

Compiler-assisted distribution of OpenMP code for improved scalability

Dissertation zur Erlangung des Doktorgrades
an der Fakultät für Mathematik, Informatik und
Naturwissenschaften
der Universität Hamburg
eingereicht beim Fachbereich Informatik von
Jannek Squar

Hamburg, Juni 2023

Gutachter:
Prof. Dr. Thomas Ludwig
Prof. Dr. Michael Kuhn
Prof. Dr. Matthias Müller

Datum der Disputation: 04.09.2023

Abstract

High performance computing is a complex field, with many homogeneous and heterogeneous hardware architectures, and numerous programming paradigms, libraries and compilers. OpenMP and netCDF are relatively widely used in Earth system research because they are comparatively easy to learn and yet can exploit the potential of a single compute node. However, Earth system scientists without the appropriate training may find it difficult to run their application on a distributed HPC infrastructure. As Earth system applications generally benefit from being able to run on large input problems, they would particularly benefit from HPC features such as process parallelisation, data reduction or parallel input and output. However, their use is not trivial and requires a lot of experience and work. In order to support them, this dissertation develops a tool that allows them to quickly apply useful HPC frameworks without having to deal with the implementation first, by automatically incorporating the necessary code changes into their application.

Different approaches are considered that can be used to automatically traverse, analyse and transform code. Based on this, the design of a new tool is presented: CATO is based on the LLVM framework and uses its rich API for automatic code analysis and transformation to add new features to an application. CATO analyses the existing OpenMP kernels of an application and transforms them into equivalent MPI code so that they can be executed on distributed memory systems. If the application also uses netCDF, it can be automatically adapted to use the data compression and parallel input/output features of the netCDF library. In this way, the user can test the effect of the HPC concepts mentioned without having to adapt his application.

The evaluation of CATO is based on a PDE solver as well as on netCDF micro-benchmarks to examine the functionality and performance of the modified applications. The tests showed that there was no runtime performance benefit due to the additional overhead caused by CATO. However, it can now use the aggregated memory of multiple nodes and the memory consumption per process is optimised. In addition, the memory footprint as well as the runtime of the I/O phase of the modified application can be significantly improved by using parallel I/O. Through the automatic integration of netCDF compression algorithms, the user can also decide at runtime to compress his output, which can also significantly reduce the memory consumption in the file system.

Zusammenfassung

High Performance Computing ist ein komplexes Gebiet, in dem es viele homogene aber auch heterogene Hardwarearchitekturen gibt, für deren Nutzung zahlreiche Programmierparadigmen, Bibliotheken und Compiler zur Verfügung stehen. OpenMP und netCDF sind in der Erdsystemforschung relativ weit verbreitet, da sie vergleichsweise einfach zu erlernen sind und dennoch das Potential eines einzelnen Rechenknotens ausschöpfen können. Für Fachanwender aus der Erdsystemforschung ohne entsprechende Ausbildung können sich jedoch Schwierigkeiten ergeben, wenn sie ihre Anwendung auf einer verteilten HPC-Infrastruktur ausführen wollen. Da Anwendungen der Erdsystemforschung generell davon profitieren, wenn sie auf großen Eingabeproblemen ausgeführt werden können, würden sie besonders von HPC-Features wie Prozessparallelisierung, Datenreduktion oder paralleler Ein- und Ausgabe profitieren. Deren Nutzung ist jedoch nicht trivial und erfordert viel Erfahrung und Arbeit. Um die Anwender dabei zu unterstützen, wird in dieser Dissertation ein Werkzeug entworfen, das es erlaubt, nützliche HPC-Frameworks schnell zu nutzen, ohne sich mit der Implementierung beschäftigen zu müssen, indem die notwendigen Code-Änderungen automatisch in die Anwendung integriert werden.

Es werden verschiedene Ansätze untersucht, die verwendet werden können, um Code automatisch zu traversieren, zu analysieren und zu transformieren. Darauf aufbauend wird das Design eines neuen Werkzeugs vorgestellt: CATO basiert auf dem LLVM-Framework und nutzt dessen umfangreiche API zur automatischen Code-Analyse und -Transformation, um neue Features in eine Anwendung zu integrieren. CATO analysiert die vorhandenen OpenMP-Kernel einer Anwendung und transformiert sie in äquivalenten MPI-Code, so dass sie auf Systemen mit verteiltem Speicher ausgeführt werden können. Wenn die Anwendung zusätzlich netCDF verwendet, kann sie automatisch so angepasst werden, dass sie auch die Datenkompression und parallele Ein- und Ausgabe der netCDF-Bibliothek nutzt. So kann der Anwender, ohne seine Anwendung selbst anpassen zu müssen, die Auswirkungen der genannten HPC-Konzepte testen.

Die Evaluierung von CATO wird auf Basis eines PDE-Tools und netCDF Micro-Benchmarks durchgeführt, um die Funktionalität und Performance der modifizierten Anwendungen untersuchen zu können. Die Untersuchungen haben gezeigt, dass sich durch den von CATO verursachten Overhead kein Vorteil in der Laufzeitperformance ergibt. Im Gegenzug kann jedoch der Speicher mehrerer Knoten genutzt und der Speicherverbrauch pro Prozess optimiert werden. Durch parallele I/O kann auch der Speicher-Footprint sowie die Laufzeit der I/O-Phase der modifizierten Anwendung deutlich verbessert werden. Durch das automatische Einbinden von netCDF Kompressionsalgorithmen kann der Benutzer zur Laufzeit entscheiden, ob er seine Daten komprimieren möchte, was ebenfalls den Speicherverbrauch im Dateisystem reduzieren kann.

Acknowledgements

First of all, I would like to thank Prof. Dr. Thomas Ludwig and Prof. Dr. Michael Kuhn for their supervision. They awakened my enthusiasm for supercomputing while I was still a student and continued to nurture it in the years that followed. They have accompanied and supported me for many years.

I would also like to thank Anna Fuchs for her support so that I could concentrate on my thesis. Special thanks go to Michael Kuhn, Anna Fuchs and Thomas Ludwig for proofreading my dissertation and for the many exciting and amusing discussions.

Over the years many students have contributed to the development of CATO and I would like to thank them: Michael Blesel and Tim Jammer for their work on the prototype, Niclas Schroeter for his work on the IO component and AGSearch and Liviana Franke for the development of LLVis.

Finally, I would like to thank my family for supporting me all these years and always believing in me.

‘Make it work, make it right, make it fast’ – Kent Beck

Contents

1. Introduction	15
1.1. Perspectives	15
1.1.1. Computer Science	16
1.1.2. Natural Science	18
1.2. Problem Statement	21
1.2.1. Research Question 1: Distributed Computing	21
1.2.2. Research Question 2: Parallel Input/Output	22
1.2.3. Research Question 3: Chunking and Compression	23
1.2.4. Research Question 4: User Support	25
1.3. Approach	25
1.3.1. Source Transformation	25
1.3.2. Target Audience	26
1.4. Outline	29
1.5. Summary	30
2. Surveying Techniques for Code Insertion	31
2.1. Specification	32
2.2. Code Transformation	33
2.2.1. User Interaction	33
2.2.2. Code Interaction	35
2.3. Compiler-Assisted Approach	38
2.3.1. LLVM Component Sequence	38
2.4. Pass Design	49
2.4.1. Code Transformation	51
2.5. Summary	53
3. Background	55
3.1. Source Transformation Techniques	55
3.1.1. Compiler	56
3.2. Parallelisation Techniques	58
3.2.1. Shared Memory	60
3.2.2. Distributed Memory	61
3.3. Input/Output	65
3.3.1. Parallel Input/Output	67
3.3.2. Compression	68
3.4. Summary	74

4. Tool Design	77
4.1. Component: Memory Handling	80
4.1.1. Communication Patterns	80
4.1.2. Distribute Memory of Stencil Pattern	84
4.1.3. Optimal Memory Consumption	88
4.1.4. Memory Model	90
4.1.5. Automatic Code Recognition	99
4.1.6. Equivalence Classes	100
4.2. Component: IO Handling	101
4.2.1. Parallel IO	101
4.2.2. Compression	104
4.3. Component: Feedback	106
4.3.1. Decompilation	107
4.3.2. Sanity Checks	113
4.3.3. Performance Metrics	114
4.3.4. Providing User Guidance	114
4.4. General CATO Workflow	115
4.5. Summary	115
5. Mapping onto LLVM	117
5.1. Using the LLVM Infrastructure	117
5.1.1. Dissecting IR	119
5.1.2. Performing Code Transformation	121
5.1.3. Equivalence Classes	124
5.2. Memory Handling	126
5.2.1. OpenMP Code Handling	128
5.2.2. Semantical Equivalence	131
5.3. Input/Output Handling	131
5.3.1. NetCDF: Parallel Input/Output	132
5.3.2. NetCDF: Compression	134
5.3.3. Semantical Equivalence	135
5.4. Providing Feedback	136
5.4.1. Trace Code Changes	136
5.4.2. Hardening Code Quality	137
5.4.3. Metric Collection	137
5.4.4. Textual User Support	138
5.5. Summary	138
6. Related Work	141
6.1. Virtual Shared Memory	141
6.1.1. Single System Image	141
6.1.2. Partitioned Global Address Space	142
6.1.3. Distribution of OpenMP	142

6.2.	Direct Code Parallelisation	144
6.2.1.	Parallel Programming Languages	144
6.2.2.	Parallel Libraries	144
6.2.3.	Domain Specific Languages	145
6.2.4.	Tool-Assisted Parallelisation	146
6.3.	Transparent Compression	148
6.4.	Summary	149
7.	Evaluation	151
7.1.	Speedup Limitations	151
7.1.1.	Fixed-Size Speedup	152
7.1.2.	Fixed-Time Speedup	153
7.2.	Test Environment	155
7.2.1.	Current State of Functionality	156
7.3.	Evaluation of CATO	157
7.3.1.	CATO Component: Memory Distribution	157
7.3.2.	Runtime Influences	162
7.3.3.	CATO Component: Input/Output	169
7.3.4.	CATO Component: Feedback	182
7.3.5.	Influence on Compilation Workflow	183
7.4.	Summary	184
8.	Conclusion	187
8.1.	Research Questions	187
8.1.1.	Distributed Computing	187
8.1.2.	Parallel Input/Output	188
8.1.3.	Chunking and Compression	189
8.1.4.	User Support	190
8.2.	Future Work	191
8.2.1.	Runtime Performance	191
8.2.2.	Feature Coverage	192
8.2.3.	Usability Improvement	192
8.3.	Summary	193
	Bibliography	195
	Appendices	217
A.	Appendix	219
A.1.	LLVM Visualisation Tools	219
	Glossary	225

1. Introduction

Computational simulation and data-driven science are the third and fourth paradigms of science, after theory and experiment. To achieve higher performance in the last two paradigms, *High Performance Computing* (HPC) systems are inevitable: They offer better *Input/Output* (I/O) and computing performance and are equipped with large memory and storage capacity. However, they come at the price of making the hardware mix quite complex and heterogeneous. The increasing complexity of these experiments and simulations, as well as the systems they run on, are forcing scientists to acquire even more computational skills (Agrawal & Choudhary, 2016). However, especially in HPC, they are usually still underrepresented and inadequately taught in materials science courses at universities (MacLachlan et al., 2020). Staying updated with the latest advancements in computing and fully capitalizing on them can be a challenging task. Especially for domain scientists who are more experienced in their research domain than in computing. They cannot be expected to know all the relevant frameworks, or be aware of the specific situations where each framework is best suited for their particular needs. Providing support would enable them to get more out of their research applications (MacLachlan et al., 2020).

There are ways to help users select and customise the right frameworks. Large data centres have dedicated user support departments that can spend up to half a year on intensive code optimisation of an application, or provide specialised training to their user group. Such staff can cost the employer around 80 000 €/yr/person in Germany¹. Another approach is automatic tool support, which offers a supplementary and easy way to get to grips with complex functions without long and specialised training. Since many scientists actively seek assistance, possess specific ideas and are willing to learn independently, their own code expertise coupled with tool-assisted optimisations could already lead to success. Automatic optimisations often cannot achieve optimal performance but provide benefits like fast prototyping for a wide range of users. This is especially useful when gaining a comprehensive understanding of existing techniques and get an impression of their potential benefits for one's own purposes.

1.1. Perspectives

The following two Sections 1.1.1 and 1.1.2 highlight different facets, presenting two contrasting perspectives on the same topic. From one perspective, computer science provides a set of tools to develop highly optimised code that can take advantage of various HPC techniques. On the other hand, the physical sciences construct simulation

¹Data has been provided via an interview at *Deutsches Klimarechenzentrum* (DKRZ)

models to represent and emulate processes. When these two domains converge, their collective synergy drives their capabilities beyond what they can achieve individually.

1.1.1. Computer Science

HPC is essential for performing simulations with high demands on computing power and memory or storage capacity. It serves as a valuable tool for tackling a wide range of numerical problems across various application domains: Complex problems in sustainable agriculture, efficient energy networks, artificial intelligence, molecular medicine and materials research are just a few of the use cases that benefit from HPC (The European Technology Platform For High-Performance Computing, 2018).

‘Supercomputing allows us to understand the past, to control the present, and in a limited number of cases to predict the future’ (Sterling et al., 2017)

Typically, applications can be transitioned from a simple development environment to an HPC environment. However, without modifications, only a small portion of the immense potential offered by an HPC system can be utilised. This limitation arises from the complex nature of the constraints and the methodologies required to overcome them, which are exclusive to the field of computer science. Today’s HPC systems consist of several components:

- Compute nodes containing multiple CPUs, each having multiple cores, and sometimes accelerators (e.g. GPU, *Field-Programmable Gate Array* (FPGA), *Data Processing Unit* (DPU)).
- I/O nodes, which handle the data management on storage (usually compute nodes are disk-less or only have a small one to keep their *Operating System* (OS) or to use as a burst buffer).
- Offloading interconnect (e.g. InfiniBand, Fujitsu Tofu, Cray Slingshot) to move data between nodes.

To achieve higher peak performance per node, the trend is towards increasing the number of CPU cores rather than increasing CPU clock speeds due to power and thermal constraints (Esmailzadeh et al., 2012). Figure 1.1 shows the relative stagnation of the clock rate and the rise of many-core server CPUs beginning in 2004. In fact, according to the *Top500* list, the first dual-core systems appeared as early as 2002 (there are 46 dual-core systems on the *June 2002* list, whereas there were none on the *November 2001* list). It is therefore inevitable to use parallelisation techniques on shared memory to fully exploit the power of a node. One prominent solution is OpenMP, which allows concurrent threads to run on shared memory within a single node. Due to OpenMP’s compiler-based directive approach, it typically requires minimal modifications to the source code and allows incremental parallelisation. Consequently, this significantly reduces the obstacles

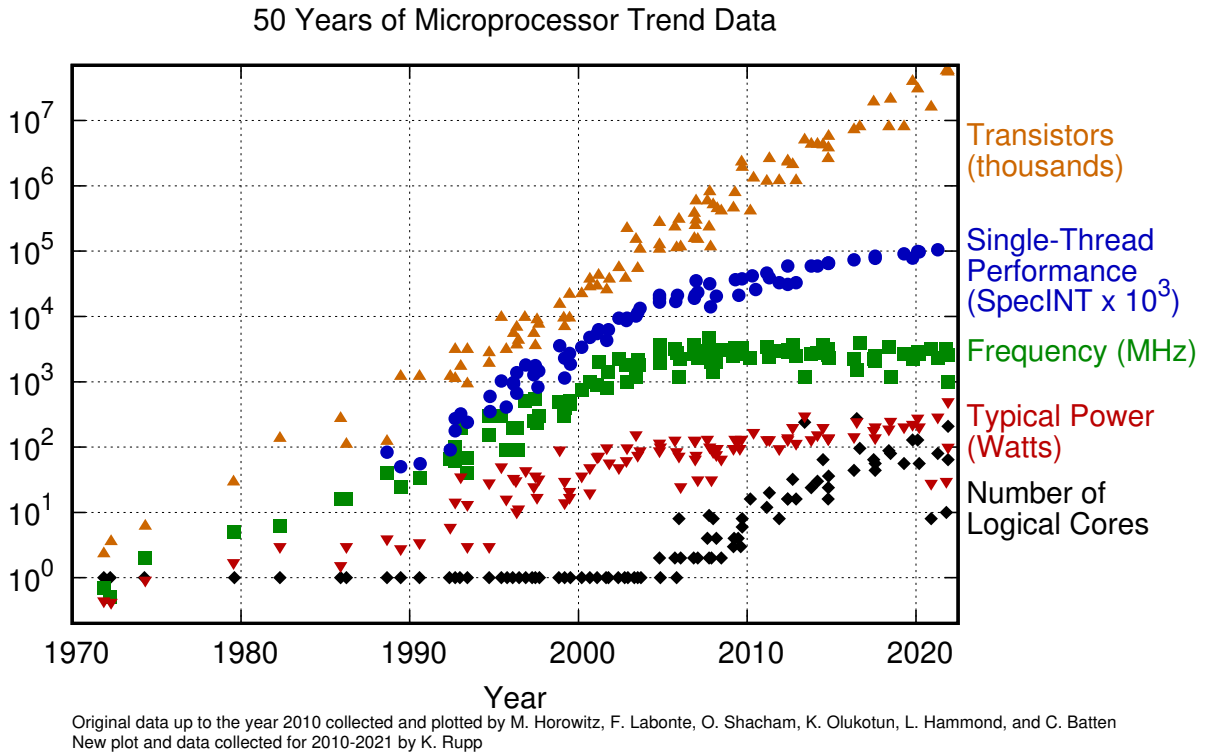


Figure 1.1.: The evolution of server CPU metrics over the last 50 years (K.Rupp et al., 2022). While the frequency and performance of a single core have been in a stalemate for some years, the number of cores continues to increase. This combination leads to ever increasing system performance, resulting in the first exascale machine Frontier, which has entered the *Top500* in June 2022 (Strohmaier et al., 2023).

faced by scientific developers, making it more accessible for them to engage in parallel programming.

However, OpenMP parallelisation is limited to a single node with shared memory. This has a direct impact on the type of problem that can be solved by a scientific application, as it must fit into the main memory of a single node. Nevertheless, solving large-scale problems necessitates the adoption of distributed computing: While single compute nodes typically offer main memory configurations in the range of several hundred gigabytes, large-scale simulations often demand several terabytes of main memory. There are several frameworks to take advantage of distributed memory, which are discussed in more detail in Section 3.2.2. They usually require code changes or even significant code restructuring, especially when used in codes that do not yet make use of distributed computing. This complexity can pose challenges for domain scientists.

In line with the growing trend of increasing data sizes (cf. Section 1.1.2), I/O emerges as a critical subject in HPC and will become even more important in the age of exascale computing (cf. Figure 1.1) (Lockwood et al., 2017; Lüttgau et al., 2018). Like distributed computing, using I/O frameworks efficiently can become difficult. When memory or

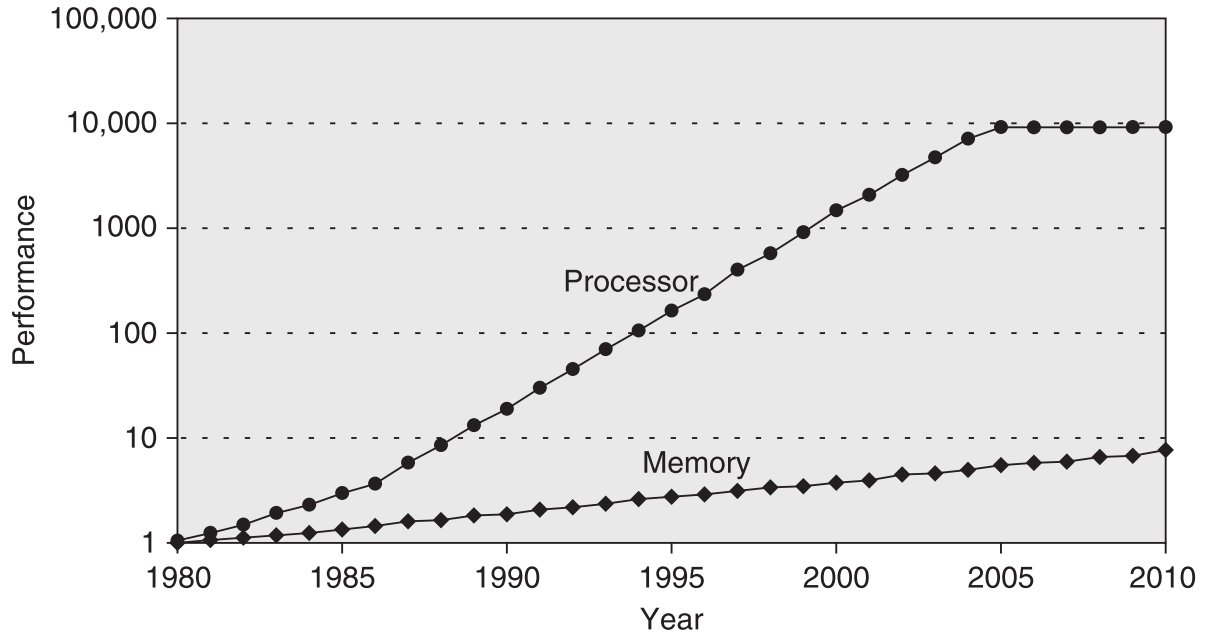


Figure 1.2.: Increasing gap between CPU and memory performance (Hennessy & Patterson, 2012, Fig. 2.2).

I/O requirements pose a challenge, one solution is to reduce the size of the input problem. Even before the advent of many-core architectures, there were discussions highlighting that the growth in peak CPU performance was outpacing the growth in memory performance (Wulf & McKee, 1995). This observation is consistent with the ongoing trend in hardware development, shown in Figure 1.2, which tends to focus on peak CPU performance. Memory performance is a secondary goal, which can lead to new problems. If compute performance per core increases faster than memory performance (be it size or bandwidth) per core, applications may tend to become memory bound.

1.1.2. Natural Science

There are several categories of natural science, such as biology, physics, chemistry, *Earth System Science* (ESS) or astronomy. All of them have use cases where HPC systems can be used quite well. ESS covers a wide range of academic fields related to the systematic study of the Earth. It is an interdisciplinary research field covering a wide range of scales, from microscopic to macroscopic in space and time. For example, ESS includes research on the pedosphere, hydrosphere, biosphere and atmosphere. Models originating from these domains can be examined independently or in relation to their mutual dynamic interactions and feedback mechanisms. The latter is usually advantageous for incorporating dynamic interactions to derive new results.

Wendland et al., 2023 describes how different processes such as infiltration, evaporation and transpiration influence precipitation runoff. Neglecting the dynamic feedback from each component would have resulted in less significant outcomes. Another example

FAQ 8.1: How do land use changes affect the water cycle?

Altering land use affects the water cycle in many ways, with subsequent consequences for the whole cycle.

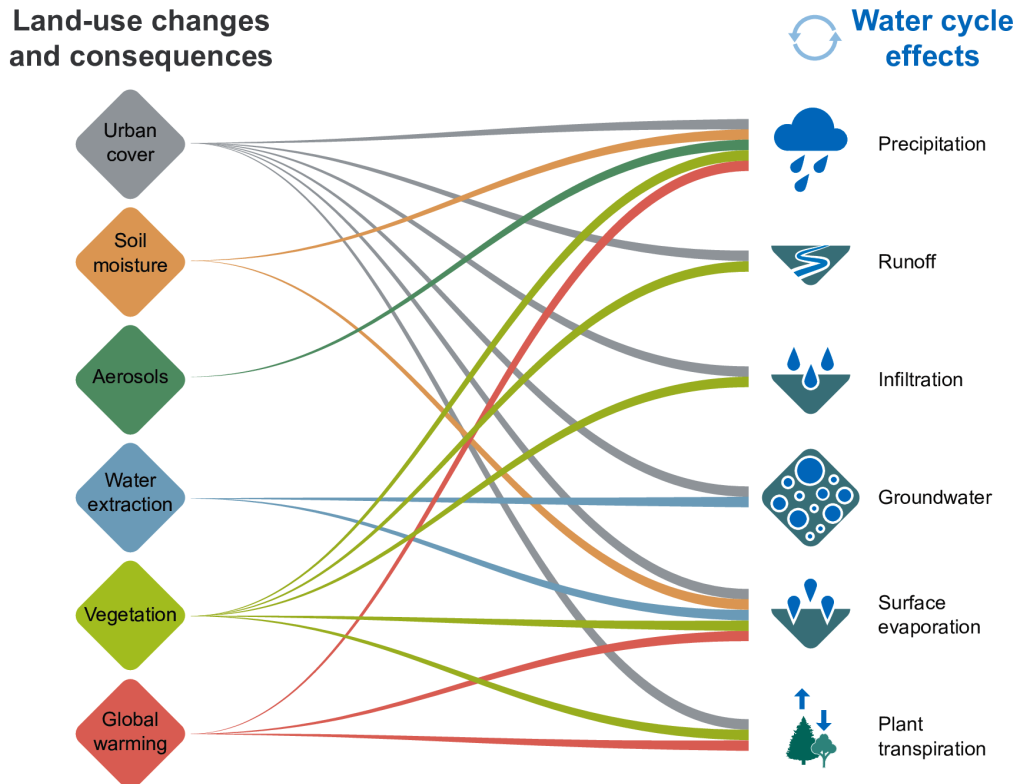


Figure 1.3.: Simplified overview of how land use changes alter the water cycle: among others, effects from the pedosphere, atmosphere, hydrosphere and biosphere as well as their interactions are involved (Douville et al., 2021, FAQ 8.1, Fig. 1)

from the last *Intergovernmental Panel on Climate Change* (IPCC) report combines several land use components and their multiple interactions to derive their impact on several components of the hydrological cycle (cf. Figure 1.3). Considering all the forcing components and their interactions provides a much more detailed drought model than considering each forcing component in isolation.

The IPCC report shows what the globalisation of science can look like. To produce this report, several institutes from around the world have collaborated and contributed their model runs to produce the final result. Figure 1.4 shows the institutions that participated. As climate change becomes more of a concern, ESS is important because it can help us to understand its progress and to evaluate climate change mitigation measures. The DKRZ is for example an HPC centre, which focuses essentially on ESS applications.

In general, the accuracy of these models benefits from larger input sizes used during

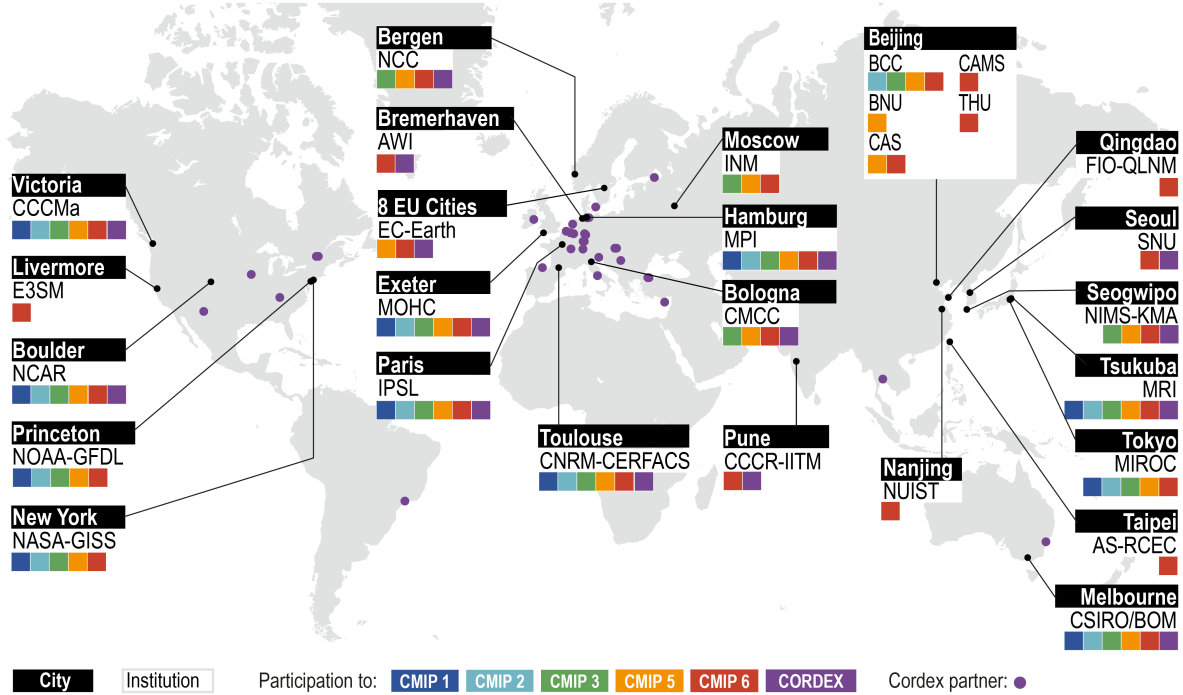


Figure 1.4.: World map of major contributors to *Coupled Model Intercomparison Project* (CMIP) to create the IPCC report (Chen et al., 2021, Fig. 1.20)

the computation phase. For many use cases, the tolerance for (moderately) longer runtimes resulting from the use of larger input problems is acceptable, usually within the same order of magnitude. Such refinement can be achieved by creating larger grids to cover a larger area, or by reducing the spacing of a grid on a fixed area. This helps to model large scale effects (e.g. monsoons) or small scale effects (e.g. convective precipitation (Brune et al., 2020)). Another approach to enhance the capability of a model is by incorporating an additional dimension. This technique can be employed to convert a static model into a dynamic one by introducing a time dimension, or to expand the spatial dimension (Hauschildt & Baron, 2006).

Computer procurement in climate research can never meet the demands of computational tasks, as these will always exceed the performance of the systems available in the medium term. Instead, the computational tasks and their accuracy must be trimmed to the limited computer power. (Palfner, 2012)

In the ESS community, netCDF is a fairly common self-describing data format that makes it much easier for scientists to share data without knowing its exact provenance. NetCDF provides a serial I/O interface as well as support for parallel I/O to access such a file from storage and to work on it in memory. It also provides features to reduce the memory or storage footprint of an application, but these require more in-depth knowledge of the library.

1.2. Problem Statement

Given unlimited time, the necessary skills and motivation, performing code transformations manually will always result in better code than automatically generated code.

The reality is that time is always limited. Acquiring a comprehensive understanding of available frameworks, their functionality, and their potential benefits for an application requires a significant amount of time and effort. Once a framework is selected, the user must invest time in learning how to effectively utilise it. Using an unfamiliar technique is likely to lead to some rookie mistakes, requiring more time to debug or even redesign the approach.

In this case, automated code transformation solutions can be beneficial. Even if individual transformations, such as various compiler optimisations, may be simple, their cumulative impact can be substantial. Performing these transformations manually would be a tedious task, particularly considering that not every transformation guarantees a noticeable benefit (Jayatilaka et al., 2021). Ideally, an automated solution should incur no additional cost for the user. On the contrary, the automatic solution may also make use of well optimised third-party transformations that the user may not even know exist.

The ultimate goal of this work is to provide domain scientists with an automatic code transformation solution so that they can benefit from HPC frameworks without having to learn the ropes first. The decision to use source transformation is discussed in more detail later in Section 2.1. In summary, the preferred approach is to utilise a tool that can automatically transform the user’s code without requiring them to make significant modifications themselves. This streamlined process ensures a more efficient and user-friendly experience. Especially since this work focuses on domain experts working in a highly heterogeneous environment, which will be discussed in Section 1.3.2, this approach is very promising.

The spread of advanced features for parallelisation and I/O is discussed in Section 3.2. The following Sections 1.2.1 to 1.2.4 will establish the fundamental research questions of this work, serving as the basis for evaluating the proposed solution. These cornerstones provide the essential focus for designing and developing the solution, aiming to achieve the ultimate goal: enabling domain scientists to build their applications and benefit from HPC concepts without the need for extensive contributions or modifications.

1.2.1. Research Question 1: Distributed Computing

In the case of a straightforward problem with no need for intercommunication, memory constraint of a single node can be bypassed by partitioning the problem into smaller sub-problems. Each sub-problem can then be solved independently by a distinct application process running on its dedicated node. These problems are called *embarrassingly parallel*. Once the sub-problems need to share data, a new distributed memory parallelisation scheme must be applied.

If the development team of a scientific application consists mainly of experts in the problem domain, building the required parallelisation scheme can become difficult. A common parallelisation scheme used by scientific users in the context of ESS is as already

mentioned OpenMP, which allows to achieve quite good parallelisation efficiency on shared memory (cf. Section 3.2.1).

Research Question 1

Assuming that OpenMP is already used: How can an automatic transformation solution enable the application to make use of a distributed memory parallelisation scheme?

The design how to replace OpenMP with a distributed memory solution will be discussed in Section 4.1.

1.2.2. Research Question 2: Parallel Input/Output

Using a single compute node inevitably imposes an upper bound on the size of the input problem. Figure 1.5 illustrates the issue that arises when the input problem becomes excessively large. This problem can also occur when the input data would fit in main memory, but the application requires additional temporary memory that is allocated during the computation phase. When the peak memory usage of an application exceeds the available memory of a node, it becomes infeasible to use the input problem for computation.

‘A supercomputer is a device for turning compute-bound problems into I/O-bound problems’ – Ken Batcher (Date unknown)

One possible solution would be to load only specific portions of the input problem, rather than loading the entire dataset all at once. Then the computation could be done one by one on subproblems that fit in memory. There are three different approaches, how this can be done (Mendez & Lührs, 2019, Ch. 2.2.2):

- The input problem is split into one file per process, which is accessed by a single process. This is at first easy to implement but can become problematic, if the input problem cannot be split easily or if multiple files need to be accessed after all (e.g. for postprocessing). If many processes shall be used, splitting the input problem every time becomes impractical.
- The input problem is stored within a single file and all processes access the file independently, so that each process has a local, process related view on the data.
- The input problem is stored within a single file and all processes access the file collectively, which provides all processes with the same global data view.

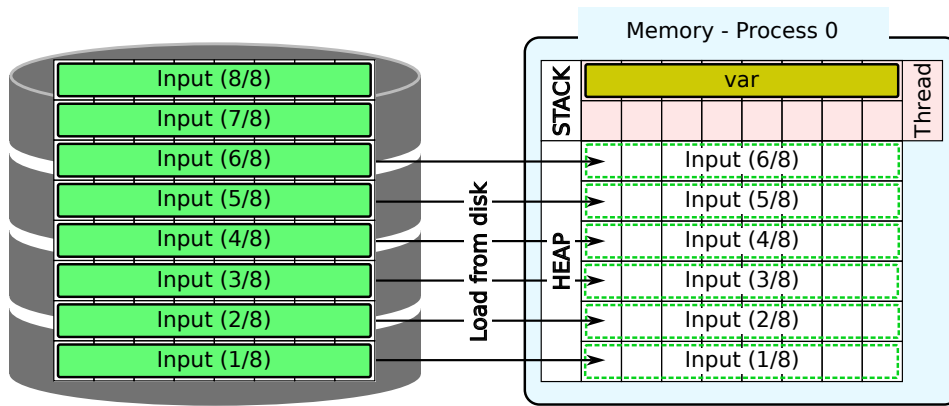


Figure 1.5.: The total main memory of a single compute node places a hard limit on the size of the input problem that can be loaded. The maximum potential input problem size must be reduced even further if a significant amount of memory is allocated on the heap at runtime, competing with the input problem for resources.

All approaches lead to the same result but may have different performance results, whereas the last approach, the parallel I/O approach, is expected to perform best.

I/O can become a significant bottleneck for an HPC application. Paul et al. (2020) conducted a survey on *Lawrence Livermore National Laboratory* (LLNL): applications that perform I/O spend, on average, 78% of their runtime on I/O. This is because only a small minority write efficient I/O code. The best parallelisation of the computation phase is therefore of no use if the execution is thwarted by I/O contention.

Research Question 2

Assuming that serial netCDF I/O is already used: How can an automatic transformation solution transform the application to use parallel I/O with reduced runtime and memory footprint?

The design how to use parallel I/O will be discussed in Section 4.2.1.

1.2.3. Research Question 3: Chunking and Compression

Besides parallel I/O, netCDF has more features to offer like chunking and compression. Chunking can improve the runtime of storage accesses and renders intrinsic data compression possible.

The utilisation of compression can provide various advantages in different scenarios (Cappello et al., 2019; Kuhn et al., 2016):

- **Less storage consumption:** Reducing the data size results in less memory and storage consumption, i.e. storage efficiency increases.

- **Faster data movement:** In comparison to intra-node access, network bandwidth and latency are typically a potential bottleneck. Compressing data before transmitting it over the interconnect to other compute nodes or I/O nodes can increase the effective bandwidth by transmitting smaller data. Disk latency and bandwidth are typically less performant than the interconnect, especially when using magnetic rather than flash-based storage devices with mechanical components. In addition, compression can help mitigate the impact of this bottleneck when transferring data from a fast storage tier to a slower storage tier. Fast data movement is crucial in large scientific instruments that collect data in real time, as it ensures that information is not lost due to the inability to rapidly transfer data from the sensor.
- **Improved cost and energy efficiency:** In return of additional demand on the CPU, the interconnect and I/O components are less stressed. They can therefore be used more efficiently, or a system can be designed with fewer demands on these components. If the additional CPU demand also optimises its idle times (reducing them without becoming a new bottleneck), then the efficiency of CPU usage is also improved. The total energy consumption can be used as a proxy to assess the system's load. If the load on the CPU increases too much then the energy consumption could increase excessively. There is no general statement possible because the scaling behaviour depends on the actual hardware (Kuhn et al., 2020; Miyoshi et al., 2002)
- **Visualisation:** The human eye and display devices have a finite capability to perceive or display resolution, which is limited by their inherent capabilities. Depending on the application, it may not be necessary to store the entire data set, and in this case lossy compression may even be appropriate.

Whether or not these benefits actually occur depends on the actual use case. To benefit from a reduced data size, data content must be compressible, i.e. the *Compression Ratio* (CR) to be greater than 1 (defining the CR as $CR = \frac{\text{original size}}{\text{compressed size}}$). The CR can be improved if the use case allows the use of lossy compression.

According to the increasing gap between peak compute and memory performance shown in Figure 1.2, the chances are high that a memory bottleneck will remain relevant for quite some time.

Research Question 3

Assuming that serial netCDF I/O is already used: How can an automatic transformation solution transform the application to transparently use chunking and compression?

The design how to adjust netCDF I/O will be discussed in Section 4.2.2.

1.2.4. Research Question 4: User Support

The basic intention of the automatic code transformation solution is to enable the user to use selected advanced HPC concepts without having to learn or even fully understand them. This enables the user to rapidly navigate through different options and assess their impact on their specific use case. As previously mentioned, automated solutions can always be outperformed by manually implementing the transformation. So if the user is then interested in gaining insight into what happened during the automatic code transformation step, it would be beneficial to provide some assistance.

Research Question 4

How can an automatic transformation solution provide feedback to enable the user to comprehend the modifications?

The design how to give feedback to the user will be discussed in Section 4.3.

1.3. Approach

A significant contribution of this work is the design and implementation of an automatic code transformation tool to automatically analyse and transform high-level code. This tool enables domain scientists to effortlessly and rapidly experiment with various HPC concepts without the need for extensive prior knowledge or learning. This will allow them to assess the benefits that their application can derive from such an approach. The tool, *Compiler Assisted Transformation of OpenMP kernels* (CATO), is named after the OpenMP focus mentioned in the first research question in Section 1.2.1. It only requires that the application already makes use of basic OpenMP if memory sharing is to be used, or basic netCDF if parallel I/O and compression are to be used. Both components can be used independently, but the parallel I/O performance of the netCDF component can benefit from the memory allocation component.

The selected HPC concepts and used frameworks are discussed in Chapter 3 in more detail.

1.3.1. Source Transformation

Automatic source code transformation has played a role in computer science from the very beginning. As soon as developers stopped writing machine code directly - that is, they used an abstraction language known as a high-level language - some kind of automatic code transformation was used to generate the low-level machine code. The ability of the computer to read and modify a program written by a human developer has had a huge impact. By leveraging an operating system and compilers, the computer can assist developers in their work by offering support in various aspects. This assistance enables

programming to be conducted at a higher level of abstraction, resulting in accelerated productivity on a larger scale.

1.3.2. Target Audience

HPC and ESS are a large domain, with diverse characteristics. Before the design of CATO is discussed it is therefore important to have a closer look on the target audience before. Those aspects have an influence on the requirements and intentions of CATO. The major questions are:

- Who shall use CATO?
- Which kind of application shall be modified with CATO?
- What hardware will be used to execute the modified application?

User Characterisation

Natural science is the driving factor behind this work as it has a large community, which can benefit from HPC systems but potentially do not have appropriate training within their curriculum. Domain experts are highly knowledgeable in their specific fields of application, possessing deep understanding and expertise. However, proficiency in computer science concepts often remains optional and necessitates self-directed learning and autodidactic efforts. In most cases, leveraging additional HPC resources scales up the scientific output of applications, making it a sensible and advantageous choice.

Indeed, the field of HPC encompasses several core areas, including compute components, internal and external interconnects, memory or storage systems, and accelerators and more. These fields are characterised by their evolving nature, with constant advances and developments. Over the last twenty years, HPC systems have become much more heterogeneous. And while some frameworks have been established for many years and are likely to remain relevant for years to come, the software is adapting just as quickly as the hardware is evolving. This makes it even more difficult for domain experts to keep track of existing solutions and their best practices.

‘The questions don’t change, but the answers do’ – Daniel Reed (2023)

Chapter 3 gives an overview of which features of the investigated HPC frameworks are commonly used and which CATO can make use of. Two additional studies, Arvanitou et al., 2021 and Amaral et al., 2020, have examined the working conditions of domain scientists, specifically focusing on their abilities and needs. According to their findings, scientific applications are increasing in complexity, demanding a greater understanding of computer science. Domain scientists are seeking tool support that is easy to learn and use. This is precisely the area where CATO aims to provide assistance and support.

Application Characterisation

Due to the extensive scope of ESS, it is challenging to pinpoint a singular application that can adequately represent the entirety of ESS development. In Section 4.1.1, several potential communication patterns are discussed, and a decision is made on which pattern to base the design of CATO on. However, due to the nature of HPC bandwidth hierarchies, it is preferable to focus on applications that are computationally intensive and require comparatively little (external) communication. This argues for concentrating on structured data structures. To reduce the overhead caused by CATO, it would be beneficial for the application to work on a single, large (i.e. memory filling) problem in heap memory.

Many domain scientists are familiar with OpenMP for shared memory parallelisation, as it has a short training period to achieve significant runtime improvements quite quickly. Therefore, the first component of CATO will use an existing OpenMP kernel to derive knowledge about which variables are worth distributing and where the computational intensity is likely to be high. These are only assumptions, but they can help to optimise CATO’s design so that it can work automatically without any additional input from the user. Since OpenMP is used not only in ESS, but in all sorts of applications that need to take advantage of shared memory parallelism, the first component of CATO can be used in any application that uses OpenMP.

Above a certain level of complexity, the application is unlikely to benefit from the automatic code replacement introduced by CATO. The more complex an application is, the more overhead CATO could introduce, and the more likely it is that a more complex replacement will be required to take account of the circumstances. The latter is difficult to achieve by automatic replacement, and is likely to require manual code changes by a developer with domain knowledge.

There are still ample applications that stand to gain significant benefits from the implementation of CATO. Some complex software, such as the *Icosahedral Nonhydrostatic* (ICON) model, which consists of multiple components coupled together and already uses parallelisation frameworks to run on the distributed memory of many compute nodes, is not in the target audience of CATO (Zängl et al., 2014).

There are many applications developed by a small team of domain scientists who also run their application on HPC hardware, but are limited to the entry-level features of HPC frameworks. This can be demonstrated examining job data from an HPC system. Therefore, job scheduling data is taken from the HPC system Mistral at DKRZ, which was collected for a master’s thesis, which I supervised (Coym, 2021). The data was provided by Slurm and covers the year 2020 (Deutsches Klimarechenzentrum GmbH, n.d.-a).

Cores	1	2	3-4	5-8	9-16	17-32	33-64	65+
	12.30%	1.51%	18.14%	16.14%	18.86%	3.10%	18.27%	11.64%

Table 1.1.: Frequency distribution of allocated cores on single node jobs. A node has two CPUs with 18 cores each.

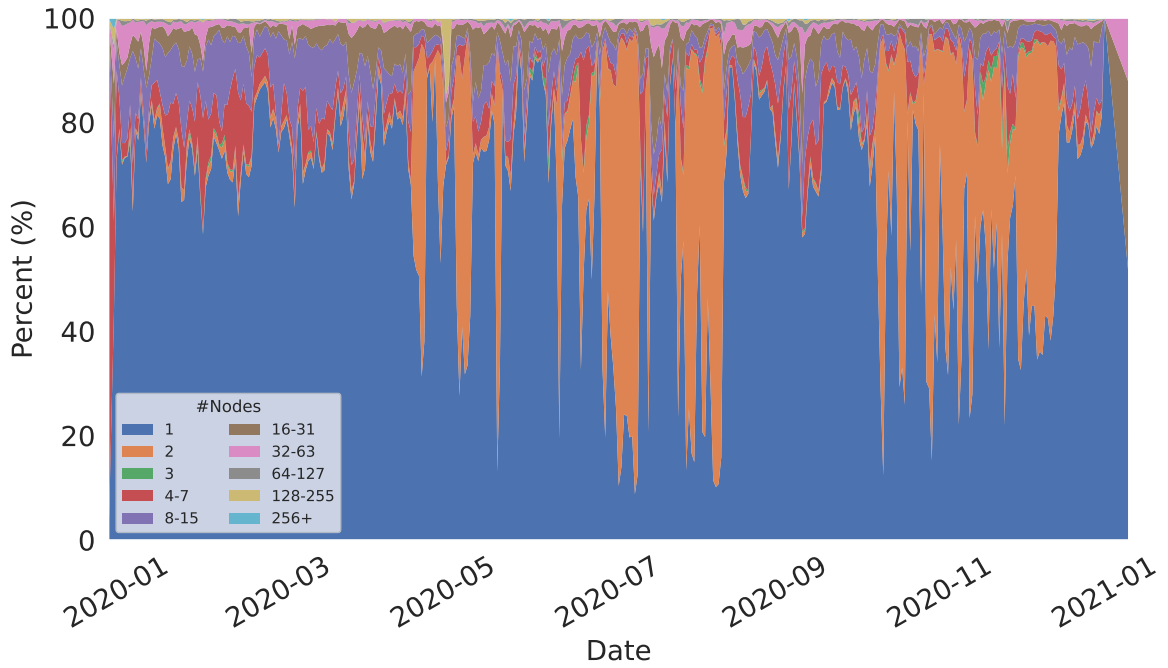


Figure 1.6.: Share of used compute nodes per job on Mistral in 2020

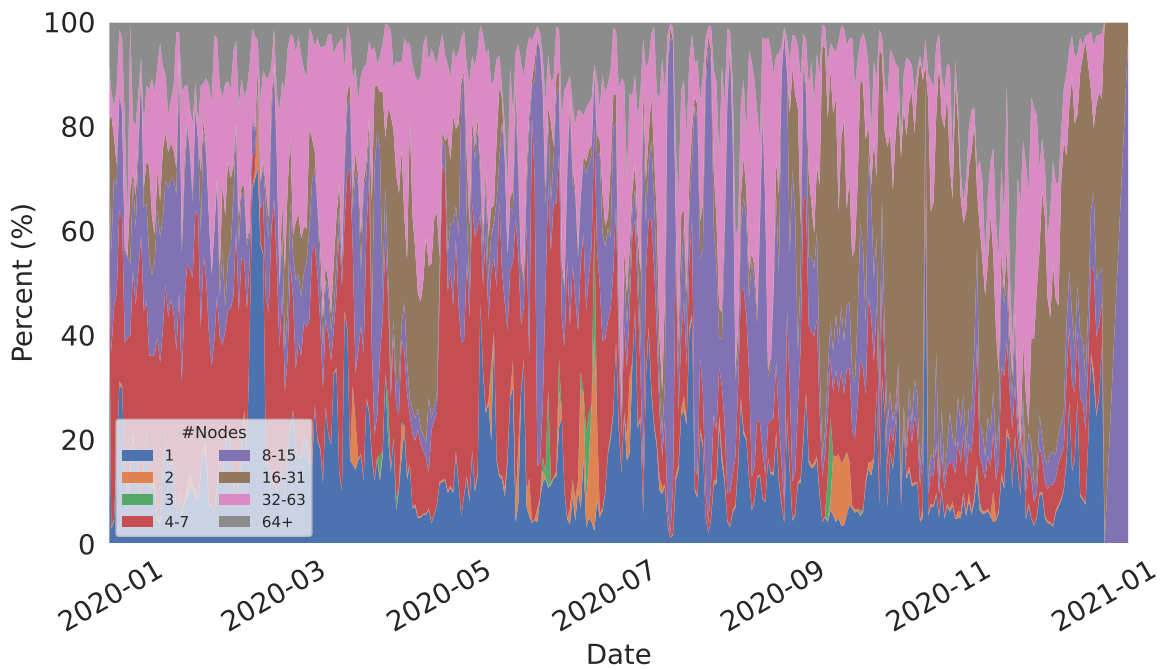


Figure 1.7.: Number of allocated cores per single node job on Mistral in 2020

Figure 1.6 shows the distribution of jobs grouped by the number of nodes allocated over the year. Job allocations for one (55.91%) and two (31.47%) nodes predominate. It is not clear from this data how many single node jobs are truly productive jobs and not just configuration tests or debugging runs, for example, but their number is still significant. Those jobs that do not take advantage of distributed memory parallelisation are potentially part of the target group.

Depending on the partition chosen, Slurm may assign jobs exclusively, so it is not possible to tell whether the jobs using a single node used some form of parallelisation. Figure 1.7 can be used as a proxy measure: It shows single-node jobs, and apparently the majority of these jobs allocated at least two cores (87.7%) on the shared partition. The share of each node configuration over a year is shown in Table 1.1.

The parallelisation framework utilised by these jobs to distribute the workload across cores, or whether they effectively utilised all the allocated cores, remains unclear. In summary, there is a significant proportion of HPC jobs that are already using shared memory. These jobs could therefore benefit from CATO.

Hardware Characterisation

Experience shows that there are many applications from ESS that are optimised to run on CPUs. HPC supercomputers such as Mistral or Levante at DKRZ primarily use CPUs. There are efforts to include more GPUs, but this is still an ongoing process (Giorgetta et al., 2022). Therefore CATO’s design will focus on OpenMP rather than a GPU centric framework like CUDA, OpenCL or OpenACC. Based on this, Chapter 3 will discuss which HPC frameworks will be used to perform the code transformation.

‘From a user perspective, the “ideal high performance computer” has an infinitely fast clock, executes a single instruction stream program operating on data stored in an infinitely large and fast single-memory, and comes in any size to fit any budget or problem.’ (Sterling et al., 2017)

1.4. Outline

The present thesis is structured into the following sections: Chapter 3 presents the HPC frameworks used to perform the code transformations needed to answer the research questions in Section 1.2. Having established this basis, the actual design of CATO is worked out. First of all, Chapter 2 discusses possible methods of code transformation in general. Once this decision has been made, the actual design of CATO is developed in Chapter 4. The elaborated design is then mapped to the chosen transformation framework in Chapter 5 to complete the construction of CATO. Chapter 6 examines related work that can be used to somehow (semi-)automatically transform the code. Differences are then addressed and compared with this work.

The evaluation of how effectively CATO performs in relation to the research questions is conducted in Chapter 7. The concluding remarks of this work can be found in Chapter 8.

1.5. Summary

CATO, a compiler-assisted high-level code transformation tool designed, developed and evaluated in this thesis, will support a collection of relevant HPC frameworks. The use of this tool will allow the target users or their written applications to benefit from those HPC frameworks that are not easy to implement manually, if done without prior training.

Targeting users and their applications from the ESS community narrows down the actual purpose of CATO to four central aspects:

- Allow OpenMP kernels to run on distributed memory.
- Turn serial netCDF I/O operations into parallel I/O operations.
- Optimise data size by extending netCDF kernels through chunking and compression.
- Soften up the black-box behaviour of CATO's design so that a user is able to understand it better.

2. Surveying Techniques for Code Insertion

In Chapter 1 some problems have been worked out, which can be boiled down to the fact that the HPC environment has become quite heterogeneous on many levels, and in order to get optimal performance complex frameworks have to be used. This can quickly become overwhelming for untrained users to use or even keep track of all the available solutions. In this work, a new tool called *Compiler Assisted Transformation of OpenMP kernels* (CATO) is constructed. CATO is designed as a toolbox that takes into account HPC concepts that can be automatically inserted into an application's source code by the user. Its design relies on an automatic source code transformation mechanism, so that the user only needs to decide which primed transformation to apply to the code. The code passages in the control are searched and adapted accordingly by prepared replacement code within CATO, the nature of the benefit depends on the CATO tool used. To demonstrate the capabilities of CATO, the design specification will focus on the transformation of OpenMP parallelisation and netCDF I/O.

This chapter gives a general overview, starting with the specification of CATO (Section 2.1). Based on this, the desired interaction with a user can be described. Once the general behaviour has been clarified, a more detailed view of CATO and the technologies it uses, which will be introduced in Chapters 3 and 6, is given. Two topics are covered in Section 2.2:

User Interaction (Section 2.2.1) There are several ways to control the code transformation by the user. They differ in how much users need to interact with CATO and adapt their application code on the one hand, and how much CATO needs to automatically derive information and perform (transparent) application modifications on the other. Since CATO's target audience are domain experts, its use should be as user-friendly as possible and provide sufficient feedback.

Code interaction (Section 2.2.2) What is the best way to handle high-level code? There are many different approaches, all with their own advantages and disadvantages. The criteria CATO must meet are discussed here, and a decision is made based on this. Once the main technology is chosen and its capabilities are known, two important issues are discussed: how to analyse high-level code, and how to detect and replace kernel patterns (cf. Section 2.3).

A more detailed description of the individual components is given later in Section 2.4 as well as in Sections 4.1 to 4.3.

The specification of CATO is formulated based on these topics and allows the selection of specific technologies and libraries to meet the requirements. Based on this general discussion, the complete workflow of the tool can be elaborated in Section 4.4.

2.1. Specification

CATO has to fulfil several tasks that are not easy to combine.

On the one hand, it needs to be quite generic and provide a very modular internal structure so that many different libraries can be integrated. This is made even harder by the fact that the only constraint is that they must be libraries from the HPC environment. Even if HPC itself is a very specific discipline, it can still be broken down into many unique methods within HPC (e.g. with a focus on GPUs, FPGAs or I/O). There are many HPC use cases and many different types of hardware and software designed to solve these particular use cases. Many HPC systems use GPUs and CPUs from different vendors (e.g. ARM, Intel, AMD), each providing its own software ecosystem, resulting in a rather heterogeneous landscape of specialised hardware and software (Trott et al., 2022).

For each type of scientific application, there is a specific type of hardware that provides the best performance.

For each type of HPC machine architecture, there is a specific type of application that uses the machine to its full capacity.

Finding the best libraries to use in an application to run on a particular HPC environment is a difficult problem for a user with limited prior knowledge, and may require lengthy implementations and performance testing. To achieve this, it is probably not enough to use the same optimisation library all the time, so users need to familiarise themselves with different libraries and learn how best to use them.

Each specialised library then has its own *modus operandi*, and it becomes difficult to fit several different libraries into a single standardised interface. This is where CATO helps the user by taking care of the implementation so that the user can concentrate on evaluating the library. Currently, the modularity of CATO is designed to integrate two different types of HPC methods, which will be described in detail later in Section 4.1 (focus on OpenMP) and Section 4.2 (focus on parallel I/O using netCDF).

On the other hand, the user, who is a domain expert but not necessarily a computer scientist, cannot be expected to invest a lot of time in training to use CATO. If it is not easy enough to use, it will not be used at all.

Balancing these two aspects is a major challenge and must be considered in the design and implementation of CATO. The main strategy is to provide a decent default mode that allows CATO to be used without any complex configuration, but still provides a creditable benefit. The user simply chooses which tools he wants to use and integrates CATO into his build chain. In addition, the user should still have the ability to provide

more complex hints or even add optimised modules to assist CATO in its work to improve the result.

2.2. Code Transformation

There are several approaches to code transformation. They cover a wide range of functionality and practicality. The focus on a specific user group (cf. Section 1.3.2) already leads to some limitations regarding the maximum reasonable complexity to use an approach and therefore excludes some of them. Among the remaining ones, the most promising one is chosen to perform the code transformation while fulfilling the requirements set by Section 1.2.

2.2.1. User Interaction

A fundamental design decision to be made at the outset is how the user is to control CATO’s interaction with the original high-level code on which the new technologies are to be applied. There are several approaches, which differ in how explicit the user needs to be (in descending order of the amount of work required from the user):

- The user manually adds new code that has a direct impact (e.g. new data structures or algorithms from external libraries).
- The user adds annotations that later lead to automatic code transformation (e.g. hints or pragmas).
- The user applies a tool to his original code, and the tool performs the necessary code changes (semi-)automatically (e.g. Polly, Chapter 6).
- The user does not need to do anything, as the runtime automatically and transparently optimises the execution.

All these approaches have their pros and cons; they differ in the degree of control the user has, but also in the amount of code changes required. To assess the suitability of each approach, they will be discussed in more detail.

Manual code changes Manually modifying the code to extend it with new libraries would require a considerable amount of effort on the part of the user, both to learn them and to integrate them into their code base. For example, the current *Message Passing Interface* (MPI) 4.0 specification contains 664 functions (not including profiling functions). Not all of them are needed (about ten functions are sufficient to write trivial MPI applications), but this is still a lot of material to work through. This has to be done for each individual library, and can quickly become tedious and error-prone if the user is not an expert. The complexity of this approach can be mitigated by using some sort of abstraction layer that allows the backend to be changed at compile-time or run-time. For example, Kokkos was created to provide

a hardware abstraction so that an application can run on different types of HPC infrastructure consisting of different types of heterogeneous or homogeneous cluster architectures (Trott et al., 2022). This solution focuses on hardware abstraction rather than software abstraction, and therefore does not meet the requirements. For CATO, only high-level codes that do not already take such modularity into account will be considered.

