentioned MPI-IO functions can be found in Appendix C.2 on pages 195–197.

2.5.3. SIONlib

SIONlib provides an I/O interface that allows scalable access to task-local files [FWP09]. It internally maps all accesses to a single or small number of physical files and aligns accesses to the file system's block size. Additionally, it strives to minimize the amount of changes necessary to use the interface by providing wrappers for the common

`fread` and `fwrite` functions. Opening and closing files requires the use of special SIONlib-specific functions, though.

```
1  int fd;
2  FILE* fp;
3
4  fd = sion_paropen_mpi(..., &fp, ...);
5
6  for (...)
7  {
8      fwrite(..., fp);
9  }
10
11 sion_parclose_mpi(fd);
```

Listing 2.3: SIONlib parallel I/O example

An example for parallel access using SIONlib is shown in Listing 2.3. A file is opened in parallel mode with the collective function `sion_paropen_mpi` that returns both a file descriptor `fd` as well as a so-called *file stream* `fp` (lines 1–4). A non-collective open is available via the `sion_open_rank` function; serial access is provided by `sion_open` and `sion_close`. After opening the file, some data is written using the standard `fwrite` function (lines 6–9). Finally, the file is closed using the `sion_parclose_mpi` function (line 11).

SIONlib is a good example for a library that exists primarily to overcome shortcomings in current file systems. On the one hand, current file systems often have problems when dealing with large numbers of files. On the other hand, shared file performance often degrades dramatically when the I/O operations are not aligned to the file system's block size due to locking overhead, which should not be necessary if only non-overlapping accesses occur. SIONlib tries to mitigate these problems by intelligently managing the number of underlying physical files and transparently aligning the data; this is achieved by allocating contiguous chunks of data for each process and remapping accesses to its own internal file layout.

In summary, using more intelligent file systems could make many libraries working around file system limitations obsolete. The additional information that is required to enable this kind of intelligence can be provided by semantical approaches such as the one proposed in this thesis.

2.5.4. HDF

HDF comprises a set of file formats and libraries that allow storing and accessing self-describing collections of data, and is widely used in scientific applications [The14a].

While HDF5 is the current version, HDF4 is still actively supported. However, due to its complicated API and several limitations – such as the use of signed 32-bit integers for addressing, limiting HDF4 files to a maximum size of 2 GiB – HDF4 is not a feasible choice for newly developed codes anymore.

HDF5 supports two major types of data structures: *datasets* and *groups*. These two objects are used analogously to files and directories, that is, datasets are used to store data, while groups are used to structure the namespace. Groups can contain several datasets as well as other groups, leading to a hierarchical layout. Datasets can store multi-dimensional arrays of a given data type. Objects within an HDF5 file are then accessed using POSIX-like paths such as `/path/to/dataset`. As can be seen, the dataset name can be used to describe the meaning of the dataset's values, such as temperature or wind speed in a climate simulation.

Additionally, arbitrary *metadata* – that is, information about the data – can be attached to datasets and groups in the form of user-defined, named *attributes*. This can be used to store information such as the allowed minimum and maximum values within a dataset together with the actual data. HDF files are self-describing and thus allow accessing them without any prior knowledge about their structure or content.

HDF5 supports multiple storage backends, including POSIX and MPI-IO. Using the MPI-IO backend, it is possible to perform parallel I/O from multiple clients into a single HDF5 file.

2.5.5. NetCDF

NetCDF, like HDF, consists of a set of libraries and self-describing file formats, and is used in scientific applications, especially from the fields of climatology, meteorology and oceanography [RD90]. Three major NetCDF formats are in existence today: the classic format, the 64-bit offset format and the NetCDF-4 format. While the former two are independent data formats, the NetCDF-4 format uses HDF5 underneath.

There are several options for performing parallel I/O using NetCDF: Most importantly, NetCDF-4 supports parallel I/O for NetCDF-4 – that is, HDF5 – files. Parallel I/O for classic and 64-bit offset files is possible using either recent versions of the official NetCDF library or the third-party Parallel-NetCDF library that features an incompatible interface.

2.5.6. ADIOS

ADIOS[22] provides a high-level I/O interface that abstracts from the usual byte- or element-oriented access as found in POSIX or MPI-IO [LKS+08, KLL+10]. It has been designed to provide high performance especially for scientific applications [PLB+09].

[22] Adaptable IO System

ADIOS outsources the actual I/O configuration into an external XML[23] file that can be used to describe which data structures should be accessed and to automatically generate C or Fortran code. Due to this, the application developer does not need to directly interact with the underlying I/O middleware or file system. ADIOS can handle elemental data types as well as multi-dimensional arrays.

```
1 <adios-config host-language="C">
2   <adios-group name="checkpoint">
3     <var name="rows" type="integer"/>
4     <var name="columns" type="integer"/>
5     <var name="matrix" type="double" dimensions="rows,columns"/>
6   </adios-group>
7   <method group="checkpoint" method="MPI"/>
8   ...
9 </adios-config>
```

Listing 2.4: ADIOS XML configuration

Listing 2.4 shows an example ADIOS XML configuration file that is used to define the data to be read or written. It specifies that C code should be generated by ADIOS's source code generator (line 1).[24] Additionally, it defines a so-called *group* with the name `checkpoint`; the group includes the variables `rows`, `columns` and `matrix` (lines 2–6). While `rows` and `columns` are integers, `matrix` is a two-dimensional array consisting of double-precision floating-point numbers. Finally, it specifies that the MPI-IO backend should be used to access this group (line 7).

```
1 adios_open(&adios_fd, "checkpoint", "checkpoint.bp", "w",
     ↪ MPI_COMM_WORLD);
2 #include "gwrite_checkpoint.ch"
3 adios_close(adios_fd);
```

Listing 2.5: ADIOS code

Using the XML configuration file, ADIOS can automatically generate C code to read and write the defined variables and stores it in the `gread_checkpoint.ch` and `gwrite_checkpoint.ch` files, respectively.[25] Listing 2.5 demonstrates how the generated code can be used to write data. First, an ADIOS file has to be opened for writing (line 1): The `adios_open` function takes parameters for a file descriptor (`adios_fd`), a group name (`checkpoint`), a file name (`checkpoint.bp`), an access mode (`w` for

[23] Extensible Markup Language

[24] The actual source code can be generated by invoking ADIOS's `gpp.py` utility and passing it the XML file's path as an argument.

[25] If Fortran code is requested, ADIOS generates analogous `.fh` files.

writing) and an MPI communicator (`MPI_COMM_WORLD`). Writing the variables defined in the configuration file is performed by simply including the generated source code (line 2). Finally, the file has to be closed again (line 3).

As can be seen, all logic required to perform the actual I/O operations necessary to store the checkpoint is contained within the automatically generated source code. Therefore, application developers do not have to care about specifying the correct amount of bytes to write or other specifics when using ADIOS.

2.6. Input/Output Semantics

In the following, the most common I/O semantics are presented and potential shortcomings are highlighted. The multitude of existing I/O semantics continues to create problems because different layers within the I/O stack might feature different semantics. Proper HPC-compatible I/O semantics on the upper layers are useless if the semantics on the lower layers ruin any potential performance benefits [HNH09].

2.6.1. POSIX

The POSIX standard features very strict consistency requirements. For example, `write` operations have to be visible to other clients immediately after the system call returns. While this might be relatively easy to support in local file systems, it can pose a serious bottleneck in parallel distributed file systems, because it effectively prohibits client-side caching from being used and might require additional locking.

> *"The adjustment of the file offset and the write operation are performed as an atomic step."*
>
> Source: [The14b]

Even though POSIX requires some atomicity as shown in the quote above, it is not specified whether the actual writing of the data has to be atomic. Technically, POSIX only specifies that write operations to pipes and FIFO special files have to be atomic if the size of the write request is not larger than `PIPE_BUF`.[26] Even though the standard intends I/O to be atomic, it does not require it to be so [IG13].

POSIX's I/O semantics can only be changed in a very limited fashion. For instance, the `strictatime`, `relatime` and `noatime` options change the file system's behavior regarding the last access timestamp. The traditional `strictatime` option causes the last access timestamp to be updated on every file access, `relatime` causes it to be only updated when it is older than the last modification timestamp and `noatime` disables

[26] POSIX requires `PIPE_BUF` to be at least 512 bytes; on Linux, it is 4,096 bytes.

updates of the last access timestamp completely. Obviously, especially `strictatime` can have a serious impact on performance, because every read operation results in an additional write operation. While this introduces significant overhead even in local file systems, parallel distributed file systems require network transfers for each write operation, increasing the overhead even further.

Additional `async` and `sync` options are also available that allow switching between asynchronous and synchronous I/O, respectively.

These options can be specified on a per-mount basis to be fixed at mount time or using the `O_NOATIME`, `O_ASYNC` and `O_SYNC` flags of the `open` and `fcntl` functions. However, the latter may not be easily possible when using high-level I/O libraries that do not expose the underlying file descriptors. Consequently, these aspects can often not be modified by users under normal circumstances.

The original POSIX interface did not offer ways to specify semantical information about the accesses or the data. A feature added in POSIX.1-2001 is called `posix_fadvise` and allows announcing the pattern that will be used to access the data.

```
1 int posix_fadvise (int fd, off_t offset, off_t length, int advice);
```

Listing 2.6: `posix_fadvise`

Listing 2.6 shows the `posix_fadvise` function that can be used to advise the file system about future accesses. It provides `advice` to the file descriptor `fd` for the file range given by `offset` and `length`. However, this does not actually change the semantics of any following I/O operations. It is typically only used to increase the readahead window (`POSIX_FADV_SEQUENTIAL`), disable readahead (`POSIX_FADV_RANDOM`), or to populate (`POSIX_FADV_WILLNEED`) and free (`POSIX_FADV_DONTNEED`) the file system cache.

2.6.2. NFS

The NFS[27] protocol provides close-to-open cache consistency by default, which implies that changes performed by a client are only written back to the server when the client closes the modified file. However, NFS offers limited support for changing this behavior: By mounting NFS using the `cto` or `nocto` options, close-to-open cache coherence semantics can be switched on or off, respectively.

Additionally, the `async` and `sync` options can be used to modify the behavior of write operations: While `async` causes writes to only be propagated to the server when necessary[28], `sync` will cause I/O operations to only return when the data has been

[27] Network File System

[28] Write operations are delayed until either memory pressure forces them to be sent or the file in question is (un)locked, synchronized or closed [Unk12].

flushed to the server. Additional mount options are available to modify the caching behavior of attributes and directory entries.

As in the POSIX case, the `async` and `sync` behavior can be specified at mount time or using the `O_ASYNC` and `O_SYNC` flags of the `open` and `fcntl` functions. The `cto` and `nocto` options, however, can only be specified at mount time by the administrator.

2.6.3. MPI-IO

MPI-IO's consistency requirements are less strict than those defined by POSIX [SLG03, CFF+95]. By default, MPI-IO guarantees that non-overlapping or non-concurrent write operations will be handled correctly; changes are immediately visible only to the writing process itself. Other processes first have to synchronize their view of the file to see the changes.

```
1 MPI_File_sync(fh);
2 MPI_Barrier(MPI_COMM_WORLD);
3 MPI_File_sync(fh);
```

Listing 2.7: MPI-IO's sync-barrier-sync construct

Listing 2.7 shows the so-called *sync-barrier-sync* construct that is necessary to handle concurrent file modifications correctly. The first `MPI_File_sync` operation makes sure that the changes of all processes are transferred to storage (line 1). The `MPI_Barrier` provides an explicit synchronization point (line 2): Write operations performed before the barrier will be visible to read operations performed after the barrier. The second `MPI_File_sync` ensures that all file modifications flushed to storage during the first call are visible to all processes (line 3).

For use cases requiring stricter consistency semantics, MPI-IO offers the so-called *atomic mode* that causes all operations to be performed atomically; it can be enabled and disabled on demand using the `MPI_File_set_atomicity` function. This special mode allows concurrent and conflicting writes to be handled correctly and also causes changes to be visible to all process within the same communicator without explicit synchronization. From the implementer's point of view, this can be difficult to achieve because MPI-IO allows non-contiguous operations and parallel distributed file systems can stripe single write operations over multiple servers [RLG+05, LRT07].

MPI-IO implementations are free to offer so-called *hints* that are mainly used to control things like buffer sizes and participating processes. Because hints are optional, however, different implementations are free to ignore them [TRL+10].

Additionally, MPI-IO offers several different access modes that can be specified when a file is opened using `MPI_File_open`. The MPI standard specifies the following access modes:

> *"The following access modes are supported (specified in amode, a bit vector OR of the following integer constants):*
>
> - *MPI_MODE_RDONLY — read only,*
> - *MPI_MODE_RDWR — reading and writing,*
> - *MPI_MODE_WRONLY — write only,*
> - *MPI_MODE_CREATE — create the file if it does not exist,*
> - *MPI_MODE_EXCL — error if creating file that already exists,*
> - *MPI_MODE_DELETE_ON_CLOSE — delete file on close,*
> - *MPI_MODE_UNIQUE_OPEN — file will not be concurrently opened elsewhere,*
> - *MPI_MODE_SEQUENTIAL — file will only be accessed sequentially,*
> - *MPI_MODE_APPEND — set initial position of all file pointers to end of file."*
>
> Source: [Mes01]

The access modes MPI_MODE_RDONLY, MPI_MODE_RDWR, MPI_MODE_WRONLY, MPI_MODE_CREATE and MPI_MODE_EXCL have the same meaning as their POSIX counterparts. MPI_MODE_DELETE_ON_CLOSE and MPI_MODE_APPEND provide convenience functionality: The former causes an implicit MPI_File_delete to remove the file when closing it, while the latter causes an implicit MPI_File_seek to set the initial position of the file pointer to the end of the file.

The only two access modes which can be considered semantical information are MPI_MODE_UNIQUE_OPEN and MPI_MODE_SEQUENTIAL; these modes provide information about *how* the file is going to be accessed and allow this information to be exploited for more intelligent access. MPI_MODE_UNIQUE_OPEN specifies that the given file will only be accessed by the current set of processes, which can be used to eliminating locking overhead. MPI_MODE_SEQUENTIAL allows optimizations based on the assumption that the given file will only be accessed sequentially.

Even though MPI_MODE_SEQUENTIAL might look similar to POSIX's POSIX_FADV_SEQUENTIAL mode, there are actually several differences: While POSIX_FADV_SEQUENTIAL simply increases the readahead window, MPI_MODE_SEQUENTIAL actually influences future operations; for instance, it is not allowed to call MPI_File_seek on files opened with MPI_MODE_SEQUENTIAL because seeking can be used to perform random accesses. Additionally, it is not permitted to combine MPI_MODE_SEQUENTIAL with MPI_MODE_RDWR according to the standard.

Discussion

As can be seen, there are numerous I/O interfaces available. This diversity can be confusing for application developers and users, making it unclear which I/O interface

should be used for a given task. Additionally, different I/O libraries typically address different use cases: For instance, while it would be beneficial to use I/O interfaces such as NetCDF that offer access to self-describing data, SIONlib allows optimizing performance when accessing shared files. It is, however, not easily possible to combine the benefits of both approaches because SIONlib is orthogonal to NetCDF and its dependencies. To make matters worse, each I/O interface typically comes with its own set of semantics. This further complicates the use of the available I/O interfaces because each one might behave differently, even for the same use case.

2.7. Namespaces

The file system's namespace defines how data can be found and organized. File system namespaces are usually organized hierarchically, starting with a so-called *root directory* that includes further files and directories. However, other organizational approaches are also possible. One popular approach is to add so-called *tags* to files and provide powerful search capabilities such as full-text indexing [SM09, BVGS06]. This frees the user from remembering where files are stored and instead allows them to access them by content and association.

2.7.1. POSIX

POSIX-compliant file systems provide a standardized way to find and access files and directories within them. The namespace is organized in a hierarchical way, with directories serving as containers for files and other directories. The fully specified name of a file or directory is called a *path*, consisting of one or more *path components* that are separated using the *delimiter* `/`.

For example, given a file `bar` located inside a directory `foo`, the file's path would be `foo/bar`. This represents a *relative path*, because the `foo` directory could be located inside any other directory. An *absolute path* starts in the file system's *root directory*, which can be accessed using the path `/`. Consequently, if the `foo` directory was located inside the root directory, the file's full path would be `/foo/bar`.

As can be seen, paths can become very long because directories can be arbitrarily nested. This, in turn, can impact performance when a large number of files are accessed. To access a file, a *path lookup* has to be performed, which involves each of the path components. Consequently, this is a relatively expensive operation because several checks and lookup operations have to be performed for each of the path components.

The following list gives an overview of the involved operations. The currently active path component is marked in **bold** and underlined.

1. `/foo/bar`
   a) The root directory's inode is read.[29]
   b) Permission checks are performed.
   c) The root directory is read and searched for `foo`.
2. `/foo/bar`
   a) The directory's inode is read.
   b) Permission checks are performed.
   c) The directory is read and searched for `bar`.
3. `/foo/bar`
   a) The file's inode is read.
   b) Permission checks are performed.
   c) The file is accessed.

2.7.2. Cloud

Cloud storage services usually offer only flat namespaces. For example, both Amazon S3[30] as well as Google Cloud Storage provide a global namespace in which users can create so-called *buckets*. This namespace is shared between all users, that is, two users can not create buckets with the same name. Within these buckets, *objects* can be created. Each object is assigned a unique *key* that can be used to access it.

All accesses are performed using standard HTTP[31] requests. See Listing 2.8 for a list of exemplary uniform resource locators (URLs) used by the Amazon and Google storage services; these can be accessed using HTTP methods such as `GET`, `POST`, `PUT`, `HEAD` and `DELETE`.

```
1 http://s3.amazonaws.com/<bucket>/<key>
2 http://<bucket>.s3.amazonaws.com/<key>
3
4 http://storage.googleapis.com/<bucket>/<key>
5 http://<bucket>.storage.googleapis.com/<key>
```

Listing 2.8: Amazon S3 and Google Cloud Storage URLs

[29] As there is no parent directory to search, the root directory's inode must be known in advance. For example, in ext4's case the root directory's inode always has the ID 2.
[30] Amazon Simple Storage Service
[31] Hypertext Transfer Protocol

The namespaces provided by cloud storage services provide the opportunity to get rid of the path traversal overhead usually found in file systems' namespaces.

However, the actual interfaces are not suitable for use in file systems due to their heavy dependence on HTTP. On the one hand, the overhead of HTTP is non-negligible for small accesses because requests consist solely of strings that have to be parsed. On the other hand, the interfaces themselves do not provide the flexibility required for file systems. For instance, it is often impossible to only access specific byte ranges of objects or even modify them once they have been uploaded completely.

Summary

This chapter has given an in-depth description of the current HPC I/O stack and its components. While kernel file systems are generally forced to offer POSIX interfaces due to their use of the VFS layer, object stores only provide basic storage management functionality and can mitigate metadata overhead. Parallel distributed file systems such as Lustre and OrangeFS typically have support for multiple data and metadata servers to distribute the load; this architecture also allows them to handle the different access patterns more efficiently. Current I/O interfaces only have very limited support for providing semantical information and their semantics have often been designed for serial use cases, making them unsuited for HPC workloads. Whereas traditional file system namespaces require expensive path lookup operations, cloud storage services usually provide flat namespaces that can reduce the associated costs.

Chapter 3.

Interface and File System Design

Based on the information gathered in the previous chapter, this chapter will be dedicated to elaborating the design of the proposed I/O interface featuring adaptable semantics. All important aspects of the file system's design will be illustrated, including the general architecture, the namespace, the data and metadata design, and – most importantly – its interface and semantics. A special focus will lie on the design choices made to avoid the bottlenecks and problems present in other contemporary file systems and interfaces.

As shown in the previous chapters, the interfaces and semantics currently used for parallel distributed file systems are suboptimal because they are either not well-adapted for the requirements and demands found in high performance computing (HPC) today or do not allow fine-grained semantical information to be specified. To further explore the optimization potential of adaptable semantics, a new I/O interface as well as a file system prototype will be designed from scratch, suited specifically for the demands found in HPC. The resulting framework is called JULEA.

While the overall design decisions and important key aspects will be explained in this chapter, the technical architecture will be described in more detail in Chapter 5.

3.1. Architecture

JULEA's general architecture will closely follow that of established parallel distributed file systems such as Lustre and OrangeFS. Machines can have one or several of three different roles: client, data server and metadata server. While it is possible to have a machine perform all three roles simultaneously, it is recommended to separate the clients from the servers to provide stable performance.[1] JULEA will support multiple data and metadata servers and allow data and metadata to be distributed among them; it will be possible to influence the actual distribution of data using distributions.

A very brief general view of JULEA's different components and their interactions with each other are shown in Figure 3.1 on the following page. Applications will be

[1] Depending on the actual access patterns, it might also be sensible to host the data and metadata servers on different machines.

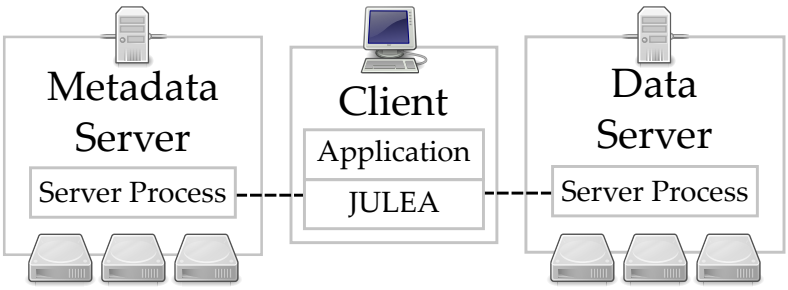


Figure 3.1.: JULEA's file system components

able to use JULEA's input/output (I/O) interface that talks directly to the data and metadata servers; it will abstract all the internal details and provide a convenient interface for developers. The metadata and data servers will run on dedicated machines with attached storage hardware.

The remaining part of this chapter is devoted to a more detailed discussion of several architectural design decisions.

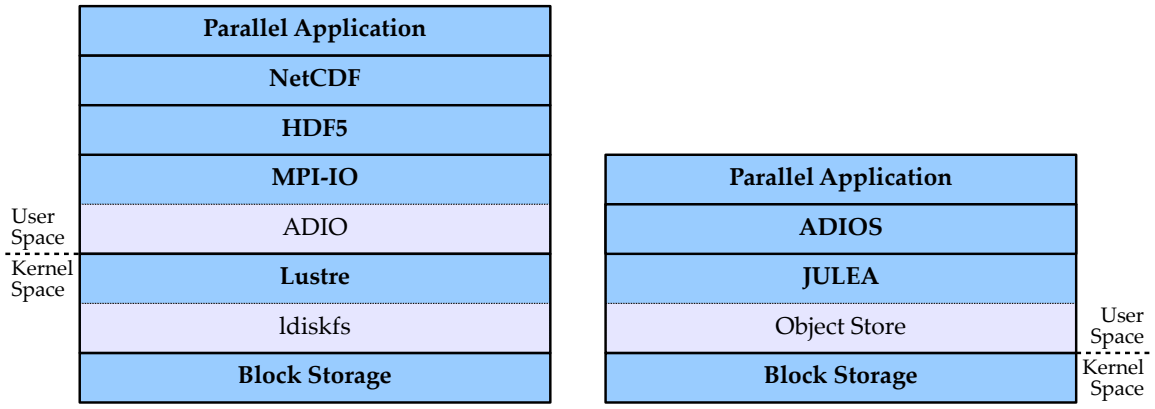
3.1.1. Layers

Figures 3.2a and 3.2b on the next page show a comparison of the current HPC I/O stack and the proposed JULEA I/O stack. In addition to the logical layers, the separation between kernel and user space is shown. All kernel space layers are either implemented directly inside the kernel or as kernel modules; the user space layers are either normal applications or libraries. As can be seen, JULEA's architecture will feature less layers, which will make it easier to analyze the actual I/O behavior of applications. It will also allow concentrating all optimizations into a single layer, reducing the implementation and runtime overhead.

Specifically, the current I/O stack is built in such a way that multiple different I/O interfaces build upon each other. This results in several transformations of the data as it is being transported through the different layers. The parallel application's data types are stored in NetCDF[2] that in turn stores its data in HDF[3]'s datasets and groups. This data is then transformed into a byte stream for MPI-IO. It then stores the data in the actual parallel distributed file system that splits up the data and stripes it across its servers, potentially storing it in yet another underlying local file system. For a more in-depth description, refer to Section 2.1 on pages 23–26.

All of these layers have additional advanced concepts for optimizing the parallel I/O. For example, NetCDF, HDF and MPI-IO all have the concept of individual and collective I/O. However, all of them perform I/O in a slightly different way with

[2] Network Common Data Form
[3] Hierarchical Data Format



(a) HPC I/O stack (b) JULEA I/O stack

Figure 3.2.: Current HPC I/O stack and proposed JULEA I/O stack

different semantics. Several MPI-IO implementations contain optimizations targeted specifically at collective I/O, such as Two-Phase I/O [TGL99, DT98] or Layout-Aware Collective I/O [CST+11]. In addition to generic optimizations for collective I/O, additional file-system-specific optimizations are also possible; for instance, ROMIO's ADIO[4] layer contains a Lustre-specific module that can exploit Lustre's capabilities to offer improved performance [YVCJ07]. Nevertheless, NetCDF and HDF perform their own optimizations on top of this. Sometimes these optimizations can be even contradictory, resulting in performance degradations instead of improvements.

An important design goal of JULEA is to remove the duplication of functionality found in the traditional HPC I/O stack. Because many distributed file systems use an underlying local POSIX[5] file system to store the actual data and metadata, a lot of common file system functionality is duplicated. For example, path lookup and permission checking are already performed by the parallel distributed file system and should not be executed again by the underlying local file system. This can be achieved by completely eliminating the underlying POSIX file systems and using suitable object stores. As presented in Section 2.3 on page 29, object stores usually assign each object a unique identifier (ID), removing the need for path lookups on the lower layers.

Because it is often unreasonable to port applications to new and experimental I/O interfaces due to their size and complexity, it makes sense to leverage a layer providing compatibility for existing applications. ADIOS[6] is an established I/O interface and specifically allows implementing different backends. To minimize the overhead, ADIOS could be used as a relatively thin layer on top of JULEA to provide convenient access for application developers.

[4] Abstract-Device Interface for I/O
[5] Portable Operating System Interface
[6] Adaptable IO System

3.1.2. Protocol

One of the first and most important decisions is the communication schema between the file system's clients and servers. In parallel distributed file systems, two basic approaches are possible for client-server communication:

1. The clients do not know which server can answer their current request and thus contact a random server. If the contacted server is not responsible, two reactions are possible:
   a) The server silently forwards the request to the appropriate server and returns the answer back to the client; this process is completely transparent for the client.
   b) The server tells the client which servers is responsible; the client communicates with the correct server from this point on.
2. The clients know which server can answer their current request and directly contact the appropriate one.

These approaches necessitate completely different communication schemes and each has its own advantages as well as disadvantages:

1.
   - **Advantages:** Clients do not need to have any prior knowledge about the distribution of data and metadata because they can simply contact any server. It is relatively easy to implement load balancing because another server can simply take over an overloaded server's responsibilities by redirecting the client.
   - **Disadvantages:** Almost all initial requests suffer from additional network latency because clients will only rarely contact the correct server right away; in case the servers transparently forward messages, this also applies to almost all subsequent requests.

2.
   - **Advantages:** The servers do not need to communicate with each other and, in fact, do not even need to know about each other. The communication protocol can be kept simple because there is no inter-server communication that has to be considered.
   - **Disadvantages:** All communication logic has to be implemented by the clients. Additionally, clients need prior knowledge about the distribution of data and metadata: For data, this usually involves contacting the appropriate metadata server first; for metadata, this implies that clients have to be able to decide autonomously which metadata server to contact.

JULEA will use the second approach: Clients will be able to autonomously decide which servers to contact whenever possible and then talk directly to the appropriate

data and metadata servers. As the servers will not have to communicate with each other, their design can be kept simple: The data servers will act as basic object stores for the clients' I/O requests. This is similar to Lustre's design – as shown in Section 2.4.1 on pages 32–35 – and has several advantages:

1. The servers' behavior is easier to comprehend because only direct interactions between the clients and servers have to be considered; the program flow only includes requests from the clients and the corresponding replies issued by the servers. Additionally, only replies from the contacted server have to be considered because no message forwarding takes place.

2. Problems in the servers are easier to debug because only one kind of communication has to be considered; this makes it much easier to understand the flow of data and narrows the number of possible causes for errors.

3. The performance behavior is easier to comprehend because the servers simply act on the clients' behalf and do not perform more intelligent actions behind their back.

3.1.3. Performance Analysis Functionality

Performance analysis of parallel distributed file systems is a complex topic and much research has been done in this regard. It is necessary to have insight into the internals of a file system to be able to understand its performance characteristics [Kun06, Tie09]. In addition to the complicated behavior regarding data performance, metadata performance continues to play an important role; increasing numbers of clients want to access increasing numbers of file system objects, quickly exposing bottlenecks in the metadata design [Bia08].

Another important point are the connections between client operations and the resulting behavior on the servers: Without the possibility to correlate the clients' activities and the resultant events on the servers, finding and solving performance problems becomes much harder [Kre06].

Consequently, JULEA will have built-in support for tracing client and server activities; it should also be possible to easily correlate them for the reasons mentioned above. This will facilitate easier performance analysis because tracing support does not have to be added retrospectively. Visualization of the resulting traces is also important because the sheer amount of trace data is impossible to analyze manually [MSM+11]. Therefore, it should also be possible to leverage existing measurement tools such as Jumpshot [LKK+07] or Vampir [GWT14] to visualize JULEA's traces.

3.2. File System Namespace

Traditional file systems allow deeply nested directory structures. To avoid the overhead caused by this, only a restricted and relatively flat hierarchical namespace will be supported. While this approach might be unsuited for a general purpose file system, JULEA is explicitly focused on specific use cases that are commonly found in HPC. Therefore, JULEA is meant to be used in conjunction with traditional file systems like NFS[7] to provide other parts of the infrastructure such as the users' home directories.

The file system namespace will be divided into *stores*, *collections*, and *items*. Each store can contain multiple collections that can, in turn, contain multiple items. This structure will be closer to that of popular cloud storage solutions than that of POSIX file systems. The goal of these changes is to minimize the overhead during normal file system operation. In traditional POSIX file systems, each component of the potentially deeply nested path has to be checked for each access. This requires reading its associated metadata, checking permissions and so forth. As this process usually happens sequentially, it can seriously hamper performance. Additionally, in distributed file systems these operations can be very costly because metadata operations are usually small in size; consequently, many small network messages are generated.

If absolutely necessary, it would be possible to extend the namespace by allowing collections to include other collections, thus creating a nested namespace. However, for all intents and purposes of the initial prototype, the flat namespace will be enough. This is not expected to have any negative influences on usability because this kind of namespace is already being commonly used in cloud-based storage solutions and document database systems.

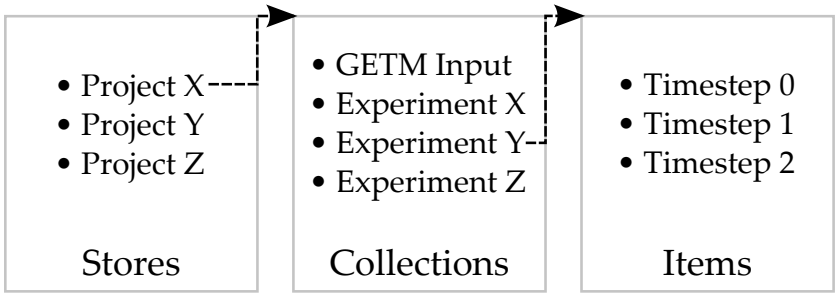


Figure 3.3.: JULEA namespace example

Figure 3.3 shows an exemplary JULEA namespace using an application from the field of earth system science. The first level of the namespace hierarchy are the stores that are used to group similar data. In this example, there are stores for different research projects with the *Project X* store being expanded to show its collections. This project is concerned with GETM[8], an open source ocean model, and includes input

[7] Network File System
[8] General Estuarine Transport Model

data for said model in the *GETM Input* collection. During the imaginary research project, several experiments have been conducted and the output of each experiment has been stored in a separate collection. In this example, the *Experiment Y* collection is expanded to show its items. Models usually perform their calculations in so-called *timesteps* that define the model's temporal resolution. For example, if a timestep comprises 30 minutes, it is possible to output the state of the model in intervals of 30 minutes for later analysis; this state is stored in the *Timestep i* items.

Obviously, this example presents only one possible use of JULEA's namespace. As with any other file system namespace, administrators, developers and users should think about a reasonable structure in advance.

To have access to a standardized way of accessing JULEA's file system objects, it makes sense to define paths in JULEA's file system namespace. Using the information above, paths are defined as follows:

- Each path consists of either one, two or three path components.
- The first path component refers to the store, the second path component refers to the collection and the third path component refers to the item.
- The path components are separated using the `/` delimiter.

Because JULEA will not have a concept of a *current working directory*, all JULEA paths are defined to be absolute.[9] Using the exemplary namespace organization from Figure 3.3 on the facing page again, the paths to refer to the store, collection and item would look like the following:

- `Project X`
- `Project X/Experiment Y`
- `Project X/Experiment Y/Timestep 1`

3.3. Interface

JULEA's interface will be designed from scratch to offer simplicity of use while still meeting the requirements of high performance and dynamically adaptable semantics. The functionality offered by the interface can be subdivided into five groups:

1. **Batches:** Multiple operations can be batched explicitly to improve performance.

[9] In traditional POSIX file systems, each process possesses a current working directory that is used when resolving relative paths. For example, assuming a current working directory of `/home/foo`, the relative path `bar` would be resolved to `/home/foo/bar`. The current working directory can be retrieved using the `getcwd` function or the `pwd` command line utility. For more information about absolute and relative paths, see Section 2.7.1 on pages 46–47.

2. **Distributions:** It will be possible to influence the distribution of data directly.

3. **Namespace:** The file system namespace will be accessible using a convenient abstraction called uniform resource identifiers (URIs).

4. **Semantics:** JULEA's semantics will be dynamically adaptable according to the applications' I/O requirements.

5. **Stores, collections and items:** It will be possible to create, remove, open and iterate over all of JULEA's file system objects.

All of the above functionality will be available publicly and directly to developers. While the underlying design principles and ideas for parts of the I/O interface will be illustrated in this chapter, JULEA's actual application programming interface (API) for use by applications will be presented in detail in Chapter 5.

The two most important features will be the ability to specify semantical information and to batch operations. Both approaches will give the file system additional information that can be used to optimize accesses.

It will be possible for developers and users to specify additional information equivalent to the coarse-grained statement "this is a checkpoint" or the more fine-grained "this operation requires strict consistency semantics". This will allow the file system to tune operations for specific applications by itself. Additionally, developers will be able to emulate well-established semantics as well as mixing different semantics within one application.

Developers will perform all accesses to the file systems via so-called *batches*. Each batch can consist of multiple operations. For example, multiple items can be created or different offsets within an item can be accessed in one batch. It will also be possible to combine different kinds of operations within one batch. For instance, one batch might create a collection and several items within it, and write data to each of the items.

Because the file system will have knowledge about all operations within one batch, more elaborate optimizations can be performed. This will also allow reordering the operations to improve network utilization whenever possible. For example, multiple metadata operations can be sent to the metadata servers with a single network message. Since batches will be executed explicitly, they provide a defined point at which all operations will be performed in contrast to traditional approaches.

Traditional POSIX file systems can also try to aggregate multiple operations to improve network utilization. However, this can only be done by caching these operations in the client's main memory for a given amount of time and then performing these optimizations. Because the POSIX interface does not provide enough information to make reliable decisions for these kinds of optimizations, it is necessary to employ heuristics. However, these heuristics are usually not correct all the time, resulting

in suboptimal behavior for borderline cases. Additionally, it is not possible to do this in all cases because it would violate the POSIX semantics. Therefore, users can never be sure when exactly operations are performed in such a system without calling synchronization functions explicitly, which can be very expensive.[10]

```
1 batch = new Batch(POSIX_SEMANTICS);
2
3 store = julea.create("test store", batch);
4 collection = store.create("test collection", batch);
5 item = collection.create("test item", batch);
6 item.write(..., batch);
7
8 batch.execute();
```

Listing 3.1: Executing multiple operations in one batch

The pseudo code found in Listing 3.1 shows an example of how the interface generally works. First, a new batch using the POSIX semantics is created (line 1). Afterwards, the store, collection and item are created (lines 3–5); the store is created in the root of the file system, the collection is created in the new store and the item is created in the new collection. Additionally, some data is written to the item (line 6). All of these operations are not executed right away but merely added to the batch that is passed to each method as the last argument. Finally, the batch is executed, which in turn executes all four operations with the previously specified semantics (line 8).

```
1  in_batch  = new Batch(DEFAULT_SEMANTICS);
2  out_batch = new Batch(POSIX_SEMANTICS);
3
4  input = collection.get("input item");
5
6  input.read(..., in_batch);
7  in_batch.execute();
8
9  /* Calculation */
10
11 checkpoint = collection.create("checkpoint item", out_batch);
12 checkpoint.write(..., out_batch);
13 out_batch.execute();
```

Listing 3.2: Using multiple batches with different semantics

[10] POSIX's synchronization functions `fsync` and `fdatasync` only allow synchronizing whole files even if this is not necessary.

An example for changing the semantics on a per-batch basis is given in Listing 3.2 on the preceding page. Two batches are created using different semantics (lines 1–2). The existing input item is opened (line 4) and then read (lines 6–7). After some calculations, a new checkpoint item is created (line 11) that is then written to (lines 12–13).

Supporting different semantics on a per-batch basis will allow using the optimal semantics for any given task. In the example given above, the semantics could additionally be tuned to instruct the file system that the `input` item will be accessed in a read-only fashion. Additionally, accesses to the `checkpoint` item could be optimized for non-overlapping write accesses from multiple clients.

JULEA will require all operations to be performed in batches, even if the batch only contains a single operation. This is a conscious design decision to make sure that the file system will always have as much information as possible to make informed optimization decisions. Even though this might appear as an inconvenience from the application developers' point of view, it will be easy for them to specify this information and will only introduce negligible overhead. However, employing heuristics and guessing appropriate optimizations after the fact is much harder and can result in suboptimal behavior in many cases. For instance, traditional I/O interfaces are unable to know whether a user is going to perform multiple operations in quick succession because each operation is executed individually.

Additionally, each batch will require the semantics to be set explicitly. Combined with the fact that all operations have to be performed in batches, this is supposed to force application developers to think about the possible performance implications of the chosen semantics.

3.3.1. Asynchronous Batches

To allow application developers to easily overlap calculations and I/O, it will be possible to execute batches asynchronously. This support will be offered natively by the I/O interface without forcing developers to resort to using background threads or similar techniques.

```
1 batch      = new Batch(DEFAULT_SEMANTICS);
2 checkpoint = collection.create("checkpoint 42", batch);
3
4 checkpoint.write(..., buffer, ..., batch);
5 batch.execute_async();
6
7 /* Calculation */
8
9 batch.wait();
```

Listing 3.3: Executing batches asynchronously

Listing 3.3 on the preceding page shows how the execution of asynchronous batches works. In this example, the writing of a checkpoint should be overlapped with some calculations to achieve optimal performance. First, a batch and an item for writing the checkpoint are created (lines 1–2). Afterwards, the write operation is added to the batch (line 4) and the batch is executed asynchronously (line 5). It is important to note that the data stored in `buffer` is not allowed to be changed until the batch execution has been completed. This is similar to MPI[11]'s non-blocking (or immediate) operations. The `execute_async` method returns immediately and allows the application to continue; calculations are then performed while the batch is executed in the background (line 7).[12] Last, the asynchronous batch is finalized by waiting for its completion (line 9).

To lower the barrier of entry and encourage application developers to use both concepts whenever appropriate, there are only two differences between the synchronous and asynchronous execution of batches; all other aspects remain exactly the same:

1. How the execution is initiated, that is, whether the `execute` or `execute_async` method is used. This also determines whether it is necessary to call the `wait` method or not.

2. Whether it is possible to reuse the data buffer immediately. Modifying the buffer during the execution of an asynchronous batch leads to undefined behavior.

3.3.2. Information Export

The file system should also export all the information that is necessary to reach optimal performance; this information can then be used by other layers of the I/O stack.

One important aspect is the information about alignment of data to the file system's stripe size. When dealing with larger numbers of clients, aligning the accesses to the file system's stripe boundaries becomes especially important [Bar14].

```
1 batch      = new Batch(DEFAULT_SEMANTICS);
2 checkpoint = collection.create("checkpoint 42", batch);
3
4 checkpoint.write(header, header_size, 0, batch);
5
6 data_size = checkpoint.get_optimal_access_size(header_size);
7 checkpoint.write(data, data_size, header_size, batch);
8
```

[11] Message Passing Interface

[12] In contrast to MPI, JULEA guarantees that the batch is executed asynchronously; the MPI standard does not mandate that implementations actually have to perform operations asynchronously but only that the operations are non-blocking and return immediately.

```
9 batch.execute();
```

Listing 3.4: Determining the optimal access size

Listing 3.4 on the preceding page shows how to extract the optimal access size from the file system. Analogous to the previous examples, a checkpoint is created (lines 1–2). However, the checkpoint contains a header this time; consequently, the actual data starts at a specific offset. First, the header of size `header_size` is written to the item at offset `0` (line 4). To be able to write the remaining data in a stripe-aligned fashion, `get_optimal_access_size` is used (line 6); it takes an offset within the item as its only argument and returns the number of bytes remaining for the responsible stripe. This information is then used to fill the current stripe with data of length `data_size` starting at offset `header_size` (line 7). Finally, the batch is executed, which causes the write of a full stripe (line 9).

Because the data distribution could vary based on the current item or even server, `get_optimal_access_size` provides a convenient way for application developers to acquire this type of file system information without resorting to uncertain assumptions. The availability of this information is especially important for higher layers within the I/O stack or applications that want to manually make use of this information to achieve optimal performance.