User annotations The second approach considers only minor code changes that have an indirect impact. Usually these are annotations that help the compiler by providing expert knowledge that only the user or domain expert has. An important type of annotation are language attributes in C, which can be used by adding `__attribute__((attr))` (non-standard implementation extension, e.g. recognised by gcc) or `[[attr]]` (since C++11). Fortran provides an equivalent approach via the `DIR$ ATTRIBUTES` compiler directives. Using them does not change the behaviour of the application, but a user can exercise control over how the compiler works, or assist the compiler in its work by providing expert knowledge to optimise the size or runtime of the final binary. For example, the compiler might omit warnings about unused variables or enforce function inlining. Another example could be directives like those in OpenMP. The user annotates parts of the high-level source code, and the compiler then transparently forks, executes and merges the necessary threads. In both cases, the user shares his expert knowledge with the compiler, which can then optimise its work during the compilation process. If an unknown attribute is used or a directive is not supported by the compiler, the compiler will issue a warning or simply ignore it.

Tool support The user is using a tool that, in the best case, can be installed in his user space. How the tool is actually used depends on the tool itself. It could be run on the application binary, which is then modified (e.g. using binary instrumentation, cf. Section 2.3.1). Another option is to integrate it into the build process of the application. In the best case, this just means replacing the compiler call within the application build process: The wrapper script passes the application code through the tool and then finishes building the binary using the original build instructions on the now modified application code. This allows the user to install the tool on his own, without any help from the system administrator; The user can also explicitly control the execution mode of the tool if he wishes. It is then absolutely clear to the user that their application has been modified and how (generally omitting unnecessary details).

Runtime environment By introducing a new runtime environment as an additional layer, all calls to the underlying layers can be intercepted and replaced. On the one hand, this reduces the user's effort, but on the other hand, with even less interaction or explicitly passed hints from the user, a runtime environment has to invest much more work in automatic detection algorithms. And even then, similar approaches like *Single System Image* (SSI) have been shown not to meet the requirements of CATO (cf. Chapter 6). Furthermore, it is questionable to transparently modify the

user’s code, as this would manipulate the user’s expectations of the runtime and could even degrade performance; this would be disappointing if the user had done this on purpose, but unacceptable if done unknowingly. Building a new runtime, probably best implemented within the OS layer, to directly access internal functions such as memory and network management for optimal performance, is therefore out of scope.

Manual code changes were ruled out as a solution because of their complexity in use. For a similar reason, user annotations were rejected; although they are easier to use, they still require code changes. Since they are comparatively simple, they could still be used as an optional tuning choice: the user could add hints to the application to influence the code transformation. Since the runtime environment was also ruled out, CATO is set up as an external tool, more precisely as a compiler wrapper. The user triggers the code changes simply by adding CATO to his build chain and choosing which of its tools to use. This has the additional advantage that the default settings of CATO make it easy to produce exploratory results quickly, without the need to add optional hints due to the short learning curve. Other tools such as *MUST* (Hilbrich et al., 2013) or *MPI-checker* (Droste et al., 2015) have already shown that such an approach can be used without extensive training and still be beneficial to the user.

Usually the installation of a compiler can be done in user space, so the user is independent of the current platform or its administrators. Write permissions to their home directory are sufficient. To simplify the installation process, CATO comes with an installation script. It uses *spack* to handle the installation of necessary software dependencies in userspace. This makes the initial setup easy for the user; what exactly the user workflow looks like is discussed in Section 4.4.

2.2.2. Code Interaction

When reviewing existing code transformation tools, it became clear that there is no single solution that meets all the requirements. They range from simple approaches, which are easy to use but perform quite explicit code replacement, to more generalised solutions, which operate at a higher level of abstraction and can therefore be more difficult to use, but can also perform more powerful code transformations.

Given the target audience of CATO, it is important that all the complexity of a more sophisticated and abstract approach is hidden within CATO’s internals, to maintain ease of use. This does not preclude advanced interfaces to provide hints, or some rewriting component to add user-defined replacement schemas, as long as they can remain optional. The default mode of CATO must be easy to use for untrained users, otherwise it would miss its core target audience. Therefore, we will only discuss solutions that can be pre-configured by an experienced user, so that the domain expert only has to exclude it without having to know anything about the internals.

Source transformation techniques are either too simple for complicated use cases or too complex to be modified or even used by non-experts (Johnson et al., 2022)

For example, a simple approach might be to automatically perform replacements using regular expressions. These expressions can be applied to strings and return a logical value or a set of matches. Tools such as `sed` or `awk` or languages such as Python or Perl support the use of regular expressions. Even more complex replacements can be performed because, despite their name *regular expression*, they are usually more powerful than regular languages and are instead context-sensitive languages. To distinguish these more powerful regular expressions from the theoretical construct, they will be referred to as *Extended Regular Expression* (Regex) (Câmpeanu et al., 2003).

An *Extended Regular Expression* (Regex) can be used to automatically search for specific text, e.g. function calls, and then perform an adjustment on the matches found. For many non-trivial use cases it is necessary to consider the context as well, e.g. to track a variable that has been modified by a function call. The larger the scope of the context to be considered, and the more diverse the semantics of the analysed code, the more complex the Regex or chain of regexes must become. Disruptive code elements such as newlines and comments, which can split the code unpredictably, further complicate this process. This can become arbitrarily complex when this approach needs to be generalised to fit different code applications written by different user groups, each following their own coding style. Therefore, this Regex-centric approach will not be used to develop CATO.

More sophisticated solutions than using a Regex are offered by MARTINI and No-brainer (Johnson et al., 2022; Savchenko et al., 2019). They take prepared snippets of code and use a *Abstract Syntax Tree* (AST) matcher to find and replace significant pieces of code. The replacement rules are relatively easy to write, but only apply to a limited scope, which is too narrow for the needs of CATO.

An advanced solution would be to use an existing tool that is capable of automatically traversing source code and performing analysis and transformation. Using a compiler framework as a foundation has several advantages, making initial development easier and harder in terms of robustness and correctness. Choosing the right compiler framework makes it possible to iterate through the high-level code, searching for specific code kernels and adjusting them as necessary. It also makes it easier to discover and incorporate the context and relationships of the kernel within the overall application code. This is a necessary feature because a kernel is not just a set of statements and expressions that are valid only in the local environment, but can affect the whole compilation unit (e.g. if memory is allocated outside the kernel).

For example, at the beginning of Listing 2.1 (line 5) a variable `error` is declared and initialised, later assigned the value of a library call (line 9) and finally read (line 14). These accesses are still traceable, but it becomes more complicated when variables are passed as parameters. The handling of the call to the `free` function on `buffer` in line 13 needs to be considered within CATO, as `free` is an important function; its appearance in application code must be expected. When a pointer is passed to an external third

```
1  #include "external_lib.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  int main() {
5      int error = 0;
6      int *buffer = malloc(10 * sizeof int);
7      int *alias = buffer;
8      // [...]
9      error = external_call(buffer);
10     alias[0] = 0;
11     printf("%d", buffer[0]);
12     // [...]
13     free(buffer);
14     return error;
15 }
```

Listing 2.1.: The scope of a variable can easily be spread through the whole application code.

party library (cf. line 9), it is unclear what is done with this variable. Without going into the source code of the external library, it is impossible for the compiler or a human being to derive this information. If CATO is supposed to do this, it needs to be applied to the external library as well. At the very least, CATO will need to parse, if not transform, the library's source code, but then the library will also need to be recompiled. If the source code of the library is not available, the task becomes even more difficult and binary instrumentation could be used. Automated evaluation of an external library using CATO would require significant additional effort and is therefore beyond the scope of this work. CATO's design focuses on explicit variable assignments and selected libraries that are relevant to the intended user group (as noted in cf. Section 1.3.2).

It can become very difficult to keep track of memory accesses when pointer aliases are involved. In line 7 a new pointer alias **alias* is created, pointing to the same memory area as **buffer*. So not only accesses to the primary pointer need to be monitored, but also to its aliases (cf. line 10). With increasing complexity, this becomes difficult to follow; multiple indirection levels induced by aliases are even NP-hard (Landi & Ryder, 1991; Ramalingam, 1994). As this is a significant problem, it will be discussed in more detail in Section 5.2.

Finally, there will be special use cases where CATO may not be able to perform the desired code transformation without additional hints. The design aims to reduce the number of such use cases by using heuristics that work at least for the intended audience.

2.3. Compiler-Assisted Approach

Based on these findings, LLVM was chosen as the framework on which to build CATO. This choice is elaborated further in Section 3.1. The architecture of LLVM and its components are described in more detail in Chapter 5. Following the tool design, the implementation of CATO is also discussed in detail there.

2.3.1. LLVM Component Sequence

In the previous section, the decision was made to implement CATO as a tool that uses the capabilities of a compiler framework. How code adaptation is handled via the compiler approach is discussed in detail in this section. Since the tool focuses on applications in the HPC context (cf. discussion in Section 4.1), only compiler frameworks capable of compiling C, C++ and Fortran code will be considered.

There have been several shifts in the way programming languages work and the kind of features they offer. At the same time, compilers have had to adapt to the new demands placed on them to support new language features and back-end technologies. A modern compiler is made up of many phases, which step by step analyse and transform the original high-level code into the final machine code that can be executed. In Section 3.1.1 the individual phases that a compiler typically consists of have been presented.

On the way from the original high-level source code to the final binary, there are three (intermediate) code states that are generally generated by a compiler. Together with the original high-level source code, this gives five code states that CATO could work on: the original high-level code, the AST, the *Intermediate Representation* (IR), the *Control Flow Graph* (CFG) and the final binary. Each level will be discussed in order to choose the most appropriate one. The explanation of individual LLVM constructs will be kept short in this section, a more detailed description will follow in Section 5.1.

A simple test application (see Listing 2.2) is used to demonstrate the techniques. Although this minimal example omits many of the language features of C, it covers four important concepts: reading from and writing to memory, conditional branching and loops. To spice things up, user input is used to decide which branch to take during application execution. This cannot be evaluated statically, but only at runtime (since it depends on the user's input). Table 2.1 shows, for selected examples of C syntax from the simple test application, which lines of the generated AST and IR output are associated. This is not always a one-to-one mapping: if it is not, only the beginning of the corresponding block is referenced. To recreate the individual code layers, LLVM tools can be used, which are listed in Appendix A.1.

Original high-level code: To work directly on the original high level code, a combination of code recognition and transformation could be used. This could be done by a combination of RegEx (code recognition) and `sed` (code transformation), but this has already been ruled out in the previous Section 2.2.2. Other approaches suffer from the same drawback of being quite language-specific, and most likely style-specific as well, since there are countless ways to achieve the same behaviour. More

```
1  #include <stdlib.h>
2
3  int main(int argc, char **argv) {
4      int result = 0;
5      int counter = 0;
6
7      if (argc == 1)
8          counter = 1;
9      else
10         counter = atoi(argv[1]);
11
12     for(int i = 0; i < counter; i++)
13     {
14         result += 1;
15     }
16     return result;
17 }
```

Listing 2.2.: Trivial C code with conditional branches depending on dynamic input to demonstrate potential levels during compilation process where CATO could step in.

	Original Code	AST	IR
result:	line 4 (Listing 2.2)	line 6 (Listing 2.3)	–
if:	line 7 (Listing 2.2)	line 11 (Listing 2.3)	line 23 (Listing 2.5)
for:	line 12 (Listing 2.2)	line 32 (Listing 2.4)	line 39 (Listing 2.6)
Unary ++:	line 12 (Listing 2.2)	line 42 (Listing 2.4)	line 51 (Listing 2.6)
Binary <:	line 12 (Listing 2.2)	line 37 (Listing 2.4)	line 42 (Listing 2.5)

Table 2.1.: Association of code elements from the original code and the AST. Often a high-level one-liner is split up into several elements on a low-level layer; in this case the line number denotes the beginning of the associated block.

```

5  `~CompoundStmt 0xf432c18 <col:33, line:33:1>
6    |-DeclStmt 0xf431610 <line:18:3, col:17>
7    | `~VarDecl 0xf431588 <col:3, col:16> col:7 used result 'int' cinit
8    |   `~IntegerLiteral 0xf4315f0 <col:16> 'int' 0
9    |-DeclStmt 0xf4316a8 <line:19:3, col:22>
10   | `~VarDecl 0xf431640 <col:3, col:7> col:7 used iteration_count 'int'
11   |-IfStmt 0xf432960 <line:21:3, line:26:3> has_else

```

Listing 2.3.: Extract of the textual and machine readable AST representation of Listing 2.2 with focus on variable usage and conditional branches.

sophisticated solutions such as *Source-to-source transformation* (S2S) compilers are discussed in Section 3.1.1 and decided against. Major code changes by the user were also ruled out as a possible solution, as CATO should be easy to use for the target audience (see Section 1.3.2).

AST Two extracts of the AST generated with `clang` are visible in Listing 2.3 and Listing 2.4 as textual representations. In Figure 2.1 the AST can be seen as a tree representation, where the structure of the original source code can be seen after it has been tokenised and parsed (cf. Section 3.1.1).

This code representation is a welcome way to perform static code analysis, as the structure is broken down. LLVM provides an API to programmatically access the AST for analysis and iteration. Individual statements in the original code can be moved to the AST.

An example of how to use this layer is demonstrated by *MPI-checker* (Droste et al., 2015), which started as a standalone tool and has since been merged into `clang-tidy` in LLVM version 4.0. It tracks MPI calls and checks for clear errors like type mismatches or undefined behaviour due to missing completion calls like `MPI_Wait`. This static approach works independently of runtime values such as the number of processes used, allowing for more generic assertions. On the other hand, it provides limited reasoning about interactions based on heap memory (Droste, 2020). The reason for this is that the AST provides only an outline of the general execution flow of the application.

This becomes a major drawback to using it as a basis for CATO, since its required memory handling capabilities rely heavily on values that are only known at runtime. Although the AST is customisable (the LLVM API does not provide setter functions for the AST, but there are existing workarounds), the LLVM philosophy is to ensure the immutability of the AST (LLVM Project, n.d.-b). Therefore, the AST could be used for information gathering, but not for the necessary source transformation, which should be done by CATO.

IR Two snippets of *Intermediate Representation* (IR) code generated with `clang` are


```

32 | -ForStmt 0xf432b98 <line:28:3, line:31:3>
33 | | -DeclStmt 0xf432a30 <line:28:7, col:16>
34 | | | -VarDecl 0xf4329a8 <col:7, col:15> col:11 used i 'int' cinit
35 | | | | -IntegerLiteral 0xf432a10 <col:15> 'int' 0
36 | | -<<<NULL>>>
37 | | -BinaryOperator 0xf432ab8 <col:18, col:22> 'int' '<'
38 | | | -ImplicitCastExpr 0xf432a88 <col:18> 'int' <LValueToRValue>
39 | | | | -DeclRefExpr 0xf432a48 <col:18> 'int' lvalue Var 0xf4329a8 'i'
    ↳ 'int'
40 | | | | -ImplicitCastExpr 0xf432aa0 <col:22> 'int' <LValueToRValue>
41 | | | | | -DeclRefExpr 0xf432a68 <col:22> 'int' lvalue Var 0xf431640
    ↳ 'iteration_count' 'int'
42 | | -UnaryOperator 0xf432af8 <col:39, col:40> 'int' postfix '++'
43 | | | -DeclRefExpr 0xf432ad8 <col:39> 'int' lvalue Var 0xf4329a8 'i'
    ↳ 'int'
44 | | -CompoundStmt 0xf432b80 <line:29:3, line:31:3>

```

Listing 2.4.: Extract of the textual and machine readable AST representation of Listing 2.2 with focus on the for loop.

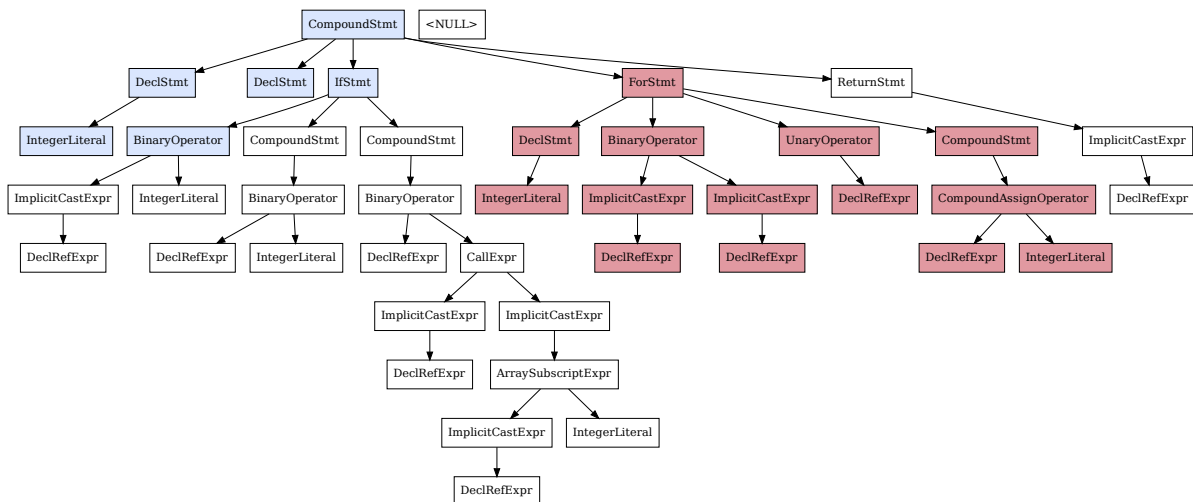


Figure 2.1.: Graphical AST representation of Listing 2.2. Blue nodes correspond to the textual extract in Listing 2.3, red nodes correspond to the textual extract in Listing 2.4. The colours have been manually added to match the colours from the CFG in Figure 2.4 to indicate associated nodes.

```
23 if.then:                                     ; preds = %entry
24     store i32 1, i32* %iteration_count, align 4
25     br label %if.end
26
27 if.else:                                     ; preds = %entry
28     %1 = load i8**, i8*** %argv.addr, align 8
29     %arrayidx = getelementptr inbounds i8*, i8** %1, i64 1
30     %2 = load i8*, i8** %arrayidx, align 8
31     %call = call i32 @atoi(i8* %2) #2
32     store i32 %call, i32* %iteration_count, align 4
33     br label %if.end
```

Listing 2.5.: IR extract of Listing 2.2 with focus on the if condition.

shown in Listing 2.5 and Listing 2.6 as textual representations. This layer of code uses a *Static Single Assignment* (SSA) form (i.e. each variable is only assigned once, which makes compiler optimisations easier) and is already quite close to assembly language compared to the original C code, but uses the LLVM instruction set. This instruction set provides instructions at a slightly higher level of abstraction than assembly, uses unlimited virtual registers and makes no assumptions about the underlying hardware, nor does it use machine instructions (LLVM Project, 2023j). Section 5.1.1 has a more detailed description of IR.

Because IR code provides a layer between high level code and assembly, it is completely abstracted from the language in which the original application was written. Therefore, it is quite general and still relatively human readable. The last point is just a nice-to-have, as it does not affect CATO’s workflow, but it helps a lot with tool development and debugging.

As with AST, LLVM provides an API to analyse and iterate it, and this time also to modify it. In Listing 2.5 you can see both branches (**then** and **else**), where the variable `iteration_count` is set depending on user input. The second code snippet in Listing 2.6 shows the loop header, where the loop condition is evaluated, and the body. The former unary increment operator from the original code (line 12 in Listing 2.2) and the corresponding AST representation (line 51 in Listing 2.6) has now become more complex (from line 51 to 55 in Listing 2.4) because there is no equivalent IR statement. Since the generation of IR code is predictable, this is not a significant problem during the modification. Details on how to perform the transformation are given later in Section 5.1.2.

CFG The *Control Flow Graph* (CFG) is a directed graph constructed on the basis of the IR. It extends the IR by an assumed control flow that includes all possible branches. It is based on *Basic Blocks* (BBs) (cf. Section 5.1.1), which are preceded by a label in IR (e.g. Listing 2.5, line 23 or Listing 2.6, line 39). Each BB is

```
39 for.cond:                                     ; preds = %for.inc,
   ↪ %if.end
40 %3 = load i32, i32* %i, align 4
41 %4 = load i32, i32* %iteration_count, align 4
42 %cmp1 = icmp slt i32 %3, %4
43 br i1 %cmp1, label %for.body, label %for.end
44
45 for.body:                                     ; preds = %for.cond
46 %5 = load i32, i32* %result, align 4
47 %add = add nsw i32 %5, 1
48 store i32 %add, i32* %result, align 4
49 br label %for.inc
50
51 for.inc:                                     ; preds = %for.body
52 %6 = load i32, i32* %i, align 4
53 %inc = add nsw i32 %6, 1
54 store i32 %inc, i32* %i, align 4
55 br label %for.cond, !llvm.loop !4
```

Listing 2.6.: IR extract of Listing 2.2 with focus on the for loop.

represented in the CFG as individual nodes, which are connected according to the BB’s terminator instruction. The result is a (potentially) cyclic and directed graph, the CFG for the test application (cf. Listing 2.2) is shown in Figure 2.4.

For each BB or node, a block frequency analysis (LLVM Project, 2023i, 2023k) is performed by LLVM. It assigns a probability to each outgoing BB edge and estimates how many times a loop could be executed to evaluate a node’s frequency. This frequency is represented as the node’s colour, with a reddish colour indicating higher frequencies than bluish colours. This estimate could be used for a static analysis of the relevance of code snippets that CATO could focus on.

Even more advanced static analysis can be performed on the CFG. An example of this is the LLVM tool PhASAR (Heing-Becker, 2019; Schubert et al., 2019), which can be used to perform model checking on the CFG. It uses a context and flow sensitive approach to solving data flow problems (Reps et al., 1995). This allows information to be inferred about possible variable values at specific places in the code. As this is done statically, it can result in a large set of possible configurations for a single variable, but it can also reduce the number of possible states and allow additional information such as reachability to be gained.

This has been used for example in the master’s thesis from Marcel Heing-Becker, which I have supervised, to check if an application uses one-sided MPI communication operations correctly. To do this, several grammars were constructed to

represent different use cases, such as

- Initialise window before access
- Compatible window flavours
- Correct use of corresponding lock calls

A complete overview of the checks performed on the AST can be found in Heing-Becker, 2019, Tbl. 4.1. *Interprocedural, Finite, Distributive, Subset problems* (IFDS) have been used to construct each check, as they allow potential variable assignments to be considered on a subset of potential paths within different parts of an application.

However, there are two major problems with this approach:

State explosion Although a source code can have a fairly compact textual description, the set of potentially reachable states can become huge. Many high-level code constructs can lead to multiple potential assignments to variables, e.g. dynamic input, non-deterministic (random) behaviour, or simply conditional branches. If these constructs do not interfere with each other, but are executed sequentially, their space states will add up. Figure 2.2 shows the expansion of the state space when two independent variables x and y are assigned values from a set of size N and M respectively. In this case the state space reaches a worst case complexity of $O(N + M)$. This changes drastically if the second assignment of the variable y depends on the value of the first assignment, e.g. due to a conditional branch in between. In this case, the state space reaches a complexity of $O(N \cdot M)$ in the worst case, as shown in Figure 2.3. When the state space of an application grows exponentially, this is called *state explosion* and has a major negative impact on the performance of static model checking (Das et al., 2002; Jhala & Majumdar, 2009).

Rice’s theorem: Any non-trivial property cannot be decided algorithmically

There are ways to improve the model checking or construction of IFDSs to reduce the state space complexity, but there is no general solution for any application. Although the reduction may work for corner cases (e.g. if static assignments such as $x = 0$ are used) or if heuristics take effect. In general, it remains undecidable which specific value, for example, a variable will be assigned. This is supported by Rice’s theorem (Rice, 1953), which states that any non-trivial property of a language is generally not decidable. The problem is exacerbated when static code analysis is applied to code that uses library symbols that are dynamically linked at runtime. This further inflates the state space and degrades the validity of the static analysis, since more conservative assumptions must be used (Mock, 2003).

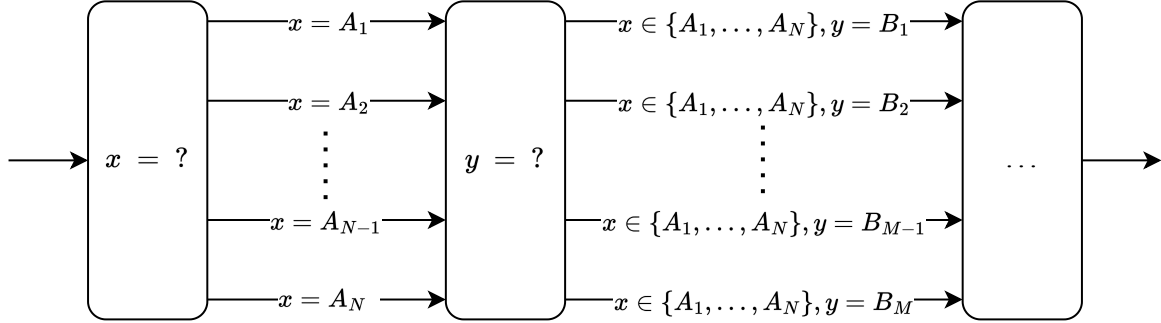


Figure 2.2.: Potential state space of two variable assignments, which are independent of each other.

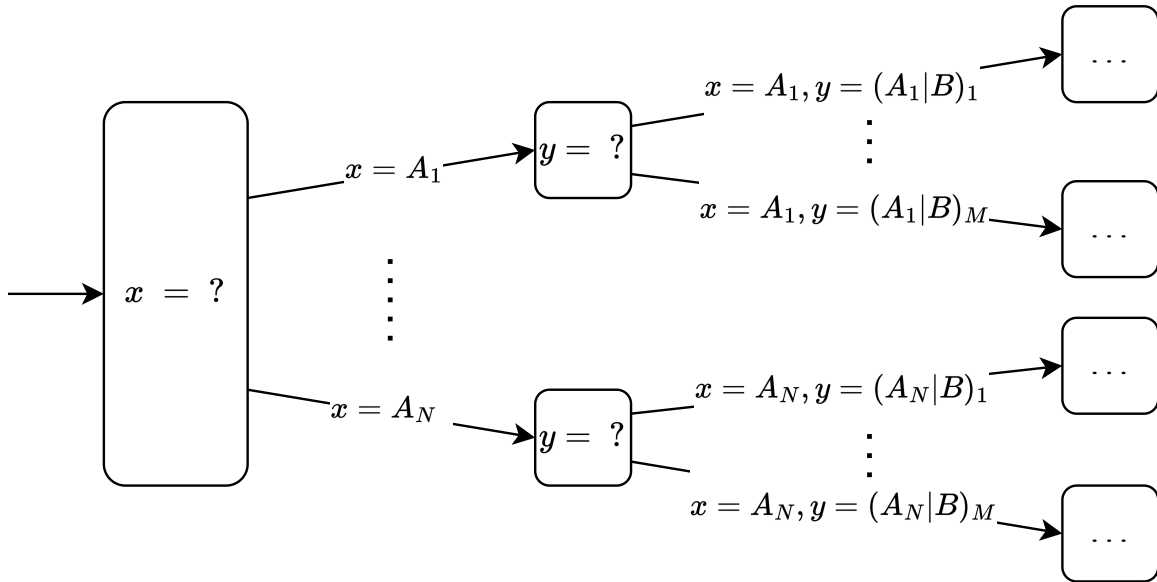


Figure 2.3.: Potential state space of two variable assignments, where the potential assignment space for y depends on the value of x and each space is disjunct.

Code transformation Like the AST, the CFG is not intended to be modified. It is intended to perform dataflow analysis, like a topographical sort of the code’s *Strongly Connected Components* (SCCs).

Binary Finally, changes could be made after the final step in the compilation chain has been executed to build the final binary. This can be done using static (offline, i. i.e. before execution) or dynamic (at runtime) binary instrumentation, which inserts new code into an existing binary. This is usually done to profile or trace an application without having to recompile it. However, it can also be used to modify or even replace calls, thereby inducing new behaviour.

There are several ways in which binary instrumentation can be performed, ranging from the simple to the complex and powerful. A simple approach, for example, is to wrap or overshadow all calls to a particular function. The GNU linker `ld` provides a flag `--wrap=symbol` which can be used to select `symbol` so that all undefined references are replaced by `__wrap_symbol`, which the user can provide. Another way, without the need to re-link the binary (which requires rebuilding the original source, or at least the availability of all the necessary object files), is to overshadow a function by providing an alternative implementation of that function. The loader can then be forced to prefer the function’s redefinition by setting the `LD_PRELOAD` environment variable.

Figure 2.5 demonstrates this with a simple example. The original application (cf. Listing A.3) simply prints a sine function to the terminal, which can be seen in Figure 2.5a. These values are taken from `sin` from the C maths library. A trivial replacement library (cf. Listing A.5) redefines the `sin` function with a constant value and is prioritised by calling `LD_PRELOAD=sin_redefinition.so`. `./original_application.x`; the result is shown in Figure 2.5b. Without making any changes to the original application, its binary has been instrumented and its behaviour changed.

This method is quite simple to implement, but has several disadvantages:

- All occurrences are replaced, a more selective approach is not possible.
- The function to be replaced must not be defined in the same translation unit, since only undefined references are replaced. As soon as the symbol definition is placed within the current translation unit, no external symbol reference is created; changing the load order during translation will have no effect on this symbol.
- This approach is rather crude and only works at the function level, it is not possible to reuse parts of the replaced function without copying the original source code into the function redefinition.

There are more advanced tools for dynamic binary instrumentation like BinOpt (Engelke & Schulz, 2020b), Instrew (Engelke & Schulz, 2020a) or CrossDBT (Li et al., 2022). A very well known one is **Valgrind**. It uses so-called *shadow values* to intercept every access to registers and memory, and can therefore perform additional

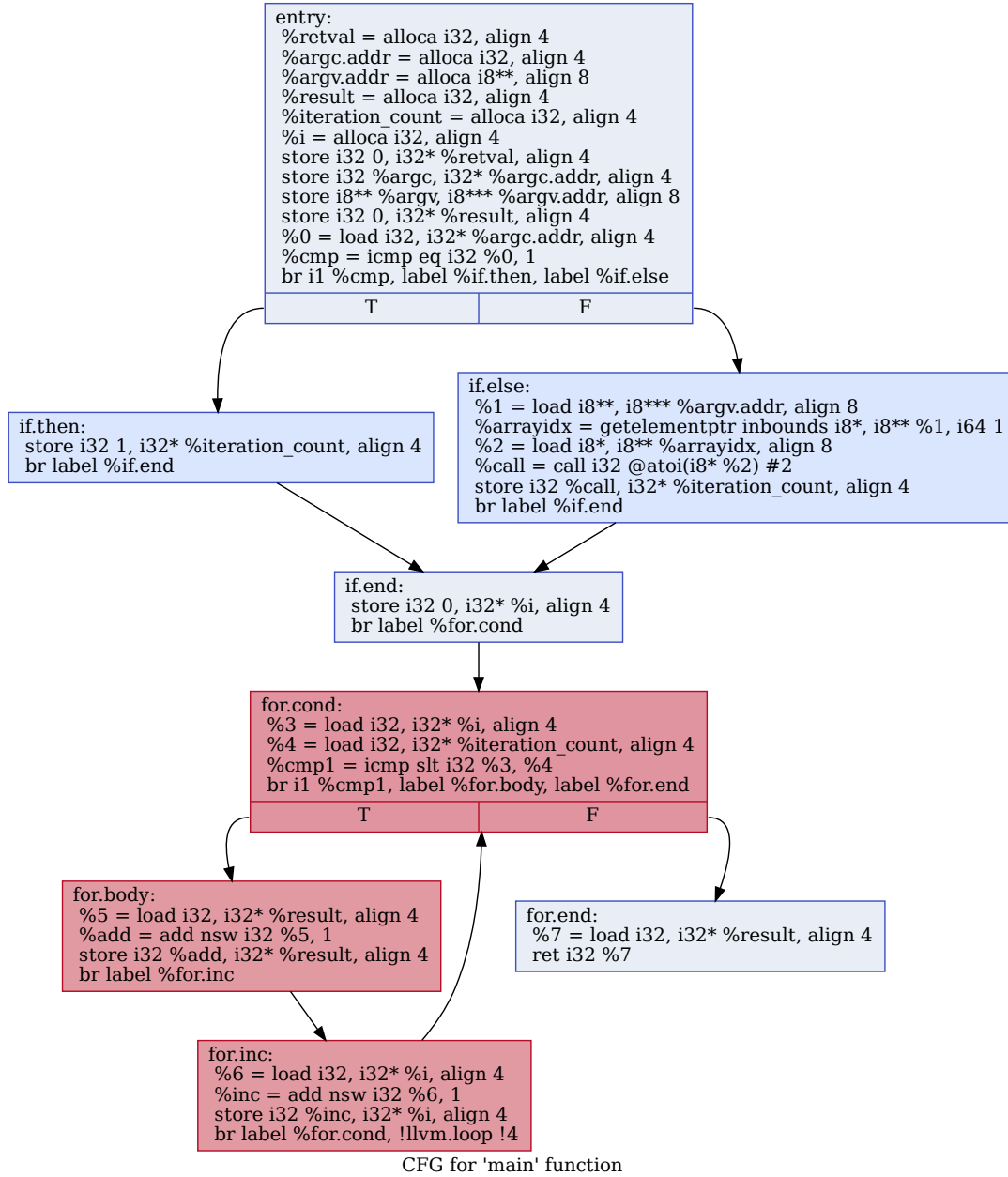
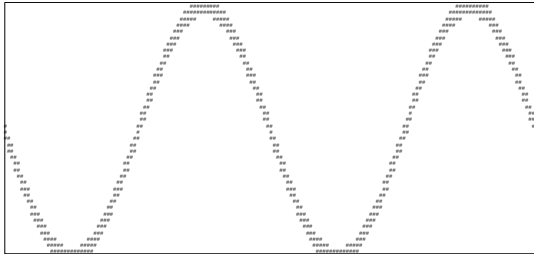
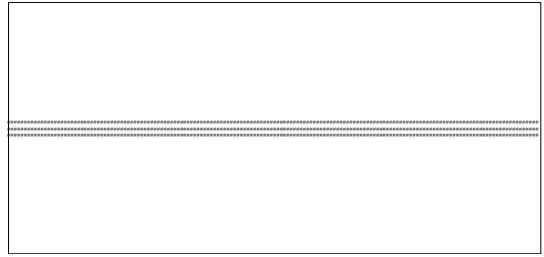


Figure 2.4.: Graphical CFG representation of Listing 2.2. The colour of each node (BB) is a sign of the expected execution frequency.



(a) Original terminal output of test application (cf. Listing A.3).



(b) Modified terminal output after overshadowing the original `sin` definition (cf. Listing A.5).

Figure 2.5.: Comparison of the outputs from the original application with the execution, where `sin` has been overshadowed during runtime.

checks in the background. In its earlier stages, it also used the `LD_PRELOAD` mechanism, but has now been extended to do the loader’s job. This allows `Valgrind` to provide its own address space manager and memory allocator, which are then automatically used by the target application (Nethercote & Seward, 2007).

A more detailed description of binary instrumentation techniques is given in (Priyadarshan, 2019).

Now that all the major LLVM layers have been presented, the most promising one can be chosen to work on implementing CATO. Working directly on the original source code has been ruled out, as has working on the AST. Using a static analysis approach has the advantage of being able to derive universally valid statements from the examined code. However, it is also a limiting factor in terms of the number of possible predictions, since other properties can very quickly become ambiguous (cf. Section 5.2). Therefore, a dynamic approach is inevitable, where properties are decided at runtime by simply checking their current value. Unlike the static approach, this does not provide possible future values and therefore limits the decision context to past and present values. Binary instrumentation has the advantage that the target application does not need to be rebuilt, but is more focused on analysis than transformation.

Choosing the IR layer also has some disadvantages, since the user has to rebuild his application, and the changes made on the IR layer are not as easy to read as on the high-level source code layer. In principle, this approach is independent of the language or coding style of the input application, and independent of the chosen machine backend. The syntax is always the same, but each front-end may do things a little differently. If the same source code written in C (Figure 2.6a) and Fortran (Figure 2.6b) is run through its respective LLVM frontend (`clang` and `flang`), the resulting IR codes (Listings 2.8 and 2.9 shows the unmodified IR code of the first BB) are still semantically equivalent, but have a slightly different sequence of operations. Not only are the names different, but some additional calling instructions are added (in this case to perform data type conversions for the Fortran `write` calls). Some level of abstraction that does not depend on a specific sequence of IR commands must therefore be preserved.

```
17  int a;
18  int b[2];
19
20  a = 10;
21  b[0] = 20;
22  b[1] = 30;
23
24  printf("%d\n", a);
25
26  for (int i = 0; i < 2; i++) {
27      printf("%d\n", b[i]);
28  }
```

(a) C code version

```
16  integer :: a, i
17  integer, dimension(2) :: b
18
19  a = 10
20  b(1) = 20
21  b(2) = 30
22
23  print*, a
24
25  do i = 1, 2
26      print*, b(i)
27  end do
```

(b) Fortran code version

Listing 2.7.: Same boilerplate code in two languages (C and Fortran)

On the other hand, the IR layer can be used to iterate, analyse and modify the original IR code, so that the modified IR can then be compiled into the final binary.

Figure 5.3 shows a selection of supported front-ends and back-ends. As long as the LLVM community keeps track of current developments, CATO does not need to be adapted and can rely on the LLVM infrastructure. For example, the emerging shift in processor architectures from *Complex instruction set computer* (CISC) to *Reduced instruction set computer* (RISC), which is becoming increasingly relevant in HPC, would already be incorporated (Shah, 2022).

2.4. Pass Design

Once the decision has been made to use the LLVM IR layer as the basis for CATO, the general construction of CATO can be done. Before the individual components of CATO are discussed in the next section, the general intentions need to be made clear.

CATO consists of several elements:

- Helper scripts for building dependencies, and the CATO LLVM (hereafter referred to as CATO pass).
- A test suite controlled by LLVM’s internal test suite `lit` (LLVM Project, 2023g).
- Wrapper scripts to run CATO to modify and build high-level source code.

LLVM makes it necessary to be careful about matching versions. As discussed in Section 3.1.1 LLVM is rather merciless in introducing API-breaking changes with the introduction of new major releases. Therefore, it is important to stick to a specific major release of LLVM and that the target application is built with the frontend of the same

```
10 define dso_local i32 @main() #0 {
11     %1 = alloca i32, align 4
12     %2 = alloca i32, align 4
13     %3 = alloca [2 x i32], align 4
14     %4 = alloca i32, align 4
15     store i32 0, ptr %1, align 4
16     store i32 10, ptr %2, align 4
17     %5 = getelementptr inbounds [2 x i32], ptr %3, i64 0, i64 0
18     store i32 20, ptr %5, align 4
19     %6 = getelementptr inbounds [2 x i32], ptr %3, i64 0, i64 1
20     store i32 30, ptr %6, align 4
21     %7 = load i32, ptr %2, align 4
22     %8 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %7)
23     store i32 0, ptr %4, align 4
24     br label %9
```

Listing 2.8.: Extract of IR from Listing 2.6a

```
15 define void @_QQmain() !dbg !3 {
16     %1 = alloca i32, i64 1, align 4, !dbg !7
17     %2 = alloca i32, i64 1, align 4, !dbg !9
18     store i32 10, ptr %2, align 4, !dbg !10
19     store i32 20, ptr @_QFEstack_array, align 4, !dbg !11
20     store i32 30, ptr getelementptr inbounds ([2 x i32], ptr
    ↪ @_QFEstack_array, i64 0, i64 1), align 4, !dbg !12
21     %3 = call ptr @_FortranAioBeginExternalListOutput(i32 -1, ptr
    ↪ @_QQc1.2E2F646966666572656E63655F666F727472616E2E66393000, i32 25),
    ↪ !dbg !13
22     %4 = call i1 @_FortranAioOutputAscii(ptr %3, ptr
    ↪ @_QQc1.5363616C6172207661726961626C653A, i64 16), !dbg !14
23     %5 = load i32, ptr %2, align 4, !dbg !15
24     %6 = call i1 @_FortranAioOutputInteger32(ptr %3, i32 %5), !dbg !15
25     %7 = call i32 @_FortranAioEndIoStatement(ptr %3), !dbg !13
26     br label %8, !dbg !16
```

Listing 2.9.: Extract of IR from Listing 2.6b

version. To make these requirements easier to handle, the build and run scripts of CATO make use of `spack`, which on the one hand allows all necessary dependencies to be built from source, and on the other hand makes it easy to comply with the version requirements. The first point in particular is a significant advantage, as CATO can be built easily in userspace without the need for root privileges. Thus, the user is not restricted to a specific Linux distribution or needs the help of the system administrator. Since CATO is used on applications from the HPC context, it is not uncommon for a user to work on variable HPC cluster systems, which are often restricted in terms of permissions and where the availability of the correct dependency versions is not guaranteed. Especially if the debug functionality of CATO is to be used, there is no way around compiling LLVM as a debug build yourself, because the `RelWithDebInfo` build (approximately 84 GiB¹) is much larger than the `Release` build (about 3.1 GiB²), and has much worse performance; therefore it is usually not available in the official distribution repositories.

2.4.1. Code Transformation

First of all, CATO is an LLVM pass (cf. Section 5.1.2) that analyses the user’s application code and automatically transforms it during the compilation phase. Unlike the original code, the rebuilt binary then makes use of new features provided by CATO, such as a hybrid parallelisation scheme, as well as parallel I/O and compression using `netCDF`.

Currently, only static information is derived from the IR level (cf. Section 8.2). However, it is necessary to perform code replacement based on the execution context. For example, it makes a difference whether a memory allocation requires only a single page of memory or most of the available memory. Unless simple static values are used (i.e. actual static numbers within a `malloc` call), the size is unpredictable at first sight. Therefore, CATO extends the IR code and encapsulates each potential code transformation in a conditional branch (cf. Mishra et al., 2020, Ch. III D). During the execution of the modified binary, the actual environment is safely set and the appropriate code replacement is executed. Listing 2.10 demonstrates the principle of this approach. In the original code in Listing 2.7a `statement_b` is executed and needs to be changed. The replacement in Listing 2.7b then adds as many branches as necessary to distinguish all relevant cases that have been previously elaborated and where different versions of the original statement are executed. For this to work well, it is important to consider each relevant case, for which an individual type of transformation must be performed by CATO. This can become quite extensive, but this drawback is mainly limited to the build time and file size of the modified binary. An example of this is how CATO handles variables within an OpenMP kernel, which will be discussed in Section 4.1. Another example in Section 4.2 shows how the user can influence the execution of the modified application via environment variables that are evaluated at runtime.

¹llvm@13.0.0 using `spack #e5bd319c19`

²See Footnote 1

```
1  int main() {  
2      // [...]  
3      statement_a;  
4      statement_b;  
5  
6  
7  
8  
9  
10     statement_c;  
11     // [...]  
12 }
```

(a) Unmodified code (vertical spaces added for easy comparability), where original **statement_b** is executed.

```
1  int main() {  
2      // [...]  
3      statement_a;  
4      if(condition_x)  
5          statement_b_x;  
6      else if(condition_y)  
7          statement_b_y;  
8      else  
9          statement_b_z;  
10     statement_c;  
11     // [...]  
12 }
```

(b) Modified code, in this case considering three different branches. On each branch, a differently modified version of **statement_b** can then be executed.

Listing 2.10.: Demonstration of how the code replacement can react according to the actual environment, even though the replacement code has already been inserted at compile time. In each conditional branch, a modified version of **statement_b** can be executed. This demonstration is only figurative, since the replacement is not performed on the original high-level code layer, but on the IR layer.

Equivalent Code Replacement

How the code transformation is actually performed differs between CATO’s components. In general, certain keywords in the original IR are searched for to identify significant kernels of code. Based on these findings, additional analysis (e.g. regarding dependencies) is performed, and then the actual replacement takes place, which is generally described in Chapter 4.

2.5. Summary

There are several abstraction layers of high-level source code that can be traversed and transformed: the original high-level itself, the AST, the IR, the CFG or even the final binary. All layers were examined and discussed for their advantages and disadvantages. From this discussion it seemed most promising to focus on the IR layer. It was decided to use a compiler-based approach to construct a LLVM pass that would provide the core functionality of CATO.

3. Background

The background chapter provides a comprehensive overview of the relevant concepts, setting the foundation for the subsequent discussions and analysis. According to the research questions from Section 1.2, there are three different areas of concepts, which will play a major role:

- Code Transformation
- Parallelisation
- I/O

For every concept, a decision must be made regarding the framework to prioritise and concentrate on. Particularly in the latter two domains, the selection is made while considering the target audience specified in Section 1.3.2, which imposes certain limitations. CATO requires that some frameworks are used, at least in a simplified form, in the domain scientist’s original high-level source code, since this is used to gather the necessary information. This is elaborated in Sections 3.1 to 3.3.

During the course of this work, a tool called AGSearch was developed. It has been used to search GitHub for specific repositories and analyse them locally to see if certain function calls are being used (Squar et al., 2022). This tool is used to get an idea of the reach of frameworks and to check if certain (advanced) features are being used or ignored by the community. These analyses are specifically focused on repositories hosted on GitHub. As a result, AGSearch excludes repositories hosted on other platforms like *GitLab* or *SourceForge*, as well as repositories that are not publicly accessible, such as the ICON model repositories.

It is important to approach the results of AGSearch with some caution, as the tool does not differentiate between source files, comments, or documentation within a repository. All files are treated equally in the analysis. However, it is assumed that the statistical distribution of meaningful and meaningless hits for each keyword searched within the repository remains consistent. With this assumption, the proportions obtained from the analysis should still hold significance.

3.1. Source Transformation Techniques

Source-to-source transformation (S2S) is a technique for taking high-level source code and modifying it. S2S is a common practice employed within compilers. It involves taking high-level source code as input and performing various optimisations and transformations.

The modified code can then be transformed into the final executable binary code. This technique can also be used deliberately by a user to automatically modify their high level code. In this case, it is most likely to be used to move high-level code from one language to another, while keeping its semantical behaviour intact. However, a survey by (Milewicz2021) showed that there are some negative perceptions in the community:

- S2S interferes with the final compiler optimisations.
- S2S tools are difficult to customise.
- S2S interferes with the otherwise solid compilation process.
- S2S requires parsing, which is difficult to manage.
- S2S is just a placeholder technology until it is replaced by something more robust and competent.

Milewicz et al., 2021 already refutes these perceptions as far as possible. Moreover, they do not really apply to CATO: The users of CATO do not need to concern themselves with customizing or configuring the parsing step of the tool. CATO is integrated into the standard LLVM compilation process, and is integrated into the Pass Manager workflow as early as possible to minimise any interference (cf. Section 5.1.2). The potential benefits of using CATO for S2S transformations outweigh these perceptions.

Given unlimited time and effort, the efficiency of manual code transformations can always match that of automatic code transformations. As time is a limited resource, automatic code transformations are a valuable and unavoidable asset for improving efficiency at low cost.

3.1.1. Compiler

In Section 2.2, the decision was made to use a compiler-based approach as a robust and feature-rich foundation. Using the ability of a compiler to apply transformations automatically is an important technique within HPC (Bacon et al., 1994). Several compiler frameworks have gained a solid reputation and are widely used in both production and research environments. These frameworks provide stability, reliability, and often offer unique benefits for various use cases. These include, but are not limited to ROSE (Quinlan & Liao, 2011), Cetus (C. Dave et al., 2009), Mercurium (Balart et al., 2004), Insieme (Jordan et al., 2013), OpenARC (S. Lee & Vetter, 2014), Xevolver (Komatsu et al., 2020), Omni (Sato et al., 1999), P_{Lu}To (Bondhugula et al., 2008), Clava (Bispo & Cardoso, 2020) and Prospect (Süßkraut et al., 2010).

Some focus on OpenMP or OpenACC, others on automatic parallelisation. What they all have in common is that they are not so much general purpose compilers as they are compilers for very specific use cases. The main general-purpose compiler frameworks

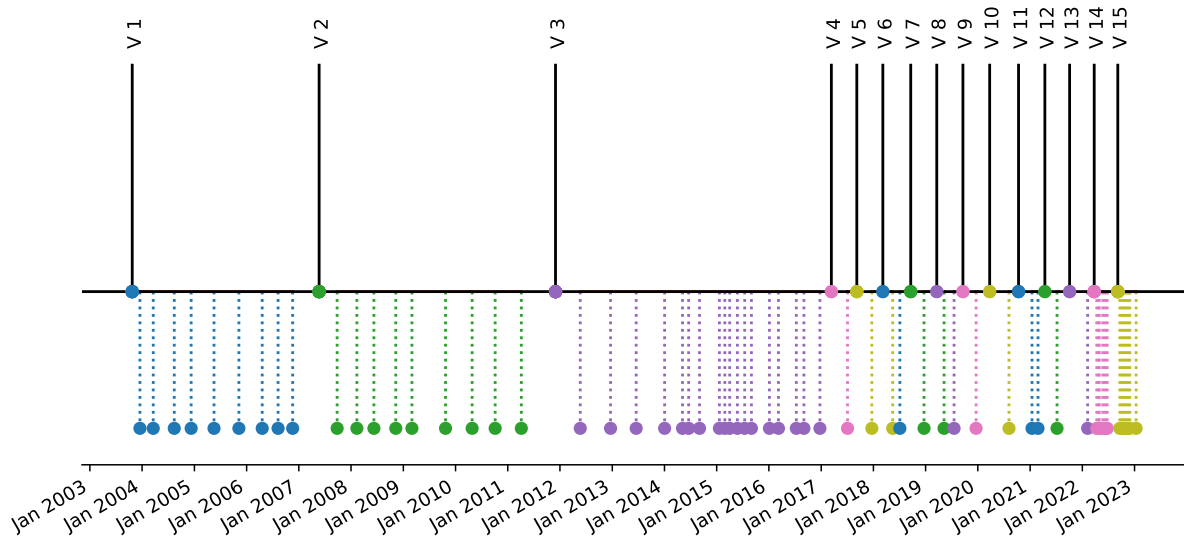


Figure 3.1.: Visualisation of the temporal distribution of LLVM releases. Major releases ($V\ x.0.0$) are printed on the baseline, and minor releases ($V\ x.y.0$ and patch releases ($V\ x.y.z$) are printed below the baseline. Since version 4.0.0, LLVM has received a major release approximately every 6 months (LLVM Project, 2023d)

that are open source are the GNU compiler suite and LLVM. GCC is older than LLVM and supports more high-level languages than LLVM, although it is hard to quantify.

For example, Fortran is well supported by the `gfortran` frontend, while the LLVM Fortran frontend, `flang` is still under development and not yet ready for production use (LLVM Project, 2023c). However, this is only a matter of time as development is progressing rapidly.

However, using LLVM as a base is advantageous as it is comparatively easy to jump into and less monolithic than GCC. It has a large community and many research related tools are built using LLVM (Desaulniers, 2017). A selection of LLVM based research tools will be discussed later in Chapter 6. Both compiler frameworks use a *Intermediate Representation* (IR): LLVM IR and its GNU counterpart GENERIC (GNU, n.d.; LLVM Project, 2023j). The IR acts as an intermediate layer that is independent of the high-level language used or the machine backend targeted. This allows for simplified and reusable code analysis and transformation techniques. Both compiler frameworks allow user-defined routines to be dynamically loaded via a LLVM pass or `gcc` plugin to traverse and transform IR. There is even a `gcc` plugin available called DragonEgg to replace middle-end and back-end components from `gcc` with equivalent components from LLVM (LLVM Project, n.d.-a). By leveraging the optimisation and code generation capabilities of LLVM, it becomes now possible to provide support for all the languages covered by GCC, expanding their potential.

A drawback of LLVM is its tendency to introduce significant API-breaking changes with each major release, occurring every six months (cf. Figure 3.1), which can hinder

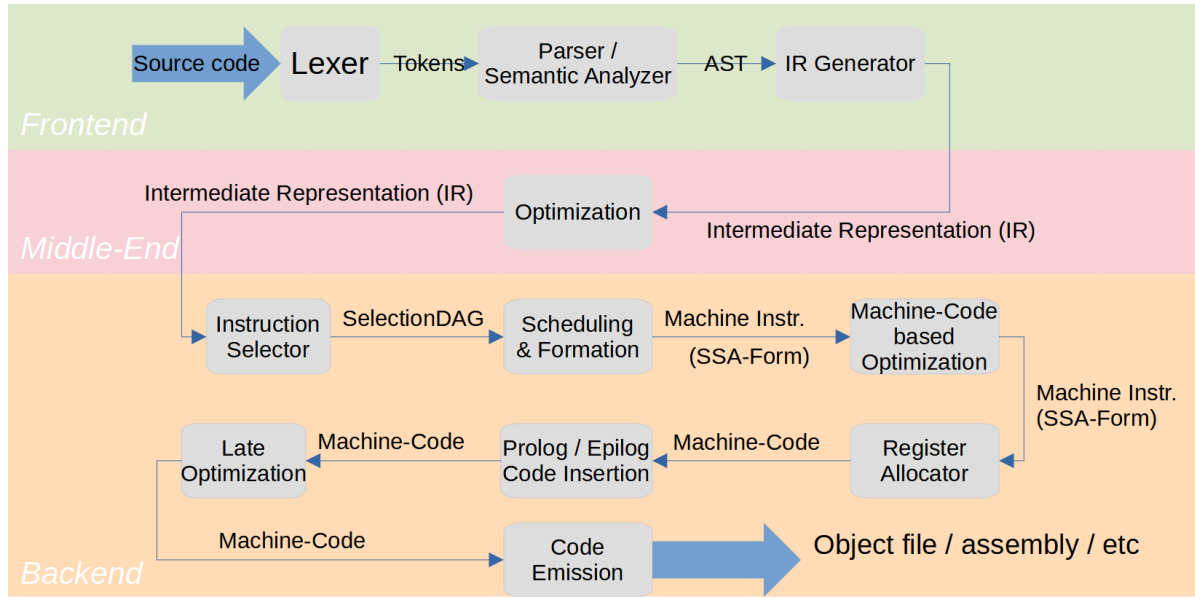


Figure 3.2.: General code transformation steps following the general composition of a compiler (cf. Figure A.1). Figure is from Winkler, 2022.

compiler developers. Adapting the developed tools is necessary when utilising new features of LLVM. Alternatively, if one chooses not to adapt the tools, they may miss out on the benefits of new features and instead opt to install a specific release version of LLVM, which can be done without requiring root privileges.

Ultimately, the choice between LLVM and GCC is a matter of personal preference, as there is no clear argument favouring one over the other. However, in the context of HPC research tools, many are built using LLVM, which has a better documentation than that of GCC. Therefore, LLVM has been selected for this project.

3.2. Parallelisation Techniques

In modern HPC hardware, there are several levels of parallelism. Some need to be explicitly addressed by the user, some are used automatically by the OS or the hardware. Figure 3.3 gives a rough overview of potential sources of parallelism. The following description is not exhaustive (Schmidt et al., 2017).

Starting from the bottom, there is a core with its components such as the *Arithmetic Logic Unit* (ALU) and registers. The execution of an instruction requires several steps (*Fetch, Decode, Execute, Memory, and Write*), all of which can be executed in a parallel pipeline if not a single instruction, but a stream of instructions, enters it. This is instruction-level parallelism. If a core has a vector unit, a single instruction can be executed on multiple data simultaneously.

One level up is the CPU, which can have multiple cores. In addition to parallelism within a single core, multiple cores can now be used simultaneously. On a motherboard with more than one *socket*, multiple CPUs can be used simultaneously. More parallelism

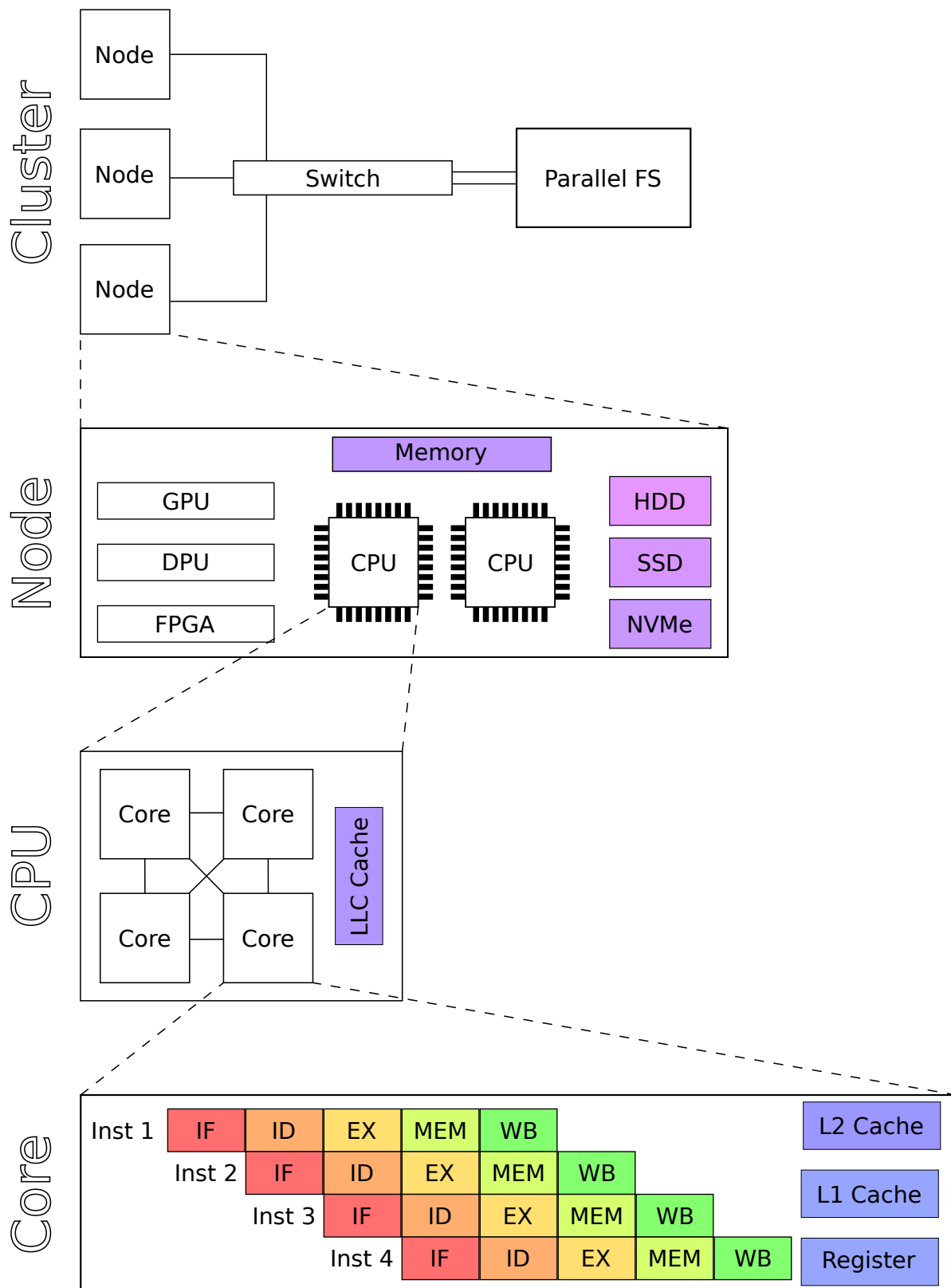


Figure 3.3.: Overview of some levels of parallelism from a single instruction up to a cluster.

can be achieved by using accelerators (e.g. GPUs), but these are not used in this work because many ESS codes do not use them prominently.