3.4. Semantics

The JULEA interface will allow many aspects of the file system operations' semantics to be changed at runtime. Several key areas of the semantics have been identified as important to provide opportunities for optimizations: atomicity, concurrency, consistency, ordering, persistency and safety. Even though it will be possible to mix the settings for each of these semantics, not all combinations will produce reasonable results. In the following, detailed explanations and design choices for these key aspects are provided. Additionally, further possible extensions for redundancy, security and transformation semantics are introduced.

The semantics can be categorized into convenience-related and performance-related ones. On the one hand, the performance-related aspects are clearly focused on achieving the maximum possible performance and require in-depth knowledge about the application's I/O behavior. On the other hand, the convenience-related ones are supposed to ease application development by providing comfort features directly within the file system.

3.4.1. Atomicity

The atomicity semantics can be used to specify whether accesses should be executed atomically, that is, whether or not it is possible for clients to see intermediate states of operations. These are possible because large operations usually involve several servers. If atomicity is required, some kind of locking has to be performed to prevent other clients from accessing data that is currently being modified. To cater to as many I/O requirements as possible, several levels of atomicity will be provided:

- **None:** Accesses are not executed atomically. For example, a single write operation that is striped over multiple data servers can be executed as several independent accesses. If not all data servers have already finished the write operation, concurrent read operations accessing the same data are able to return partly written data.

  No locking is required at all.

- **Operation:** Single operations are executed atomically. For example, a single write operation that is striped over multiple data servers is guaranteed to be executed atomically. Read operations accessing the same data concurrently are not able to return partly written data, even if not all data servers have finished the write operation. Instead, these operations are blocked until the write operation is finished completely.

  Locking is only required for pre-determined ranges within objects.

- **Batch:** Complete batches are executed atomically. Other batches accessing the same data are blocked until the batch finishes.

  Locking is required for potentially multiple complete objects.

The atomicity semantics is clearly performance-related. It can be used to avoid unnecessary locking overhead by completely avoiding locking whenever possible. Atomic accesses operating on the same data have to be serialized, which implies a performance penalty. If atomicity is not required, all operations can be executed in parallel.

Being able to specify the atomicity requirements has obvious advantages in contrast to static approaches such as those dictated by POSIX because lockless access to shared files can improve performance dramatically. For instance, many POSIX-compliant file systems perform atomic write operations even if all clients accessing a shared file never read or write to overlapping regions of the file. Since application developers know the access patterns of their applications, they can easily specify whether atomicity is required or not.

It is important to note that atomicity only applies to the visibility of modifications in this context. That is, operations could still be only partially performed in case of errors. Such guarantees are typically provided by atomicity, consistency, isolation

and durability (ACID) transactions as found in database systems and are not part of JULEA's initial design; for a discussion regarding full-featured transactions, see Section 7.1.2 on pages 163–164.

3.4.2. Concurrency

The concurrency semantics can be used to specify whether concurrent accesses will take place and, if so, how the access pattern will look like. This allows the file system to appropriately handle different patterns without the need for heuristics recognizing them. Depending on the level of concurrency, different algorithms might be appropriate for file system operations such as locking or metadata access; additionally, the level of concurrency has an impact on whether locking is necessary at all. To support as many I/O patterns as possible, several configurations will be available:

- **None:** No concurrent accesses will take place at all. The concerned objects will only be modified by one client at a time and the results of concurrent accesses are unspecified.

  Efficient centralized algorithms can be used.

- **Non-overlapping:** Concurrent accesses might take place. However, no two remote clients will modify the same area of an object. The results of modifying the same area concurrently are unspecified.

  Distributed algorithms have to be used but certain optimizations might be possible because the operations do not access the same data.

- **Overlapping:** Concurrent accesses might take place and might modify the same area of an object.

  Distributed algorithms have to be used and no assumptions about access patterns can be made.

Concurrency semantics are performance-related by allowing simpler and faster centralized algorithms to be used when no concurrent access is happening. Additionally, the information about the actual access patterns can be used to make more intelligent decisions. For instance, atomicity is only required for overlapping accesses. In case of strictly serial accesses, even more optimizations are possible because no other clients will be able to observe potential inconsistencies.

The use of centralized and distributed algorithms applies to different aspects of the parallel distributed file system. For example, it is advisable to use different metadata management approaches depending on the level of concurrency; this aspect will be elaborated in Section 3.5.2 on pages 71–73.

3.4.3. Consistency

The consistency semantics can be used to specify if and when clients will see modifications performed by other clients and applies to both metadata and data. This information can be used to enable client-side read caching whenever possible. To support different consistency requirements, several levels will be supported:

- **None:** Clients might never have a consistent view of the file system, that is, modifications performed by other clients might not be visible locally at all. This is similar to NFS's session semantics.

  Allows data and metadata to be cached indefinitely.

- **Eventual:** Clients will eventually have a consistent view of the file system, that is, modifications performed by other clients might not be immediately visible locally. For example, reading an object's modification time or size can return a cached value. The period during which the view is inconsistent is unspecified.

  Allows data and metadata to be cached for an unspecified amount of time.

- **Immediate:** Clients will always have a consistent view of the file system, that is, modifications performed by other clients are immediately visible locally.

  Data and metadata can not be cached; all data and metadata is retrieved directly from the appropriate servers.

The consistency semantics is performance-related and can allow caching data and metadata locally. It can be used to reduce the network traffic and thus increase performance. This is especially important for metadata because sending and receiving large amounts of small network messages can cause significant overhead.

3.4.4. Ordering

The ordering semantics can be used to specify whether operations within a batch are allowed to be reordered. Because batches can potentially contain a large number of operations, the additional information can be exploited to optimize their execution.

- **Relaxed:** Operations are allowed to be reordered as long as correct execution can be guaranteed, that is, the batch's result corresponds to that of the original batch. For instance, a write operation can never be reordered to be performed before the corresponding create operation. The order of two write operations can be changed to allow merging them, however.

  Inefficient operation orderings can be optimized to the best extent possible; results must be identical to the original ordering.

- **Semi-relaxed:** Operations are allowed to be reordered as long as operations pertaining to the same object are executed in the original order. For example, write operations to several items can be reordered such that each item's write operations are executed together.

  Inefficient operation orderings can be optimized to some extent; results must be identical to the original ordering.

- **Strict:** Operations are not allowed to be reordered. All operations within a batch are executed in exactly the same order as they are added to the batch.

  Inefficient operation orderings can not be optimized. The overhead of reordering can be avoided, however; this is especially useful if developers already perform operations in the optimal order.

The ordering semantics is performance-related as it allows operations to be reordered for more efficient access. It is especially important to group operations of the same type to reduce the amount of network overhead. Additionally, it is usually beneficial to order read and write operations by their offset because this might allow them to be merged. While these optimizations are mainly aimed at delivering improved I/O performance, they can also help to reduce the load on other involved components such as the central processing unit (CPU) and network interface card (NIC).

3.4.5. Persistency

The persistency semantics can be used to specify if and when data and metadata must be written to persistent storage. This can be used to enable client-side write caching whenever possible. To support different persistency requirements, several levels will be supported:

- **None:** Data might never be written to persistent storage, that is, the data might reside in a client-side cache forever. This can be useful for local temporary data, for example.

  Allows modified data and metadata to be cached indefinitely and be discarded when closing the concerned object.

- **Eventual:** Data will eventually be written to persistent storage, that is, the data might reside in a client-side cache even after the operation finishes. A crash may cause the data to be lost if it has not been transferred to the file system servers. The period until the data is written is unspecified.

  Allows caching modified data and metadata for an unspecified amount of time.

- **Immediate:** Data will be written to persistent storage immediately, that is, as soon as the operation finishes the data will not be cached anymore.

Data and metadata can not be cached; all data and metadata must be immediately sent to the appropriate servers.

The persistency semantics is performance-related and allows caching modified data and metadata locally. For example, temporary data can be cached more aggressively and does not necessarily need to be written to persistent storage at all. This can be especially advantageous when different levels of storage such as node-local SSDs are available as it allows writing the temporary data to the fast local storage without communicating via the network at all.

3.4.6. Safety

The safety semantics can be used to specify how safely data and metadata should be handled. It provides guarantees about the state of the data and metadata after the execution of a batch has finished.

- **None:** No safety guarantees are made, that is, data and metadata might be lost due to network or storage errors.

  Data and metadata are sent to the file system servers but no checking is done on whether the changes have been successful or not.

- **Network:** It is guaranteed that changes have been transferred to the servers as soon as the operation finishes.

  Data and metadata are sent to the file system servers and their reply is awaited.

- **Storage:** It is guaranteed that changes have been stored persistently on the storage devices as soon as the operation finishes.

  Data and metadata are sent to the file system servers and their reply is awaited. Additionally, the file system servers flush the changes to disk before sending their reply.

The safety semantics is performance-related by allowing to adjust the overhead incurred by data safety measures. For example, on the one hand, disabling data safety can be used to eliminate one of two network messages by not requesting the server's acknowledgment when sending unimportant data; this allows having more operations in-flight because their results do not have to be received and processed before sending the next operation.[13] On the other hand, it can be used to make sure that important data will survive a system failure by flushing it to the storage devices immediately.

[13] Batches can be used to reduce this problem to a certain extent by also batching replies. The general problem remains the same, however: Waiting for a reply before sending the next operation at least halves the throughput.

3.4.7. Further Ideas

The semantics presented above are going to be implemented and evaluated as part of this thesis. However, even more semantical aspects lend themselves to being configurable and will be briefly presented for completeness.

Redundancy

Redundancy semantics could provide users with a convenient way to store multiple copies of file data or metadata. This could be used to ensure that very important data is safe in case of system failures such as broken hard disk drives (HDDs). Similar options are already being offered by providers of long-term archival services [Ger14].

While this feature has proven its worth in the context of long-term archival, it is not clear whether the decision to store multiple copies of data and metadata should be taken by the users of file systems. Therefore, this option is only mentioned here for reference. Parallel distributed file systems are usually deployed in such a way that the loss of single storage devices does not result in data loss. Consequently, it might make more sense to leave this decision up to the storage system's administrators. In any case, proper accounting of the used file system resources is necessary, otherwise users could simply force redundant storage of all data without consequences.

Security

Security semantics could be changed depending on the file system environment, enabling or disabling more strict permission checks. JULEA's current security policy checks the permissions once when opening a collection or an item. That is, even if the ownership of said collection or item is changed, all clients still holding an open handle will continue to have access to it. Other environments might have different requirements regarding the security policy, however.

Conducting these checks frequently – for example, for every access – can severely impact performance because the required metadata has to be fetched. Therefore, it would be worthwhile to consider making the security policy dynamic through this extension; for instance, the following configurations are conceivable:

- **None:** No security policy is enforced, that is, every client can access and modify all data and metadata.
- **Open:** Permissions are only checked when opening a collection or an item and not rechecked while the client still holds an open handle.

  This is the current security policy.
- **Time-based:** Permissions are rechecked periodically but not for every access.

- **Strict:** Permissions are rechecked for every access.

Obviously, the security semantics would need to be stored together with the relevant file system object and should only be changeable by the object's owner; otherwise, other users could simply specify different security semantics to circumvent permission checks.

Transformation

Transformation semantics could be useful to allow users to transform the data in some way – for example, by compressing, deduplicating or encrypting it. Moving this functionality into the file system would have the advantage of being completely transparent to users and applications. For instance, application developers usually know whether it makes sense to compress the produced data and could easily use this semantics to handle it appropriately without the need to painstakingly adapt each application or I/O library.

As illustrated previously, today's HPC applications can produce tremendous amounts of data due to the ever increasing computational power of supercomputers. The storage systems, however, usually do not scale as well. One way to alleviate this problem is to compress the data. Previous studies have shown that compression can reduce power consumption as well as increase performance in certain use cases [CDKL14, KKL14]. Other techniques such as deduplication can also help to reduce the amount of stored data [MKB+12]. Nevertheless, due to their associated costs, it makes sense to only apply them when there is a clear benefit.

3.4.8. Interactions

All previously presented semantical aspects can be combined arbitrarily, resulting in a huge amount of possible configurations.[14] While some combinations of semantical settings do not actually affect each other or might simply be unreasonable, there are some interesting interactions between some of them:

- **Concurrency: None**
  - It is possible to set the **atomicity** semantics to **none** because no operations will be executed in parallel. Consequently, it is impossible for concurrent operations to observe partially completed operations.
  - The **consistency** semantics can be set to **none** because the relevant file system objects will not be modified by other clients concurrently. Consequently, it is possible to aggressively cache data and metadata.

[14] To be precise, there are currently six semantical aspects with three different settings each; this results in $3^6 = 729$ possible combinations.

- **Concurrency: Non-overlapping**
  - It is possible to set the **atomicity** semantics to **none** if only write operations are performed. Because write operations will only write to non-overlapping regions of items, it is not necessary to lock them if no concurrent read operation could potentially observe partial writes.

- **Persistency: None**
  - It is possible to set the **safety** semantics to **none** because data will not be sent to the data servers immediately. Therefore, it is not necessary to enforce strong safety semantics.

For simplicity and performance reasons, the semantics will not be checked for conflicts; application developers are responsible for ensuring that no contradictory semantics will occur. For instance, different clients accessing the same file system object with a mix of non-overlapping and overlapping concurrency semantics at the same time will lead to undefined behavior.

3.4.9. Templates

To provide application developers with a convenient way of using different semantics, semantics templates will provide predefined templates for specific use cases. The following list provides an overview of the semantics templates that will be available in the prototype; it also lists their concrete settings and reasonings for those:

- **Default:** This template provides JULEA's default semantics. It is optimized for concurrent clients executing non-overlapping operations; this is the kind of access pattern that is often found in contemporary scientific applications.
  - Atomicity: None
    Atomicity is rarely required in parallel applications because I/O is usually done in separate read and write phases.
  - Concurrency: Non-overlapping
    Parallel applications commonly write shared files using non-overlapping accesses because each client is responsible exclusively for part of the data.
  - Consistency: Eventual
    As reading and writing is usually done in separate I/O phases, it is also not necessary to provide immediate consistency.
  - Ordering: Semi-relaxed
    The actual ordering of I/O is usually not important as long as the result is identical to the one specified by the application developer.

  - Persistency: Immediate
    Write operations should be synchronous by default to follow the principle of least astonishment.
  - Safety: Network
    Completed operations should have reached the file system servers as application crashes occur more frequently than file system server crashes.

- **POSIX:** This template is intended to mimic the current POSIX semantics as closely as possible. It is provided for backwards compatibility with applications that depend on POSIX semantics being available.
  - Atomicity: Operation
    Even though POSIX does not strictly mandate atomic operations (see Section 2.6.1 on pages 42–43), this is a common expectation.
  - Concurrency: Overlapping
    To correctly handle arbitrary access patterns, overlapping accesses have to be supported.
  - Consistency: Immediate
    Changes to file system objects have to visible immediately to all clients, as specified by POSIX.
  - Ordering: Strict
    Even though POSIX does not explicitly mention the ordering of operations, it might have an influence on the visibility of changes to other clients.
  - Persistency: Immediate
    The same reasoning as for the default semantics template applies.
  - Safety: Network
    The same reasoning as for the default semantics template applies.

- **Temporary (local):** This template is tuned for process-local temporary data. Its semantics should also allow for transparent use of advanced technologies such as burst buffers.
  - Atomicity: None
    Atomicity is not required because no concurrent accesses will be performed.
  - Concurrency: None
    No concurrent accesses will be performed because each process will access its own data.
  - Consistency: None
    Consistency is not necessary as no concurrent accesses will be performed.
  - Ordering: Semi-relaxed
    The same reasoning as for the default semantics template applies.

- Persistency: None
  As the data is only of a temporary nature, it does not have to be stored persistently within the file system.
- Safety: None
  Safety is not required because temporary data can be recreated if necessary.

The predefined semantics templates obviously can not cover all possible use cases. Therefore, they should be viewed as bases upon which application-specific semantics can be built. While it might be desirable to have support for user-definable semantics templates, such functionality will not be included in the initial prototype; it will, however, be possible to easily adapt the templates as shown in the following example.

```
1 atomic_semantics = new Semantics(DEFAULT_SEMANTICS);
2 atomic_semantics.set(ATOMICITY, ATOMICITY_OPERATION);
3
4 sync_semantics = new Semantics(POSIX_SEMANTICS);
5 sync_semantics.set(SAFETY, SAFETY_STORAGE);
```

Listing 3.5: Adapting semantics templates

Listing 3.5 shows how to adapt the predefined semantics templates. In the first example, the default semantics are modified to provide atomic access (lines 1–2); this is similar to enabling MPI-IO's atomic mode. In the second example, the POSIX semantics are adapted to provide synchronous I/O (lines 4–5); this is similar to specifying `O_SYNC` when opening a file using the POSIX interface.

However, JULEA's concept is more flexible because it allows the semantics to be applied selectively by associating them with batches. In contrast, opening a POSIX file with `O_SYNC` implies that all I/O operations will be synchronous.

The presented semantics parameters are a first proposal of factors that are important for HPC applications. They have been determined by analyzing the use cases of applications as well as the underlying causes for prevailing performance problems found in contemporary parallel distributed file systems. More analyses and discussions are necessary to come up with a final list that is suitable for widespread adoption in other file systems and I/O interfaces.

3.5. Data and Metadata

3.5.1. Distribution

By default, data will be distributed among all available data servers using a round-robin scheme as commonly found in parallel distributed file systems. However,

support for multiple distribution schemes will be provided to allow optimizing I/O performance. The distribution of metadata will also be supported explicitly to avoid performance bottlenecks and scaling problems.

Previous studies have shown that different distributions can be beneficial for certain kinds of files. For instance, distributing small files across many servers often does more harm than good [KKL08, KKL09, CLR+09]. As application developers can most accurately estimate the expected benefits of adapting the distribution, it has to be easy for them to manually adapt the distributions; that is, the I/O interface should have direct and adequate distribution support.

3.5.2. Metadata Management

As shown in Section 2.2.1 on pages 27–29, file systems usually keep a lot of metadata. To reduce JULEA's metadata overhead, collections and items will feature only a reduced set of metadata. The following list gives an overview of the metadata that needs to be stored for the different types of file system objects:

- Name (collection and item)
- Ownership (collection and item)
  - User
  - Group
- Distribution (item only)
  - *Varies depending on the chosen distribution*
- Status (item only)
  - Size
  - Modification time

As already mentioned, unnecessary metadata will be omitted. For example, the last access time will not be stored because it would introduce write overhead for each read operation. While this information might be appropriate for general purpose file systems, its usefulness in parallel distributed file systems targeted at HPC workloads is questionable.[15]

File system metadata is usually stored in inodes that have a fixed format. Due to JULEA's dynamic nature, its metadata does not fit into such a fixed schema, because different semantics can make it necessary to store different metadata. One obvious

[15] In fact, current versions of Linux only update the last access time under certain circumstances even for local file systems due to the implicit overhead [Zak14]. Linux versions 2.6.30 and up default to `relatime`, which is explained in Section 2.6.1 on pages 42–43.

example is the distribution information which varies based on the chosen distribution function. While it would be possible to reserve a certain amount of space for distribution information and future extensions, this would introduce the same inflexibilities found in current inode designs.

However, other factors can also make it necessary to modify the metadata schema. One of those factors is the rate with which the metadata is accessed and modified. Regarding its access rate, the metadata can be separated into three groups:

1. **Write-once**
   - The data distribution metadata is written once when the item is created and not modified afterwards.
2. **Occasionally changing**
   - The name and ownership metadata is only modified if explicitly requested by the user.
3. **Frequently changing**
   - The status metadata is potentially modified for each access.

While write-once and occasionally changing metadata can easily be kept on the metadata server, also storing frequently changing metadata there can result in a performance bottleneck in specific cases. Fundamentally, there are two possibilities to manage this information:

1. Frequently changing metadata is stored on the metadata servers. Even if metadata is distributed across multiple metadata servers, the metadata of a single item is usually managed by exactly one metadata server. A large amount of clients modifying a single item concurrently can cause a storm of updates on this single metadata server, causing the already mentioned performance bottleneck.
2. Frequently changing metadata is not stored explicitly, but rather retrieved and computed on demand. This can be achieved by collecting information about the different data stripes from all data servers. For instance, while the item's size can be summed up over all servers, only the maximum of all servers' last modification times would be used to determine the item's modification time. These can be expensive operations as they involve contacting a potentially large amount of data servers.

JULEA's concurrency semantics provide information about the number of clients accessing an item and can thus be conveniently used to determine the method to use; this will make sure that frequently changing metadata such as the file size and modification time are only stored explicitly for non-parallel workloads.

Even though parts of the metadata are write-once or occasionally changing, large numbers of concurrently accessing clients can still cause congestions inside the metadata servers due to high rates of metadata operations. Batches provide the means to solve this particular problem by aggregating many metadata operations and thus reducing the metadata overhead.

Summary

This chapter has illustrated the design of JULEA's parallel distributed file system and I/O interface; the design includes the general architecture, the namespace, the interface, the semantics and considerations regarding data and metadata handling. JULEA's possible semantics, their interactions and consequential optimization opportunities have been highlighted specifically. In contrast to traditional I/O interfaces, JULEA allows its semantics to be adapted dynamically; this allows applications to fine-tune the file system's behavior according to their I/O requirements instead of the other way around.

Chapter 4.

Related Work

In this chapter, an overview of existing work from the fields of parallel distributed file systems, I/O optimizations, interfaces and semantics will be given. Comparisons with existing approaches will focus on their ability to provide semantical information for optimization and convenience purposes as well as their capabilities regarding dynamic semantics.

4.1. Metadata Management

The traditional approach of metadata management is to have one or more metadata servers and partition the file system namespace statically. In addition to this, more sophisticated techniques for handling the increasing requirements regarding metadata performance have started to emerge. A selection of popular ones will be presented and compared to JULEA's design.

GIGA+ GIGA+ presents a new file system directory service that is supposed to handle millions of files and has been integrated into OrangeFS [PGLP07, PG11]. It stripes directories over many servers by effectively splitting directories into multiple partitions by hashing the name of directory entries; the appropriate partitions and servers are found using low-overhead bitmaps. It supports traditional POSIX[1] semantics and is built for high throughput and scalability by minimizing the necessary amount of shared state. Additionally, it can handle incremental growth of directories as well as provide adequate burst performance.

GIGA+'s design is built and improved upon in [XXSM09] to efficiently support a trillion files by employing an adaptive two-level directory partitioning scheme. The presented approach allows scalable access to very large directories and dynamic partitioning of the file system namespace for load balancing purposes.

One of GIGA+'s similarities with JULEA is the fact that metadata is also split into different categories: Infrequently updated metadata such as the owner or creation time are managed at a centralized server; highly dynamic metadata such as access and

[1] Portable Operating System Interface

modification times are allowed to vary across servers. The latter is then dealt with by the clients that have to ensure consistency by themselves.

Coupled Data and Metadata Instead of providing dedicated metadata servers, it is also possible to eliminate the metadata servers as shown in [ADD+08]. The authors move as much metadata as possible to the data servers, leaving only a dedicated server handling directory operations. On the one hand, this approach has the advantage that it is not necessary to contact additional metadata servers when the data servers have to be contacted anyway. On the other hand, metadata and data operations influence each other because the hardware resources are shared.

Additionally, it makes it harder to handle metadata and data separately: As mentioned previously, it makes sense to use alternative storage technologies such as dedicated solid state drives (SSDs) for metadata because metadata and data servers usually experience completely different access patterns.

hashFS A new file system approach is presented in [LMB10, LCB13] that eliminates the current need for many small accesses to get the metadata of all path components during path lookup. By using the hashed file path to directly look up the related data and metadata, this can be reduced to only require one read operation per file access. While this can significantly decrease metadata overhead and increase small file performance, the use of the full file path for hashing implies that the renaming of parent directories causes the hashes of all their children to change. There are two approaches to handle this fact:

1. All hashes are recomputed immediately after a rename operation. This approach might lead to a lot of computational overhead, depending on the rate of rename operations in the file system.

2. Rename operations are recorded in a translation table. While this approach avoids costly recomputations, additional translation table lookup operations have to performed for each metadata access.

As can be seen, both approaches introduce additional management effort. JULEA does not use hashed path lookups for this reason, but implements a flat namespace to keep metadata lookup overhead low.

SmartStore SmartStore provides a new metadata organization paradigm for file systems [HJZ+09]. The authors have identified the traditional hierarchical file system namespace as an obstacle for future scalability requirements. Instead of providing a hierarchical namespace, SmartStore allows searching for data using database-like queries. To be able to efficiently execute these queries, SmartStore exploits semantical information to group metadata of correlated files.

In contrast to SmartStore's query-based approach, JULEA provides a traditional namespace but limits its depth to minimize the metadata overhead.

4.2. Semantics Compliance

As already mentioned in Chapter 2, the POSIX input/output (I/O) interface and semantics are a common choice among parallel distributed file systems. The following list gives an overview of the supported I/O semantics of popular parallel distributed file systems and their degree of standards compliance:

- **Lustre:** Lustre's goal is to provide a fully POSIX-compliant file system even though its current implementation might not be 100 % compliant. Among other features, it provides POSIX-compliant handling of file sizes even in the context of striping [SKH+08].
- **GPFS:** GPFS[2] has been designed to be fully POSIX-compliant. Like Lustre, GPFS can guarantee POSIX-compliant handling of file sizes and also supports strict POSIX atomicity semantics [SKH+08, JKY00].
- **OrangeFS:** OrangeFS is not POSIX-compliant but provides support for atomic non-overlapping writes, even if the write operations are non-contiguous [TSP+11, LRT04]. One of OrangeFS's new goals is to explore configurable semantics.
- **CephFS:** Ceph's file system CephFS provides near-POSIX semantics [WBM+06]. One major difference is the fact that write operations are not guaranteed to be atomic if they cross object boundaries. That is, similar to OrangeFS, if two clients write to the same overlapping location, the resulting data might contain partial data from different clients.
- **GlusterFS:** GlusterFS claims to be fully POSIX-compliant; no further details are provided [Glu11].

As can be seen, even though many parallel distributed file systems provide some kind of POSIX compliance and some are even fully POSIX-compliant, there are subtle differences depending on the used file system. Therefore, application developers still have to make sure that their applications work correctly on different file systems, even though they use a seemingly portable I/O interface. One of the reasons for this state of affairs is the fact that supporting POSIX semantics in a parallel distributed file system is a complex task; striving to do the same while providing high performance only exacerbates the problem.

[2] General Parallel File System

Another problem stems from the fact that the POSIX specifications are sometimes not explicit enough and allow for different interpretations of the standard. For instance, the different possible interpretations of POSIX's atomicity semantics are the subject of an ongoing debate (see Section 2.6.1 on pages 42–43). This ambiguity can also lead to unexpected behavior; the `write` function's manual contains the following statement:

> *"POSIX requires that a read(2) which can be proved to occur after a write() has returned returns the new data. Note that not all filesystems are POSIX conforming."*
>
> Source: [The14b]

Even for local file systems, this behavior does not imply that data has been stored on a storage device persistently; an additional call of `fsync` or `fdatasync` is required to make it so. In the context of parallel distributed file systems, however, this has additional implications that are illustrated based on the number of client machines accessing a shared file:

- **Single client machine:** If the accesses to the shared file originate from only a single client machine, the parallel distributed file system does not have to send the data to the data servers for every single `write` call. Instead, it can aggregate the data in the machine-local cache to increase performance. This behavior is POSIX-compliant because all `read` calls can be satisfied from the client machine's cache. This allows for high performance even in the presence of suboptimal I/O patterns because caching can be used to mitigate the problem.
- **Multiple client machines:** If the accesses to the shared file originate from multiple client machines, the parallel distributed file system has to modify its behavior. It has to send every `write` call's data to the data servers immediately or employ a locking scheme because clients on different client machines might issue `read` calls that have to return the newly written data according to POSIX.

Consequently, applications will exhibit different performance characteristics depending on the currently used number of client machines even though the actual I/O pattern does not change. This can be surprising for application developers and is another fact to be taken into account when performing parallel I/O. The effects of this behavior will be examined in more detail in Chapter 6.

4.3. Adaptability

There are a few approaches to provide configurable behavior and semantics in parallel distributed file systems. However, they are usually limited to single aspects of the file system or too static because they do not allow changes at runtime [PGG+09].

MosaStore MosaStore is a versatile storage system that is configurable at application deployment time and thus allows application-specific optimizations [AKGR10].

This approach is similar to the JULEA approach, however, MosaStore provides a storage system bound to specific applications instead of a globally shared one. Additionally, the storage system can not be reconfigured at runtime and keeps the traditional POSIX I/O interface.

CAPFS CAPFS introduces a new content-addressable file store that allows users to define data consistency semantics at runtime [VNS05]. While providing a client-side plug-in API allows users to implement their own consistency policies, CAPFS is limited to tuning the consistency of file data and keeps the traditional POSIX interface. Additionally, the consistency semantics can only be changed on a per-file basis.

Configurable Security In [GAKR08], the authors present a configurable security approach that allows using scavenged storage systems – that is, storage systems consisting of unused workstation hardware – in trusted, partially trusted and untrusted environments in a secure way.

While JULEA does not use scavenged storage hardware and currently does not support dynamic security semantics, the cited work shows that configurable security can be achieved with relatively low overhead.

4.4. Semantical Information

The problem of missing semantical information making heuristics necessary to improve performance is of course not unique to file systems. Many fields in informatics are affected by this and can benefit from the additional knowledge provided by developer-provided information.

Custom Metadata In [SNAKA+08], the authors propose to use custom metadata such as extended attributes for cross-layer optimizations in storage systems. This means that applications can provide additional information to the storage system via custom metadata and vice versa. The authors give several examples about how this can be used to improve the storage system's efficiency:

- Files can be annotated as temporary and thus be treated differently: Temporary files can be cached more aggressively or be purged automatically.
- Annotations can be used to specify quality of service requirements such as durability, security and privacy.
- Consistency requirements can be specified to manage performance tradeoffs.

The idea of custom metadata is very similar to JULEA's semantical information. The main difference between the two approaches is that custom metadata is explicitly stored and interpreted by the storage system, while JULEA's semantical information is specified for each batch and passed directly to the file system. Additionally, the authors present a generic approach, while JULEA is tailored to high performance computing (HPC) applications.

Amino Amino's authors have designed and implemented a file system supporting atomicity, consistency, isolation and durability (ACID) semantics [Wri06, WSSZ07]. Amino is a POSIX-compliant user space file system that uses the `ptrace` tracing framework to intercept POSIX I/O system calls. It is built on top of Berkeley DB (BDB), which provides a well-tested infrastructure for transactions.

```
1 amino(BEGIN_TXN, "/path/to/file", 0);
2
3 fd = open("/path/to/file", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR |
     ↪ S_IWUSR);
4 pwrite(fd, data, sizeof(data), 0);
5 close(fd);
6
7 amino(COMMIT_TXN, 0);
```

Listing 4.1: Amino transactions

Listing 4.1 shows pseudo code for Amino transactions. The transaction is started for a given path using the `amino` function with the `BEGIN_TXN` parameter (line 1). Afterwards, arbitrary POSIX I/O functions can be executed (lines 3–5). Finally, the transaction is committed by passing the `COMMIT_TXN` parameter to the `amino` function. In case of an error, the transaction could be aborted using the `ABORT_TXN` parameter.

As can be seen, the concept of transactions is similar to that of JULEA's batches even though the latter do not offer ACID support. A downside of Amino's transactions is that they can not be adapted dynamically when full ACID semantics are not required.

Networking A feature found in TCP[3] is the Nagle algorithm that tries to aggregate small network messages into larger ones to reduce the number of packets sent over the network. For instance, an application sending ten messages containing 1 byte each would generate ten network packets with a size of at least 41 bytes each.[4] Conse-

[3] Transmission Control Protocol

[4] In addition to the actual data, each packet carries several headers. While TCP adds a header of 20 bytes, the size of the header added by the Internet Protocol (IP) depends on the protocol version: An IPv4 header has a size of 20 bytes and an IPv6 header has a size of 40 bytes. The underlying network technology – such as Ethernet – usually increases the packet size even further.

quently, this application would generate ten network packets with a cumulative size of 410 bytes. The Nagle algorithm can aggregate all these small messages into one network packet with a size of 50 bytes, reducing the overhead by more than 85 %.

However, the Nagle algorithm uses heuristics to decide which messages to aggregate and when to actually send a network packet. Due to several factors, this can result in delays of up to 500 ms [MSMV00]. While it is possible to disable the Nagle algorithm using `setsockopt`'s `TCP_NODELAY` option, this undoes all possible optimizations. A better approach is the so-called *corking*: The `TCP_CORK` option allows developers to manually control the message aggregation feature [MM01]. This can be used to *cork* the connection before sending many small messages, which causes them to be queued and aggregated instead of being sent immediately. As soon as the connection is *uncorked*, the queued messages are flushed and sent using as few network packets as possible.

```
1  int fd;
2  int flag;
3
4  flag = 1;
5  setsockopt(fd, IPPROTO_TCP, TCP_CORK, &flag, sizeof(flag));
6
7  write(fd, &flag, sizeof(flag));
8  write(fd, &flag, sizeof(flag));
9  write(fd, &flag, sizeof(flag));
10
11 flag = 0;
12 setsockopt(fd, IPPROTO_TCP, TCP_CORK, &flag, sizeof(flag));
```

Listing 4.2: TCP corking

Listing 4.2 shows code demonstrating the use of TCP corking. The file descriptor `fd` is assumed to be an open network socket (line 1); the integer variable `flag` will be used to pass arguments to the `setsockopt` function and used as dummy data (line 2). Before sending any data, the `setsockopt` function is used together with the `1` flag to cork the connection (lines 4–5). Afterwards, several small messages are sent (lines 7–9). Finally, the connection is uncorked using the `0` flag (lines 11–12).

As can be seen, this is similar to the concept of batch operations in JULEA but on a much lower level. In fact, the additional semantical information provided by JULEA's batch operations can and will be utilized to make use of this TCP optimization.

Memory Ordering In parallel programming for shared memory architectures, memory ordering and consistency are important factors for both performance and correctness. Because central processing units (CPUs) usually reorder memory load and store

operations to improve performance, it is necessary to take this fact into account when using multiple threads to access shared memory [GLL+90, GGH91].

```
1 /* Thread 1: */
2 x = 1;
3 r1 = y;
4
5 /* Thread 2: */
6 y = 1;
7 r2 = x;
```

Listing 4.3: Memory operation reordering

Listing 4.3 shows the memory operations of two concurrent threads (lines 1–3 and 5–7, respectively). `x` and `y` are variables in the shared memory that are initialized with `0`. Each thread writes to one of the variables (thread 1 to `x` and thread 2 to `y`) and then reads the variable written to by the other thread into a register (threads 1 and 2 write into `r1` and `r2`, respectively).

The order of operations suggests that at least one of the registers will contain the value `1` after both threads have finished running. However, due to reordering, both registers could actually contain the value `0`: The CPU could first execute both load operations into the registers and then store the values into `x` and `y`.

```
1 #include <stdatomic.h>
2
3 atomic_int guide = ATOMIC_VAR_INIT(42);
4 atomic_init(&guide, 42);
```

Listing 4.4: Atomic variables in C11

Modern concepts such as those supported by C++11 and C11 allow developers to specify different constraints to achieve optimal performance while still maintaining correct execution of their applications [ISO11]. Those features are usually leveraged by making use of atomic variables. Listing 4.4 shows how atomic variables can be declared and defined: First, it is necessary to include the `stdatomic.h` header providing the atomic functionality (line 1). This makes available new atomic data types that are denoted by the `atomic_` prefix.[5] Afterwards, an atomic integer is declared and initialized using the `ATOMIC_VAR_INIT` function (line 3). Alternatively, it is possible to initialize atomic variables using the `atomic_init` function (line 4).

Depending on the used CPU architecture, memory operations can be reordered differently. Consequently, C++11 and C11 allow providing information that can be

[5] Alternatively, data types can be made atomic using the `_Atomic` type qualifier.

used to produce the optimal code for each CPU architecture. The `memory_order` type defines several possible orderings that can be used to specify the semantics necessary to obtain the correct results:

- `memory_order_seq_cst`: Guarantees that no reordering is performed and provides sequential consistency.
- `memory_order_acquire`: Guarantees that no subsequent load operation is moved before the current one.
- `memory_order_release`: Guarantees that no preceding store operation is moved beyond the current one.
- `memory_order_acq_rel`: Combines the previous two guarantees.
- `memory_order_consume`: Provides guarantees similar to `memory_order_acquire` but only for operations that are dependent on the current load operation.
- `memory_order_relaxed`: All orderings are allowed.

Using these memory ordering settings, the above example can be rewritten to solve the problem by forcing the CPU to not reorder operations in an incorrect way.

```
1 /* Thread 1: */
2 atomic_store_explicit(&x, 1, memory_order_seq_cst);
3 r1 = atomic_load_explicit(&y, memory_order_seq_cst);
4
5 /* Thread 2: */
6 atomic_store_explicit(&y, 1, memory_order_seq_cst);
7 r2 = atomic_load_explicit(&x, memory_order_seq_cst);
```

Listing 4.5: Atomic operations in C11

Listing 4.5 shows the example from Listing 4.3 on the preceding page in a modified form to use atomic operations and sequential consistency [Lam79]. This guarantees that at least one of the registers contains the value `1` because the operations are not allowed to be reordered due to the requested memory ordering.

Being able to specify the memory ordering has several advantages: On the one hand, it allows the compiler to produce optimal code for the given CPU architecture. On the other hand, it is not necessary to force sequential consistency at all times to guarantee correctness. JULEA's ordering semantics provide the same benefits by allowing the developer to provide additional semantical information to optimize execution.

ADIOS As shown in Section 2.5.6 on pages 40–42, ADIOS[6] offers a novel and developer-friendly I/O interface: It allows specifying the I/O configuration in an XML[7] file that can be changed without recompiling the application. Newer releases of ADIOS have added features to provide improved performance and convenience:

1. **Read scheduling:** Version 1.4 has added support for scheduling read operations. Several read operations can be scheduled using the `adios_schedule_read` function and then executed using the `adios_perform_reads` function.

2. **Data transformations:** Version 1.6 has added support for on-the-fly data transformations. Each variable can be assigned a different transformation using the XML configuration file or the `adios_set_transform` function.[8] Currently, data transformations allow compressing variables using different compression algorithms. However, as the transformations are implemented in the form of plug-ins, additional transformations can be added [BLZ+14].

```
1 adios_schedule_read(adios_fd, NULL, "var1", 0, 1, &var1);
2 adios_schedule_read(adios_fd, NULL, "var2", 0, 1, &var2);
3 adios_schedule_read(adios_fd, NULL, "var3", 0, 1, &var3);
4 adios_perform_reads(adios_fd, 1);
```

Listing 4.6: ADIOS read scheduling

Listing 4.6 shows an example of read scheduling. First, three variables `var1`, `var2` and `var3` are scheduled for reading using the `adios_schedule_read` function (lines 1–3). Afterwards, all scheduled reads are executed using the `adios_perform_reads` function (line 4). In the best case, this allows reading a contiguous chunk of data instead of many small ones.

```
1 <var name="matrix" type="double" dimensions="rows,columns"
    ↪ transform="bzip2"/>
```

Listing 4.7: ADIOS variable transformation (XML)

Data transformations can be assigned to individual variables in the XML file, as shown in Listing 4.7. The `matrix` variable, which is a matrix of dimensions `rows`×`columns` and contains `double` values, is transformed using the `bzip2` data transform.

[6] Adaptable IO System
[7] Extensible Markup Language
[8] The `adios_set_transform` function has been added in ADIOS version 1.9.

```
1 int64_t var_id;
2
3 var_id = adios_define_var(...);
4 adios_set_transform(var_id, "bzip2");
```

Listing 4.8: ADIOS variable transformation

Listing 4.8 shows how data transformations can be used manually. First, a variable has to be defined using the `adios_define_var` function (line 3); the function returns a 64-bit integer identifier (ID). Afterwards, it is possible to set the data transformation using the `adios_set_transform` function by passing the variable ID as well as the desired data transformation `bzip2` (line 4).

Read scheduling is very similar to JULEA's batches: It allows aggregating multiple operations for improved performance. However, in contrast to JULEA's batches that can contain arbitrary operations, ADIOS's read scheduling is limited to read operations. The availability of this information could be exploited when using ADIOS on top of JULEA. ADIOS's data transformations implement the transformation semantics proposed in Section 3.4.7 on page 67, demonstrating their usefulness. Exposing this functionality in a convenient way lifts the burden of manually implementing data compression from the application developers.

Summary

This chapter has presented related work and compared it with JULEA's approach. The related concepts have been grouped according to their metadata management, the adaptability of semantics and the ability of specifying additional semantical information. Additionally, a comparison as to the semantics compliance of several parallel distributed file systems has been performed. While other fields of informatics have used semantical information to improve performance, support for similar facilities is very sparse with respect to file systems.

Chapter 5.

Technical Design

In this chapter, the technical design of the file system and I/O interface with dynamically adaptable semantics will be presented. Because both have been designed with modularity in mind, extension points will be highlighted specifically. Additionally, important software components that have been used during the development phase will be introduced.

Modifying an existing parallel distributed file system to implement the design presented in Chapter 3 seems like an obvious choice. While it has been considered to use Lustre or OrangeFS, this has been deemed unreasonable because neither Lustre nor OrangeFS are prepared for this kind of functionality. Therefore, large parts of the their existing implementations would have to be changed. On the one hand, their respective input/output (I/O) interfaces would have needed to be adapted to support batch operations and to allow specifying semantical information. On the other hand, modifications to the actual file system code would have been required to take advantage of the information delivered by the interface.

Previous experience has shown that it can be difficult to adapt existing architectures for features affecting the fundamental file system functionalities [KKL08, KKL09]. In addition to the actual implementation, this would have meant getting to know the existing large code bases: Lustre and OrangeFS contain more than 550,000 and 250,000 lines of code, respectively. While OrangeFS can be run completely in user space, Lustre's client and server code have been implemented in the form of kernel modules and patches (see Chapter 2). Consequently, even more major modifications would have been required because Lustre's client interface is limited by Linux's virtual file system (VFS) layer as described in Section 2.2 on pages 26–29. Due to its complexity, OrangeFS's native I/O interface is usually not used directly but only through either MPI-IO or POSIX[1]; that is, changes to the OrangeFS interface would have also required modifications to at least one of those additional I/O interfaces. Therefore, it has been decided to implement a prototypical file system with all other necessary components to evaluate the proposed design. The resulting framework is called JULEA.

While the file system prototype has been built from scratch to suit the needs of the proposed I/O interface, care has been taken to use existing technology whenever

[1] Portable Operating System Interface

possible. This approach has two major advantages: First, it helps minimizing the development overhead while implementing the already complex parallel distributed file system. Second, widely used software components are expected to be well-tested and thus contain fewer bugs than self-developed ones.

Because developing and maintaining a kernel module can present a significant burden, JULEA will run completely in user space. JULEA provides a user space library that can be linked to applications, allowing them to use the JULEA I/O interface. An additional user space daemon handles storing the file data on the data servers. Metadata is stored in a NoSQL database system called MongoDB [Cat10]. It can be scaled horizontally by simply adding more servers and will be explained in more detail in Section 5.2 on pages 91–93. The library communicates via TCP[2] with both the JULEA daemons and the MongoDB servers running on the data and metadata servers, respectively. By providing all functionality in user space, JULEA is largely independent of the used operating system and can be easily ported to new software environments. An increasing number of parallel distributed file systems – such as CephFS, GlusterFS and OrangeFS – also prefer this kind of architecture.

Implementing a parallel distributed file system in user space has several advantages: First, user space code is much more portable because – in contrast to the internal kernel interfaces – the user space application programming interface (API) and application binary interface (ABI) are stable. Second, in case of problems, user space applications can simply be restarted; kernel problems might make it necessary to restart the whole machine. Third, support for user space performance analysis and debugging is better than for kernel space due to comprehensive tool support.

However, providing file system functionality from user space also has disadvantages: For instance, instead of faster mode switches, more expensive context switches might be necessary because file systems will run as normal user space processes.[3] In real-world usage, this problem can typically be neglected due to the high latencies that occur when accessing the storage devices and the network.

Many distributed file systems use underlying local POSIX file systems to store the actual data. For example, Lustre uses ldiskfs that is based on the ext4 file system; OrangeFS is able to use arbitrary POSIX file systems. Obviously, this introduces additional overhead because a lot of the common file system functionality – such as path lookup or permission checking – is duplicated. While all this functionality is already present and handled in the parallel distributed file system, the underlying local file systems perform the same work redundantly. As presented in Chapter 3, the goal is to use an object store for storing the actual file data. Since an object store only

[2] Transmission Control Protocol

[3] A mode switch denotes a switch from user mode to kernel mode and vice versa. A context switch denotes a switch from one user space process to another process and requires more state to be saved and restored; this makes them slower by a factor of up to 50.

provides the most essential functions like creating, reading, writing and removing objects, this will help avoiding the overhead mentioned before.

In order to provide as much flexibility as possible regarding the underlying storage system, JULEA offers native support for multiple backends in the form of interchangeable modules. These backends implement a common interface that abstracts the actual storage system and provides transparent access to different storage technologies. While this makes it possible to easily support different object stores as intended, it also allows using existing file systems for compatibility reasons.

JULEA's software framework also features auxiliary tools such as command line utilities and a FUSE[4] file system for compatibility with POSIX applications. Additionally, unit tests and benchmarks are included to be able to easily find functionality and performance regressions.

5.1. Architecture

Figure 5.1 on the following page illustrates JULEA's architecture in more detail. As mentioned before, JULEA's architecture comprises three major component classes: clients, data servers and metadata servers. All communication between instances of these three components is handled using TCP. While it is possible to run all of them on the same physical nodes for testing purposes, production environments usually have dedicated nodes for each component instance. The figure shows the default configuration that groups related services together on the same node; however, specific services such as the router and config servers could also be placed on separate nodes.

Client To make use of JULEA's features, client applications simply have to be linked against JULEA's client library called `libjulea.so`. The library provides a cleanly separated namespace for JULEA's functionality: All function names begin with `j_`, all data types are prefixed with `J` and all preprocessor macros and enum values have a leading `J_` in their name.

Applications are usually executed on designated *compute nodes* that are reserved exclusively for computational tasks. There is no limit as to the number of clients; depending on the particular use case, one or multiple clients can be run on each node. While the application and client library communicate via shared memory, the router is connected to via TCP.

- **Application:** The JULEA client library can be used with any kind of application, including parallel applications using MPI[5] and threads.

[4] Filesystem in Userspace
[5] Message Passing Interface

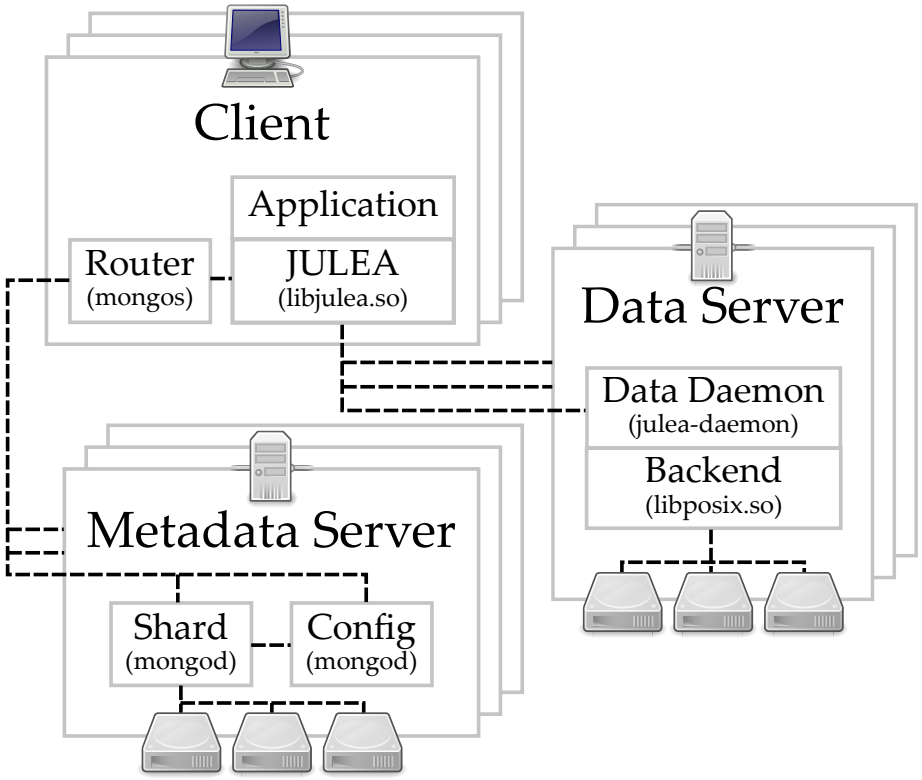


Figure 5.1.: JULEA's general architecture

- **Client library:** All functionality is contained in JULEA's `libjulea.so`. An additional library called `libjulea-private.so` provides internal functionality for other JULEA components; this separation allows minimizing the publicly available interface. Concentrating as much functionality as possible in these central libraries avoids code duplication and facilitates reuse of existing code. The client library transparently handles all communication with the data and metadata servers by communicating with JULEA's data daemon and MongoDB's daemon or router as necessary.

- **Router:** The MongoDB router is used in case MongoDB's sharding is activated. Client applications can connect to the router process `mongos` that behaves like a normal MongoDB server. It retrieves the shard configuration from the MongoDB config server and routes all traffic to the appropriate MongoDB shards.

Metadata Server The metadata servers are executed on a subset of the so-called *storage nodes* and make use of the MongoDB database system. While each of the nodes runs a shard, only three of them house a config server; both communicate via TCP connections. The metadata servers consist of two user space processes each:

- **Shard:** A MongoDB *shard* holds a part of the complete MongoDB database; the

actual distribution is performed by the `mongos` routers. In non-sharded configurations, there is only one MongoDB server that holds the complete database.

- **Config server:** A MongoDB *config server* holds sharding metadata. While it is possible to run only a single config server, production sharded clusters are supposed to contain exactly three config servers for redundancy and safety reasons. All sharding metadata is stored in a special `config` database that can be accessed using the normal MongoDB interface if absolutely necessary.

Data Server The data servers run a user space daemon called `julea-daemon` that handles all I/O on behalf of JULEA's clients. This daemon has access to multiple storage backends that are compiled as shared libraries and loads one of them at startup. In this example, the POSIX backend contained in `libposix.so` is used.

The data servers are also executed on a subset of the storage nodes; depending on the use case, it might or might not overlap with the one used by the metadata servers. Each node houses a single data daemon that is linked to a single storage backend; the data daemon and storage backend communicate via shared memory.

- **Data daemon:** JULEA's data daemon runs as a normal user space process and waits for TCP connections from JULEA's client library. Each one is uniquely associated with one client process and handled by its own dedicated thread.

- **Storage backend:** The storage backend is dynamically loaded by the data daemon on startup. Consequently, only one storage backend can be active at the same time. All I/O requests are handled by the data daemon and delegated to the storage backend for the actual processing.

5.2. Metadata Servers

To reduce the implementation overhead, JULEA's metadata servers are realized using an existing database system. JULEA's metadata design has two main requirements that must be met by the potential candidates:

1. **Scalability:** It must be possible to scale the metadata servers horizontally without much effort. Centralized services can quickly become performance bottlenecks with the ever increasing numbers of accessing clients.

2. **Flexibility:** Metadata must not be constrained into a fixed format. JULEA's dynamic behavior makes it necessary to store different kinds of metadata depending on the current semantics.

Even though traditional SQL database systems such as MySQL Cluster offer possibilities for horizontal scaling [Ora11], they were not considered because the fixed

format of their tables is not suited for JULEA's dynamic metadata format. NoSQL database systems are often designed with horizontal scalability in mind. Additionally, document-oriented NoSQL database systems usually offer the ability to store documents with differing schemas.

5.2.1. MongoDB

MongoDB is such a document-oriented NoSQL database system with support for dynamic document schemas [10g13] that are well-suited for JULEA's non-uniform metadata. A multitude of programming languages are supported through official and third-party client interfaces and libraries. MongoDB supports replication and high availability for use in production environments.

MongoDB's namespace is organized into *databases*, *collections* and *documents*: Multiple related documents can be combined into collections and a database can consist of multiple collections. Collections are referred to by the concatenation of the database name and the collection name with a period. For example, the `bar` collection in the `foo` database would be accessed using `foo.bar`.

Documents are sets of key-value pairs. While keys are always strings, values can have arbitrary data types such as strings, integers or even arrays. This makes MongoDB documents the perfect candidate to store JULEA's metadata.

```
1 {
2     "_id" : ObjectId("51caae667d1a000000000014"),
3     "text" : "Lorem ipsum dolor sit amet, ...",
4     "length" : 42
5 }
```

Listing 5.1: MongoDB document in JSON format

Listing 5.1 shows an exemplary MongoDB document in JSON[6] format. Each document has a unique identifier (ID) called `_id` (line 2); the ID is automatically generated if it is omitted when creating the document. As can be seen, values can be of arbitrary type: While the value belonging to the key `text` is a string (line 3), the value associated with the key `length` is an integer (line 4). By default, the `_id` key is indexed, allowing fast lookups using this key; additional indexes can be added easily, however.

MongoDB also supports a technique called *sharding* that enables horizontal scaling. It allows the documents to be distributed across multiple servers and is performed on a per-collection basis. By default, the distribution is handled automatically by MongoDB. However, it is also possible to specify the so-called *shard key* MongoDB

[6] JavaScript Object Notation

uses to determine the distribution for more fine-grained control; this allows optimizing the way the documents are distributed.

5.3. Data Servers

The data daemon handles all access to item data on behalf of the clients that do not have direct access to the actual storage hardware. Like the JULEA library, it is implemented as a user space application to minimize portability issues. The data daemon is completely threaded and handles each connection in its own separate thread. This ensures that clients can not block each other from proceeding and guarantees fast response times. Its source code can be found in the `daemon` directory and more specifically in `daemon/daemon.c`.

5.3.1. Storage Backends

The JULEA daemon uses so-called *storage backends* to abstract the underlying storage technologies. These backends can be easily exchanged and allow using existing technologies as well as fast prototyping of new approaches. For instance, storage backends can be adapted for a given computer system without having to modify the internals of the daemon. Additionally, they can be used to integrate new approaches such as objects stores into the system. JULEA already includes numerous storage backends for different use cases that can be found in the `daemon/backend` directory:

- **NULL** (`daemon/backend/null.c`): This storage backend is intended for performance measurements of the overall I/O stack. It excludes the influence of underlying storage hardware by returning dummy information and discarding all incoming data.
- **POSIX** (`daemon/backend/posix.c`): This storage backend provides compatibility with existing POSIX file systems. Due to using a full-featured file system as the storage backend, certain functionalities – such as path lookup and permission checking – are duplicated within the I/O stack. It is intended as an interim solution until object stores with sufficient functionality are available.
- **GIO** (`daemon/backend/gio.c`): This storage backend uses the GIO library that provides a modern, easy-to-use VFS API supporting multiple backends including POSIX, FTP[7] and SSH[8]. It is mainly intended as a proof of concept and allows experimenting with GIO's more exotic backends.

[7] File Transfer Protocol
[8] Secure Shell

- **ZFS** (`daemon/backend/jzfs.c`): This storage backend uses ZFS[9]'s data management unit (DMU) to provide a low-overhead data store. Since the underlying object store only provides the most essential I/O operations, no high-level file system functionality is duplicated.
- **LEXOS** (`daemon/backend/lexos.c`): This storage backend uses LEXOS[10] to provide a light-weight data store [Sch13]. The underlying object store only provides basic I/O operations.

Object Stores

The ZFS storage backend uses the JZFS library that has been developed to provide user space access to the ZFS DMU; its source code can be found in the `zfs` directory. It provides a convenient object store interface and can handle ZFS pools, object sets and objects. However, ZFS's DMU interface is largely undocumented and is apparently not intended to be used from user space. Several patches are required to make it work from user space; it is therefore considered experimental and unstable.

While initial evaluations have been promising and have demonstrated good performance, more in-depth analysis has revealed problems regarding multi-threading that have not been solved until now. Because JULEA's data daemon uses multi-threading extensively, it has not been possible to use the ZFS storage backend. Consequently, it is mainly intended as a proof of concept and has been deprecated in favor of the LEXOS storage backend. Because LEXOS is still in an earlier stage of development, the POSIX storage backend remains the default, however.

5.3.2. Backend Interface

JULEA uses a modular approach for the storage backends: They are provided as so-called *modules* in the form of shared libraries that are loaded dynamically by the data daemon at runtime. JULEA defines a common backend interface that is implemented by all storage backends.

```
1 gboolean backend_init (gchar const* storage_path);
2 void backend_fini (void);
3
4 gpointer backend_thread_init (void);
5 void backend_thread_fini (gpointer data);
6
7 gboolean backend_create (JBackendItem* backend_item, gchar const*
    ↪ store, gchar const* collection, gchar const* item, gpointer
    ↪ data);
```

[9] Zettabyte File System
[10] Low-Level Extent-Based Object Store

```
 8 gboolean backend_delete (JBackendItem* backend_item, gpointer data);
 9
10 gboolean backend_open (JBackendItem* backend_item, gchar const*
      ↪ store, gchar const* collection, gchar const* item, gpointer
      ↪ data);
11 gboolean backend_close (JBackendItem* backend_item, gpointer data);
12
13 gboolean backend_status (JBackendItem* backend_item,
      ↪ JItemStatusFlags status_flags, gint64* modification_time,
      ↪ guint64* size, gpointer data);
14 gboolean backend_sync (JBackendItem* backend_item, gpointer data);
15
16 gboolean backend_read (JBackendItem* backend_item, gpointer buffer,
      ↪ guint64 length, guint64 offset, guint64* bytes_read, gpointer
      ↪ data);
17 gboolean backend_write (JBackendItem* backend_item, gconstpointer
      ↪ buffer, guint64 length, guint64 offset, guint64*
      ↪ bytes_written, gpointer data);
```

Listing 5.2: JULEA's storage backend interface

Listing 5.2 on the facing page shows the generic storage backend interface that all storage backends have to implement; it can be found in `daemon/backend/backend-internal.h`. The interface is simple by design and only provides support for the most essential operations: creating, deleting, opening and closing an item, getting an item's status, syncing an item's data to the underlying storage, and reading from and writing to an item (lines 7–17). Additionally, there are operations to initialize and finalize the storage backend both globally and per thread (lines 1–5).

All operations return error codes and can store additional state and information in the opaque `JBackendItem` structure. The per-thread initialization function can also return an opaque pointer that is passed to all file operations as their last parameter.

It is important to note that these functions are not called directly by clients; instead, everything is handled transparently by the data daemon that receives high-level operations and calls the appropriate low-level storage backend functions. Because of this, storage backends can rely on their functions being called in a specified sequence:

- `backend_init` (once)
  - `backend_thread_init` (once per thread)
    - `backend_create` or `backend_open` (once per operation)
    - `backend_status`, `backend_sync`, `backend_read` and `backend_write` (multiple times and in arbitrary order)

        * `backend_close` or `backend_delete` (once per operation)
    - `backend_thread_fini` (once per thread)
- `backend_fini` (once)

This guaranteed calling sequence makes it easy to build upon and use information from earlier function calls: The `status`, `sync`, `read`, `write`, `close` and `delete` functions can be sure that the item has been successfully opened or created before and do not have to handle different cases. Additionally, more elaborate functionality is relatively easy to implement: For instance, the POSIX storage backend uses both the global and per-thread initialization functions to implement a file descriptor cache in order to avoid opening the underlying files multiple times. The data daemon makes sure to pass the cache returned by the per-thread initialization function to all other functions.

```
1  G_MODULE_EXPORT
2  gboolean
3  backend_status (JBackendItem* bf, JItemStatusFlags flags, gint64*
       ↪ modification_time, guint64* size, gpointer data)
4  {
5      gint fd = GPOINTER_TO_INT(bf->user_data);
6      gint ret = -1;
7
8      (void)data;
9
10     j_trace_enter(G_STRFUNC);
11
12     if (fd != -1)
13     {
14         struct stat buf;
15
16         j_trace_file_begin(bf->path, J_TRACE_FILE_STATUS);
17         ret = fstat(fd, &buf);
18         j_trace_file_end(bf->path, J_TRACE_FILE_STATUS, 0, 0);
19
20         if (flags & J_ITEM_STATUS_MODIFICATION_TIME)
21         {
22             *modification_time = buf.st_mtime * G_USEC_PER_SEC;
23
24 #ifdef HAVE_STMTIM_TVNSEC
25             *modification_time += buf.st_mtim.tv_nsec / 1000;
26 #endif
27         }
```

```
28
29          if (flags & J_ITEM_STATUS_SIZE)
30          {
31              *size = buf.st_size;
32          }
33      }
34
35      j_trace_leave(G_STRFUNC);
36
37      return (ret == 0);
38  }
```

Listing 5.3: JULEA's POSIX storage backend

To demonstrate the usefulness of the storage backend interface, Listing 5.3 on the preceding page shows the `status` operation implemented by the POSIX storage backend as found in `daemon/backend/posix.c`. This function is a good example of how to use information returned by the underlying POSIX file system and fit it into JULEA's metadata concept. The file descriptor that was previously opened by the `open` operation has been stored in the `JBackendItem`'s `user_data` member and can be used by all other operations (line 5). The per-thread data returned by the `thread_init` function is ignored (line 8). Before the actual work starts, JULEA's tracing framework is used to trace the `status` function's invocation (line 10); similarly, the function's completion is traced after all work has been done (line 35). The POSIX storage backend uses the `fstat` function to obtain the underlying file's metadata (line 17); `fstat`'s invocation is traced in more detail (lines 16 and 18). The `status` operation supports specifying exactly which parts of the metadata should be returned using the `flags` parameter (lines 20–32). Because POSIX's `stat` functions always return all metadata, there is no significant advantage in this case; other backends could use this information to avoid performing unnecessary work, however. Finally, `fstat`'s return value is used to determine the `status` function's return value (line 37).

```
1  G_MODULE_EXPORT
2  gboolean
3  backend_write (JBackendItem* bf, gconstpointer buffer, guint64
       ↪ length, guint64 offset, guint64* bytes_written, gpointer data)
4  {
5      (void)buffer;
6      (void)data;
7
8      j_trace_enter(G_STRFUNC);
9
```

```
10      j_trace_file_begin(bf->path, J_TRACE_FILE_WRITE);
11      j_trace_file_end(bf->path, J_TRACE_FILE_WRITE, length, offset);
12
13      if (bytes_written != NULL)
14      {
15          *bytes_written = length;
16      }
17
18      j_trace_leave(G_STRFUNC);
19
20      return TRUE;
21  }
```

Listing 5.4: JULEA's NULL storage backend

The NULL storage backend is very useful for analyzing the performance of JULEA's general architecture because the I/O operations are not actually performed. However, all operations are still recorded using the tracing framework. Listing 5.4 on the preceding page shows the `write` operation implemented by the NULL storage backend as found in `daemon/backend/null.c`. This particular function nicely demonstrates how different use cases can be covered using the storage backend interface. All function call and I/O activity is traced (lines 8, 10–11 and 18) while all other data and information is discarded (lines 5–6). To maintain compatibility with existing applications, the caller is told that all data has been written to storage successfully (lines 13–16 and 20).

5.4. Client Library

The JULEA library allows applications to use the native JULEA interface to perform I/O. All other JULEA components such as the FUSE file system and command line utilities use this library to interact with the servers. Its source code can be found in the `lib` directory; all headers are located in the `include` directory.

A code example demonstrating the use of JULEA's client library can be found in Appendix C.3 on pages 198–200.

5.4.1. Data Distributions

To allow analyzing the influence of different data distributions on overall performance and to facilitate future research in this direction, JULEA contains a generic distribution interface that allows implementing different data distributions with a relatively low implementation overhead. JULEA already provides a number of different data distributions that can be found in the `lib/distribution` directory:

- **Round robin** (`lib/distribution/round-robin.c`): The round robin distribution divides the data into equally sized blocks and distributes them in a round-robin fashion across all data servers. The starting server is picked randomly to distribute the load evenly; developers can also specify the starting server manually, however.
- **Single server** (`lib/distribution/single-server.c`): The single server distribution stores all data on a single data server that is chosen randomly to distribute the load; as with the round robin distribution, the server can be specified manually by developers if necessary. Data is still divided into equally sized blocks to enable locking on a block level.
- **Weighted** (`lib/distribution/weighted.c`): The weighted distribution divides the data into equally sized blocks and applies user-specified per-server weights to determine how much data each data servers holds. The distribution always starts at the first data server but single servers can be excluded by setting their weight to `0`.

All data distribution functions use a default block size of 4 mebibytes (MiB) that can easily be changed using the `j_distribution_set_block_size` function.

```
1  struct JDistributionVTable
2  {
3      gpointer (*distribution_new) (guint server_count);
4      void (*distribution_free) (gpointer distribution);
5
6      void (*distribution_set) (gpointer distribution, gchar const*
           ↪ key, guint64 value);
7      void (*distribution_set2) (gpointer distribution, gchar const*
           ↪ key, guint64 value1, guint64 value2);
8
9      void (*distribution_serialize) (gpointer distribution, bson*
           ↪ bson_object);
10     void (*distribution_deserialize) (gpointer distribution, bson
           ↪ const* bson_object);
11
12     void (*distribution_reset) (gpointer distribution, guint64
           ↪ length, guint64 offset);
13     gboolean (*distribution_distribute) (gpointer distribution,
           ↪ guint* index, guint64* new_length, guint64* new_offset,
           ↪ guint64* block_id);
14 };
15
```

```
16 typedef struct JDistributionVTable JDistributionVTable;
```

Listing 5.5: Data distribution interface

Listing 5.5 on the previous page shows JULEA's distribution interface that can be found in `lib/distribution/distribution.h`. It provides functions to instantiate and free distribution objects (lines 1–2): The `distribution_new` function takes the number of data servers and returns a distribution object; the `distribution_free` function frees an existing distribution object.

The `distribution_set` and `distribution_set2` functions allow setting various distribution attributes such as the block size or the starting server. The only difference between the functions is the number of arguments they accept.

To store and restore the distribution information on JULEA's metadata servers, the `distribution_serialize` and `distribution_deserialize` functions serialize and deserialize the distribution information, respectively; the information is returned in the form of BSON[11] objects that can be stored directly in MongoDB.

The `distribution_reset` function allows initializing the distribution with a given offset and count. Finally, the `distribution_distribute` function actually calculates the data distribution by returning the data server's index, offset, count and a unique *block ID* that is used for locking.

```
1  static
2  gboolean
3  distribution_distribute (gpointer data, guint* index, guint64*
   ↪ new_length, guint64* new_offset, guint64* block_id)
4  {
5      JDistributionRoundRobin* distribution = data;
6
7      gboolean ret = TRUE;
8      guint64 block;
9      guint64 displacement;
10     guint64 round;
11
12     j_trace_enter(G_STRFUNC);
13
14     if (distribution->length == 0)
15     {
16         ret = FALSE;
17         goto end;
18     }
19
```

[11] Binary JavaScript Object Notation

```
20      block = distribution->offset / distribution->block_size;
21      round = block / distribution->server_count;
22      displacement = distribution->offset % distribution->block_size;
23
24      *index = (distribution->start_index + block) %
           ↪ distribution->server_count;
25      *new_length = MIN(distribution->length,
           ↪ distribution->block_size - displacement);
26      *new_offset = (round * distribution->block_size) + displacement;
27      *block_id = block;
28
29      distribution->length -= *new_length;
30      distribution->offset += *new_length;
31
32  end:
33      j_trace_leave(G_STRFUNC);
34
35      return ret;
36  }
```

Listing 5.6: Round robin distribution

To demonstrate how the data distribution interface enables easy prototyping of different data distribution functions, Listing 5.6 on the facing page shows the `distribution_distribute` function as found in `lib/distribution/round-robin.c`. As can be seen, the data distribution's execution is traced using JULEA's tracing framework (lines 12 and 33). The function splits up the item into equally sized blocks (lines 20–22). Afterwards, it determines which data server handles this particular block and calculates the data-server-local length and offset (lines 24–27). This process is repeated until no data is left to distribute (lines 14–18 and 29–30). The function returns `TRUE` if there is still data to distribute; otherwise, `FALSE` is returned (lines 7 and 35).

5.4.2. Metadata Serialization and Deserialization

MongoDB stores its documents in the so-called BSON format that has been designed to be lightweight and efficient. In contrast to JSON, BSON does not require string parsing, allowing it to be processed very quickly. BSON does not force specific schemas to be used and thus fits perfectly to MongoDB's schema-less design.

JULEA stores different metadata for each object depending on the current semantics. The schema-less design of BSON allows easy serialization and deserialization of JULEA metadata.

```
1  {
2      "_id" : ObjectId("51c999896035000000000014"),
3      "Collection" : ObjectId("51c99989603500000000000"),
4      "Name" : "test-19",
5      "Credentials" : {
6          "User" : 1000,
7          "Group" : 1000
8      },
9      "Distribution" : {
10         "Type" : 1,
11         "BlockSize" : NumberLong(4194304),
12         "StartIndex" : 0
13     }
14 }
```

Listing 5.7: JSON representation of an item's metadata using default semantics

Listing 5.7 shows the serialized metadata of an item created with the default semantics. Each collection and item is assigned a unique BSON `ObjectId` (lines 2–3) and name (line 4). Additionally, user and group credentials are stored to enable permission checking (lines 5–8). The item's data distribution is also stored with the metadata (lines 9–13); each data distribution may have different parameters, however. In this example, the data distribution's `Type` specifies that the round robin distribution is used; `BlockSize` is set to the default of 4 MiB and the `StartIndex` key indicates the data server that holds the first block of data.

```
1  {
2      "_id" : ObjectId("51caae667d1a000000000014"),
3      "Collection" : ObjectId("51caae667d1a000000000000"),
4      "Name" : "test-19",
5      "Status" : {
6          "Size" : NumberLong(0),
7          "ModificationTime" : NumberLong("1372237414990586")
8      },
9      "Credentials" : {
10         "User" : 1000,
11         "Group" : 1000
12     },
13     "Distribution" : {
14         "Type" : 2,
15         "BlockSize" : NumberLong(4194304),
16         "Index" : 0
```

```
17      }
18  }
```

Listing 5.8: JSON representation of an item's metadata using custom semantics

In contrast, Listing 5.8 on the facing page shows the serialized metadata of an item using different concurrency semantics and a different data distribution. As can be seen, the serial concurrency semantics caused the item's size and modification time to be stored on the metadata server as described in Section 3.5.2 on pages 71–73 (lines 5–8). In contrast to the previous example, the data distribution's `Type` shows that the single server distribution is used. This distribution also does not have a `StartIndex` key but rather an `Index` key that specifies the server all of the data is stored on.

As mentioned before, the item's metadata is serialized into BSON format by the data distribution's `serialize` function. Afterwards, the data distribution's `deserialize` function can construct an item out of the BSON data returned by MongoDB. The `Type` key is handled by JULEA's data distribution interface and is used to select the appropriate implementation for deserialization.

5.5. Miscellaneous

The JULEA framework contains a multitude of miscellaneous functionality that will be briefly described for completeness. This includes support for POSIX applications, tools for convenient use of the parallel distributed file system as well as tests to track functionality and performance regressions.

5.5.1. Tracing Framework

JULEA includes its own tracing framework to provide as much information as possible to developers and users; its implementation and headers can be found in `lib/jtrace.c` and `include/jtrace-internal.h`, respectively.[12] This can be used to visualize the inner workings in a graphical way and can be very helpful when debugging errors or searching for performance issues. It supports tracing of functions, file operations and counters with precise timestamps. While function tracing only shows when and which functions have been entered and left, file tracing also records information about the actual file operation and its result: The traces include the operation's type (for example, `open`, `close` or `delete`) as well as the number of accessed bytes and the file offset for `read` and `write` operations. Counters allow collecting statistics such as the total amount of accessed data or the number of created files. Additionally, the tracing framework is fully thread-safe and supports multiple backends:

[12] Even though the tracing framework is integrated into JULEA's client library, it does not have any JULEA-specific dependencies and can be easily built as a standalone library for external use.

- **Echo:** The `echo` backend simply outputs the trace information to the standard error output stream (`stderr`). This allows easy debugging without the need for complex graphical tools.
- **HDTrace:** The `hdtrace` backend is based on the HDTrace tracing library developed within the research group [MMK+12]. It features a file format based on XML[13] and a relatively simple interface that allows storing arbitrary parameters in the resulting XML file. HDTrace trace files can be visualized using the Sunshot visualization tool [LKK+07].
- **OTF:** The `otf` backend is based on the widely used OTF[14] tracing library [KBB+06]. It makes use of a portable ASCII[15] encoding and can merge multiple so-called *streams* into a single trace. Its interface is complex and does not easily allow storing arbitrary parameters. OTF trace files can be visualized using the Vampir visualization tool [GWT14].

```
1 $ J_TRACE=echo,hdtrace ./application
2 $ J_TRACE=echo J_TRACE_FUNCTION=j_batch*,j_distribution*
    ↪ ./application
```

Listing 5.9: JULEA tracing framework

Listing 5.9 shows an example of how to use JULEA's tracing framework. Its behavior can be modified using environment variables: The `J_TRACE` variable allows enabling one or more tracing backends at the same time by simply giving the appropriate values separated by commas; in this case, the `echo` and `hdtrace` backends are activated (line 1). Additionally, it is possible to filter the tracing framework's output in order to reduce the trace's size: The `J_TRACE_FUNCTION` variable can be used to only include the listed functions in the resulting trace; in this example, all functions pertaining to JULEA's batches (`j_batch*`) and data distributions (`j_distribution*`) are traced while all other function calls are discarded (line 2).

Figure 5.2 on the facing page shows exemplary traces of the client (top) and data daemon (bottom) activities that have been created using the OTF tracing backend and visualized using Vampir [GWT14]. The y-axis shows several so-called *timelines* containing the activities of separate threads. The timelines themselves are annotated with the performed functions; only long-lasting functions are shown by default, zooming in allows viewing shorter ones. In this example, the client is a benchmark application that first writes data and then reads it back; all I/O is performed using blocks of a size of 4 MiB. The benchmark uses 12 threads, each writing and reading 25

[13] Extensible Markup Language
[14] Open Trace Format
[15] American Standard Code for Information Interchange

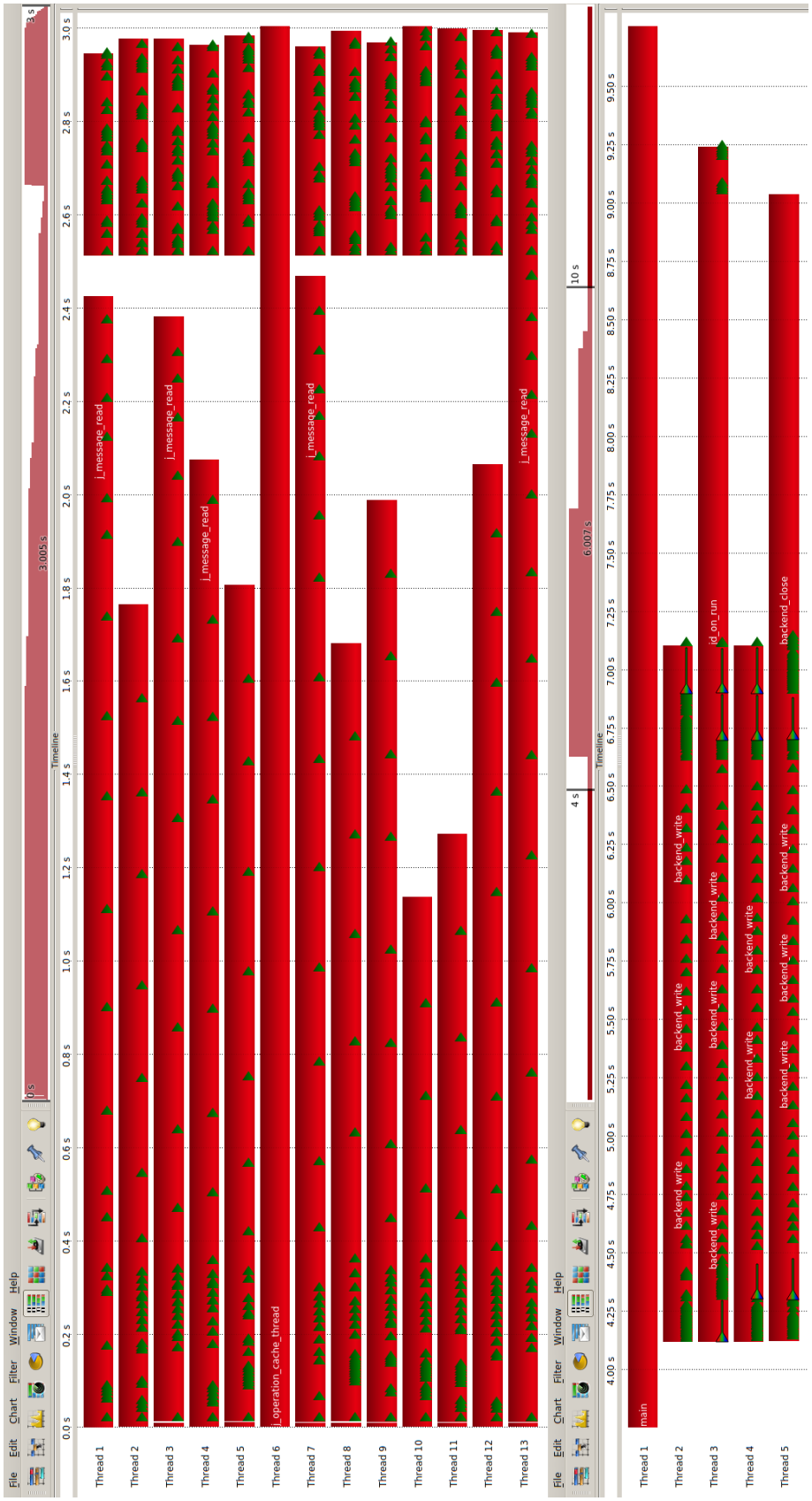


Figure 5.2.: Traces of the client and data daemon's activities

blocks; consequently, each thread accesses a total of 200 MiB comprising 100 MiB of written and 100 MiB of read data. The client library is configured to use a maximum of four connections to connect to the data daemon; consequently, the data daemon uses four threads to service the client's requests.

The top of Figure 5.2 on the previous page shows the client's trace that contains 13 threads in total. This is due to the fact that JULEA starts an internal thread for background operations that is used when the persistency semantics are modified. Thread 6 simply waits inside the `j_operation_cache_thread` function all the time because no background operations are taking place. All threads synchronize between the write and read phase; this can be seen at the 2.5 s time mark when the last thread finishes writing and all threads begin reading. All threads take another 0.5 s for reading and are finished after a total runtime of 3 s.

The bottom shows the data daemon's trace with five threads in total: four threads to service client requests (threads 2–5) and the data daemon's idle main thread (thread 1). As can be seen, not all threads finish at the same time: While threads 2 and 4 finish after 3 s when the client finishes reading, threads 3 and 5 take around 2 s longer to delete the files. This slowdown is likely due to the underlying file system.

5.5.2. POSIX Compatibility Layer

Looking at the number of different I/O interfaces in existence today, it is unrealistic to expect all existing applications to be ported to new I/O interfaces. For proprietary software that does not offer source code access and other special cases it might even be impossible to do so. Therefore, to keep compatibility with existing and widely used software, a POSIX compatibility layer is provided.

There are several possibilities to implement such a compatibility layer. For instance, the environment variable `LD_PRELOAD` instructs the dynamic linker to *preload* a specified library before all other shared libraries. This allows overwriting existing functions such as `open`, `close`, `read` and `write`. Using this mechanism, it would be possible to provide wrappers for the POSIX I/O functions that use the JULEA interface to perform the actual I/O. However, there are several problems regarding this approach: One has to be very careful when overwriting low-level I/O functions using the `LD_PRELOAD` approach because it not only wraps the function calls within the actual application but also all calls within other libraries and low-level functions.

Therefore, another approach has been used to realize JULEA's POSIX compatibility layer. It has been accomplished using the FUSE framework and can be found in the `fuse` directory. The FUSE framework provides a stable and easy-to-use interface to implement POSIX-compliant file systems in user space. It consists of a user space library, a kernel module and some auxiliary command line utilities. FUSE file systems run as ordinary applications in user space that are linked against the `libfuse.so` library. This library communicates with the FUSE kernel module that, in turn, relays

I/O accesses done via the VFS to the user space file system. While this allows conveniently implementing POSIX-compliant file systems in user space, the additional indirection of I/O accesses has impacts on I/O performance [RG10, IMOT12]. Even though there have been recent improvements to FUSE, kernel file systems still offer higher performance in many cases [Duw14]. However, as the compatibility layer's main objective is to provide backwards compatibility instead of high performance, this is not an obstacle in this case. An important advantage of this approach is that FUSE file systems can be used by ordinary non-root users.

```
1 $ mkdir /tmp/julea-fuse
2 $ julea-fuse /tmp/julea-fuse
3 $ ls -l /tmp/julea-fuse
4 $ fusermount -u /tmp/julea-fuse
5 $ rmdir /tmp/julea-fuse
```

Listing 5.10: FUSE file system

Listing 5.10 shows how to use JULEA's POSIX compatibility layer. First, the FUSE file system's mount point is created (line 1). All FUSE file systems require an existing directory within the normal file system namespace to be used as a mount point. Afterwards, the actual FUSE file system – which is called `julea-fuse` – is mounted on top of the given directory (line 2). As soon as the FUSE file system is mounted, all accesses within the mount point `/tmp/julea-fuse` will be handled by JULEA's FUSE file system and can be accessed by POSIX-compliant clients (line 3). When POSIX compatibility is no longer required, the FUSE file system has to be unmounted (line 4). This is accomplished using the `fusermount` command that is part of the FUSE software package. As the last step, the mount point is cleaned up (line 5).

5.5.3. Command Line Tools

Data management on supercomputers is typically performed using the command line. When using JULEA, it is impossible to use existing command line tools such as `cp`, `mv`, `stat` or even `cat` because these tools only support the POSIX interface. While it would be possible to use them on top of JULEA's POSIX compatibility layer, native command line tools are preferable for performance and reliability reasons. Therefore, special command line tools are provided to allow easy data management outside of full-blown applications. They have support for all basic operations such as creating, deleting, listing and getting the status of stores, collections and items. Additionally, items can be copied between collections.

```
1 $ julea-cli create-all julea://foo/bar/baz
2 $ julea-cli list julea://foo/bar
3 $ julea-cli status julea://foo/bar/baz
4 $ julea-cli delete julea://foo/bar/baz
```

Listing 5.11: JULEA command line tools

Listing 5.11 shows how JULEA's command line tools can be used: All functionality is available through the `julea-cli` application that supports several different commands. First, the `create-all` command is used to create the `foo` store, `bar` collection and `baz` item (line 1); in contrast to the `create` command that only creates the last path component, all missing path components are created when using `create-all`. Afterwards, it is possible to use the `list` command to list the contents of stores and collections; in this case, the `bar` collection's items are listed (line 2). The `status` command returns all available metadata for collections and items; in this case, it lists the credentials, modification time and size of the newly created `baz` item (line 3). Finally, the `delete` command is used to delete the `baz` item (line 4); the `bar` collection and `foo` store are not deleted.