Going up one level, we move from a single compute node to a whole cluster of compute nodes connected by some kind of network. Theoretically, you could go up another level and connect clusters together to form a grid or, more abstractly, a cloud, but this is not covered in this work.

These levels of hardware parallelism can now be used to run applications concurrently. Some levels can only be influenced indirectly by the developer, since the hardware itself or the OS takes care of this. Other levels can be used explicitly. Within the application, he can choose between two paradigms, which can also be mixed:

- **Code-level parallelism:** The application can be divided into successive phases (e.g. preprocessing, computation, postprocessing) and then executed in parallel. Using a data stream, it is then possible to build an execution pipeline in which each actor performs different operations. This is similar to the instruction-level parallelism discussed earlier.
- **Data-level parallelism:** The data is split and processed simultaneously, with each actor performing the same operations.

The central execution unit in modern computer systems is a *process* representing an application. This abstraction contains not only the actual instructions of the application, but also a snapshot of its current execution, for example a reference to the current instruction or memory allocations. Within a process there is a master thread, but more threads can be forked during execution (OpenMP Architecture Review Board, 2021, Ch. 1.3). The user can control this by spawning new processes or forking new threads within a process. Threads are bound to the same shared memory address space assigned to their process, and are therefore confined to a single node. Processes are more independent (they can communicate with each other) and can therefore be distributed across multiple compute nodes and operate on distributed memory.

3.2.1. Shared Memory

There are many frameworks that allow developers to parallelise their code using threads, for example OpenMP, TBB and *POSIX Thread* (pThread) (Stpiczynski, 2018). This work is based on OpenMP 5.2, which offers many advantages:

Easy usage OpenMP uses pragmas, which are interpreted by the preprocessor, and a few runtime routines and environment variables. So, by using simple compiler pragmas, the user does not write the actual parallelisation code, but only gives directives to the compiler. The compiler and the OpenMP runtime library take care of thread creation and the distribution of data and computation. Parallelising a trivial workshare is easy and can be kept fairly short. Because of its low threshold, OpenMP is very popular - especially scientific software makes heavy use of it.

Incremental use Another aspect of using pragmas is that a serial application can easily be upgraded with OpenMP parallelisation. The programmer does not need to add extensive parallelisation logic beforehand, nor do they need to rewrite all the code at once – the application can be adapted incrementally. Therefore, the resulting code looks similar to the serial version and debugging is easier (Basumallik & Eigenmann, 2005).

If the application is compiled without the OpenMP flag, all pragmas are ignored and the original serial application is built. Only the few runtime routines need to be encapsulated with `#ifdef` macros. This makes it easy to gradually add the OpenMP code and compare the serial and parallel applications.

Availability To give an example of how widespread OpenMP is, a survey of He (2015) was consulted. As Figure 3.5 shows, OpenMP is quite popular on *National Energy Research Scientific Computing Center* (NERSC), making up the second largest share. Since it is closely coupled to the compiler, and all major compilers have OpenMP support, it is available on most HPC systems. If an application is to run on a machine without an OpenMP runtime, it should be sufficient to build the application without the OpenMP flag - there is no need to provide an additional version without OpenMP pragmas in this case. Also, OpenMP is available for the major languages used for scientific applications: C, C++ and Fortran.

Using AGSearch to search GitHub gives a ranking of the most used OpenMP features. 9430 repositories were found, of which 7418 use C or C++ as their main language (about 78.7%). Figure 3.4 shows the most used directive, which is `pragma omp parallel`, followed by `pragma omp parallel for` and `pragma omp for`. On the eighth position is `pragma omp task`, which is most likely used with GPUs, which has been omitted in this work. This shows that over 88% of the repositories found focus on `pragma omp parallel` or `pragma omp parallel for`. This is a strong indicator to focus on these first when developing CATO to get more coverage.

3.2.2. Distributed Memory

In Chapter 6 about related work, there are many frameworks that can be used to work on distributed memory with some kind of coordination. MPI is quite popular in HPC, on NERSC it is the main framework used (cf. Figure 3.5). Looking through GitHub with AGSearch found 10791, of which the majority use C or C++ as their main programming language (cf. Figure 3.7a).

MPI can be used for both intra-node and inter-node communication, allowing the user to run their application on multiple compute nodes. By default, processes do not share the same address space, so messages are passed between them to exchange data over the interconnect. An MPI implementation provides a convenient abstraction between the application and the network. The most commonly used interconnect types on HPC systems are according to the *Top500* list of June 2023 (cf. Figure 3.6) InfiniBand and Ethernet. There are several implementations of MPI (e.g. *MPICH*/*MVAPICH* or *Open*

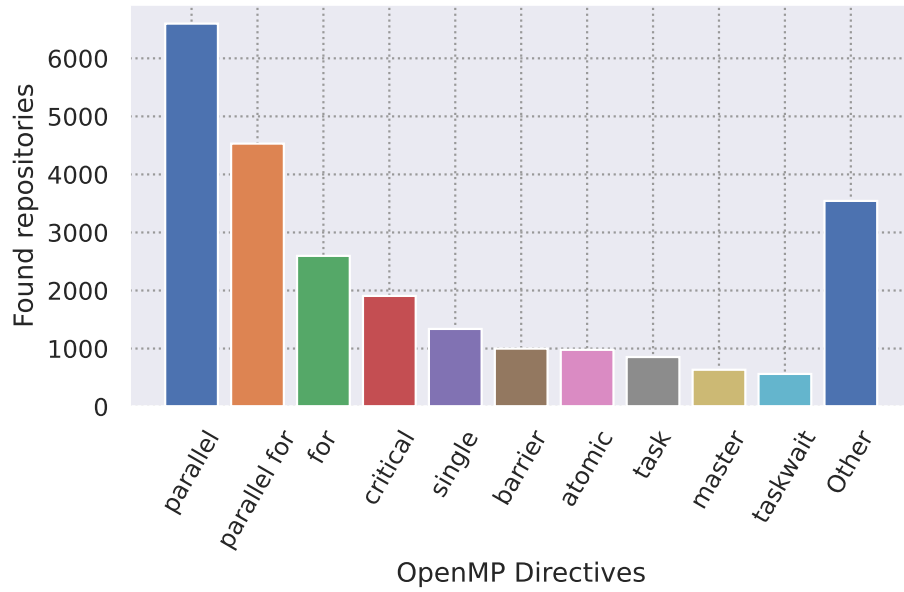


Figure 3.4.: Popularity of OpenMP directives and functions derived from 7418 public GitHub repositories.

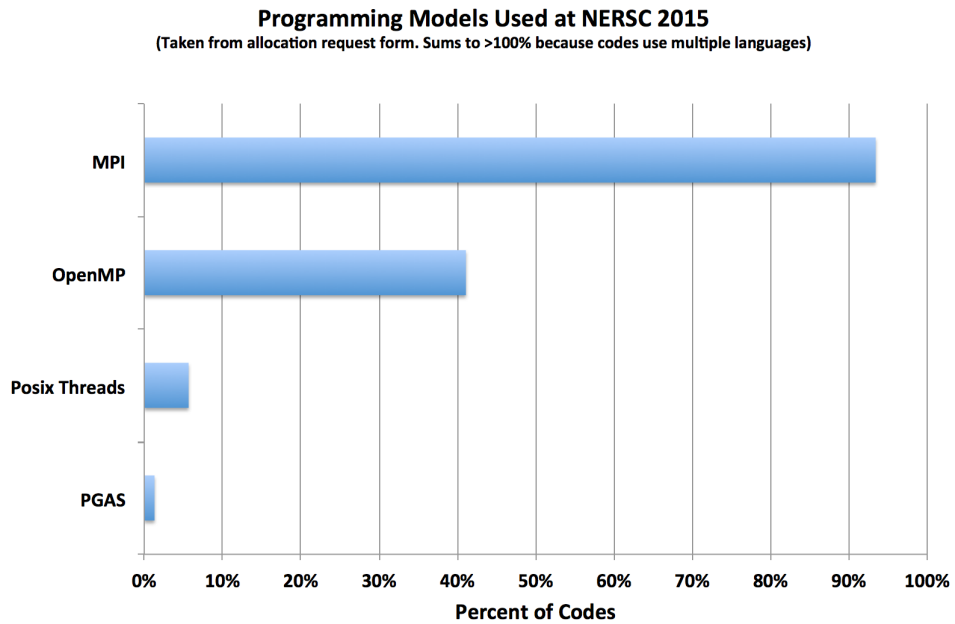


Figure 3.5.: Share of used parallelisation frameworks at NERSC. Figure is from He, 2015.

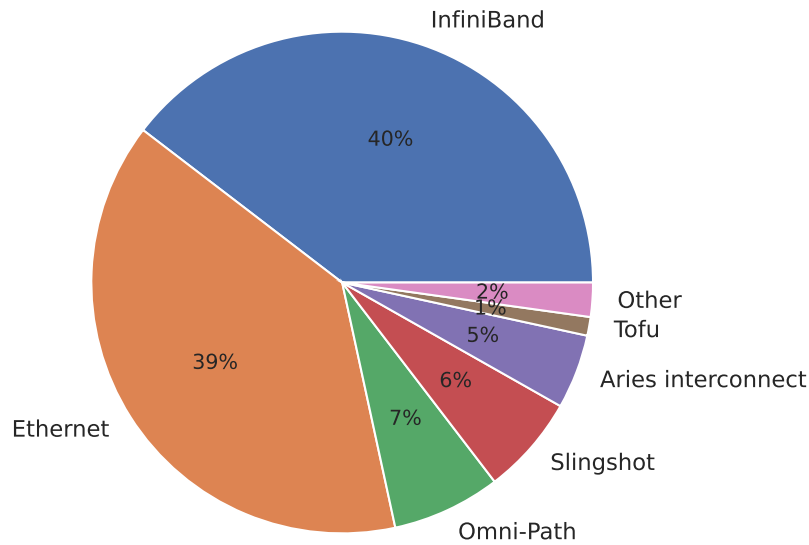
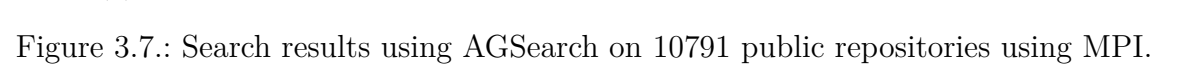


Figure 3.6.: Overview of the interconnect architectures used on the *Top500* list as of June 2023 (Strohmaier et al., 2023).

MPI) that can be used on these interconnects. The implementation uses the appropriate API (e.g. *sockets* or *Verbs*) in the background, the user API is unaffected.

Communication with *MPI* can be via *point-to-point communication* or *collective communication*; communication can be blocking or non-blocking (there are more categories, but they will not be discussed further). Another important addition to *MPI* is one-sided communication, which is practical when unpredictable access patterns are used, but the data distribution is rather static (MPI Forum, 2021, Ch. 12). The process that initiates the communication is the *source* rank, and the process that has the accessed data is the *target* rank. The novelty of using one-sided communication is that only the origin rank needs to perform the communication operation. The target rank only needs to actively participate in synchronisation calls if *active target communication* is used. If *passive target communication* is used, the target rank does not even participate in the synchronisation. The latter is similar to using shared memory. For this to work, each participating rank must explicitly mark the memory if it is to be accessible via one-sided communication. This is a so-called *window*, of which there are several flavours. One-sided *MPI* operations have been added in *MPI* 2 and extended with *MPI* 3. Figure 3.8 shows an (incomplete) overview of one-sided *MPI* operations (represented as leaves in the tree). Most relevant *MPI* implementations have implemented one-sided communication (MPI Forum, n.d.). Still, it is not so popular like two-sided *MPI* communication, since only 4% of all found repositories also used one-sided *MPI* operations (cf. Figure 3.7b). One reason for this is probably that it is more difficult to use than two-sided *MPI* operations.

Using one-sided *MPI* communication increases the difficulty of development. Notwithstanding the considerable number of operations available for communication and synchronisation, the correct handling of epochs can become quite complicated. It is necessary to keep track of the epochs opened and to be careful not to damage the integrity



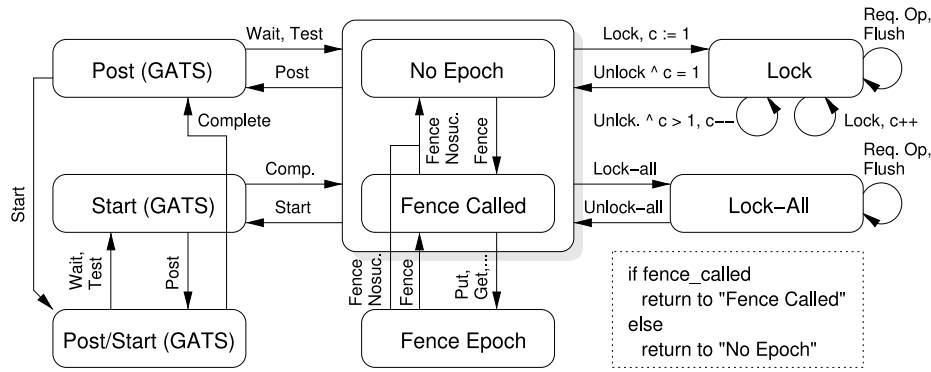


Figure 3.9.: Overview diagram of MPI one-sided ‘locking’ mechanisms. Figure is from Dinan et al., 2016.

or consistency of the data. The developer must be particularly careful with the latter, since the one-sided communication differs between the two memory models (*RMA unified* and *RMA separate*), depending on the hardware actually used. In addition, windows can be accessed during an open epoch using MPI operations or local reads or writes, which must be done carefully. Figure 3.9 represents a state machine showing the correct use of synchronisation for a single process on a single window. Using multiple processes on multiple windows at the same time increased the complexity.

MPI communication can be performed via *Remote Direct Memory Access* (RDMA), which can provide lower latency and higher bandwidth. RDMA can be used if a compatible interconnect such as InfiniBand or Cray Slingshot (using *RDMA over Converged Ethernet* (RoCE)) is available (MacArthur et al., 2017; Sensi et al., 2020). In addition, it can facilitate the inclusion of shared memory and irregular access patterns. On top of this one-sided MPI communication can potentially achieve better latency values than two-sided communication, because the target process is less involved (Gerstenberger et al., 2013).

3.3. Input/Output

I/O in HPC often takes a back seat to compute performance, but can play a significant role in performance if the application is memory bandwidth bound or particularly I/O bandwidth bound. Two approaches to reducing the I/O load are parallel I/O or compression. Both can be used to reduce the load on the I/O interface of a single compute node, but come with a few postulates:

- **Parallel I/O:** Multiple compute nodes are involved so that the aggregated bandwidth of their I/O interfaces can be used to access multiple I/O servers concurrently. The application’s I/O operations are suitable for partitioning.
- **Compression:** The data is compressible ($CR > 1$) and the runtime overhead caused by (de-)compression is either acceptable or only uses idle CPU resources.

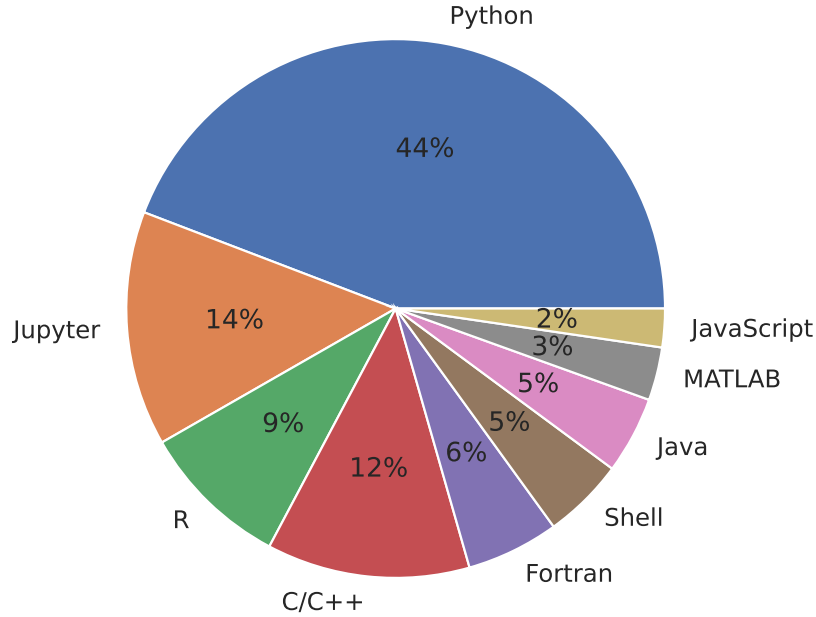


Figure 3.10.: Distribution of languages using netCDF.

The following description in Section 3.3.1 gives a brief overview of how I/O can be handled in the context of HPC. This sequence fits a specific HPC system (in this case Levante at DKRZ), details may differ on other systems. However, the general concept remains the same. In Section 3.3.2 the potential benefits of compression are discussed.

In both cases, the netCDF library is used for demonstration purposes, which is a highly relevant I/O library in the field of ESS (cf. Section 1.2.2). AGSearch was used to search public GitHub repositories to show how netCDF is used within the community.

345 repositories use C/C++ or Fortran as their main programming language. Figure 3.10 shows that the majority use Python or *Jupyter Notebook* (which usually uses a Python kernel). Since Python is easy to learn and use, and popular among data scientists, many Python repositories are probably used for pre- or post-processing of netCDF files (e.g. to create visualisations). CATO is not suitable for working on Python code, so the most common language must be omitted.

Looking at the frequencies of found netCDF functions in Figure 3.11 shows that usually parallel I/O or compression is not used. Similar functions are grouped together: For example, there are many functions to read data with different granularities (single, array, strided array or mapped array) on different datatypes, which have been collected in *Read some*. Their counterparts are the read functions in *Read all*, which read the whole variable and differ only in the datatype. Three lessons can be learnt:

1. Usually only serial I/O is used.
2. Reading the whole variable rather than just a part has a slight advantage.
3. Functions related to compression (alignment, chunking, filtering) are rarely used.

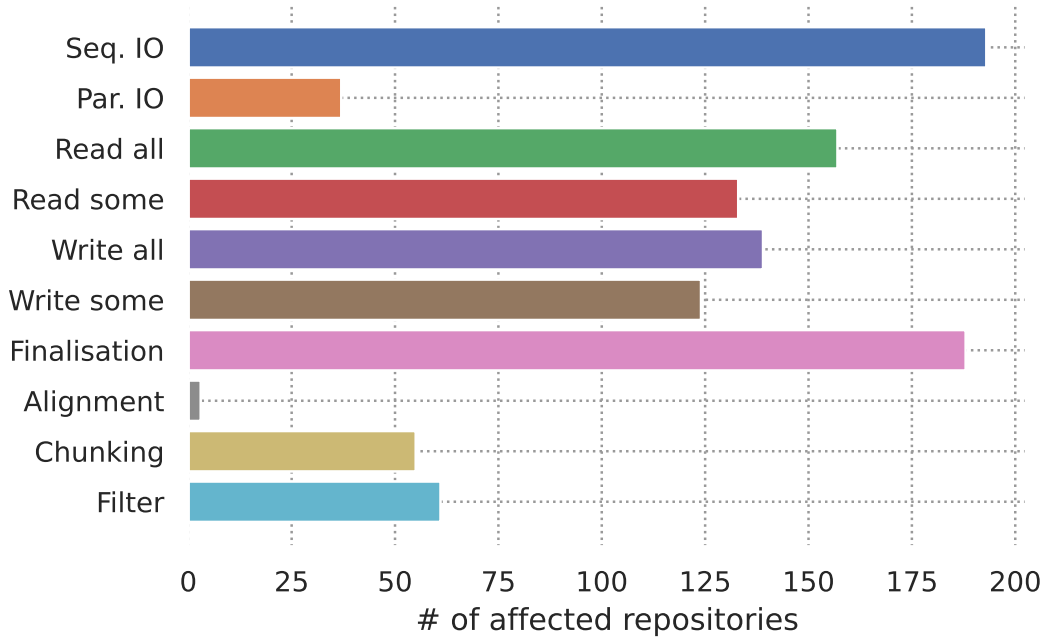


Figure 3.11.: Frequency of netCDF function usages divided into groups.

It can therefore be assumed that there is fundamental potential for improvement, which CATO could provide.

3.3.1. Parallel Input/Output

Portable Operating System Interface (POSIX) provides an interface for performing file I/O, which provides all the basic operations required. It provides a file handle through which the OS grants access to the data on the storage device. The same file can be accessed concurrently, the user is then responsible for maintaining data integrity and consistency. There are other I/O libraries such as MPI-IO, HDF5 or netCDF which add an additional layer of abstraction to assist the user. Be it by collecting multiple I/O calls before the actual (expensive) system call is made on a batch or by taking care of proper synchronisation.

The general procedure, which is displayed in Figure 3.12 is as follows:

1. The HPC application wants to access data in a file in parallel.
2. An I/O library like netCDF 4 is used, which provides functions to open and access a file.
3. NetCDF uses its HDF5 backend.
4. HDF5 forwards the request to its MPI-IO backend.
5. Often MPI-IO uses POSIX to perform the parallel file access.

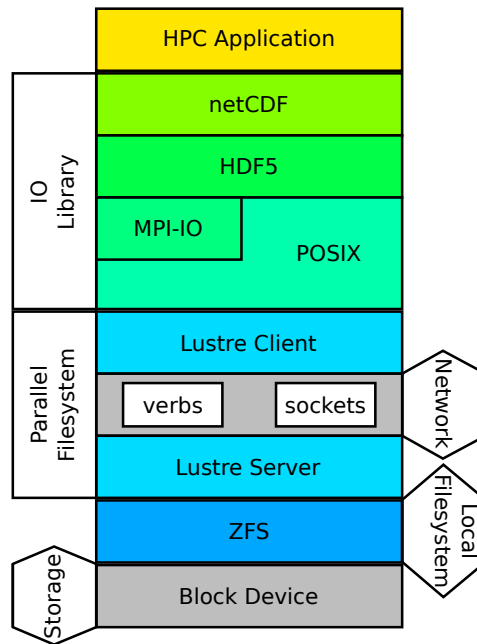


Figure 3.12.: Layers of the I/O stack through which data passes until it is written to the actual block device.

6. The file does not reside on a storage device that is physically integrated into the compute node. Instead, the file location is only mounted within the compute node's local *File System* (FS).
7. The Lustre client on the compute node attempts to forward the I/O request over the interconnect.
8. A network API such as *Verbs* is used to deliver the request.
9. If the Lustre client does not know, where the file stripes are stored, the file layout is requested from the *Metadata Server* (MDS) server (cf. Section 4.2.1). Then a request is sent to the correct *Object Storage Server* (OSS).
10. On the OSS the request is forwarded to the *Object Storage Target* (OST) on the right, where the local file system, ZFS or *ldiskfs*, performs the I/O operation on the block device.

In order to benefit from this procedure, which is mostly transparent, the user must ensure that his data is stored in a parallel FS such as Lustre and that the file stripes match the use case (e.g. the number of I/O processes).

3.3.2. Compression

Regardless of whether an application uses parallel I/O, it can benefit from compression. In addition to a more efficient use of memory, computation and I/O can benefit from

compression by increasing the effective bandwidth beyond the physical limits of the hardware.

If lossless compression is used, the data can be recovered by decompression without loss of information. The only limit to achieving higher CRs is how compressible the data is, as the entropy of the data sets a theoretical limit. Entropy is a measure of the information density of data (or a bit). A high entropy means that the bits are very important, and taking away a few would severely damage the data's integrity. Conversely, low entropy means that the information density is low and the contribution of a single bit is unlikely to be very significant. Compressing data means removing unnecessary bits or finding a shorter representation for a set of bits. The information content remains the same while the data size is reduced, so the entropy of the compressed data increases. When the maximum entropy is reached, the data cannot be compressed any further (Balakrishnan & Touba, 2007; Hansel et al., 1992).

Under certain conditions, the use of a lossy compressor may also be appropriate. In this case, data is potentially lost by applying a non-bijective function. This does not necessarily increase the entropy because information is potentially lost. However, it can be used in conjunction with another compressor that has a higher degree of efficiency. Especially for floating-point data, which is usually hard to compress, this can be a real advantage.

Most computer systems use the *IEEE Standard for Floating-Point Arithmetic* representation, which uses one *sign* bit, 8 *exponent* bits and 23 *mantissa* bits to represent a 32-bit floating point number ('IEEE Standard for Floating-Point Arithmetic', 2019). Figure 3.13 shows the information density of bits of different CAMS variables to demonstrate the potential benefits of using compression. Grey boxes provide no valuable content to define the value of the variable, so these bits provide no valuable information. In this case, they could simply be omitted without changing the value of the variable. Because of the fixed structure of floating-point variables, they cannot simply be left out, but since they are *zeroed out*, applying lossless compression can definitely reduce the number of bits needed. If some error is acceptable (which could be the case if the use case does not require the precision provided, or the precision is not needed due to data noise induced by the measurement methods used), then lossy compression can also be used. Most variables show great potential in this case as well, as some have many bits that store only the last percent of information.

Neglecting the impact on runtime and the additional load on CPU and memory, lossless compression can always be applied transparently. The data is not damaged by the (de-)compression. On the other hand, this is not possible with a lossy compressor. It depends on the use case how much information loss is tolerable and this can only be decided by the domain scientist who knows what the data is used for (Cappello et al., 2019).

Choosing a Compressor

Using the right compressor is heavily depending on the use case and the user's requirements:

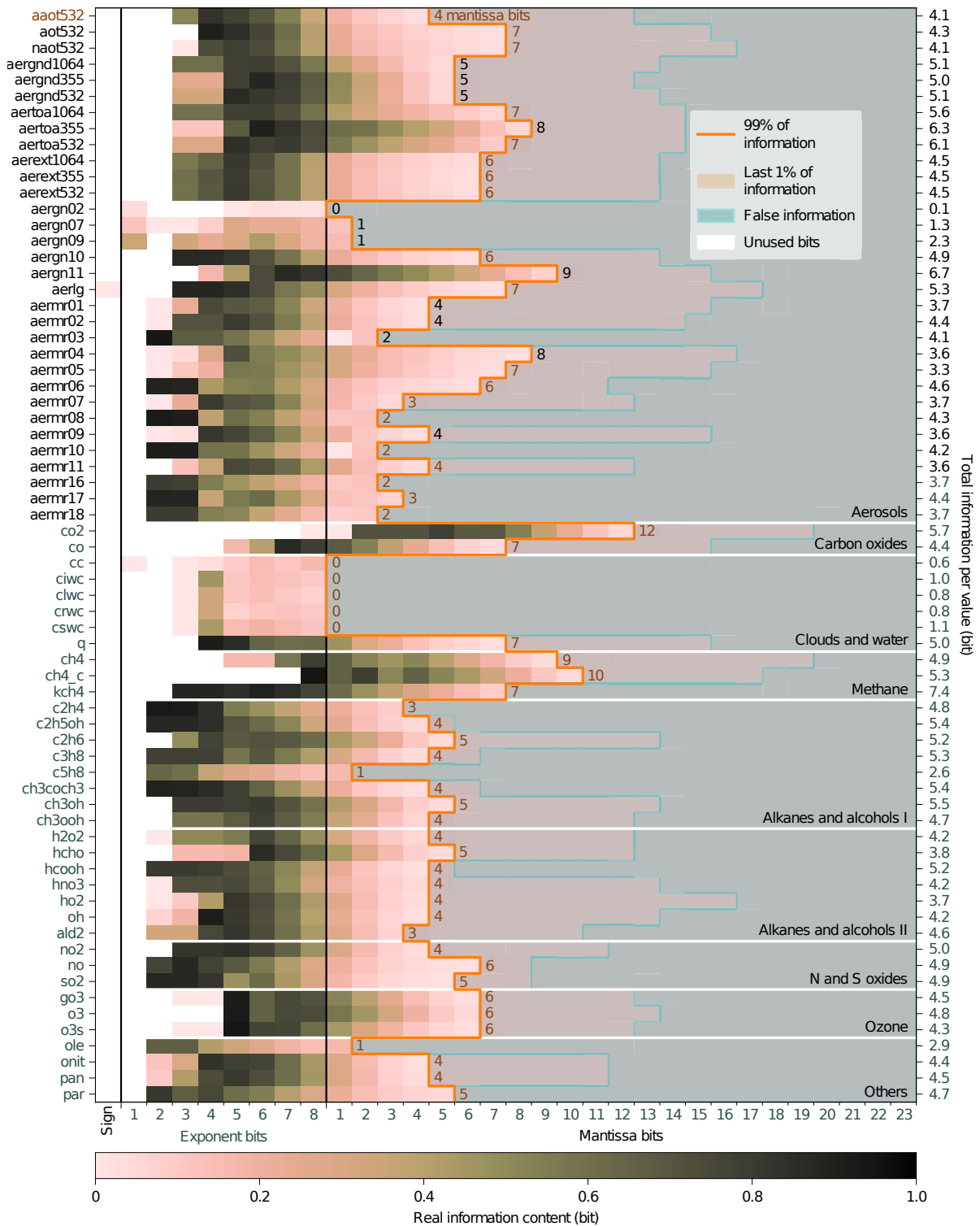


Figure 3.13.: Analysis by Klöwer et al. of the significance of bits that are part of the floating-point representation of variables from *Copernicus Atmosphere Monitoring Service* (CAMS) (provided by Inness et al. (2019)). While the exponent bits (which indicate the size of the variable) are quite important (i.e. their bits have a high information content), the mantissa bits are less significant. Figure is from Klöwer et al., 2021, Fig. 2.

- Is a lossless algorithm required or can a lossy algorithm also be used?
- Which compression metric has priority? *Compression Ratio* (CR), compression speed or decompression speed?
- Is the application compute-bound or memory-bound? Can (un)compressing data only use idle resources, or can runtime performance be compromised in favour of compression?

These questions are not easy to answer. Whether the algorithm can be lossy or must be lossless cannot be decided automatically, but only by the user who knows the requirements of his application. In addition, there are many compression algorithms available, which may already have different compression and decompression speeds. Improving the compression ratio usually increases the time needed for (de-)compression and vice versa. So every algorithm already has a bias, in which case it works quite well. There are algorithms that can be rejected if there are alternatives that outperform them in every metric. However, this still leaves a wide range of algorithms. To make matters worse, their metrics can change depending on the data they are applied to. For example, it can make a significant difference if the data uses integers or floats, or if it has high entropy (e.g. due to noise induced by your measurement process).

NetCDF has some lossless compressors and a lossy preprocessor built in, which will be discussed later in Section 5.3.2. More compressors can be used by adding them as HDF5 filters. Duwe et al., 2020, Ch. 1.1, 1.2 lists many available compressors and the methods they use, some of which are also available as HDF5 filters (e.g. SZ and ZFP (Di & Cappello, 2016)). In order to use a compressor, the variable definition in the source code must be adjusted by the user after deciding which compressor to use.

Chunking Data

Data is usually stored contiguously in memory. When data is accessed, it has to be loaded into the cache and neighbouring data is also prefetched. If the application's use case requires the data to be accessed in the same order, this is beneficial in terms of runtime performance because the data has already been loaded. This is a sensible heuristic because it allows the correlation of spatial and temporal locality to be exploited, which is a common property of real-world datasets (cf. discussion of nature's locality in Section 4.1.1). Jumping between memory addresses can be detrimental to runtime if it leads to an increased number of cache misses (if data is moved from memory to cache) or additional physical repositioning of the disk read head (if data is moved from disk to memory). The average access time to memory is defined by Equation (3.1) with $t_m \gg t_c$ (Wulf & McKee, 1995). The aim is therefore to minimise the cache miss rate.

$$t_{\text{avg}} = p \times t_c + (1 - p) \times t_m \quad (3.1)$$

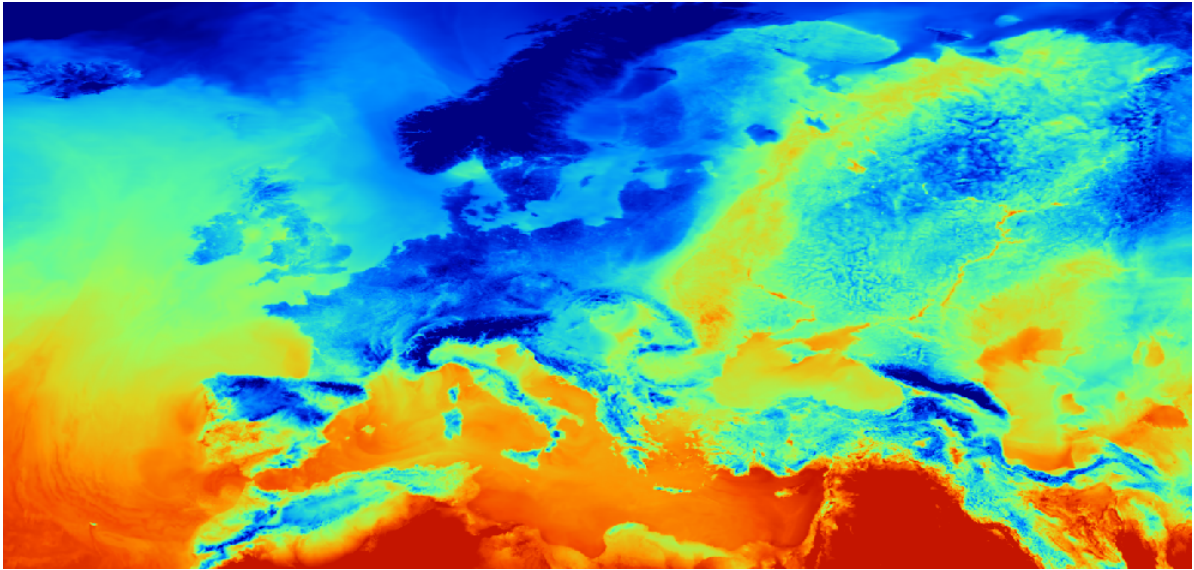


Figure 3.14.: DWD OpenData ICON 2m temperature from 2023-05-18 (spatial view).

With:

p	Probability of cache hit
t_m	Access time memory
t_c	Access time cache

Very often data is multi-dimensional (e.g. four dimensions for 3D time series) but is still mapped into one-dimensional memory. Now it depends on the use case if the data access is still contiguous or if more and more jumps have to be performed.

For demonstration purposes the time series of a 2 metre temperature output is taken from ICON. It is stored as a netCDF file and its internal structure is such that the time dimension is the first, slow moving dimension and the two spatial dimensions are the fast moving dimensions. One use case could be to look at the map at a particular point in time, which is shown in Figure 3.14. The access to the virtual data is shown in Figure 3.16a. The data access iterates over the fast, spatial dimensions and cache misses only occur when the prefetched data has been fully consumed. This access is optimal and has a low average time.

Another use case would be if you were only interested in the temperature forecast for a specific point (e.g. ‘How warm will it be in Hamburg over the next few days?’), which is shown in Figure 3.15. The data access would now move along the time axis as in Figure 3.16b. However, this time the slow-moving dimension is iterated, and for a given minimum dimension size this results in a cache miss for every data point and a high average access time. Another demonstration of good and bad chunking is given by Rew, 2013b.

Writing a netCDF file allows you to define the default Chunks size for each variable.

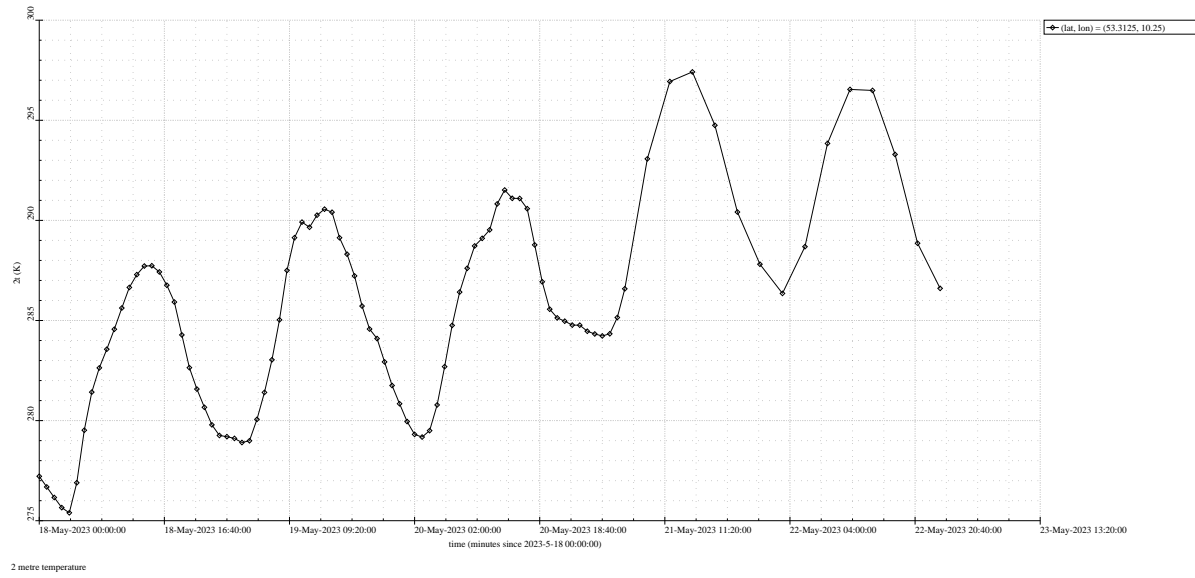
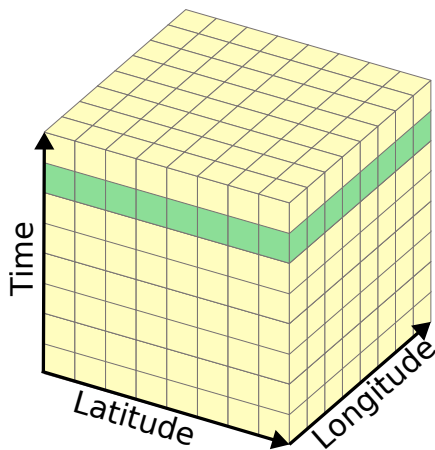
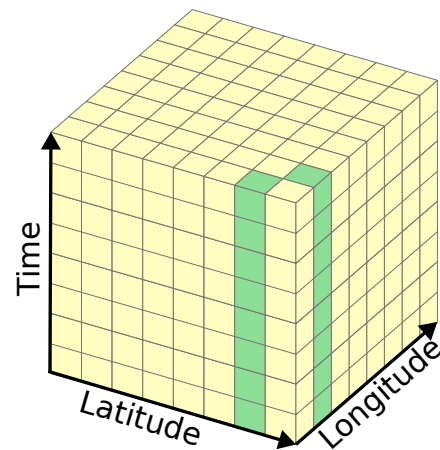


Figure 3.15.: DWD OpenData ICON 2m temperature forecast starting on 2023-05-18 in Hamburg (time view)



(a) Access pattern favouring spatial dimensions.



(b) Access pattern favouring time dimension.

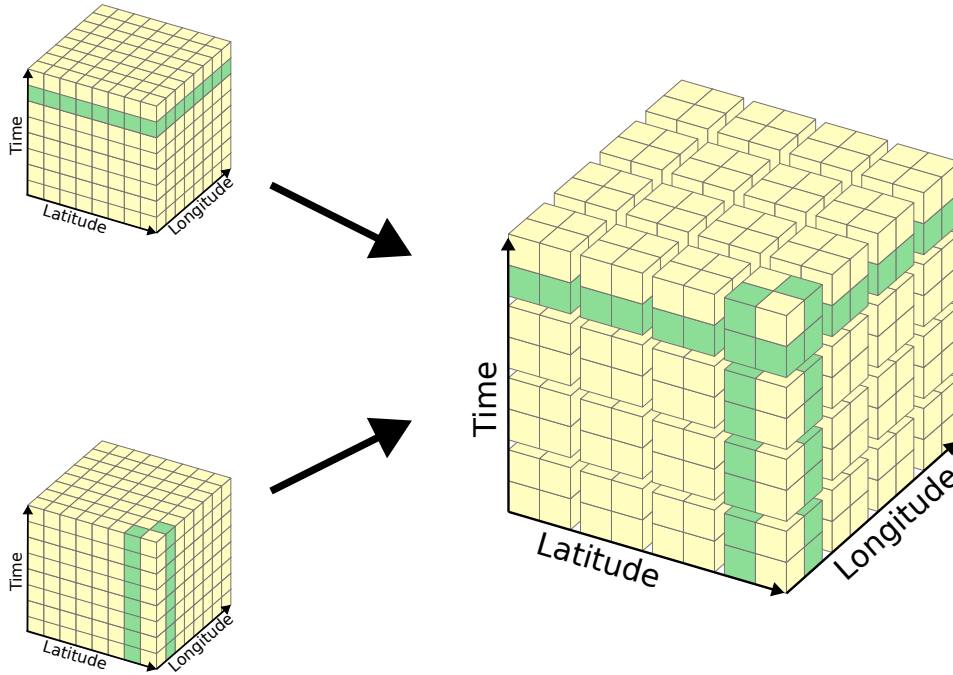


Figure 3.17.: Merging potential chunking directions.

This is particularly useful if it is foreseeable that the data will be accessed in a particular pattern most of the time. Then the Chunk shape can be adjusted to fit that access pattern. Even if there is no dominant access pattern, chunking can help to reduce the worst-case time. Even if the best case time gets worse, improving the worst case time helps to optimise the average time. For many decades it has been shown that a good choice is to use multidimensional rectangular chunks, which can be thought of as multidimensional tiles (Rew, 2013a, 2013b). How the demonstration data could look like using this Chunk pattern is shown in Figure 3.17.

The shape of the Chunks has a significant impact on performance and is also mandatory if compression is to be used. A Chunk defines the block of data to be compressed. Using only standard Chunk sizes could affect the compression result (Vader, 2016).

3.4. Summary

LLVM has been chosen as the compiler framework. Based on this decision, the design of CATO can now be done in the next Chapter 4. After looking at several layers of hardware and software that could be used for parallelisation, it was decided to focus on applications that already use OpenMP and modify them using one-sided MPI operations. This preserves the semantical behaviour of the original application, but allows the modified application to use distributed memory. Building on this, netCDF is considered to introduce parallel I/O and compression into the original application. The use of chunking or compression may have a negative impact on the runtime performance of the application, as (de-)compression is not free in terms of CPU load, but the benefit

of reduced data size may still be worth it. If the application is not compute-bound but memory-bound, using compression can provide some relief if it is done during the idle time of the CPU.

4. Tool Design

In order to demonstrate the versatility of CATO as a toolbox, three basic components have been designed to serve as the basis for further extensions. The following three Sections 4.1 to 4.3, will outline these components and their respective requirements for integration with CATO. In addition to the general description, each topic will highlight two key aspects.

1. Minimum requirements: Users should be able to enable or disable specific components within CATO. However, CATO must remain operational without additional user input. This is particularly important when a component not only performs analysis but also code transformations. It is necessary to define the requirements for each component to include all relevant code kernels belonging to the component's use case (high *true positive* rate), while ensuring that the component does not interfere with unrelated code kernels (high *true negative* rate). As the feedback component described in Section 4.3 does not involve any code transformations, the declaration of minimal required characteristics only applies to the first two components.
2. Generic replacement: By using predefined characteristics to identify relevant code kernels, we can make assumptions about the code structure, such as that certain data structures or function calls are used in a particular way in that kernel. This allows to prepare generic replacement code that can be inserted at compile time and automatically calibrated and executed at runtime.

This is why a replacement code in CATO is called a *Equivalence Class* (not to be confused with a class in the traditional sense of object-oriented programming). To enable this, CATO requires a mapping between identified code properties and specific *Equivalence Classes* (ECs). An EC needs to be designed with sufficient generality to accommodate the original code kernel's behaviour. While the replacement code may introduce additional functionality, it must preserve the original semantical behaviour. Although these considerations hold in theory, practical constraints may arise.

In the current design of CATO, a single generic replacement kernel is intended for each potential code kernel. However, it is possible to offer several alternative kernels. There are two general ways to determine the choice of kernel:

- User selection: The user can manually select the desired kernel for replacement.
- Automatic selection: CATO can automatically determine the most appropriate kernel for substitution.

Allowing the user to choose a replacement kernel may seem straightforward, but it places an additional decision burden on the user, which could potentially be complex. On the other hand, the second approach shifts this responsibility to CATO, but requires the implementation of a new decision component. Developing such a component can be a significant task, given the number of decision criteria that may exist.

As discussed in Section 2.1, the performance of an application depends not only on the application itself but also on the software and hardware environment in which it is executed. If CATO needs to choose between several equivalent replacement kernels, an abstract cost function would be required. This function would provide a performance model to estimate important metrics such as absolute runtime, peak memory/storage consumption, or the trade-off between redundancy, recomputation and communication.

Calibration of this abstract cost function would require consideration of the performance metrics of the specific environment. There are several approaches to deriving these metrics:

Benchmarks Specialised benchmarks can be used to derive realistic performance metrics that reflect the capabilities of the underlying hardware. These benchmarks serve as upper bounds, determining the optimal performance that similar code cannot exceed. Various benchmarks exist for stress testing specific components such as I/O (*MACSio* (LLNL, 2020)), MPI or the network layer (*Intel MPI Benchmarks* (Intel, 2018), *OSU Microbenchmarks* (Barrett & Hemmert, 2009), *SKaMPI* (Reussner et al., 1998), *GPCNeT* (Chunduri et al., 2019)) or CPU (*HPL* (netlib, 2018), *HPCG* (HPCG Benchmark, 2022)), or the entire system (*NAS Parallel Benchmarks* (Bailey et al., 1993; NASA, 2022), *StressBench* (Chester et al., 2021)).

Models Benchmarks can also be used to create additional model layers, such as a roofline model. This helps to increase the expressiveness of the benchmarks and to identify potential system boundaries based on the arithmetic intensity of the original kernel (Williams et al., 2009).

Profiles A profile-guided approach involves profiling or tracing the original, unmodified application to capture its actual behaviour on the specific hardware. This approach provides direct observation of the application, eliminating any doubt introduced by proxy measurements. However, the profiling process itself can introduce overhead and potentially distort timing results. Trace generation, which provides more detailed insight than sampling approaches, can have a significant impact on the application’s timing. Tools such as *ScalaBenchGen* (Wu et al., 2012) can assist with profiling.

Ultimately, this process would lead to a potential calibration that aligns with both hardware and software, allowing the selection of a specific replacement kernel. However, achieving such a calibration would require extensive research and implementation efforts. As a result, the current design of CATO only allows for a single replacement kernel per component, as this simplifies the complexity associated with research and implementation.

The parts of CATO that perform code analysis will be hardwired into the code itself, since detection and analysis must be deeply integrated into CATO. Unlike the core modifications made to the IR, the transformation components will be relatively customizable. These components can be included as C++ files, allowing users to make required adjustments as described in Section 5.1.3. In addition, developers working on CATO have the flexibility to easily modify or extend these components.

In the current design of CATO, users have access to two different tools:

1. OpenMP distribution tool (cf. Section 4.1)
2. I/O parallelisation tool (cf. Section 4.2)

The first tool allows easy testing of the benefits of distributed computation using MPI compared to single node computation using OpenMP. This approach offers potential benefits in terms of overall performance by utilising additional computing resources beyond the limitations of a single node with shared memory. In addition, data can be partitioned, allowing scalability of usable memory, which is limited by the number of independent *Concurrent Operational Units* (COUs) (CPUs, cores) on a single node. This capability allows the computation of larger input problems that would exceed the memory capacity of a single node (see Sections 4.1 and 4.2).

The primary focus of this tool is not runtime improvement, but rather extended memory utilisation that would otherwise be infeasible due to the limited hardware resources of a single node. Converting an algorithm that relies on shared memory into a form that uses distributed memory involves several aspects. Not only do communication calls need to be replaced, but in some cases the data structure or even the algorithm itself needs to be modified.

The second tool focuses on improving parallel file access for reading and writing, prioritising it over serial file I/O. In the current design, this capability is implemented specifically for netCDF, but it can be extended by incorporating additional I/O libraries that support parallel I/O interfaces. This I/O component simplifies data distribution and collection by eliminating the need for sequential data handling by the primary COU. In doing so, the tool reduces the complexity of the application's internal logic and improves I/O performance. This tool provides significant benefits for applications that involve I/O operations on large data sets throughout their runtime, including the creation of checkpoints, rather than just at the beginning and end.

Modifications made to the original high-level source code of the user's application are not directly observable. These modifications occur at the IR code level, which is then compiled into the final binary. While users have the ability to examine the modified IR code, it can be difficult to interpret, especially for those without specialised training, as it is very similar to assembly code. To address this issue, CATO includes an additional component that provides feedback to the user. The function of this component is to provide a summary of the code changes and give hints about the general nature of the changes that have been made. In this way, the user can gain an approximate understanding of the changes made and grasp their complexity. Details of the feedback component are presented in Section 4.3.

4.1. Component: Memory Handling

The target audience of users and applications has already been addressed in Section 1.3.2. As a result, the focus is primarily on scientific applications that predominantly use basic shared memory parallelisation, such as OpenMP, and I/O functionalities, especially those of netCDF. In line with the motivation described in Section 7.1, efficient data handling is a critical aspect. The I/O component also makes use of the memory handling capabilities.

4.1.1. Communication Patterns

In Section 1.1.2 the targeted applications have already been outlined. Based on the scientific background, there is a distinct variety of communication patterns that are used. In particular, heap variables marked `shared` within the OpenMP kernel are assumed to be crucial for optimal handling by CATO. Therefore, CATO focuses specifically on understanding and analysing the communication patterns of these shared heap variables.

The communication pattern plays an important role in determining the optimal replacement approach. In particular, an intensive *read-only* (RO) pattern could benefit from a higher degree of redundancy, resulting in reduced communication requirements. On the other hand, a write-heavy pattern requires more communication to ensure data integrity and consistency. Therefore, the communication pattern has a significant impact on the choice of the optimal replacement strategy.

Given the target audience of CATO as discussed in Section 1.3.2, the communication patterns under consideration are derived from numerical methods used in science and engineering. These methods can be grouped into clusters based on similarities in computational and data access behaviour, while still maintaining some level of abstraction from individual implementations. One such cluster is called Dwarf. While the set of Dwarves may not be exhaustive in capturing all important behaviours, they are considered representative of the most important patterns in science and engineering (Che et al., 2009). CATO’s design was created specifically with these patterns in mind, allowing it to address a wide range of algorithmic problems within its target audience by focusing on these Dwarves.

Initially there were seven Dwarves based on general numerical methods, as mentioned in (Colella, 2004). Subsequently, six more Dwarves were introduced, derived from benchmarks on embedded, desktop and server computing (four Dwarves) and machine learning (two Dwarves) (Asanovic et al., 2006). A brief overview of these Dwarves is given in Table 4.1.

An important communication pattern found in various application areas is the stencil, which is associated with the *Structured Grid* Dwarf. In the real world, many phenomena can be described in terms of particles that have both wave and particle properties and interact with each other. A perfect simulation of reality would require taking into account the interaction between each particle and every other particle in the observed system. However, such an approach introduces a complexity overhead of $O(n^2)$ for each particle interaction, which becomes infeasible at a certain point. Fortunately, in most cases, a

Dwarf	Description
Dense Linear Algebra	Densely packed vector/matrix with local accesses
Sparse Linear Algebra	Sparsely packed (i.e. compressed) vector/matrix with local accesses
Spectral Methods	Decomposition into partial frequencies usually using butterfly stages
N-Body Methods	Interaction between a set of discrete points, without optimisation every point needs to communicate with every other point
Structured Grids	Regular grid with high spatial locality (e.g. using a stencil), working in-place or in a buffer
Unstructured Grids	Like <i>Structured Grids</i> but the access order is only predestined by the application
Monte Carlo	Statistical methods working on random trials, usually embarrassingly parallel
Combinational Logic	Simple (logical) operations on large data sets usually exploiting bit-level parallelism
Graph traversal	Data is organised in a graph and needs to be quickly iterated, usually lookup is more decisive than computation.
Construct Graphical Models	Problem domain is converted into a graph, where some variables are connected over attributes
Finite State Machine	Machine transitions between distinct states based on external input
Dynamic Programming	Solve problem by solving overlapping subproblems, re-ordering and caching their computation avoid redundant computation
Backtracking	Divide and conquer approach, which skips subproblems, which cannot fulfil the original question (usually an optimisation)

Table 4.1.: Overview of 7+6 Dwarves; the first seven entries are the original Dwarves, the latter six entries are the extension.

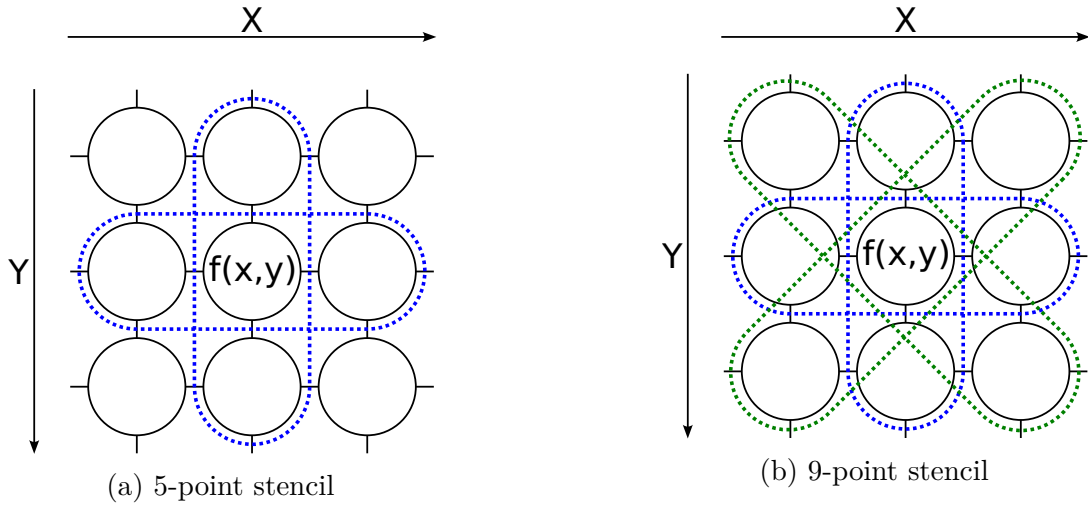


Figure 4.1.: 2D examples for stencils, which traverse through the structured grid to update grid cells. The corresponding Equations (4.1) and (4.2) are simplified and do not consider boundary conditions.

cutoff radius can be used to limit the interaction between particles to a certain distance. This allows a significant reduction in complexity. The cutoff radius approach is effective because the strength of most physical forces tends to decrease with distance. Although the strength of a physical force between two particles never reaches zero, it decreases exponentially with distance and can be considered negligible beyond a certain point. Of the four fundamental forces, only the *Strong Interaction* behaves differently, as it increases proportionally with distance. However, the *Strong Interaction* and the *Weak Interaction* operate primarily at the subatomic level, while other forces arise from the *Gravitational Interaction* and/or *Electromagnetic Interaction* (Braibant et al., 2012).

Nature is local, therefore caching works.

A stencil can be interpreted as a cut-off radius, where the parameter n in an n -point stencil represents the number of (neighbouring) grid cells involved in updating the value of a single cell. While in theory the distribution pattern of the involved cells could be arbitrarily distributed across the grid, nature tends to favour locality. As a result, the cells involved are typically concentrated in the immediate vicinity of the target cell. This fixed pattern is used to update the value of a grid cell by taking into account the values of the other involved cells. The importance of stencils is reflected in their inclusion in supercomputer benchmarks such as *NAS Parallel Benchmarks* ([web:nas_benchmarks](#); Bailey et al., 1993) and *Rodinia* ([web:Rodinia_benchmarks](#); Che et al., 2009). They are also widely used in climate research applications.

Figure 4.1 shows two examples of different stencil patterns: A 2D 5-point stencil (cf. Equation (4.1)) and a 2D 9-point stencil (cf. Equation (4.2)). Whether the update is performed in-place or in a buffer depends on the underlying algorithm.