5.5.4. Correctness and Performance Tests

JULEA includes a wide range of tests and benchmarks that are used to periodically check its correctness and performance. Because providing efficient access to data is one of a file system's main features, it is not only necessary to provide unit and regression tests for correctness but also for performance.

In addition to the possibility to execute these checks manually, JULEA includes functionality to trigger them automatically whenever a code change occurs. These automatisms have been realized using so-called *hooks* provided by the Git version control system (VCS) that is used for JULEA development [Fuc13]. The hooks allow performing fast correctness tests before each commit and more elaborate performance tests after each commit. All performance results are kept in a separate Git repository and linked to the commit that has been used to produce them. This can be used to effectively analyze JULEA's performance history and assess the influence of specific changes as it allows correlating performance changes with individual commits.

Figure 5.3 on the facing page demonstrates how JULEA's performance history can be visualized over time. It has been generated by running a benchmark application for a selected range of commits in JULEA's Git repository; the x-axis contains the times and IDs for all examined commits. Individual commits can be analyzed in more detail using the `git show -p` command by specifying the commit ID as its argument. Several observations can be made using the available performance data:

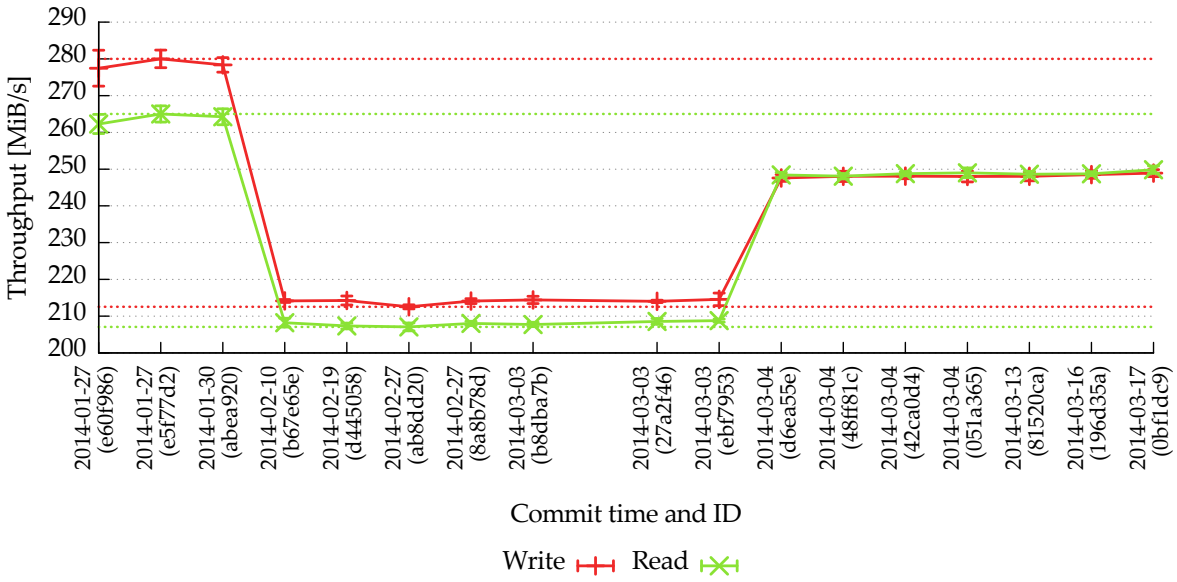


Figure 5.3.: Performance history over time

1. Commit `b67e65e` on 2014-02-10 has significantly reduced both read and write performance. Examining the commit reveals that a bug was fixed that caused JULEA to open more TCP connections than the allowed maximum. This bug lead to higher performance in this limited local benchmark; too many TCP connections can be detrimental to performance in large-scale scenarios, however.

2. Performance data is missing between commits `b8dba7b` and `27a2f46` on 2014-03-03. Consulting the raw data reveals that commit `77d5df8` on 2014-03-03 introduced a bug that crashed the benchmark and thus did not deliver performance data.[16]

3. Commit `d6ea55e` on 2014-03-04 introduced the use of TCP corking as described in Section 4.4 on pages 80–81. As can be seen, merging multiple TCP packets provides clear performance benefits in this case.

Making this kind of fine-grained information available can help analyzing performance problems during the file system's development. The above example has only used a single benchmark but the automatic hooks perform a multitude of benchmarks pertaining to different areas of the file system. Users upgrading from one version to another could use this information to find the reason for changes in performance regarding specific access patterns.

[16] It is necessary to look at the raw data because gnuplot does not draw the x-axis entry for missing data.

Summary

This chapter has presented an in-depth description of JULEA's technical design. The main points have been the general architecture as well as detailed specifications for the client library, the data servers and the metadata servers. Additionally, JULEA's built-in tracing framework, POSIX compatibility layer, command line tools, and framework for automated correctness and performance tests have been explained. JULEA has been implemented completely in user space to make use of the comprehensive tool support for both analysis and debugging purposes.

Chapter 6.

Performance Evaluation

In this chapter, the efficiency of the new I/O interface with dynamically adaptable semantics will be evaluated using synthetic benchmarks as well as real-world applications. While the synthetic benchmarks will be used to analyze the specific optimizations made possible by the file system's additional knowledge about the applications' I/O requirements, the real-world applications will be used to demonstrate the applicability for existing software.

Benchmarks will be used to evaluate different performance aspects of JULEA and other selected parallel distributed file systems. Specifically, data and metadata performance will be evaluated independently. Lustre and OrangeFS have been selected as representative parallel distributed file systems: While the former strives to support POSIX[1] semantics, the latter is optimized for non-overlapping writes.

In addition to comparing JULEA to the other parallel distributed file systems, a number of different semantics will be evaluated. However, due to the sheer amount of different semantics combinations, only those expected to have a significant impact on performance will be analyzed in more detail. JULEA's data performance will be evaluated using different atomicity, concurrency and safety semantics; its metadata performance will be benchmarked using different concurrency and safety semantics. Additionally, the usefulness of batches will be analyzed.

6.1. Hardware and Software Environment

All evaluations have been conducted on the cluster of the Scientific Computing research group at the University of Hamburg. The benchmarks have been performed using a total of 20 nodes, with 10 nodes running the file system clients and 10 nodes hosting the file system servers. The nodes' hardware and software setup is as follows:

The client nodes each have two Intel Xeon Westmere EP HC X5650 central processing units (CPUs) (2.66 GHz, 12 cores total), 12 gigabytes (GB) DDR3/PC1333 error-correcting code (ECC) random access memory (RAM), a 250 GB SATA2 Seagate Barracuda 7200.12 hard disk drive (HDD) and two Intel 82574L gigabit (Gbit) Ethernet

[1] Portable Operating System Interface

network interface cards (NICs). They run Ubuntu 12.04.3 LTS with Linux 3.8.0-33-generic and Lustre 2.5.0 (client); the MPI[2] implementation is provided by Open-MPI 1.6.5.

The server nodes each have one Intel Xeon Sandy Bridge E-1275 CPUs (3.4 GHz, 4 cores total), 16 GB DDR3/PC1333 ECC RAM, three 2 terabytes (TB) SATA2 Western Digital WD20EARS HDDs, one 160 GB SATA2 Intel 320 solid state drive (SSD) and two Intel 82579LM/82574L Gbit Ethernet NICs. They run CentOS 6.5 with Linux 2.6.32-358.18.1.el6_lustre.x86_64 and Lustre 2.5.0 (server).

6.1.1. Performance Considerations

To allow a proper assessment of the results, the following theoretical performance considerations should be kept in mind.

- Even though all client and server nodes are equipped with two NICs each, only one of them is used. OpenMPI transparently uses all found NICs whenever possible; however, since only insignificant amounts of data are transmitted via MPI in the following measurements, this is negligible.
- The theoretical maximum performance of Gbit Ethernet is 125 megabytes (MB)/s.[3] However, it is usually not possible to reach more than 117 MB/s due to overhead. Consequently, the maximum achievable performance between the clients and servers is approximately 1,170 MB/s.[4]
- SATA2 has a transfer rate of 3 Gbit/s which translates to 300 MB/s due to 8b/10b encoding.[5] Consequently, storage devices are able to deliver a maximum throughput of 300 MB/s. Because this is much higher than the maximum network transfer rate, this limitation can be ignored for the measurements.
- While the HDDs' maximum throughput is 117 MiB/s for both reading and writing, the SSDs deliver up to 251 MiB/s when reading and 164 MiB/s when writing. Because these numbers are higher than the network throughput, they can also be ignored when determining the maximum performance.
- The average round-trip time (RTT) between the client and server nodes is 0.228 ms.[6] Ignoring actual processing times, it is therefore possible to send and receive 4,386 requests/s.

[2] Message Passing Interface

[3] 8 bits = 1 byte, consequently 1 Gbit = 1,000 megabits (Mbits) = 125 MB.

[4] 1,170 MB/s correspond to 1,115 mebibytes (MiB)/s which is the unit that will be used in the following measurements.

[5] A 8b/10b encoding requires 10 bits to transfer 8 bits of information and is commonly used for communication technologies.

[6] The average RTT has been sampled using the `ping` command with at least 100 packets; the standard deviation was 0.019 ms.

6.2. Data Performance

The file systems' data performance will be evaluated using a large number of concurrently accessing clients that first write data and then read it back again; the write and read phases are completely separated and barriers ensure that only one type of operation takes place at any given time. The benchmark uses MPI to start multiple processes accessing the file systems in a coordinated fashion. There are two basic modes of operation:

1. **Individual files:** Each process only accesses its own file or item.[7] Even though all processes access the file system concurrently, the individual files are accessed serially because only one process has exclusive access to it.

2. **Shared file:** All processes access a single shared file. Consequently, the shared file will be accessed concurrently.

All accesses use a variable block size and are non-overlapping, that is, no write conflicts occur. The following block sizes have been used for the following evaluation: 4 kibibytes (KiB), 16 KiB, 64 KiB, 256 KiB and 1,024 KiB. The processes repeatedly read or write data using the block size until each process has accessed 2 gibibytes (GiB) per phase; the number of iterations is denoted by `m`. This allows evaluating the file systems' behavior with many small accesses as well as fewer large ones.

Process	0	0	...	0
Iteration	0	1	...	m
⋮				
Process	n	n	...	n
Iteration	0	1	...	m

Figure 6.1.: Access pattern using individual files

Process	0	1	...	n	...	0	1	...	n
Iteration	0	0	...	0	...	m	m	...	m

Figure 6.2.: Access pattern using a single shared file

Figures 6.1 and 6.2 show the access patterns when using individual files and a shared file, respectively. Each rectangle represents one file and each column inside a rectangle denotes one data block. For each data block, its accessing process and iteration are given. The areas of the file that are accessed concurrently are enclosed in

[7] For readability reasons, the rest of the chapter will only mention files when either files or items are considered.

double lines. When using individual files, each process possesses its own file that it accesses exclusively, as can be seen in Figure 6.1 on the preceding page. All accesses are done sequentially from the start of the file to its end. For the shared file, however, the accesses of all processes happen in an interleaved fashion, as can be seen in Figure 6.2 on the previous page.

To evaluate the file systems' behavior with different numbers of accessing clients, the following `n/p` configurations (where `n` stands for the number of client nodes and `p` stands for the total number of client processes) have been used: 1/1, 1/2, 1/4, 1/8, 1/12, 2/24, 3/36, 4/48, 5/60, 6/72, 7/84, 8/96, 9/108 and 10/120. These numbers have been chosen because each of the client nodes has 12 cores, therefore, real applications would strive to use all of them to reach optimal performance. All parallel distributed file systems have been set up to provide ten data servers and one metadata server.

The benchmark supports several input/output (I/O) interfaces to allow comparing different parallel distributed file systems using their respective interfaces. Currently, POSIX, MPI-IO and JULEA are available.

Each benchmark has been repeated at least three times to calculate the arithmetic mean as well as the standard deviation. To force the clients to read the data from the data servers during the read phase, the clients' cache was dropped after the write phase.[8] The servers' caches were dropped by completely restarting and remounting the file systems after each configuration; the server caches were not touched between the write and read phases, however. This represents a realistic use case because it is common for applications to write out results that are afterwards post-processed by different applications that do not have access to the cached contents. The servers, however, try their best to keep requested data in their caches.

This benchmark represents a very simple and common I/O pattern because all data is accessed sequentially, that is, lower offsets are accessed before higher ones. Consequently, reading can be sped up using readahead and data can be written in a streaming fashion without the need for random I/O.

6.2.1. Lustre

For the following measurements, Lustre has been set up using its default options except for the stripe count that has been set to `-1` to enable striping over all available object storage targets (OSTs); the stripe size has been set to 1 MiB. While each OST has been provided by one of the servers' HDDs, the meta data target (MDT) has been provided by one of the SSDs.

[8] For Lustre, the `/proc/sys/vm/drop_caches` file was used; for OrangeFS and JULEA, nothing was done because both file systems do not cache data on the clients by default.

POSIX

Lustre has been mounted using the client module as a normal POSIX file system with the `flock` option that enables support for file locking. The option should not have any influence on the benchmark results because they do not use file locking.

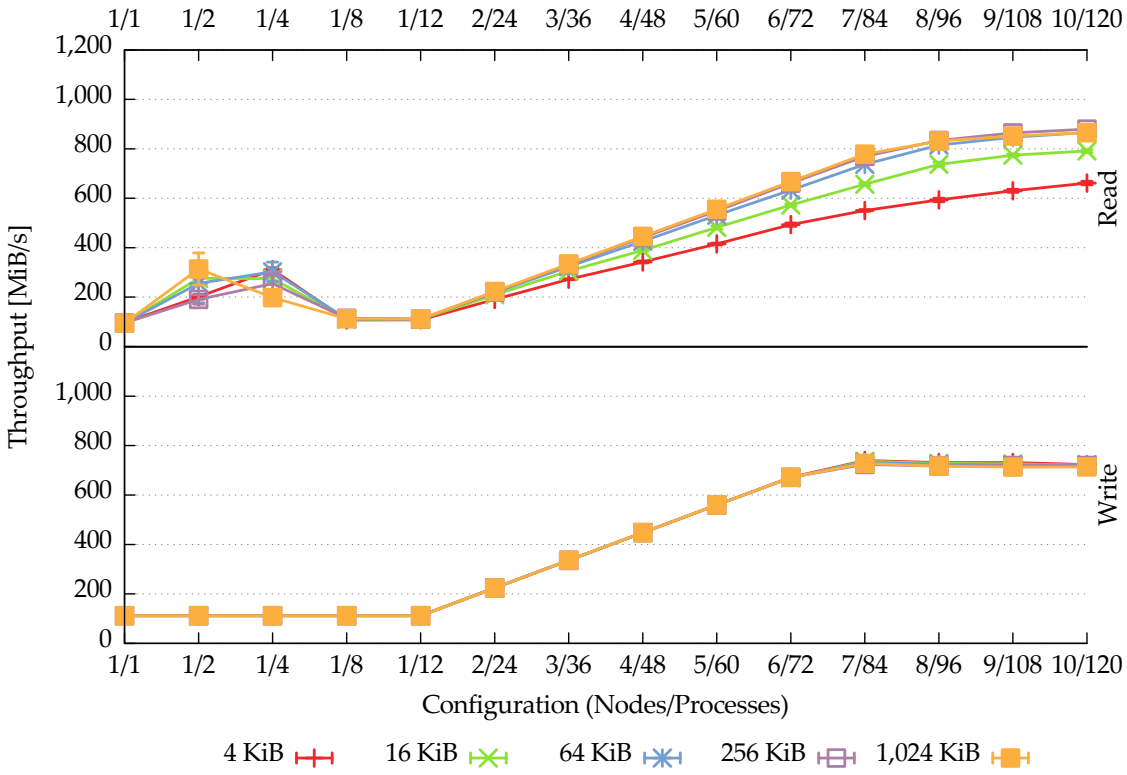


Figure 6.3.: Lustre: concurrent accesses to individual files via the POSIX interface

Individual Files Figure 6.3 shows Lustre's read and write performance when using individual files via the POSIX interface.

Regarding read performance, it is interesting to note that configurations with a single node exhibit different performance characteristics depending on the number of processes. While the configurations with one, eight and twelve processes all achieve a throughput of roughly 100 MiB/s, the configurations with two and four processes deliver 200-300 MiB/s; while this effect has to be related to some data being read from the cache of the operating system (OS), the exact reasons for this are unclear. As explained earlier, the benchmark drops all caches between the read and write phases, therefore, this effect should not occur. The remaining configurations gradually deliver more performance as more nodes are added until reaching their maximum performance with ten nodes; the block sizes of 64 KiB, 256 KiB and 1,024 KiB all achieve a maximum of roughly 850 MiB/s. As expected, smaller block sizes result in lower read

performance due to additional overhead. However, it is interesting to note that even with a single process and a block size of 4 KiB, Lustre achieves a read performance of roughly 100 MiB/s. As mentioned in Section 6.1.1 on pages 112–113, the Gbit Ethernet network can transfer at most 4,386 requests/s. Taking this into account, Lustre should only be able to read at a maximum of 17 MiB/s. This discrepancy is due to Lustre performing client-side readahead to increase performance.

When considering write performance, it can be seen that all block sizes deliver the same performance. This is most probably due to Lustre's use of client-side write caching. Because individual files are used and each file is only accessed by one node, Lustre can utilize caching without sacrificing POSIX compliance. Using the POSIX I/O interface and semantics, each access theoretically needs one network round trip to send the actual data to the data server and return its reply. Consequently, accesses can not be pipelined because the write operations block until the reply has been received. Lustre seems to use a different approach in this case that can be demonstrated using the following theoretical performance estimation: When assuming a maximum of 4,386 requests/s and using a block size of 4 KiB, this results in a maximum throughput of roughly 17 MiB/s per process. While the configurations using twelve client processes per node can overlap multiple write operations to achieve higher performance, the configuration using one node and one process should not be able to deliver more than the previously mentioned 17 MiB/s. Due to this and the fact that Lustre manages to deliver the same performance regardless of the chosen block size, it can be concluded that it does not actually send each request to the data servers and instead collects data in the local cache to aggregate accesses. This also implies that the number of bytes that has been written – as returned by the `write` function – does not originate from the data server but instead from the local cache.

Shared File Figure 6.4 on the next page shows Lustre's read and write performance when using a single shared file via the POSIX interface.

The read performance for the configurations using one node behaves in a similar way to the test case with individual files. When using more than two nodes, however, the results are distinctly different: For block sizes of 4 KiB, 16 KiB and 64 KiB not all results could be collected because Lustre's performance was too low and the jobs exceeded the job scheduler's time limit. For 256 KiB and 1,024 KiB, the performance increases until six and seven nodes, respectively. Afterwards, performance drops with each additional node. This result is surprising because only read operations are performed by all accessing clients, that is, no locking should be required. However, it appears that Lustre still introduces some overhead for these accesses, decreasing overall performance significantly.

For the write phase, an interesting effect occurs: While using only a single node, performance is stable for all block sizes. As soon as the number of accessing nodes is larger than one, performance drops for all block sizes less than 1,024 KiB. This is likely

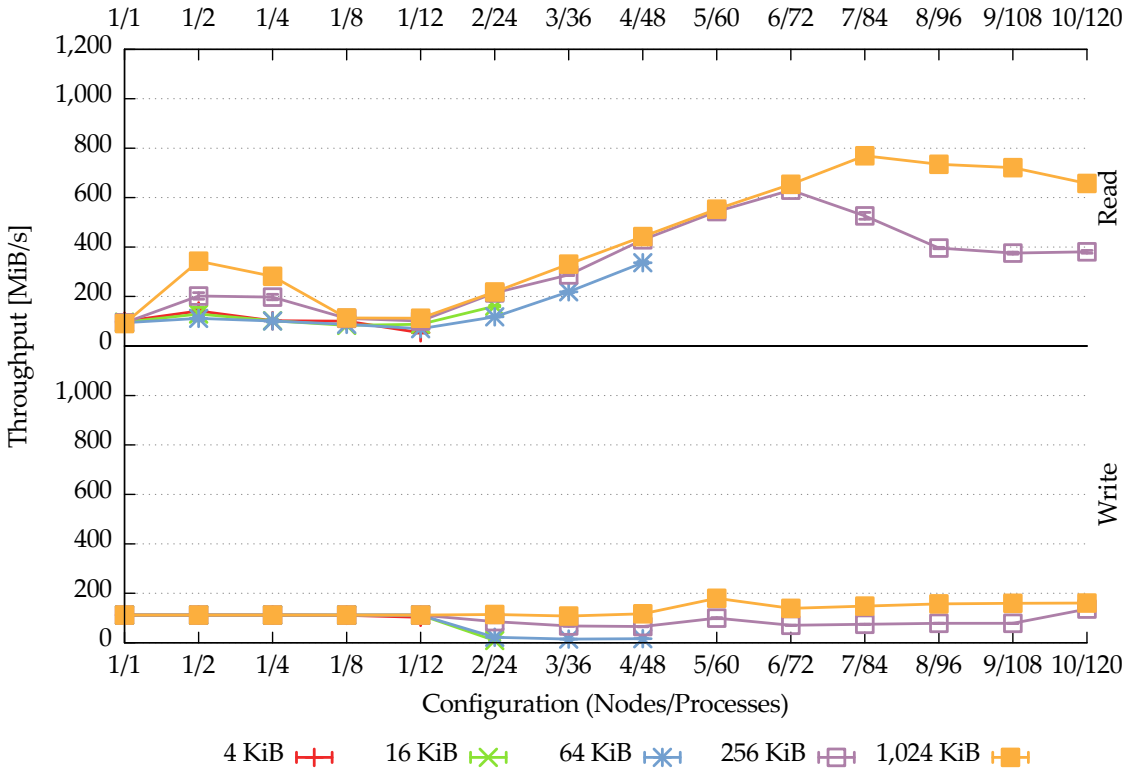


Figure 6.4.: Lustre: concurrent accesses to a shared file via the POSIX interface

due to the effect described in Section 4.2 on pages 77–78: As soon as multiple nodes are involved, Lustre has to send all write operations directly to the data server to achieve POSIX compliance. Using the same estimation as before, a block size of 1,024 KiB and 4,386 requests/s results in a theoretical maximum of 4.3 GiB/s which is in stark contrast to the actual maximum of 180 MiB/s when using five nodes. Consequently, additional factors have to be responsible for Lustre's low performance in this case. One of them could be write locking that needs to be performed due to the concurrently accessing clients.

MPI-IO (Atomic Mode)

The following results demonstrate Lustre's performance when accessed using the MPI-IO interface. Because it was not possible to compile ADIO[9]'s native Lustre backend, MPI-IO falls back to its generic POSIX backend. Because the results for both individual and shared files using non-atomic accesses are largely identical to their POSIX counterparts, they have been omitted.

[9] Abstract-Device Interface for I/O

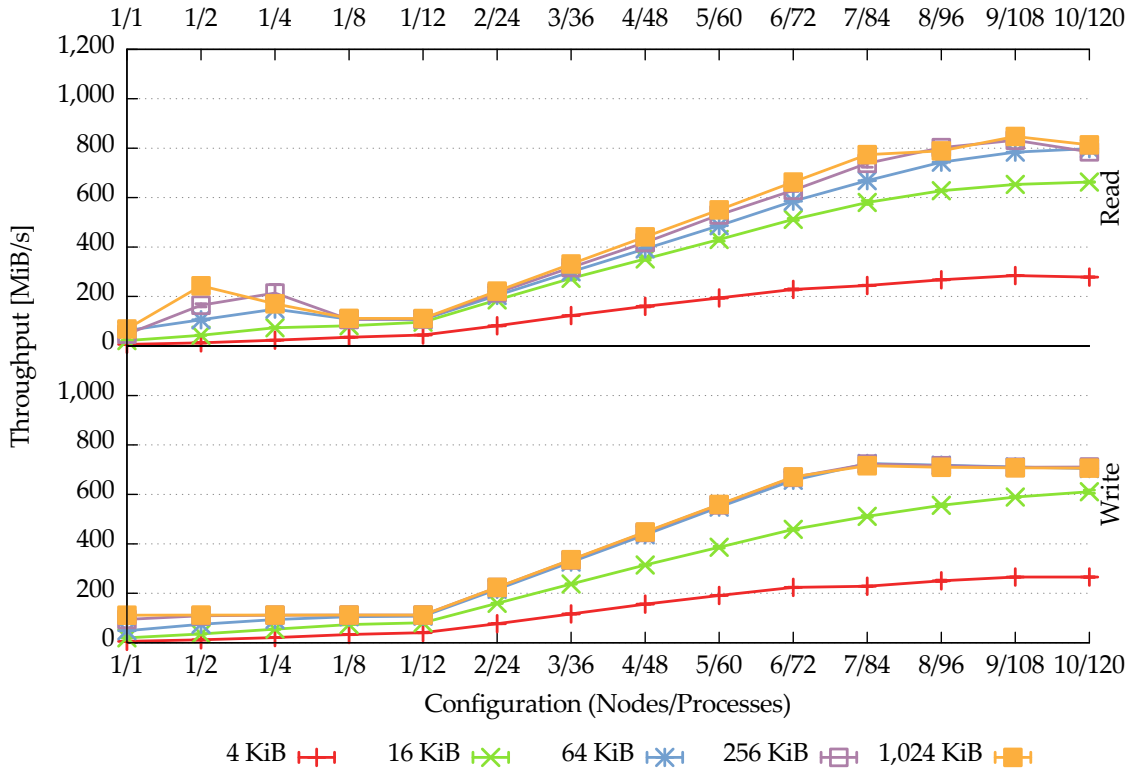


Figure 6.5.: Lustre: concurrent atomic accesses to individual files via the MPI-IO interface

Individual Files Figure 6.5 shows Lustre's read and write performance when using individual files via the MPI-IO interface with atomic mode.

Regarding read performance, the results are largely identical to the POSIX results for the larger block sizes of 1,024 KiB to 64 KiB. For block sizes of 16 KiB and 4 KiB, there are significant drops in performance: Using ten nodes, it decreases from 800 MiB/s to 700 MiB/s and 700 MiB/s to 300 MiB/s, respectively. Consequently, the overhead introduced by atomic mode can likely be neglected for block sizes equal to or larger than 64 KiB. One noteworthy exception is the performance when using ten nodes, which is slightly lower than the one with nine nodes for almost all block sizes. Since measurements have only been performed with a maximum of ten nodes, it is not possible to determine whether performance would continue to drop when using more than ten nodes or if this effect is limited to this specific configuration.

Considering write performance, the results look similar to the ones using the POSIX interface: Performance is identical for the block sizes from 1,024 KiB to 64 KiB and flattens out when using seven nodes or more; as in the read phase, performance actually decreases slightly when using more nodes. For the block sizes of 16 KiB and 4 KiB, performance drops significantly due to the introduced overhead.

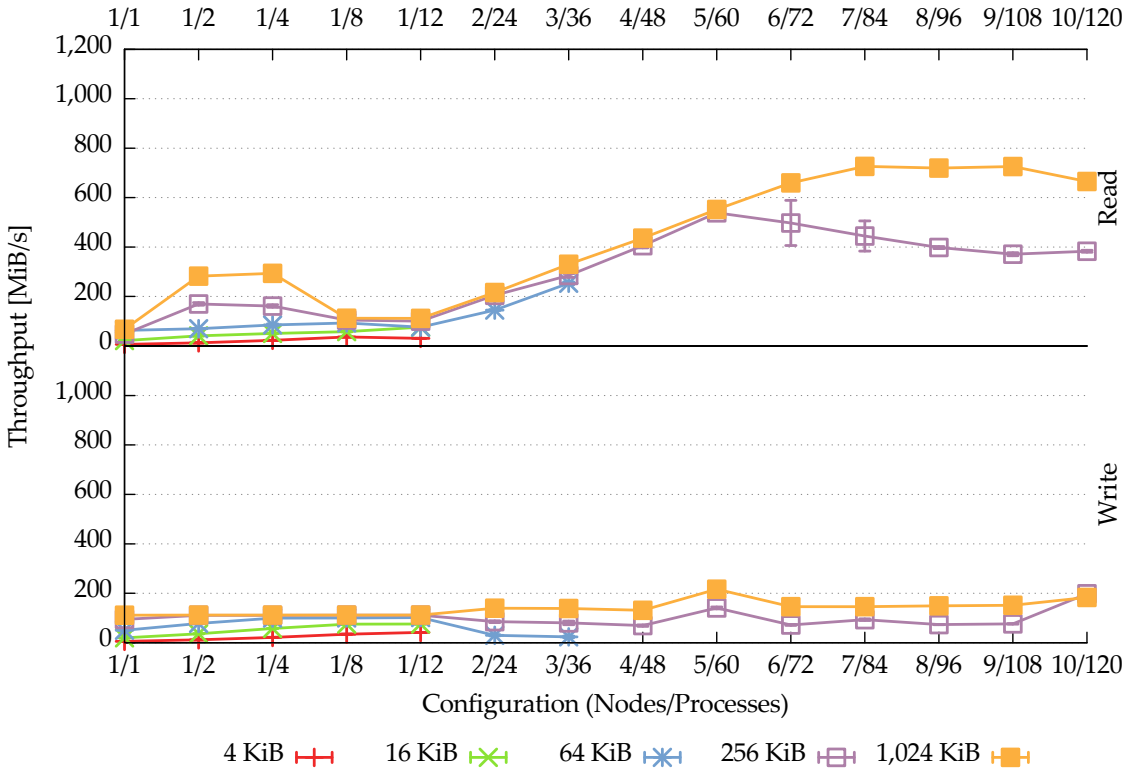


Figure 6.6.: Lustre: concurrent atomic accesses to a shared file via the MPI-IO interface

Shared File Figure 6.6 illustrates Lustre's read and write performance when using a single shared file via the MPI-IO interface with atomic mode.

For both the read and write phases, performance is almost identical to that of their POSIX counterparts. Since performance was already poor for the POSIX case due to the overhead introduced by the shared file accesses, the additional overhead caused by atomic mode does not decrease performance further. One noteworthy exception is the read performance using a block size of 256 KiB: While performance in the POSIX case was identical to the one using a block size of 1,024 KiB until six nodes were used and then decreased, the overhead produced by MPI-IO's atomic mode causes performance to already drop when using six nodes.

6.2.2. OrangeFS

OrangeFS has been set up using its default configuration. Its storage space for both data and metadata has been provided by an ext4 file system located on the data servers' system HDDs. Placing the metadata on an HDD should not have negatively influenced performance because the number of metadata operations can be neglected for the data benchmark.

MPI-IO

All benchmarks have been performed using the MPI-IO interface and ADIO's native OrangeFS backend; since the backend does not support atomic mode, only non-atomic results are provided.

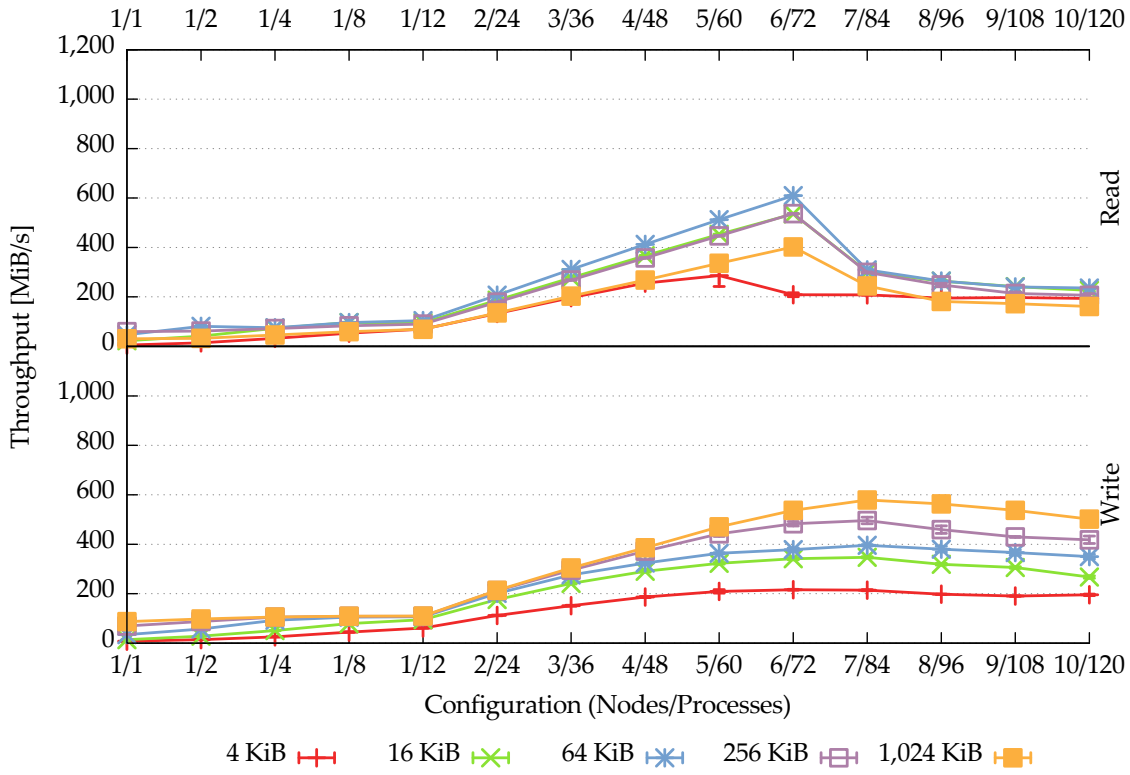


Figure 6.7.: OrangeFS: concurrent accesses to individual files via the MPI-IO interface

Individual Files Figure 6.7 displays OrangeFS's read and write performance when using individual files via the MPI-IO interface.

When considering read performance, it can be observed that larger block sizes do not necessarily result in higher performance as it was the case with Lustre. Instead, performance increases until a block size of 64 KiB is reached and then drops again for larger ones; a block size of 1,024 KiB performs worse than 256 KiB. This is likely due to the fact that OrangeFS's default stripe size is 64 KiB; it is unclear why larger block sizes are handled in such a suboptimal way, however. Apart from this inconsistency, performance increases steadily up to 600 MiB/s until six nodes are used; as soon as more nodes are used, performance drops to 200–300 MiB/s. As will be explained in more detail later, this is due to the underlying POSIX file system.

Regarding write performance, larger block sizes result in higher overall performance as expected. That is, the performance inconsistency caused by the striping seems to be

limited to read operations. Performance improves to a maximum of 600 MiB/s with seven nodes and decreases slowly as more nodes are added. Again, this is due to the underlying POSIX file system.

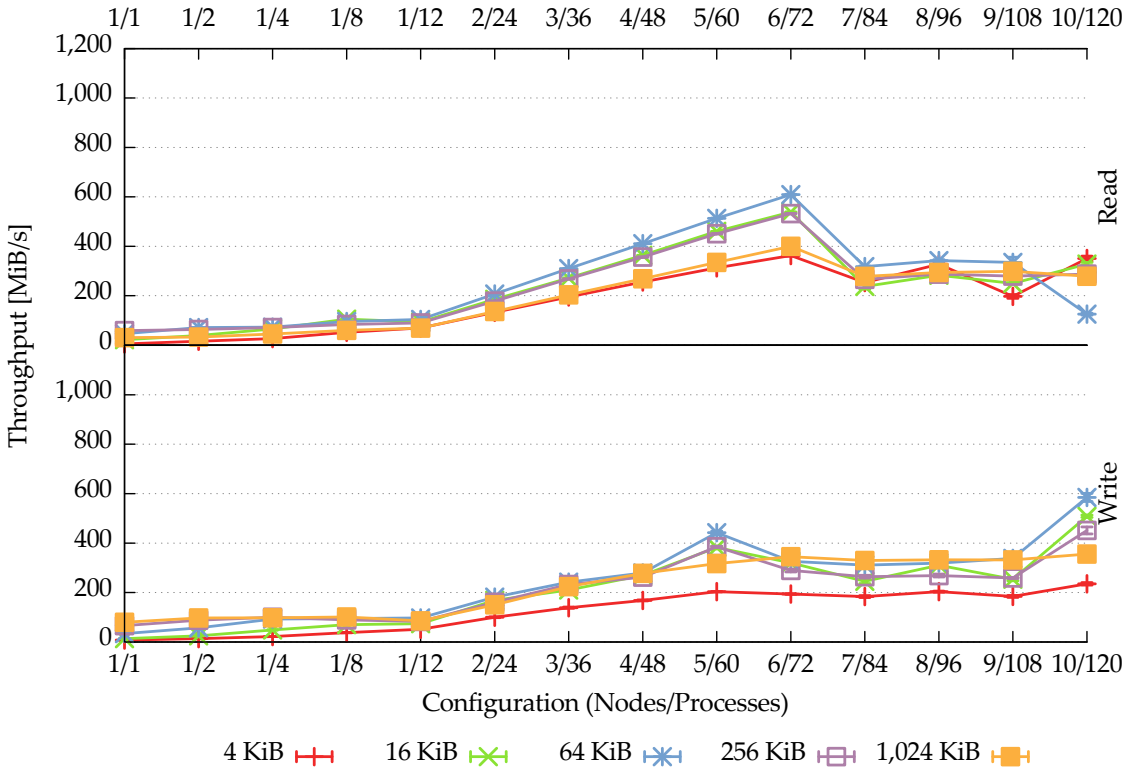


Figure 6.8.: OrangeFS: concurrent accesses to a shared file via the MPI-IO interface

Shared File Figure 6.8 shows OrangeFS's read and write performance when using a single shared file via the MPI-IO interface.

During the read phase, performance looks largely similar to that of the individual case except for configurations using eight or more nodes. While the performance curve flattened out when using this amount of nodes in the individual case, the shared case shows much more erratic performance behavior for all but the largest block size of 1,024 KiB, which remains relatively stable. For instance, when using a block size of 4 KiB, performance increases when going from seven to eight nodes, then decreases for nine nodes and finally increases again for ten nodes. However, overall performance using small block sizes is better than in the individual case: While the block size of 4 KiB achieved a maximum of roughly 280 MiB/s with the configuration using five nodes for individual files, it manages a maximum of 360 MiB/s with six nodes when using a shared file. This abnormal behavior is likely due to scheduling problems inside the underlying file system and will be explained in more detail later.

During the write phase, the performance curve again looks erratic except for the largest block size of 1,024 KiB. For example, for all but the smallest and the largest block sizes, performance abruptly increases for the configuration using five nodes and then decreases again for more nodes; when going from nine to ten nodes, it increases sharply again. Even though the largest block size of 1,024 KiB manages to deliver stable performance for all configurations, its performance is significantly lower than when using individual files: Instead of reaching roughly 600 MiB/s, performance is reduced to a maximum of 350 MiB/s when using a shared file; this corresponds to a performance drop of more than 40 %. Again, the unpredictable performance behavior is likely due to the underlying file system and will be analyzed later.

6.2.3. JULEA

JULEA has been configured to use the data daemon's POSIX storage backend due to the experimental nature of the object store storage backends. Both the storage backend as well as MongoDB stored their data within an ext4 file system located on the data servers' system HDDs. Analogous to OrangeFS, placing the metadata on an HDD should not have influenced performance negatively for the given benchmark. Additionally, JULEA was set to use a maximum of six client connections per node because it was observed that the default of twelve caused severe performance problems due to the large amount of TCP[10] connections.[11]

Default Semantics

The following measurements have been performed using JULEA's default semantics to establish a performance baseline. The default semantics provide support for non-overlapping parallel accesses and do not cache data; for a detailed explanation, see Section 3.4.9 on pages 68–70. Missing values are due to the benchmarks exceeding the job scheduler's time limit.

Individual Items Figure 6.9 on the facing page shows JULEA's read and write performance when using individual items via the native JULEA interface.

Regarding read performance, it is interesting to note that JULEA's performance figure looks very similar to the OrangeFS counterpart. In contrast to OrangeFS, however, performance increases with growing block sizes. Using the block size of 1,024 KiB, performance increases until a maximum of approximately 700 MiB/s is reached with six nodes. Afterwards, performance drops drastically to about 250 MiB/s and continues to decrease as more nodes are used. This is likely due to an inefficiency

[10] Transmission Control Protocol

[11] The default value for the maximum number of connections is determined based on the number of cores present in the system.

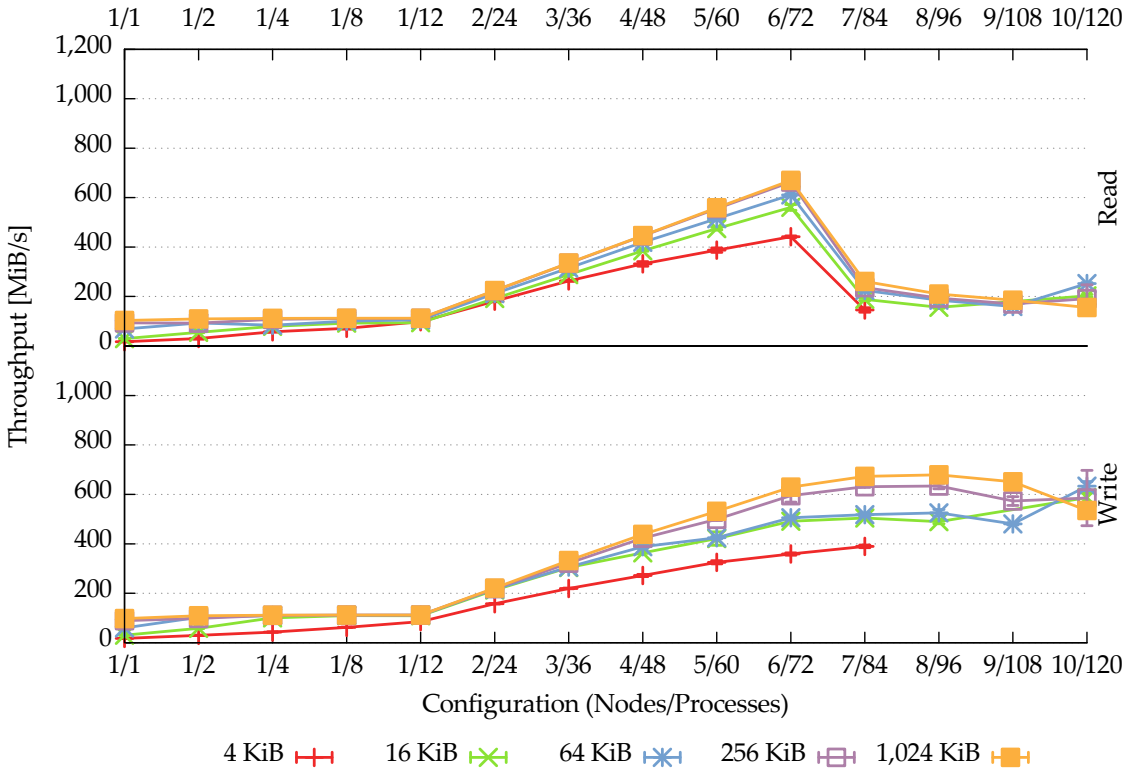


Figure 6.9.: JULEA: concurrent accesses to individual items

inside the Linux kernel that is exposed when using a large amount of parallel I/O streams and will be analyzed in more detail later.

Regarding write performance, JULEA's performance figure again looks similar to the OrangeFS counterpart except for a higher overall performance. While OrangeFS reaches a maximum performance of roughly 600 MiB/s using a block size of 1,024 KiB, JULEA manages to achieve 700 MiB/s. The performance with a block size of 4 KiB is especially noteworthy because JULEA's maximum of roughly 400 MiB/s is almost double that of OrangeFS's 200 MiB/s. Performance begins to decrease as soon as more than eight nodes are used, regardless of the block size. This is due to too many parallel I/O streams that can not be handled efficiently anymore.

Shared Item Figure 6.10 on the next page shows JULEA's read and write performance when using a single shared item via the native JULEA interface.

During the read phase, the performance curve looks almost identical to its counterpart using individual items up to six nodes. Even though the same performance drop is present when using more than six nodes, its extent is less severe: Instead of dropping from roughly 700 MiB/s to 250 MiB/s, JULEA still manages to deliver 350 MiB/s when using a shared item. Additionally, performance improves slightly again when more than eight or nine nodes are used, depending on the block size.

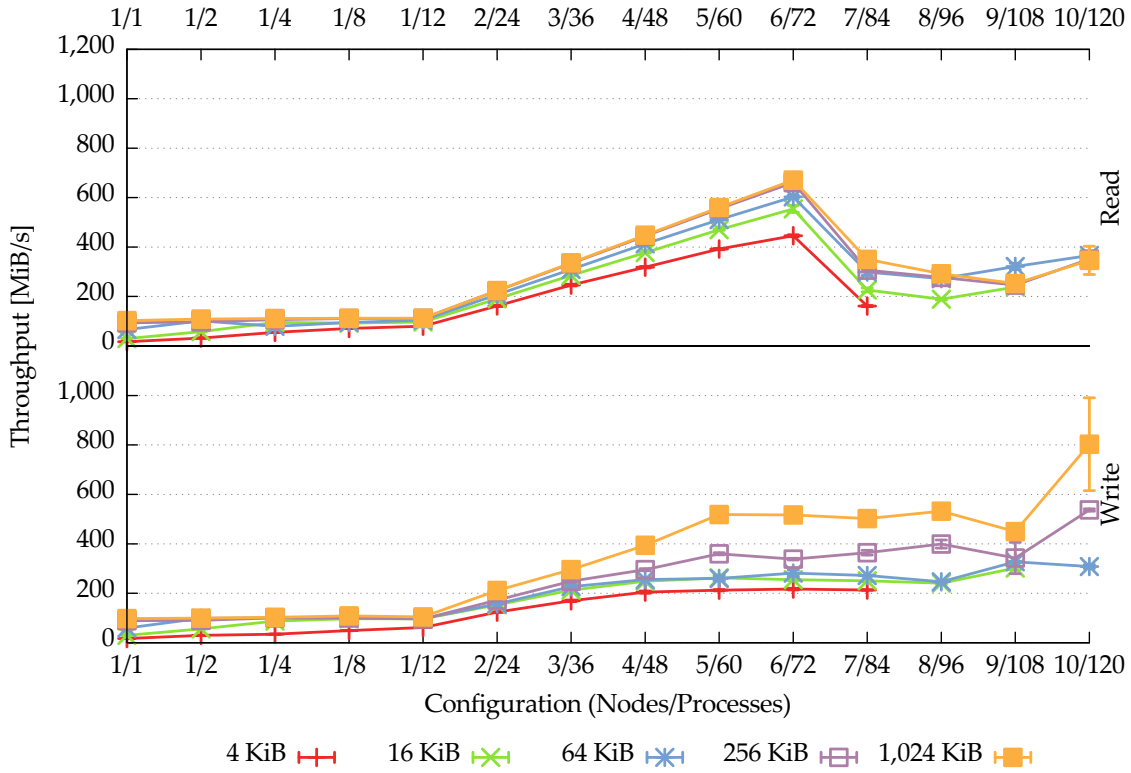


Figure 6.10.: JULEA: concurrent accesses to a shared item

During the write phase, the performance is even more irregular than when reading. While performance increases until five nodes are used, it fluctuates when more nodes are used. Whereas performance remains relatively constant for five to eight nodes for the block sizes of 256 KiB and 1,024 KiB, there is a performance drop when using nine nodes, followed by a significant performance increase for ten nodes. This effect is similar to the one found when using OrangeFS. Overall, performance is lower than when using individual items, especially for the smaller block sizes. While the block size of 4 KiB reached a maximum of 400 MiB/s using individual items, it only achieves slightly more than 200 MiB/s when using a shared item; this corresponds to a performance drop of roughly 50 %. When comparing the results to their counterparts using individual items, the erratic behavior can only be explained by an inefficient handling of shared files by the Linux kernel.

To analyze the performance problems further, additional measurements have been performed using varying numbers of connections per client, a different underlying POSIX file system and the NULL storage backend. Measurements using two and six connections per clients have shown that these problems are present regardless of the number of connections; the results using two connections will not be presented because they are almost identical to those when using six connections. Additionally, XFS has been used for comparison purposes using three and six connections per client;

these measurements can be found in Appendix A.1 on pages 181–184 and show that the performance problems are independent of the underlying file system. To check whether the problems lies within JULEA's implementation, measurements using the NULL storage backend will be presented in the following section.

NULL Storage Backend

The NULL storage backend allows analyzing JULEA's architecture for performance bottlenecks by excluding the influence of the underlying POSIX file system or object store. JULEA's behavior is not changed in any way except for the storage backend not actually accessing a storage device.

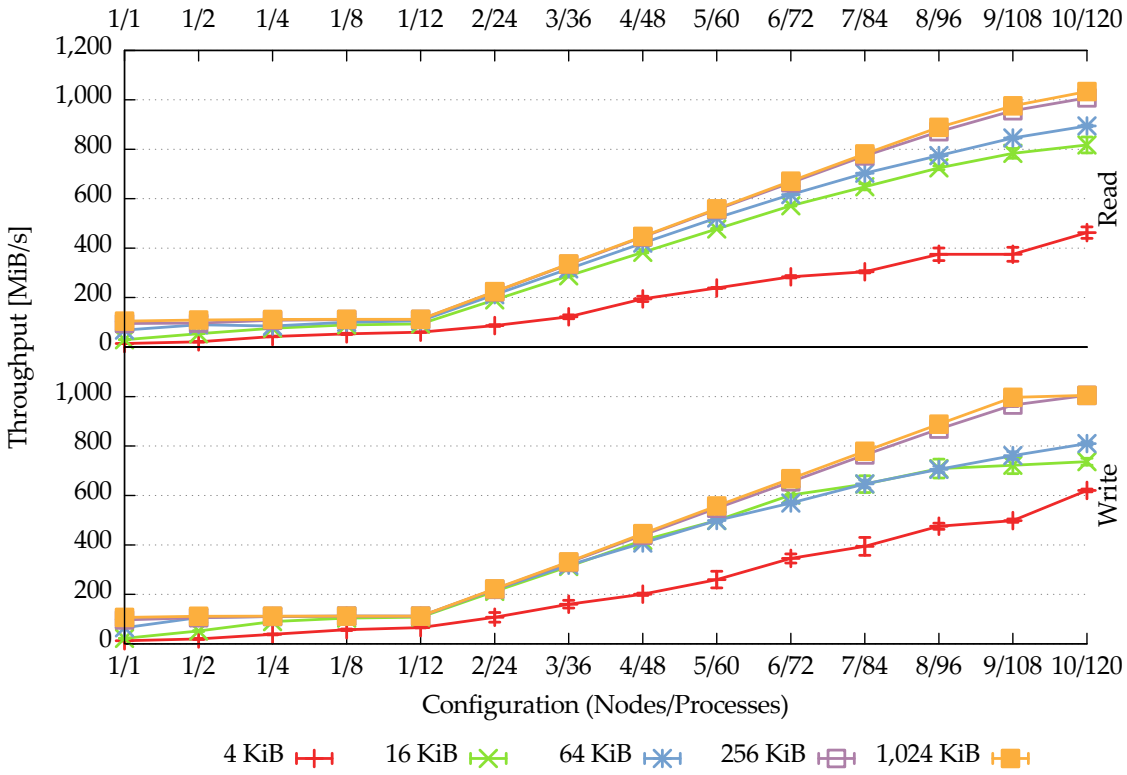


Figure 6.11.: JULEA: concurrent accesses to individual items using the NULL storage backend

Individual Items Figure 6.11 shows JULEA's read and write performance when using individual items via the native JULEA interface using the NULL storage backend.

During the read phase, performance is improved by larger block sizes, with 256 KiB and 1,024 KiB providing almost identical performance. Both block sizes almost reach the maximum possible performance and end up with 1,000 MiB/s when using ten nodes; the speedup decreases slightly when going from nine to ten nodes.

During the write phase, performance is very similar to the read phase; however, performance for the smallest block size of 4 KiB is higher while performance for block sizes of 16 KiB and 64 KiB is lower. Again, block sizes of 256 KiB and 1,024 KiB achieve almost the same throughput and are also close to the maximum possible performance; the speedup decreases significantly when going from nine to ten nodes.

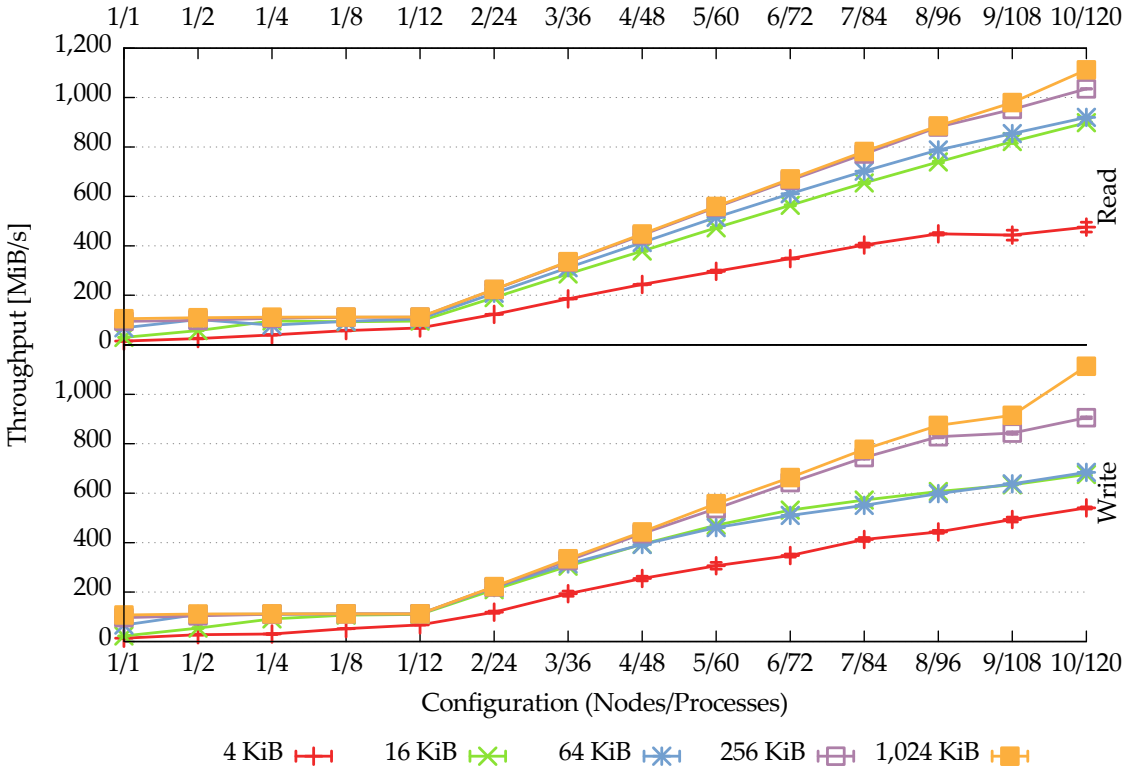


Figure 6.12.: JULEA: concurrent accesses to a shared item using the NULL storage backend

Shared Item Figure 6.12 shows JULEA's read and write performance when using a single shared item via the native JULEA interface using the NULL storage backend.

During the read phase, performance is improved slightly across the board when compared with its individual counterpart. The only difference between the individual and shared cases is the number of accessed items which leads to a different communication scheme between the clients and data servers:

- **Individual:** Because the data distribution's starting server is chosen randomly, communication happens with all data servers at once in a random fashion as soon as enough clients start accessing them. Therefore, each client node will likely communicate with all data servers at once.

- **Shared:** Because all clients share the same item and thus the same starting server, communication is more uniform. Due to JULEA's default stripe size of 4 MiB, consecutive clients are likely to communicate with the same data server. Consequently, each client node will likely communicate only with a small number of data servers at once. For example, when using a block size of 4 KiB, all clients only have to communicate with one or – less likely – two data servers in each iteration because only 480 KiB are accessed per iteration. Using a block size of 1,024 KiB, all twelve clients on a single node only have to communicate with at most four data servers.

Even though the clients are not synchronized for each iteration, this communication pattern improves overall performance and eliminates the speedup's slowdown that was present in the individual case when using nine and ten nodes.

During the write phase, the same effect has a negative influence on overall performance, especially for block sizes of 16 KiB and 64 KiB.

In conclusion, the following observations can be made about the underlying performance problems found using OrangeFS and JULEA:

1. The inefficiency is independent of the number of files because the same behavior occurs regardless of whether individual items or a shared item are used. Using the POSIX storage backend, each item results in one file being created on each data server that holds data of this item.

2. The number of open file descriptors seems to be irrelevant as the POSIX storage backend only keeps one open file descriptor for each individual file to avoid running out of file descriptors.[12] Consequently, only one file descriptor is used in the shared case.

3. The problem is also independent of the number of I/O threads as it also occurs with two, three and six connections per client; this number directly translates to two, three and six I/O threads per client node within the data servers.

4. The underlying file system has no effect on this problem as it occurs with at least XFS and ext4. This makes it likely that it is a fundamental problem inside the Linux kernel and not a problem restricted to one specific file system.

Additional specialized analyses are necessary to be able to pinpoint the exact reason for this performance anomaly.

[12] This is necessary because the number of open file descriptors is usually limited to 1,024 per process. It is possible for users to raise this soft limit to the hard limit of 4,096 using the `ulimit` command.

Default Semantics (Reduced Number of Clients)

The only way to mitigate the performance problem found when using both OrangeFS and JULEA as well as a large number of concurrently accessing clients is to reduce the number of clients. Consequently, to make sure that the results are not influenced by this underlying performance problem and to be able to demonstrate JULEA's different semantics, the remaining performance measurements have been performed with a reduced number of clients.

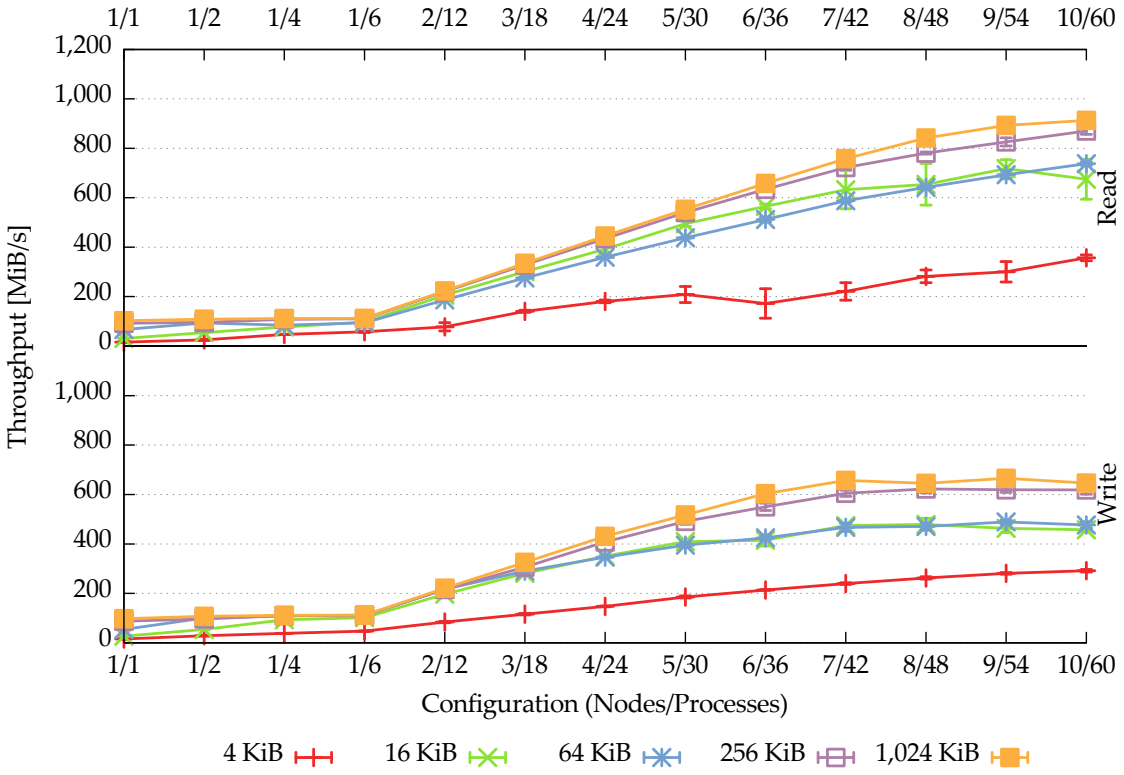


Figure 6.13.: JULEA: concurrent accesses to individual items

Individual Items Figure 6.13 shows JULEA's read and write performance when using individual items via the native JULEA interface.

Regarding read performance, it can be seen that the scaling is much improved when compared to the measurements using twelve clients per node. Instead of the steep performance drop when using more than six nodes, they provide almost linear scaling until seven to eight nodes are used. Afterwards, the speedup slows down, reaching a maximum of more than 900 MiB/s using a block size of 1,024 KiB. As expected, smaller block sizes provide a lower overall performance with the exception of 16 KiB and 64 KiB that are reversed. It is also interesting to note that the block size of 4 KiB is

the only one to suffer from the reduction of clients; its performance is roughly halved when compared to twelve clients.

Regarding write performance, the same effects as in the read case can be observed. While the reduced number of clients per node provides more stable performance results, it does not actually improve performance in this case. However, it is noteworthy that even though the performance does not increase with more than seven clients, it remains at a stable level in contrast to its counterpart using twelve clients.

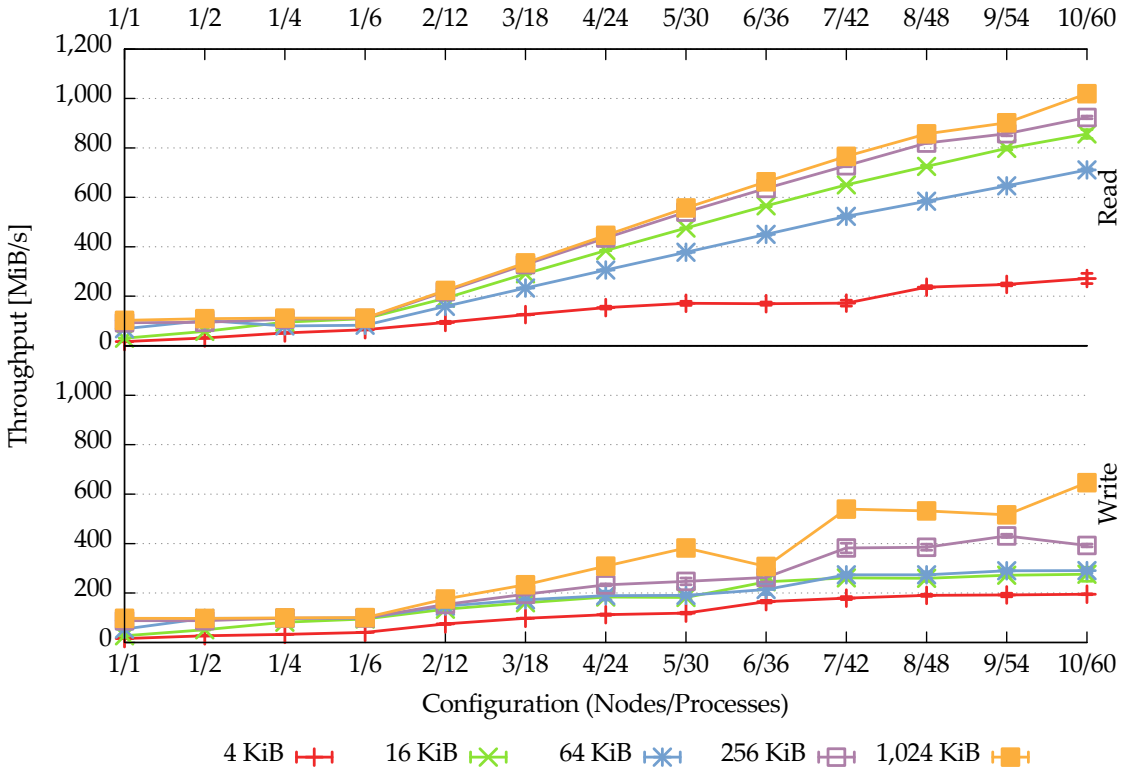


Figure 6.14.: JULEA: concurrent accesses to a shared item

Shared Item Figure 6.14 shows JULEA's read and write performance when using individual items via the native JULEA interface.

During the read phase, the performance curve looks almost identical to its counterpart using individual items when using large block sizes and less than ten nodes. While the performance speedup slowed slightly when going from nine to ten nodes using individual items, the shared item case is not affected by this drop and reaches a maximum of more than 1,000 MiB/s. Additionally, the block size of 16 KiB provides a more stable performance curve. It is interesting to note that the block size of 16 KiB consistently provides better performance than the block size of 64 KiB; the reason for this is unclear and has to be looked into further.

During the write phase, the performance curve looks less smooth than when using individual items. For instance, using the largest block size of 1,024 KiB, performance drops when increasing the number of nodes from five to six, only to rise again when using seven nodes. Overall, performance is more stable than when using twelve clients per node, however. The fact that overall performance is lower than when using individual items and roughly on the same level as when using twelve clients indicates that the handling of shared files is suboptimal in the Linux kernel. As demonstrated using the NULL storage backend, these performance inconsistencies only occur if the underlying file system is actually accessed using shared files.

To reduce the number of results and exclude the influences of the performance inconsistencies when using a single shared file, the following measurements have only been performed using individual items.

Batch Operations

The following measurements have been performed using JULEA's batch support. To limit the batch size to a reasonable amount, at most 1,000 operations have been grouped together into a batch.

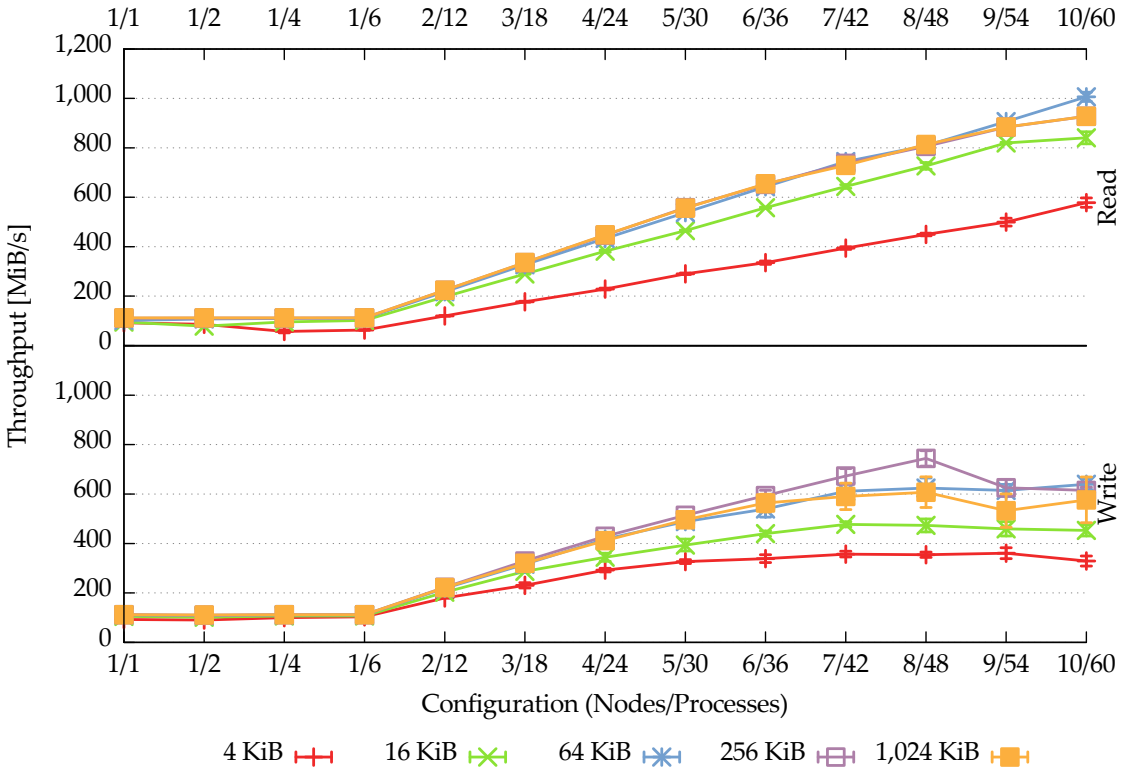


Figure 6.15.: JULEA: concurrent batch accesses to individual items

Individual Items Figure 6.15 on the facing page shows JULEA's read and write performance when using individual items via the native JULEA interface.

Regarding read performance, it can be seen that batches provide improved performance especially for smaller block sizes: While the block size of 4 KiB achieved a maximum of 350 MiB/s using individual operations, batches boost this number to almost 600 MiB/s; this corresponds to an increase of 65 %. For the larger block sizes, this effect is not as pronounced but it is interesting to note that the block sizes of 64 KiB, 256 KiB and 1,024 KiB all reach the same performance. The only exception occurs when using nine or ten nodes, where the speedup for the two largest block sizes starts to slow down. The block size of 64 KiB continues scaling and reaches a maximum of 1,000 MiB/s, however. The results can be explained as follows, based on the used block size:

- **16 KiB:** A batch of 1,000 operations bundles read operations of 16,000 KiB, that is, 15.63 MiB. Due to the default stripe size of 4 MiB, each client contacts four servers. This does not introduce enough parallelism to reach maximum performance.

- **64 KiB:** The batch reaches a size of 64,000 KiB, that is, 62.5 MiB. This implies that each client reads data from all ten data servers in parallel.

- **256 KiB and 1,024 KiB:** The batches are sized 250 MiB and 1,000 MiB, respectively. These huge batches reduce performance because they exclusively lock the connections for too long.

Consequently, the results indicate that it might prove beneficial to limit the size of batches internally to improve parallelism.

Regarding write performance, it can be observed that batches provide significant performance boosts especially for small amounts of client processes: A single process already reaches a performance of more than 90 MiB/s even for the smallest block size of 4 KiB. Overall, batches deliver a mixed picture regarding their impact on performance. On the one hand, they reduce performance for the largest block size of 1,024 KiB: While individual operations achieved roughly 650 MiB/s, batches only deliver 550 MiB/s. On the other hand, batches deliver significant improvements for the smallest block size of 4 KiB: Individual operations delivered a maximum of roughly 290 MiB/s, while batches manage to achieve more than 350 MiB/s. Additionally, the performance maximum is reached using a smaller amount of nodes. Overall, there is still room for improvements. Even though the data server handles batches more efficiently by merging multiple write operations, the clients do not perform such optimizations yet. Batching 1,000 operations with even a small block size of 4 KiB should be able to deliver at least the same performance as individual operations using a block size of 1,024 KiB.

Safety Semantics

The following measurements have used the safety semantics to disable write acknowledgments for all write operations. A detailed description of the safety semantics can be found in Section 3.4.6 on page 65.

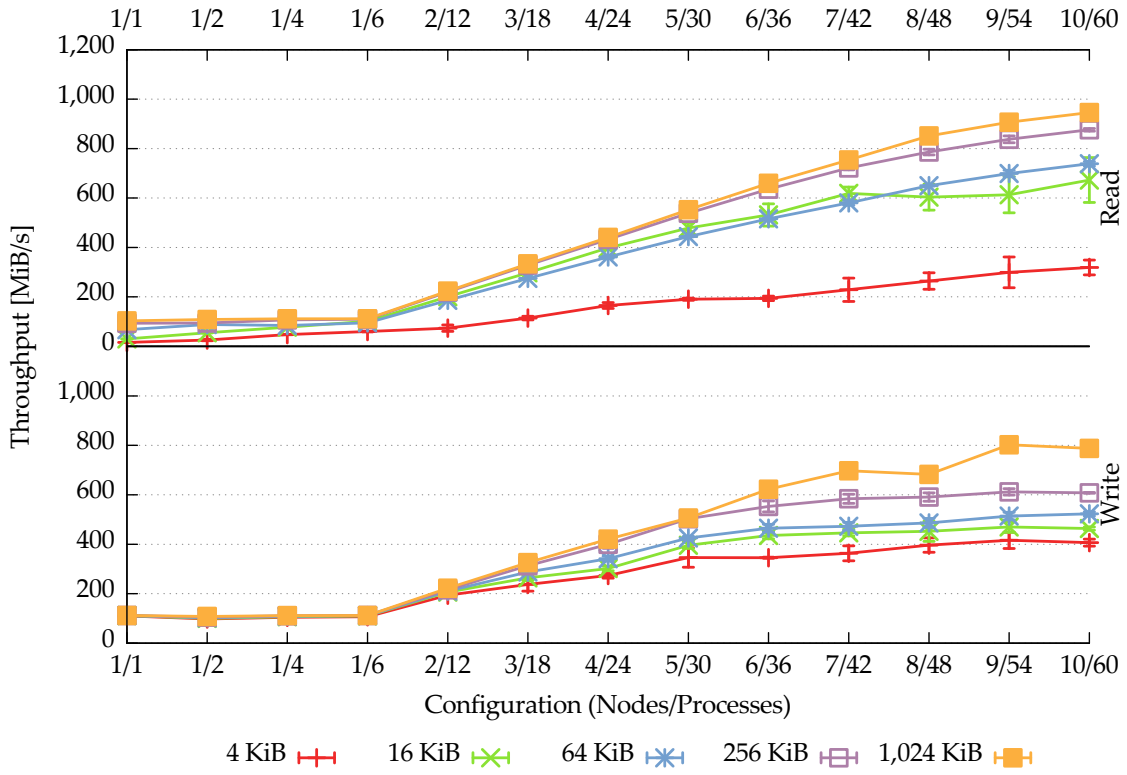


Figure 6.16.: JULEA: concurrent accesses to individual items using unsafe safety semantics

Individual Items Figure 6.16 shows JULEA's read and write performance when using individual items via the native JULEA interface.

During the read phase, there are only minor differences in performance in comparison to the default semantics (see Section 6.2.3 on pages 128–130). This is to be expected because the read operations are not handled differently depending on the safety semantics.

During the write phase, performance is improved across the board for all block sizes. It is especially interesting to note that even a single process achieves the maximum performance of 110 MiB/s using a block size of 4 KiB because the clients do not have to wait for the write acknowledgments from the data servers. Using a block size of 4 KiB, the maximum performance is increased from less than 300 MiB/s to roughly 400 MiB/s when using ten nodes; this corresponds to an improvement of 33 %.

The largest block size of 1,024 KiB manages to achieve a maximum performance of approximately 800 MiB/s, an improvement of 23 % when compared to the maximum of 650 MiB/s delivered by the default semantics.

Atomicity Semantics

The following measurements have used the atomicity semantics to enforce atomic access for each read and write operation. For a detailed explanation of the atomicity semantics, see Section 3.4.1 on pages 61–62.

JULEA currently implements atomicity using a centralized locking algorithm. As explained in Section 5.4.1 on pages 98–101, JULEA's data distributions split up items into blocks of equal size. Locking is then performed on a per-block basis by inserting and removing documents from a MongoDB collection. Each lock operation requires one insert operation and each unlock operation needs one remove operation.

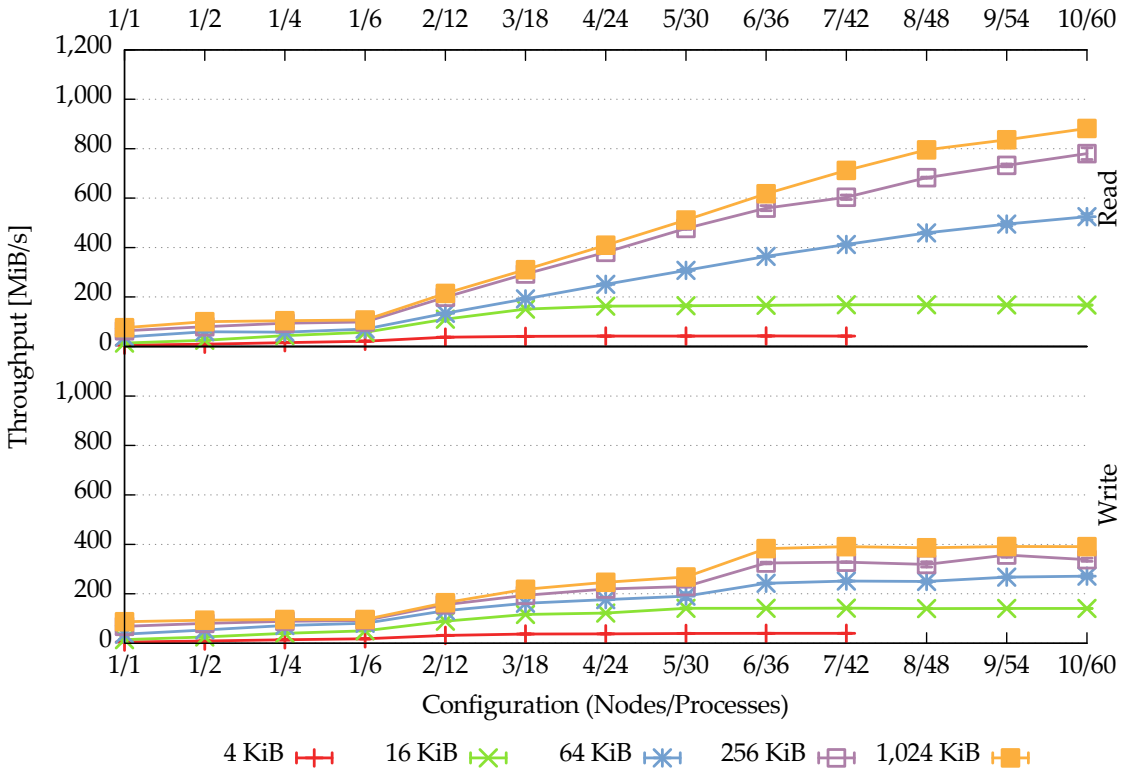


Figure 6.17.: JULEA: concurrent accesses to individual items using per-operation atomicity semantics

Individual Items Figure 6.17 shows JULEA's read and write performance when using individual items via the native JULEA interface.

Regarding read performance, it is interesting to note that different block sizes show different scaling behavior: While the block sizes of 4 KiB and 16 KiB quickly reach a maximum and stay at this level, the remaining block sizes deliver more performance as more nodes are used. This behavior can be explained using a rough performance estimation: As will be presented in Section 6.3, MongoDB manages to deliver roughly 20,000 inserts/s and 6,000 removes/s. Taking into account that each read or write operation requires one insert and one remove operation, a maximum of 13,000 operations/s can be performed.[13] This implies a maximum performance of roughly 50 MiB/s for a block size of 4 KiB and 200 MiB/s for a block size of 16 KiB. According to the measurements, 42 MiB/s and 170 MiB/s are reached for block sizes of 4 KiB and 16 KiB, respectively. Because a block size of 64 KiB can already support up to 800 MiB/s according to this approximation, the remaining block sizes' performance scales with the number of nodes. Interestingly, the largest block size of 1,024 KiB almost reaches the same performance as when using the default semantics: While the default semantics manage to deliver slightly more than 900 MiB/s, the atomicity semantics achieve a maximum of 880 MiB/s. For smaller block sizes, the slowdown is more severe, however. The maximum performance using a block size of 64 KiB drops from roughly 740 MiB/s to 530 MiB/s; this corresponds to a decrease of almost 30 %.

Regarding write performance, the small block sizes manage to deliver almost the same performance as during the read phase. While the block size of 4 KiB reaches a maximum of 40 MiB/s, the block size of 16 KiB is limited to 140 MiB/s. The remaining block sizes perform much worse, however. This is due to the lower write performance that is already present when using the default semantics. Whereas the maximum performance of roughly 650 MiB/s is reached when using seven or more nodes with the default semantics, the atomicity semantics achieve a maximum of slightly less than 400 MiB/s when using six or more nodes. This corresponds to a performance degradation of almost 40 % even when using the largest block size of 1,024 KiB.

6.2.4. Discussion

The results demonstrate that the current state of parallel distributed file systems is mixed and that performance can be very hard to predict and understand. Even simple access patterns as the ones used for the presented benchmarks do not achieve the maximum performance. This is true for all tested file systems but has different reasons for each of them.

Lustre deals well with a large number of concurrent clients. This is most likely because Lustre can easily use the OS's file system cache due to being implemented in kernel space. This allows Lustre to aggregate accesses and thus reduce the load on the servers. However, Lustre's performance is abysmal when accessing a single

[13] This number is only intended to provide a rough estimate. In practice, the number might be lower due to the high discrepancy between insert and remove performance.

shared file as commonly done in scientific applications: Read performance decreases with more than seven client nodes and write performance does not scale beyond one client node. Consequently, only individual files are efficiently usable because it is not possible to inform Lustre about the application's I/O requirements to mitigate these performance problems.

OrangeFS's handles shared files much better but its overall performance is held back by problems found within the underlying OS and file systems. In contrast to Lustre, it is not possible to use OrangeFS for I/O patterns requiring correct handling of overlapping writes.

While JULEA suffers from the same problems as OrangeFS when using an underlying POSIX file system, its NULL storage backend demonstrates that the overall architecture is able to handle high throughputs. Additionally, its different semantics allow it to adapt to a wide range of I/O requirements:

- Its default semantics enable performance results similar to those of Lustre when using large block sizes. Lustre has advantages for small block sizes due to its client-side caching and readahead functionalities. However, these advantages vanish as soon as shared files are used.
- JULEA's batches can be used to improve throughput for small block sizes by reducing the number of network messages and round trips. However, there is still potential to improve their use for large block sizes.
- The safety semantics can be used to reduce the network overhead by not awaiting the data servers' replies. This is similar to Lustre's default behavior when using individual files.
- Atomic operations can be achieved by using the atomicity semantics. While the performance of large read operations is not reduced significantly, write operations suffer a performance penalty of up to 40 %. However, using JULEA's fine-grained semantics, it is possible to use atomic operations only when absolutely necessary.

In contrast to Lustre and OrangeFS, JULEA can be adapted to different applications by setting its semantics appropriately. While it is neither possible to improve Lustre's shared file performance due to its POSIX compliance nor to use OrangeFS for workloads requiring overlapping writes, it is possible for JULEA to support and to be tuned for these specific use cases.

6.3. Metadata Performance

Due to the growing number of clients accessing parallel distributed file systems concurrently, metadata performance plays an increasingly important role for overall

file system performance. Therefore, the following measurements are meant to provide an overview of the current state of metadata performance and to highlight possibilities of exploiting semantical information to improve it.

The file systems' metadata performance will be evaluated using a large number of concurrently accessing clients that perform a variety of metadata operations: First, a number of files is created. Afterwards, the files are opened and their status is retrieved. Finally, all files are deleted again. The benchmark uses MPI to start and coordinate multiple processes accessing the file systems. There are two basic modes of operation:

1. **Individual directories:** Each process only accesses its own directory or store.[14] Even though all processes access the file system concurrently, the individual directories are accessed serially because only one process has exclusive access.

2. **Shared directory:** All processes access a single shared directory. Consequently, the shared directory will be accessed concurrently.

To evaluate the file systems' behavior with different numbers of accessing clients, the following `n/p` configurations (where `n` stands for the number of client nodes and `p` stands for the total number of client processes) have been used: 1/1, 1/2, 1/4, 1/8, 1/12, 2/24, 3/36, 4/48, 5/60, 6/72, 7/84, 8/96, 9/108 and 10/120; these are the same configurations as used for the evaluation in Section 6.2. All parallel distributed file systems have been set up to provide ten data servers and one metadata server.

The benchmark supports several I/O interfaces to support the comparison of different parallel distributed file systems using the respective interfaces. Currently, POSIX and JULEA are available. MPI-IO has not been included due to its inability to query more metadata than just the file size. Additionally, OrangeFS has been excluded due to its metadata performance problems: Previous results have shown that OrangeFS has been unable to deliver more than approximately 100 operations/s for all metadata operations that perform write accesses [Kuh13].