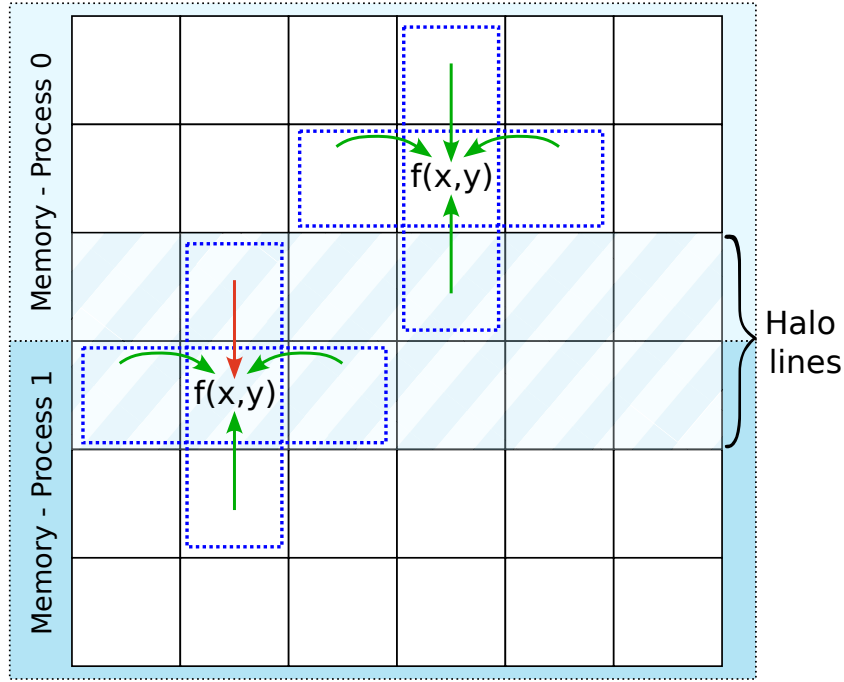


Figure 4.2.: Schematic diagram of the stencil pattern executed concurrently by multiple processes. Green arrows represent reading from local memory, while red arrows indicate the need to obtain values from neighbouring processes using MPI. The halo lines denote the cells that will be targeted by a neighbouring process for reading. The buffer memory required for MPI communication is omitted in this diagram.

$$\begin{aligned}
 f(x, y)_{5-point} = & C + a_0 \cdot f(x, y) \\
 & + a_1 \cdot f(x-1, y) + a_2 \cdot f(x+1, y) \\
 & + a_3 \cdot f(x, y-1) + a_4 \cdot f(x, y+1)
 \end{aligned} \tag{4.1}$$

$$\begin{aligned}
 f(x, y)_{9-point} = & C + a_0 \cdot f(x, y) \\
 & + a_1 \cdot f(x-1, y) + a_2 \cdot f(x+1, y) \\
 & + a_3 \cdot f(x, y-1) + a_4 \cdot f(x, y+1) \\
 & + a_5 \cdot f(x-1, y-1) + a_6 \cdot f(x-1, y+1) \\
 & + a_7 \cdot f(x+1, y-1) + a_8 \cdot f(x+1, y+1)
 \end{aligned} \tag{4.2}$$

Building on this discussion, CATO is developed primarily with a focus on stencils. This includes an emphasis on heap memory allocation. The heap memory is divided into even shares, which reduces the amount of communication required between processes. During the stencil iteration through the memory block, inter-process communication is only required at boundaries where reading $f(x, y \pm 1)$ is required but belongs to the neighbouring process (cf. Figure 4.2).

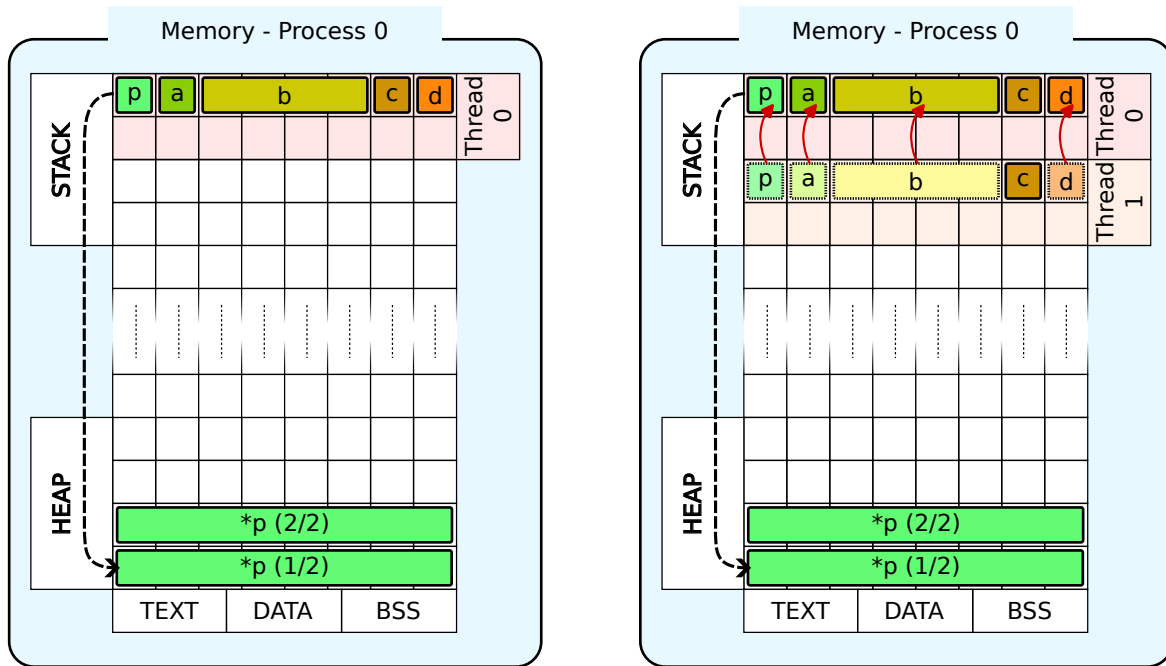
```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <omp.h>
4
5  int main(int argc, char const *argv[])
6  {
7      double *p = malloc(sizeof(double) * 64);
8      double a = 0.0;
9      double b[2] = {1.0,1.0};
10     double d = 10.0;
11
12     #pragma omp parallel for shared(a,b,d)
13     for (size_t i = 0; i < 2; i++)
14     {
15         double c = 2.0 ;
16         #pragma omp critical
17         a = 3.0;
18         b[i] = i*d;
19     }
20
21     free(p);
22     return 0;
23 }
```

Listing 4.1.: Simple OpenMP code to demonstrate the process’ memory management (cf. Figure 4.3a)

4.1.2. Distribute Memory of Stencil Pattern

Listing 4.1 provides a code example that demonstrates how CATO handles different types of variables. When the source code is compiled and executed, the OS spawns a new process with a nested master thread. The properties of a process, such as the binary code, initialised static/global variables, accounting information, child processes, and the address space for heap and stack data, are shared by all nested threads. Each thread is associated with a process and has its own set of properties, including program counters, registers and stack memory within the process address space (Tanenbaum, 2009). Figure 4.3a shows the memory layout at the end of the execution of the source code with a single thread. Not all properties are visualised in this figure, the focus is on the address space, specifically the stack and the heap.

The variables in the code example differ in four aspects: data sharing between threads, memory location (stack or heap), access type (read or write) and whether they are scalars or arrays. An overview of the possible combinations is given in Table 4.2. In line 7 of



(a) Executing a binary means to create a new process and assign memory to it. Each process has a single master thread, where the actual execution is happening.

(b) Spawning a new thread within an process assigns a private stack to the new thread.

Figure 4.3.: Visualisation of a process' memory composition, highlighting the stack and heap as the most significant components. The presentation is idealised and does not consider technical details such as data type sizes or alignment. It merely represents the general memory layout depicted in Listing 4.1. The loop variable `i`, which requires special handling by CATO, and other implicit data stored on the stack, such as function calls, are omitted.

	Data-Sharing	Memory	Access	Dimension
<code>p</code>	shared	Stack	read	scalar
<code>*p</code>	shared	Heap	write	dim.
<code>a</code>	shared	Stack	write	scalar
<code>b</code>	shared	Stack	write	dim.
<code>c</code>	private	Stack	write	scalar
<code>d</code>	shared	Stack	read	scalar
<code>i</code>	private	Stack	read/write	scalar

Table 4.2.: Overview of variable property combinations from Listing 4.1.

Listing 4.1 dynamic memory is allocated and assigned to the pointer variable `*p`. The allocated memory is stored on the heap, while the pointer variable that references it is stored on the stack. Since the allocated memory is on the heap, which is shared by all threads, declaring it `private` would not add any value if the allocation happened outside the OpenMP kernel. This is because even though the pointer variables are private, they all point to the same memory value, which can be accessed by all threads. If the allocation happens inside the OpenMP kernel, then each thread gets a unique region of the shared heap memory. This work does not address the scenario of allocating memory within the OpenMP kernel itself.

The other variables are `a` (cf. line 8 (Listing 4.1)), `b` (cf. line 9 (Listing 4.1)), `c` (cf. line 15 (Listing 4.1)) and `d` (cf. line 10 (Listing 4.1)) are allocated statically during compile time and are therefore also stored on the thread's stack. `c` is created on each thread's stack, the other variables are created on the master thread's stack and are referenced by the other threads. The loop variable `i`, which is used by OpenMP to distribute work among newly spawned threads, requires special handling by CATO during the load balancing phase.

If the code is built with OpenMP support, additional threads will be spawned during execution. The execution flow remains the same until line 12, where a significant change occurs. As soon as a thread encounters the `parallel` directive, it creates a new team of threads and becomes their master. Until the end of the OpenMP kernel is reached, the memory layout is modified as shown in Figure 4.3b (a team size of two is assumed for demonstration purposes). The private variable `c` declared within the OpenMP kernel is created on the stack as usual. However, the other implicitly shared variable `p`, as well as the explicitly shared variables `a`, `b` and `d`, which already exist on the master thread's stack, are not copied but only referenced (cf. red arrows) by the new team threads (OpenMP Architecture Review Board, 2021, Ch. 1.4.1). Therefore, their individual memory addresses are the same in each thread.

This is where CATO comes in. Instead of using threads, new MPI processes are spawned to replicate the behaviour of the original threads. These processes are also able to make use of the heap by dividing it up and sharing it equally between all the processes involved. If the processes are running on different compute nodes, they can use shared memory over the network, which was not possible in the original OpenMP version. In this setup, each process has its own copy of the original stack and only its allocated portion of the original heap memory, as shown in Figure 4.4.

This requires additional operations to ensure data integrity and consistency, as required by the use of MPI, as well as the introduction of new data structures on the stack (represented by the generic red blocks in Figure 4.4). The sizes of instances of these new data structures are not necessarily the same as the original data. In particular, when dealing with scalar variables of primitive data types, their handling may require noticeably more memory. This overhead is only necessary for shared variables written to within the OpenMP kernel (i.e., variables `a` and `b`). The `d` variable is read only and remains unchanged, so it can simply be copied.

CATO distinguishes between three memory schemes for handling variables, primarily based on where they are stored. The main distinction is between variables stored on the

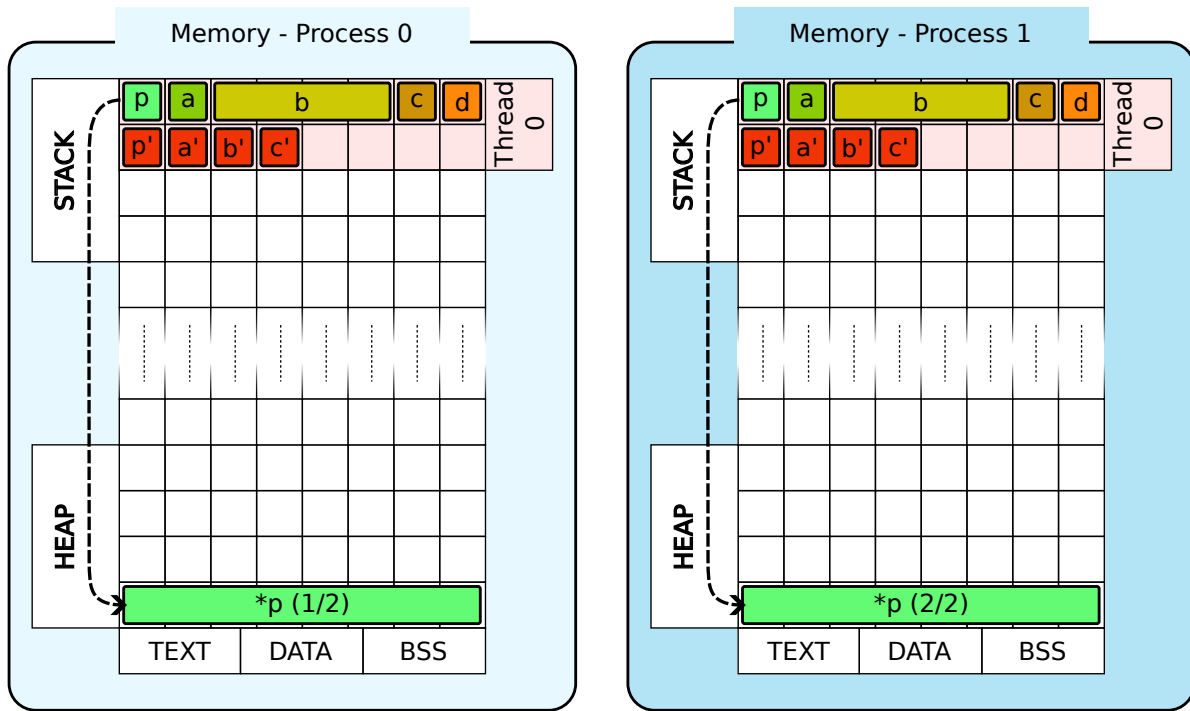


Figure 4.4.: Visualisation of CATO’s memory adjustments. To ensure proper tracking of changes and accommodate new data structures, additional auxiliary stack variables need to be created. However, in return, significant variables residing on the heap can now be partitioned and distributed across multiple compute nodes connected through an interconnect.

stack and those stored on the heap. Stack variables are further categorised based on OpenMP’s data sharing attributes. However, it does not make sense to further subdivide heap variables into **shared** or **private** categories, since the pointer variable itself only references the allocated heap memory, which is inherently shared by all threads within the same process.

Private stack variable: Each process has its own private copy of this variable. All read and write operations are thread-local. Private variables, unlike shared variables, are not intended for communication between threads. This includes stack variables that are explicitly declared as **private** within an OpenMP clause, or implicitly marked as **private** because they are declared within the OpenMP kernel (OpenMP Architecture Review Board, 2021, Ch. 5.1.1). For simplicity, the term *private variable* includes all derivatives of the **private** clause, such as **firstprivate** and **lastprivate**.

Shared stack variable: For a stack variable that is shared between threads (or processes after the changes made by CATO), CATO distinguishes between read-only variables and variables that are updated within the OpenMP kernel. In the case of a shared RO variable, it is simply copied to every process.

When a variable is written, additional effort is required to protect data integrity against data races. Updates must also be synchronised between all participating COUs to maintain data consistency. In the case of OpenMP, this was relatively straightforward, as shared stack variables were accessed via their references, and no additional data transfer was required apart from the necessary locking mechanisms. However, when using MPI, additional communication operations are required to replicate this behaviour. To minimise unnecessary communication and keep the communication scheme simple, these variables are always updated at the master level. Consequently, each process must retrieve the current values from the master rank each time it accesses them and communicate its updates back to the master rank.

Heap variable: Of particular interest are heap variables, which store a sequence of primitive data types. These sequences can be shared equally by all participating processes. If equal distribution is not possible, the overlap is shared by the processes in front using the *modulo* operator. In the original OpenMP version, all threads could access the shared heap directly. However, in the modified version, each process must keep track of the specific segment of the original continuous memory that it needs to access. This ensures that the correct message handler is used for the corresponding MPI read or write operation.

The methods described for handling shared and private variables stored on the stack have the drawback of increased memory consumption, as these variables are duplicated, and only the access pattern changes with respect to the **shared** or **private** use case. OpenMP use case. However, this overhead is likely to be negligible as the stack size is much smaller compared to the potential size of heap memory. Typical maximum stack sizes are usually in the low megabyte range, such as 8 MiB on an Intel Core i7-6700 CPU running Fedora 37¹. Although this limit can be increased, it is generally more appropriate to store large amounts of data on the heap rather than the stack. In addition, as discussed earlier in Section 1.3.2, CATO focuses on applications with a small number of main variables that are significant in terms of relevance and memory consumption. While it is possible for a local scalar stack variable to have a significant impact on application performance, such as a tracking variable that is accessed disproportionately often, this particular use case is not considered by the heuristic.

4.1.3. Optimal Memory Consumption

The first two replacement schemes for handling stack variables are primarily aimed at ensuring correct replacement rather than providing performance improvements at this stage. The significant performance improvement is achieved by the third replacement scheme, which handles heap memory. With this scheme, heap memory can be shared among all participating processes, allowing the combined main memory of all participating compute nodes (in the extreme case, one physical compute node per process) to be

¹Result of `ulimit -s`

utilised. This fulfils the initial goals of using distributed memory (cf. Section 1.2.1) and increasing the potential size of the input problem (cf. Section 1.1.2).

Peak memory consumption is crucial for limiting the size of the problem, as the capacity of main memory sets an irrevocable physical limit that cannot be exceeded by the application’s requirements. Usually a scientific application consists of several phases of memory consumption.

1. Initialisation
2. Computation
3. Result collection

The initialisation step can be optional if the application does not require any input. Usually, the user can control the application by setting input parameters (rather small memory footprint) or providing input files (can range from small to large memory footprint). After the application has used the input to initialise its internal data structures, the actual computations are performed. Finally, some form of output is usually provided, ranging from small terminal output to huge output files. These phases are not necessarily strictly separated, but can be interchanged (some applications split up the computation phase and insert (multiple) I/O phases, for example to create checkpoints).

Since CATO distributes heap variables, this can be used to increase the maximum input problem provided during the initialisation phase. There are two potential optimisations that CATO handles. Splitting and distributing the heap memory allocated during the computation phase reduces the peak memory consumption; the larger the ratio of temporary heap variables during the computation phase to the total memory consumption, the more significant the savings. The other optimisation takes effect when each process collects only a part of the input problem. This can drastically reduce the peak memory consumption during the initialisation phase. The latter will be discussed in more detail later in Section 4.2.

The original application loads the input problem into the heap (cf. Figure 4.5a). If no additional heap memory is needed during the runtime of the application, the whole heap can be used for the initialisation data. Otherwise, the size of the input problem must be limited to leave enough free memory for the computation phase (cf. Figure 4.5b). An example for this is `partdiff`, an application used for teaching at the *Scientific Computing* research group at the University of Hamburg. It is a solver for *Partial Differential Equations* (PDEs) using the *Gauss-Seidel* and *Jacobi* methods. The choice of method has a significant effect on the temporary memory consumption. If *Gauss-Seidel* is used, the result of the calculation is stored directly on the input matrix. If *Jacobi* is used, another temporary matrix must be allocated and used to hold the values of the previous iteration. This means that the memory requirement on the heap is doubled in this mode, or the size of the initial matrix must be halved. In general, the ratio between the initial input memory and the temporary heap memory is not fixed, but depends on the computation. To fully utilise a node’s memory, the size of the input and the resolution of the computation must be optimally matched.

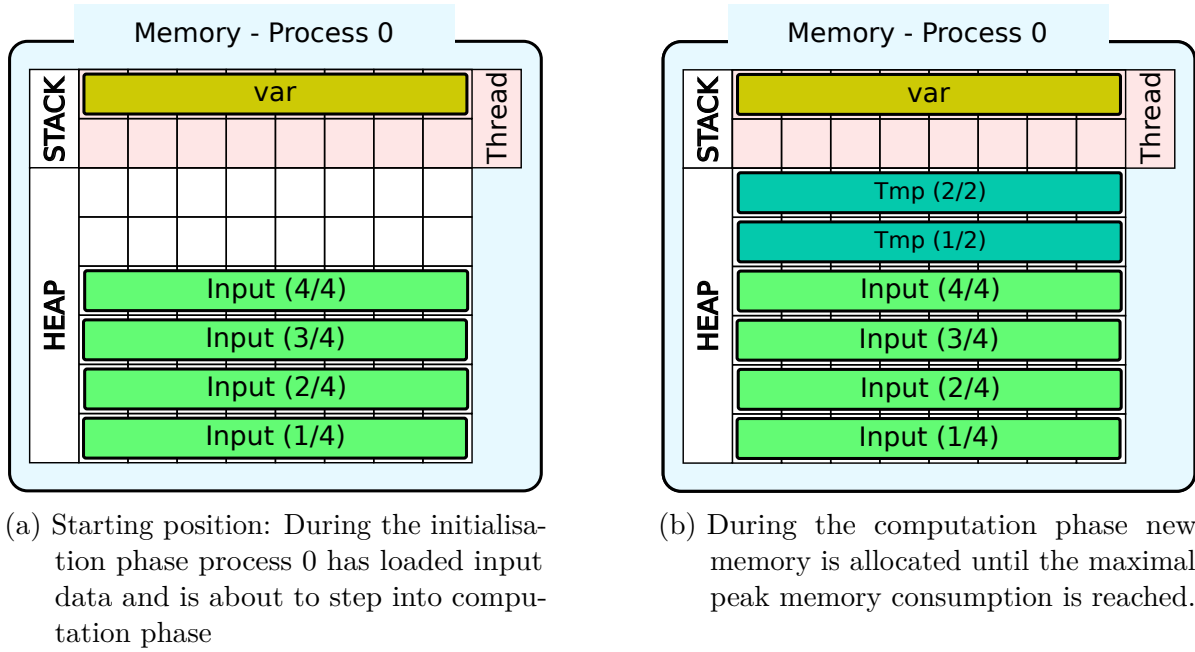


Figure 4.5.: The unmodified application's memory usage.

CATO would distribute the input data after the initialisation phase, so that each process only stores a share of the initial data. If the computation phase is inflexible with respect to the size of the temporary memory, the same amount of additional heap memory is required, but this time the node's memory would not be used to its full capacity (cf. Figure 4.6). On the other hand, if the application can be tuned for increased memory consumption during the computation phase (e.g.. due to finer resolution or additional layers used by the computation algorithm as described in Section 1.1.2), the modified application could now use twice as much temporary memory (cf. Figure 4.7).

Apart from the memory consumption, this distribution also has an impact on the runtime performance of the modified application. By default, CATO optimises for maximum potential memory consumption, so that it avoids unnecessary data redundancy, but distributes the input data evenly. Because of the minimised redundancy, this is likely to result in higher communication demands if a process needs to access data that has been moved to another process. On the other hand, the application now benefits from the additional computing power that has become available by using many more cores from multiple nodes instead of cores from a single node. There are many factors that come into play to influence the final runtime performance, ranging from the network topology and bandwidth, to the compute pattern that defines how often communication needs to occur, to the node hardware performance metrics (i.e. memory and cores).

4.1.4. Memory Model

As the focus of this component is on improved memory usage, the baseline of what is possible needs to be analysed. To do this, a generalised model of memory usage is

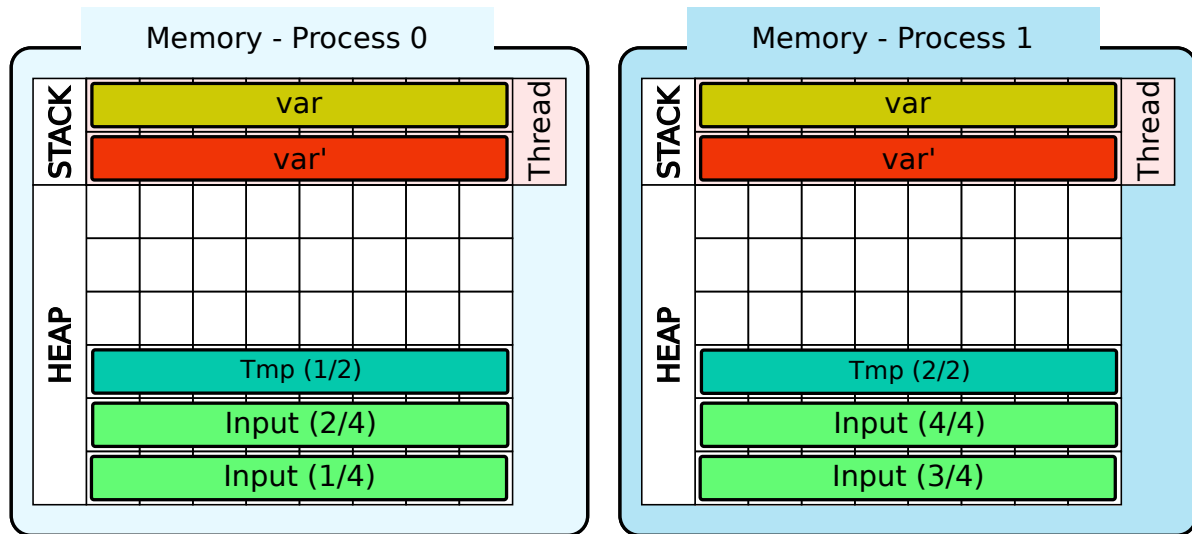


Figure 4.6.: The initial data is automatically distributed by CATO, but the computation phase' demand for heap memory cannot be increased.

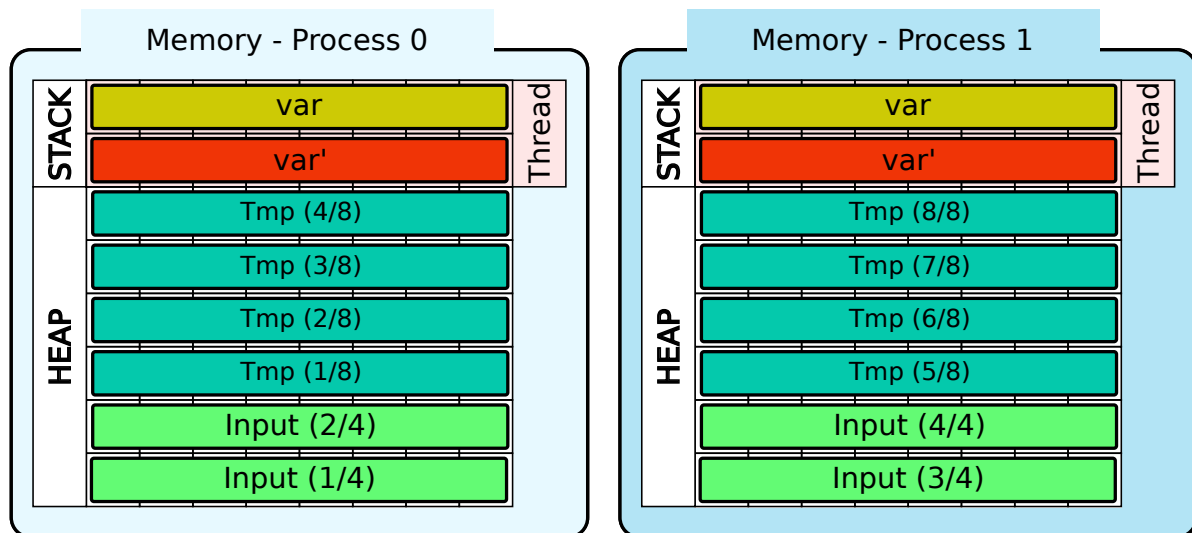


Figure 4.7.: The initial data is automatically distributed by CATO and this time the computation phase benefits from the additional heap memory.

constructed, focusing on the main participants. This omits anything not directly involved in the CATO’s process, such as the call stack; although CATO is likely to affect this as new function calls are inserted, its impact is likely to be negligible.

First, we need to determine the status quo, which is defined by the use of the stack and heap on a single compute node. Superscript terms identify the general environment, and subscript terms and control variables are used to specify the term. Constant terms refer to a specific variable within the source code. Functional terms refer to the same variables, but this time also take into account how their memory demand changes when declared and stored within multiple COUs, which may be spread across multiple nodes. The specific meaning of a *Concurrent Operational Unit* (COU) is either a thread or a process, depending on the context.

Starting with Equation (4.3), the total stack memory used is the sum of all stack variables. Since CATO must behave differently on different types of stack variables, they are further subdivided into **private** and **shared** variables; the latter are further subdivided into variables that are *read-only* (RO) or *read-write* (RW). The combined size of all these variables represents the total stack memory occupied when a single thread is used. If more threads are used (n_T many), additional overhead is introduced, so the requirement for a single thread cannot simply be multiplied by the number of threads. Private stack variables are simply copied to each thread’s stack, so their consumption scales by a factor of 1 with the number of threads on a single node (cf. Equation (4.4)). Shared variables on the other side remain in the stack memory of the master thread that forked the additional threads. Each forked thread stores a reference to the master thread’s stack variable (cf. Figure 4.3b), which still claims some memory (cf. Equations (4.5) and (4.6)²) and are represented by the constants $C_{ro,i,t}$ and $C_{rw,i,t}$. Equation (4.7) then sums up all these terms and represents the actual stack memory consumption on a single node when n_T many threads are used.

For heap memory, no such differentiation is necessary, so Equation (4.8) can be kept simple in comparison and simply sums all heap variables.

$$S^{omp}(1) = \underbrace{\sum_i S_{pr,i}}_{S_{pr}} + \underbrace{\sum_i S_{sh(ro),i}}_{S_{sh(ro)}} + \underbrace{\sum_i S_{sh(rw),i}}_{S_{sh(rw)}} + C^{omp}(1) \quad (4.3)$$

$$S_{pr}^{omp}(n_T) = n_T \cdot S_{pr} \quad (4.4)$$

$$S_{sh(ro)}^{omp}(n_T) = \sum_i \left(S_{sh(ro),i} + \sum_{t=1}^{n_T-1} C_{ro,i,t}^{omp} \right) \quad (4.5)$$

$$S_{sh(rw)}^{omp}(n_T) = \sum_i \left(S_{sh(rw),i} + \sum_{t=1}^{n_T-1} C_{rw,i,t}^{omp} \right) \quad (4.6)$$

²Sum starts at $t = 1$ as the master thread does not need a reference to its own variables and is therefore excluded

$$S^{omp}(n_T) = S_{pr}^{omp}(n_T) + S_{sh(ro)}^{omp}(n_T) + S_{sh(rw)}^{omp}(n_T) + C^{omp}(n_T) \quad (4.7)$$

$$H^{omp} = \sum_i H_i^{omp} \quad (4.8)$$

With:

$S^{omp}(n_T)$	Total used stack memory on a single node using n_T threads
$S_{pr,i}$	Individual private stack variable
S_{pr}	All private stack variables
$S_{pr}^{omp}(n_T)$	All private stack variables on a single node using n_T threads
$S_{sh(ro),i}$	Individual shared stack variable, which is RO within the OpenMP kernel
$S_{sh(ro)}$	All shared stack variables, which are RO within the OpenMP kernel
$S_{sh(ro)}^{omp}(n_T)$	All shared RO stack variables on a single node using n_T threads
$S_{sh(rw),i}$	Individual shared stack variable, which is RW within the OpenMP kernel
$S_{sh(rw)}$	All shared stack variables, which are RW within the OpenMP kernel
$S_{sh(rw)}^{omp}(n_T)$	All shared RW stack variables on a single node using n_T threads
$C_{ro,i,t}^{omp}$	Reference overhead of RO stack variable i on thread t
$C_{rw,i,t}^{omp}$	Reference overhead of RW stack variable i on thread t
$C^{omp}(n_T)$	Remaining stack variables like function calls on a single node using n_T threads
H_i^{omp}	Individual heap variable
H^{omp}	Total used heap memory on a single node (this value does not depend on the number of used threads)

In total, this gives the total memory consumption of a given configuration (cf. Equation (4.9)), which can then be compared with the total memory available and which could be used by the application (cf. Equation (4.10)).

$$M^{omp}(n_T) = S^{omp}(n_T) + H^{omp} \quad (4.9)$$

$$M_{max}(1) \stackrel{!}{\geq} M^{omp}(n_T) \quad (4.10)$$

With:

$M^{omp}(n_T)$	Total memory consumption of original application on a single node using n_T threads
$M_{max}(x)$	Aggregated memory on x nodes

$^{omp}(n_T)$ and H^{omp} take into account not only stack and heap variables accessed within the OpenMP kernel, but every variable. By default, each variable is **private** if declared inside the kernel, or **shared** otherwise, unless explicitly mentioned in a data sharing clause.

As described earlier, CATO now optimises memory usage by prioritising heap variables that offer the greatest potential for improvement (cf. Section 1.3.2). However, some stack variables must also be transformed to maintain the correctness of the application. Figure 4.4 shows a selection of variable classes; CATO’s handling of these must be tailored to your particular needs.

Stack variables, which are private, need no special treatment from CATO. Each process simply declares and initialises its own instance and performs any read or write operation on it. Since they are private, there is no need to update the instances of other processes (cf. Equation (4.11)). The same goes for stack variables that are declared as **shared** but are RO within the OpenMP kernel. A basic premise is that the original application only uses OpenMP for parallelisation. It follows that every other piece of code is executed sequentially by a single thread. And since this variable is only accessed in a RO manner in the OpenMP kernel, it does not need to be updated in parallel, and therefore no mechanism is needed to maintain its integrity and consistency. As with private variables, it is therefore sufficient for each process to declare and initialise its own instance (cf. Equation (4.12)). The last case, RW shared stack variables, requires special treatment. Again, each process has its own copy of the variable $S_{sh(rw),i}$. However, these are just buffers to store the results of the communication in order to get an up-to-date version. The master rank acts as the primary node where the current version of $S_{sh(rw),i}$ is stored. If a process needs to access the variable, it must perform the following steps to get the current value from the master rank:

1. Lock the variable at the master level.
2. Get the current value of the variable.
3. In the case of writing, the process returns an updated value to the master.
4. Release the lock.

How exactly this mechanism works is explained in Section 5.2. Since each process has a copy of this variable with additional management overhead ($C_{rw,i}^{cato}$), this memory requirement is needed for each process (cf. Equation (4.13)).

$$S_{pr}^{cato}(n_P) = n_P \cdot \sum_i S_{pr,i} \quad (4.11)$$

$$S_{sh(ro)}^{cato}(n_P) = n_P \cdot \sum_i S_{sh(ro),i} \quad (4.12)$$

$$S_{sh(rw)}^{cato}(n_P) = n_P \cdot \sum_i (S_{sh(rw),i} + C_{rw,i}^{cato}) \quad (4.13)$$

With:

$S_{pr}^{cato}(n_P)$	All private stack variables on n_P processes
$S_{sh(ro)}^{cato}(n_P)$	All RO shared stack variables on n_P processes
$S_{sh(rw)}^{cato}(n_P)$	All RW shared stack variables on n_P processes
$C_{rw,i}^{cato}$	Constant overhead of CATO for managing concurrent access on shared RW stack variable

The total stack memory demand is shown in Equation (4.14). Looking at the heap variables³, now the big advantage becomes visible: As shown in Figure 4.4, each process does not store the whole heap variable, but only a part of it. Therefore the total memory consumption is the sum of all heap variables without any significant overhead (cf. Equation (4.15)).

$$S^{cato}(n_P) = S_{pr}^{cato}(n_P) + S_{sh(ro)}^{cato}(n_P) + S_{sh(rw)}^{cato}(n_P) + C^{cato}(n_P) \quad (4.14)$$

$$H^{cato} = \sum_i H_i^{cato} \quad (4.15)$$

With:

$S^{cato}(n_P)$	Total used stack memory on n_P processes
H^{cato}	Total used heap memory (by design of CATO this value does not depend on the number of used processes)
H_i^{cato}	Individual heap variable

Unlike the original application, the binary built with CATO is then able to use processes instead of threads, and can therefore run on more than one compute node. This increases the amount of memory potentially available: Instead of the memory of a single

³In this discussion, scalar heap variables (e.g. a mutex variable) will be omitted, as they are rarely useful and usually have no significant effect on memory usage

compute node, it is now the aggregated sum of multiple compute nodes. The memory consumption of the modified application is the sum, which is shown in Equation (4.16). If each process is spawned on a single compute node, the maximum usable aggregated memory can be achieved, which is shown in Equation (4.17) and which can be used by the modified application (cf. Equation (4.18)).

$$M^{cato}(n_P) = S^{cato}(n_P) + H^{cato} \quad (4.16)$$

$$M_{max}(n_P) = n_P \cdot M_{max}(1) \quad (4.17)$$

$$M_{max}(n_P) \stackrel{!}{\geq} M^{cato}(n_P) \quad (4.18)$$

With:

$M^{cato}(n_P)$	Total memory consumption of modified application using n_P processes
$M_{max}(x)$	Aggregated memory on x nodes

Having established the basics of the memory demands of the original and modified applications, we can now compare the two versions. The initial assumption was that the user can directly control the initial heap consumption by tinkering with the input parameters or choosing an appropriate input file. To achieve the initial goal of being able to compute larger input problems, it would be beneficial if two goals could be met:

1. The modified application uses less stack memory than the original application.
2. The modified application can deliberately use more heap memory than the original application.

Comparison: Stack

At first, the stack memory Equation (4.7) for threads of the original application will be reduced. The reduction of both terms for RO and RW shared stack variables proceeds in the same way. All used constants are non-negative, therefore the inequations in Equations (4.20) and (4.25) can be used.

$$S_{sh(ro)}^{omp}(n_T) = \sum_i \left(S_{sh(ro),i} + \sum_{t=1}^{n_T-1} C_{ro,i,t}^{omp} \right) \quad (4.19)$$

$$\leq \sum_i \left(S_{sh(ro),i} + \sum_{t=0}^{n_T-1} C_{ro,i,t}^{omp} \right) \quad (4.20)$$

$$= \sum_i S_{sh(ro),i} + \sum_i \sum_{t=0}^{n_T-1} C_{ro,i,t}^{omp} \quad (4.21)$$

$$= S_{sh(ro)} + n_T \cdot \sum_i C_{ro,i}^{omp} \quad (4.22)$$

$$= S_{sh(ro)} + n_T \cdot C_{ro}^{omp} \quad (4.23)$$

$$S_{sh(rw)}^{omp}(n_T) = \sum_i \left(S_{sh(rw),i} + \sum_{t=1}^{n_T-1} C_{rw,i,t}^{omp} \right) \quad (4.24)$$

$$\leq \sum_i \left(S_{sh(rw),i} + \sum_{t=0}^{n_T-1} C_{rw,i,t}^{omp} \right) \quad (4.25)$$

$$= \sum_i S_{sh(rw),i} + \sum_i \sum_{t=0}^{n_T-1} C_{rw,i,t}^{omp} \quad (4.26)$$

$$= S_{sh(rw)} + n_T \cdot \sum_i C_{rw,i}^{omp} \quad (4.27)$$

$$= S_{sh(rw)} + n_T \cdot C_{rw}^{omp} \quad (4.28)$$

Both results (cf. Equations (4.23) and (4.28)) can be substituted into Equation (4.7).

$$S^{omp}(n_T) \leq n_T \cdot S_{pr} + S_{sh(ro)} + n_T \cdot C_{ro}^{omp} + S_{sh(rw)} + n_T \cdot C_{rw}^{omp} + C^{omp}(n_T) \quad (4.29)$$

$$= n_T \cdot S_{pr} + S_{sh(ro)} + S_{sh(rw)} + n_T \cdot C_{sh}^{omp} + n_T \cdot C^{omp}(1) \quad (4.30)$$

$$= n_T \cdot S_{pr} + S_{sh} + n_T \cdot C_{sh}^{omp} + n_T \cdot C^{omp}(1) \quad (4.31)$$

Now the same can be done for the modified application using the result of Equation (4.14).

$$S^{cato}(n_P) = S_{pr}^{cato}(n_P) + S_{sh(ro)}^{cato}(n_P) + S_{sh(rw)}^{cato}(n_P) + C^{cato}(n_P) \quad (4.32)$$

$$= n_P \cdot \left(\sum_i S_{pr,i} + \sum_i S_{sh(ro),i} + \sum_i (S_{sh(rw),i} + C_{rw,i}^{cato}) \right) + C^{cato}(n_P) \quad (4.33)$$

$$= n_P \cdot \left(S_{pr} + S_{sh(ro)} + S_{sh(rw)} + \sum_i C_{rw,i}^{cato} \right) + C^{cato}(n_P) \quad (4.34)$$

$$= n_P \cdot \left(S_{pr} + S_{sh} + \sum_i C_{rw,i}^{cato} \right) + C^{cato}(n_P) \quad (4.35)$$

$$= n_P \cdot S_{pr} + n_P \cdot S_{sh} + n_P \cdot C_{rw}^{cato} + C^{cato}(n_P) \quad (4.36)$$

$$= n_P \cdot S + n_P \cdot C_{rw}^{cato} + C^{cato}(n_P) \quad (4.37)$$

$$= n_P \cdot (S + C_{rw}^{cato} + C^{cato}(1)) \quad (4.38)$$

The same number of COUs is used, i.e. $n_P \stackrel{!}{=} n_T$. A contradiction proof can be used to show that the stack memory usage of the original application is less than that of the modified application. Suppose $S^{cato}(n_P) < S^{omp}(n_T)$:

$$S^{cato}(n_P) \stackrel{!}{<} S^{omp}(n_T) \quad (4.39)$$

$$n_P \cdot (S + C_{rw}^{cato} + C^{cato}(1)) < S_{sh} + n_T \cdot (S_{pr} + C_{sh}^{omp} + C^{omp}(1)) \quad (4.40)$$

$$n_P \cdot (S_{sh} + C_{rw}^{cato} + C^{cato}(1)) < S_{sh} + n_T \cdot (C_{sh}^{omp} + C^{omp}(1)) \quad (4.41)$$

$$(n_P - 1) \cdot S_{sh} + n_P \cdot (C_{rw}^{cato} + C^{cato}(1)) < n_T \cdot (C_{sh}^{omp} + C^{omp}(1)) \quad (4.42)$$

In Chapter 5 it will become obvious that by design of CATO it is impossible that the overhead induced by OpenMP ($n_T \cdot (C_{sh}^{omp} + C^{omp}(1))$), which by itself is marginal, could become larger than $((n_P - 1) \cdot S_{sh} + n_P \cdot (C_{rw}^{cato} + C^{cato}(1)))$, even if only a single process (i.e. $n_P = 1$) is used. Therefore Equation (4.42) cannot be true.

The modified application consumes more stack memory than the original application, resulting in an increased stack memory usage.

Comparison: Heap

The second comparison has a slightly different goal. In the case of stack usage, which is kept low in the best case, Section 4.1.4 compares the minimum stack usage required. In this case, the goal is exactly the opposite: The goal is not to keep the used heap memory to a minimum, but to keep it to a maximum, which is one of the main research questions (cf. Section 1.2.1). Therefore, it must be analysed whether the original or modified version of the target application can use more heap memory (assuming that the user can influence the behaviour of the application via input parameters). A contradiction

proof can be used to show that the potential heap memory consumption of the original application is less than that of the modified application. Suppose $H_{max}^{cato} \leq H_{max}^{omp}$, where $H_{max}^{omp|cato}$ is the maximum heap usage configuration; again, the same number of COUs will be used.

According to Equation (4.10) and Equation (4.18), the maximum potential size of the heap is related to the maximum size of the memory of the participating compute nodes. The stack memory also uses some of the memory, but is ignored in this calculation because the stack memory consumption is significantly less than the heap memory consumption. Assuming that the heap is increased in order to use the corresponding compute nodes at full capacity, the heap memory usage can be replaced by the compute node memory in Equation (4.44):

$$H_{max}^{cato} \stackrel{!}{<} H_{max}^{omp} \quad (4.43)$$

$$M_{max}(n_P) < M_{max}(1) \quad (4.44)$$

$$n_P \cdot M_{max}(1) < M_{max}(1) \quad (4.45)$$

Since $n_P \geq 1$ this gives that Equation (4.45) is false. Assuming that the maximum heap memory requirement is significantly larger than the minimum stack memory requirement, it became clear that the design of CATO allows the original application to be modified to compute larger input problems.

The modified application can consciously use the combined memory of all participating compute nodes.

Memory Model Consequences

In Section 7.1 two different approaches are presented, how the evaluation of CATO can be performed. The results from this section will be revisited there to discuss the relevance of each approach.

4.1.5. Automatic Code Recognition

A primary goal is to minimise the need for user interaction; users should not have to guide CATO in their work. Therefore, CATO must automatically detect relevant code snippets. Automatic code recognition based on semantical usage is not trivial. There are several possible approaches, ranging from the analysis of lexicographical locality to the use of a data flow graph, *Control Flow Graph* (CFG) or the (extended) *Abstract Syntax Tree* (AST) (Ben-Nun et al., 2018). By focusing on the distribution of OpenMP kernels, CATO can drastically simplify this step: The developer of the original application has written the code and added OpenMP instructions, taking into account his expert knowledge of the problem domain in which his application operates, in order to gain a runtime performance advantage. Under these conditions, valuable information can be derived by focusing on the existing OpenMP kernels:

- **Sections of code to run concurrently.** This is really useful, because otherwise the auto-replacement would have to decide for itself which sections of code to parallelise and which to run sequentially (to avoid unnecessary overheads from parallelising unworthy workloads). By focusing on the OpenMP kernels, it is clear that the developer considered these to be significant in terms of runtime performance.
- **Shared variables indicate memory that should be processed concurrently.** If a variable is declared as `private`, this means that there is no communication between threads regarding this variable. It is therefore very likely that it would be more logical for each process to have its own local copy, rather than forcing sharing. On the other hand, a variable created on the heap and declared `shared` allows two guesses: It is quite large (so it is allocated on the heap) and it is either RO (reading from a single instance without having to lock write access might be advantageous) or threads need to communicate updates of this variable.

4.1.6. Equivalence Classes

Once the memory to be shared has been identified, the code needs to be transformed. Most of the application is likely to remain unchanged, with only the affected memory accesses being adapted. Sometimes equivalent operations already exist. For example, both OpenMP and MPI have the ability to reduce values aggregated over all participating COUs: OpenMP has a `reduction` clause and MPI provides the `MPI_Reduce` operation. These constructs can be substituted directly without much concern. In other cases, additional effort is required to replicate or preserve the behaviour of the original code in a multi-stage semantical reproduction. There are already well-established MPI communication patterns available for certain Dwarves (e.g. an MPI-3 implementation of a 2D stencil solver (Kumar & Blocksome, 2014) or an existing framework for multidimensional stencils (Dursun et al., 2009) or optimised matrix multiplication using MPI (an Mey, 2008)) which can be used as a basis for the replacement code.

Without further analysis of the exact pattern, automatically performing a performance-optimal memory allocation is a hard problem. Related to this is, for example, the optimisation of cache-conscious data placement, which has been shown to be NP-hard (Petrunk & Rawitz, 2002). Therefore, heuristics are used to optimise memory allocation. One consequence of this is that the auto-inserted replacement kernel may be less scalable and perform less well than equivalent hand-written, optimised code. However, since CATO's focus is on additional horizontal weak scaling, and is aimed at users who do not have the ability to implement the memory allocation themselves, this is only a minor drawback. Furthermore, since the user does not have to write the replacement code themselves, as it is already nested within CATO, advanced features of MPI can be used, which are beneficial but might otherwise be too cumbersome and buggy to be written by untrained users. The CATO replacement kernels will use the one-sided communication operations of MPI-3 (cf. Section 3.2.2).

In the best case, for every Dwarf there would be a corresponding EC available in CATO.

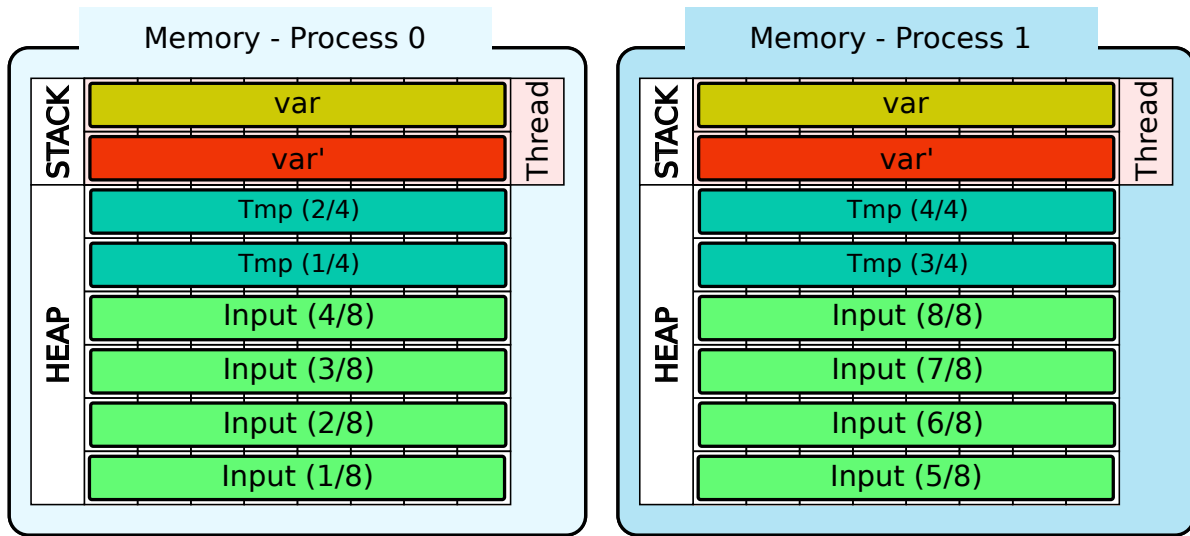


Figure 4.8.: If not only rank 0 but all processes read their share of the input problem concurrently the peak memory consumption of a single node is reduced and allows to load input, which would not fit into a single node's memory otherwise (cf. Figure 1.5).

In fact, it would be even better if there were alternating ECs for each individual Dwarf, since the categorisation on the Dwarf level is still somewhat abstract and could be further subdivided. Also, unpredictable environmental effects (e.g. idle waves induced by shared CPU times (Afzal et al., 2021)) could negatively affect a particular EC while leaving an alternative EC unaffected. This would require an improved design for automatic Dwarf detection, which is currently out of scope, so the focus remains on a single EC for the stencil Dwarf.

Section 5.2 will show how the code detection and replacement is actually done.

4.2. Component: IO Handling

In Section 1.2.2 another enhancement was proposed in addition to memory distribution: CATO allows the user to compute input problems that would otherwise not fit into the main memory of a single compute node (cf. Figure 1.5). There are many ways to I/O (e.g. POSIX I/O or MPI-IO) How to solve this problem is part of the design of this section, which will examine the possibilities of distributing I/O (Section 4.2.1) and reducing the storage footprint using compression (Section 4.2.2).

4.2.1. Parallel IO

Using parallel I/O is not as trivial as using serial I/O, but it has two significant advantages:

1. Each process can load or write its own share of the data.

2. By enabling parallel data access, potentially faster runtime performance can be achieved.

The situation in Figure 1.5 illustrates the problem that the input problem does not fit into the memory of a single node. To solve this problem, the input problem needs to be split up and distributed across enough compute nodes, as shown in Figure 4.8. The trivial approach to doing this distribution is to put the master rank in charge: It reads the input file in chunks and forwards each one to the appropriate process. This sequential order of loading and distributing can be optimised using parallel I/O: Each process participates in the initial read I/O phase and loads its share of the input data directly.

Parallel I/O not only facilitates the loading of large input data but also potentially provides a runtime speedup. Typically, network and disk bandwidth is less than a node's memory and subsequent bus bandwidth and therefore acts as a bottleneck when large data needs to be moved. To solve this problem, parallel FSs, which are common in HPC, can be used. An example of such a parallel FS is Lustre. Lustre is deployed on dedicated storage nodes on which the Lustre servers run and with which the Lustre clients running on the compute nodes can interact. By separating storage and compute resources in this way, Lustre can provide optimal performance for data-intensive workloads. Figure 4.9 shows a rough structure of Lustre:

- MGS A Lustre deployment uses a *Management Server* (MGS), which handles the overall Lustre configuration, and to which every server and client connect at the beginning.
- MDS The MDSs manage the index of the entire FS, which includes the directory hierarchy of Lustre, file names and permissions and other attributes, and most importantly the file layout, i.e. which file chunks are allocated where. A MDS can have several *Metadata Targets* (MDTs) attached to it to improve its storage, but also access times. For small Lustre deployments, it is sufficient to run one instance of a MGS and MDS on a single server.
- OSS Usually there are several OSSs embedded in Lustre. While the MGS/MDS are responsible for managing the metadata, the OSSs store the actual file chunks. Many OSTs, which can be any kind of storage device (e.g. *Hard Disk Drive* (HDD), *Solid State Drive* (SSD) or *Non-Volatile Memory Express* (NVMe)), are attached to a OSS.

It is important to get at least a glimpse of the structure of a Lustre deployment to take advantage of its potential runtime benefits. An example use case of Lustre is demonstrated in Figure 4.9, where chunks of **File A** are stored on a single OSS, but chunks of **File B** are distributed across the OSSs: **File B** is striped across three OSSs. In Figure 4.9a a single compute node loads **File A**. In this case, there are several potential bottlenecks through which the file must be moved:

1. A single OSS must respond to the data request (this bottleneck may be reduced if the file is at least striped across several OSTs on that OSS⁴).

⁴Not shown in Figure 4.9

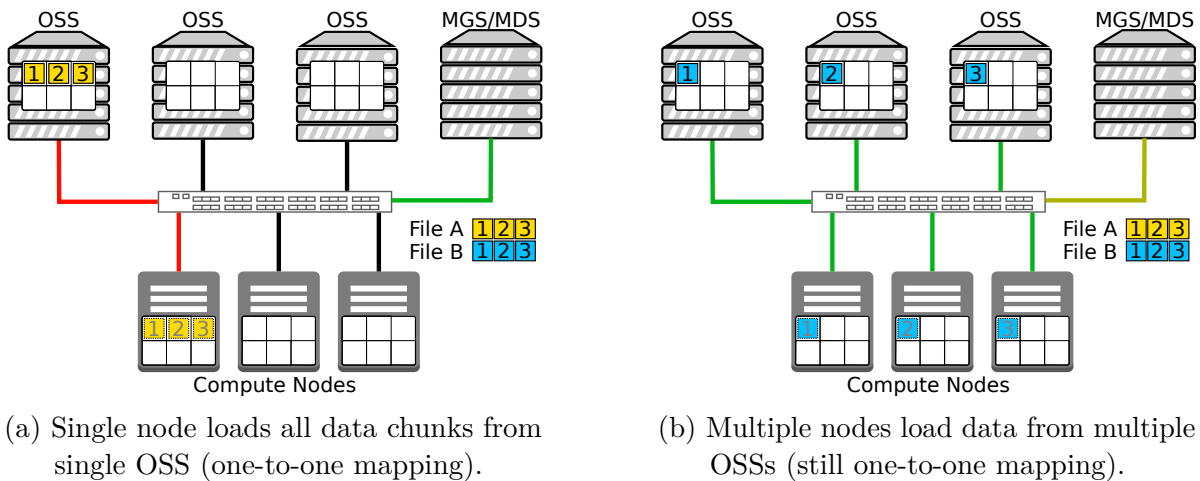


Figure 4.9.: Leveraging performance of parallel I/O by using Lustre. Coloured connections indicate network load.

2. A network.

3. A single compute node that must retrieve the data and move all the chunks from the *Network Interface Controller* (NIC) into memory.

To overcome these bottlenecks, an improved pattern can be used, shown in Figure 4.9b. The requested **File B** is striped across multiple OSSs and each compute node only requests file chunks from a single OSS. In this case, the aggregated bandwidth of the network can be used (the central switch is usually not a problem as all ports are fully connected internally). The only disadvantage is the increased load on the MDS as more compute nodes send requests, but this can usually be neglected.

To benefit from both advantages of parallel I/O, the original code must be transformed by replacing serial I/O instructions with equivalent parallel I/O instructions. This can be done automatically by CATO. The second benefit (improved runtime) depends strongly on the correct striping settings of the file to be used for I/O, and can be changed by the user (theoretically this could also be done by CATO, but this is currently out of scope). In the worst case, the striping does not match the I/O pattern, and all compute nodes are fetching chunks from the same OSS, which would again affect the network bottleneck.

It depends on the I/O framework used to choose a suitable replacement for parallel I/O, if one exists. For example, to perform parallel I/O using POSIX one could use pThread or MPI-IO could be used, or if serial netCDF calls are used, the parallel netCDF interface could be used instead. This design will focus on the handling of the netCDF interface.

Automatic Code Recognition and Replacement

There are two important classes of netCDF operations that need to be replaced, and therefore need to be recognised:

1. Opening an existing or new netCDF file
2. *read* and *write* operations

NetCDF offers special functions for parallel I/O to include all involved processes together: `nc_open_par` and `nc_create_par`. CATO must look for the occurrence of the serial variants `nc_open` and `nc_create`, which can then be modified by the EC. The code allocation calls, on the other hand, do not need to be modified because the memory allocation has already been done by the CATO component for memory handling (cf. Section 4.1). In the current design, every heap allocation call is distributed, so this works regardless of the presence or absence of OpenMP.

What remains are the *read* and *write* operations on the allocated memory buffer, which has already been shared. In the original version, when a single process or its master thread accesses the netCDF file on the FS, the data is mirrored to the heap, on which the master thread (and possibly other threads) can operate directly using local I/O functions (cf. Figure 4.10a). Therefore all calls to `nc_get_var*` and `nc_put_var*` have to be detected and modified, because the local buffers are now smaller than in the original serial version and the address references have to be adapted. `nc_get_vara*` and `nc_put_vara*` have to be inserted with adapted vectors indicating the *start* and *count* of the accessed values. These are only needed if the local memory area is to be read from or written to the file within Lustre. All other memory accesses can be handled by unmodified local operations or must be replaced by MPI operations. The latter is already done by the memory handling component and therefore does not need to be considered in the I/O component (cf. Figure 4.10b).

How the code replacement is done in detail is shown in Section 5.3. Preparing the code of the EC in advance offers significant advantages, as it allows for the consideration of best practices and potential optimisations (Bartz et al., 2015; Lawrence et al., 2017) that might otherwise be too complex for the user to implement or even be aware of.

4.2.2. Compression

Sections 1.2.3 and 3.3.2 has shown that there are several ways to reduce the physical size of data using compression, this work focuses on compression using netCDF. Using a binary compressor directly on the input or output data already works, but has several drawbacks: The metadata is also compressed, and before the data can be used it must be fully decompressed. Therefore, using the compression capabilities of netCDF is preferable, as it still allows access to the metadata. I/O operations via the netCDF API will perform the necessary compression or decompression operations transparently and only on the required chunks of data rather than on the whole file at once. The current netCDF-4 library backend uses HDF5 (cf. overview of I/O layers in Figure 3.12) and can therefore use 30 filters (HDF Group, 2023) officially supported by the *HDF Group* to apply lossless or lossy compression to data chunks. Since HDF5 1.10.2 compression can be used in combination with MPI-IO, netCDF supports this HDF5 feature since version 4.7.4 (HDF Group, 2018; Unidata News Comments, 2020).

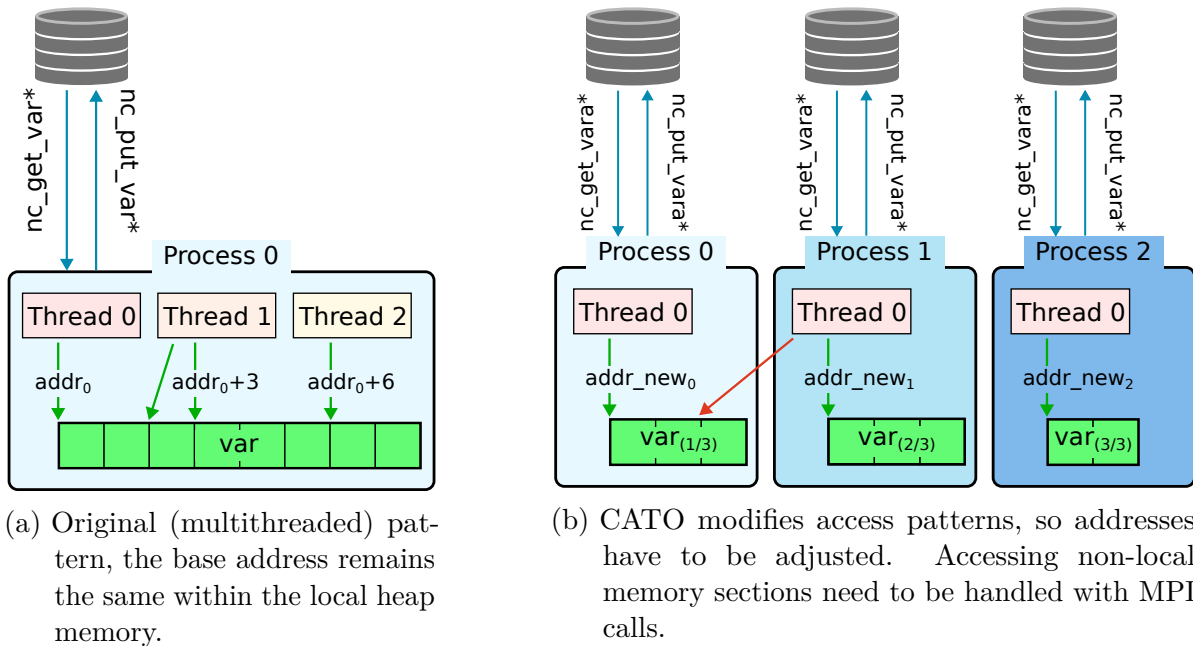


Figure 4.10.: Demonstration how the access pattern has to be adjusted by CATO. Green arrows signalise accesses on local memory, red arrows on remote memory and blue arrows FS access (e.g. Lustre).

To further improve the compression ratio of lossless compression, lossy compression can be used. There are already some lossy compressors available as HDF5 filters (e.g. SZ (Di & Cappello, 2016) or ZFP (Lindstrom, 2014)), but netCDF itself also provides a quantisation filter (Unidata, n.d.-a). This can be applied to floating-point variables and harmonises insignificant bits (using *Bit Grooming*, *Granular Bit Round* or *Bit Round*). By itself, this filter is lossy, but does not yet reduce the data size. However, it can significantly improve the performance of lossless compressors applied on top of it.

The steady state of disks is full (Vader, 2016)

Automatic Code Recognition and Replacement

It is very difficult to choose the best compressor for a specific use case, which is discussed in more detail in Section 3.3.2. Choosing the right algorithm for a specific use case is a research topic in itself and is therefore beyond the scope of this work (Plehn et al., 2022). Instead of an automatic decision component, the user, who knows his use case best, has to decide if and how to apply compression to the data. To support the user and relieve him of the burden of implementation, CATO takes care of this part.

To use compression, chunking must be enabled, and the compressor must be enabled during variable definition. CATO looks for these (`nc_def_var`) in the original code and adds the necessary netCDF operations during their definition epoch. To enable chunking,

`nc_def_var_chunking()` is used, but the way to enable compression is inconsistent as for some explicit netCDF functions:

- `nc_def_var_blosc` for Blosc (The Blosc Developers, 2023)
- `nc_def_var_bzip2` for bzip2 (Seward, 2019)
- `nc_def_var_szip` for Szip (HDF Group, n.d.)
- `nc_def_var_deflate` for Zlib (Gailly & Adler, 2023)
- `nc_def_var_zstandard` for Zstandard (Collet, n.d.)

The other HDF5 compressors can be used as a so-called filter, which is set during the definition of the variable via `nc_def_var_filter` and takes a certain number of parameters depending on the individual filter. How the code replacement is done in detail is described in Section 5.3.2.

The decision about which type of compression and which settings to use is made at the time the application is run. Otherwise, the user would have to recompile his application every time he wants to change the compressor or its settings. Therefore, all potential compressors are built in but remain shielded behind `if` conditions. By setting the correct environment variables to select a compressor and control its parameters, the user can then decide which branch of the EC to run without knowing the details of the implementation itself. CATO provides instructions (cf. Section 4.3) for the user to know how to control this approach.

Most compressors require their library to be installed on the system, or netCDF cannot use them. The same applies if the data is to be read again on another system: The compressor must also be available there. The reading application does need to be modified, as the compressor and its configuration are automatically written into the variable's attributes, and netCDF can therefore handle this automatically. Setting up the necessary libraries is not the responsibility of CATO but of the user.

4.3. Component: Feedback

The goal of CATO is to allow the user to benefit from HPC techniques that are automatically and transparently integrated into the original application. The resulting improved binary could then be executed by the user, but the modified code snippets remain a black box. Only the modified IR code can be viewed, but due to its larger size and low-level syntax, it is difficult to read for anybody, who is not familiar with compilers or assembly. Therefore, a number of ways are explored to help users understand the changes and how to apply them themselves. By addressing the following points, the questions posed in Section 1.2.4 can be answered:

- How can CATO give the user an idea of what the modified high-level code might look like? (Section 4.3.1)

- How can CATO help the user to avoid known pitfalls? (Section 4.3.2)
- How can CATO help the user to gather performance metrics? (Section 4.3.3)
- How can CATO help the user to understand how to implement the HPC frameworks used by CATO? (Section 4.3.4)

4.3.1. Decompile

Obviously, an easy way to allow the user to understand the code changes would be to give them the high-level code version of the modified code. According to the decisions made in Chapter 2, the modifications provided by ECs written in C++ are performed at the IR level, which means that a high-level code version of the modified code is now available. However, it is still possible to generate an equivalent high-level code by using a decompiler. A decompiler generates high-level code from low-level code, i.e. it reverses the original compilation process. It has almost the same phases as a compiler (cf. Section 3.1.1) and produces an equivalent high-level code based on the derived CFG and IR code (Cifuentes, 1994). The resulting high-level code is usually not literally equivalent, but semantically equivalent, because some information is lost during the compilation process. This can result in code that compiles and behaves the same as the original code, but is difficult for humans to read.

Since CATO generates not only the final binary, but also the modified IR code of the original application, there are two possible approaches: The decompiler generates high-level source code from the binary, or from the IR code. Theoretically, there is a third approach, which is to decompile the binary into IR (e.g. by using Dagger (Bougacha, 2017; Kirchner & Rosenthaler, 2017), `llvm-mctoll` (Yadavalli & Smith, 2019, 2022)) or Rellume (Engelke & Schulz, 2020a) first and then use a decompiler on that code, but this will not provide any additional information that is not already in the IR generated by CATO and will therefore not be examined further.

Binary Decompile

To decompile a binary into high-level C code, RetDec (Křoustek, 2014, 2023) can be used. According to the corresponding GitHub page, development of the tool is currently on hold, but it still works with current versions of LLVM⁵. The results are quite readable compared to the original source code. As mentioned earlier, some information is lost in the compilation process. For example, the reference to the size of the `int` data type for memory allocation in the original code (Listing 4.2, line 5) is completely lost in the decompiled version (Listing 4.3, line 6), where only the exact machine-dependent number of bytes remains. Also the original `for` loops (Listing 4.2, lines 6 and 9) have been replaced by `while` loops (Listing 4.3, lines 13 and 25); reading and writing variables has also become much more cryptic than before. And similar to memory allocation, the first loop condition (Listing 4.2, line 6 respectively Listing 4.3, line 13) has changed

⁵Tested with LLVM 13.0.0 and LLVM 15.0.7

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main() {
5      int *buffer = malloc(sizeof(*buffer) * 4);
6      for(size_t i = 0; i < 4; i++) {
7          buffer[i] = i*10;
8      }
9      for(size_t i = 0; i < 4; i++) {
10         printf("Buffer at position %zu:%d\n",i,buffer[i]);
11     }
12     return 0;
13 }
```

Listing 4.2.: Trivial C code for demonstration of decompilation techniques.

semantically. Instead of iterating over the sequence 0, 1, 2, 3, it has now changed to iterate over 0, 10, 20, 30. This has the same effect as the original version, but is not an obvious change that might make it difficult for the user to follow the code changes.

Later in Section 7.3.4 there will also be an evaluation of how well this approach works with binaries created by CATO.

IR Decompilation

To decompile IR code to high level code, it used to be possible to do this with LLVM in house tools. `llc` could be passed `-march=c` to perform the decompilation, but this feature was removed with LLVM 3.1: ‘It had numerous problems, to the point of not being able to compile any non-trivial program’ (LLVM Project, 2012). An unofficial successor has been built, called LLVM-CBE (JuliaHubOSS, 2022). Currently LLVM-CBE only supports LLVM up to version 10, but is able to decompile the trivial example from Listing 4.2. Listing 4.4 shows a snippet of the decompiled C code.

Compared to the version generated by RetDec (cf. Listing 4.3) this is much harder to read. The whole snippet only shows the memory allocation using `malloc` in line 86, the rest of the code from line 94 is needed to express the first `for` loop.

An alternative tool for decompiling IR into high level code is `llvm2c`. It has similar version restrictions on LLVM as LLVM-CBE, but compared to RetDec and especially LLVM-CBE, the decompiled code in Listing 4.5 looks more readable. The allocation call in line 16 is not much different, but the two loops in line 20 and line 29 are easier to see. `llvm2c` has significantly fewer runtime options than LLVM-CBE and RetDec, so it is questionable how each decompiler behaves on more complex code.

Another approach is *LLVis*, which has been created in a project supervised by me, which is based on the idea of merging the CFG and the original high-level code. *LLVis*

```
1  #include <stdint.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main() {
6      int64_t mem = (int64_t)malloc(16); // 0x401149 //
7      int64_t v1 = mem; // 0x401156
8      int64_t v2 = 0;
9      *(int32_t *)v1 = (int32_t)v2;
10     int64_t v3 = v2 + 10; // 0x40116e
11     v1 += 4;
12     int64_t v4 = 0; // 0x40116e
13     while (v2 != 30) {
14         // 0x401160
15         v2 = v3;
16         *(int32_t *)v1 = (int32_t)v2;
17         v3 = v2 + 10;
18         v1 += 4;
19         v4 = 0;
20     }
21     uint32_t v5 = *(int32_t *) (4 * v4 + mem); // 0x401180
22     printf("Buffer at position %zu:%d\n", v4, (int64_t)v5);
23     int64_t v6 = v4 + 1; // 0x401193
24     v4 = v6;
25     while (v6 != 4) {
26         // 0x401180
27         v5 = *(int32_t *) (4 * v4 + mem);
28         printf("Buffer at position %zu:%d\n", v4, (int64_t)v5);
29         v6 = v4 + 1;
30         v4 = v6;
31     }
32     // 0x40119c
33     return 0;
34 }
```

Listing 4.3.: Extract of the decompiled code using RetDec on the binary, which has been built from the high-level code on Listing 4.2.

```
85  _1 = 0;
86  _5 = malloc(16);
87  _2 = (((uint32_t*)_5));
88  _3 = 0;
89  goto _18;
90
91  do {      /* Syntactic loop '' to make GCC happy */
92  _18:
93  _6 = _3;
94  if ((((((uint64_t)_6) < ((uint64_t)UINT64_C(4)))&1))) {
95      goto _19;
96  } else {
97      goto _20;
98  }
99
100 _19:
101 _7 = _3;
102 _8 = _2;
103 _9 = _3;
104 *((&_8[(((int64_t)_9)])) = (((uint32_t)(llvm_mul_u64(_7, 10))));
105 goto _21;
106
107 _21:
108 _10 = _3;
109 _3 = (llvm_add_u64(_10, 1));
110 goto _18;
111
112 } while (1); /* end of syntactic loop '' */
```

Listing 4.4.: Extract of the decompiled code using LLVM-CBE on the IR code of Listing 4.2.

```
9  int main(void){
10     unsigned int var0;
11     unsigned int* var1;
12     unsigned long var2;
13     unsigned long var3;
14     block0:
15     var0 = 0;
16     var1 = ((unsigned int*)malloc(16));
17     var2 = 0;
18     goto block1;
19     block1:
20     if (var2 < 4) {
21         (((unsigned int*)(var1)) + var2)) = ((unsigned int)(var2 *
↪ 10));
22         var2 = (var2 + 1);
23         goto block1;
24     } else {
25         var3 = 0;
26         goto block5;
27     }
28     block5:
29     if (var3 < 4) {
30         printf(&_str[0]), var3, (((unsigned int*)(var1)) + var3));
31         var3 = (var3 + 1);
32         goto block5;
33     } else {
34         return 0;
35     }
36 }
```

Listing 4.5.: Extract of the decompiled code using `11vm2c` on the IR code of Listing 4.2.

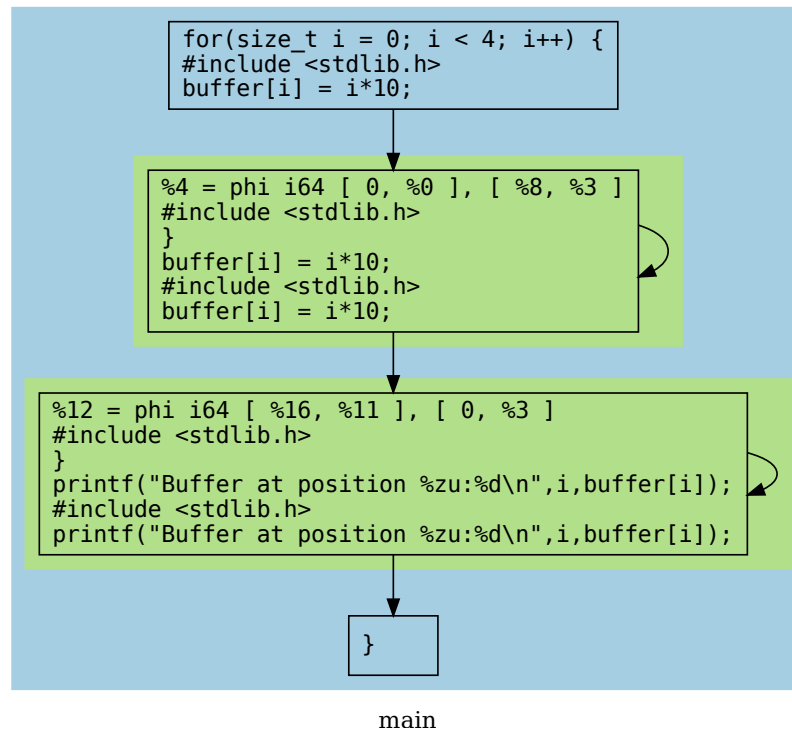


Figure 4.11.: CFG of Listing 4.2 created by *LLVis*. Original high level code replaces corresponding IR code to increase readability and stay close to the real original code.

is an LLVM pass, which is based on the original LLVM *RegionInfoPass* and has been extended. The original pass creates a visualisation of the application's CFG and collects IR instructions that belong to the same region. *LLVis* replaces the IR code with the corresponding lines from the original high-level source whenever possible, using the mapping provided by debug symbols. This works to some extent: Figure 4.11 shows the corresponding CFG with the high level code mapping of Listing 4.2. The rough structure of the original application is visible, but some important code sections are missing (e.g. `malloc`) or have been added several times.

Providing a decompiled version of the modified IR code seems an obvious solution to the question of how to show the user what has happened during the transformation phase. Although there are (limited) promising results for the trivial application, it may be much harder to get a decompiled version of the transformed code. It may not even help the user to understand the changes due to the generic nature of the ECs. Unlike the manually written and therefore more polished code, the decompiled modified code could confuse the user rather than help them understand the changes.

One solution might be to let CATO do the decompilation step, since it has access to

both the original source code and the high-level replacement code. However, this is not a trivial task, since there are also direct changes at the IR level without corresponding high-level source code. Therefore, it will probably be necessary to modify an existing decompilation tool there, which is beyond the scope of this work.

4.3.2. Sanity Checks

The current design of CATO relies on detecting and replacing function calls in unmodified IR code. Therefore, the automatic replacement of function calls also provides an opportunity to perform sanity checks that are not always performed in user-written code. One reason for this may be that including them can become tedious, or the user may simply forget to include them. Using the automatic approach of CATO allows you to work out a sanity check once and include it in the replacement code so that the check is applied automatically.

For example, these three sanity checks for netCDF code can be included quite easily:

Error checking: Each netCDF function returns an integer error code which must be explicitly checked, otherwise an error may go unnoticed at runtime. To enforce that the return value is at least caught, a function could be annotated with the attribute `__attribute__((warn_unused_result))`, which is supported by gcc (GCC Team, n.d.) and clang (LLVM Project, 2023a). If the return value of a function with this annotation is not assigned to a variable in the source code, a warning is raised at compile time. However, this would require code changes to the declaration of the (presumably external) function itself, and so far netCDF does not use them. CATO can work around this problem by intercepting the return values of netCDF functions and inserting checks if they are not equal to zero. In this case a warning is printed by default (cf. Section 5.4.2).

File existence: Attempting to open a file that does not exist with `nc_open` or `nc_open_par` fails and the return value is set accordingly. CATO can check beforehand if the requested file really exists. This is similar to the error checking above, but can give a more sophisticated answer even before the netCDF function is called.

Close file: Any file opened by `nc_open` must be closed at the end. CATO can check if this is done for the corresponding file handles.

How a failed sanity check is handled is up to the user and can be changed by setting an environment variable. If a sanity check fails, a warning may be printed, the application may be aborted, or it may simply be ignored. Even if the sanity checks are simple, they can, like compiler sanitisers, improve the overall quality of the code without additional user effort. In the case of netCDF, if the user does not check return values to implement some kind of error handling, the application may still continue, but its state will be undefined. Depending on the behaviour of the subsequent application, or even luck, the application may still exit successfully or fail during execution.

A study conducted on the IBM Blue Gene/Q Mira HPC system looked at the causes of job failures, examining hundreds of thousands of jobs over a period of more than five years. They found that I/O errors such as missing files were a problem. Focusing only on jobs that failed, and excluding jobs that failed only because of a timeout (which is not necessarily a bug, but rather a misconfiguration of the job), I/O-related failures account for about 8% of all job failures (Di et al., 2019). The intended simple sanity checks could reduce the number of I/O-related failures if they are applied extensively.

4.3.3. Performance Metrics

Measuring performance metrics is an important step when benchmarking an application to evaluate the efficiency of code changes or different runtime parameters. One important metric is the time it takes the application to run. Another, especially with respect to the memory focus of CATO, is peak memory consumption. Focusing on a single node, if the application had a peak memory usage above the available memory, it would be terminated prematurely. It is therefore beneficial for the user to gain a general understanding of the memory behaviour of the modified application.

By way of comparison, measuring runtime is fairly straightforward, but measuring peak memory usage requires a bit more effort. If CATO could collect these metrics automatically, it would be helpful to the user.

4.3.4. Providing User Guidance

CATO performs the IR code modification transparently. The purpose of CATO is to allow the user to easily try out different HPC technologies to observe the effects and assess their suitability for their use cases. By definition, it is always possible to write code manually that is at least as good as automatically generated code (not counting the effort involved). In the long run, it would be best if the user could do the code transformation themselves to write more efficient code that is specifically tailored to their use case. To help the user in his learning effort, CATO can provide an overview of relevant netCDF or MPI properties and resources such as learning material and code examples. Relevant properties could be, for example, an estimate of the level of difficulty and entry points that are important with respect to the HPC environment.

The common structure might change depending on the associated HPC technique, but in general interesting topics to be mentioned by CATO might be:

- Generic description
- Reference to the (official) documentation
- Tips for usage
- Reference to existing code examples

This documentation could be outsourced to a separate helper script, or integrated directly into CATO. In the latter case, the output of the help messages could be controlled by the user using additional flags or environment variables. The information provided by CATO is discussed in Section 5.4.

4.4. General CATO Workflow

A major goal is to provide a user-friendly interface for using CATO (cf. Section 1.3.2). The generation of the IR code, the execution of the pass and the building of the final binary are handled by the LLVM infrastructure itself. To use CATO, the following steps are considered, of which only the first and last need to be performed by the user:

1. Execute CATO’s build script on the original application source code
2. LLVM frontend translates original code into IR
3. CATO analyses and modifies IR
4. LLVM backend translates IR into machine code
5. *optional*: Set environment variables to influence the behaviour of the modified application
6. Execute modified binary via `mpirun` or `srun`

To build the modified binary, CATO must provide a build script. Any necessary flags such as include or library paths or required libraries should be passed by setting appropriate environment variables, which could then be picked up by CATO’s build script to pass to the LLVM frontend or backend. Additional environment variables are used to fine-tune the behaviour of the modified binary, which need to be included by CATO.

Since the modified binary then contains MPI function calls, it must also be run as an MPI application. Consequently, the execution of the application (possibly controlled by a run script) must also be adapted. This involves deciding on the number of processes and threads per process. In addition, the user must decide on the optimal binding of processes and threads to the hierarchical topology of his HPC system. Choosing the correct thread/process/data affinity is important to achieve optimal performance (Rabenseifner et al., 2009).

4.5. Summary

Three components are designed to be integrated into CATO to demonstrate the feasibility and versatility of this approach. The first is the memory handling component, which automatically detects OpenMP code and replaces it with equivalent MPI code. This

allows distributed memory to be used, so that larger problem sizes can be used within applications, benefiting their expressiveness. In addition, this is an opportunity to introduce advanced MPI features that are not well known, but can again provide additional benefits. An important computational pattern in the earth and climate science community is the stencil, so the design is geared towards this. Although the stack memory consumption increases due to the overhead of CATO, the big advantage is that the modified application can then use the aggregated memory of the participating compute nodes instead of just a single one.

Based on the memory handling, the second component is designed to take advantage of netCDF and its capabilities regarding parallel I/O and compression. This component can take advantage of the synergy of the memory handling component and benefit from improved performance during the I/O phase, if the hardware environment is suitable.

Finally, the design of a feedback component is presented. This allows us to give real feedback to the user, to give them an idea of what is happening during the modification. By providing hints and guidance on MPI and netCDF, CATO is not just a black box, but can be a resource for the user if they wish to delve further into these frameworks.

5. Mapping onto LLVM

In Chapters 2 and 4 several approaches were discussed on how to perform the central mechanics of CATO, code detection and transformation by ECs. The decision was made to perform the analysis and transformation step at the IR level using LLVM. First, the Section 5.1 explores different approaches to how the interaction on the IR can actually be realised. Based on these findings, the transformation pass, which forms the core of CATO, is derived in Section 5.1.2. The chosen approach is then applied to the three main components of CATO: *Memory Handling* (Section 5.2), *I/O* (Section 5.3) and *Feedback* (Section 5.4).

5.1. Using the LLVM Infrastructure

LLVM is a three-phase compiler framework, as shown in Figure 5.1, and allows developers to interact with all phases of the compilation of a high-level code by reusing already established parsing and transformation components of the framework. It is also easy to install in userspace as it does not require elevated privileges (which are rarely granted on HPC systems). It contains many quality of life tools (e.g. for linting, testing and debugging) and has been used as a backend for many projects (cf. LLVM Project, n.d.-c). Since its initial introduction in 2004 by Lattner and Adve, it has been widely adopted in many areas and is used by major institutions in both research and industry.

As shown in Figure 5.2, the initial high-level source code is passed to the language-specific compiler front-end, which parses and pre-processes the input code (cf. Figure 3.2). The high-level code is first transformed into an *Intermediate Representation* (IR) and regardless of the chosen frontend, the syntax of the IR code remains consistent. However, they may differ in their general procedure for transforming the code (cf. Listing 2.8 and Listing 2.9). Within the optimiser, the IR code is further analysed and modified, and then passed to the machine-specific backend. There the IR code is transformed into the final binary that can be executed.

Frontend Assuming that the source code is written in C, `clang` would be used as the

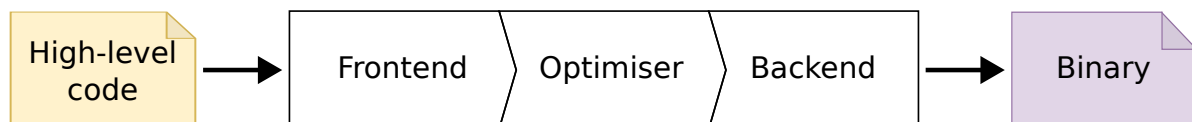


Figure 5.1.: General component overview of a three-phase compiler like LLVM (based on (Lattner, n.d.)).

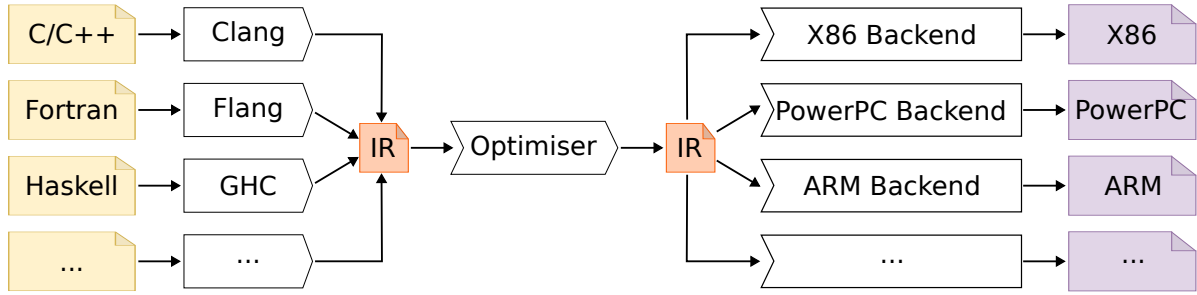


Figure 5.2.: Modular LLVM infrastructure (based on (Lattner, n.d.)).

frontend. Therefore, there are now several ways of using `clang` to interact with the source code (LLVM Project, 2023b).

- *LibClang*: Limited interface for interacting with the AST, which is fairly stable between major releases (LLVM Project, 2023e).
- *Clang Plugin*: A shared library that is loaded at runtime by `clang` to interact with AST in a more comprehensive way.
- *LibTooling*: This library allows you to write standalone tools using `clang` as a backend, with full control over the AST (LLVM Project, 2023f).

Optimiser The optimiser runs passes to iterate, analyse and modify the IR code. It provides good code coverage by making it easier to handle conditions that can only be evaluated at runtime. Since a pass can be freely moved within the chain of passes, the code can be optimised before or after, depending on what best supports the intended workflow.

Backend The main focus of CATO is to analyse and modify the original code, so there is no need for interaction within the backend phase, as its primary task is to generate equivalent machine code from IR.

In Section 2.3.1, the AST layer has been ruled out for code modification, and using the backend to work on low-level machine code does not provide any additional benefits for the required use case. Therefore, the code modification focuses on the optimisation phase of LLVM. In addition, focusing on this phase allows to be somewhat independent of the limitations listed in Section 2.3.1. Moreover, focusing on this phase allows to be independent of the high-level language used, which would otherwise be bound to C if `clang` were used. Since no code changes are required within the LLVM architecture itself (an LLVM pass is a shared library dynamically loaded by the optimiser), there is less hassle when updating the LLVM installation, or the range of acceptable versions is wider. As discussed in Section 3.1.1, there are regular major release updates, in which it is not uncommon to introduce API breaking changes, which would require an update of the pass as well.

The main functionality of CATO’s is implemented as LLVM pass, which can then be inserted into the chain of passes executed by the Pass Manager during the optimisation

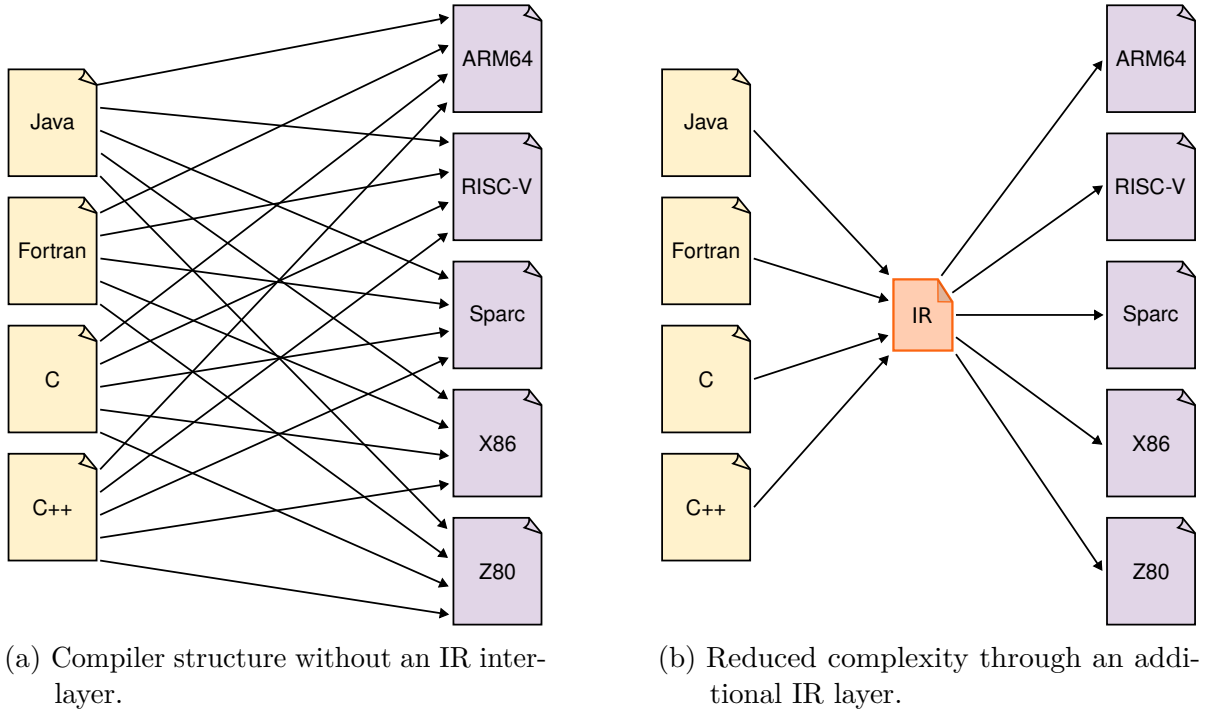


Figure 5.3.: Comparison of compilation workflows with and without an intermediate language layer (based on (Appel, 1998)).

phase. This allows the IR code to be analysed and modified according to the tool design elaborated in Chapter 4. How the new pass will interact with the IR code is discussed in Section 5.1.2.

5.1.1. Dissecting IR

A three-phase compiler uses frontends to handle different types of high-level languages allowed as input, and backends to address different machine architectures to produce the final binary. In an HPC environment, there is no single dominant high-level language or machine architecture. One could argue that C, C++ and Fortran are dominant, but depending on the scientific domain, other high-level languages such as Python or Java become relevant. The same applies to machine backends. Looking at the installed CPU architectures used by the first five entries in the *Top500* list (Nov 2022 (Strohmaier et al., 2023)), AMD EPYC, Fujitsu ARM, Intel Xeon and IBM POWER9 are represented. Without an abstraction layer between the frontend and the backend, each possible combination of high-level language and machine architecture would have to be considered separately (cf. Figure 5.3a).

Using *Intermediate Representation* (IR) as an additional layer adds complexity, but has significant advantages. It decouples the language-specific compiler frontend from the machine-specific backend. In Figure 5.3b, the IR code generated by each frontend, which always uses the same syntax, is integrated in between. This simplifies the development

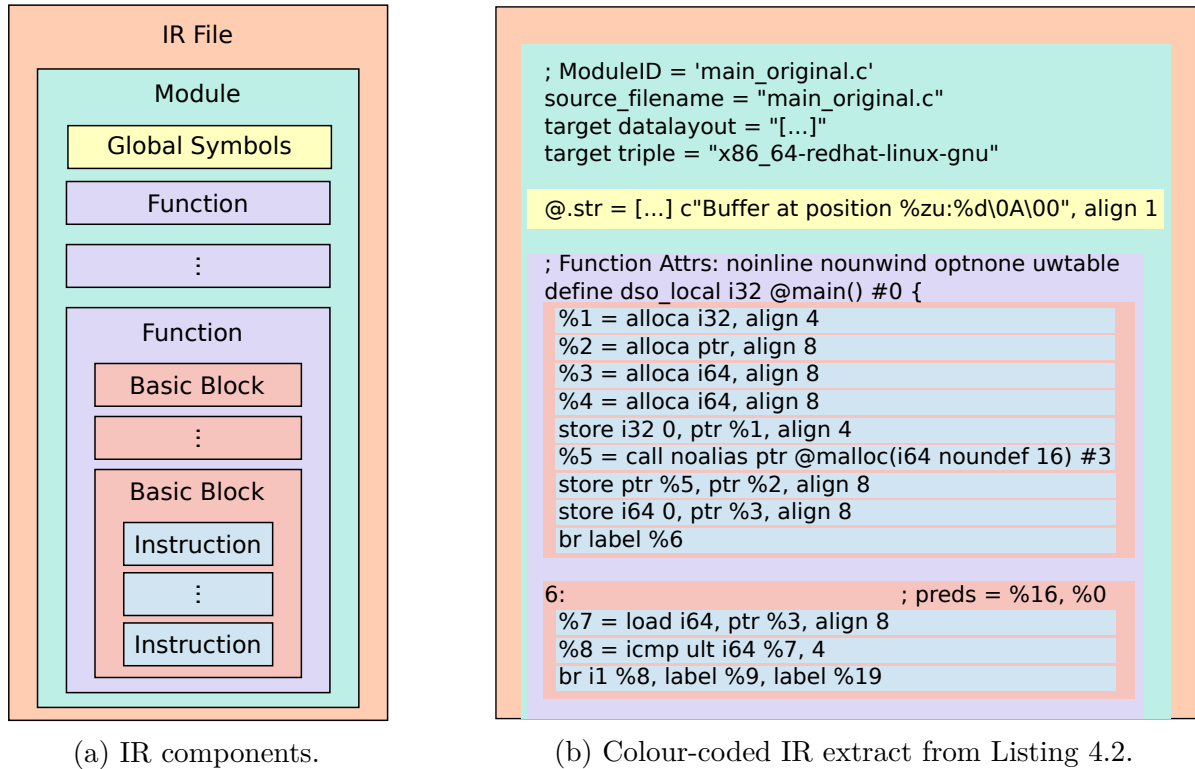


Figure 5.4.: Converting high-level code into IR code creates a hierarchy of LLVM containers, which orient themselves at the high-level code’s structure.

and maintenance of all frontends and backends. It also has the great advantage that code optimisations can be applied at the IR level, so they always work regardless of the high-level language or machine infrastructure used. Many frontends such as `clang` (C/C++) and `flang` (Fortran) and several backends are natively supported by LLVM. For a list of supported backends, see `llc --version`. Additional backends such as support for older hardware such as the Z80 are also available (Cocoacrumbs, 2021).

IR code follows a hierarchical structure, shown in Figure 5.4a. Usually a module corresponds to a compilation unit within the high-level language, although it is possible to have multiple modules within a single IR file. A module basically has global symbols (e.g. global variables or strings) and functions (like the `main` function). Now each function consists of at least one *Basic Block* (BB), which contains a sequence of instructions. A BB has a single entry point and a single exit point (also called *terminator*), which can for example return to the calling frame or switch to another BB based on a branch condition.

The smallest unit in this hierarchy is an instruction, which is a single line of IR code such as a load/store or call instruction (an overview of available instructions can be found in the official LLVM documentation (LLVM Project, 2023j)). Instructions have an assembly-like appearance, but do not impose a specific runtime environment: There are no machine-specific commands and the registers used are virtual, not physical. Variables

are strongly typed and in SSA form to improve the efficiency of dataflow analysis, since each variable or virtual register is defined only once (cf. Appel, 1998, Ch. 19).

At this level, CATO performs the code analysis and transformations.

5.1.2. Performing Code Transformation

During the optimisation phase, the Pass Manager executes a predefined chain of passes to perform code analysis and transformations on IR code and return valid IR code. This allows to change the order of the passes or to insert a new pass anywhere in the chain. In LLVM 15.0.7 there are 332 passes available within the optimiser¹.

There are three different categories of passes built into the optimiser (LLVM Project, 2023l):

Analysis pass: Derives information from the IR code without making any changes, which can be used by other passes or provide insight into the IR (e.g. for debugging or visualisation purposes).

Transform pass: May use information from previous analysis passes and transform the IR in some way.

Utility pass: Any pass that does not fit into the other categories (e.g. a pass to generate LLVM bitcode).

LLVM provides a **Pass** superclass, which provides the generic interface. There are five subclasses (omitting two legacy pass types), which differ in which part of the IR hierarchy (cf. Figure 5.4a) they are executed in (LLVM Project, 2023m):

- **ModulePass:** Performed on the module level
- **CallGraphSCCPass:** The IR code is divided into SCCs on which the pass is executed.
- **FunctionPass:** Runs on individual functions
- **LoopPass:** Performed on individual loops
- **RegionPass:** A region is a collection of simple subregions and/or BBs that have a single entry BB and a single exit BB (cf. Figure 5.5). So the use of a **RegionPass** is not necessarily limited to a single BB.

CATO needs access to the whole compilation unit, since heap operations are not limited to a single function. In fact, it needs to access the whole file, so only single file applications are currently supported. This can be done for any application, but may require some manual work (e.g. to resolve name collisions). Therefore, CATO inherits from **ModulePass** and redefines the corresponding **runOnModule** entry point, which is executed by the Pass Manager.

¹Result of `opt -print-passes`.

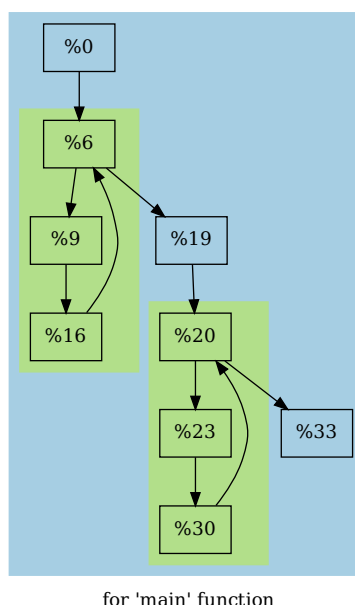


Figure 5.5.: Region plot of Listing 4.2, the numbers are labels and each represents a single BB. Each coloured block represents a (sub)region.

The pass must be registered so that the Pass Manager knows to run it. There are twelve different possible entry points², but not each one can be used for every pass subclass.

To benefit from the optimisations of the pass pipeline, it makes sense to register CATO as early as possible in the Pass Manager’s workflow. Therefore `EP_ModuleOptimizerEarly` is used so that CATO is executed before the `main` module is optimised. In addition, the pass defines a `isRequired` function which returns `true`, so that the Pass Manager cannot simply decide to ignore this pass (LLVM Project, 2023n). Otherwise, elements with the `optnone` attribute could be ignored. The workflow of CATO is shown in Figure 5.6:

1. The user calls the CATO wrapper on their C high-level source, `clang` generates the corresponding IR code.
2. The IR code is passed to the Pass Manager, which is responsible for running the IR through the passes pipeline.
3. CATO is inserted early in the pass pipeline, then the modified IR code is run through the rest of the pipeline as usual.
4. The modified IR code is compiled by the backend into the final machine code.

Currently CATO supports transforming a single C code file using this approach. This could be extended by using `llvm-link` to link all IR code files into a single one, which could then be modified by CATO.

²cf. `include/llvm/Transforms/IP0/PassManagerBuilder.h`

```
1 std::vector<std::unique_ptr<MemoryAllocation>>
  ↪ find_memory_allocations(Module &M) {
2     std::vector<std::unique_ptr<MemoryAllocation>> mem_allocs;
3     std::vector<StringRef> allocation_functions = {"malloc", "calloc",
  ↪     "_Znam", "_Znwm"};
4
5     for (auto alloc_func : allocation_functions) {
6         auto alloc_users = get_function_users(M, alloc_func);
7
8         for (auto &user : alloc_users) {
9             if (auto *call = dyn_cast<CallInst>(user)) {
10                 mem_allocs.push_back(std::make_unique<MemoryAllocation>(call));
11             }
12         }
13     }
14     return mem_allocs;
15 }
```

Listing 5.1.: Searching allocation calls within IR code.

Since CATO is a `ModulePass`, the `main` function serves as an entry point into the IR code. The pass traverses the different levels (cf. Figure 5.4a) looking for specific IR instructions to perform the required analysis and transformations. Listing 5.1 shows an example of how such an analysis can be performed at the module level:

1. line 3: A vector of allocation functions to search for with CATO (`_Znam` and `_Znwm` are mangled names of C++ `new` expressions). C++ allocation functions are considered, but C++ has not been extensively tested.
2. line 6: Returns all `llvm::User` references that call one of the allocation functions.
3. line 9: Check if a `llvm::User` reference is actually a `llvm::CallInst`. If it is, CATO creates a `MemoryAllocation` abstraction object to handle this particular allocation call.

After the analysis, the code transformation itself can be performed. These transformations range from simple replacements of function calls (while retaining the arguments of the replaced function call) to entirely new added operations. Simple replacement of a function can be done by using the function `setCalledFunction` of a `llvm::CallInst` to be replaced. This is only sufficient if the original arguments remain the same. Otherwise, an `IRBuilder` can be used to construct a new LLVM statement, which can then be inserted at an explicitly set insertion point.

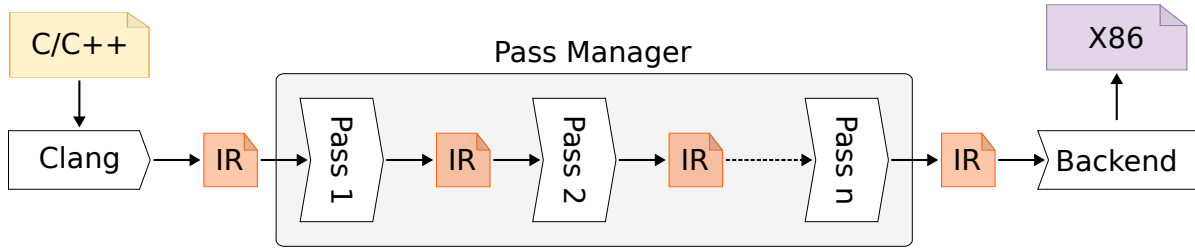


Figure 5.6.: Workflow through the Pass Manager (based on (Sampson, 2015))

```

1 MemoryAbstraction::MemoryAbstraction(...);
2 MemoryAbstraction::~~MemoryAbstraction();
3
4 void MemoryAbstraction::store(...);
5 void MemoryAbstraction::load(...);
6 void MemoryAbstraction::sequential_store(...);
7 void MemoryAbstraction::sequential_load(...);
8
9 void MemoryAbstraction::pointer_store(...);
10
11 void *MemoryAbstraction::get_base_ptr();
12 long MemoryAbstraction::get_size_bytes();
13 MPI_Datatype MemoryAbstraction::get_type();

```

Listing 5.2.: Generic memory abstraction interface.

5.1.3. Equivalence Classes

CATO has two parts: The first is the LLVM pass itself, which is built as a shared library and then loaded by the optimiser at compile time. How this part works in general has already been discussed.

The second part is the runtime library, where the replacement codes or ECs are stored. ECs are implemented as C++ classes that follow an elaborate interface shown in Listing 5.2. This ensures that there is a suitable operation for every possible situation, and that one EC can be supplemented or replaced by another. The interface includes operations for initialisation (line 1) and load (lines 5 and 7) and store accesses (lines 4 and 6, 9).

The runtime library is built separately as LLVM bytecode and loaded by CATO as an LLVM module using the `getLazyIRFileModule` function. Following this modular design principle, these functions can then be accessed and used by CATO. The central function is shown in Listing 5.3, which allows you to use a function from the replacement module referenced by its mangled name. First it is loaded from the module in the line 3. Then in line 5 a table lookup of the symbol is performed to get the function callee, from which the callee pointer can be inserted into the IR code. Since the called pointer is just an

```
1 void RuntimeHandler::match_function(llvm::Function
  ↳ **function_declaration, llvm::StringRef name)
2 {
3     llvm::Function *func = _rtlib_module->getFunction(name);
4     if (func != nullptr) {
5         *function_declaration =
  ↳ cast<llvm::Function>(_M->getOrInsertFunction(func->getName(),
  ↳ func->getFunctionType()).getCallee());
6     }
7     else {
8         errs() << name << " was not found in rtlib\n";
9         *function_declaration = nullptr;
10    }
11 }
```

Listing 5.3.: Setting up the connection between the CATO’s core and the external replacement code.

instance of `llvm::Value *`, this must be cast to `llvm::Function *`.

Loading the replacement from an external runtime library has the great advantage that the replacement code can be written in C++, which is then translated by LLVM into IR code. This makes the development of replacement code much easier and less error-prone, as it does not have to be written manually as IR code. However, now a new category of uncertainty comes into focus: semantical correctness. The local behaviour is different, but the result due to side effects must be the same. As far as the output of the application is concerned, the user should see no difference between the original and the modified binary.

The ECs must mimic the behaviour of the original code while using newly added frameworks such as MPI. This makes developing an EC more complex, and correctness more difficult to ensure. Proving correctness is already a very difficult (generally impossible) task, usually using a model that ignores the underlying hardware (M. A. Dave, 2003; D’Silva et al., 2015). Measures can be implemented to increase confidence in the correctness of the replacement code. For example, one could use PhASAR, which has already been introduced in Section 2.3.1. This may improve the code quality, but is limited to a specific code snippet. Whether a replacement code works correctly can only be determined by running it before and after the original code. Due to the potentially long runtime and effort to implement such a test case, this has been postponed.

Instead, another testing strategy is implemented in CATO:

Unit tests : Individual functions are tested to see if they behave correctly for a given input. CATO uses *GoogleTest* to perform unit tests on some functions within CATO (e.g. from `environment_interaction.cpp`) (Google, 2023). They are quite fast (currently under a second), and therefore serve as a regression test when CATO

is built (they are registered as a test in CATO’s CMake setup).

End-to-end tests : The unit tests are not sufficient to verify that the code replacement works as intended. Therefore, more complex end-to-end tests are set up using LLVM’s internal test suite `lit`. There are currently three categories of end-to-end tests:

- Basic non-OpenMP tests
- Basic OpenMP tests
- Intermediate tests

First, they check if CATO works at all by running it on trivial applications without any OpenMP kernels that would be replaced by CATO. Then there are trivial applications that use OpenMP and are modified by CATO. Finally, there are more intermediate tests that check if CATO can handle not only microkernels, but also a combination of them. The tests consist of several use cases such as (multidimensional) dynamic arrays, pointer aliasing, critical sections, barriers and reductions. `lit` runs CATO on these tests and compares the output with the expected output.

Since these tests have a longer runtime (currently about ten seconds), they are run manually using CATO’s test script `run_test.sh`.

The next Sections 5.2 and 5.3 discuss in detail, how the replacement is performed to achieve the memory distribution and I/O handling.

5.2. Memory Handling

The first use case of CATO deals with memory allocation using MPI processes. How this is done in general has already been discussed in Section 4.1.

The final binary contains MPI operations, so how the modified binary is executed compared to the unmodified binary changes. CATO only modifies the application code, but does not take any action during execution. In order to take advantage of the code modification, the user has to make two adjustments.

- Run the application as an MPI application using the correct startup commands (e.g. `mpirun` or `srun`).
- The input problem or runtime parameters must be adjusted so that the modified binary takes advantage of the memory gains (cf. Section 4.1.3).

CATO uses advanced MPI features that are not easy for untrained personnel to use. Section 3.2.2 has shown that using one-sided MPI communication can be advantageous in terms of performance compared to two-sided or collective communication. On the other hand, the usage can be confusing (considering that some synchronisation calls are called ‘lock’ but handle epochs) and it is hard to keep track of active and passive target

communication operations. Even the official MPI specification document acknowledges this difficulty, as there are only two dedicated chapters on correctness: one for collective communication and one for one-sided communication (MPI Forum, 2021, ch. 12.7). Therefore, the automatic substitution of CATO provides a great way for the user to use one-sided MPI operations without having to fully understand them. During the development, the replacement codes have been carefully prepared and extensively tested.

CATO assumes that the high-level code does not yet include MPI communication. If it did, it would make the replacement much more complicated. And if the user was already able to insert some MPI communication, then he probably had enough knowledge to do the whole replacement himself. In that case, he would probably not benefit from CATO, so this hybrid case is omitted.

Before CATO takes care of the OpenMP kernels, there is some preparation to be done: First, the MPI environment needs to be initialised, which requires the insertion of `MPI_Init` as well as the insertion of `MPI_Comm_rank` and `MPI_Comm_size`. This is quite trivial because the application has only one entry point. So the required functions are simply inserted before the initial BB. The finalisation is a bit more complicated, as there can be several potential exit points of the application. So a call to `MPI_Finalize` is added before each exit point.

The next step is to modify the memory allocation calls. By default, CATO assumes that every heap variable is relevant. This heuristic can lead to *false positives*, but there are probably no relevant (in terms of size) variables on the stack. The chances of missing a relevant variable (or *false negative*) are therefore lower. Listing 5.1 shows how to search for allocation calls. Each occurrence is replaced by an instance of the memory abstraction class that handles the memory allocation. This is an important step: each process allocates only its share of the original memory size using a static load balancing algorithm. At this point, the benefits of the aggregated memory of multiple nodes can be fully exploited. CATO distinguishes between dimensioned and single-valued data, and internally maps the actual memory addresses to their corresponding abstraction object, so that later references to an address can be correctly mapped. During memory allocation, the corresponding MPI windows are also created (using `MPI_Win_create`), which are needed for one-sided MPI communication. All `free` calls will be adjusted accordingly to respect the changed allocation strategy and free the created windows as well.

This is sufficient for one-dimensional variables, but requires additional actions for multidimensional variables. Checking which specific memory address a high-dimensional pointer is actually pointing to is a pointer alias problem, which is a relevant topic in computer science (Hind, 2001; LLVM Project, 2023h; Mock, 2003). Static analysis of pointer aliasing is a difficult problem, which for some circumstances can become NP-hard (Landi & Ryder, 1991). A fully exact pointer analysis is generally undecidable unless heuristics are used (Ramalingam, 1994). Using a dynamic analysis alleviates this problem, but is then limited to a specific run (Guo et al., 2006). The advantage of the CATO approach is that this problem does not need to be solved statically. Instead, it builds trees of all explicit usages of a shared pointer variable to collect all potential usage paths through the IR code, resolving the pointer hierarchy in the process. Each node is

then checked to see if it is a `StoreInst`, `LoadInst` or `CallInst` (e.g. `free`) and replaced accordingly. This approach is limited and currently only works with one-dimensional and two-dimensional pointer variables.

How store and load operations on memory are handled by CATO is discussed in the following Section 5.2.1.

5.2.1. OpenMP Code Handling

There is no general conclusion as to whether an MPI-only approach should be preferred over a hybrid MPI-OpenMP approach, as their suitability varies from case to case (Yan & Regueiro, 2018). In the current state of CATO, the replacement process removes all OpenMP functionality from the original application.

Given the previous chapter on handling memory allocation, the focus is on heap variables. The OpenMP kernels are used to determine how processes can concurrently work on distributed memory. By setting up the kernel, the user provides valuable information about the intended semantics of the code, which can be exploited during automatic modification:

- Which loop is worth parallelising?
- Which variables should be treated as private or shared?

It is difficult to automatically determine which sections of code should be parallelised. There are tools such as Polly, but this is not a challenge to be solved by CATO. Instead, this knowledge is derived directly from the user’s actions. By parallelising sections of code with OpenMP, he has shown that they are worthwhile in this sense.

Moreover, the data-sharing attribute, which the user (implicitly) sets for all variables involved in any way in the OpenMP kernel, allows CATO to make assumptions about how variables are likely to be accessed. According to these assumptions, the replacement can then be performed in the correct way.

The following steps are performed by CATO:

1. **Replace OpenMP functions:** There are some OpenMP runtime functions for which an exactly equivalent MPI version exists. Since the OpenMP functions do not take parameters, they can be replaced by corresponding MPI wrapper functions (cf. Table 5.1).

Original	Replacement
<code>omp_get_thread_num</code>	<code>→ MPI_Comm_rank</code>
<code>omp_get_num_threads</code>	<code>→ MPI_Comm_size</code>
<code>__kmpc_barrier</code>	<code>→ MPI_Barrier</code>

Table 5.1.: Replacement of OpenMP functions by MPI wrapper.

```
1 void __kmpc_fork_call (ident_t *loc,  
2                       kmp_int32 argc,  
3                       kmpc_micro microtask,  
4                       ...)
```

Listing 5.4.: Function signature of `__kmpc_fork_call`.

The OpenMP barrier replacement includes not only explicit barriers, but also implicit ones. For example, an implicit barrier is set when a worksharing construct terminates but the thread team has not yet terminated.

2. **Find and replace fork calls:** The OpenMP kernels can be automatically detected within the IR code. For example, `#pragma omp parallel for` specifies a parallel worksharing loop, which means that new threads are likely to be created. To do the actual forking of threads, `__kmpc_fork_call` (cf. Listing 5.4) is called, which takes the microtask and pointers to shared variables as parameters, among other things. The microtask is an outlined function that represents the body of the corresponding OpenMP kernel. To replace the OpenMP kernel, the fork call is removed and the outlined microtask is called directly instead. This does not spawn any threads, instead each process executes the microtask on its own.
3. **Adjust loops:** This step is performed when a loop has been marked for parallel work sharing (e.g. by using `#pragma omp for`). In the original OpenMP code, the loop is shared among the responsible thread team by assigning different values of the loop counter variable to each member of the team. It depends on the type of `schedule` clause used (`static`, `dynamic`, `guided`, `auto`, `runtime`) how the potential loop counter values are assigned. When using the static scheduler, each thread, knowing its own thread id, computes the loop interval for which it is responsible using `__kmpc_for_static_init_*` functions. The end of the parallel block is marked by `__kmpc_for_static_fini`.

In its current state, CATO is able to split the loop iteration across processes when using the static scheduler. It retrieves the values for the lower bound, upper bound and increment of the loop counter from the parameters of `__kmpc_for_static_init_*`, which is then removed. Then the behaviour of the OpenMP implementation is mimicked and new bounds are computed with respect to the current process rank and the total number of processes. This allows the original behaviour to be replicated using processes instead of threads.

4. **Handle reductions and criticals:** Replacing the critical construct is quite simple. CATO looks for calls to `__kmpc_critical` (start point) and `__kmpc_end_critical` (end point) to determine the sequence of code that must be executed sequentially. These calls are replaced by corresponding calls to a mutex implemented with MPI. It is initialised at the beginning of the microtask and destroyed at the end. As

```
1 kmp_int32 __kmpc_reduce (ident_t *loc,  
2                          kmp_int32 global_tid,  
3                          kmp_int32 num_vars,  
4                          size_t reduce_size,  
5                          void *reduce_data,  
6                          void(*) (void *lhs_data, void *rhs_data) func,  
7                          kmp_critical_name *lck)
```

Listing 5.5.: Function signature of `__kmpc_reduce`.

soon as the critical section is about to be entered by a process, the lock is set (and unset after leaving the section).

Replacing reductions is more complex. Reductions are found by looking for the corresponding function call of `__kmpc_reduce`, shown in Listing 5.5. Parsing the parameters gives information about which variables to reduce with which operation. In the original OpenMP code, the master thread simply stores its variable value in a buffer. Each other thread of the corresponding team performs an atomic read-modify-write operation on this buffer to contribute its variable’s value. CATO determines the type of reduction from the latter branch and replaces the reduction with an `MPI_Allreduce` call with the appropriate parameters. Currently, reductions using the `Add`, `Min` and `Max` operations are supported.

5. **Replace serial and shared accesses:** How the general replacement must be done was already discussed in Section 4.1.5. There are two use cases: variable access inside the microtask and variable access outside the microtask. For each use case there are corresponding functions of CATO, which are shown in Listing 5.2. For access within the microtask, the `MemoryAbstraction` class provides `load` (cf. line 5) and `store` (cf. line 4) and for external accesses there are corresponding `load` (cf. line 7) and `store` (cf. line 6) operations.

This distinction is necessary because the basic idea of the replacement is to focus the parallelisation efforts on the OpenMP kernels and make only minimal changes to the primary serial parts. Since the user does not provide any additional information about the serial parts, CATO treats these parts conservatively, and each rank runs the same code. The main difference is which MPI operations are used. Since the code outside the microtask is executed by each rank simultaneously, the sequence of communication is the same for all ranks and follows the same logical time steps (neglecting system-induced delays). Therefore, active target communication is used and the epochs are handled by `MPI_Win_fence`. To avoid unnecessary multiple write operations, which would all be performed with the same value, only rank 0 performs write operations. Read operations are then performed by all operations. On the other hand, the sequence of operations within the microtask is potentially less homogeneous and therefore passive target communication is used

for memory accesses using exclusive epoch locks (`MPI_Win_lock/MPI_Win_unlock`). To determine if a rank needs to access local or remote memory, CATO provides an operation `get_target_rank_and_disp_for_offset` to get the owner's rank and local offset on its window.

This only applies to shared variables that are also written to within the microtask. If a variable is private, then each rank has its own copy and there is no need for a collective update. The same applies to RO shared variables, as they behave semantically the same. So each process performs the same computations on its own copy, and in the microtask section each process can read its own value, which is the same for all processes. And because there are no write accesses within the microtask, the processes do not update it at all.

5.2.2. Semantical Equivalence

The general testing strategy has already been presented in Section 5.1.3. It is difficult to prove semantical equivalence between the ECs and the corresponding original code kernels. However, the design was chosen to be as close to the original as possible to reduce the number of potential pitfalls.

One potential problem could still arise from floating-point arithmetic. It was discussed in Section 1.2.3 that floating-point accumulation is not associative, although in theory it should be. However, this is a problem that already arises in the OpenMP kernel, where the result can depend on the order and number of threads used (van der Pas et al., 2017, Ch. 2.4.3). The reduction result can be forced to be reproducible by using the `KMP_DETERMINISTIC_REDUCTION` environment variable to specify a fixed order of execution, which is supported by at least the Intel and LLVM OpenMP implementation.

MPI also assumes that the reduction operation is associative, but also notes the non-associative nature of floating-point arithmetic. To enforce a specific order of execution, this could be done by first performing the reduction locally with `MPI_Reduce_local` and then sending it via a fixed communication pattern (MPI Forum, 2021, Ch. 6.9).

So using MPI instead of OpenMP does not introduce any new problems, and if the same number of COUs is used, in many cases it will not be a problem at all. Other than the approaches already mentioned, there are also alternative solutions to this problem (Kapre & DeHon, 2007).

5.3. Input/Output Handling

‘Many users think I/O is free’ – Philippe Deniel

To use this component, it is essential that the original application already uses serial netCDF to read and write data. CATO does not currently support the inclusion of netCDF in code without such an existing basis. Since netCDF is a fairly common data

```

1 // [...]
2 nc_create(...);
3 nc_def_var(...);
4 nc_enddef();
5
6 nc_put_var_float(...);
7 nc_close(...);
8 // [...]

```

(a) Original pseudocode.