Each benchmark has been repeated at least five times to calculate the arithmetic mean as well as the standard deviation. To force the clients to contact the metadata servers, the clients' cache was dropped after the write phase.[15] The servers' caches have been dropped by completely restarting and remounting the file systems after each configuration; the server caches have not been touched between the different phases, however.

[14] For readability reasons, the rest of the chapter will only mention directories when either directories or stores are considered.

[15] For Lustre, the `/proc/sys/vm/drop_caches` file has been used; for JULEA, nothing has been done because no metadata has been cached on the clients.

6.3.1. Lustre

Lustre has been set up using its default options and the ldiskfs backend. The MDT has been provided by one of the SSDs, while each OST has been provided by one of the servers' HDDs.

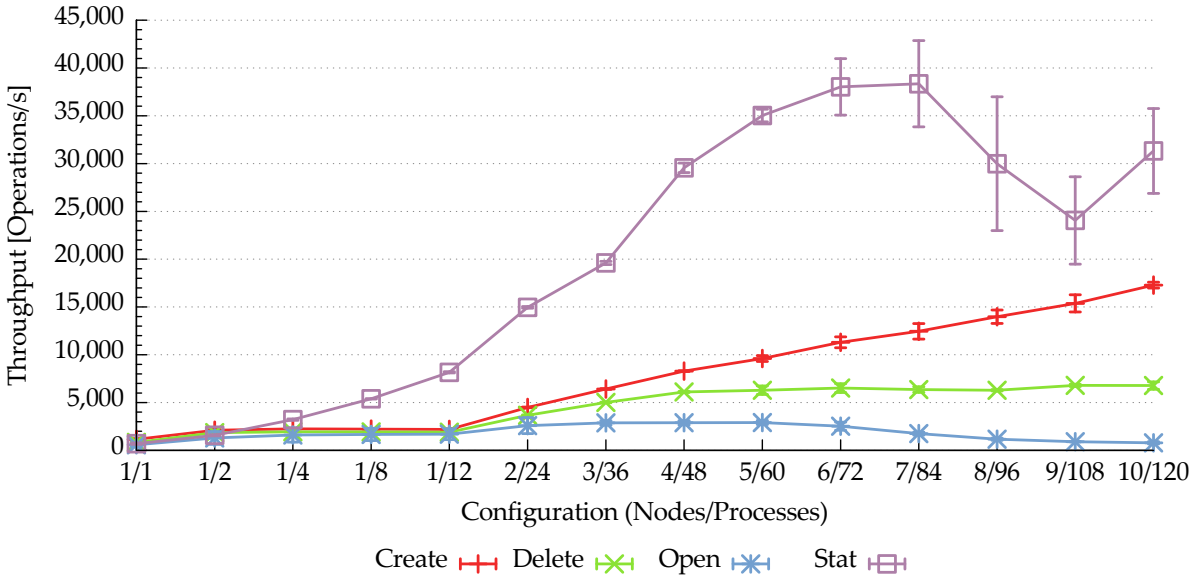


Figure 6.18.: Lustre: concurrent metadata operations to individual directories via the POSIX interface

Individual Directories Figure 6.18 shows Lustre's metadata performance when using individual directories via the POSIX interface.

As can be seen, the create performance increases together with the growing number of client nodes until it reaches its maximum of roughly 17,000 operations/s with ten nodes. The delete performance already reaches its maximum of 6,500 operations/s with five nodes and remains relatively constant even with more nodes. The performance of the open operation already reaches its maximum of 3,000 operations/s with three nodes and stays steady until five nodes are used; as soon as more nodes are used, it drops until it reaches a low point of 800 operations/s with ten nodes. It is unclear why the open operations performs so badly; the behavior might be due to `atime` updates as explained in Section 2.6.1 on pages 42–43 but more investigation regarding the underlying cause is necessary. As can be expected, the stat operation delivers high performance because it does not require any write operations; it reaches its maximum of almost 40,000 operations/s with seven nodes and sharply drops to 24,000 operations/s with nine nodes. When using ten nodes, performance increases again, which may be partly due to measurement inaccuracies because the standard

deviation is extremely large when using more than six nodes. These varying results hint at congestions inside Lustre's meta data server (MDS).

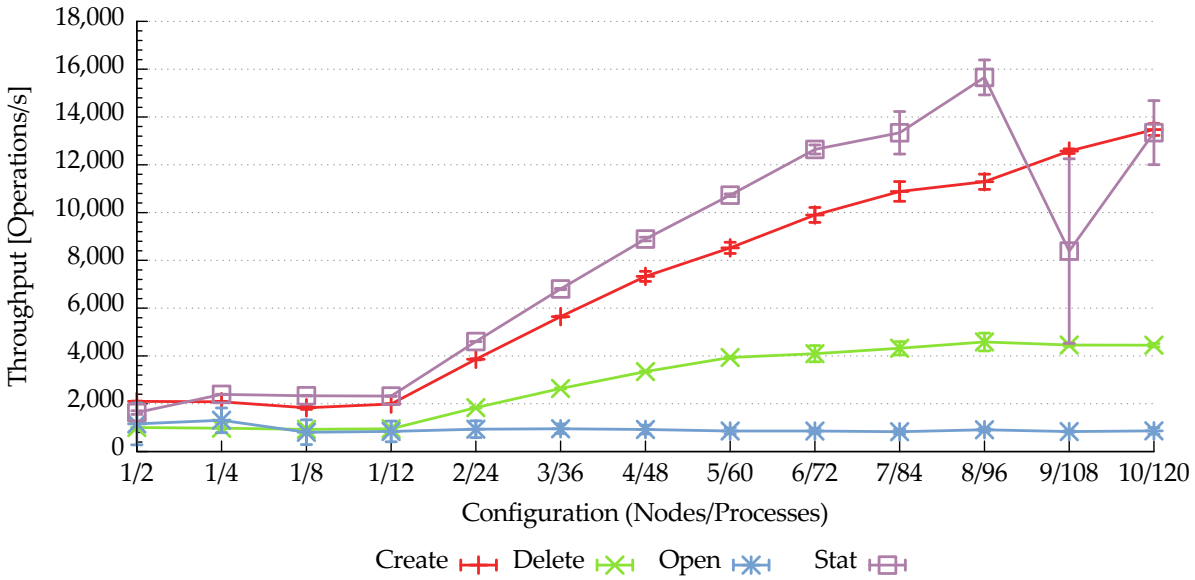


Figure 6.19.: Lustre: concurrent metadata operations to a shared directory via the POSIX interface

Shared Directory Figure 6.19 shows Lustre's metadata performance when using a shared directory via the POSIX interface.

Regarding create performance, the performance curve looks similar to the individual case except for a lower overall performance; the maximum performance with ten nodes is roughly 13,500 operations/s and thus about 20 % lower than when using individual directories. The delete operation is 30 % slower than its individual counterpart. In contrast to the individual case, the open operation's performance stays constant at roughly 800 operations/s regardless of the number of nodes. The stat operation's performance curve shows a similar form as previously with a sharp drop at nine nodes and an increase with ten nodes; however, the maximum performance is reduced by more than 60 % when compared to the individual case. The reduced performance across all metadata operations can be explained by the fact that all clients access the same shared directory which makes additional locking necessary.

6.3.2. JULEA

JULEA has been configured to use a maximum of six connections per node and to utilize the data daemon's POSIX storage backend. MongoDB has stored its data within an ext4 file system on one of the servers' SSDs, while the storage backend has used an ext4 file system located on the servers' system HDDs.

Default Semantics

The following measurements have been performed using JULEA's default semantics to establish a performance baseline; for a detailed explanation of them, see Section 3.4.9 on pages 68–70.

Shared Collection JULEA uses separate MongoDB databases for each of its stores; all collections and items within a store are saved within the corresponding database. As MongoDB performs locking on a per-database basis, using individual collections within the same store results in the same performance as using a shared one; this is different from traditional file systems where locking is usually performed on a per-directory basis. Due to this, the benchmarks using a shared collection are presented first and used as a baseline. Because developers are unlikely to use multiple stores simultaneously, this represents a more common use case.

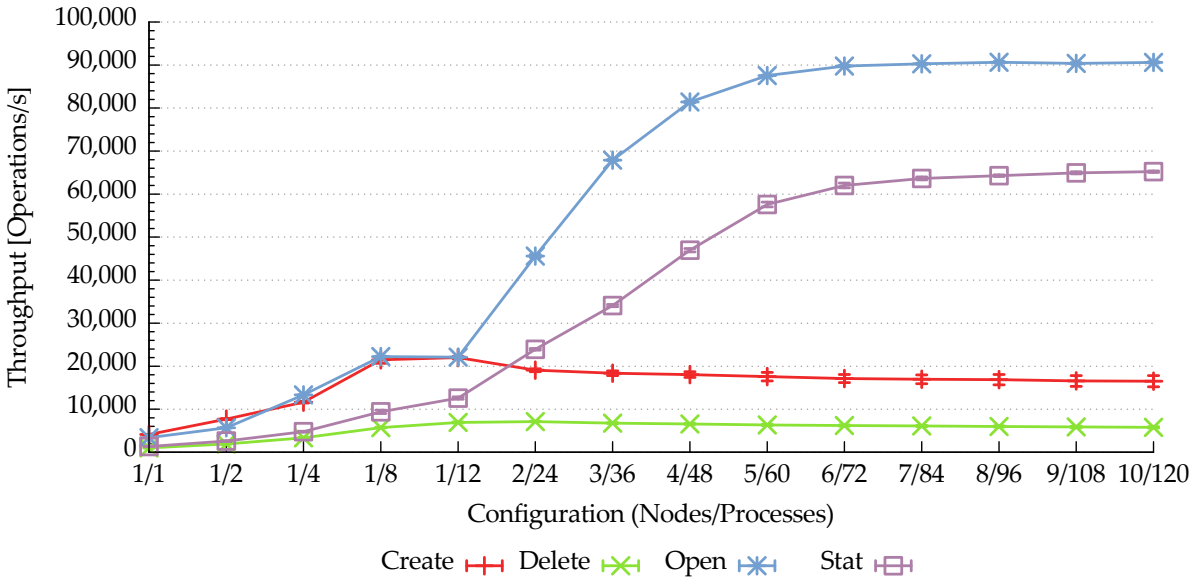


Figure 6.20.: JULEA: concurrent metadata operations to a shared collection

Individual Operations Figure 6.20 shows JULEA's metadata performance when using a shared collection.

Regarding create performance, it is interesting to note that performance increases when using more clients even on a single node. Using twelve clients yields the maximum performance of 22,000 operations/s; the performance increase from eight to twelve clients is negligible which is to be expected since JULEA has been configured to use a maximum of six connections per client. As soon as a second node is used, performance drops to roughly 19,000 operations/s and continues to do so as

more nodes are used, reaching 16,500 operations/s with ten nodes. Delete performance decreases slightly as more nodes are used; while JULEA achieves roughly 7,000 operations/s when using one or two nodes, performance decreases to slightly less than 6,000 operations/s for the ten node configuration. The low performance is due to a combination of two factors:

1. Write operations in MongoDB are inherently slower than read operations because the indexes have to be updated and the changed data has to be synchronized to stable storage. Even though JULEA does not wait for the synchronization by default, the updates still result in lower performance.

2. Delete operations do not only have to contact the metadata servers but also all data servers that contain data of the item that is to be deleted. Even though the data servers are contacted in parallel, their reply has to be awaited by default, slowing down this part of the operation.

Open performs very well because JULEA sets up MongoDB indexes for fast lookups. The open operation reaches its maximum performance of 90,000 operations/s with six clients and stays constant for more clients. This presents a stark contrast to Lustre, where the open operation is the slowest metadata operation. It is especially important for JULEA to have a high open performance because the other metadata operations (that is, delete and stat) require opening the corresponding item first. The stat operation's performance curve looks similar to that of the open operation but the overall performance is considerable lower. It reaches its maximum performance of 65,000 operations/s with seven nodes and increases only slightly for more clients. The lower performance is caused by the fact that the item's metadata has to be fetched from the data servers by default. Again, all data servers are contacted in parallel to maximize throughput but this additional step decreases overall performance.

Batch Operations Figure 6.21 on the next page shows JULEA's batch metadata performance when using a shared collection.

The performance of the stat operation is almost identical to that of its counterpart using individual operations due to the fact that it is currently not handled differently even if executed in batch mode. Even though the delete and open operations also do not perform optimizations when batched, their throughput is improved by the reduced overhead. The delete operation increases its maximum throughput to almost 13,000 operations/s, which equals an improvement of more than 110 % when compared to individual operations. Open reaches its maximum performance of 120,000 operations/s when using six nodes and stays constant when using up to ten nodes. Consequently, the open operation is sped up by 33 % when batching operations. The largest performance gain can be witnessed for the create operation. While its maximum performance was slightly less than 10,000 operations/s when using individual operations, batch operations boost this number to roughly 160,000 operations/s.

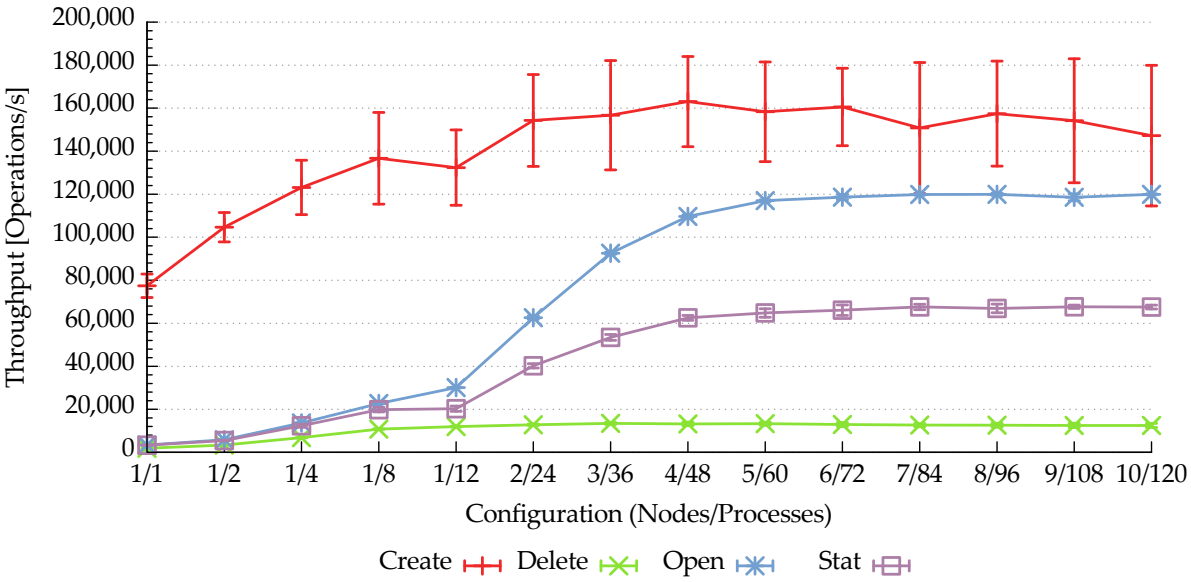


Figure 6.21.: JULEA: concurrent batch metadata operations to a shared collection

This huge performance improvement is due to the fact that JULEA makes use of MongoDB's support for so-called *bulk inserts* when possible. This allows MongoDB to improve throughput when inserting a large number of documents.[16] The create operation achieves its maximum performance when using four to six nodes and drops slightly when using more nodes. Additionally, the performance numbers show much higher deviations due to congestion in the MongoDB servers.

Individual Collections and Stores The following measurements have been performed using individual collections and stores to analyze the scaling behavior with multiple MongoDB databases. To keep the number of MongoDB databases at a reasonable level, one store per client node has been used; all clients located on this node have then created their individual collections within this store.

Individual Operations Due to a bug in MongoDB, it has not been possible to collect reliable measurements when using individual stores combined with individual operations. The high rate of operations consistently triggers the bug when multiple nodes are used. Consequently, only results for batch operations will be presented.

Batch Operations Figure 6.22 on the following page shows JULEA's batch metadata performance when using individual stores.

[16] MongoDB versions 2.6 and later support more generic *bulk write operations* that extend the bulk concept to all write operations. JULEA does not yet make use of this new feature, however.

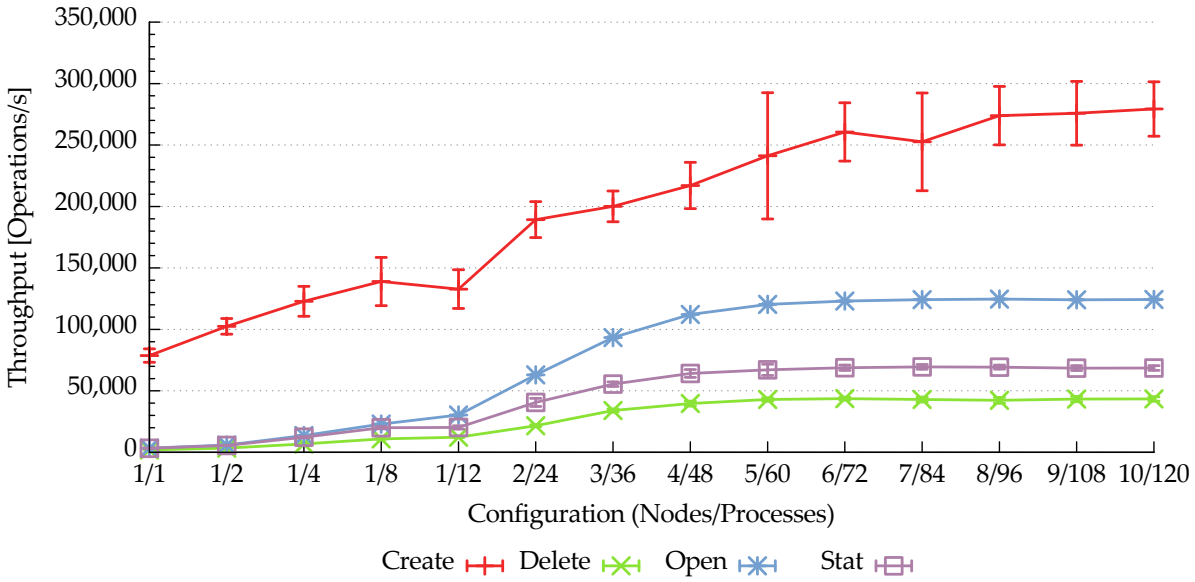


Figure 6.22.: JULEA: concurrent batch accesses to individual stores

The open and stat operations' performance is slightly higher than when using a shared collection. While the open operation reaches a maximum performance of 125,000 operations/s instead of 120,000 operations/s, the stat operation achieves 69,000 operations/s instead of 67,000 operations/s; this corresponds to improvements of 4 % and 3 %, respectively. These results are to be expected because these operations do not modify data within MongoDB which is where bottlenecks would occur during locking. The delete operation's performance curve behaves differently to its counterpart using a shared collection: Instead of delivering constant performance regardless of the number of accessing clients, performance increases until reaching its maximum of 43,000 operations/s when using five or more nodes. This corresponds to a performance increase of 230 % when compared to the results using a shared collection. The same applies to the create operation's performance: Instead of providing roughly the same performance for all configurations, using individual stores enables better scaling. It reaches its maximum performance of 280,000 operations/s when using ten nodes; this equals a performance increase of 75 %.

Concurrency Semantics

The following measurements have used differing concurrency semantics as explained in Section 3.4.2 on page 62.

Shared Collection The following measurements have been performed using a shared collection.

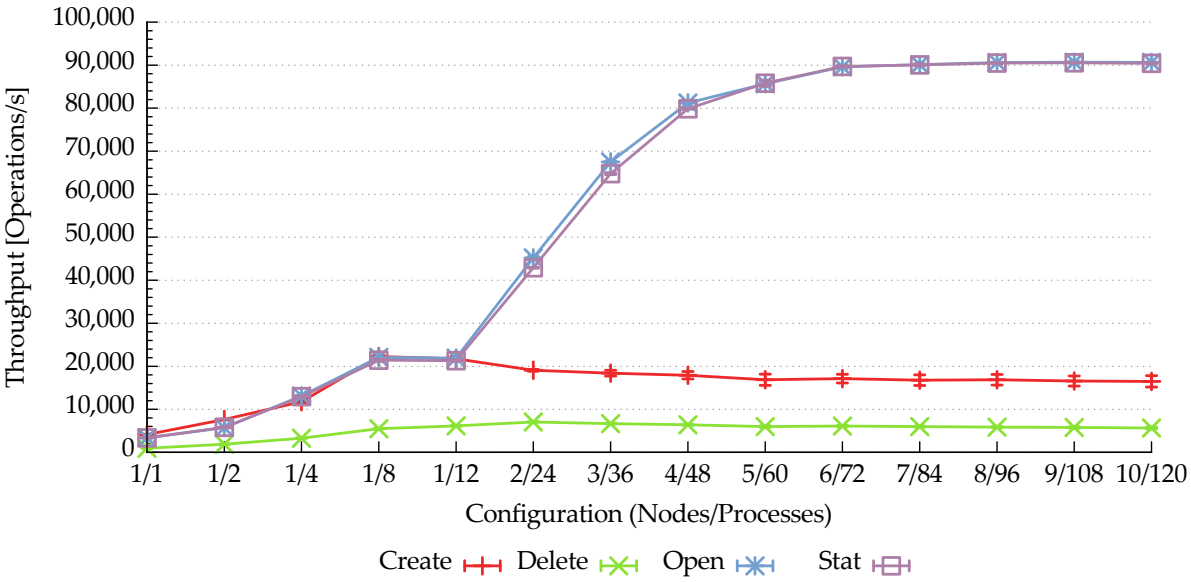


Figure 6.23.: JULEA: concurrent metadata operations to a shared collection using serial concurrency semantics

Individual Operations Figure 6.23 shows JULEA's metadata performance when using a shared collection and serial concurrency semantics.

The performance of the create and open operations is identical to that of their counterparts using the default semantics; the delete operation is slower by about 100 operations/s. There are, however, some subtle differences in behavior that might or might not have an impact on performance:

1. The create operation has to send more metadata to the MongoDB servers because the serial concurrency semantics cause the items' sizes and modification times to be stored in MongoDB. This does not have a negative effect on performance in this case because the total amount of metadata is still relatively low.[17]

2. The additional metadata also has to be fetched from the MongoDB servers when opening items because the complete MongoDB document is requested by default. This also does not have any measurable effect in this case.

3. The delete operation has to remove more data from MongoDB due to the increased document size. Because the indexes have to be updated in addition to the actual deletion, this causes a slight performance drop.

Regarding stat performance, it is interesting to note that it is nearly identical to the open operation's performance, resulting in a speedup of almost 40 % when compared

[17] Specifically, 20,000 operations/s are not enough to saturate the network due to the small size of each individual operation.

to the default semantics. This is due to the fact that the items' sizes and modification times can be fetched from MongoDB. Even though this still involves two lookup operations instead of one (that is, first opening the item and then getting its status), it can be overlapped efficiently and results in higher performance than with the default semantics. In contrast to the default semantics, this reduces the number of internal operations from eleven to two; instead of contacting ten data servers, only one additional MongoDB lookup is required. Additionally, MongoDB lookups are very fast due to the previously mentioned indexes.

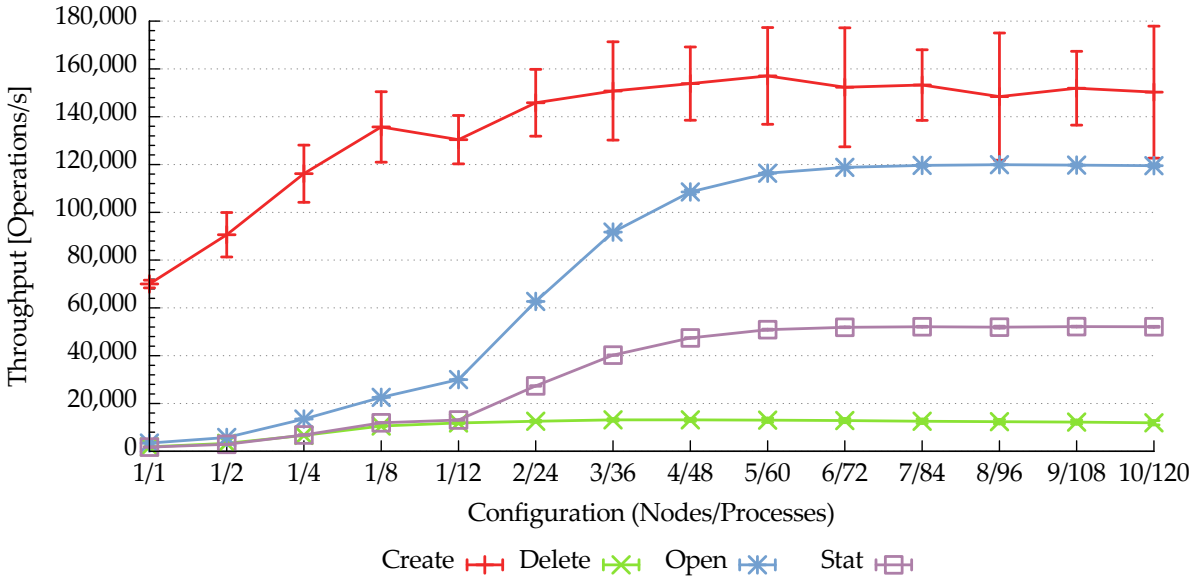


Figure 6.24.: JULEA: concurrent batch metadata operations to a shared collection using serial concurrency semantics

Batch Operations Figure 6.24 shows JULEA's batch metadata performance when using a shared collection and serial concurrency semantics.

The delete and open operations' performance is almost identical to that of their counterparts using the default semantics. The create operation's performance, however, decreases from a maximum of 163,000 operations/s to 157,000 operations/s. This slight slowdown of approximately 4 % is most likely due to the increased amount of metadata that has to be sent to the metadata server. While this effect was negligible for individual operations due to their low create performance, it becomes noticeable for batch operations. The stat performance drops from a maximum of 65,000 operations/s to 50,000 operations/s. While performance is increased by 40 % when using serial concurrency semantics with individual operations, batch operations actually slow down throughput by 25 %. This performance drop is likely due to the fact that batch operations can cause less opportunity for overlapping metadata operations because they currently lock a MongoDB connection for their entire duration.

Individual Collections and Stores The following measurements have been performed using individual collections and stores.

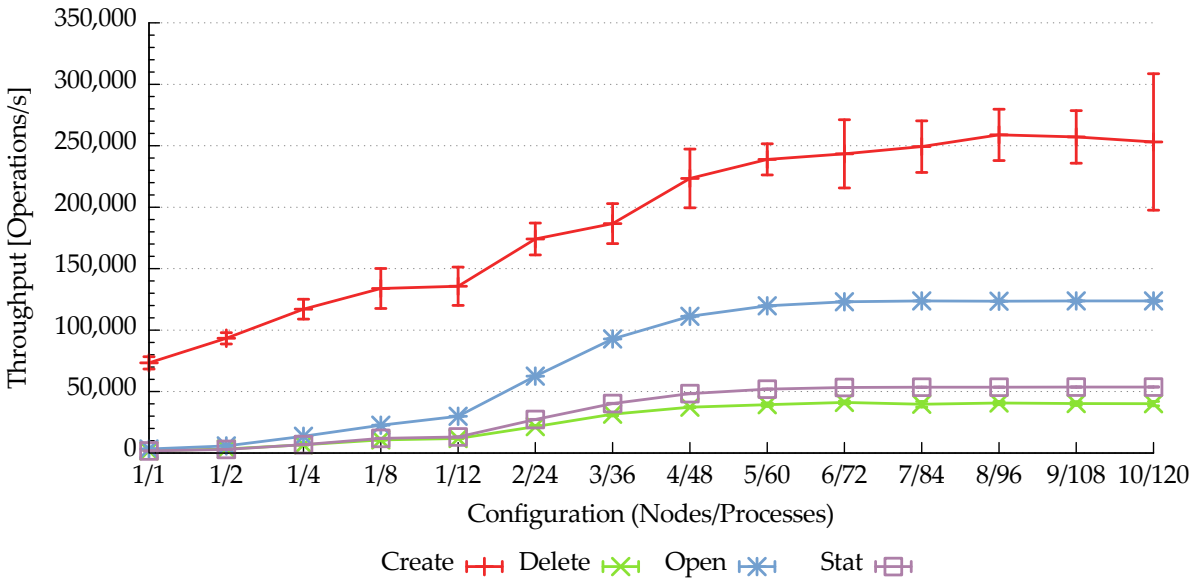


Figure 6.25.: JULEA: concurrent batch accesses to individual stores using serial concurrency semantics

Batch Operations Figure 6.25 shows JULEA's batch metadata performance when using individual stores and serial concurrency semantics.

The open operation provides almost identical performance when compared to the default semantics; all other operations are slowed down, however. The create operation reaches a maximum performance of roughly 260,000 operations/s which corresponds to a slowdown of 7 %. While the delete operation's performance is decreased by approximately 5 % with a maximum throughput of 41,000 operations/s, the stat operation achieves a maximum of 53,000 operations/s which equals a decrease of more than 20 %. As explained earlier, there are two factors that are responsible for the decline in performance:

1. There is less opportunity for overlapping operations when using batch operations because the MongoDB connections are exclusively locked for the batch operation's entire duration.

2. More information has to be sent to and retrieved from the metadata servers which is due to the serial concurrency semantics causing additional metadata to be stored in MongoDB.

While the reduced amount of overlapping is responsible for the stat operation's performance drop, the additional metadata causes slight slowdowns for both the create and delete operations.

Safety Semantics

The following measurements have used differing safety semantics as explained in Section 3.4.6 on page 65.

Shared Collection The following measurements have been performed using a shared collection.

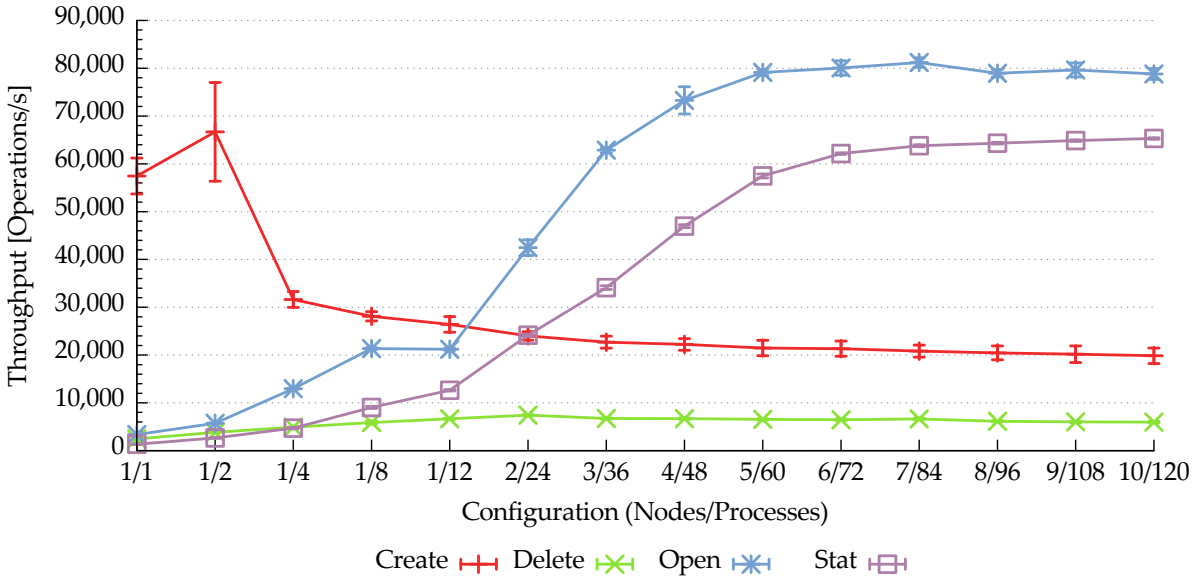


Figure 6.26.: JULEA: concurrent metadata operations to a shared collection using unsafe safety semantics

Individual Operations Figure 6.26 shows JULEA's metadata performance when using a shared collection and unsafe safety semantics.

The open operation's performance is identical to that of its counterparts using the default semantics. The delete operation's behavior is slightly modified by not awaiting MongoDB's reply when removing documents which improves performance by roughly 200 operations/s. Regarding create operation, it is interesting to note that there is a huge performance spike of 55,000–65,000 operations/s when using a single node and one or two clients. Afterwards, performance decreases to roughly 25,000 operations/s for twelve clients on one node. As in the previous cases, increasing the amount of nodes causes performance to gradually decrease until it reaches 20,000 operations/s when using ten nodes. Overall, performance is increased by 4,000–5,000 operations/s which corresponds to a 20 % improvement. The open operation's performance is reduced, however. This is most likely due to the fact that its performance is measured directly after the create operation. Because JULEA does not await

MongoDB's reply for the create operations in this case, the server might still be busy inserting documents when the benchmark's open phase begins.

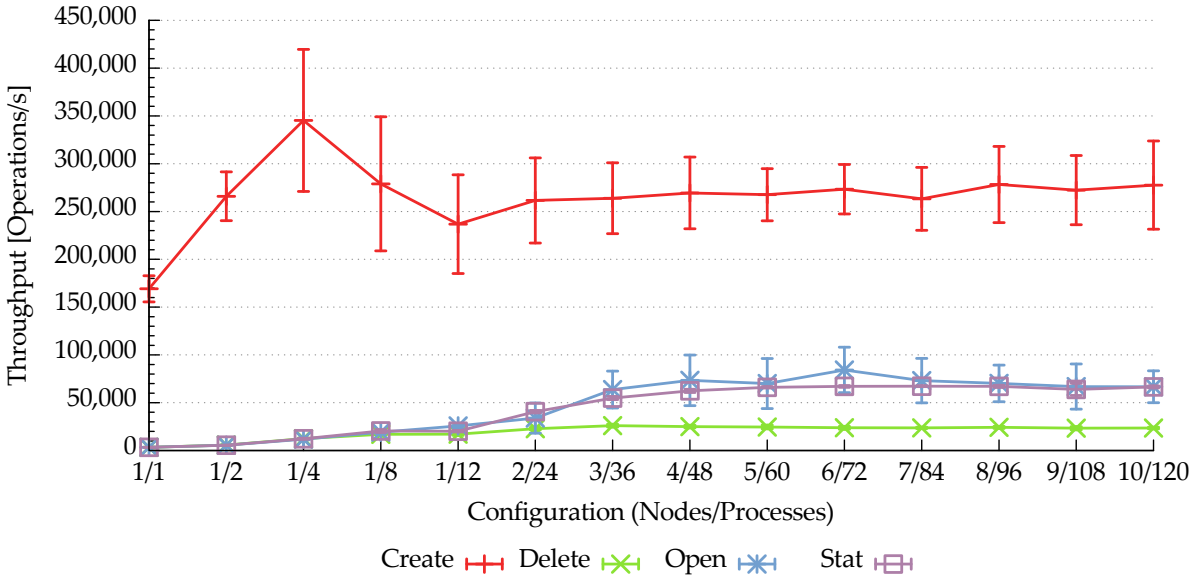


Figure 6.27.: JULEA: concurrent batch metadata operations to a shared collection using unsafe safety semantics

Batch Operations Figure 6.27 shows JULEA's batch metadata performance when using a shared collection and unsafe safety semantics.

The stat operation's performance is identical to that of its counterpart using the default semantics; since no optimizations can be performed in this case, this is expected behavior. Regarding the create operation, it can be seen that the performance remains more or less constant when using more than one node; overall, performance is much higher than when using the default semantics. This is due to the fact that no write acknowledgment is requested from MongoDB. In comparison to the default semantics, performance is increased by almost 70 % from 160,000 operations/s to 270,000 operations/s. Due to the decreased overhead facilitated by the batch operations, the delete operation's maximum performance reaches 25,000 operations/s when using more than three nodes; compared to the default semantics, this equals an increase of roughly 85 %. Again, this is due to the fact that no write acknowledgments are requested from MongoDB. Curiously, the open operation's performance is reduced to a maximum of approximately 70,000 operations/s, even though it does not behave differently depending on the chosen safety semantics. Compared to the maximum of 120,000 operations/s using the default semantics, this equals a performance drop of more than 40 %. This performance decline can be explained by the fact that the open operation is measured directly after the create operation in the benchmark application. Due to the high create throughput coupled with JULEA not waiting for write

acknowledgments, the MongoDB server is likely still busy inserting documents and updating its indexes when the open phase starts. The high performance deviations are also an indication of this because the open operation provided very stable results in all other benchmarks.

Individual Collections and Stores The following measurements have been performed using individual collections and stores.

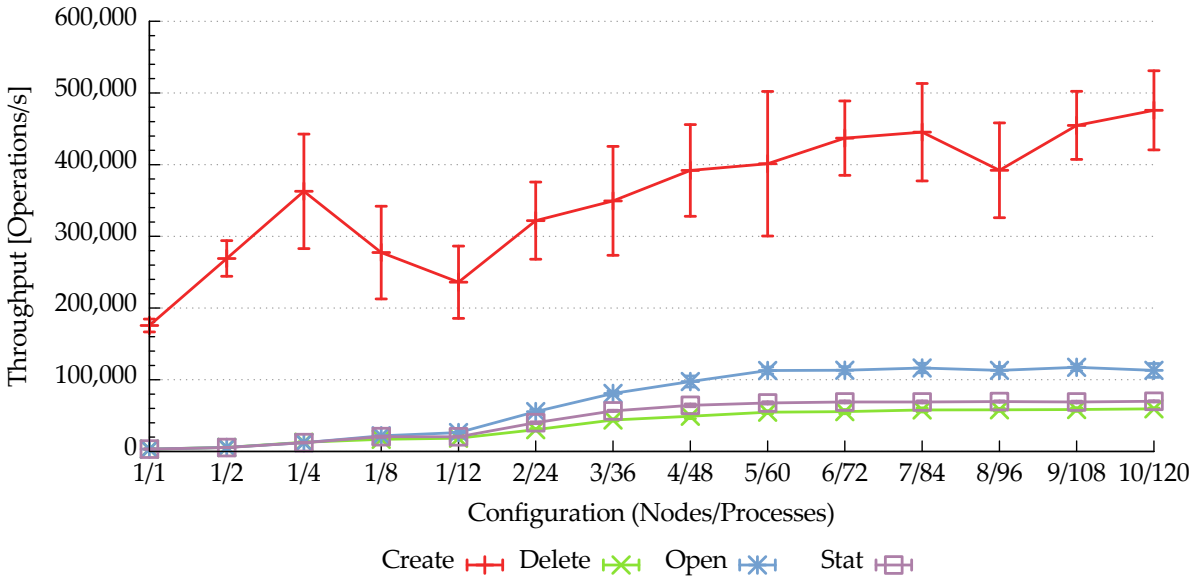


Figure 6.28.: JULEA: concurrent batch accesses to individual stores using unsafe safety semantics

Batch Operations Figure 6.28 shows JULEA's batch metadata performance when using individual stores and unsafe safety semantics.

The stat operation's performance is mostly identical when compared to that of its counterpart using the default semantics; this is not surprising because there are no differences regarding this operation when changing the safety semantics. As expected, the create operation's performance is greatly increased by almost 70 %; it reaches its maximum of 475,000 operations/s with ten nodes. Again, this improvement is due to not requesting write acknowledgments from MongoDB. Due to the congestion caused by the higher rate of document insertions, the performance deviations are much higher than with the default semantics. For the same reason, the delete operation's maximum performance increases from 43,000 operations/s with the default semantics to almost 60,000 operations/s with the unsafe safety semantics; this corresponds to an improvement of 40 %. The open operation's performance drops from 125,000 operations/s to 117,000 operations/s in comparison to the default semantics; this corresponds to a

drop of roughly 6 %. In contrast to the results when using a shared collection – where performance was decreased by 40 % – the effect of MongoDB's congestion causing a slowdown during the open phase is not as pronounced. This is likely due to the fact that multiple databases allow MongoDB to distribute the load more efficiently.

6.3.3. Discussion

The results demonstrate that Lustre's metadata performance is relatively low, even though the MDS has been configured to use an SSD as its MDT. It is especially interesting to note that while the create operation's performance scales with the number of concurrently accessing client nodes, the open operation's performance is significantly lower and deteriorates with an increasing number of clients. Consequently, this makes it possible to create files with a high rate but impossible to open them again in a passable amount of time. Additionally, the stat operation's performance is very unstable when using more than five nodes and actually decreases with an increasing number of clients. Using a shared directory again degrades the overall performance, though not as pronounced as when using shared files.

JULEA delivers performance that is capable of competing with Lustre for the create and delete operations. The open and stat operation's performance, however, is much higher than in Lustre's case. This demonstrates that the use of optimized database systems such as MongoDB can make sense for metadata servers. Projects such as the Robinhood policy engine also use database systems to speed up common file system operations [CEA14]. The different semantics and batch operations can provide significant benefits regarding metadata performance: While the concurrency and safety semantics can help to improve the performance of the stat and create operations, batch operations reduce the overall overhead caused by many small metadata requests. However, the results indicate that more fine-tuning is required for batches because they can actually reduce performance in some cases depending on the workload and metadata operation in question.

6.4. Lustre Observations

The following observations have been made while performing the previous data and metadata measurements. It has emerged that Lustre's behavior is different from that of other file systems in various ways; it is important to keep the following quirks in mind to obtain meaningful results.

Lustre caches data very aggressively. For example, when a `write` call returns, the data has usually not reached the object storage servers (OSSs) yet but has only been cached in the client's RAM. Additionally, subsequent `read` calls do not request any

data from the OSSs if the data is still cached. Consequently, it is necessary to force Lustre to flush the data to the OSSs and also retrieve the data from there.