```

1 // [...]
2 err = nc_create_par(...);
3 err = nc_def_var(...);
4 err = nc_enddef();
5 err = nc_var_par_access(...);
6 err = nc_put_vara_float(...);
7 err = nc_close(...);
8 // [...]

```

(b) Modified pseudocode using parallel I/O. Checks of error values are omitted for brevity reasons.

Listing 5.6.: Comparison how a netCDF code is roughly modified by CATO with focus on parallel I/O.

format in several scientific fields (e.g. in geoscience or climate science), this is a reasonable decision.

The general procedure for handling I/O with netCDF is quite similar to the memory handling in the previous section. Again, relevant netCDF functions are searched for and replaced with wrapper functions. Unlike the memory handling component, the I/O component inserts many functions that are hidden behind environment variable conditions. They are inserted at compile time, but the user can control which functions are executed at runtime by setting the appropriate environment variables. Only optional functions such as compression are shielded in this way; if the user does not set an environment variable, the application will still work.

CATO assumes that only significant variables are worth handling with netCDF, so there is no preselection routine to potentially sort out I/O kernels. The replacement is done in the following domains:

- File initialisation and closing
- Variable definition
- Data access

The general intentions of the replacement steps have already been discussed in Section 4.2, in the following Sections 5.3.1 and 5.3.2 the actual transformations are discussed.

5.3.1. NetCDF: Parallel Input/Output

Figure 4.8 on page 101 shows the goal to be achieved with CATO. The memory handling component has already made it possible to split up heap variables that would otherwise not fit into the memory of a single node. Before the computation can begin, the data needs to be initialised somehow. If this is done dynamically without any input,

this is not a problem, as each process can do this on its own. However, if the initialisation is done by reading data from storage, it is best to use parallel I/O. Each process can then get its own share and does not have to read the whole oversized file first.

Listing 5.6b shows an idealisation of what the replacement might look like in pseudocode. Using parallel I/O in netCDF requires adapting the file initialisation and any operations that read or write directly to the file in memory. CATO looks for initialisation calls for serial I/O, namely `nc_open` and `nc_create`, and replaces them with appropriate wrapper calls (cf. line 2). Inside these wrappers are, among other things, netCDF calls to initialise parallel I/O: `nc_open_par` and `nc_create_par`. The same goes for the file close operation, `nc_close` is searched for and replaced by a wrapper function.

There are two possible parallel access modes that can be selected for a file: *independent* or *collective*. The netCDF API suggests using the collective mode whenever possible (Unidata, 2023a). However, which mode works best depends on the communication pattern used. In general, the independent mode is preferable if few synchronisations are required (e.g. in an 1-to-1 client-OST pattern or if data is only read). The collective mode, on the other hand, might be preferable if more synchronisations are required (e.g. in an all-to-all client-OST pattern or if data is being written) (Bartz et al., 2015). CATO allows to change the used mode by inserting `nc_var_par_access` (cf. line 5), which can then be configured by the user at runtime. The collective mode is used by default, but the user can explicitly set the mode by setting the environment variable `CATO_NC_PAR_MODE` to `COLLECTIVE` or `INDEPENDENT`.

Now the data itself can be accessed. Accessing data via the netCDF functions requires passing a pointer to the memory area which is used to mirror the data from storage to memory and vice versa. CATO’s I/O component utilises the benefits from the memory component if the used memory section is allocated on the heap (which is most probably the case for relevant input data). All local memory accesses are also handled by the CATO memory component, but all file accesses using netCDF need to be modified as well. Each process only accesses its share after the modification. So instead of reading the whole variable, the corresponding *array* variant of the original I/O operation is used (cf. line 6).

After this step, each process can initialise its input or save its results to disk on its own. This can now be used to not only enable parallel I/O, but also in combination with a parallel FS to improve the runtime of the I/O phases. Figure 4.9 showed how an application could benefit from spreading the accesses of all processes across multiple OSSs to benefit from the aggregated bandwidth of the network and disks. To take advantage of this, the file being accessed needs to be striped across multiple OSSs so that all process accesses are evenly distributed across multiple OSSs. This is not done by CATO, as it requires detailed knowledge of the application’s use case and the parallel FS environment available. However, if the user takes care of the correct setup, then CATO’s modification will allow the application to benefit from this.

5.3.2. NetCDF: Compression

The use of compression is enabled in netCDF by calling the appropriate functions during the definition phase of a variable. NetCDF compression does not affect the metadata and only works on the actual data section. To enable compression, two netCDF functions need to be considered and set appropriately, which have been introduced in Section 3.3.2:

Chunking: If compression or any other HDF5 filter is to be used, chunking must be used. Since netCDF 4.7.4, using HDF5 1.10.2 or later, it is also possible to use filters and parallel I/O at the same time (Unidata, 2020). In this case, the use of collective mode for parallel access is mandatory. The maximum size of a (multidimensional) chunk is limited and must not exceed 4 GiB (Unidata, 2023b).

The expected runtime performance of accesses to chunked data is lower than that of accesses to contiguous data, but since the compression is applied to individual chunks and not to the whole piece of data, this must be accepted.

Alignment: If data chunks are not aligned to the underlying disk block size or stripe size of a parallel FS, runtime performance is much worse (Bartz et al., 2015). Since netCDF 4.9.0, the API provides a function to enforce the alignment of any variable. Using alignment can potentially leave holes between file objects, so there is a sweet spot between access speed and file size (Unidata, 2023a). A threshold can be set to ignore small files that would not benefit from this alignment. By default CATO uses the recommended threshold of 8192 B (Unidata, n.d.-b).

The current focus of CATO is on the *quantize* filter for lossy preprocessing, the lossless compressor Zlib, and the *filter* interface, which allows arbitrary use of any filter that is part of HDF5 and installed on the system. NetCDF also provides calls to the Blosc, bzip2, Szip, Zstandard compressors, but these have been omitted for now, as they are just wrapping calls to `nc_def_var_filter`. It is up to the user to decide if and which filter to use, as this can have a significant impact on the runtime performance of the application. Therefore, CATO inserts all the necessary netCDF functions, but hides chunking, alignment and filter usage behind conditional checks if the appropriate environment variables have been set. The modified application is equipped with the necessary functions, and the user can decide at runtime which ones to use with which configuration.

A demonstration of how CATO does this is shown in Listing 5.7 using the example of the Zlib compressor. In line 2 the environment variable `CATO_NC_CMPR_DEFLATE` is parsed, which contains at most two values (cf. line 3). A small sanity check in line 6 ensures that only supported compression levels are used. Finally, if the shuffle parameter is set (which reorders data bytes to improve compression results) (cf. line 10), the netCDF call to use the deflate compressor is inserted in line 12.

To demonstrate the effect on the original source code, the example from Listing 5.6 is extended by adding an arbitrary filter. The modified pseudocode is shown in Figure 5.8b. The alignment and chunking are set in line 2 and line 5. If the user selects a valid filter (cf. line 6) and enables quantisation, it will be enabled in line 8.

```
1  if (std::getenv("CATO_NC_CMPR_DEFLATE")) {  
2      deflate_values = parse_env_list("CATO_NC_CMPR_DEFLATE");  
3      if(deflate_values.size() > 2)  
4          // [ Error handling ]  
5      else {  
6          deflate_level = std::stoi(deflate_values.at(0));  
7          if(deflate_level < 0  deflate_level > 9)  
8              // [ Error handling ]  
9          else {  
10             if(deflate_values.size() == 2)  
11                 shuffle = std::stoi(deflate_values.at(1));  
12             err = nc_def_var_deflate(ncid, varid, shuffle, 1, deflate_level);  
13         }  
14     }  
15 }
```

Listing 5.7.: Truncated code snippet for handling netCDF compression using Zlib.

CATO takes care of setting up the mandatory functionality, the user has to explicitly choose parameters to suit his use case. Since there are many combinations available, it can be a non-trivial task to find the best one. However, he can also use `nccopy` to try them out until he finds a satisfactory combination, which he can then pass to the modified application via environment variables. In addition, CATO offers also a script, to automatically try out different configurations so that the user can then use the best fitting configuration (cf. Section 7.3.3).

5.3.3. Semantical Equivalence

Most netCDF modifications are not a problem, because they don't change the semantical behaviour. Changes to the file initialisation and closing calls have no additional visible effect, only the internal handling within netCDF changes. The same applies to changes in the definition of a variable (e.g. alignment and chunking). They affect runtime performance and storage requirements, but not the data integrity itself. Applying compression filters does not affect data integrity as long as a lossless compressor is chosen. Data is potentially altered when a lossy compressor is used, but since the user must explicitly enable lossy compression, he is aware of this.

A critical point is the change in access operations. They are replaced by different access operations working on a different part of the data. If there were errors in the replacement scheme, this would result in incorrect data. CATO's testing infrastructure can reduce the risk of errors, but it is impossible to guarantee correctness.

<pre> 1 // [...] 2 3 nc_create(...); 4 nc_def_var(...); 5 6 7 8 9 10 nc_enddef(); 11 12 nc_put_var_float(...); 13 nc_close(...); 14 // [...] </pre>	<pre> 1 // [...] 2 err = nc_set_alignment(...); 3 err = nc_create_par(...); 4 err = nc_def_var(...); 5 err = nc_def_var_chunking(...); 6 if(nc_inq_filter_avail(...)) { 7 err = nc_def_var_quantize(...); 8 err = nc_def_var_filter(...); 9 } 10 err = nc_enddef(); 11 err = nc_var_par_access(...); 12 err = nc_put_vara_float(...); 13 err = nc_close(...); 14 // [...] </pre>
--	---

(a) Original pseudocode.

(b) Modified pseudocode using parallel I/O and filters. Error value checks are omitted for brevity.

Listing 5.8.: Comparison of how a netCDF code is roughly modified by CATO, focusing on parallel I/O and compression.

5.4. Providing Feedback

Rather than simply using a black box, the feedback component can allow the user to understand and try out the change for themselves. The previous Section 4.3 defined four questions about how CATO can help the user understand the changes:

- How can CATO give the user an idea of what the modified high-level code might look like?
- How can CATO help the user to avoid known pitfalls?
- How can CATO help the user to gather performance metrics?
- How can CATO help the user to understand how to implement the HPC frameworks used by CATO?

They are answered in the following four Sections 5.4.1, 5.4.2 and 5.4.4.

5.4.1. Trace Code Changes

Preliminary evaluations of decompiling the modified IR or binary have shown that the target user is unlikely to benefit from them due to their high complexity. Therefore, this approach was not pursued further.

5.4.2. Hardening Code Quality

To avoid the common pitfalls mentioned in the second question, CATO can apply sanity checks to the original source code (cf. Section 4.3.2). Three different types of sanity checks have been included so far:

1. Error checking
2. File existence
3. Closing file

Each netCDF function returns an integer error code that can be checked. If the user does not check these errors, it is easy to overlook them until a subsequent, more severe error occurs in a later section of code. In the worst case, the error goes unnoticed and the application suffers from undefined behaviour. This can make debugging quite frustrating for the user. CATO catches the return value of all replaced netCDF functions and checks them. It would also be possible to check for each original netCDF function call if the return value is somehow retrieved and checked. If not already checked, CATO could also add error checking for unmodified netCDF calls. However, this has not yet been implemented.

The second sanity check is applied to each replaced `nc_open` function. If the file to be opened does not exist, the function returns an appropriate error code, but otherwise simply continues execution. Reading data from a non-existent file results in junk data and may go unnoticed if the user does not check. Replacing `nc_open` with `nc_open_par` adds a sanity check to verify the existence of the file.

The third sanity check can ensure that any file that is opened or created will be closed at some point. This sanity check simply scans the code for a call to `nc_close` with the correct `ncid`. It does not perform an analysis of the CFG to check if this function call appears on every possible execution path.

If a sanity check registers an error, a warning is printed by default, but the user can also set an environment variable to force abort or disable error checking.

5.4.3. Metric Collection

To assist the user in evaluating the performance of the modified application, CATO offers the possibility to collect the runtime and peak memory performance.

The C library offers two functions, which can be used to achieve this: `gettimeofday` (time) and `getrusage` (peak memory consumption). Both functions collect their metrics at the time of call within structs. Since the user is interested in the metrics over the whole applications runtime, they are added at the earliest and latest possible moment. Section 5.2 already describes that CATO inserts initialisation and finalisation functions immediately at the entry and all exit BBs and this is where the metric function calls are inserted as well.

At the exit BBs the difference of timestamp and peak memory consumption structs are calculated. The metrics are collected for each process independently, therefore rank

0 first collects the sum of all values via an MPI reduction and then divides the result by the number of participating processes to derive an average value, which is then printed. Results for the peak memory consumption are not determined byte-exactly from OS, but can fluctuate slightly (Siebenmann, 2012). This can be neglected if at least a few kilobytes are used.

To avoid unnecessary overhead, which could be induced for example by the reduction, the metric functions are encapsulated behind a conditional check, if the user had the appropriate environment variables set to enable the metric collection. To activate enable the collection of both metrics, the user can for example set `CATO_METRICS=TIME:MEM`, but `TIME` and `MEM` can also be set solely.

5.4.4. Textual User Support

Section 1.2 set the baseline for CATO: To provide an easy-to-use tool for domain scientists who have to cope with the sheer number of available HPC tools and frameworks if they want to make full use of the available hardware. CATO supports them by providing a framework with a diverse set of tools, which can be further extended with additional HPC libraries. This allows them to try them out without having to learn them the hard way. If the user likes the changes made to his source code, CATO can help him implement the changes himself.

For each library included, CATO provides a quick reference guide with a general description and a link to the official documentation. More importantly, it also provides a section with general usage tips and references to existing code examples. Especially the last point can be very helpful, as the user can simply look at how the library can be used. A library with good documentation will also provide code examples, but very often these are trivial and quite short. Sometimes there are unofficial repositories of tutorials and code skeletons that CATO can point to.

The help overview also lists all the environment variables supported by CATO. Using them allows the user to control which components of the replaced code are executed at runtime.

5.5. Summary

After defining the general requirements of CATO in Chapter 4, this chapter deals with its implementation. LLVM is a large framework with many possible starting points. In the end, it was decided to build the core of CATO as LLVM pass to be introduced into the Pass Manager's pipeline. The changes are made at the IR level, which has the great advantage that they are decoupled from a specific frontend (with reservations) and from a specific backend. By using a `ModulePass` as a base class, the modifications can be applied to the entire translation unit without losing contextual information. To implement the memory handling component and the I/O handling component, replacement codes are prepackaged as ECs, which perform the modification at compile time and are then

executed at runtime. Setting environment variables allows the user to influence the behaviour of the modified application at runtime to suit their needs.

6. Related Work

Leaving aside the feedback component of CATO, the development focused on three main themes:

- Memory handling
- Parallel I/O
- Compression

There are many solutions that address the problem of memory allocation, and a few that also address parallel I/O and compression. This chapter provides an overview of alternatives to CATO that offer a similar feature set, but differ in their ultimate purpose. They cover a wide range of usability concepts, from fully transparent mechanisms to programming languages and libraries that the user has to consciously use. This chapter follows this idea and discusses them in order of increasing need for active user interaction.

6.1. Virtual Shared Memory

Virtual Shared Memory describes the concept of a transparent layer within the memory access workflow, in order to hide from the user where the memory device is actually located. This is already done anyway, as the user does not have to worry about which actual bank they want to access, the memory address is already a transparent layer. However, virtual shared memory takes this principle one step further by hiding the actual node where the data is stored. It aggregates distributed memory and provides a whole new virtual address space, the only limit being the aggregated memory of all participating nodes and the upper limit of possible memory addresses. The actual mapping of virtual memory addresses to the corresponding node bank is done transparently by a runtime environment or library. Depending on the actual solution, and ignoring usability and performance limitations, this concept is very advantageous because the user does not have to worry about hardware configuration and can act as if he were using a single, very powerful node.

6.1.1. Single System Image

Virtual shared memory comes in several variations. *Single System Image* (SSI) is the one that probably requires the least user interaction, as it provides a unified system view of all included nodes. Strictly speaking, the concept of SSI is not limited to memory,

but can include other components such as process, device or file space. From this point of view, a parallel FS like Lustre is also a kind of SSI, as it hides the complex storage handling behind the Lustre client. It allows to potentially improve I/O performance if the application uses parallel I/O by removing the bottleneck represented by a single block device.

The user's application can be run inside a SSI without changing the source code. Healy et al. (2016) have done a survey on SSIs and listed examples for different system levels on which a SSI can be used. There are hardware implementations such as the Stanford Dash multiprocessor (Lenoski et al., 1992) and software implementations such as a OS, e.g. *Popcorn Linux*, *openMosix* or *OpenSSI* (Barbalace et al., 2014; Lottiaux et al., 2005). Healy et al. show that there was a peak of interest in this concept in the 2000s, but there were also early critical voices (Buyya et al., 2001). Some drawbacks include limited scalability and the need to set up the entire HPC cluster using a SSI image. There were HPC systems like the now decommissioned *HLRBII* at *Leibniz-Rechenzentrum* or the *Cenju-4* from NEC that used SSI for memory, but these systems are in the minority (GWDG, n.d.; Kusano et al., 2001). And since the user cannot install a SSI without root privileges, he cannot rely on its existence.

6.1.2. Partitioned Global Address Space

Partitioned Global Address Space (PGAS) is a parallel programming concept that uses a software layer to emulate a virtual shared memory space. Each node has allocated its share of the total memory space so that the actual node holding the data to be accessed can be computed by the PGAS system. Locality and latency can become a serious problem, as the user is no longer forced to think about where to put his data, which makes it easier to use, but can add stress due to data movement and measurements to keep the cache coherent on the system (Sterling et al., 2017, Ch. 21.5.3). On the other hand, the PGAS system can optimise data movement that might otherwise be beyond the user's capabilities. Two popular PGAS libraries respectively standards are GASPI and OpenSHMEM. Regarding the performance there is no definite statement if PGAS performs better or worse than MPI. It depends on the use case and which implementation has been used. H. Shan et al. (2012) (which do not use one-sided but two-sided MPI operations for comparison) show that PGAS outperforms MPI but on the other hand Hammond et al. (2014) and Si et al. (2021) use MPI to implement OpenSHMEM.

PGAS is not considered in this thesis because it already provides a way to use the shared memory of all participating nodes. And if the user's application does not already use a PGAS library, he would have to rewrite his entire application.

6.1.3. Distribution of OpenMP

The focus of this thesis is on OpenMP, and there are some approaches that are capable of executing OpenMP code on distributed memory. Shortly after the initial release of the OpenMP specification in 1997, an OpenMP compiler was proposed by Sato et al. (1999). The Omni OpenMP compiler does not actually distribute, but relies on an existing

software SSI such as TreadMarks or the built-in XcalableMP extension (Amza et al., 1996). There is also an extension to the Omni compiler to create a PGAS and simplify the handling of multiple GPUs by virtually merging their memory (Nakao et al., 2019). As mentioned above, data locality and cache coherency can be a major drawback of a SSI, and the Omni compiler aims to optimise an OpenMP application on such a system.

Based on TreadMarks, Intel developed *Cluster OpenMP*, an OpenMP implementation for using distributed memory (Hoefflinger, 2006). Case studies showed that it was at least difficult to use without sacrificing scalability, and the project has since been abandoned (Terboven et al., 2008; Wong et al., 2008).

Unlike the XcalableMP extension, CATO does not need mandatory compiler directives, but derives the necessary information from the existing OpenMP directives. Therefore, CATO knows which variables to distribute, and the focus on the stencil pattern already provides an optimised distribution strategy. Of course, the optimised distribution strategy could perform worse if another communication pattern is used, but this is beyond the scope of this thesis and will be discussed in Section 8.2.3.

Other approaches target the `task` directives, which define a kernel of code that can be executed independently. Defining multiple tasks allows the OpenMP runtime to execute the tasks concurrently. *OpenMP Cluster Programming Model* (OMPC) creates a new OpenMP runtime that uses the task definition and adopts the OpenMP directive `pragma omp target`, which is used to offload code to local GPUs. The customisation adds the ability to specify another node as a target, to which a task and its dependent data is offloaded and then executed on (Yviquel et al., 2022). A similar approach is taken by *OmpSs-2@Cluster*, which is an extension of the OmpSs-2 programming model (Mena et al., 2022). OmpSs-2 is an alternative to OpenMP, but uses a very similar set of directives and clauses, and can be used to develop and benchmark new features for OpenMP. This extension takes a similar approach to OMPC and derives a dependency graph from task directives. A virtual address space is set up and all participating nodes contribute their share to this space. The OmpSs-2 runtime then manages the distributed execution of the application, including the necessary data movement.

A fairly new approach is the remote offloading implementation of Patel and Doerfert (2022). It works like the usual offloading component that could be used to run code kernels on local GPUs, but has been extended to allow the use of devices on remote nodes. The authors claim that, at least for now, they are not aiming to compete with MPI or other distributed programming models. Currently, this approach is gaining a lot of traction and a lot of work is being done on it (Lu et al., 2022; B. Shan et al., 2023).

This approach requires the use of additional mandatory compiler directives that are not needed for CATO. However, as this is still fairly new, it is worth keeping an eye on it so that it can be re-evaluated in due course.

Recent developments in OpenMP distribution are quite promising and could become a serious approach to replacing, or at least complementing, MPI. However, it is important to remember that the tasking and offloading capabilities of OpenMP are at least intermediate, if not advanced, features of OpenMP. The sighting of used OpenMP features on GitHub in Section 3.2.1 showed that the majority of user-generated code does not use them yet. So they cannot benefit from the remote offloading features either.

6.2. Direct Code Parallelisation

There are several approaches to parallelising (serial) code in a more explicit way than those described in the previous section. The approaches differ in the amount of work that needs to be done directly by the user. Starting with the most involved and invasive, this section shows different approaches and lists them in order of decreasing complexity and increasing automatic support.

6.2.1. Parallel Programming Languages

There are many programming languages, each with its own merits and focus. Some stand out for their ability to express parallelism. This is not an exhaustive list, but some notable ones include

- Chapel (Chamberlain et al., 2007)
- Cilk (Frigo et al., 1998) (C++ extension)
- Fortress (Jr., 2006)
- X10 (Charles et al., 2005)

Their purpose is usually to meet the needs of numerical applications. For example, Fortress provides a notation that is very close to the actual mathematics to make it easier for the domain scientist to express himself. Chapel, on the other hand, provides a high level of abstraction by using anonymous threads, which makes it much easier to parallelise an application and achieve high performance. It allows to target different platforms such as accelerators or FPGAs.

6.2.2. Parallel Libraries

Using a library is less invasive than learning a new programming language (assuming the library supports the user’s usual programming language). You can use specialised parallel data structures such as Python Dask (Dask core developers, 2022) or TBB (Intel, n.d.), provide functions for message passing such as MPI or functions for parallel I/O such as MPI-IO and netCDF. Others like *Charm++* (University of Illinois, 2021) or Kokkos (Trott et al., 2022), focus on performance portability by abstracting the device (e.g. CPU, GPU, FPGA, ...) on which code kernels run, so that they can be easily moved between HPC systems with very different hardware configurations. Still others expose the functionality of a PGAS environment, e.g. OpenSHMEM (Silicon Graphics International Corp., 2022), GASPI (H. Shan et al., 2012) or Unified Parallel C (Schmidt et al., 2017, Ch. 10).

Listing 6.1 shows a code snippet to demonstrate what OpenSHMEM code can look like. There are clear similarities between the OpenSHMEM functions used and MPI. Line 3 initialises the library and is the counterpart to `MPI_Init`. The so-called *Processing*

```
1 static long source[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
2 long target[10];
3 start_pes(0);
4 if (_my_pe() == 1) {
5     /* get 10 words into target from PE 0 */
6     shmem_long_get(target, source, 10, 0);
7 }
8 if (_my_pe() == 1) {
9     for(i=0;i<10;i++)
10     printf("target[0] on PE %d is %d\n", _my_pe(), target[0]);
11 }
```

Listing 6.1.: Code snippet using OpenSHMEM (Pophale & Curtis, n.d.).

Element (PE) is given an ID in line 4, in MPI the process would get its rank from `MPI_Comm_rank`. And finally the PE gets an element via one-sided communication in line 6, which in MPI would have been done with `MPI_Get`. There are other similarities such as synchronisation and collective operations. Unlike MPI, OpenSHMEM focuses only on one-sided communication and makes global and static variables automatically accessible from remote. Local variables can also be made remotely accessible, but this must be done via an explicit operation call. The concept of MPI epochs echoes in OpenSHMEM, but in OpenSHMEM the concept is more relaxed and does not ensure the completion of operations, only the order.

6.2.3. Domain Specific Languages

Domain Specific Languages (DSLs) are not necessarily easier to use than the parallel libraries mentioned above, since both categories have trivial and complex examples. The way they are implemented can also vary considerably: some are simply C/C++ or Fortran libraries, others are a whole language with its own compiler. PATUS, for example, uses a C-like syntax and adds special keywords that act like OpenMP directives and clauses (Christen et al., 2011). It has been optimised to work on stencil codes on structured grids, offering a number of strategies that differ in how the data and computations are mapped to the hardware. It is quite limited in what the data layout can look like and requires the structured grid to be represented as a struct of arrays. Its code generation component uses OpenMP for parallelisation and CUDA to address GPUs. Listing 6.2 shows a short code snippet using PATUS which performs two domain decompositions (the subdomains are then processed in parallel) and then applies the stencil operation in line 3.

On the other hand, Green-Marl is a more descriptive language that ends up being compiled into equivalent C++ code. It specialises in parallelising graph analysis algorithms (Hong et al., 2012). Listing 6.3 shows an example of code to traverse a graph.

```
1 for subgrid v(blk1) in u[:,t] parallel
2   for subgrid w(blk2) in v[:,t] parallel
3     for point p in w[:,t]
4       ...
```

Listing 6.2.: Example of PATUS to apply a stencil (Christen et al., 2011).

```
1 Int sum=0;
2 Foreach(s: G.Nodes) {
3   Int p_sum = u.A;
4   Foreach(t: s.Nbrs)
5     p_sum *= t.B;
6   sum += p_sum;
7 }
8 Int y = sum / 2;
```

Listing 6.3.: Example of Green-Marl to traverse a graph (Hong et al., 2012).

In lines 2 and 4, the application iterates over all nodes and all their neighbours in parallel, using a fork-join approach to sum one of their attributes.

OP2 and SPar are two DSLs that are preferably used to perform computations on unstructured or structured grids (Christen et al., 2011; Mudalige et al., 2012).

The examples have clearly demonstrated the benefits of using DSLs: The programming focus is not so much on constructing the necessary data structures or setting up the parallelisation environment, but rather on expressing the problem itself. Since a DSL knows its domain, it can then convert the expressed problem into actual code, taking advantage of features such as compression, which can then be compiled into the final binary.

6.2.4. Tool-Assisted Parallelisation

The last category is probably the easiest to use. This does not necessarily mean that these solutions are less powerful than the previous ones. In some cases, they have a fairly narrow use case, such as OpenACC, which focuses on offloading code kernels to GPUs. Others like OpenMP mainly provide compiler directives that are transformed during the compilation process. More explicit solutions such as pThreads may be able to achieve better runtime performance, but are more difficult to use. And if difficult usability is a deterrent to the majority of domain scientists, then a simpler solution, which may result in lower potential performance, is preferable. OpenMP’s feature set has been growing for over 25 years, and several examples throughout this thesis have also shown that it has become quite capable.

Since OpenMP and MPI have become a quasi-standard in HPC, there are a number of tools that can be used to automatically generate code that includes OpenMP or MPI. Polly analyses the IR of a source and uses polyhedral techniques (analysis focusing on time dependencies and potential concurrency) to optimise it by parallelising loops with inserted OpenMP directives (Grosser et al., 2012). Further development led to Molly, which distributes computations in addition to Polly (Kruse, 2014). Its runtime uses MPI for data exchange, but it is intended more as a prototype than for productive use. It also requires that the loop boundaries can be derived statically. If they are based on a dynamic user, for example, Molly cannot adjust the loop. On the other hand, CATO can handle dynamic loop boundaries.

Two other approaches, like OpenMP, provide a language extension to C, but with the ability to use distributed memory: XcalableMP and OpenMPD (J. Lee et al., 2007; XcalableMP Specification Working Group, 2018). The user must explicitly cite the data and work sharing using these directives. OpenMPD is based on the Omni OpenMP compiler, which translates the directives into MPI code. XcalableMP has been integrated into the Omni compiler and uses special directives which can then be transformed using modern MPI features such as one-sided communication.

Hamidouche et al. (2011) developed a two-stage automatic framework for creating hybrid MPI-OpenMP codes. The first phase is an analysis component to predict the potential runtime of each function within the original code based on the size of the input data. The user must then provide an XML file that roughly describes which functions should be parallelised and when communication or synchronisation should take place. The code generator then uses a *Bulk synchronous parallel* model with actual node and core counts to estimate the best load distribution and generates the modified code.

Arora et al. (2014) have built an interactive tool, *Interactive Parallelization Tool* (IPT), with a *Graphical User Interface* (GUI) so that a user can manually select code kernels to parallelise. The user can decide whether the parallelisation should be done using OpenMP, MPI or CUDA (a hybrid version is also possible). In a later extension, support for MPI-IO was also added, so that the user can also instruct the tool to automatically add parallel I/O to the code (Arora & Ba, 2019). It uses the ROSE compiler as a backend to perform the code transformations. User interaction is not optional in this tool, the user must provide the necessary information. And the I/O component is quite limited, as it requires a specific pattern and delimiters.

Schneider et al. (2023) created *MPI-rical*, a tool for automatically inserting MPI operations into serial code. *MPI-rical* uses machine learning to identify equivalent MPI substitution code and a serial code. It has been trained on over 25000 serial code snippets and 50000 MPI code snippets scraped from GitHub. It is already showing promising results, but is still in an early stage of development, and as machine learning is based on probabilities, there is no guarantee that the replacement code will be semantically equivalent. So additional testing and manual user checks are probably advisable.

There are some tools that focus on code with existing OpenMP directives and translate it to equivalent code using MPI instead. Saà-Garriga et al. (2015) has created a S2S compiler using Mercurium as a code generation backend. The code analysis is triggered by adding a new `target` clause and the transformation is then performed on the AST

layer. A similar approach by Basumallik and Eigenmann (2005) uses Cetus as the code transformation backend. Shared data is assigned to all processes, and updates are sent based on a producer-consumer flow that tries to estimate potential consumers. Millot et al. (2008) developed a solution for Fortran code, focusing on `OMP SECTION` and `OMP PARALLEL DO`. All of these solutions focus on two-sided MPI communication which could lead to higher latency.

There are also solutions for other languages such as Python and Java. They usually analyse read and write operations in the source code to construct a dependency graph of the data. From this graph, tasks are derived that describe a self-contained workflow of operations. Fonseca et al. (2016) provides a runtime to create these tasks and schedule their execution. Shirako et al. (2022) separates the focused tasks from the rest of the code and uses polyhedral techniques similar to Polly to optimise and then distribute them.

There is a large and diverse field of concepts for automatically distributing the computation phase of an application. Many of them require the user to provide additional information about what part of the code is of interest and how it should be handled. This is a feasible approach for computer scientists, but could be overwhelming for domain scientists who are not trained to use such tools. Fully automated concepts such as SSI help to reduce this barrier, but are rarely found in the HPC domain. Other tools that do not require the user to modify their code are limited to a specific use case or do not care about optimising usable memory space.

6.3. Transparent Compression

Automated and transparent compression, where the user doesn't have to do anything, is currently limited in its implementation. Looking at different levels, from the application close to the user to the system further away from the user, there are layers such as hardware, system (low-level, operating system, file system), middleware and application.

In the context of HPC, there are compression options at the hardware level, such as sensor-based compression, which filters data in physical experiments (Hung et al., 2013). Closer to the HPC system, there are hardware components and technologies such as QAT that provide efficient compression tools that can be used by lower system layers (Hu et al., 2019).

At the FS level, there are several local FSs with limited adoption in the HPC domain. The most prominent FSs in HPC are Spectrum Scale and Lustre, both of which support server-side compression (Lustre only with a ZFS backend) (IBM, 2021; Kuhn et al., 2016). There are ongoing developments for client-side compression (Blagodarenko & Faber, 2023). These layers are virtually invisible to the user, require no action, and are not consciously chosen. If the user is using such a system, they will automatically use its features. However, these features are not widely established.

Middleware provides more or less automated compression through the use of data formats. Many data formats such as FITS (Flexible Image Transport System), CDF (Common Data Format), DICOM (Digital Imaging and Communications in Medicine),

HDF5 and netCDF provide compression functionality. In rare cases, such as when data is transferred to the GRIB format, even lossy compression may occur without user intervention (Iza-Teran & Lorentz, 2005). In most cases, the application developer needs to explicitly enable compression (Delaunay et al., 2019). This can be more or less complicated and acts as a threshold for conscious use (Knox & Pourmal, 2018; Pourmal & Knox, 2017). At this point, the user has control over, but also responsibility for, the conscious use of the feature.

In the case of netCDF and its HDF5 backend, compression is applied opaquely, so the user must at least insert specific calls during the definition of a variable to enable compression. If this step is not taken, no compression will be applied. CATO intervenes in this step and inserts the necessary compression calls. To the best of my knowledge, there is no related work that automatically transforms source code to enable the compression features of a used library.

6.4. Summary

In this section several different approaches have been presented for running computations on distributed memory. One promising approach was the concept of virtual shared memory, as this makes the underlying distributed memory hardware transparent to the user. In particular, the SSI concept means that the application does not need to be modified at all. It is simply executed on an SSI and the OS takes care of handling the application’s memory access requests. SSI had its heyday in the 2000s, and at least within the software layer it is hardly used within HPC.

PGAS describes an abstraction of memory space within the software layer. Following this concept, for example, is the OpenSHMEM standard, which offers a similar feature set as one-sided MPI.

In addition, several approaches have been presented that aim to achieve distributed execution of OpenMP applications. Although there are some promising concepts, they are no longer supported (e.g. TreadMarks) or require code changes (e.g. the Omni OpenMP compiler and OpenMP’s remote offloading capabilities). Although the code changes would probably be unambiguous, at least they belong to the intermediate feature set, which could be a problem for domain scientists.

The alternative is not to rely on virtual shared memory, but to make the memory allocation more explicit. And although there are many different solutions, such as parallel libraries and DSLs, they probably require even more code changes in this case. A real alternative might be tools that do the necessary code transformations automatically. And while there are a few that can actually transform OpenMP code into equivalent MPI, this is only the first step of CATO, which also supports parallel I/O and compression based on netCDF.

The only tool from , which could at least support automatic insertion of MPI and MPI-IO for parallel I/O into a high-level source was IPT. However, it requires interactive input from the user and is quite limited, at least within the I/O component (only one element can be written at a time and a delimiter is required).

7. Evaluation

Now that the design of CATO is complete, a prototype has been built and will now be tested. The discussion in the introduction (cf. Chapter 1) led to the decision to implement three components in CATO, which will be evaluated in this chapter. The main components are *memory distribution* (Section 7.3.1) and *parallel IO and compression* (Section 7.3.3). The feedback component is also considered in Section 7.3.4, although it is less relevant in terms of performance. Secondary side effects on the binary caused by CATO will be discussed in Section 7.3.5 at the end.

At first, an appropriate testing strategy is set up in Section 7.1 and the test environment is defined in Section 7.2.

7.1. Speedup Limitations

This section runs and evaluates a number of micro-benchmarks, focusing on OpenMP and netCDF. The use of automatic code transformation will have an impact on the resulting performance of the modified binaries as new frameworks are used and new execution configurations become possible. The question then emerges as to what the measurement strategy should be to assess whether or not the initial objectives of the Section 1.2 are being met. To clarify this, a general performance model is used to get a feel for the potential outcome.

The general question is how well the modified application benefits from an HPC system, given an unlimited amount of resources. Using speedup or speedup efficiency gives a single result that can be used to assess the performance benefit of an application using more COUs. Whether a COU is a software unit (e.g. thread or process) or a hardware unit (e.g. core or CPU) depends on the context. For this theoretical investigation, the true nature of the COUs is irrelevant. The following definitions follow Sun and Ni (1993). The intermediate steps are therefore omitted, and only the relevant results are used in this section.

Speedup describes the scaling behaviour of an application as the number of COUs increases. There are different views on this term, each with a different emphasis: Fixed-size speedup and fixed-time speedup. Their effect on the time and workload metrics varies, Figure 7.1 shows an abstract representation. The following equations will confirm this.

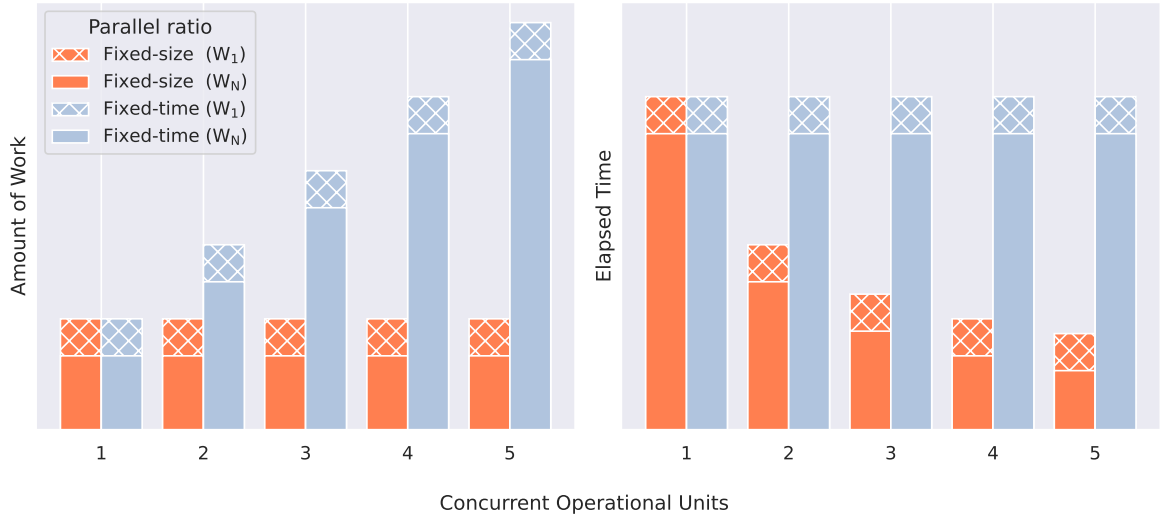


Figure 7.1.: Comparing the effects of focusing on fixed-size or fixed-time speedup on the handling of time and workload (based on Sun and Ni, 1993).

7.1.1. Fixed-Size Speedup

For fixed-size speedup, the workload remains the same while the runtime varies. The focus is on the ability to reduce execution time. Equation (7.1) shows the baseline.

$$S_N(W) = \frac{T_1(W)}{T_N(W)} \quad (7.1)$$

With:

W	Workload
$T_i(W)$	Execution time to complete workload W using i many COUs
$S_N(W)$	Speedup on workload W using N many COUs

Sun and Ni are now considering some restrictions:

- The workload cannot be divided into arbitrarily small chunks. If the limit is m , then $W = \sum_{i=1}^m W_i$.
- Once there are at least two COUs involved, they must interact with each other. Whether it is using locks on shared memory or sending messages over the interconnect, the interaction introduces latency. This latency is accounted for by the notion of $Q_N(W)$.

Following these constraints, Equation (7.1) is rewritten:

$$S_N(W) = \frac{\sum_{i=1}^m W_i}{\left(\sum_{i=1}^m \frac{W_i}{i} \lceil \frac{i}{N} \rceil\right) + Q_N(W)} \quad (7.2)$$

With:

m	Maximum parallelism degree of W
$Q_N(W)$	Latency induced by using N COUs on workload W

If latency is neglected and it is assumed that there are only two workload classes W_1 and W_N with $W_1 + W_N = 1$, Equation (7.2) can be further simplified:

$$S_N(W) = \frac{W_1 + W_N}{W_1 + \frac{W_N}{N}} \quad (7.3)$$

$$= \frac{1}{W_1 + \frac{W_N}{N}} \quad (7.4)$$

$$\lim_{N \rightarrow \infty} S_{N,\max}(W) \leq \frac{1}{W_1} \quad (7.5)$$

So this simplification leads to Equation (7.4) or its estimate in Equation (7.5) for the case of infinitely many COUs, which is *Amdahl's law* (Amdahl, 1967) and allows an estimate of the maximum potential fixed-size speedup.

7.1.2. Fixed-Time Speedup

In fixed-time speedup, the execution time is fixed and the workload size varies. In this case, the focus is on the ability to scale up a problem and still solve it within the fixed time interval. W' is again the workload, but this time it is scaled proportionally to the number of COUs. The first term looks like Equation (7.1) using now dynamic workload sizes. Since the time is fixed, the condition $T_1(W) = T_N(W')$ is true (cf. Figure 7.1).

$$S_N(W') = \frac{T_1(W')}{T_N(W')} \quad (7.6)$$

$$= \frac{\sum_{i=1}^{m'} W'_i}{\left(\sum_{i=1}^{m'} \frac{W'_i}{i} \lceil \frac{i}{N} \rceil\right) + Q_N(W')} \quad (7.7)$$

Using the same simplification as in the treatment of fixed-size speedup, Equation (7.7) can be simplified:

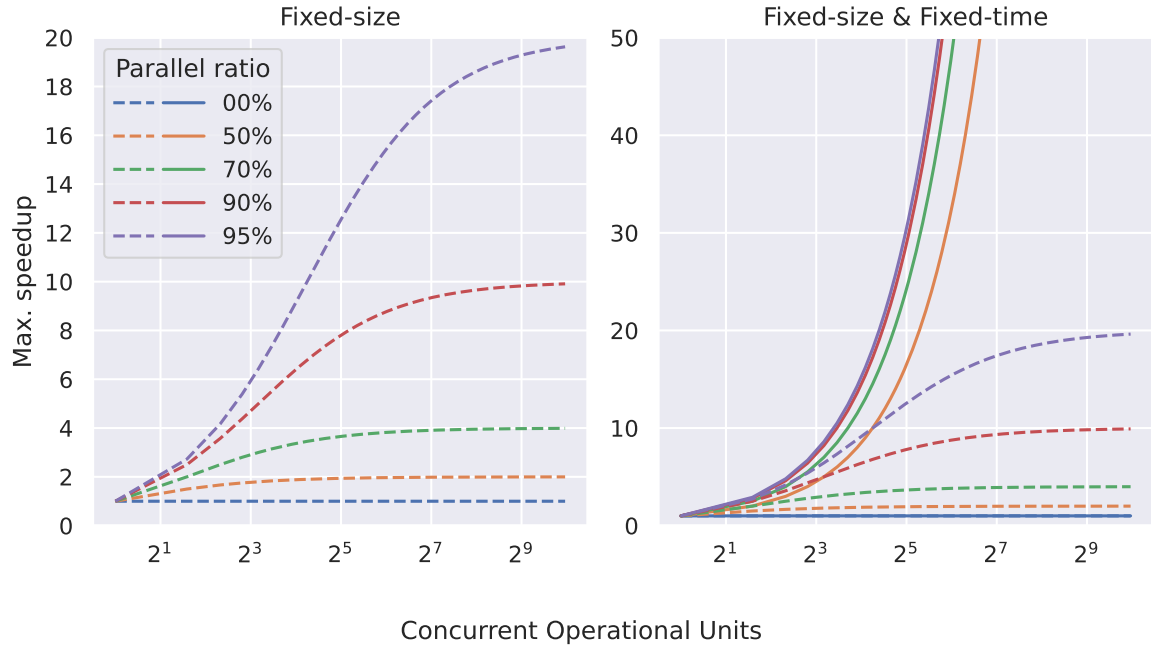


Figure 7.2.: Estimate the maximum potential speed-up as the number of COUs increases from a fixed size or time size perspective.

$$S_N(W') = \frac{W_1 + N \cdot W_N}{W_1 + W_N} \quad (7.8)$$

$$= W_1 + N \cdot W_N \quad (7.9)$$

$$\lim_{N \rightarrow \infty} S_{N,\max}(W') \leq \infty \quad (7.10)$$

Equation (7.9) or its estimate in Equation (7.10) for the case of infinitely many COUs is the result of *Gustafson's law* (Gustafson, 1988).

Choosing a Speedup Variant

Choosing the fixed-size speedup following Amdahl would lead to a rather pessimistic speedup estimate (Furtunato et al., 2020). Using a fixed-size speedup puts the emphasis on a high proportion of code that can be executed in parallel. The higher the potential maximum speedup, the more efficiently the processing units can be used. This view is of interest if the goal is to minimise execution time by using more COUs, or if the input problem does not scale well.

The second approach, using a time-fixed speedup according to Gustafson, emphasises the ability to scale the input problem with minimal negative impact on the computational performance per COU. This approach is preferable when the input problem can be scaled easily and the user benefits from it.

Figure 7.2 shows the potential maximum speedup with respect to different ratios between serial and parallel code. Amdahl requires high parallel ratios to achieve significant speedup, but there is actually an upper limit. Gustafson offers a more optimistic speedup with no upper bound.

For comparison purpose both speedup variants will be analysed during the final evaluation. Amdahl’s approach will be considered during the *strong scaling* benchmarks, in which the problem size stays fixed and the number of COUs is increased. The other side, Gustafson’s approach, is represented through *weak scaling* benchmarks, in which the problem size and number of COUs are increased proportionally. The question remains which speedup variant is more relevant for the problem statement of this work. In Section 1.1.2 use cases were discussed that benefit from larger datasets to extend or improve their expressiveness. An automatic code transformation approach to try out several HPC frameworks that will be used to enable larger data sizes suggests that the weak scaling benchmark will give a meaningful evaluation. Since the original goal was to reduce the memory consumption per COU, one evaluation will focus on the memory trend.

7.2. Test Environment

Three systems were used to run the benchmarks. The first two are typical HPC systems on which most of the benchmarks were run.

The first is Levante, located at DKRZ in Hamburg (Deutsches Klimarechenzentrum GmbH, n.d.-b, 2021). It is currently ranked 60th on the *Top500* list from June 2023 (Strohmaier et al., 2023). The compute nodes used have two *AMD 7763* CPUs with a total of 128 cores and 256 GB/512 GB/1024 GB main memory. They use a 100 Gb/s InfiniBand interconnect and are connected to an *ExaScaler 5* (ES7990X) storage system from DDN (Microway, 2023). The storage nodes even use 200 Gb/s InfiniBand, so unless multiple nodes are fully loading a single Lustre server, the compute nodes’ 100 Gb/s InfiniBand interconnect will not bottleneck the I/O servers. The analysis of Slurm job logs carried out in Section 1.3.2 was performed on Levante’s predecessor system, Mistral. The nodes are running a *Red Hat Enterprise Linux Release 8.6* OS. As a parallel FS it uses Lustre 2.12, modified by DDN. The corresponding libraries were installed using `spack`: LLVM 13.0.0, netCDF 4.9.2 with HDF5 1.10.8 and *MPICH* 3.4.2.

The other HPC system is the research cluster of the *Scientific Computing* group (from now on abbreviated to *WR-Cluster*) at the University of Hamburg (WR, 2023). The compute nodes used have two *Intel Xeon X5650* CPUs with a total of 12 cores and 12 GB of memory. They use a 1 Gb/s Ethernet connection and are equipped with a Lustre 2.15.2 parallel FS. The clients’ OS is *Ubuntu 20.04* and the relevant libraries have been installed using `spack`: LLVM 13.0.0, netCDF 4.9.0 with HDF5 1.12.2 and *MPICH* 4.0.2. The servers use *Rocky 8.7* and have a *Intel Xeon E31275* CPU with four cores and 16 GB of memory.

The last system is a desktop PC used for the compression benchmarks in Section 7.3.3. It has a single *Intel Core i7-6700* CPU with a total of 4 cores and 32 GB of main memory.

Its local FS is ext4 and the OS is *Fedora 37*. The appropriate libraries have been installed using **spack**: LLVM 13.0.0, netCDF 4.9.0 with HDF5 1.12.2 and *MPICH 4.0.2*.

To measure the runtime and memory consumption of an application run, the CATO’s metric collection component is used (cf. Section 5.4.3), which prints out the total runtime, an estimate of the total maximum memory consumption, and the maximum memory consumption per process.

The benchmarks are run against a variety of applications. For the runtime and memory consumption tests in the evaluation, a mimic version of **partdiff** is used, which was presented in Section 4.1.3. The emulated version of **partdiff** removes most of the user interface and reduces the complexity of the code, but retains the central computation component, which uses a stencil pattern to compute on a two-dimensional matrix using the *Jacobi* method. As with the original version of **partdiff**, the runtime of the mimic version is manipulated by a single parameter that sets the side length of the internal two-dimensional matrix. Therefore, the computational cost is expected to increase quadratically with the side length of the matrix. All mentions of **partdiff** in this chapter refer to the mimicked version.

Since **partdiff** does not use I/O, it could not be used for the I/O benchmarks. Instead, a netCDF micro-benchmark was written, which has no significant computational load, but focuses on reading a netCDF file. For compression evaluation, a netCDF micro-benchmark has been created that writes netCDF files with different data patterns.

7.2.1. Current State of Functionality

partdiff covers already many popular OpenMP features (cf. analysis of GitHub repositories in Section 3.2.1). CATO is not feature complete, it is not capable of transforming arbitrary OpenMP code. Unsupported directives are simply ignored, or cause the application to behave differently if the ignored directive had a significant impact. The focus of CATO is on the **parallel** or **parallel for** directive and the **private** and **shared** clauses. Related clauses such as **firstprivate** or **lastprivate** work in principle, even though they are not used by **partdiff**. The **reduction** clause and the **critical** directive are also supported. A more detailed description can be found in Section 5.2.1.

To check the functionality of CATO, a small test script has been created to modify several micro-testkernels with CATO. Each micro-testkernel focuses on a single variable that is accessed in a specific way in an OpenMP kernel. The type of variable also changes, it can be a scalar or dimensional variable allocated on the stack or heap. In the OpenMP kernel it is declared as **shared** or **private** and is read or written to. In some cases where a private heap variable should be accessed, it was necessary to use **firstprivate** instead of **private** so that the initial value is retained when the OpenMP kernel is entered. Otherwise, the memory address allocated before the OpenMP kernel will not be preserved when the OpenMP kernel is entered.

The result of the functionality check is shown in Table 7.1. Most of the 16 possible combinations work, with a few exceptions. These were not needed for **partdiff** and have been postponed. Some can also be handled with a workaround, e.g. the variable from the *private stack read scalar* test case could be embedded in a single element array.

		Heap		Stack	
		read	write	read	write
private	dim	✓	✓	✓	✓
	scalar	✓	~	X	✓
shared	dim	✓	✓	✓	~
	scalar	✓	✓	✓	✓

Table 7.1.: Overview of the implementation status of CATO regarding variable properties.

X: CATO cannot generate modified binary yet.

~: Modified binary is created, but its execution hangs in an infinite loop.

~: Binary execution terminates, but the result is wrong.

✓: Result of the modified binary matches the output of the original binary.

7.3. Evaluation of CATO

This section evaluates the suitability of CATO to support the users of the ESS community addressed in Section 1.3.2. The following Sections 7.3.1, 7.3.3 and 7.3.4 will use benchmarks to evaluate CATO from the perspectives identified as relevant by the research questions set out in Section 1.2. The final Section 7.3.5 will address some final secondary considerations.

7.3.1. CATO Component: Memory Distribution

The main goal of the CATO’s memory component is to automatically allocate memory so that larger input problems can be computed. First, the impact of the change on the runtime is examined, and how well it scales with increasing number of processes and nodes. The number of nodes and COUs (threads or processes) used are combined into a configuration $C(N-n)$, where N is the number of nodes used and n is the number of COUs used per node. This notation is used in almost all visualisations.

There are two relevant runtimes that are measured: the loop time and the total time. The loop time is measured and printed internally by the application itself and includes only the actual computation loop. It excludes everything before and after it, such as data initialisation. The total time is measured with CATO for the modified binary or with Linux `time` command for the unmodified binary. Measuring both values gives additional insight into the potential overhead of the modified version during the initial setup and finalisation phases.

During testing, it became apparent that the same problem sizes could not be used for the unmodified and modified binaries. Jobs running the modified binary ran significantly longer than jobs running the unmodified binary and were aborted due to system timeout. Therefore, larger input sizes have been used for jobs running the unmodified binary, so that the runtime trend of both versions can still be compared. The length of the square matrix dimension is given as MD in the figure captions to facilitate comparison. For strong scaling tests the dimension size remains the same for all configurations, for

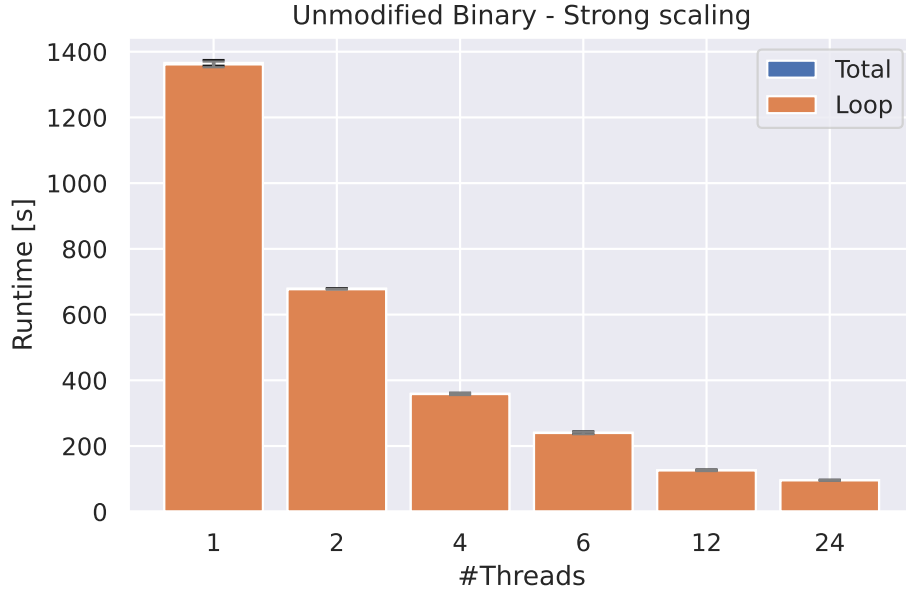


Figure 7.3.: Strong scaling with unmodified binary (WR-Cluster, MD: 25000).

weak scaling tests it is the dimension size for the first configuration C(1-1) (even if it is not printed in the corresponding visualisation) and is then scaled according to the number of participating COUs. To facilitate comparison of the results of the modified binary of Levante and WR-Cluster, the same job files with the same `partdiff` input parameters have been used. Only the number of processes used has been adjusted to suit the system configuration. To reduce the effect of the initialisation phase of the binaries, 40 iterations of `partdiff` were computed in all benchmarks. The results of the strong and weak scaling benchmarks are discussed first, followed by an analysis of the runtime behaviour.

Strong scaling

First, the strong scaling behaviour is evaluated, i.e. the number of COUs is increased while keeping the input size fixed. Figure 7.3 shows the results of the WR-Cluster using up to 24 threads (taking advantage of hyper-threading). The results of the corresponding benchmark run on Levante are shown in Figure 7.4. The visualisations show a good strong scaling behaviour, so the threads are used quite efficiently. The compute nodes on Levante's are better equipped than the compute nodes on WR-Cluster, so the absolute runtimes on Levante are faster.

Contrary to expectations, the modified binary has a significantly worse runtime. In order to stay within system time limits (especially on Levante), the problem size had to be reduced from 25000 (used by the unmodified binary) to 60.

Figure 7.5 shows runtime results for the WR-Cluster. On a single node (C(1-x)) the runtime decreases as more processes are used, but from C(1-12) the runtime increases

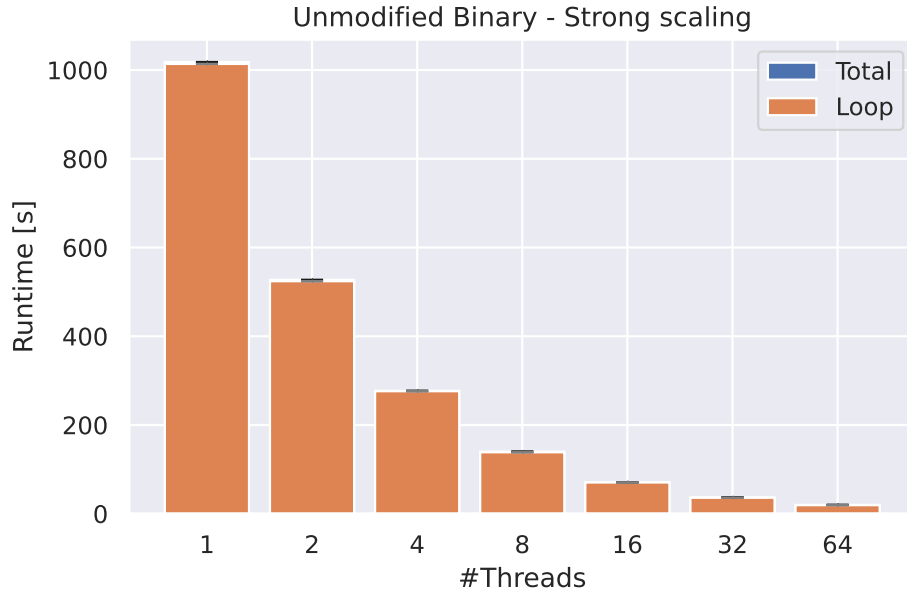


Figure 7.4.: Strong scaling with unmodified binary (Levante, MD: 25000).

again. This could be due to the hardware configuration of WR-Cluster: each compute node uses two CPUs with six cores each (two NUMA nodes). So, probably starting with C(1-12), the performance of intra-node communication decreases as soon as another NUMA node needs to be accessed.

Even more drastic is the increase in runtime when a second compute node is used. In the best case, C(1-2) and C(2-1) should have the same run time, but the Ethernet connection is likely to degrade performance. Except for the C(5-x) configurations, all multi-node configurations have significantly worse strong scaling behaviour, as the average runtime does not decrease but increases with more processes. On the C(5-x) configurations there is at least a decrease in runtime, except when switching to C(5-24).

The pattern of `partdiff` described in Section 4.1.1 does not require all-to-all communication between processes, but only communication between neighbouring processes. Therefore, at most two processes need to perform inter-node communication, the rest will always perform intra-node communication. Going from a single node configuration to a dual node configuration showed a significant increase in runtime: This may be an offset in the runtime that affects any multi-node configuration.

The corresponding runtime results on Levante are shown in Figure 7.6. Not only is the absolute runtime worse, but the scaling behaviour is also comparatively worse. The absolute runtime of the longest runs is significantly longer, even though the same matrix size is used: 23.04s on C(4-4) (WR-Cluster) instead of 17 886.1s on C(5-64) (Levante). Since the input matrix has a rather small dimension (60×60), C(5-64) (320 processes in total) is probably not suitable anyway, as the amount of computation on each process is too small to achieve good performance. However, even on a probably more suitable configuration, such as C(1-4), the result is similar: 0.834 455s on WR-Cluster

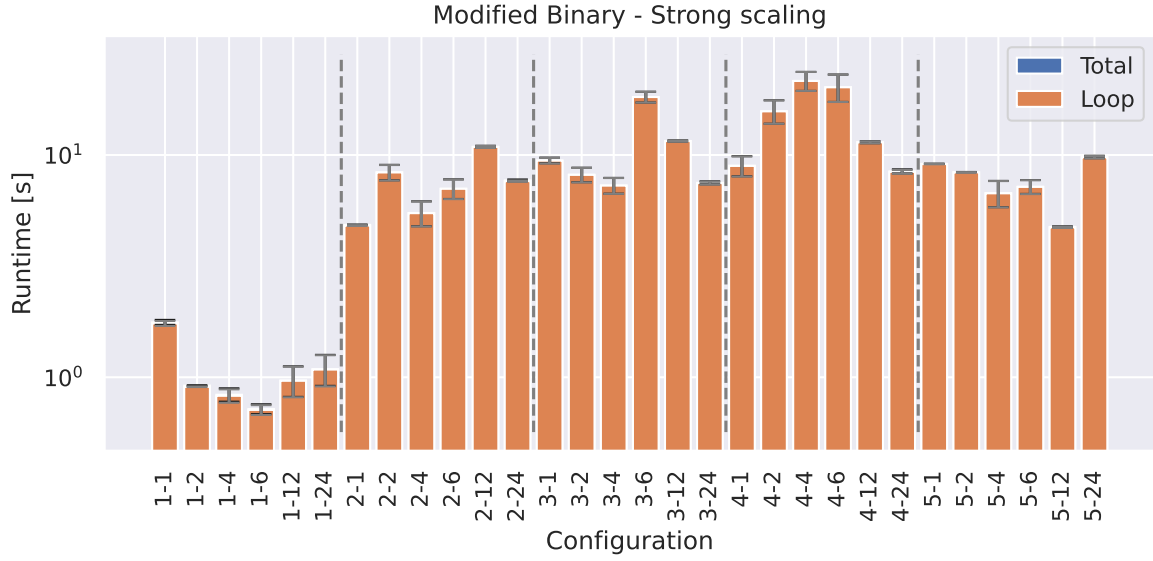


Figure 7.5.: Strong scaling with modified binary (WR-Cluster, MD: 60).

compared to 913.4s on Levante. This is unexpected, as both the compute component and the InfiniBand interconnect are significantly more powerful than their WR-Cluster counterparts, as demonstrated by the strong scaling benchmarks using the unmodified binary. Apart from a longer worst case runtime, the single node configuration also has really long runtimes with core counts starting at four. This cannot be attributed to an interconnect bottleneck.

Weak Scaling

Given the primary goal of CATO, the weak scaling behaviour is important. The size of the matrix dimension in `partdiff` is chosen so that the number of matrix elements increases linearly with the number of COUs. Figure 7.7 and Figure 7.8 show the weak scaling results on Levante and WR-Cluster respectively for the unmodified binary. As before in the weak scaling benchmarks, the unmodified binary has good scaling behaviour as the runtime is quite stable. There is a significant increase in runtime when using 24 instead of 12 threads on WR-Cluster, but this can probably be attributed to hyper-threading as a single node only has 12 physical cores in total. The run with 64 threads on Levante shows a visible difference between the total runtime and the loop runtime; this may be due to the thread creation overhead.

The benchmarks on the modified binary show a similar overall picture to the high scaling benchmarks. For the first four configurations on WR-Cluster in Figure 7.9, the impression is still that there might be a fairly constant runtime, which is to be expected in a weak scaling benchmark. However, the following configurations all have significantly worse runtimes. Again, the bottleneck is communication between nodes, as C(2-1) has a longer runtime than C(1-2). The overhead introduced by CATO and the share of

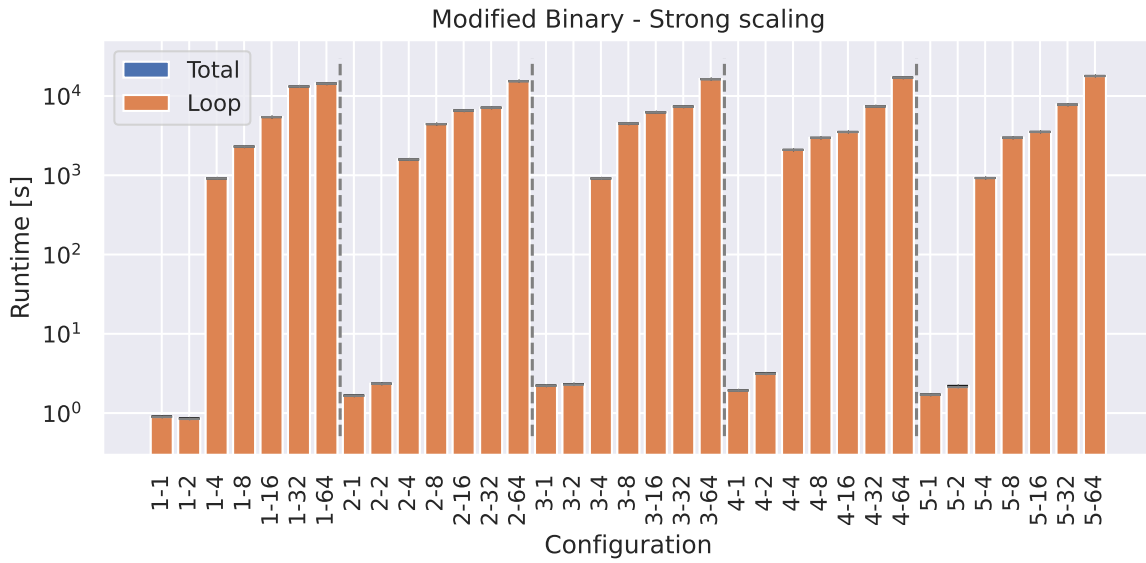


Figure 7.6.: Strong scaling with modified binary (Levante, MD: 60).

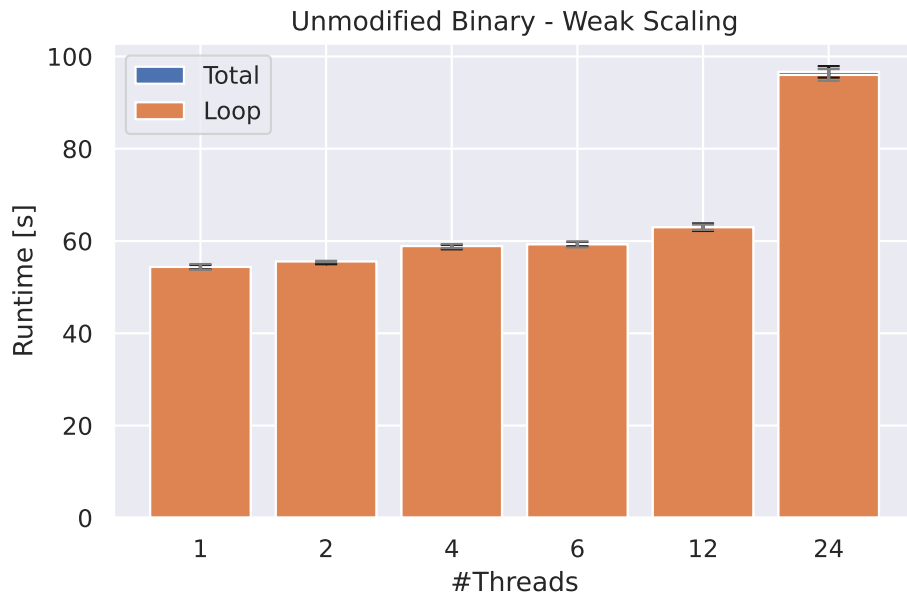


Figure 7.7.: Weak scaling with unmodified binary (WR-Cluster, MD: 5100).

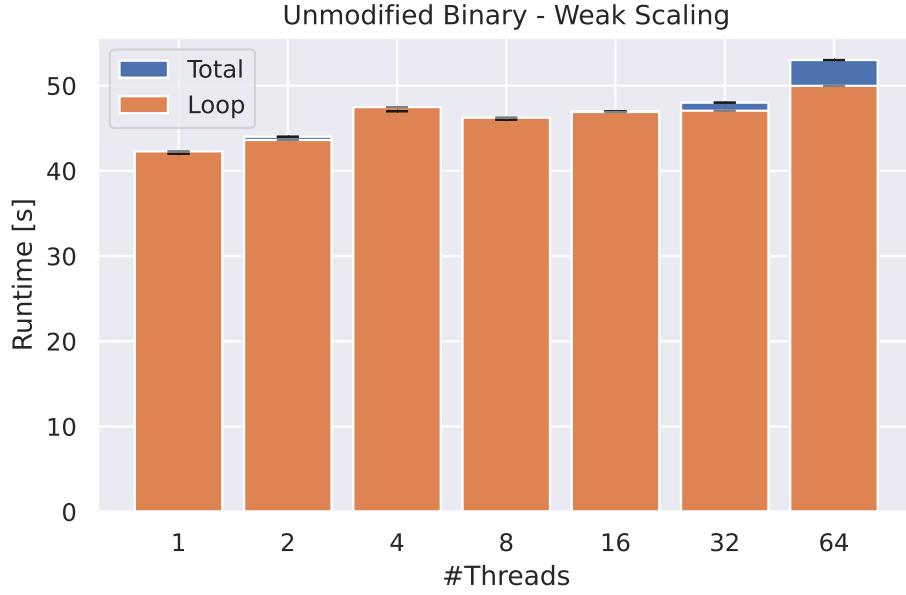


Figure 7.8.: Weak scaling with unmodified binary (Levante MD: 5100).

message passing communication affect the runtime performance.

Again, the general view in Figure 7.10 on Levante shows similar behaviour to the corresponding strong scaling benchmark (cf. Figure 7.6). Configurations C(x-1) and C(x-2) have passable scaling behaviour, but all other configurations have a large offset in runtime that does not resemble a constant trend in runtime. In this case, some configurations (e.g. C(5-8)) could not be executed within the time limit on Levante. As the initial size of the matrix dimension is set to 40, which is already quite small, the missing configurations have been omitted, as it is expected that they still follow the trend.

7.3.2. Runtime Influences

The scaling benchmarks show that the modified binary performs worse than the unmodified binary in terms of absolute runtime. Although the basic idea is that a slower runtime is acceptable in exchange for the ability to compute larger input problems, the change in runtime cannot be ignored.

The benchmarks show a wide range of runtimes. To facilitate comparison, the following four visualisations, Figures 7.11 to 7.14, show the performance factors.

A performance factor indicates how much a particular configuration deviates from the expected result if it had perfectly followed the weak scaling or strong scaling trend. As the single-node jobs ran significantly faster than the multi-node jobs, all factors refer to the corresponding C(x-1) configuration. Optimal weak scaling results in a constant runtime, so the performance factor is the result of $\frac{C(x-y)}{C(x-1)}$. For the strong scaling factor, the number of participating processes per node has multiplied up ($\frac{C(x-y)}{C(x-1)} \cdot y$), because

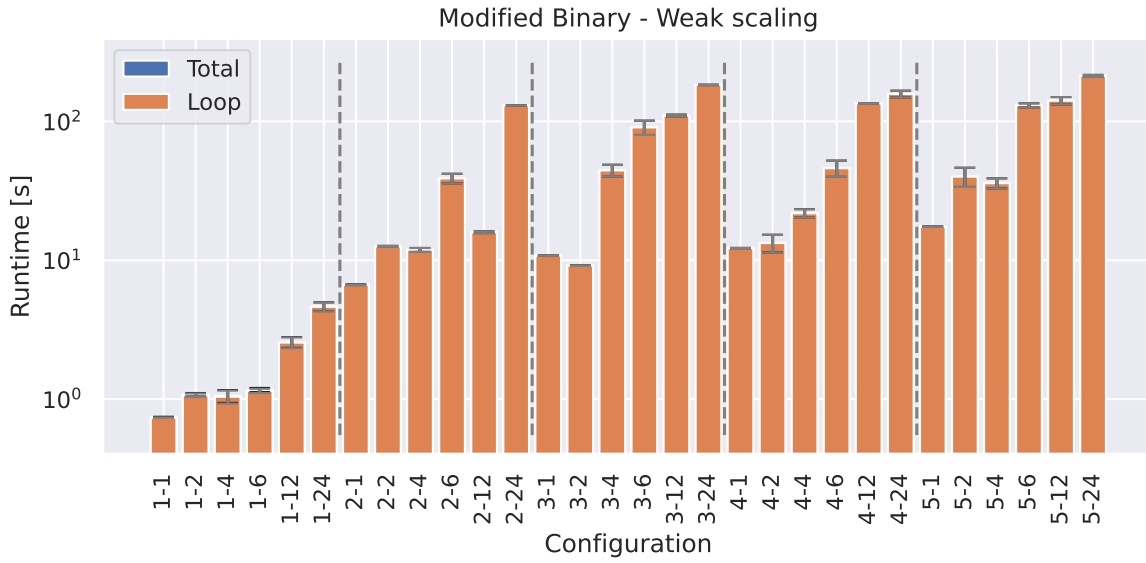


Figure 7.9.: Weak scaling with modified binary (WR-Cluster, MD: 40).

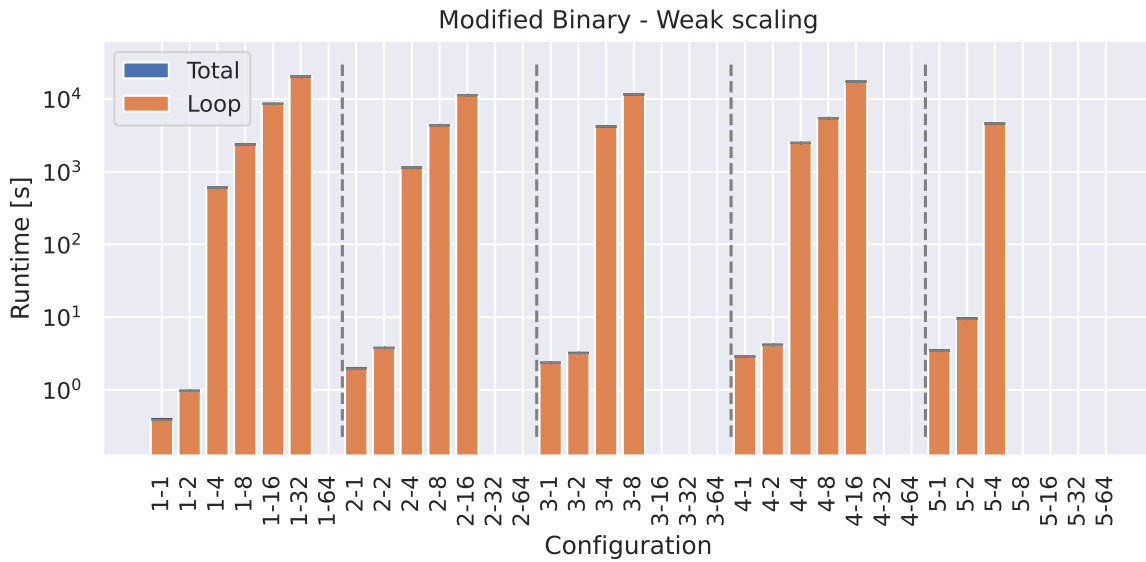


Figure 7.10.: Weak scaling with modified binary (Levante, MD: 40).

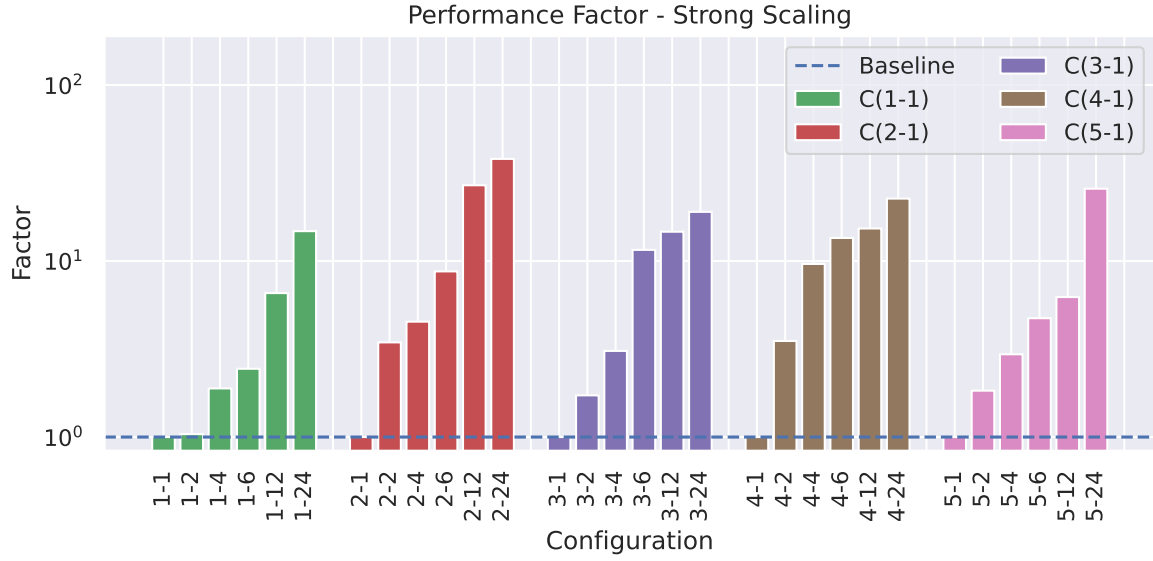


Figure 7.11.: Performance factors of strong scaling with modified binary (WR-Cluster, MD: 60).

optimal strong scaling does not have a constant, but a linearly decreasing runtime. If a configuration behaves as expected, it has a factor of 1 (dashed line).

The results for WR-Cluster (cf. Figures 7.11 and 7.13) and for Levante (cf. Figures 7.12 and 7.14) show that in both cases the weak scaling performs better than the strong scaling. Regardless of the absolute runtime, this is promising since the design of CATO was deliberately chosen to focus on weak scaling. Figure 7.13 shows that there are several configurations with good prospects near the expected baseline. In particular, configurations that use few processes per node do quite well. This suits CATO well, as the main goal is to allow the computation of larger input problems, and the fewer processes involved, the greater the potential memory share per process on a node.

There are a few factors that may have affected the runtime, and these will be discussed.

Interconnect The offset that appeared in the runtime of the modified binary once the execution was spread across multiple nodes was expected. CATO adds MPI communication operations to the original code, which are executed quite often during the benchmarks. Since the latency and bandwidth of the interconnect is an order of magnitude worse than that of the internal bus system of a single node, this leads to a bottleneck as soon as inter-node communication is used. When deriving Gustafson’s law (cf. Section 7.1), the authors neglected the influence of latency to keep things simple. This latency now shows up in the benchmark results.

Conditional branches A general runtime degradation is expected due to the way CATO handles variables on the heap (cf. Section 5.2). As the template traverses the matrix, it accesses each cell and its four immediate neighbours. The modified binary must then check for each access whether the memory address belongs to

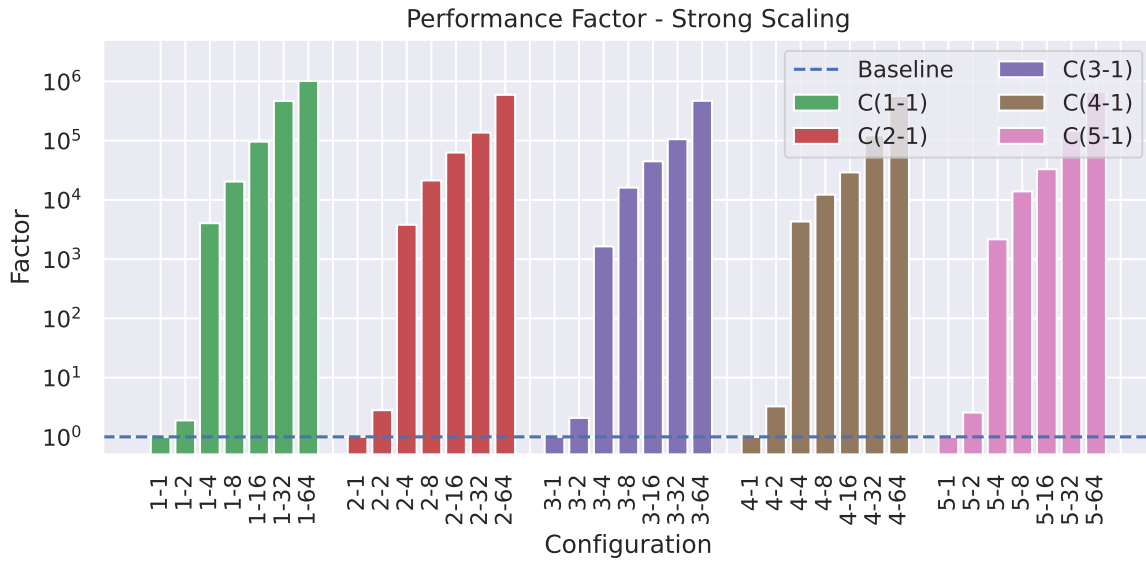


Figure 7.12.: Performance factors of strong scaling with modified binary (Levante, MD: 60).

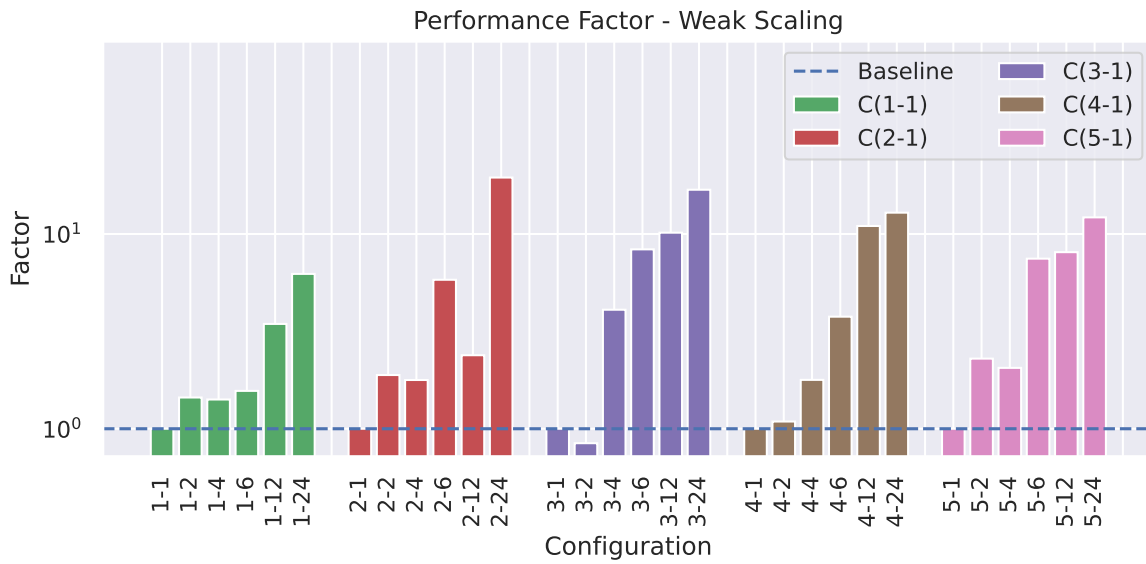


Figure 7.13.: Performance factors of weak scaling with modified binary (WR-Cluster, MD: 40).

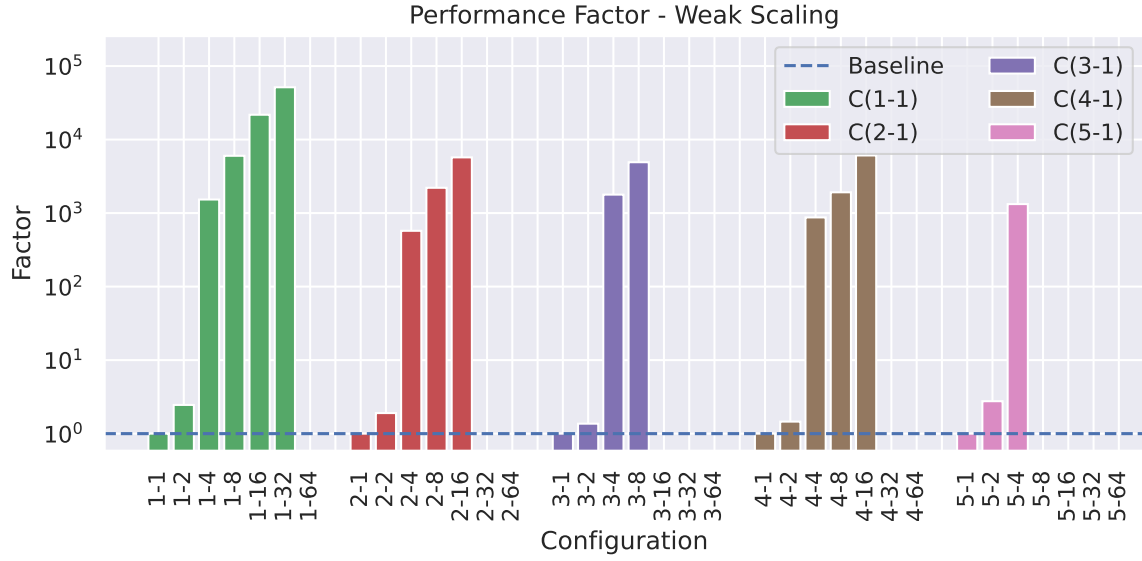


Figure 7.14.: Performance factors of weak scaling with modified binary (Levante, MD: 40).

its own share or is held by another process. If it is on another process, it must be retrieved using MPI, otherwise a local access operation is used.

Halo lines The processes must exchange data from the boundary lines, the halo lines (cf. Figure 4.2). Let the dimension of the matrix be dim and the number of processes n . CATO's modification of `partdiff` distributes the matrix by rows and keeps the number of columns intact. Assuming two neighbouring processes (and therefore two halo lines), and neglecting that the neighbouring processes also request local lines, this results in the following proportion H of halo line elements to *internal* (not part of a halo line) elements:

$$H(n) = \frac{2 \cdot \text{dim}}{\frac{\text{dim}^2}{n}} \quad (7.11)$$

$$= \frac{2 \cdot n}{\text{dim}} \quad (7.12)$$

Equation (7.12) shows that the share of halo line elements increases as more processes are used or as the matrix dimension becomes smaller.

The fraction can be adjusted with respect to a strong scaling factor s , which increases the number of processes, or a weak scaling factor w , which increases both the number of processes and the number of elements.

$$H(n, s) = \frac{2 \cdot n \cdot s}{\text{dim}} \quad (7.13)$$

$$H(n, w) = \frac{2 \cdot n \cdot w}{\text{dim} \cdot \sqrt{w}} \quad (7.14)$$

$$= \frac{2 \cdot n \cdot \sqrt{w}}{\text{dim}} \quad (7.15)$$

The proportion of halo line elements grows faster for strong scaling (Equation (7.13)) than for weak scaling (Equation (7.14)). This again favours CATO's focus on weak scaling, as internal (intra-process) communication on non-halo elements is likely to be cheaper than external (inter-process) communication.

Matrix size Compared to the unmodified binary benchmark, the size of the matrix dimension had to be kept small to meet the maximum job runtime on the test systems. In this case, load balancing can become difficult because the number of rows per process decreases when using a small matrix but many processes. If the distribution is not even (there are remaining rows), then some processes will have to take an extra row. And if there are not many rows in total, some processes will have to take on significantly more load than other processes, and therefore take more time. In addition, a smaller matrix size leads to a higher proportion of (expensive) halo elements, which could particularly affect the strong scaling results.

Experimental Software Cache

It can be assumed that the overhead caused by MPI communication, compared to the local operations performed by the original OpenMP code, degrades runtime performance. Some relief could be provided by implementing a software cache. `partdiff` has many single element accesses during the stencil operations, which are replaced by CATO with a corresponding MPI. It would be beneficial to use a software cache, so that an MPI operation does not just retrieve a single value, but an array of values, speculating that they will be needed in the near future. The stencil pattern moves through the matrix line by line, so this would definitely lead to many cache hits. And these would reduce the runtime.

There is an experimental version of a software cache in CATO, which has been implemented in a master's thesis supervised by me. Based on these results, which have also been published in Squar et al., 2020, Figures 7.15 and 7.16 show a modified `partdiff` version running on the WR-Cluster. The first figure demonstrates the strong scaling capabilities of the WR-Cluster. There is still some offset between the two runtimes, but this time the unmodified and modified binaries have been run with the same input parameters. And this time the offset is only about a factor of 2, while the weak scaling behaviour remains as before.

However, this feature has been implemented with very specific optimisations to the stencil pattern. Before it can be integrated into CATO, further work on generalisations is needed.

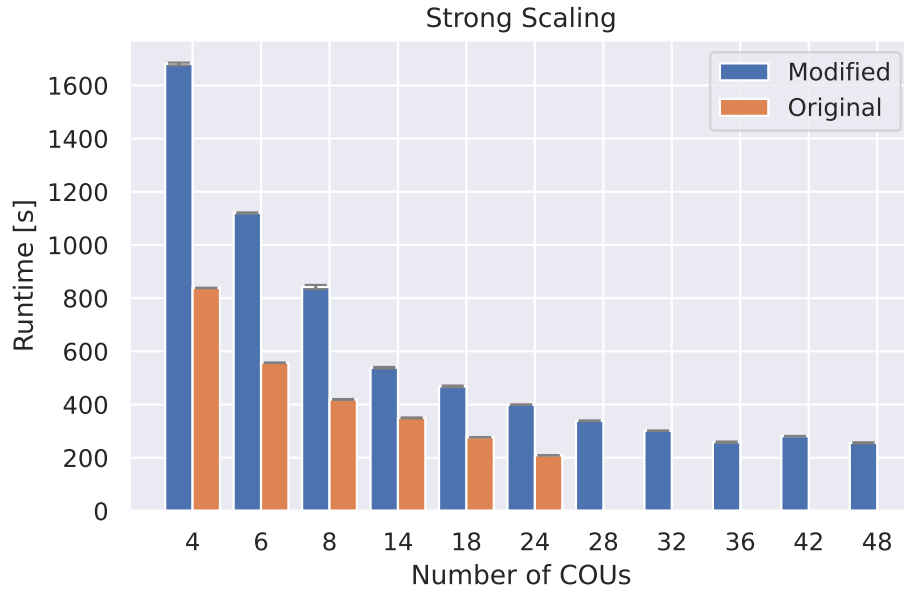


Figure 7.15.: Potential strong scaling using a software cache on up to two nodes (WR-Cluster, MD: 10008) (Squar et al., 2020).

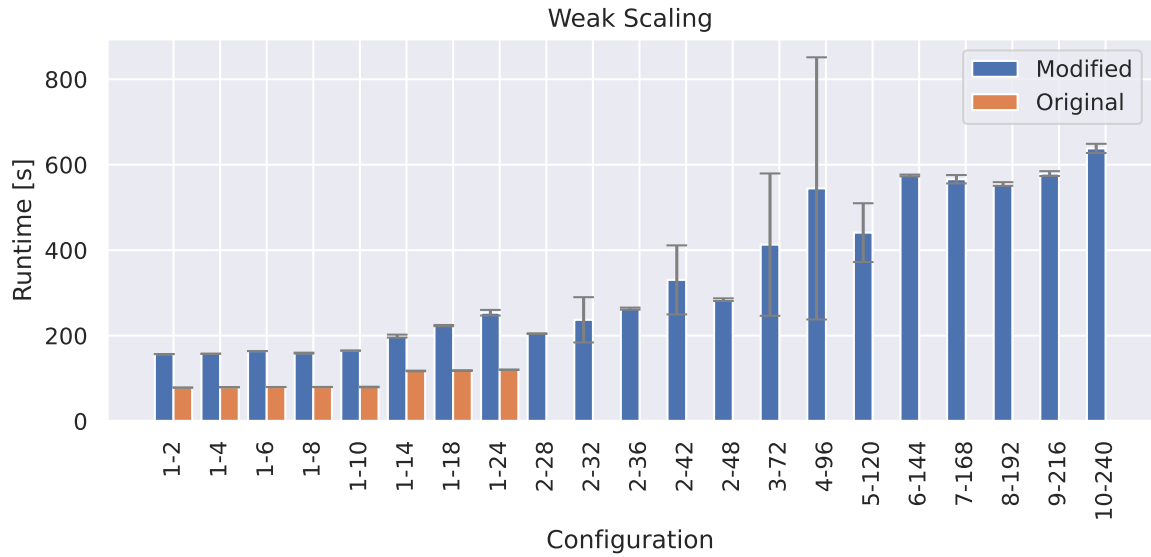


Figure 7.16.: Potential weak scaling using a software cache (WR-Cluster, MD: 1552) (Squar et al., 2020).

Memory Consumption

CATO focuses on optimising the memory footprint of an application by distributing heap memory so that larger problems can be computed. Therefore the memory consumption is examined. For demonstration purposes, the measurements from the weak scaling benchmarks have been used.

The benchmark on the unmodified binary on WR-Cluster and Levante respectively is shown in Figures 7.17 and 7.18. The peak memory consumption starts at 408 977 kB (WR-Cluster) or 409 916 kB, which is quite close to the minimum requirement of the stencil matrix: $5100 \cdot 5100 \cdot 2 \cdot 8B \approx 406406kB$ (two matrices filled with values of type `float`). The other configurations also scale quite well.

The corresponding benchmarks on the modified binary are shown in Figures 7.19 and 7.20 and the initial peak memory consumption is 1034 kB on both systems. The matrix used has a dimension size of 40, so its share of the process memory requirement is $40 \cdot 40 \cdot 2 \cdot 8B = 25kB$. Moving to a two-process configuration has a visible effect on the peak memory usage. This should not be caused by CATO, and perhaps the reason lies in the MPI implementation, which handles a single process differently to a multi-process environment. This is the only notable jump, the trend in peak memory usage behaves as expected and scales quite well. In particular, the average peak memory usage has a good scaling behaviour. On WR-Cluster it starts at C(1-1) with an average peak consumption of 1034 kB and ends at C(5-24) with an average peak consumption of 9105 kB. The memory footprint of a single process increased by a factor of 8.8, while a total of 120 processes could be used. The average peak memory consumption is not optimal, because CATO adds some mandatory overhead to the application's memory footprint (cf. Figure 4.4). For larger input problems this would not be significant, because the overhead scales with the number, but not the size, of heap variables.

7.3.3. CATO Component: Input/Output

The previous section showed some bugs in the runtime of the modified binary, but also showed that the memory usage is optimised so that larger input problems can now be computed. Based on this result, the parallel I/O component of CATO is now evaluated.

Parallel Input/Output

To demonstrate the ability of CATO to perform parallel I/O, a microbenchmark is used that loads a netCDF file with a one-dimensional variable whose size is adjusted during the evaluation. The input file is freshly generated inside the attached parallel FS (Lustre) before each benchmark run to avoid cache effects. Each run is repeated three times. On WR-Cluster there are five compute nodes connected to Lustre, so this is the upper limit of the possible node count. On Levante all 2832 nodes from the *compute* partition are connected to Lustre. To facilitate comparison with the results from WR-Cluster, and because no additional insights are expected from using more nodes, the limit has been set at ten nodes. Therefore, the striping of the input files has been set to 5 for

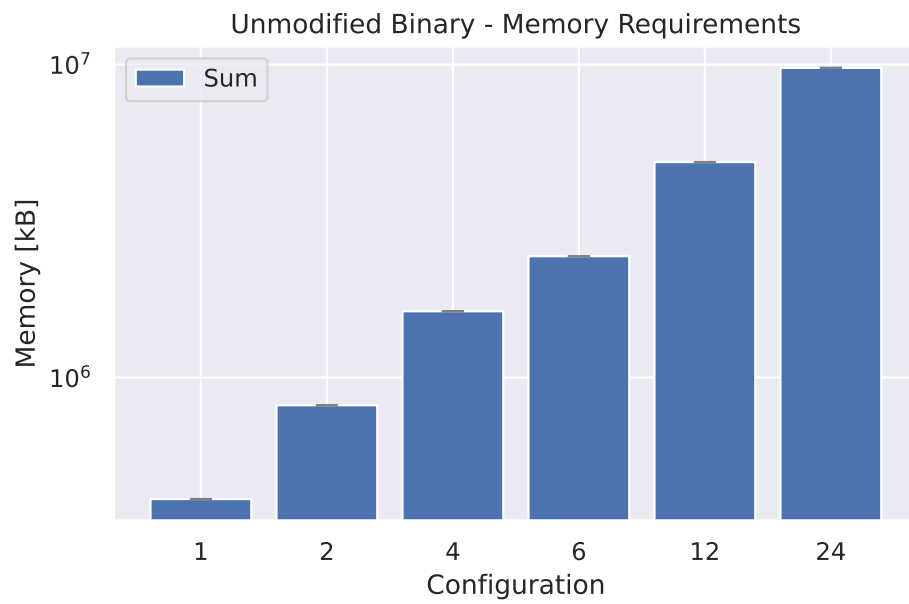


Figure 7.17.: Peak memory consumption of unmodified binary using weak scaling (WR-Cluster, MD: 5100).

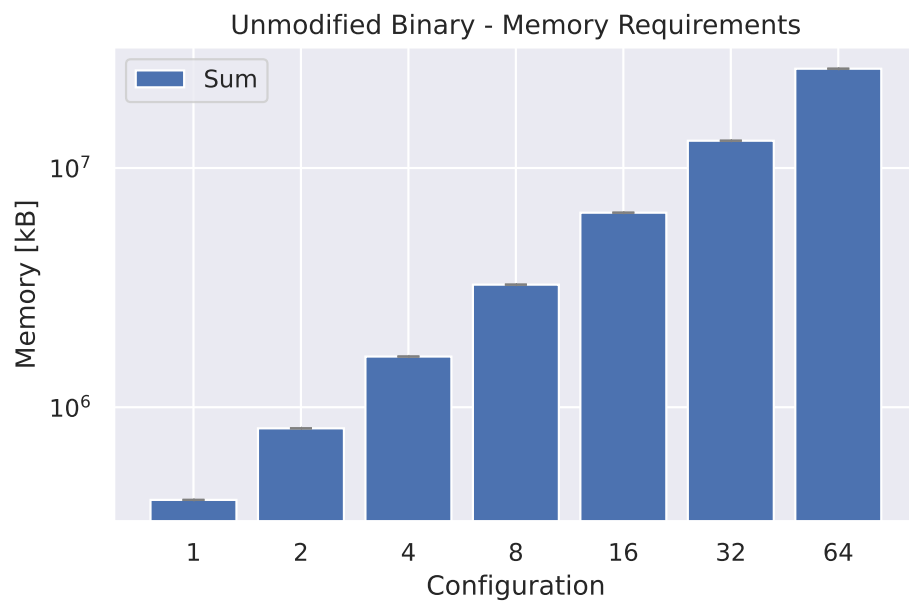


Figure 7.18.: Peak memory consumption of unmodified binary using weak scaling (Levante, MD: 5100).

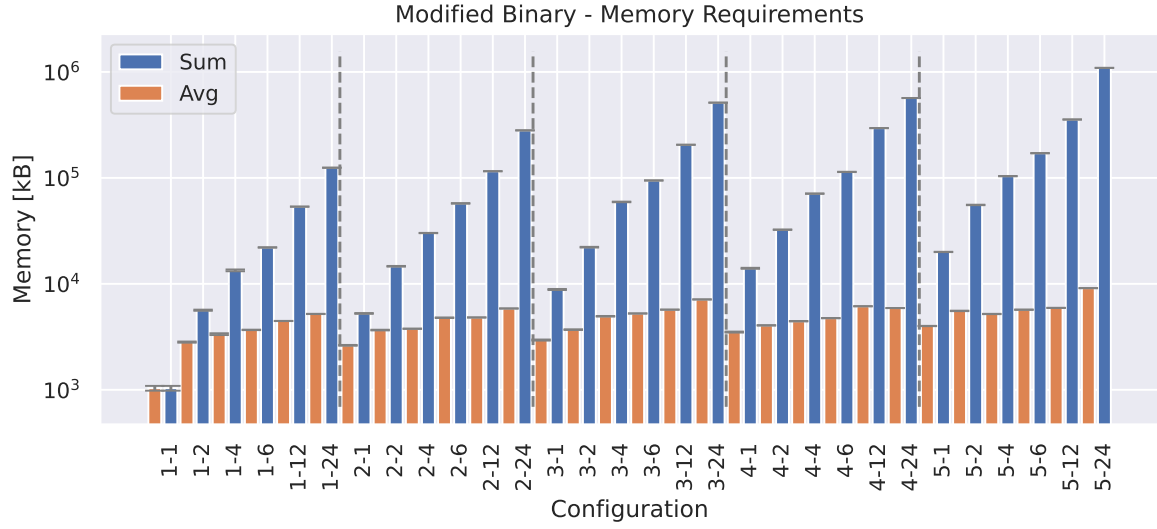


Figure 7.19.: Peak memory consumption of modified binary using weak scaling (WR-Cluster, MD: 40).

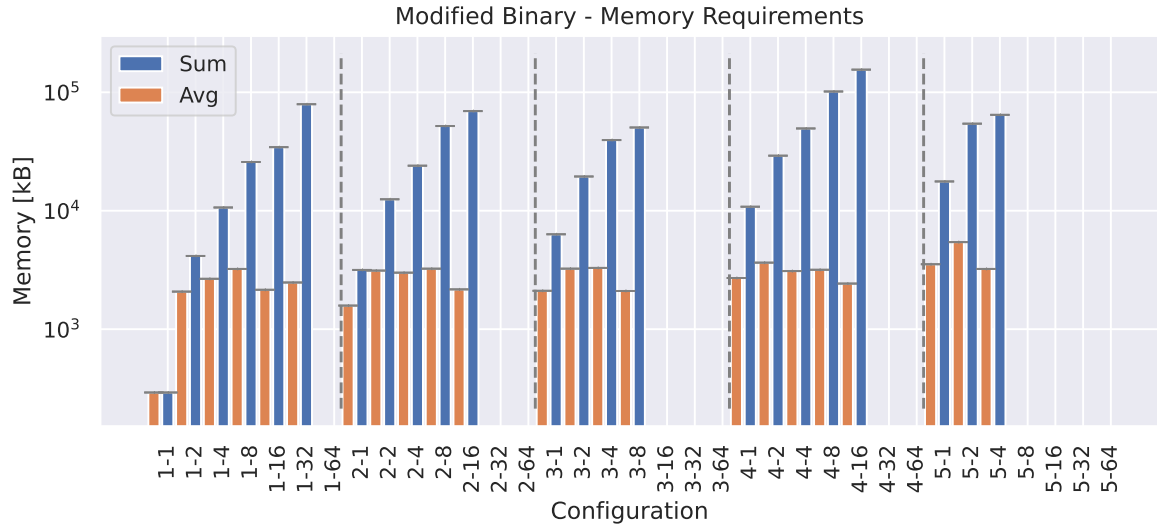


Figure 7.20.: Peak memory consumption of modified binary using weak scaling (Levante, MD: 40).

WR-Cluster and 10 for Levante to match the maximum number of nodes used in this benchmark, so that in the best case each process accesses a different block device.

The efficiency metric can be used to judge how well additional COUs improve the runtime and memory footprint. It is calculated as defined in Equation (7.17):

$$S(n) = \frac{M(1)}{M(n)} \quad (7.16)$$

$$E(n) = \frac{S(n)}{n} \quad (7.17)$$

With:

n	Number of used COUs
M	Measured metric (e.g. time or memory consumption)
$S(n)$	Metric speedup using n COUs
$E(n)$	Metric efficiency using n COUs

All tested configurations use one process per node and three file sizes have been tested: 5 GiB, 10 GiB and 50 GiB.

Figure 7.21 shows the results of the 5 GiB test case executed on WR-Cluster. The first configuration *1-1* shows that there is no advantage to the modified binary if no additional processes are used. Both binaries (unmodified and modified) are bottlenecked by the 1 Gb Ethernet network, which sets the minimum runtime for reading a 5 GiB file to at least 42.95 s. This is roughly the runtime of both binaries (with some additional initialisation overhead), and both processes have a memory footprint of 5 GiB. This changes once the modified binary can use additional processes: The runtime scales down quite efficiently, as does the memory footprint of a single process (cf. Table 7.2). If two processes are used on two nodes, the runtime is approximately halved because each process only needs to read half the data due to memory sharing.

The same benchmark has been repeated on Levante (cf. Figure 7.22) and the results from Table 7.3 show again that memory consumption scales quite well with the increasing number of nodes used. Compared to the same run on WR-Cluster, the runtime is worse than the unmodified binary. Only the *8-8* configuration could keep up. Compute nodes on Levante are connected via a 100 Gb InfiniBand to the Lustre storage servers, so the network bottleneck limits the time to read a 5 GiB file to 0.43 s. The unmodified binary takes considerably longer, probably due to the startup overhead. As the I/O phase is quite short, it no longer has a significant impact on the runtime. The same reasoning applies to the modified binary, where the initialisation of the application and its MPI environment takes significantly longer than I/O, which cannot be sped up by using more processes. So there is no advantage or disadvantage in terms of runtime, only a decrease in runtime efficiency. However, the application benefits from a smaller memory footprint, which again scales quite well with increasing number of nodes.

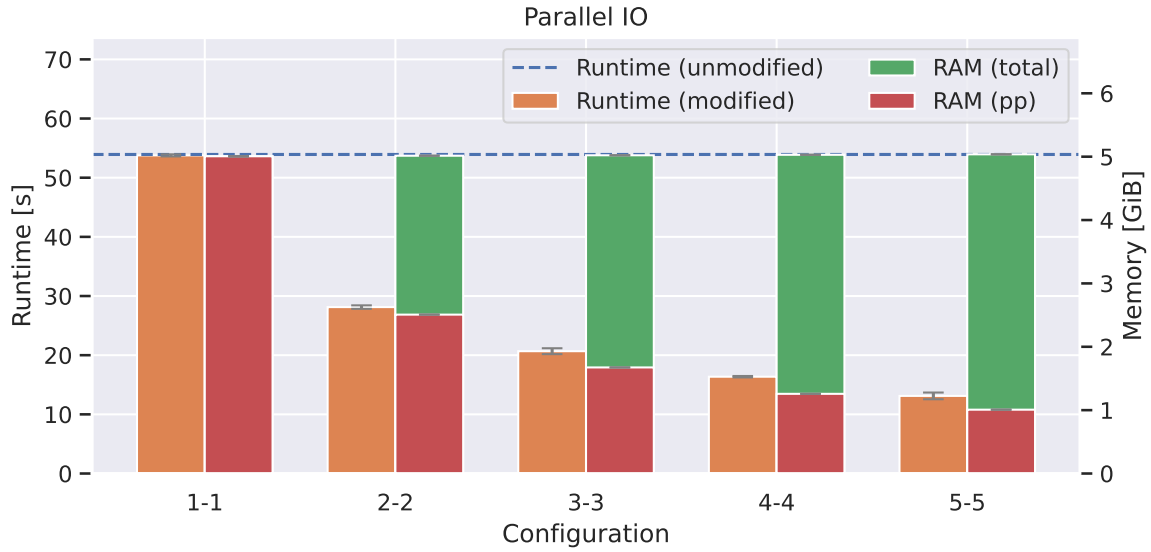


Figure 7.21.: Trend of memory consumption and runtime on WR-Cluster using a 5 GiB netCDF file. Table 7.2 shows the efficiency.

Configuration	1-1	2-2	3-3	4-4	5-5
Runtime Efficiency [%]	100.3	95.9	86.9	82.4	82.2
Memory Efficiency [%]	99.9	99.7	99.6	99.4	99.3

Table 7.2.: Efficiency of parallel I/O on WR-Cluster reading a 5 GiB netCDF file visualised in Figure 7.21.

Configuration	1-1	2-2	3-3	4-4	5-5	6-6	7-7	8-8	9-9	10-10
Runtime Efficiency [%]	88.6	26.7	21.5	19.3	14.9	11.2	11.8	12.2	9.9	9.0
Memory Efficiency [%]	99.9	99.8	99.7	99.7	99.6	99.5	99.4	99.2	99.2	99.1

Table 7.3.: Efficiency of parallel I/O on Levante reading a 5 GiB netCDF file visualised in Figure 7.22.

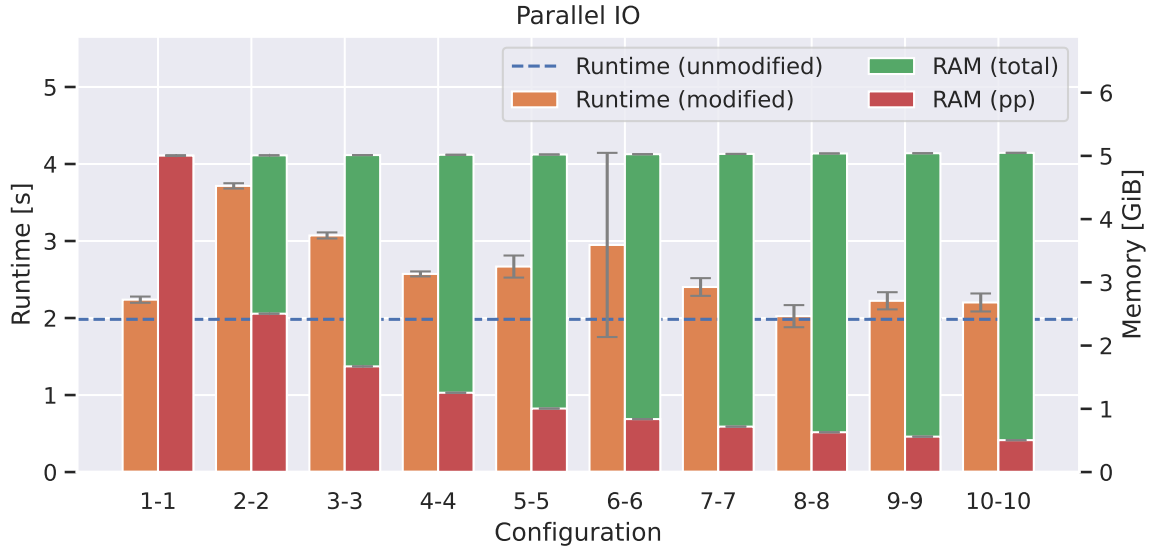


Figure 7.22.: Trend of memory consumption and runtime on Levante using a 5 GiB netCDF file. Table 7.3 shows the efficiency.

Figure 7.23 shows the results of reading the 10 GiB file on WR-Cluster. In this test, the first configuration with one process and one node (1-1) failed. It failed with error code `NC_EHDFERR` inside the netCDF. *partial read* call. According to the netCDF documentation, this indicates an error in the HDF5 layer. This error could be reproduced on both systems with sufficiently large files. Therefore, the measurements for configuration 1-1 for the 10 GiB file and all configurations from 1-1 to 6-6 for the 50 GiB file are missing because they all failed with the same error. In all cases, this error was thrown when a single process tried to read a chunk size of at least something between 7.1 GiB and 8.3 GiB. This will be discussed in future work in Section 8.2.

Other than that, memory usage behaves as before. The total memory consumption is quite stable, and the memory share per process decreases in proportion to the number of nodes. The runtime of the modified binary also decreases in proportion to the number of nodes, as each process only needs to read a portion of the whole file. Again, the network is likely to be the bottleneck: Reading a 10 GiB file over an Ethernet network with a 1 Gb bandwidth takes at least 85.9s. This is close to the serial I/O duration of 106.7s.

Running the same test case on Levante gives the results shown in Figure 7.24. Due to the bug mentioned above, C(1-1) failed. The limit imposed by InfiniBand to read a 10 GiB file is now 0.86s, so still not the dominant cause of the runtime. However, the effect can already be seen, as the general level of runtime is higher than in Figure 7.21, which used the same configuration except for the input file. Apart from that, the efficiency in Table 7.5 is comparable to the run on the 5 GiB file (cf. Table 7.3).

The last I/O benchmark is run on a 50 GiB netCDF file. This could only be run on Levante because the bug that is triggered when reading more than 7 GiB requires at least seven nodes connected to Lustre, which is not the case on WR-Cluster. The

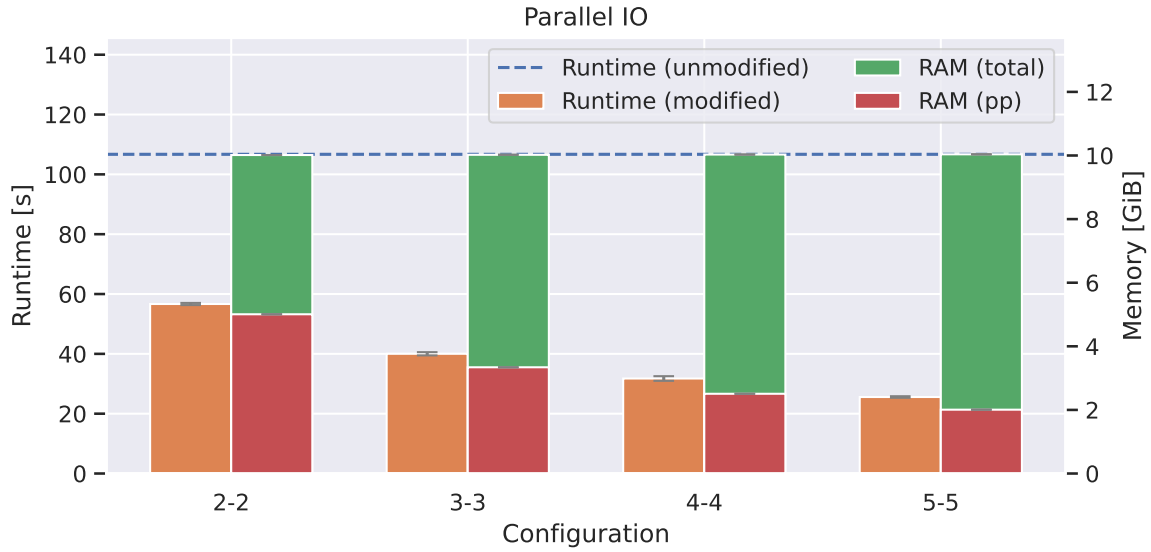


Figure 7.23.: Trend of memory consumption and runtime on WR-Cluster using a 10 GiB netCDF file. Table 7.4 shows the efficiency.

Configuration	2-2	3-3	4-4	5-5
Runtime Efficiency [%]	94.2	88.9	84.0	83.5
Memory Efficiency [%]	99.9	99.8	99.7	99.6

Table 7.4.: Efficiency of parallel I/O on WR-Cluster reading a 10 GiB netCDF file visualised in Figure 7.23.

Configuration	2-2	3-3	4-4	5-5	6-6	7-7	8-8	9-9	10-10
Runtime Efficiency [%]	27.5	22.2	19.4	15.4	15.3	16.4	13.1	9.9	9.0
Memory Efficiency [%]	99.9	99.9	99.8	99.8	99.7	99.7	99.6	99.6	99.5

Table 7.5.: Efficiency of parallel I/O on Levante reading a 10 GiB netCDF file visualised in Figure 7.24.

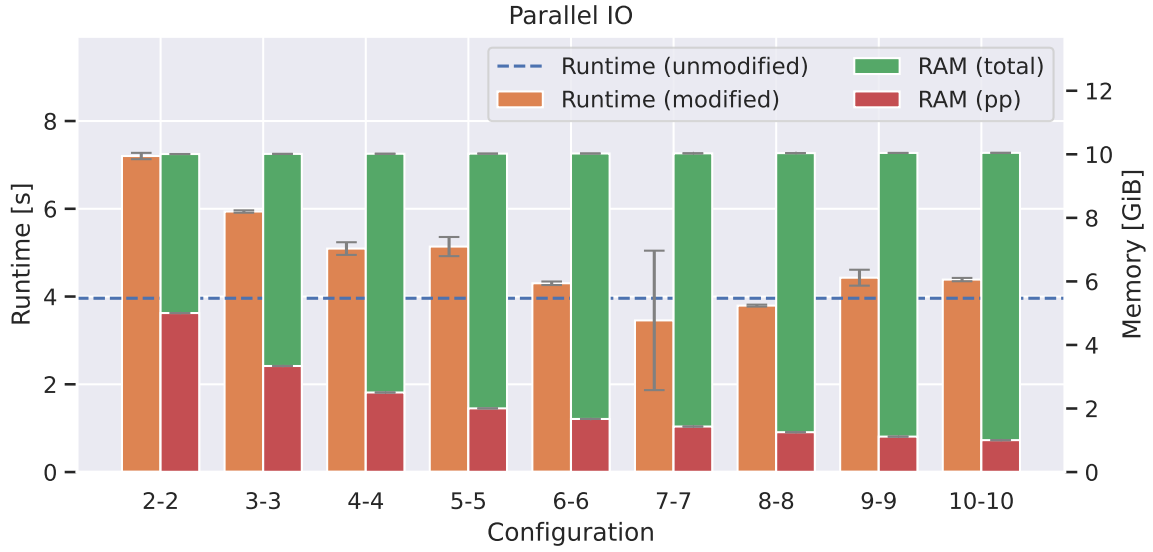


Figure 7.24.: Trend of memory consumption and runtime on Levante using a 10 GiB netCDF file. Table 7.5 shows the efficiency.

Configuration	7-7	8-8	9-9	10-10
Runtime Efficiency [%]	61.8	57.3	60.9	50.6
Memory Efficiency [%]	99.9	99.9	99.9	99.9

Table 7.6.: Efficiency of parallel I/O on Levante reading a 50 GiB netCDF file visualised in Figure 7.25.

results are shown in Figure 7.25 and the corresponding efficiency in Table 7.6. Again, the memory efficiency scales very well. The runtime efficiency became much better than in the benchmark on the 10 GiB file (cf. Table 7.5). The limit set by InfiniBand for reading the 50 GiB file is 4.29s. So the unmodified binary now takes significantly more time to read the file, probably because the file has to be fetched sequentially from up to 10 different OSSs, which could still cause some overhead, or because of other internal handling done by the netCDF library.

The evaluation of the parallel I/O component has demonstrated how an application can benefit from using CATO. By using the memory handling component to distribute large heap memory to multiple processes on independent nodes, the CATO’s parallel I/O component allows the modified application to perform I/O on larger files that would not fit in the memory of a single node.

Compression

The third major transformation component of CATO deals with compression. As a test case, a netCDF micro-benchmark was created that writes a one-dimensional `float`

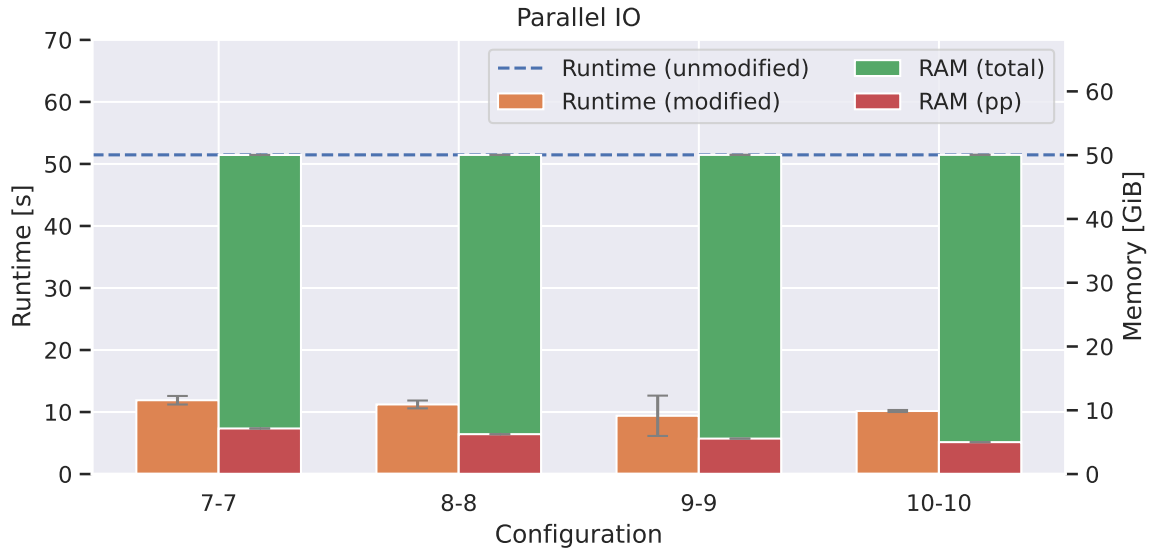


Figure 7.25.: Trend of memory consumption and runtime on Levante using a 50 GiB netCDF file. Table 7.6 shows the efficiency.

variable with a fixed dimension size. CATO was then used to modify the micro-benchmark to automatically insert the necessary netCDF calls to enable compression.

This time, the focus is not on the runtime or memory performance of the micro-benchmark, but on the resulting size of the written output file. As the chosen data pattern has a strong influence on the entropy of the data and therefore how well it can be compressed, six different patterns were used, which are visualised in Figure 7.26:

zeros: Zeros only.

lin_inc: The data values increase linearly from 0.0 to the last number (depending on the chosen file size) with 1.0 increments.

lin_mod: Like *lin_inc* but a modulo of 10 has been applied so that the data ranges are in $[0.0, 10.0)$.

rand_1000: Random values between 0.0 and 1000.0.

rand_norm: Random values between 0.0 and 1.0.

rand_mod_2: Random values between 0.0 and 1.0 and every other value is set to 0.0.

To avoid transparent compression by the FS, which is not uncommon on HPC systems, the execution and measurements were performed on a desktop PC, as described in Section 7.2. Since compression has an impact on runtime and memory performance, and in the case of lossy compression also on data quality, the user must explicitly enable compression by setting the appropriate environment variables. These settings

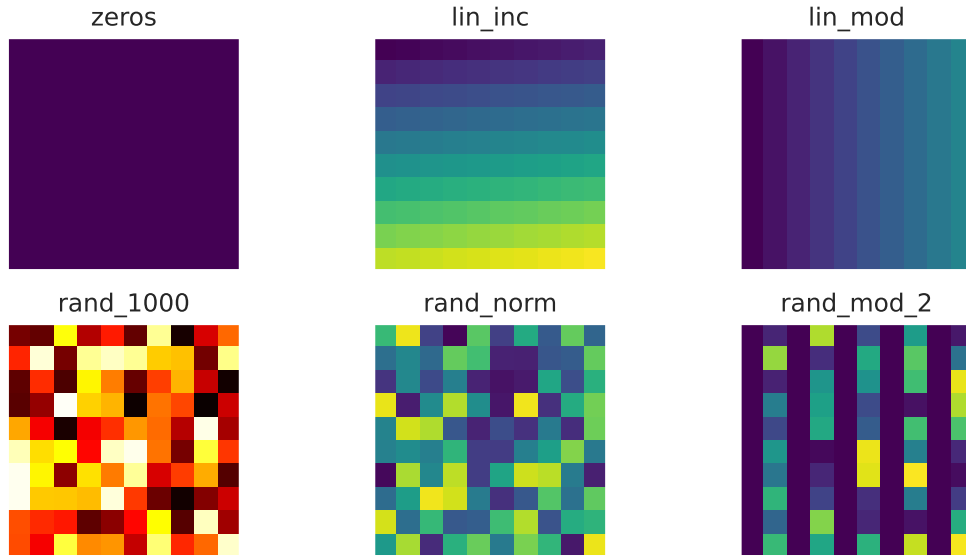


Figure 7.26.: Sample visualisation of examined data patterns.

include *alignment* and *chunking* in netCDF, as well as two parameters (*mode* and *number of significant digits*) for the lossy *quantize* preprocessor and two parameters (*level* and *shuffle*) for the *deflate* compressor. Since there are many possible combinations (all parameters can be set independently), a script for iterating and evaluating these combinations, which is part of the CATO’s script collection, has been used. This allows the combination space to be automatically explored to find the most appropriate combination of parameters. A selection of 384 combinations were tested on each pattern, for a total of 2304 combinations.

During the investigation, an unforeseen peculiarity was discovered. The modified binary inserts a large, empty data block between the metadata section and the data section of the written netCDF file. To demonstrate this, `binocle` was used to visualise the contents of the file (Peter, 2023).

Figure 7.27 shows a comparison of the netCDF files written by the unmodified and modified binaries. In the upper part of Figure 7.27 you can see some non-zero lines belonging to the metadata header of the file. The big black block is just zeros followed by the actual data using the *lin_inc* pattern. It is currently unclear why this block is inserted into the output file. It is not part of the data itself, since it can still be read, and using `ncdump` shows that the data is intact and has the correct dimensional length, so netCDF does not consider these zeros to be part of the data.

Two Linux tools were used to determine the file size needed to evaluate the compression results: `ls` and `du`. `ls` shows the file size in bytes and `du` shows the actual use of disk space, allocated in blocks. `ls` includes the unknown block between the metadata and the actual data, so the file size results are larger than the size of the file created by the unmodified binary. `du` excludes this block and therefore reports smaller file size results. `ls` includes the block when printing the file size, `ncdump` does not show the block within

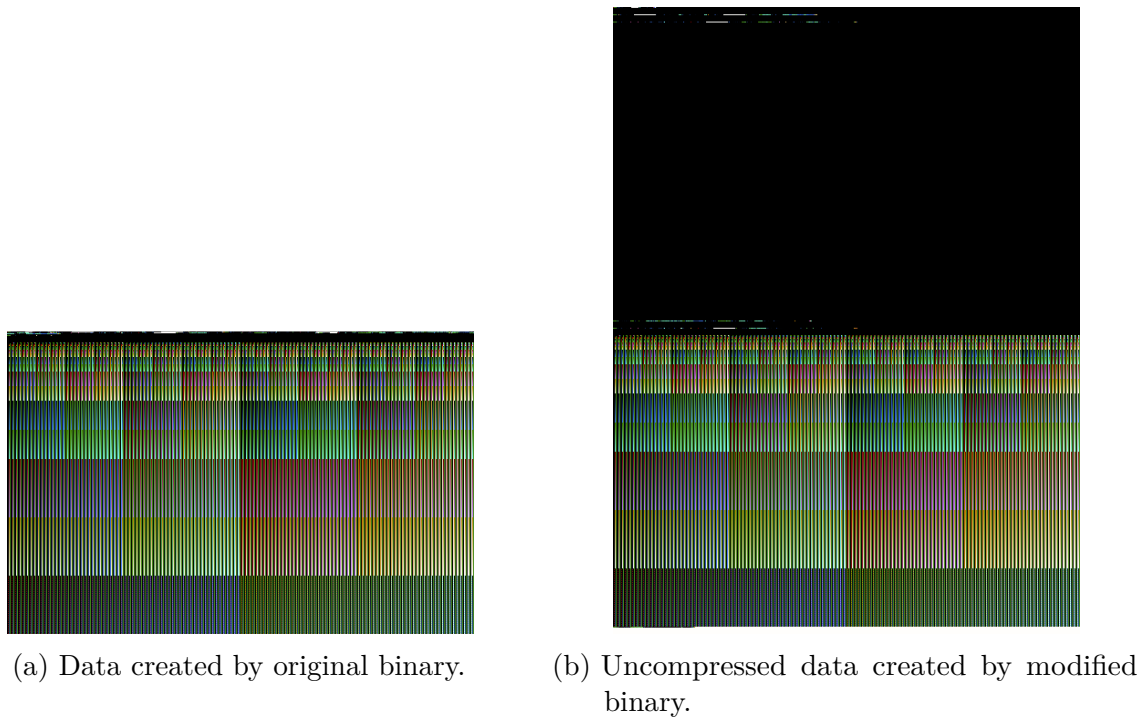


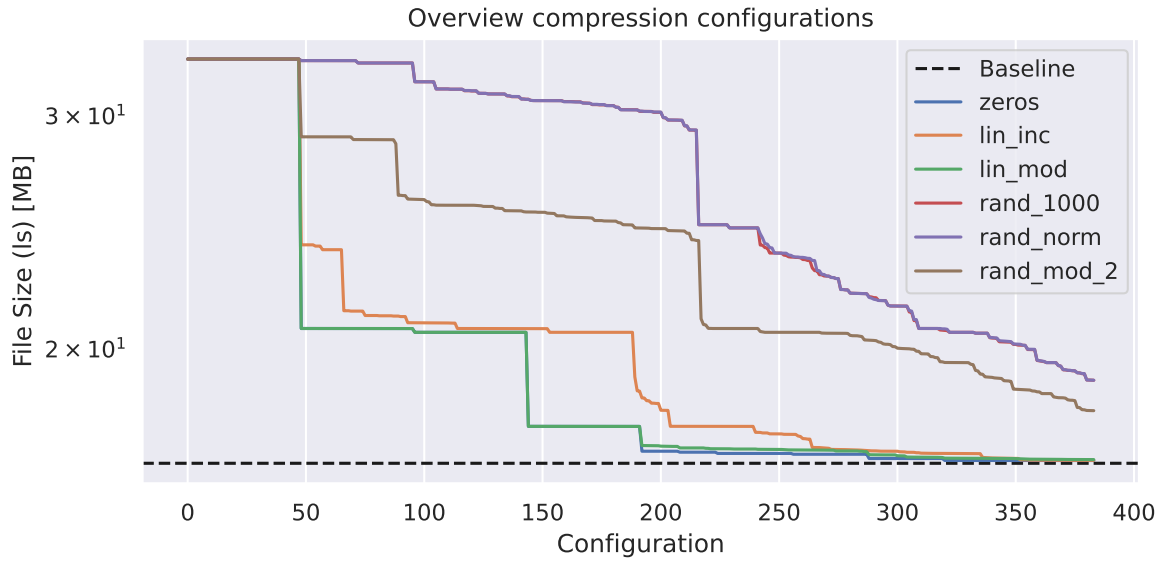
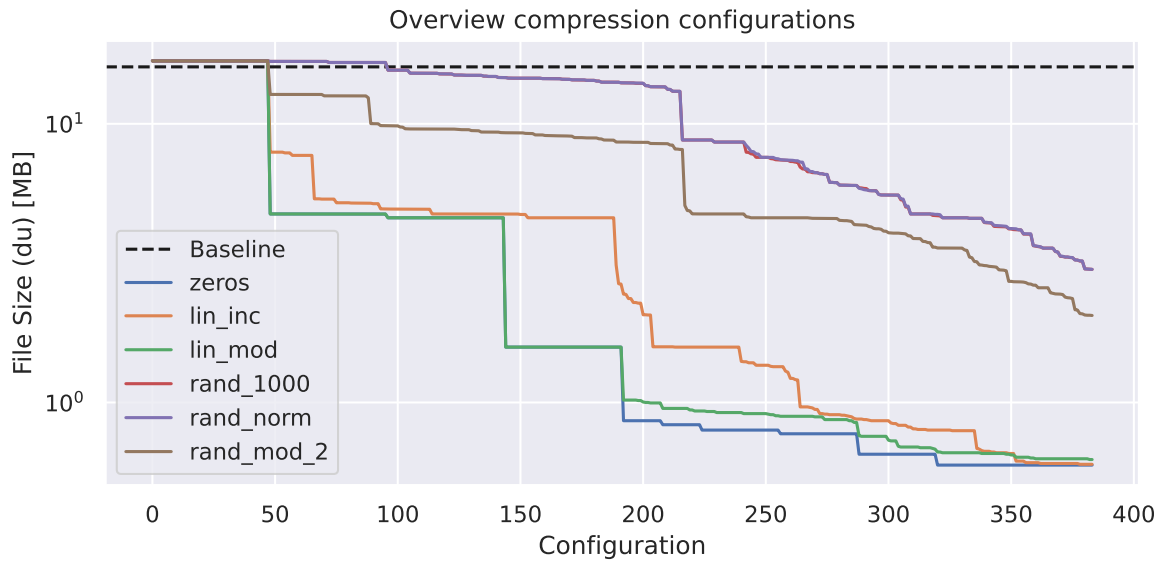
Figure 7.27.: Visualisation of actual data bytes on disk using `binocle`.

the data and `du` excludes the data when printing the disk usage.

As it is not clear which statement is correct, the resulting file sizes have been plotted with both `ls` and `du`. The trend of both graphs is the same, the only difference is the offset added by `ls`. Figure 7.28 shows the results of `ls` and Figure 7.29 shows the results of `du`. The configurations have been sorted by the compression ratio achieved to give a clearer picture.

The output file from the unmodified binary always has a size of 16 390 144 Byte (about 15.6 MiB), regardless of the chosen pattern. As expected, the more trivial data pattern *zeros* achieves a high compression ratio, and the more noisy *rand_** patterns are harder to compress. In Figure 7.29 there are some configurations visible that increase the file size above 15.6 MiB, so compression can indeed lead to larger output files. This is not the fault of CATO but is simply something that can happen, if inappropriate compression settings are chosen. Therefore, the user cannot use any configuration without checking its effect. Table 7.7 gives an overview of the five best configurations for each data pattern.

The effect of compression is visualised with `binocle` in Figure 7.30. On the left is the uncompressed output file created by the unmodified binary. On the right is the compressed file created by the modified binary. Again, the metadata header is visible at the top, followed by a large block. This block is probably related to the black block of zeros in Figure 7.27b, but now these are all *FF* values (hence the block is white). You can see a small section at the bottom, which is the supposedly compressed data. `ls` reports that both files are about the same size, while `du` reports a file size of about 0.57 MiB.

Figure 7.28.: Overview of all benchmarked configurations using the results of `ls`.Figure 7.29.: Overview of all benchmarked configurations using the results of `du`.

ID	Pattern	ls [MB]	du [MB]	alignment [B]	chunking [B]	quantize	deflate
213	zeros	16.47	0.60	8	4096	2:6	6:1
149	zeros	16.47	0.60	8	4096	2:6	6:0
348	zeros	16.47	0.60	1	4096	3:6	9:1
349	zeros	16.47	0.60	8	4096	3:6	9:1
276	zeros	16.47	0.60	1	4096	2:6	9:0
316	lin_inc	16.47	0.60	1	4096	3:1	9:0
252	lin_inc	16.47	0.60	1	4096	3:1	6:1
308	lin_inc	16.47	0.60	1	4096	2:1	9:0
180	lin_inc	16.47	0.60	1	4096	2:1	6:0
380	lin_inc	16.47	0.60	1	4096	3:1	9:1
380	lin_mod	16.50	0.62	1	4096	3:1	9:1
252	lin_mod	16.50	0.62	1	4096	3:1	6:1
372	lin_mod	16.50	0.63	1	4096	2:1	9:1
220	lin_mod	16.50	0.63	1	4096	3:6	6:1
348	lin_mod	16.50	0.63	1	4096	3:6	9:1
252	rand_1000	18.94	3.00	1	4096	3:1	6:1
253	rand_1000	18.94	3.01	8	4096	3:1	6:1
380	rand_1000	18.94	3.01	1	4096	3:1	9:1
381	rand_1000	18.94	3.02	8	4096	3:1	9:1
317	rand_1000	19.15	3.22	8	4096	3:1	9:0
252	rand_norm	18.94	3.00	1	4096	3:1	6:1
253	rand_norm	18.94	3.01	8	4096	3:1	6:1
380	rand_norm	18.94	3.01	1	4096	3:1	9:1
381	rand_norm	18.94	3.02	8	4096	3:1	9:1
316	rand_norm	19.16	3.22	1	4096	3:1	9:0
380	rand_mod_2	17.96	2.05	1	4096	3:1	9:1
381	rand_mod_2	17.96	2.06	8	4096	3:1	9:1
252	rand_mod_2	17.96	2.06	1	4096	3:1	6:1
253	rand_mod_2	17.97	2.06	8	4096	3:1	6:1
316	rand_mod_2	17.99	2.08	1	4096	3:1	9:0

Table 7.7.: Overview of the five best compression configurations. Rows are sorted by size within each pattern. *ID* is a random but unique identifier of a specific configuration.

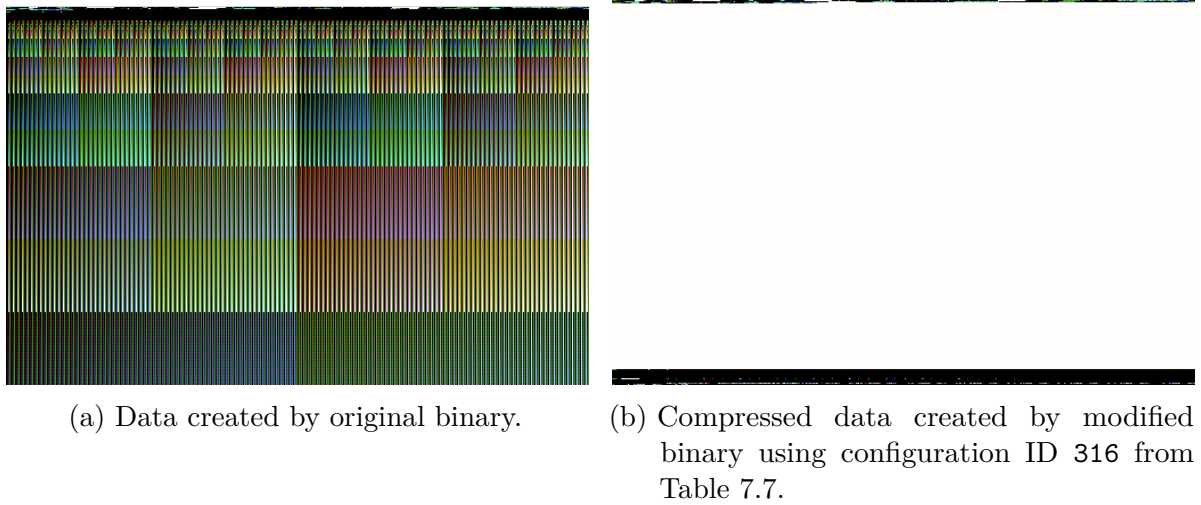


Figure 7.30.: Visualisation of actual data bytes on disk using `binocle`.

Depending on the use case and the nature of the data being written, compression can be very beneficial in reducing the memory and storage footprint of an application. It can also be used to improve the efficiency of hardware usage by utilising idle components or to speed up data transfer over the network. The potential advantages and disadvantages of compression have been discussed in Section 3.3.

7.3.4. CATO Component: Feedback

As discussed in Section 5.4 CATO provides four different ways, to assist the user in using CATO:

- Help the user to reenact the code changes.
- Improve the code quality.
- Provide performance metrics.
- Help the user to use CATO.

All together they help to avoid mistakes, provide more insight if the user is interested and eases the handling of CATO.

In Section 4.3.1 decompilers have been examined if they are a reasonable choice to show to the user what the modified code could look like. The tested solutions, which decompiled the modified binary or modified IR code back into high-level code, worked but already for trivial examples the results became hard to read for untrained people. An alternate solutions, *LLVIs*, had been tried, which took advantage, that the original source code of the application is available and combined the CFG of the unmodified source code with the IR code of CATO’s ECs. Trivial examples were promising but already showed some flaws. If CATO is applied on the same source code and then the CFG with

high-level code mapping is generated, the output quality becomes worse. This was to be expected, because in the current design CATO does not pay attention to the debug symbols, which means, that misleadingly debug symbols of replaced IR instructions are taken over or the debug symbols are simply missing. *LLVis* had been tried on the netCDF micro-benchmark, which was used to read the netCDF file during the parallel I/O evaluation. The output of *LLVis* after CATO has modified the code is visible in Figure A.2, which barely has any high-level code.

Measures to improve the code quality have already been discussed in Section 5.4.2, therefore there is no need to pick up this topic here again.

Section 5.4.3 described how CATO inserts code to collect metrics about the runtime as well as the total peak memory consumption. Printing them can easily be enabled by the user at runtime and makes it possible for him to easily assess the performance of the modified binary. This is especially useful if the user wants to try out several runtime configurations for the modified binary for example, as their runtime and memory metrics can easily be compared. This feature has also been used heavily during the evaluation in this chapter to gather the values of runtime, total peak memory consumption and the assumed average peak memory consumption per process.

Section 5.4.4 described how CATO can present its features to the user. CATO adds all possible functions during the replacement into the transformed code. By default only the memory distribution and parallel I/O using netCDF is enabled. All compression features are not enabled since the user should decide for himself, if his use case can benefit from compression. To keep the overview of all environment variables, which are used to enable optional features, CATO provides a help page, which lists all environment variables but also provides links for official documentation and repositories with example codes, which the user can use as teaching material if he is interested. At the moment there are help pages for the general usage of CATO, netCDF and MPI. Figure 7.31 is a screenshot, which shows an extract of CATO's help page about netCDF.

7.3.5. Influence on Compilation Workflow

Using CATO not only directly changes the final binary, but also affects the user's workflow. The compilation process is adjusted and things like compile time or binary size may change. For small applications this is probably not worth mentioning, but for large applications it can become significant. A larger binary size is probably not a problem, since the targeted HPC applications are usually not executed in a space-constrained environment (e.g. embedded systems). In fact, the micro-benchmarks used during the evaluation had an even smaller binary size. Using the same level of optimisation (`-O2`), the unmodified `partdiff` binary has a size of 41 kB, while the modified binary has a size of only 29 kB. The reason for this is that OpenMP is removed during the transformation, which means that for example all the code needed for forking and merging and load balancing the threads is removed. In return, CATO modifies and adds to the IR code, but most of the replacement code, which is wrapped in ECs, is outsourced to a shared runtime library. Function calls are often added to the IR code or existing function calls are simply replaced. The runtime library currently has a size of 1.7 MB and is needed to