When writing, this can be easily achieved using the `fsync` function that forces data to be flushed to stable storage, that is, the OSSs.[18] When reading, the easiest method to guarantee that data is actually retrieved from the OSSs is to empty the caches. However, there is no single simple method to accomplish this. One could use the previously mentioned `posix_fadvise` function together with its `POSIX_FADV_DONTNEED` advice. Because these advices are not well-specified and could have different behavior depending on the software environment, they are not suited for this purpose. Alternatively, Linux offers a mechanism to drop the OS's page cache: By writing the value `3` to the `/proc/sys/vm/drop_caches` file, the OS drops all non-dirty cached pages.[19] However, Lustre apparently does not mark its cached pages as non-dirty immediately after calling `fsync`. This makes it necessary to implement workarounds such as sleeping for a certain amount of time or repeatedly using this mechanism while monitoring the amount of cached pages to make sure that all cached data has been dropped.

Lustre provides very inconsistent performance results directly after starting the file system servers and mounting the file system. It is therefore necessary to wait for an appropriate amount of time before the file system has settled down.

It is sometimes not possible to unmount the Lustre file system directly after the end of a benchmark because it is still busy. In this case, it is also necessary to wait for a certain amount of time or to repeatedly check whether the file system has become idle.

As with any other kernel module, bugs can make it necessary to reboot the complete machine in order to restore functionality. It is usually not a problem to reboot the Lustre client nodes because job schedulers will take care of restarting applications. However, the load caused by the highly parallel benchmark applications also frequently made it necessary to reboot the Lustre server nodes. This was especially pronounced when performing parallel metadata measurements.

6.5. Partial Differential Equation Solver

To evaluate the different I/O interfaces' behavior with real-world applications, additional benchmarks using the *partdiff* application have been performed. partdiff solves partial differential equations (PDEs) using the Jacobi and Gauß-Seidel methods and is parallelized using MPI. Its basic memory structure is a matrix that is refined

[18] To be precise, `fsync` flushes data as well as metadata to stable storage. If only data should be flushed, `fdatasync` can be used.

[19] The value to be written is actually a bitmask: A value of `1` causes the page cache to be dropped, while a value of `2` causes cached directory entries and inodes to be dropped. Consequently, a value of `3` drops all of the above.

iteratively with each MPI process being responsible exclusively for a contiguous part of the matrix.

As mentioned previously, a common operation in scientific applications is checkpointing: All information that is necessary to resume the application later is written to storage. partdiff implements checkpointing by writing out the complete matrix to two alternating files; as soon as a checkpoint has been written out successfully, another file is used to guarantee that one valid checkpoint is available at all times, even if the application happens to crash during checkpointing. The checkpointing rate can be configured to be able to manage the I/O overhead. Since each process is responsible for a part of the matrix, the processes can write the matrix without any coordination or overlapping.

partdiff supports several different I/O interfaces to perform its checkpointing: POSIX, individual and collective MPI-IO, and JULEA. For the following evaluation, the first three I/O interfaces have been used on top of Lustre; both Lustre and JULEA have been configured as in Section 6.2.

Nodes	**Matrix Size**
1	4.89 GiB
2	9.77 GiB
3	14.65 GiB
4	19.54 GiB
5	24.42 GiB
6	29.30 GiB
7	34.18 GiB
8	39.06 GiB
9	43.95 GiB
10	48.83 GiB

Table 6.1.: partdiff matrix size depending on the number of client nodes

To evaluate the scaling behavior of the I/O interfaces and file systems, the measurements have been performed using an increasing number of clients. partdiff allows specifying the matrix's size which has been chosen in relation to the number of client nodes. To keep the amount of required computation and I/O constant, the matrix size was adjusted in such a way that doubling the amount of nodes also doubled the matrix size. The chosen matrix sizes can be found in Table 6.1. Consequently, assuming perfect scaling, the runtime for both computation and I/O should remain constant for all configurations.

partdiff has been configured to calculate 100 iterations and write checkpoints for six of these iterations; the checkpoints have been distributed evenly across all iterations. Because each client node writes 4.89 GiB of data, this results in a total of 29.32 GiB per

node and 293.15 GiB when all ten nodes are used. Using the theoretical maximum of 1,115 MiB/s for all ten nodes, partdiff's I/O is expected to take at least 262.9 s to complete. The writing of the checkpoint is accomplished using a single call to the I/O library's respective write function, that is, block sizes are not relevant in this case.[20]

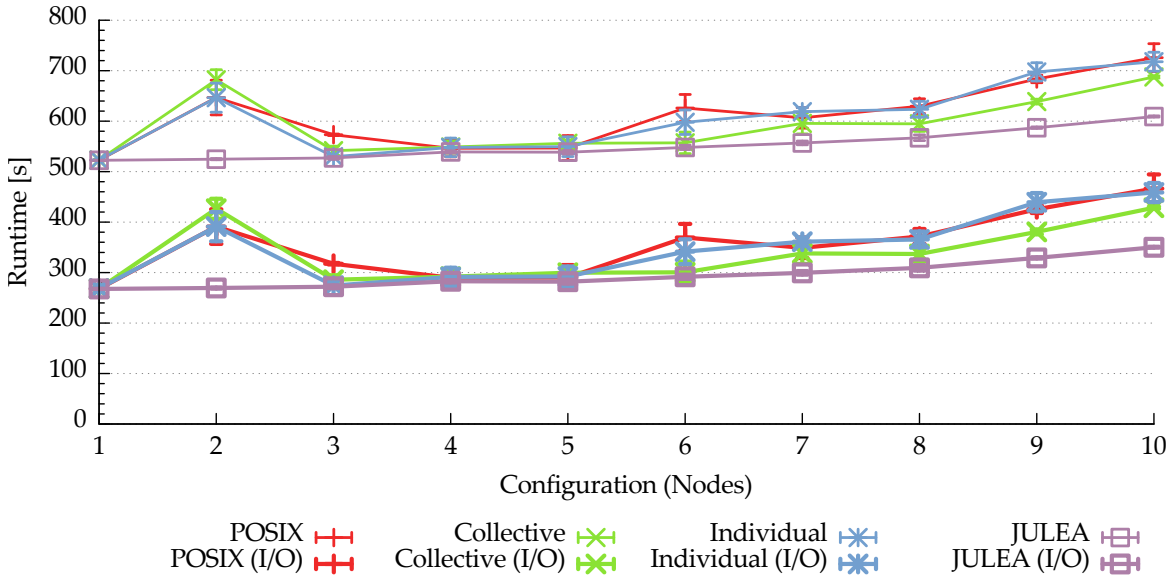


Figure 6.29.: partdiff checkpointing using one process per node

One Process Per Node Figure 6.29 shows partdiff's runtime and I/O time using different I/O interfaces with one MPI process per node. The time for computation is roughly the same for all I/O interfaces with approximately 255 s when using one node and increases slightly to about 260 s with ten nodes.

As can be seen, all I/O times increase as more nodes are used. Additionally, the changes in the total runtime mirror those in the I/O time, that is, the time consumed for computation remains constant as expected. All I/O interfaces achieve an I/O time of 268 s when using a single node. POSIX's I/O time increases to 467 s; this equals a slowdown of 74 %. The I/O time of MPI-IO's individual mode lengthens to 459 s, which corresponds to an increase of 71 %. Using MPI-IO's collective mode, it grows to 428 s, resulting in an increase of 60 %. JULEA's I/O time increases by 31 % to 350 s.

As expected, the behavior of POSIX and individual MPI-IO is largely equivalent. This is due to the fact that ADIO's POSIX backend is used and individual MPI-IO thus is simply a wrapper around the POSIX interface. Using MPI-IO's collective mode provides higher performance due to the optimizations enabled by the additional information provided by the mode's functionality.

[20] In MPI-IO's case, the checkpoint writing had to be split up into multiple calls because different MPI-IO implementations are still not fully 64-bit-safe, making it impossible to write more than 2 GiB per call.

It is interesting to note that the I/O time of all I/O interfaces except for JULEA sharply increases when going from one to two nodes and then decreases to a normal level again; this is most likely due to Lustre changing its behavior to be POSIX-compliant. Even though all I/O interfaces are only slightly slower than the theoretical maximum when using one node, all of them slow down as more nodes are used. However, this could be due to the relatively low amount of parallelism caused by limiting the amount of processes per node to one.

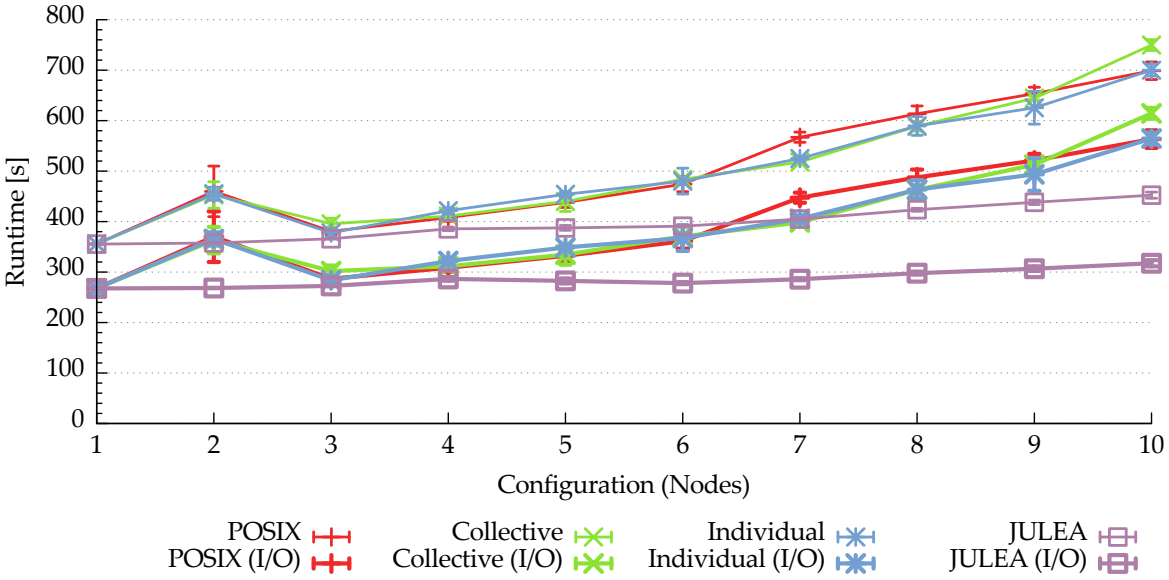


Figure 6.30.: partdiff checkpointing using six processes per node

Six Processes Per Node Figure 6.30 shows partdiff's runtime and I/O time using different I/O interfaces with six MPI processes per node. The time for computation is roughly the same for all I/O interfaces with approximately 90 s when using one node and increases to about 135 s with ten nodes.

As seen before, all I/O interfaces start out with an I/O time of 268 s when using one node. That is, more MPI processes do not influence the throughput when using a single node. However, the picture changes when more nodes access the file system concurrently. Even though the slowdown is less pronounced than in the previous benchmarks in Section 6.2, the I/O time of POSIX and both MPI-IO modes increases rapidly when more than one node is used. This is due to locking overhead introduced by many clients accessing the same shared file even when using a larger access size.

POSIX's I/O time lengthens to 563 s, which equals a slowdown of 110 %. While the I/O time of MPI-IO's individual mode increases to 565 s (111 %), the collective mode's time lengthens to 614 s (129 %). Interestingly, the collective mode does not provide performance benefits in this case. While the I/O time is roughly the same as for

MPI-IO's individual mode until nine nodes are used, the collective mode slows down significantly with ten nodes. Additionally, both modes are slower than when using only a single MPI process per node. Consequently, the increased parallelism actually degrades performance. Even though the computation is sped up by the additional processes, the overall runtime with ten nodes remains the same as when using a single process per node due to the massive I/O slowdown.

In contrast to the other I/O interfaces, JULEA's performance is improved by more concurrent clients: When going from one to ten nodes, JULEA's I/O time only grows to 318 s, which corresponds to an increase of 19 %.

6.5.1. Discussion

The measurements using partdiff represent a very simple and common use case because checkpointing is frequently used in high performance computing (HPC) applications. Additionally, partdiff's distribution of data across several MPI processes results in a seemingly uncomplicated I/O pattern of streaming and non-overlapping writes to a shared file.

Lustre shows scaling inefficiencies even when using only one MPI process per node. Its performance decreases by 40–45 % when going from one to ten client nodes. Using more processes per node exacerbates the problem, causing a performance drop of 55–60 %. Increasing the number of processes per node further is expected to make this problem even worse as demonstrated in the earlier benchmarks. While JULEA's performance drops by 33 % when using one process per node under the same circumstances, more processes per node improve performance and reduce the performance degradation to 16 %. This is due to the different semantics found in Lustre and JULEA. While it is not possible to modify Lustre's behavior to support this use case better, JULEA's semantics handle it well by default. Additionally, JULEA's semantics can be changed dynamically to support different I/O requirements.

These results make it clear that even uncomplicated use cases require workarounds to achieve maximum performance when using parallel distributed file systems such as Lustre. One such approach could be having dedicated I/O processes per node to reduce the amount of concurrent file system clients. That is, the applications have to be adapted to the parallel distributed file system instead of the other way around. This problem is especially severe if applications are supposed to run efficiently on a number of different file systems. It could be mitigated by informing the file system about the applications' actual I/O requirements as supported by JULEA.

Summary

This chapter has presented a detailed performance evaluation of multiple parallel distributed file systems and I/O interfaces. Both the data and metadata performance of Lustre, OrangeFS and JULEA has been measured using different use cases and workloads; OrangeFS has been skipped for specific measurements due to its low performance. Additionally, JULEA's batches and semantics have been thoroughly analyzed; being able to batch operations and dynamically adapt the file system's semantics depending on the applications' I/O requirements can have significant benefits. It has been shown that static approaches such as Lustre's POSIX semantics can degrade performance dramatically even for common use cases.

Chapter 7.

Conclusion and Future Work

In this chapter, the thesis will be concluded and its results will be summarized. Additionally, an outlook regarding future work will be presented. This mainly includes additional features and improvements for the proposed JULEA I/O stack that were out of scope for this thesis.

This thesis presents a new approach for handling application-specific input/output (I/O) requirements in high performance computing (HPC). The JULEA framework includes a prototypical implementation of a parallel distributed file system and provides a novel I/O interface featuring dynamically adaptable semantics. It allows applications to specify their I/O requirements using a fine-grained set of semantics. Additionally, batches enable the efficient execution of file system operations.

The results obtained in this thesis demonstrate that there is need for I/O interfaces that can adapt to the requirements of applications in order to provide adequate performance for a variety of different use cases.

While Lustre's POSIX[1] interface has advantages regarding portability, its inflexibility can cause considerable performance degradations: When using shared files, Lustre has to perform locking to remain POSIX-compliant; because there is no way to tell the file system that POSIX semantics are unnecessary or unwanted, it is not possible to avoid this performance penalty. These performance problems are even noticeable for small amounts of client processes and straightforward I/O patterns: For example, checkpointing writes data in large contiguous blocks and results in streaming I/O. However, the overhead incurred by Lustre's shared file handling still slows down performance significantly. These issues also affect higher levels of the I/O stack because Lustre effectively forces POSIX semantics upon other layers.

Other file systems such as OrangeFS are not affected by this particular issue because they do not aim to be POSIX-compliant; they are, however, also limited to their respective semantics. While this allows to deliver high performance for shared access, it excludes other I/O patterns such as conflicting and overlapping writes. For instance, this makes it impossible to use these file systems for workloads involving coordinated access to shared data structures such as file headers. Supporting them requires implementing appropriate synchronization schemes outside or on top of

[1] Portable Operating System Interface

the file system. Consequently, the prevailing problems can not be solved by simply relaxing or tightening the semantics. All static approaches have the drawback of being only suitable for a subset of use cases and workloads.

The current circumstances effectively leave application developers with two choices to be able to achieve the best possible performance:

1. Make use of different parallel distributed file systems depending on the applications' specific I/O patterns. That is, applications requiring correct handling of conflicting write operations have to be executed using a POSIX-compliant file system, while applications in need of efficient shared file handling have to use different file systems such as OrangeFS.

2. Adapt applications to work around limitations found in specific file systems. That is, applications utilize a single available file system but have to implement additional measures to make efficient use of them. For instance, writing checkpoints to node-local files could be used to circumvent Lustre's poor shared file performance. This is sometimes accomplished using specialized high-level I/O libraries such as SIONlib.

Because the first option is generally not feasible due to the given hardware and software environment of the used supercomputers, developers are usually forced to adapt their applications. An indication for this is the wide variety of I/O libraries dealing with particular file system constraints.

Even though developers and users are theoretically able to execute arbitrary user space applications – including user space file systems –, access to the supercomputers' dedicated storage is usually restricted. That is, user space file systems can typically only be set up to use the storage space available on the compute nodes. Since compute nodes are only assigned temporarily and are sometimes not even equipped with user-accessible local storage devices, this solution is not viable.

Additionally, HPC applications are often executed on multiple supercomputers that, in turn, might use different parallel distributed file systems. This can significantly increase the development and maintenance overhead because applications have to be optimized for different file systems' semantics instead of being able to optimize the file systems according to their I/O requirements.

Current file systems and I/O interfaces do not allow semantical information to be specified by the application developers even though this information could be used to optimize the file systems' behavior and thus enable high performance for a wider range of use cases. Instead, applications have to be adapted to work around the file systems' specific limitations that are imposed by their respective semantics.

JULEA presents a first approach of how application-provided semantical information can be used to adapt the file system's behavior to the applications' I/O requirements. Measurements using a wide range of I/O interfaces and workloads show that

the exploitation of this information can significantly improve performance even for common use cases.

The concept introduced by the JULEA framework fills the gap by allowing applications to adapt the file system to their exact I/O requirements instead of the other way around. For instance, this can be used to determine whether atomicity is required to handle overlapping writes correctly. Because JULEA offers a large amount of possibilities to influence the file system's semantics, only certain aspects could be evaluated in detail. Nevertheless, the available results show that the supplementary semantical information can be used to adapt the file system's behavior in such a way as to optimize performance for specific use cases. A discussion regarding application support and ideas to ease the porting of existing applications to JULEA will be presented later.

Overall, JULEA provides data and metadata performance comparable to that of other established parallel distributed file systems. In contrast to the existing file systems, its flexible semantics allow it to cover a wider range of use cases efficiently. JULEA's data performance is currently being held back by underlying problems in Linux's I/O stack: Too many parallel I/O streams significantly reduce performance even for relatively easy access patterns such as streaming I/O. Additionally, more investigation and tweaking of the MongoDB configuration will be required to eliminate the performance drop-off with larger amounts of client processes. Sharded configurations of MongoDB are also expected to increase performance even further.

These underlying problems might make it necessary to take control of the complete I/O stack to deliver high performance. JULEA is already prepared for this with its storage backend interface that makes it possible to easily support custom backends such as user space object stores. Providing all functionality of a parallel distributed file system in user space has several advantages:

1. Kernel space implementations are not as portable as those in user space due to changing kernel interfaces. An example of this is Lustre's requirement for special enterprise kernels; it is not easily possible to use Lustre's server components in combination with newer kernel versions.

2. Problems in kernel space code can make it necessary to reboot the complete machine. This is especially true for problems in Linux's virtual file system (VFS) layer that can render the complete system unusable.

3. Analyzing and debugging user space code is much easier. While a plethora of user space tools – such as GDB, Valgrind or VampirTrace – provide sophisticated and easy-to-use debugging and performance analysis functionality, analyzing and debugging the kernel is usually more tedious.

However, user space file systems also have disadvantages regarding performance: Because the file system is a normal user space process, additional context switches

might be necessary whenever a file system operation is invoked. In contrast to the mode switches that are required for kernel space file systems, context switches are more expensive because more state has to be saved and restored. Due to the high latencies of the involved network and storage operations, these additional costs can often be ignored. Overall, the benefits outweigh the drawbacks in the context of parallel distributed file systems.

JULEA's convincing metadata performance results also imply that modern database systems such as MongoDB present an interesting alternative to traditional metadata server designs. Database indexes allow fast lookups that are necessary to achieve high performance. This has also been recognized by other projects such as the Robinhood policy engine that exploit the superior performance of database systems to speed up common metadata-intensive file system operations.

Overall, the need for a more dynamic approach for parallel distributed file systems as the one implemented by JULEA is reinforced by a trend observed in several other data-centric software packages: As already presented in Section 4.4 on pages 84–85, ADIOS[2] has recently added support for read scheduling and data transformations. While read scheduling introduces the batching of read operations to improve performance, data transformations allow – among other things – to transparently compress data and thus reduce the amount of required storage and network capacities. Additionally, MongoDB has lately gained support for write concerns and bulk write operations. Write concerns allow specifying the required safety level for data and bulk write operations can be used to improve throughput. These approaches are very similar to JULEA's concepts of batches and dynamic semantics.

While there are detached activities to improve I/O interfaces, there is no uniform approach that allows the semantical information to be exploited across the complete I/O stack. This is mainly due to the fact that such activities are usually focused on high-level I/O libraries such as ADIOS. Low-level layers like MPI-IO or the actual file system are not changed. JULEA's semantics, however, establish a way to hand this information down into the file system and allow adapting it to a wide range of I/O requirements. While the aspects of atomicity, concurrency and safety have been evaluated in detail, more adjustments are possible. Additionally, semantics templates make it easy to use and adapt JULEA's semantics.

Even though JULEA provides a convenient testbed to experiment with different semantics and prototype new functionality, it is necessary to provide dynamically adaptable semantics for established I/O interfaces and parallel distributed file systems for widespread adoption of these new features. These interfaces have to be standardized and supported by a sufficiently large subset of file systems to provide consistent functionality across different implementations.

[2] Adaptable IO System

First of all, it is necessary to agree on default semantics suited for modern HPC applications and a common set of parameters that should be configurable. While POSIX allows portability across a wide range of existing file systems, it does not seem to be suited for contemporary HPC demands, as demonstrated by the results at hand. The semantics presented in this thesis are meant to provide a good starting point for further evaluation. Backwards compatibility for existing applications could also be ensured using a concept akin to JULEA.

Although JULEA's primary motivation is to establish and evaluate dynamically adaptable I/O semantics for HPC, another important goal is providing an environment to foster research. This includes file systems and object stores in general as well as novel approaches regarding I/O interfaces and semantics. It has already proven to be a good testbed for a number of bachelor and master theses that have been conducted in relation to it:

- Different parallel distributed file systems as well as I/O interfaces and semantics have been evaluated in [Jan11].
- JULEA's automatic correctness and performance regression framework has been developed in [Fuc13].
- The LEXOS[3] object store and the related JULEA storage backend have been created in [Sch13].
- A detailed analysis regarding the scalability of different I/O interfaces including HDF[4] and NetCDF[5] has been conducted in [Bar14].
- The potential performance disadvantages of user space file systems implemented using FUSE[6] have been analyzed in [Duw14].

[3] Low-Level Extent-Based Object Store
[4] Hierarchical Data Format
[5] Network Common Data Form
[6] Filesystem in Userspace

7.1. Future Work

While the prototypical JULEA framework demonstrates that semantical information can be exploited to adapt the file system's behavior, not all of its possibilities could be explored in the frame of this thesis. The following sections will give an overview of several ideas for future work.

7.1.1. Application Support

As mentioned previously, it is often unreasonable to port applications to new I/O interfaces due to their size and complexity. Because many applications already use high-level I/O libraries such as ADIOS or NetCDF, JULEA could be integrated into applications by providing backends for these I/O libraries. While ADIOS includes its own backends and could thus be extended to provide a native JULEA backend, NetCDF support could be achieved by adding a JULEA backend to HDF. HDF already includes support for POSIX and MPI-IO, and NetCDF simply delegates all I/O operations to HDF, making this a viable approach.

However, ADIOS's design is closer to JULEA due to its support for read scheduling and other advanced I/O features. Providing a backend for ADIOS would enable all ADIOS-aware applications to use JULEA without any further modifications.

ADIOS

ADIOS makes use of XML[7]-based configuration files to specify the applications' I/O. This could be easily extended to add more semantical information about the actual data, similar to what has been done in [KMKL11]. A prerequisite for this is a native JULEA backend for ADIOS as this additional information currently can not be handed down in the storage stack. Otherwise, the optimizations made possible by this information would have to be implemented within ADIOS – or any other high-level I/O library wanting to support such features. This is due to the fact that the lower layers do not support such semantical information or that it is lost through the layers. Therefore, it would be beneficial to be able to pass this information into the file system, thus alleviating the need to implement such optimizations over and over again within the upper layers as well as providing more room for optimizations in general.

```
1 <adios-config host-language="C">
2   ...
3   <semantics group="checkpoint" safety="storage"/>
4   <semantics group="temp_data" template="temporary-local"/>
5 </adios-config>
```

[7] Extensible Markup Language

Listing 7.1: ADIOS extensions

Due to ADIOS's rich XML configuration format, it would be relatively easy to extend it to support the semantical information understood by JULEA as shown in Listing 7.1 on the facing page. Analogous to the current way of being able to select the I/O method per group, the new `semantics` element would allow defining arbitrary semantics on a per-group basis. In this example, the application is supposed to write a checkpoint and some temporary data: Since the purpose of a checkpoint is to be able to restart the program in the event of a crash, it is important that it is written to persistent storage and does not end up in some kind of cache. Therefore, the safety semantics are used to ensure this property (line 3). Temporary data, however, may not need to be written to persistent storage at all. JULEA provides a semantics template for this use case that can be used (line 4).

7.1.2. Transactions

While databases usually offer atomicity, consistency, isolation and durability (ACID) semantics by means of full-featured transactions, file systems do not provide such guarantees or features. As even standard non-database applications deeply care about at least atomicity, consistency and durability, application developers have to implement appropriate measures themselves to ensure these properties. Consequently, it would be desirable to have support for transactions within file systems in some cases [WSSZ05].

While JULEA supports changing the atomicity semantics, this currently only applies to single operations within a batch and does not provide all the features real transactions provide. The atomicity semantics only apply to the operation's visibility by other processes; operations can still complete only partially in case of an error. A single failing operation within a batch could leave the data in an unexpected state and thus force the application developer to closely check each operation's result.

```
1  semantics = new Semantics(DEFAULT_SEMANTICS);
2  semantics.set(TRANSACTION, TRANSACTION_BATCH);
3
4  batch = new Batch(semantics);
5
6  item.write(..., batch);
7  item.write(..., batch);
8  item.write(..., batch);
9
10 if (!batch.execute())
```

```
11 {
12     error();
13 }
```

Listing 7.2: JULEA transactions

The pseudo code in Listing 7.2 on the previous page shows how transactions could be used in JULEA. First, a new semantics object is created (line 1) and its transaction semantics are set to provide transactions for the complete batch (line 2). Afterwards, a batch is created using these semantics (line 4). Several write operations are performed using this batch (lines 6–8). Should any of these write operations fail, the whole batch will fail and the item's contents will equal those before the batch's execution (lines 10–13).

Providing this kind of support directly in the file system would free application developers from the burden of complex error handling. Transactions fit naturally into the concept of batches since there are already well-defined start and end points for batches, similar to transactions. Support for full-featured transactions would be more oriented towards programming efficiency rather than increased performance as they mainly offer convenient ways for error handling and cleanup. However, it would allow developers to focus on the actual I/O instead of worrying about correct error handling, which would in turn lead to cleaner and more maintainable code.

7.1.3. Object Store

As explained in Chapter 3, it would be beneficial for JULEA to make use of an object store in order to avoid the overhead of a full-featured POSIX file system. As JULEA already handles most file system operations itself, it is not necessary for an underlying file system to perform redundant operations such as path lookup and permission checking. As the storage backends' primary purpose it to efficiently handle parallel I/O streams and the actual block allocation, object stores provide a fitting alternative. Another important aspect is the fact that JULEA's performance is negatively impacted by Linux's current VFS layer. To take control of the complete I/O stack, it is necessary to eliminate this dependency and provide a storage backend that is tailored to JULEA's requirements regarding batches and semantics. This can be accomplished most easily and effectively using user space object stores as they are easier to adapt than full-featured kernel space file systems.

LEXOS is an initial prototype of such an object store and has been implemented as a shared library in user space. This allows it to be used easily in other projects and would enable JULEA to have complete control over all aspects of the resulting I/O path. It already supports batches and could thus be easily integrated into JULEA's I/O stack. However, it still requires further evaluations and optimizations for massively parallel workloads as found in parallel distributed file systems.

7.1.4. Client Optimizations

The results shown in Chapter 6 demonstrate that there are still a few possibilities for further optimizations within JULEA:

- Merging of write operations currently happens inside the data daemon. However, overall performance could be increased by already merging them in the client library to reduce the overhead of the involved network messages. This is due to the fact that the offsets and lengths of all operations have to be sent to the data daemon only to be merged there. Consequently, merging them in the client library would reduce the amount of data that has to be sent across the network.
- Even though the use of TCP[8] corking reduces the network overhead, it should be investigated whether small write operations can be handled more efficiently by storing the data of all operations in a contiguous buffer to reduce the number of network send operations.
- Merging of operations is currently only performed for write operations. Read operations could benefit from a similar handling, both within the client library and the data daemon.
- Scheduling many small operations within a batch currently involves many memory reallocations. More intelligent algorithms could be used to reduce the number of reallocations and thus speed up the handling of large batches.

Summary

This chapter has summarized the insights gained in this thesis. Due to their static approaches regarding I/O semantics, traditional parallel distributed file systems can not be suited for all possible use cases and workloads. JULEA's dynamically adaptable semantics present a first approach for exploiting application-provided semantical information to optimize I/O performance. Additionally, several tasks for future work have been presented to improve JULEA's coverage of the I/O stack: While support of existing applications can be eased by providing backends for ADIOS or HDF, object stores provide opportunities to become independent of underlying POSIX file systems.

[8] Transmission Control Protocol

Bibliography

[10g13] 10gen, Inc. MongoDB. `http://www.mongodb.org/`, 2013. Last accessed: 2014-11.

[ADD+08] Nawab Ali, Ananth Devulapalli, Dennis Daless, Pete Wyckoff, and P. Sadayappan. Revisiting the Metadata Architecture of Parallel File Systems. Technical Report OSU-CISRC-7/08-TR42, 2008.

[AEHH+11] Sadaf R. Alam, Hussein N. El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. Parallel I/O and the Metadata Wall. In *Proceedings of the sixth workshop on Parallel Data Storage*, PDSW '11, pages 13–18, New York, NY, USA, 2011. ACM.

[AKGR10] Samer Al-Kiswany, Abdullah Gharaibeh, and Matei Ripeanu. The Case for a Versatile Storage System. *SIGOPS Oper. Syst. Rev.*, (1):10–14, 01 2010.

[Bar14] Christopher Bartz. An in-depth analysis of parallel high level I/O interfaces using HDF5 and NetCDF-4. Master's thesis, University of Hamburg, 04 2014.

[Bia08] Christoph Biardzki. *Analyzing Metadata Performance in Distributed File Systems*. PhD thesis, Heidelberg University, Germany, 12 2008.

[BLZ+14] D.A Boyuka, S. Lakshminarasimham, Xiaocheng Zou, Zhenhuan Gong, J. Jenkins, E.R. Schendel, N. Podhorszki, Qing Liu, S. Klasky, and N.F. Samatova. Transparent in Situ Data Transformations in ADIOS. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 256–266, May 2014.

[BVGS06] Stephan Bloehdorn, Max Völkel, Olaf Görlitz, and Simon Schenk. TagFS — Tag Semantics for Hierarchical File Systems. In *Proceedings of the 6th International Conference on Knowledge Management*, 2006.

[Cat10] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, (39-4):12–27, 2010.

[CDKL14] Konstantinos Chasapis, Manuel Dolz, Michael Kuhn, and Thomas Ludwig. Evaluating Power-Performace Benefits of Data Compression

in HPC Storage Servers. In Steffen Fries and Petre Dini, editors, *IARIA Conference*, pages 29–34. IARIA XPS Press, 04 2014.

[CEA14] CEA. Robinhood Policy Engine. `https://github.com/cea-hpc/robinhood/wiki`, 08 2014. Last accessed: 2014-11.

[CFF+95] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO Parallel I/O Interface. In *IPPS'95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, 1995.

[CLR+09] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-File Access in Parallel File Systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing*, IPDPS '09, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[Clu02] Cluster File Systems, Inc. Lustre: A Scalable, High-Performance File System. `http://www.cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf`, 11 2002. Last accessed: 2014-11.

[CST+11] Yong Chen, Xian-He Sun, Rajeev Thakur, Philip C. Roth, and William D. Gropp. LACIO: A New Collective I/O Strategy for Parallel I/O Systems. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11, pages 794–804, Washington, DC, USA, 2011. IEEE Computer Society.

[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.

[DT98] Phillip M. Dickens and Rajeev Thakur. A Performance Study of Two-Phase I/O. In *In Proceedings of the 4th International Euro-Par Conference. Lecture Notes in Computer Science 1470*, pages 959–965. Springer-Verlag, 1998.

[Duw14] Kira Isabel Duwe. Comparison of kernel and user space file systems. Bachelor's thesis, University of Hamburg, 08 2014.

[Fel13] Dave Fellinger. The State of the Lustre File System and The Lustre Development Ecosystem. `http://www.opensfs.org/wp-content/uploads/2013/04/LUG_2013_vFinal.pdf`, 04 2013. Last accessed: 2014-11.

[Fuc13] Anna Fuchs. Automated File System Correctness and Performance Regression Tests. Bachelor's thesis, University of Hamburg, 09 2013.

[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 '09, New York, NY, USA, 2009. ACM.

[GAKR08] Abdullah Gharaibeh, Samer Al-Kiswany, and Matei Ripeanu. Configurable security for scavenged storage systems. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, Storage '08, pages 55–62, New York, NY, USA, 2008. ACM.

[Ger14] German Climate Computing Center. Tape Archive – HPSS filesystems. `https://www.dkrz.de/Nutzerportal-en/doku/hpss`, 11 2014. Last accessed: 2014-11.

[GGH91] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance Evaluation of Memory Consistency Models for Shared-memory Multiprocessors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 245–257, New York, NY, USA, 1991. ACM.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 15–26, New York, NY, USA, 1990. ACM.

[Glu11] Gluster, Inc. GlusterFS General FAQ. `http://gluster.org/community/documentation/index.php/GlusterFS_General_FAQ#What_file_system_semantics_does_GlusterFS_Support.3B_is_it_fully_POSIX_compliant.3F`, 05 2011. Last accessed: 2014-11.

[GWT14] GWT-TUD GmbH. Vampir. `https://www.vampir.eu/`, 07 2014. Last accessed: 2014-11.

[HJZ+09] Yu Hua, Hong Jiang, Yifeng Zhu, Dan Feng, and Lei Tian. SmartStore: A New Metadata Organization Paradigm with Semantic-Awareness for Next-Generation File Systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. ACM.

[HK04] Rainer Hubovsky and Florian Kunz. Dealing with Massive Data: from Parallel I/O to Grid I/O. Master's thesis, University of Vienna, Department of Data Engineering, 01 2004.

[HNH09] Dean Hildebrand, Arifa Nisar, and Roger Haskin. pNFS, POSIX, and MPI-IO: A Tale of Three Semantics. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 32–36, New York, NY, USA, 2009. ACM.

[IG13] The IEEE and The Open Group. Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7. *IEEE Std 1003.1, 2013 Edition (incorporates IEEE Std 1003.1-2008, and IEEE Std 1003.1-2008/Cor 1-2013)*, pages 1–3906, April 2013.

[IMOT12] Shun Ishiguro, Jun Murakami, Yoshihiro Oyama, and Osamu Tatebe. Optimizing Local File Accesses for FUSE-Based Distributed Storage. *High Performance Computing, Networking Storage and Analysis, SC Companion:*, 0:760–765, 2012.

[ISO11] ISO/IEC JTC 1/SC 22 – Programming languages, their environments and system software interfaces. ISO/IEC 9899:2011 – Information technology – Programming languages – C. 12 2011.

[Jan11] Christina Janssen. Evaluation of File Systems and I/O Optimization Techniques in High Performance Computing. Bachelor's thesis, University of Hamburg, 12 2011.

[JKY00] T. Jones, A Koniges, and R.K. Yates. Performance of the IBM General Parallel File System. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 673–681, 2000.

[KBB+06] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the Open Trace Format (OTF). In Vassil N. Alexandrov, Geert Dick Albada, Peter M.A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, number 3992 in Lecture Notes in Computer Science, pages 526–533, Berlin / Heidelberg, Germany, 2006. Springer-Verlag GmbH.

[KKL08] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Directory-Based Metadata Optimizations for Small Files in PVFS. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 90–99, Berlin, Heidelberg, 2008. University of Las Palmas de Gran Canaria, Springer-Verlag.

[KKL09] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Dynamic file system semantics to enable metadata optimizations in PVFS. *Concurrency and Computation: Practice and Experience*, pages 1775–1788, 2009.

[KKL14] Julian Kunkel, Michael Kuhn, and Thomas Ludwig. Exascale Storage Systems – An Analytical Study of Expenses. *Supercomputing Frontiers and Innovations*, pages 116–134, 06 2014.

[KLL+10] Scott Klasky, Qing Liu, Jay Lofstead, Norbert Podhorszki, Hasan Abbasi, CS Chang, Julian Cummings, Divya Dinakar, Ciprian Docan, Stephanie Ethier, Ray Grout, Todd Kordenbrock, Zhihong Lin, Xiaosong Ma, Ron Oldfield, Manish Parashar, Alexander Romosan, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Yuan Tian, Matthew Wolf, Weikuan Yu, Fan Zhang, and Fang Zheng. ADIOS: powering I/O to extreme scale computing. In *SciDAC 2010 Conference Proceedings*, pages 342–347, 2010.

[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.

[Kre06] Stephan Krempel. Tracing the Connections Between MPI-IO Calls and their Corresponding PVFS2 Disk Operations. Bachelor's thesis, Heidelberg University, 03 2006.

[Kuh13] Michael Kuhn. A Semantics-Aware I/O Interface for High Performance Computing. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing*, number 7905 in Lecture Notes in Computer Science, pages 408–421, Berlin, Heidelberg, 06 2013. Springer.

[Kun06] Julian Kunkel. Performance Analysis of the PVFS2 Persistency Layer. Bachelor's thesis, Heidelberg University, 02 2006.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[LCB13] Paul Hermann Lensing, Toni Cortes, and André Brinkmann. Direct lookup and hash-based metadata placement for local file systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 5:1–5:11, New York, NY, USA, 2013. ACM.

[LCC+12] N. Liu, J. Cope, Philip H. Carns, C. D. Carothers, Robert B. Ross, G. Grider, A. Crume, and C. Maltzahn. On the Role of Burst Buffers in Leadership-Class Storage Systems. In *Proceedings of MSST/SNAPI 2012*, Pacific Grove, CA, 04/2012 2012.

[LKK+07] Thomas Ludwig, Stephan Krempel, Michael Kuhn, Julian Kunkel, and Christian Lohse. Analysis of the MPI-IO Optimization Levels with the PIOViz Jumpshot Enhancement. In Franck Cappello, Thomas Hérault, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 4757 in Lecture Notes in Computer Science, pages 213–222, Berlin / Heidelberg, Germany, 2007. Institut national de recherche en informatique et automatique, Springer.

[LKS+08] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, CLADE '08, pages 15–24, New York, NY, USA, 06 2008. ACM.

[LM10] Paulo A. Lopes and Pedro D. Medeiros. pCFS vs. PVFS: Comparing a Highly-Available Symmetrical Parallel Cluster File System with an Asymmetrical Parallel File System. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 131–142, Berlin, Heidelberg, 2010. Springer-Verlag.

[LMB10] Paul Lensing, Dirk Meister, and André Brinkmann. hashFS: Applying Hashing to Optimize File Systems for Small File Reads. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, SNAPI '10, pages 33–42, Washington, DC, USA, 2010. IEEE Computer Society.

[LRT04] Robert Latham, Robert B. Ross, and Rajeev Thakur. The Impact of File Systems on MPI-IO Scalability. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 3241 in Lecture Notes in Computer Science, pages 87–96. Springer, 2004.

[LRT07] Robert Latham, Robert Ross, and Rajeev 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.

[MCB+07] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, 2007.

[Mea03] Ryan L. Means. Alternate Data Streams: Out of the Shadows and into the Light. `http://www.giac.org/paper/gcwn/230/alternate-data-streams-shadows-light/104234`, 2003. Last accessed: 2014-11.

[Mes01] Message Passing Interface Forum. Opening a File. `http://www.mpi-forum.org/docs/mpi-2.0/mpi-20-html/node175.htm`, 09 2001. Last accessed: 2014-11.

[Mes12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 3.0. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, 09 2012. Last accessed: 2014-11.

[MKB+12] Dirk Meister, Jürgen Kaiser, Andre Brinkmann, Michael Kuhn, Julian Kunkel, and Toni Cortes. A Study on Data Deduplication in HPC Storage Systems. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC)*. IEEE Computer Society, 11 2012.

[MM01] J. C. Mogul and G. Minshall. Rethinking the TCP Nagle Algorithm. *SIGCOMM Comput. Commun. Rev.*, 31(1):6–20, January 2001.

[MMK+12] Timo Minartz, Daniel Molka, Julian Kunkel, Michael Knobloch, Michael Kuhn, and Thomas Ludwig. *Tool Environments to Measure Power Consumption and Computational Performance*, chapter 31, pages 709–743. Chapman and Hall/CRC Press Taylor and Francis Group, 6000 Broken Sound Parkway NW, Boca Raton, FL 33487, 2012.

[MSM+11] Christopher Muelder, Carmen Sigovan, Kwan-Liu Ma, Jason Cope, Sam Lang, Kamil Iskra, Pete Beckman, and Robert Ross. Visual Analysis of I/O System Behavior for High–End Computing. In *Proceedings of the third international workshop on Large-scale system and application performance*, LSAP '11, pages 19–26, New York, NY, USA, 2011. ACM.

[MSMV00] Greg Minshall, Yasushi Saito, Jeffrey C. Mogul, and Ben Verghese. Application Performance Pitfalls and TCP's Nagle Algorithm. *SIGMETRICS Perform. Eval. Rev.*, 27(4):36–44, March 2000.

[Ora11] Oracle. Guide to Scaling Web Databases with MySQL Cluster, 10 2011.

[PG11] Swapnil Patil and Garth Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[PGG+09] Swapnil Patil, Garth A. Gibson, Gregory R. Ganger, Julio Lopez, Milo Polte, Wittawat Tantisiroj, and Lin Xiao. In search of an API for scalable file systems: Under the table or above it? In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[PGLP07] Swapnil V. Patil, Garth A. Gibson, Sam Lang, and Milo Polte. GIGA+: Scalable Directories for Shared File Systems. In *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, pages 26–29, New York, NY, USA, 2007. ACM.

[PLB+09] Milo Polte, Jay Lofstead, John Bent, Garth Gibson, Scott A. Klasky, Qing Liu, Manish Parashar, Norbert Podhorszki, Karsten Schwan, Meghan Wingate, and Matthew Wolf. ...And eat it too: High read performance in write-optimized HPC I/O middleware file formats. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 21–25, New York, NY, USA, 2009. ACM.

[RD90] Russ Rew and Glenn Davis. Data Management: NetCDF: an Interface for Scientific Data Access. *IEEE Computer Graphics and Applications*, (10-4):76–82, 1990.

[RG10] Aditya Rajgarhia and Ashish Gehani. Performance and Extension of User Space File Systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, New York, NY, USA, 2010. ACM.

[RLG+05] R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen. Implementing MPI-IO atomic mode without file system support. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, number 2 in CCGRID '05, pages 1135–1142, Washington, DC, USA, 2005. IEEE Computer Society.

[Ros08] P. E. Ross. Why CPU Frequency Stalled. *IEEE Spectr.*, 45(4):72–72, April 2008.

[Sch13] Sandra Schröder. Design, Implementation, and Evaluation of a Low-Level Extent-Based Object Store. Master's thesis, University of Hamburg, 12 2013.

[Seh10] Saba Sehrish. *Improving Performance and Programmer Productivity for I/O-Intensive High Performance Computing Applications*. PhD thesis, School of Electrical Engineering and Computer Science in the College of Engineering and Computer Science at the University of Central Florida, 2010.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association, USENIX Association.

[SKH+08] Jan Stender, Björn Kolbeck, Felix Hupfeld, Eugenio Cesario, Erich Focht, Matthias Hess, Jesús Malo, and Jonathan Martí. Striping without Sacrifices: Maintaining POSIX Semantics in a Parallel File System. In *First USENIX Workshop on Large-Scale Computing*, LASCO'08, Berkeley, CA, USA, 2008. USENIX Association.

[SLG03] Thomas Sterling, Ewing Lusk, and William Gropp. *Beowulf Cluster Computing with Linux*. MIT Press, Cambridge, MA, USA, 2 edition, 2003.

[SM09] Margo Seltzer and Nicholas Murphy. Hierarchical File Systems are Dead. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[SNAKA+08] Elizeu Santos-Neto, Samer Al-Kiswany, Nazareno Andrade, Sathish Gopalakrishnan, and Matei Ripeanu. Hot Topic: Enabling Cross-Layer Optimizations in Storage Systems with Custom Metadata. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 213–216, New York, NY, USA, 2008. ACM.

[TGL99] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, pages 182–, Washington, DC, USA, 1999. IEEE Computer Society.

[The14a] The HDF Group. Hierarchical data format version 5. `http://www.hdfgroup.org/HDF5`, 07 2014. Last accessed: 2014-11.

[The14b] The Linux man-pages project. write(2). `http://man7.org/linux/man-pages/man2/write.2.html`, 05 2014. Last accessed: 2014-11.

[The14c] The TOP500 Editors. TOP500. `http://www.top500.org/`, 06 2014. Last accessed: 2014-11.

[Tie09] Tien Duc Tien. Tracing Internal Behavior in PVFS. Bachelor's thesis, Heidelberg University, 10 2009.

[TRL+10] Rajeev Thakur, Robert Ross, Ewing Lusk, William Gropp, and Robert Latham. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. `http://www.mcs.anl.gov/research/projects/romio/doc/users-guide.pdf`, 04 2010. Last accessed: 2014-11.

[TSP+11] Wittawat Tantisiriroj, Seung Woo Son, Swapnil Patil, Samuel J. Lang, Garth Gibson, and Robert B. Ross. On the Duality of Data-intensive

File System Design: Reconciling HDFS and PVFS. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 67:1–67:12, New York, NY, USA, 2011. ACM.

[Unk12] Unknown. nfs(5). `http://linux.die.net/man/5/nfs`, 10 2012. Last accessed: 2014-11.

[VLR+08] M. Vilayannur, S. Lang, R. Ross, R. Klundt, and L. Ward. Extending the POSIX I/O Interface: A Parallel File System Perspective. Technical Report ANL/MCS-TM-302, 10 2008.

[VNS05] Murali Vilayannur, Partho Nath, and Anand Sivasubramaniam. Providing Tunable Consistency for a Parallel File Store. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, Berkeley, CA, USA, 2005. USENIX Association.

[VRC+04] Murali Vilayannur, Robert B. Ross, Philip H. Carns, Rajeev Thakur, Anand Sivasubramaniam, and Mahmut Kandemir. On the Performance of the POSIX I/O Interface to PVFS. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, page 332, 2004.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[Wik14a] Wikimedia Commons. File talk:Hard drive capacity over time.svg. `http://commons.wikimedia.org/wiki/File_talk:Hard_drive_capacity_over_time.svg`, 11 2014. Last accessed: 2014-11.

[Wik14b] Wikipedia. Festplattenlaufwerk – Geschwindigkeit. `http://de.wikipedia.org/wiki/Festplattenlaufwerk#Geschwindigkeit`, 11 2014. Last accessed: 2014-11.

[Wik14c] Wikipedia. Fork (file system). `http://en.wikipedia.org/wiki/Fork_(file_system)`, 11 2014. Last accessed: 2014-11.

[Wik14d] Wikipedia. IOPS. `http://en.wikipedia.org/wiki/IOPS`, 11 2014. Last accessed: 2014-11.

[Wik14e] Wikipedia. Mark Kryder – Kryder's Law. `http://en.wikipedia.org/wiki/Mark_Kryder#Kryder.27s_Law`, 11 2014. Last accessed: 2014-11.

[WKRP06] An-I Andy Wang, Geoff Kuenning, Peter Reiher, and Gerald Popek. The Conquest File System: Better Performance Through a Disk/Persistent-RAM Hybrid Design. *Trans. Storage*, (3):309–348, 08 2006.

[Won14] Darrick J. Wong. Ext4 Disk Layout. `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout`, 11 2014. Last accessed: 2014-11.

[Wri06] Charles Philip Wright. *Extending ACID Semantics to the File System via ptrace*. PhD thesis, Stony Brook University, 05 2006.

[WSSZ05] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Amino: Extending ACID Semantics to the File System. In *FAST 2005 2nd Usenix Conference on File and Storage Technologies*, Berkeley, CA, USA, 2005. USENIX Association.

[WSSZ07] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID Semantics to the File System. *ACM Transactions on Storage (TOS)*, (2):1–42, 06 2007.

[XXSM09] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. Adaptive and scalable metadata management to support a trillion files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. ACM.

[YVCJ07] Weikuan Yu, Jeffrey Vetter, Shane R. Canon, and Song Jiang. Exploiting Lustre File Joining for Effective Collective IO. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pages 267–274, Washington, DC, USA, 2007. IEEE Computer Society.

[Zak14] Karel Zak. mount(8). `http://man7.org/linux/man-pages/man8/mount.8.html`, 07 2014. Last accessed: 2014-11.

Appendices

Appendix A.

Additional Evaluation Results

A.1. JULEA (XFS Storage Backend)

This section contains additional evaluation results regarding JULEA's data performance. Due to the performance problems that are present when using JULEA's data daemon with ext4 (see Section 6.2.3 on pages 122–125), additional measurements have been conducted with XFS to assess whether these problems are specific to ext4.

Three Connections

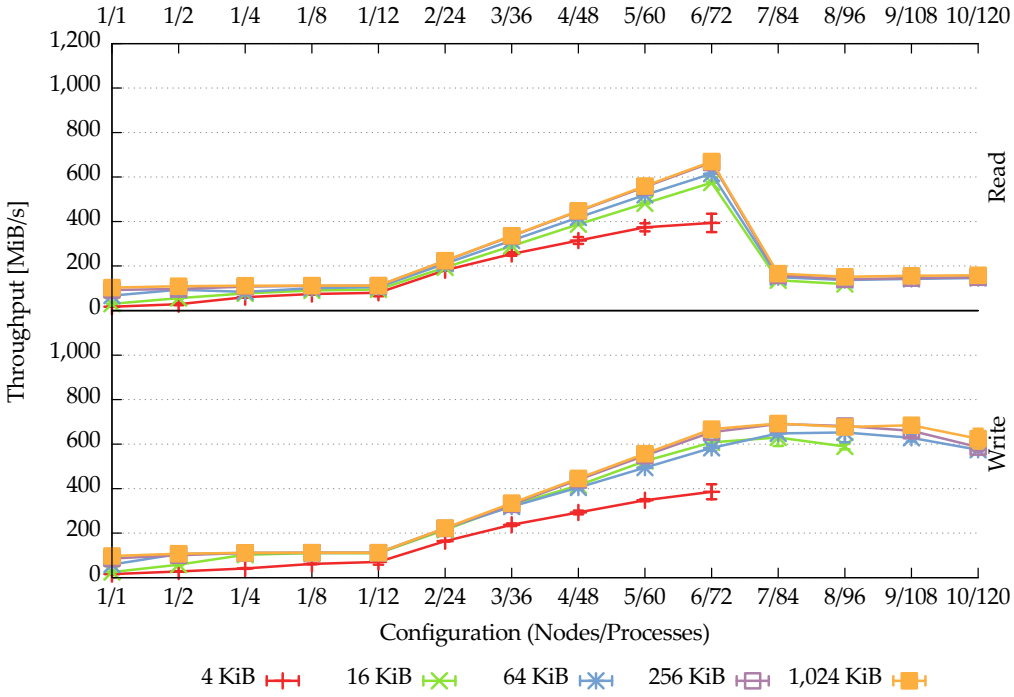


Figure A.1.: JULEA: concurrent accesses to individual items using XFS and three connections per client

Individual Items Figure A.1 on the preceding page shows JULEA's performance when using individual items via the native JULEA interface using XFS and three connections per client.

Regarding read performance, the performance curve looks very similar to its counterpart using ext4 (see Figure 6.9 on page 123). The only significant difference can be observed for the configurations using seven to ten nodes: Whereas ext4's performance decreased when using more nodes, XFS immediately drops to roughly 160 mebibytes (MiB)/s when going to seven nodes and stays at this level. Additionally, in contrast to ext4, XFS's results are almost identical regardless of the chosen block size.

Regarding write performance, it can be seen that XFS reaches almost the same throughput for all block sizes larger than 4 kibibytes (KiB); when using ext4, the block sizes of 16 KiB and 64 KiB performed significantly worse than the block sizes of 256 KiB and 1,024 KiB. When using a block size of 4 KiB, performance is almost identical to that of ext4.

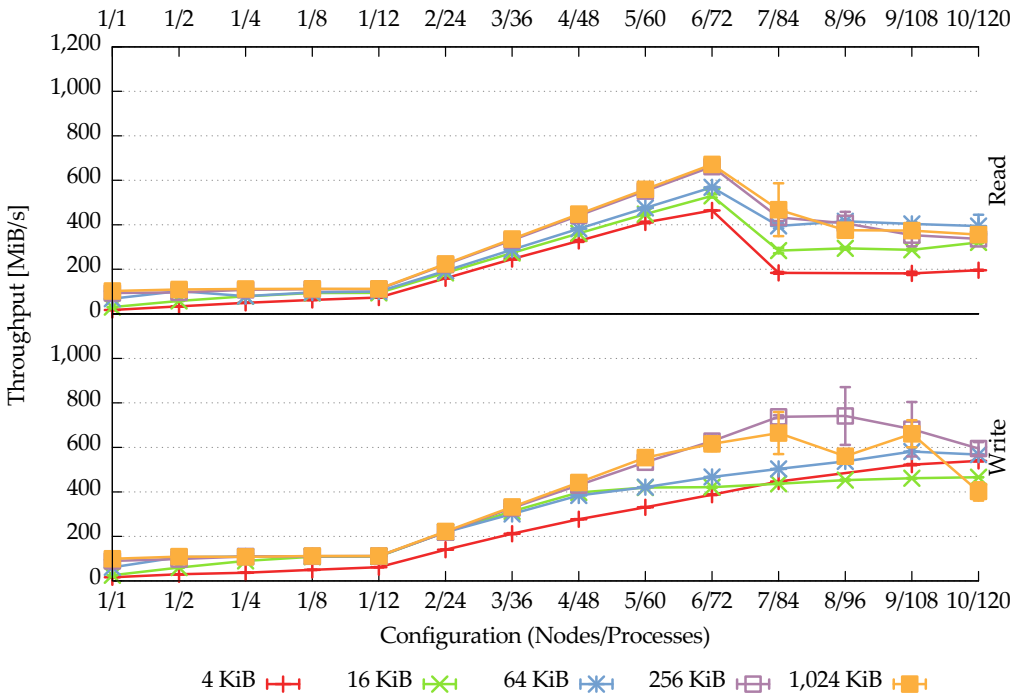


Figure A.2.: JULEA: concurrent accesses to a shared item using XFS and three connections per client

Shared Item Figure A.2 shows JULEA's performance when using a single shared item via the native JULEA interface using XFS and three connections per client.

Regarding read performance, the performance curve looks similar to that of its counterpart using ext4 (see Figure 6.10 on page 124). However, the performance drop when using more than six client nodes is less severe. Additionally, the performance is

more stable and roughly the same when using seven to ten nodes; in ext4's case, it decreased from six to eight or nine nodes, and then increased again.

Regarding write performance, XFS remains stable until six nodes are used, similar to the read case. Afterwards, performance becomes more erratic, especially for the larger block sizes of 256 KiB and 1,024 KiB. It is interesting to note that the block size of 4 KiB achieves better performance than the block size of 16 KiB when using more than seven nodes. Overall, XFS seems to handle the shared case better than ext4; it also provides better performance, especially for the smaller block sizes.

Six Connections

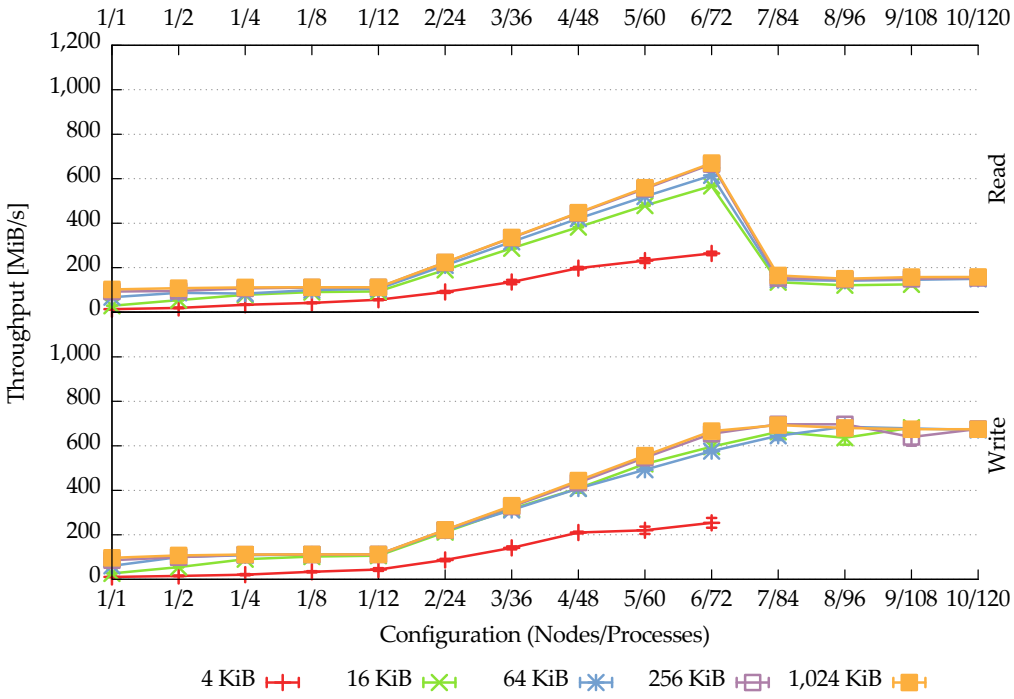


Figure A.3.: JULEA: concurrent accesses to individual items using XFS and six connections per client

Individual Items Figure A.3 shows JULEA's performance when using individual items via the native JULEA interface using XFS and six connections per client.

During the read phase, for all block sizes except for 4 KiB, the performance is mostly identical to that of its counterpart using three connections per client. When using a block size of 4 KiB, however, performance is degraded by roughly 33 %; this is likely due to the fact that the increased parallelism causes additional congestion.

During the write phase, as long as less than ten client nodes are used, the performance is roughly the same as that of its counterpart using three connections per client for all block sizes except for 4 KiB. While performance is decreased for large numbers

of nodes when using three connections per client, it is more stable when using six connections per client. When using a block size of 4 KiB, performance is reduced by approximately 35 %; like in the read case, this is likely due to additional congestion.

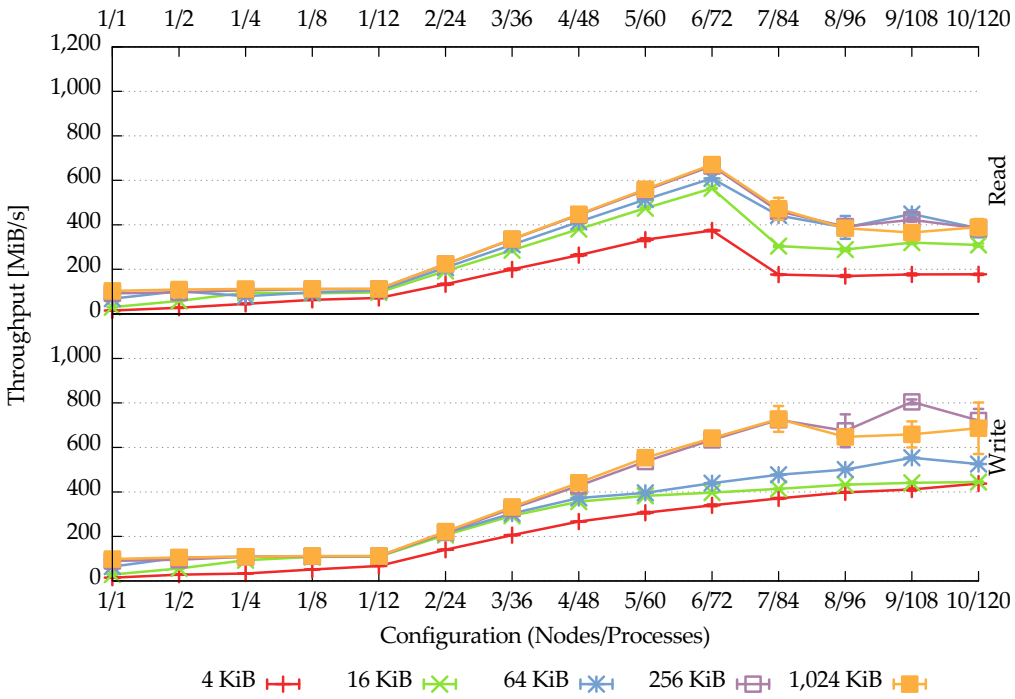


Figure A.4.: JULEA: concurrent accesses to a shared item using XFS and six connections per client

Shared Item Figure A.4 shows JULEA's performance when using a single shared item via the native JULEA interface using XFS and six connections per client.

During the read phase, performance is largely identical to that of its counterpart using three connections per client for the larger block sizes of 256 KiB and 1,024 KiB. While performance is only slightly decreased for the block sizes of 16 KiB and 64 KiB, it drops by 20 % when using a block size of 4 KiB. It is also interesting to note that the performance decrease only applies to the configurations using less than seven nodes, that is, performance is not degraded further for the congested case.

During the write phase, the performance is more stable than when using three connections per client, especially for the larger block sizes of 256 KiB and 1,024 KiB. While the performance using block sizes of 16 KiB and 64 KiB is more or less the same, it is decreased significantly when using a block size of 4 KiB. XFS's peculiar performance behavior in this case hints at inefficient handling of very small accesses.

Appendix B.

Usage Instructions

B.1. JULEA

The following instructions show how to set up JULEA from scratch.

B.1.1. Downloading Source Code and Dependencies

```
1 $ git clone https://redmine.wr.informatik.uni-hamburg.de/git/julea
```

Listing B.1: JULEA download

Listing B.1 shows how to download JULEA's source code, which is available via Git.

```
1  # Debian/Ubuntu
2  $ apt-get install libglib2.0-dev
3  $ apt-get install libfuse-dev
4  # Fedora
5  $ yum install glib2-devel
6  $ yum install fuse-devel
7
8  $ cd julea/external
9  $ ./mongodb-client.sh
10 $ ./mongodb-server.sh
11 $ ./hdtrace.sh
12 $ ./otf.sh
```

Listing B.2: JULEA dependencies

Listing B.2 shows how to install all external dependencies. GLib and FUSE[1] have to be installed using the software management provided by the operating system (OS); all other dependencies can be installed using the provided scripts that automatically

[1] Filesystem in Userspace

download and compile the source code locally. Except for GLib and MongoDB, all dependencies are optional and can be omitted.

B.1.2. Configuring, Compiling and Installing

```
1 $ cd julea
2 $ ./waf configure --prefix=${PWD}/install --debug
```

Listing B.3: JULEA configuration

```
1  Setting top to                           : ${PWD}
2  Setting out to                           : ${PWD}/build
3  Checking for 'gcc' (c compiler)          : /usr/bin/gcc
4  Checking for program pkg-config          : /usr/bin/pkg-config
5  Checking for 'gio-2.0' >= 2.32           : yes
6  Checking for 'glib-2.0' >= 2.32          : yes
7  Checking for 'gmodule-2.0' >= 2.32       : yes
8  Checking for 'gobject-2.0' >= 2.32       : yes
9  Checking for 'gthread-2.0' >= 2.32       : yes
10 Checking for 'fuse'                      : yes
11 Checking for header bson.h               : yes
12 Checking for header mongo.h              : yes
13 Checking for header hdTrace.h            : yes
14 Checking for header otf.h                : yes
15 Checking for stat.st_mtim.tv_nsec        : yes
16 'configure' finished successfully (0.543s)
```

Listing B.4: JULEA configuration output

Listing B.3 shows how to configure JULEA. It is possible to specify a custom installation prefix using the `--prefix` option. During development, it is strongly recommended to enable debug mode using the `--debug` option. The output produced by the configuration step should look similar to Listing B.4.

```
1 $ cd julea
2 $ ./waf
3 $ ./waf install
```

Listing B.5: JULEA compilation and installation

Listing B.5 shows how to compile and install JULEA once all dependencies have been installed and the project has been configured.

```
1 $ cd julea
2 $ ./waf environment > env.sh
3 $ . ./env.sh
4 $ julea-config --local --data=$(hostname) --metadata=$(hostname)
    ↪ --storage-backend=posix --storage-path=/tmp/julea-$(id -nu)
```

Listing B.6: JULEA configuration file

Listing B.6 shows how to create a configuration file for JULEA. The `environment` command allows exporting all necessary environment variables to be able to use an installation in a custom path.

B.1.3. Tests and Benchmarks

```
1 $ cd julea
2 $ ./waf test
3 $ ./waf benchmark
```

Listing B.7: JULEA tests and benchmarks

Listing B.7 shows how to run JULEA's basic tests and benchmarks; they are integrated into JULEA's build system and can be called through `waf`.

B.1.4. Documentation

```
1 $ cd julea
2 $ doxygen
3 $ xdg-open html/index.html
```

Listing B.8: JULEA documentation

Listing B.8 shows how to generate and view JULEA's documentation.

B.2. Benchmarks

The following instructions show how to set up the benchmarks used in Chapter 6.

B.2.1. Downloading Source Code and Dependencies

```
$ git clone \
  https://redmine.wr.informatik.uni-hamburg.de/git/julea-benchmarks
```

Listing B.9: Benchmarks download

Listing B.9 shows how to download the benchmarks' source code, which is also available via Git.

```
# Debian/Ubuntu
$ apt-get install libglib2.0-dev
$ apt-get install libopenmpi-dev openmpi-bin
# Fedora
$ yum install glib2-devel
$ yum install openmpi
$ module add mpi/openmpi-$(arch)

$ cd julea-benchmarks/external
$ ./mongodb-client.sh
$ ./orangefs.sh
```

Listing B.10: Benchmarks dependencies

Listing B.10 shows how to install all external dependencies. While GLib and MPI[2] have to be installed using the OS's software management, all other dependencies can be installed using the provided scripts; again, they automatically download and compile the source code locally. All dependencies except for GLib are optional and can be omitted.

B.2.2. Configuring, Compiling and Installing

```
$ cd julea-benchmarks
$ . ${JULEA}/env.sh
$ ./waf configure --prefix=${PWD}/install --debug
```

Listing B.11: Benchmarks configuration

[2] Message Passing Interface

```
Setting top to                           : ${PWD}
Setting out to                           : ${PWD}/build
Checking for 'gcc' (c compiler)          : /usr/bin/gcc
Checking for program mpicc               :
   ↪ /usr/lib64/openmpi/bin/mpicc
Checking for program pkg-config          : /usr/bin/pkg-config
Checking for 'glib-2.0' >= 2.32          : yes
Checking for 'openssl'                   : yes
Checking for 'julea'                     : yes
Checking for 'lexos'                     : yes
Checking for header bson.h               : yes
Checking for header mongo.h              : yes
Checking for header math.h               : yes
Checking for header pthread.h            : yes
Checking for header pvfs2.h              : yes
Checking for header mpi.h                : yes
'configure' finished successfully (0.479s)
```

Listing B.12: Benchmarks configuration output

Listing B.11 on the facing page shows how to configure the benchmarks. A custom installation prefix can be specified using the `--prefix` option. It is strongly recommended to enable debug mode during development; this can be accomplished using the `--debug` option. The configuration step should produce output similar to that shown in Listing B.12.

```
$ cd julea-benchmarks
$ ./waf
$ ./waf install
```

Listing B.13: Benchmarks compilation and installation

As soon as all dependencies have been installed and the project has been configured, the benchmarks can be compiled and installed as shown in Listing B.13.

B.3. Lustre

The following instructions show how to configure Lustre's distributed namespace (DNE) as explained in Section 2.4.1 on pages 32–35.

B.3.1. Distributed Namespace

```
1 $ lfs mkdir --index 0 /lustre/home
2 $ lfs mkdir --index 1 /lustre/scratch
```

Listing B.14: Setting up Lustre's Distributed Namespace

Listing B.14 shows the necessary commands to set up DNE in such a way that metadata accesses inside `/lustre/home` are handled by meta data target (MDT) number `0`, while metadata accesses within `/lustre/scratch` are served by MDT number `1`.

Appendix C.

Code Examples

To give a rough idea of the structure and usability of the major input/output (I/O) interfaces, the following code examples all implement the same basic application:

1. A new file or item is created.
2. 42 bytes of data are written to the file or item.
3. The file or item's metadata is queried.
4. 42 bytes of data are read from the file or item.
5. The file or item is closed.
6. The file or item is deleted.

The application has been implemented using the POSIX[1], MPI-IO and JULEA interfaces to be able to compare them to each other. The following sections contain code examples using the interfaces' respective functionality including error handling as well as detailed descriptions of the different implementations.

[1] Portable Operating System Interface

C.1. POSIX

```c
#define _POSIX_C_SOURCE 200809L

#include <fcntl.h>
#include <inttypes.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int
main (int argc, char const* argv[])
{
    struct stat stat_buf;
    char data[42];
    int fd;
    int ret;
    ssize_t nbytes;

    memset(data, 42, sizeof(data));
    fd = open("/tmp/posix", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR |
       ↪ S_IWUSR);

    if (fd == -1)
    {
        goto error;
    }

    nbytes = pwrite(fd, data, sizeof(data), 0);

    if (nbytes != sizeof(data))
    {
        goto error;
    }

    ret = fstat(fd, &stat_buf);

    if (ret == -1)
    {
```

```
39          goto error;
40      }
41
42      nbytes = pread(fd, data, sizeof(data), 0);
43
44      if (nbytes != sizeof(data))
45      {
46          goto error;
47      }
48
49      printf("File size is %" PRIdMAX " bytes.\n",
           ↪ (uintmax_t)stat_buf.st_size);
50      printf("File was last modified at %" PRIdMAX ".\n",
           ↪ (uintmax_t)stat_buf.st_mtime);
51
52      ret = close(fd);
53
54      if (ret == -1)
55      {
56          goto error;
57      }
58
59      ret = unlink("/tmp/posix");
60
61      if (ret == -1)
62      {
63          goto error;
64      }
65
66      return 0;
67
68  error:
69      return 1;
70  }
```

Listing C.1: POSIX example

Listing C.1 on page 192 shows the application as implemented using the POSIX interface. First, the most recent POSIX standard is enabled by defining the appropriate preprocessor macro (line 1). Afterwards, all necessary headers are included (lines 3–9); as can be seen, the POSIX interface's functionality is spread over a multitude of different headers.

As the actual application's first step, the file `/tmp/posix` is created using the `open` function (line 21). While specifying the `O_CREAT` flag causes the file to be created, the `O_TRUNC` flag indicates that the file should be truncated to size `0` if it already exists. Additionally, the file is made readable and writable only by the current user by specifying the `S_IRUSR` and `S_IWUSR` permission bits. The `open` function's success is checked by comparing its returned file descriptor to `-1` (lines 23–26); a return value of `-1` traditionally indicates that an error has happened.

Afterwards, the data is written to the file using the `pwrite` function at offset `0` (line 28). The function's success is checked by comparing the returned number of written bytes to the data's size (lines 30-33).

As the next step, the file's metadata is queried using the `fstat` function (line 35); it stores the metadata of an opened file into a `stat` structure. `fstat`'s success is then checked analogously to `open` (lines 37–40).

Reading the data is accomplished using the `pread` function (line 42); it accepts the same parameters as the `pwrite` function. Again, its success is checked by comparing the number of read bytes to the data's size (lines 44–47).

Afterwards, the file descriptor is closed using the `close` function (line 52). This operation can potentially fail and will return a value of `-1` in that case (lines 54–57).

Finally, the file is deleted using the `unlink` function (line 59). POSIX does not provide a way to delete a file based on an open file descriptor; therefore, the file's path is passed to the function. Again, its success is checked by comparing its return value to `-1` (lines 61–64).

C.2. MPI-IO

```
1  #include <mpi.h>
2
3  #include <inttypes.h>
4  #include <stdio.h>
5  #include <string.h>
6
7  int
8  main (int argc, char const* argv[])
9  {
10     MPI_File fh;
11     MPI_Offset size;
12     MPI_Status status;
13     char data[42];
14     int ret;
15     int nbytes;
16
17     MPI_Init(&argc, (char***)&argv);
18
19     memset(data, 42, sizeof(data));
20     ret = MPI_File_open(MPI_COMM_WORLD, "/tmp/mpi-io",
          ↪ MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh);
21
22     if (ret != MPI_SUCCESS)
23     {
24         goto error;
25     }
26
27     MPI_File_write_at(fh, 0, data, sizeof(data), MPI_BYTE, &status);
28     ret = MPI_Get_count(&status, MPI_BYTE, &nbytes);
29
30     if (ret != MPI_SUCCESS || nbytes != sizeof(data))
31     {
32         goto error;
33     }
34
35     ret = MPI_File_get_size(fh, &size);
36
37     if (ret != MPI_SUCCESS)
38     {
```

```
39          goto error;
40      }
41
42      MPI_File_read_at(fh, 0, data, sizeof(data), MPI_BYTE, &status);
43      ret = MPI_Get_count(&status, MPI_BYTE, &nbytes);
44
45      if (ret != MPI_SUCCESS || nbytes != sizeof(data))
46      {
47          goto error;
48      }
49
50      printf("File size is %" PRIdMAX " bytes.\n", (uintmax_t)size);
51
52      ret = MPI_File_close(&fh);
53
54      if (ret != MPI_SUCCESS)
55      {
56          goto error;
57      }
58
59      ret = MPI_File_delete("/tmp/mpi-io", MPI_INFO_NULL);
60
61      if (ret != MPI_SUCCESS)
62      {
63          goto error;
64      }
65
66      MPI_Finalize();
67
68      return 0;
69
70  error:
71      MPI_Finalize();
72
73      return 1;
74  }
```

Listing C.2: MPI-IO example

Listing C.2 on page 195 shows the application as implemented using the MPI-IO interface. First, MPI[2]'s single header `mpi.h` is included to make all functionality available (line 1).

Before being able to use any MPI functionality, it is necessary to initialize the MPI library using the `MPI_Init` function (line 17). Applications that use threads have to call the `MPI_Init_thread` function instead.

Afterwards, the `/tmp/mpi-io` file is created using the `MPI_File_open` function (line 20). Its `MPI_MODE_RDWR` and `MPI_MODE_CREATE` flags are analogous to POSIX's `O_RDWR` and `O_CREAT` flags, respectively. In contrast to POSIX, MPI-IO does not allow the file to be truncated if it exists already. The function's success can be checked using its return value (lines 22–25); MPI specifies `MPI_SUCCESS` as well as several error codes for this purpose.

The data is then written to the file using the `MPI_File_write_at` function (line 27). It works in a similar fashion as POSIX's `pwrite` function and accepts an offset. To signify that the data is an array of bytes, the `MPI_BYTE` data type is used. The `MPI_Get_count` function is used to check the number of written bytes (line 28). This information is used to check whether the write operation was successful (lines 30–33).

In contrast to POSIX, MPI only allows querying a limited subset of metadata. For instance, it is not possible to get the last modification time. Therefore, only the file's size is checked using the `MPI_File_get_size` function (line 35). Its success can be checked using its return value (lines 37–40).

Afterwards, the data is read again using the `MPI_File_read_at` function (line 42). It takes the same arguments as the `MPI_File_write_at` function and works like POSIX's `pread` function. Again, the function's success is checked using the number of read bytes as returned by the `MPI_Get_count` function (line 43–48).

The file is then closed using the `MPI_File_close` function (line 52). Similar to POSIX, closing the file can return an error that is checked using the return value (lines 54–57).

Finally, the file is deleted using the `MPI_File_delete` function (line 59). Like POSIX, it is necessary to specify the file name instead of being able to use an existing file handle to delete the file.

Before terminating the application, it is necessary to finalize the MPI library using the `MPI_Finalize` function (line 66).

[2] Message Passing Interface

C.3. JULEA

```
#include <julea.h>

#include <stdio.h>
#include <string.h>

int
main (int argc, char const* argv[])
{
    JBatch* batch;
    JItem* item;
    JURI* uri;
    gboolean ret;
    guint64 nbytes;
    char data[42];

    j_init();

    memset(data, 42, sizeof(data));
    batch = j_batch_new_for_template(J_SEMANTICS_TEMPLATE_DEFAULT);

    uri = j_uri_new("julea://tmp/tmp/julea");
    ret = j_uri_create(uri, TRUE, NULL);

    if (!ret)
    {
        goto error;
    }

    item = j_uri_get_item(uri);
    j_item_write(item, data, sizeof(data), 0, &nbytes, batch);
    j_batch_execute(batch);

    if (nbytes != sizeof(data))
    {
        goto error;
    }

    j_item_get_status(item, J_ITEM_STATUS_ALL, batch);
    ret = j_batch_execute(batch);
```

```
40
41      if (!ret)
42      {
43          goto error;
44      }
45
46      j_item_read(item, data, sizeof(data), 0, &nbytes, batch);
47      j_batch_execute(batch);
48
49      if (nbytes != sizeof(data))
50      {
51          goto error;
52      }
53
54      printf("File size is %" G_GUINT64_FORMAT " bytes.\n",
           ↪ j_item_get_size(item));
55      printf("File was last modified at %" G_GINT64_FORMAT ".\n",
           ↪ j_item_get_modification_time(item));
56
57      j_collection_delete_item(j_uri_get_collection(uri), item,
           ↪ batch);
58      ret = j_batch_execute(batch);
59
60      if (!ret)
61      {
62          goto error;
63      }
64
65      j_uri_free(uri);
66      j_batch_unref(batch);
67
68      j_fini();
69
70      return 0;
71
72  error:
73      j_fini();
74
75      return 1;
76  }
```

Listing C.3: JULEA example

Listing C.3 on page 198 shows the application as implemented using the JULEA interface. First, JULEA's single header `julea.h` is included (line 1). This takes care of making available all of JULEA's functionality.

Before any JULEA functionality can be used, the library has to be initialized using the `j_init` function (line 16). A batch using the default semantics is created to be able to execute any operations (line 19).

Afterwards, a JULEA uniform resource identifier (URI) is created to refer to the `tmp` store, the `tmp` collection and the `julea` item (line 21). URIs provide a convenient way for application developers to use JULEA's interface without performing too many operations manually. Afterwards, the item and its parent collection and store are created using the `j_uri_create` function (line 22). Its success is checked by means of the returned boolean value (lines 24–27).

Afterwards, the data is written to the item by scheduling a write operation using the `j_item_write` function (line 30) and executing the batch (line 31). The write operation's success can be checked by comparing the number of written bytes to the data's size (lines 33–36).

The item's metadata is queried by scheduling a get status operation using the `j_item_get_status` function (line 38) and executing the batch (line 39). Similar to POSIX, JULEA provides a single function to query an item's metadata; JULEA allows specifying which metadata should be returned, however. By specifying the `J_ITEM_STATUS_ALL` flag, all metadata is returned. Again, the `j_batch_execute` function's boolean return value is used to check the operation's success (lines 41–44).

Reading the data is performed by scheduling a read operation using the `j_item_read` function (line 46) and then executing the batch (line 47). Its success can be checked by comparing the number of read bytes to the data's size (lines 49–52).

Finally, the item is deleted by scheduling its deletion using the `j_collection_delete_item` function (line 57) and executing the batch (line 58). Again, its success is checked using the `j_batch_execute` function's boolean return value (lines 60–63).

In contrast to POSIX and MPI-IO, JULEA does not provide a function to explicitly close an item. Instead, the item is closed implicitly by freeing the URI (line 65).

Before terminating the application, the JULEA library has to be finalized using the `j_fini` function (line 68).

Index

List of Acronyms

ABI ..... Application binary interface
ACID ..... Atomicity, consistency, isolation and durability
ACL ..... Access control list
ADIO ..... Abstract-Device Interface for I/O
ADIOS ..... Adaptable IO System
Amazon S3 ..... Amazon Simple Storage Service
API ..... Application programming interface
ASCII ..... American Standard Code for Information Interchange
BDB ..... Berkeley DB
BSON ..... Binary JavaScript Object Notation
btrfs ..... B-tree file system
CPU ..... Central processing unit
DMU ..... Data management unit
DNE ..... Distributed namespace
EBOFS ..... Extent and B-tree-based Object File System
ECC ..... Error-correcting code
EOFS ..... European Open File Systems
FIFO ..... First in, first out
FLOPS ..... Floating-point operations per second
FTP ..... File Transfer Protocol
FUSE ..... Filesystem in Userspace
GB ..... Gigabyte (10^9 bytes)
Gbit ..... Gigabit (10^9 bits)
GETM ..... General Estuarine Transport Model
GiB ..... Gibibyte (2^{30} bytes)
GPFS ..... General Parallel File System
GPL ..... GNU General Public License
HDD ..... Hard disk drive
HDF ..... Hierarchical Data Format
HFS+ ..... Hierarchical File System Plus
HPC ..... High performance computing
HTTP ..... Hypertext Transfer Protocol
I/O ..... Input/output
ID ..... Identifier
IOPS ..... Input/output operations per second

IP ..... Internet Protocol
IPC ..... Inter-process communication
IPS ..... Instructions per second
JSON ..... JavaScript Object Notation
KB ..... Kilobyte (10^3 bytes)
KiB ..... Kibibyte (2^{10} bytes)
LEXOS ..... Low-Level Extent-Based Object Store
MB ..... Megabyte (10^6 bytes)
Mbit ..... Megabit (10^6 bits)
MDS ..... Meta data server
MDT ..... Meta data target
MiB ..... Mebibyte (2^{20} bytes)
MPI ..... Message Passing Interface
NetCDF ..... Network Common Data Form
NFS ..... Network File System
NIC ..... Network interface card
NTFS ..... New Technology File System
OpenSFS ..... Open Scalable File Systems
OS ..... Operating system
OSS ..... Object storage server
OST ..... Object storage target
OTF ..... Open Trace Format
PB ..... Petabyte (10^{15} bytes)
PDE ..... Partial differential equation
POSIX ..... Portable Operating System Interface
PVFS ..... Parallel Virtual File System
RAID ..... Redundant array of independent disks
RAM ..... Random access memory
RPM ..... Revolutions per minute
RTT ..... Round-trip time
SAN ..... Storage area network
SSD ..... Solid state drive
SSH ..... Secure Shell
TB ..... Terabyte (10^{12} bytes)
TCP ..... Transmission Control Protocol
URI ..... Uniform resource identifier
URL ..... Uniform resource locator
VCS ..... Version control system
VFS ..... Virtual file system (or virtual filesystem switch)
XML ..... Extensible Markup Language
ZFS ..... Zettabyte File System

List of Figures

List of Listings

List of Tables

Eidesstattliche Versicherung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort, Datum	Unterschrift