```

Environment variables to enable optional netCDF features
#####

IO mode:
  CATO_NC_PAR_MODE
  Choose collective mode for parallel netCDF4. Available options are:
    COLLECTIVE  IO is performed collectively by all processes, which potentially offers better I/O performance (default)
    INDEPENDENT Each process can independently perform I/O (not recommend, currently no use case in replacement code)

#####

Compression
  CATO_NC_CMPR_DEFLATE  deflate_level[:shuffle]
  CATO_NC_CMPR_QUANTIZE  nsd:mode
                        nsd:  Number of significant digits (at least 1)
                        mode:  Quantization mode
                                0:  NC_NOQUANTIZE
                                1:  NC_QUANTIZE_BITGROOM
                                2:  NC_QUANTIZE_GNANULARBR
                                3:  NC_QUANTIZE_BITROUND

#####

Chunking
  CATO_NC_CHUNKING
  Add chunking calls to variables (Value is interpreted as chunk size, 0 enables automatic determination of chunk size)

Alignment
  CATO_NC_ALIGNMENT
  Data is aligned on an address, which is a multiple of this specific block size and should match the system disk block size or a multiple of it (default: 4096)

#####

Sanitizer:
  CATO_SANITIZER_NC_ERROR Set how additional error checks shall be performed. Available options are:
    WARN  If the error value of a netCDF function is not 0, a warning is printed (default)
    ERROR If the error value of a netCDF function is not 0, the application is terminated
    OFF   Results of error checking are ignored
    DISABLE No error checking functionality is added (needs to be set before the compilation!)

```

Figure 7.31.: Screenshot of CATO’s help page about netCDF showing a snippet of usage hints. They can be used when running the modified binary to enable or disable netCDF features at runtime.

run the modified binary. It would be possible to link the replacement code statically via a static library rather than dynamically via a shared library. This would increase the size of the modified binary, but could reduce the load cost at runtime.

7.4. Summary

In this chapter, all the main features of CATO have been evaluated: memory sharing with MPI, as well as parallel I/O and compression with netCDF. There was some inconsistent behaviour, such as the unknown block added during compression, or the fact that parallel I/O does not work if a single process reads a data block size above a certain limit. Otherwise, the parallel I/O and compression components worked as expected and showed very good results in reducing the memory footprint of the application.

The memory reduction of the memory handling component also worked. A drawback of this component is the increase in runtime. There are many reasons why the runtime performance could have become worse, some important ones are indicated by the *Starvation, Latency, Overhead, Waiting for Contention* (SLOW) acronym (Sterling et al., 2017, p. 19f.):

Starvation: Not all components are running at full capacity. There are several possible reasons: There is not enough parallelism, the load is imbalanced, or the algorithm can only use a limited set of hardware features. It is likely that the user can avoid this degradation by using the correct process configuration. During the evaluation,

the modified binary could only be executed with a fairly small matrix size, which in combination with high process counts could lead to a load imbalance.

Latency: Messages take some time to reach their destination. This problem tends to get worse when multiple nodes are used, because the latency on the interconnect is generally worse than on the internal bus system. Therefore, swapping threads for processes leads to increased latency. The evaluation showed that for several benchmarks there was a shift in the runtime once the application was running on more than one node.

Overhead: CATO adds additional data structures and encapsulates original variables to handle the more complex communication using MPI. Even if some OpenMP structures are removed, this is probably not enough to compensate.

Waiting: The amount of waiting for resources may remain relatively the same, because OpenMP and MPI have to perform locking on the same data. Differences in the duration of each other's locking mechanism are already taken into account in the *latency* or *overhead* part.

An experimental software cache proved that there are ways to improve the runtime of a modified binary, but this requires additional work.

8. Conclusion

In this thesis, the needs and challenges of domain scientists in using HPC systems have been analysed and the decision has been made to support their use cases through an automated, tool-based approach. This takes away some of the user's responsibilities and allows them to focus on their science. To achieve this, several approaches were discussed and it was decided to create an LLVM pass, which makes all the necessary changes during compilation of the original high-level source code. The modified binary is then equipped with all the features needed to support the use cases of the domain scientists. The LLVM pass, a test environment and helper scripts have been bundled into the tool that this work is about: CATO.

8.1. Research Questions

In Section 1.2 four research questions have been formulated, and the following summary will assess whether CATO provides adequate answers. Open questions from this discussion are then reviewed, leading to the concluding section on future work.

8.1.1. Distributed Computing

OpenMP is a compiler extension for using threads on shared memory and is quite popular in ESS. With a handful of compiler directives, an application can use the entire computing capacity of a single node. In the HPC domain, a node can consist of multiple CPUs, each with multiple cores. However, if a single node is too limited for the scientist's use case and additional nodes are required, a distributed computing framework must be used in addition. Compared to OpenMP, they tend to be more complex and harder to learn than OpenMP. For example, MPI is a popular solution in HPC when distributed memory is to be used.

Research Question 1: Assuming that OpenMP is already used: How can an automatic transformation solution enable the application to make use of a distributed memory parallelisation scheme?

The evaluation was done on a benchmark that uses a stencil pattern on a regular grid. So the communication pattern is predictable, but the CATO implementation uses a mixture of active-target and passive-target one-sided MPI communication, so CATO could have handled irregular patterns as well.

Without any additional input from the domain scientist, CATO analyses the original high-level code using OpenMP and transforms it to use MPI instead. All accesses within the serial part as well as within the OpenMP kernels are automatically adjusted. The evaluation showed that this worked, as the modified application was not limited to a single node any more and `partdiff` still returned a correct result even if it was executed on several nodes.

Memory has been distributed and the memory footprint of a single process has been reduced accordingly. In Section 7.3.2 C(1-1) had a memory footprint of 1034 kB during the weak scaling benchmarks, which increased to 9105 kB for C(5-24). On average, this is only an increase of $67.8 \frac{\text{kB}}{\text{process}}$. There is some overhead caused explicitly by CATO, so some degradation in memory scaling was to be expected. CATO's overhead per process essentially scales with the number of variables accessed within the unmodified OpenMP kernel, not with their size. So even if the matrix size in this benchmark only had a dimension size of 40, this is still a good result as something similar can be expected for larger matrix sizes.

One drawback was the runtime performance overhead, which was noticeable. This is probably due to the fact that every single access to a heap variable has to be encapsulated to check whether it belongs to local memory or needs to be fetched from another process. This led to the need to reduce the size of the input problem to meet the time limits of the test systems, which conflicts with the original goal of using the sum of the memory of all participating nodes. On the other hand, the evaluation of the experimental software cache showed very promising results that the runtime limitation can be relaxed.

CATO can therefore approve the first research question of how existing OpenMP kernels can be automatically transformed to use distributed memory, but currently has still some limitations in terms of its practicality. However, this is a flaw in the current implementation of CATO, which can be mitigated.

8.1.2. Parallel Input/Output

The memory handling component distributes the allocation and initialisation of heap memory. For dynamically generated input data this works already quite well, i.e. the application generates the data itself at runtime, but there are many use cases where the input data is provided via a file. NetCDF is a popular ESS library that can perform the required file I/O. Using the default, every process would make the same serial I/O calls to update its local memory segment and the segments of the other processes. This would lead to a lot of redundant and inefficient communication calls.

Research Question 2: Assuming that serial netCDF I/O is already used: How can an automatic transformation solution transform the application to use parallel I/O with reduced runtime and memory footprint?

The netCDF interface provides operations for both serial and parallel I/O. CATO replaces all serial I/O calls and modifies the access operations so that each process

does not load the entire input file, but only its share. If a process needs to access data from another process, this is not done via the netCDF interface, but via a local access operation. This is therefore already handled by the first component of CATO's, the memory distribution component. Therefore, the parallel I/O component can use the synergy with this component to achieve a better (i.e. reduced) memory footprint. As all processes only access their share of the file concurrently, there is a potential runtime benefit. The latter requires that the file is read from a parallel FS, otherwise the parallel netCDF I/O operations will be sequenced by a single block device.

In addition, the input file must be striped across multiple I/O servers to match the number of processes used. This must be done by the user, as it is outside the control of the application, and therefore not done by CATO. CATO's help page gives some information about this, so that the user knows that he has to take care of it, and has an idea where to start.

The parallel I/O benchmark in Section 7.3.3 showed really good memory efficiency, never falling below 99% for any configuration. In comparison, the runtime efficiency scaled a bit worse, but there was still a benefit from shorter runtimes, at least on WR-Cluster. This was easy to observe because the bottleneck of the 1 GiB interconnect throttles the data transfer rate. On Levante, runtime efficiency was lower because the 100 GiB interconnect provides fast data transfer, and therefore other sources of overhead became more important. Regardless of the number of processes used, the runtime is always the same order of magnitude as the runtime of the unmodified file, which is still a success. The larger the input file and the slower the interconnect bandwidth, the better the runtime efficiency.

A problem arises when a single process has to read more than 7.1 GiB, because in this case the modified binary throws an error. As a workaround, the user can make sure to use enough processes to keep the size of their file share below the critical line. The runtime could also be improved if the interconnect acted as a bottleneck.

8.1.3. Chunking and Compression

Many use cases in ESS can benefit from using larger datasets, allowing larger areas to be simulated, finer resolution to be used, or more variables to be considered. A FS such as ZFS can already provide transparent compression capabilities. The user is unaware of this because ZFS does it automatically and decompresses a file when it is accessed again.

NetCDF can also use compression, as it has some compressors like Zlib built in, but it can also use any filter supported by its HDF5 backend. In this case, however, compression must be explicitly enabled using specific functions within the application code.

Research Question 3: Assuming that serial netCDF I/O is already used: How can an automatic transformation solution transform the application to transparently use chunking and compression?

CATO scans for netCDF variable definitions and adds the necessary functions. The evaluation showed that this worked quite well and can result in smaller file sizes: The original file has a size of 15.6 MiB and the best configuration was able to compress it down to at least 3.0 MiB (rand_norm and rand_1000 pattern) or even 0.6 MiB (zeros and lin_inc). On the downside, it is important that the user chooses the right configuration of compression parameters, as a bad configuration could actually increase the file size. This is currently not done automatically by CATO, as it depends on the use case which configuration is good (usually a balance has to be found between compression ratio and (de-)compression bandwidth). However, CATO provides a script that can be used to automatically check any number of configurations, so that the user can choose the best one based on the results.

The analysis also showed that an unknown block of memory appears between the metadata header and the actual data of the output file. The reason for this is currently unknown, but at least it does not affect the data integrity (according to `ncdump`), nor does it have a negative impact on the number of blocks allocated by the local FS, according to the output of `du`.

Since any compressor can be used for which a corresponding filter has been registered in HDF5, netCDF can use more advanced filters such as SZ or Zstandard. In this case, it is necessary to set up the filters on each system where the compressed data is to be accessed, in order to preserve the portability of the application. However, this is a general requirement of HDF5 filters that is not imposed by CATO first.

8.1.4. User Support

CATO acts conservatively so that no changes are made that could have a negative impact on data integrity. During the transformation, CATO has already inserted all available features, but has disabled them by encapsulating them behind a conditional check if a specific environment variable is set by the user at runtime. Therefore, there are some parameters that are set by environment variables that allow the user to enable a specific feature at runtime. This allows CATO to include many different features that are not necessarily interdependent. For example, the CATO' compression component and the memory handling component do not need each other and could be used independently. Therefore, the functionality of CATO can be extended by a developer without any limitation.

The primary purpose of CATO is to allow the user to quickly try out different HPC frameworks without having to learn the ropes. It can be seen as an experimental kit, where the focus is on trying out new concepts rather than using it within a productive workflow.

Research Question 4: How can an automatic transformation solution provide feedback to enable the user to comprehend the modifications?

CATO offers two main types of feedback:

Help pages: Integrated into CATO are help pages for each of the included frameworks. Currently there are help pages for a general manual on how to use CATO, and for the MPI and netCDF components. They provide information about the features and how to enable them via environment variables, as well as links to (un)official documentation and repositories of sample code. Especially linked collections of sample code can be very useful when learning a new framework, and the official developers do not always provide enough examples.

Metrics collection: Using CATO’s metrics collection feature makes it easy to compare modified binaries built with different configurations. Usually time is not an issue, but obtaining values for peak memory usage and the average share of a process in that memory usage can be more difficult for domain scientists. CATO integrates the metrics collection into the modified binary so that the user can intentionally enable it at any time.

This allows the user to quickly try out new HPC concepts. And if the tests with CATO are promising, they will also know where to get educational resources if they want to start integrating this feature into their codebase by hand to get a more optimised version that is easier to maintain.

8.2. Future Work

During the development and evaluation of CATO some points unfolded, which could be worked on further. The following Sections 8.2.1 to 8.2.3 present three potential directions for future work.

8.2.1. Runtime Performance

Improving runtime performance by reducing the overhead caused by CATO would benefit the memory handling component, and therefore the parallel I/O component could also benefit indirectly. During the evaluation, single value accesses were identified as a potential source of runtime degradation. The evaluation of an experimental software cache in Section 7.3.2 showed at least an improvement in runtime, allowing larger input sizes to be computed within acceptable runtimes. Molly uses a similar approach by bundling multiple accesses into a single communication operation, so this could be used as a starting point (Kruse, 2014). This could be further refined if, for example, additional information is used. Currently, no information about the underlying hardware is used. Collecting this could allow additional optimisations such as optimised message sizes. In addition, more information could be gathered from the AST and CFG of the application. The `llvm::BlockFrequencyInfo` class can be used to estimate how often a BB is executed, which could lead to a distinction between important and unimportant variables.

Another memory handling strategy could also be considered: instead of performing memory allocation using user-written ECs, the suitability of the new OpenMP remote

offloading directive could be explored (Patel & Doerfert, 2022). Another approach that could be used for CATO has been proposed by Wahlgren et al. (2022), who suggest creating a memory pooling system for HPC systems based on the new *Compute Express Link* (CXL) standard, which could directly interconnect the local memory of each node. This would remove the need for the memory distribution component and could provide performance benefits. However, CXL is still at an early stage of development, and what exactly it will be able to achieve will only become evident in the future. If it only works on a few HPC systems, a fallback solution will still be needed.

Another important point is that OpenMP is completely removed in the current state. The evaluation of parallel I/O has shown that in this case the best configuration is to use a single process per node, as this allows to maximise the size of the share of the input problem. To make full use of the node, CATO could use OpenMP again. In this case, more advanced solutions could be incorporated, such as partitioned MPI operations, which could be of significant benefit when using multithreaded MPI (Jammer & Bischof, 2021). However, using a hybrid parallelisation scheme may place higher demands on a good startup configuration, which will need to be found by the domain scientist (Gahvari et al., 2015).

8.2.2. Feature Coverage

There are some code patterns that do not yet work (cf. Section 7.2.1) and fixing them would improve the robustness of CATO. One problem is for example that the modified application will abort if a process tries to read a file share larger than about 7.1 GiB during parallel I/O. One possible workaround is to just read a share of a file up to a certain limit, and if a larger share is about to be accessed, then this request will be split into more smaller requests, which will then be executed sequentially.

Apart from this basic functionality, there are variants of already implemented features that are missing. For example, the memory handling can only work on limited pointer depths. And the parallel I/O component expects the original application to use `nc_get_var` or `nc_get_var` on `int` or `float` variables. At the moment it cannot handle other datatypes. Also, if the original application already reads only a part of the file instead of the whole file (e.g. by using `nc_*_vara_*`, `nc_*_var1_*`, `nc_*_varm_*` or `nc_*_vars_*`), they are simply ignored. And the compression component can currently only use the *quantize* and Zlib filters, and has not yet included the Blosc, bzip2, Szip or Zstandard compressors.

8.2.3. Usability Improvement

Currently, the only way the user can influence the transformation process is through environment variables, which currently do not allow a feature to be restricted to a particular variable. An example would be to use different compression configurations on different variables. At the moment the user can only set this configuration for all variables. One way to do this would be to create new pragmas which can then be used to annotate individual variables within the source code (Mishra et al., 2020). This would

disable the ability to switch an optional feature on or off at runtime, as using pragmas limits the user’s influence to compilation time. However, this would also allow the user to fine-tune the replacement, for example by ignoring heap variables that the user knows from domain knowledge are not important for performance. In addition, more CATO functionality could be integrated so that the analysis phase provides more information that can be exploited.

To facilitate the development and integration of new ECs, which are currently somehow hard-coded into CATO (replacement code is already outsourced to C++ files), associated identification snippets could be used, as in MARTINI(Johnson et al., 2022). It uses snippets to find specific code and then uses the appropriate matching snippet to replace it. In the case of MARTINI, templates are used for both matching and replacement, which would probably not be sufficient for CATO, where the detection of relevant code snippets usually requires more abstract information, which may be difficult to capture in a template.

Regardless of how the replacement is done, the code recognition strategy could also use a more sophisticated approach. Currently it is based on the stencil pattern. Evaluations of PGAS (cf. Section 6.1.2) have criticised that performance degrades quickly if data locality is not taken into account. So this is a problem that could also apply to CATO if a different communication pattern is used. CATO requires additional replacement and data placement strategies for other Dwarves to avoid performance degradation due to misplaced data. In addition, CATO needs to detect which Dwarf an application most resembles, assuming it has a prominent pattern. A simple approach would be to use the expertise of the domain scientist, who is likely to know the communication pattern of his application. So he could give CATO a hint about what kind of communication pattern is being used. This only works if the domain scientist wrote the software being used. If he is not the developer, but just a user, he is less likely to be able to identify the communication pattern correctly. In this case, it would be more useful if CATO could detect the communication itself. Pattern recognition can become arbitrarily complicated, as this is a large field of research in itself.

8.3. Summary

The research questions established at the beginning of this thesis (cf. Section 1.2) have been evaluated according to the results of the evaluation in Chapter 7. The primary goal of CATO’s is to provide an automatic solution to these questions for a domain scientist, so that he can easily try out HPC concepts that are quite complicated to learn (making a quick manual evaluation difficult) or that he didn’t even know existed. He only needs to use OpenMP and netCDF together if he wants to make use of all components of CATO, but they can also be used independent from each other (with reduced efficiency). OpenMP is comparatively easy to use as well as the serial I/O interface of netCDF. CATO can then automatically analyse and transform the code to replace OpenMP with MPI and serial netCDF with parallel netCDF. So just by building their application with CATO it can now be executed on multiple nodes and can utilise a parallel FS at its full

potential. In addition the user can enable the optional compression component, so that the size of the written data can be reduced based on the data properties and chosen compression configuration.

The poor runtime performance of the memory allocation component is a drawback, but otherwise all questions could be answered satisfactorily.

Bibliography

Cited Publications

- Afzal, A., Hager, G. & Wellein, G. (2021). Analytic modeling of idle waves in parallel programs: Communication, cluster topology, and noise impact. *Lecture notes in computer science* (pp. 351–371). Springer International Publishing. https://doi.org/10.1007/978-3-030-78713-4_19. (Cit. on p. 101)
- Agrawal, A. & Choudhary, A. (2016). Perspective: Materials informatics and big data: Realization of the “fourth paradigm” of science in materials science. *APL Materials*, 4(5), 053208. <https://doi.org/10.1063/1.4946894> (cit. on p. 15)
- Amaral, V., Norberto, B., Goulão, M., Aldinucci, M., Benkner, S., Bracciali, A., Carreira, P., Celms, E., Correia, L., Grelck, C., Karatza, H. D., Kessler, C. W., Kilpatrick, P., Martiniano, H. F. M. C., Mavridis, I., Pllana, S., Respício, A., Simão, J., Veiga, L. & Visa, A. (2020). Programming languages for data-intensive HPC applications: A systematic mapping study. *Parallel Comput.*, 91. <https://doi.org/10.1016/j.parco.2019.102584> (cit. on p. 26)
- Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, 30, 483–485. <https://doi.org/10.1145/1465482.1465560> (cit. on p. 153)
- Amza, C., Cox, A. L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W. & Zwaenepoel, W. (1996). TreadMarks: Shared memory computing on networks of workstations. *Computer*, 29(2), 18–28. <https://doi.org/10.1109/2.485843> (cit. on p. 143)
- an Mey, D. (2008). Matrix Multiplication using MPI (D. an Mey, Ed.) [Last accessed: 2023-04-13]. https://blog.rwth-aachen.de/hpc_import_20210107/attachments/3475011/3703298.pdf. (Cit. on p. 100)
- Appel, A. W. (1998). *Modern compiler implementation in C*. Cambridge University Press. (Cit. on pp. 119, 121, 220).
- Arora, R. & Ba, T. N. (2019). Semi-automatic Code Modernization for Optimal Parallel I/O. *Communications in computer and information science* (pp. 140–154). Springer Singapore. https://doi.org/10.1007/978-981-13-7729-7_10. (Cit. on p. 147)
- Arora, R., Olaya, J. C. & Gupta, M. (2014). A Tool for Interactive Parallelization. In S. A. Lathrop & J. Alameda (Eds.), *Annual conference of the extreme science and engineering discovery environment, XSEDE '14, atlanta, ga, USA - july 13*

- 18, 2014 (51:1–51:8). ACM Press. <https://doi.org/10.1145/2616498.2616558>. (Cit. on p. 147)
- Arvanitou, E., Ampatzoglou, A., Chatzigeorgiou, A. & Carver, J. C. (2021). Software engineering practices for scientific software development: A systematic mapping study. *J. Syst. Softw.*, 172, 110848. <https://doi.org/10.1016/j.jss.2020.110848> (cit. on p. 26)
- Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W. & Yelick, K. A. (2006, December 18). *The Landscape of Parallel Computing Research: A View from Berkeley* (tech. rep.). Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley. (Cit. on p. 80).
- Bacon, D. F., Graham, S. L. & Sharp, O. J. (1994). Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4), 345–420. <https://doi.org/10.1145/197405.197406> (cit. on p. 56)
- Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S. et al. (1993, July). *The nas parallel benchmarks* (tech. rep.). NASA Ames Research Center. (Cit. on pp. 78, 82).
- Balakrishnan, K. J. & Toubia, N. A. (2007). Relationship between entropy and test data compression. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 26(2), 386–395. <https://doi.org/10.1109/TCAD.2006.882600> (cit. on p. 69)
- Balart, J., Duran, A., González, M., Martorell, X., Ayguadé, E. & Labarta, J. (2004). Nanos mercurium: a research compiler for OpenMP. *Proceedings of the European Workshop on OpenMP*, 8, 56 (cit. on p. 56).
- Barbalace, A., Ravindran, B. & Katz, D. (2014). Popcorn: A replicated-kernel os based on linux. *Proceedings of the Linux Symposium, Ottawa, Canada* (cit. on p. 142).
- Barrett, B. W. & Hemmert, K. S. (2009). An application based MPI message throughput benchmark. *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA*, 1–8. <https://doi.org/10.1109/CLUSTER.2009.5289198> (cit. on p. 78)
- Bartz, C., Chasapis, K., Kuhn, M., Nerge, P. & Ludwig, T. (2015). A Best Practice Analysis of HDF5 and NetCDF-4 Using Lustre. In J. M. Kunkel & T. Ludwig (Eds.), *High performance computing - 30th international conference, ISC high performance 2015, frankfurt, germany, july 12-16, 2015, proceedings* (pp. 274–281). Springer. https://doi.org/10.1007/978-3-319-20119-1_20. (Cit. on pp. 104, 133, 134)
- Basumallik, A. & Eigenmann, R. (2005). Towards automatic translation of OpenMP to MPI. *Proceedings of the 19th Annual International Conference on Supercomputing*, 189–198. <https://doi.org/10.1145/1088149.1088174> (cit. on pp. 61, 148)
- Ben-Nun, T., Jakobovits, A. S. & Hoefler, T. (2018). Neural code comprehension: A learnable representation of code semantics. *CoRR*, abs/1806.07336. <http://arxiv.org/abs/1806.07336> (cit. on p. 99)

- Bispo, J. & Cardoso, J. M. P. (2020). Clava: C/C++ source-to-source compilation using LARA. *SoftwareX*, 12, 100565. <https://doi.org/10.1016/j.softx.2020.100565> (cit. on p. 56)
- Blesel, M., Kuhn, M. & Squar, J. (2021). heimdallr: Improving Compile Time Correctness Checking for Message Passing with Rust. In H. Jagode, H. Anzt, H. Ltaief & P. Luszczek (Eds.), *High performance computing - ISC high performance digital 2021 international workshops, frankfurt am main, germany, june 24 - july 2, 2021, revised selected papers* (pp. 199–211). Springer. https://doi.org/10.1007/978-3-030-90539-2_13
- Bondhugula, U., Hartono, A., Ramanujam, J. & Sadayappan, P. (2008). Pluto: A practical and fully automatic polyhedral program optimization system. *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)* (cit. on p. 56).
- Braibant, S., Giacomelli, G. & Spurio, M. (2012). *Particles and fundamental interactions*. Springer Netherlands. <https://doi.org/10.1007/978-94-007-2464-8>. (Cit. on p. 82)
- Brune, S., Buschow, S. & Friederichs, P. (2020). Observations and high-resolution simulations of convective precipitation organization over the tropical atlantic. *Quarterly Journal of the Royal Meteorological Society*, 146(729), 1545–1563. <https://doi.org/10.1002/qj.3751> (cit. on p. 20)
- Buyya, R., Cortes, T. & Jin, H. (2001). Single system image. *Int. J. High Perform. Comput. Appl.*, 15(2), 124–135. <https://doi.org/10.1177/109434200101500205> (cit. on p. 142)
- Câmpeanu, C., Salomaa, K. & Yu, S. (2003). A formal study of practical regular expressions. *Int. J. Found. Comput. Sci.*, 14(6), 1007–1018. <https://doi.org/10.1142/S012905410300214X> (cit. on p. 36)
- Cappello, F., Di, S., Li, S., Liang, X., Gok, A. M., Tao, D., Yoon, C. H., Wu, X., Alexeev, Y. & Chong, F. T. (2019). Use cases of lossy compression for floating-point data in scientific data sets. *IJHPCA*, 33(6). <https://doi.org/10.1177/1094342019853336> (cit. on pp. 23, 69)
- Chamberlain, B. L., Callahan, D. & Zima, H. P. (2007). Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3), 291–312. <https://doi.org/10.1177/1094342007078442> (cit. on p. 144)
- Charles, P., Grothoff, C., Saraswat, V. A., Donawa, C., Kielstra, A., Ebcioğlu, K., von Praun, C. & Sarkar, V. (2005). X10: an object-oriented approach to non-uniform cluster computing. In R. E. Johnson & R. P. Gabriel (Eds.), *Proceedings of the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA 2005, october 16-20, 2005, san diego, ca, USA* (pp. 519–538). ACM. <https://doi.org/10.1145/1094811.1094852>. (Cit. on p. 144)
- Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S. & Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*, 44–54. <https://doi.org/10.1109/IISWC.2009.5306797> (cit. on pp. 80, 82)

- Chen, D., Rojas, M., Samset, B., Cobb, K., Diongue Niang, A., Edwards, P., Emori, S., Faria, S., Hawkins, E., Hope, P., Huybrechts, P., Meinshausen, M., Mustafa, S., Plattner, G.-K. & Tréguier, A.-M. (2021). Framing, context, and methods. In V. Masson-Delmotte, P. Zhai, A. Pirani, S. Connors, C. Péan, S. Berger, N. Caud, Y. Chen, L. Goldfarb, M. Gomis, M. Huang, K. Leitzell, E. Lonnoy, J. Matthews, T. Maycock, T. Waterfield, O. Yelekçi, R. Yu & B. Zhou (Eds.), *Climate change 2021: The physical science basis. contribution of working group i to the sixth assessment report of the intergovernmental panel on climate change* (pp. 147–286). Cambridge University Press. <https://doi.org/10.1017/9781009157896.003>. (Cit. on p. 20)
- Chester, D. G., Groves, T. L., Hammond, S. D., Law, T., Wright, S. A., Smedley-Stevenson, R. P., Fahmy, S. A., Mudalidge, G. R. & Jarvis, S. A. (2021). Stress-bench: A configurable full system network and I/O benchmark framework. *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021*. <https://doi.org/10.1109/HPEC49654.2021.9774494> (cit. on p. 78)
- Christen, M., Schenk, O. & Burkhart, H. (2011). PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, 676–687. <https://doi.org/10.1109/IPDPS.2011.70> (cit. on pp. 145, 146)
- Chunduri, S., Groves, T. L., Mendygral, P., Austin, B., Balma, J., Kandalla, K., Kumaran, K., Lockwood, G. K., Parker, S., Warren, S., Wichmann, N. & Wright, N. J. (2019). Gpcnet: Designing a benchmark suite for inducing and measuring contention in HPC networks. In M. Tauber, P. Balaji & A. J. Peña (Eds.), *Proceedings of the international conference for high performance computing, networking, storage and analysis, SC 2019, denver, colorado, usa, november 17-19, 2019* (42:1–42:33). ACM. <https://doi.org/10.1145/3295500.3356215>. (Cit. on p. 78)
- Cifuentes, C. (1994). *Reverse compilation techniques* (Doctoral dissertation). Queensland University of Technology. (Cit. on p. 107).
- Colella, P. Defining software requirements for scientific computing. In: presentation, 2004 (cit. on p. 80).
- Coym, J. (2021, October). *Analysis of elastic Cloud solutions in an HPC Environment* (Master’s Thesis). Universität Hamburg. (Cit. on p. 27).
- Das, M., Lerner, S. & Seigle, M. (2002). ESP: path-sensitive program verification in polynomial time. In J. Knoop & L. J. Hendren (Eds.), *Proceedings of the 2002 ACM SIGPLAN conference on programming language design and implementation (pldi), berlin, germany, june 17-19, 2002* (pp. 57–68). ACM. <https://doi.org/10.1145/512529.512538>. (Cit. on p. 44)
- Dave, C., Bae, H., Min, S., Lee, S., Eigenmann, R. & Midkiff, S. P. (2009). Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(11), 36–42. <https://doi.org/10.1109/MC.2009.385> (cit. on p. 56)

- Dave, M. A. (2003). Compiler verification: A bibliography. *ACM SIGSOFT Softw. Eng. Notes*, 28(6), 2. <https://doi.org/10.1145/966221.966235> (cit. on p. 125)
- Delaunay, X., Courtois, A. & Gouillon, F. (2019). Evaluation of lossless and lossy algorithms for the compression of scientific datasets in netcdf-4 or hdf5 files. *Geoscientific Model Development*, 12(9), 4099–4113 (cit. on p. 149).
- Di, S. & Cappello, F. (2016). Fast error-bounded lossy HPC data compression with SZ. *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, 730–739. <https://doi.org/10.1109/IPDPS.2016.11> (cit. on pp. 71, 105)
- Di, S., Guo, H., Pershey, E., Snir, M. & Cappello, F. (2019). Characterizing and understanding HPC job failures over the 2k-day life of IBM bluegene/q system. *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, 473–484. <https://doi.org/10.1109/DSN.2019.00055> (cit. on p. 114)
- Dinan, J., Balaji, P., Buntinas, D., Goodell, D., Gropp, W. & Thakur, R. (2016). An implementation and evaluation of the MPI 3.0 one-sided communication interface. *Concurrency and Computation: Practice and Experience*, 28(17), 4385–4404. <https://doi.org/10.1002/cpe.3758> (cit. on p. 65)
- Douville, H., Raghavan, K., Renwick, J., Allan, R., Arias, P., Barlow, M., Cerezo-Mota, R., Cherchi, A., Gan, T., Gergis, J., Jiang, D., Khan, A., Pokam Mba, W., Rosenfeld, D., Tierney, J. & Zolina, O. (2021). Water cycle changes. In V. Masson-Delmotte, P. Zhai, A. Pirani, S. Connors, C. Péan, S. Berger, N. Caud, Y. Chen, L. Goldfarb, M. Gomis, M. Huang, K. Leitzell, E. Lonnoy, J. Matthews, T. Maycock, T. Waterfield, O. Yelekçi, R. Yu & B. Zhou (Eds.), *Climate change 2021: The physical science basis. contribution of working group i to the sixth assessment report of the intergovernmental panel on climate change* (pp. 1055–1210). Cambridge University Press. <https://doi.org/10.1017/9781009157896.010>. (Cit. on p. 19)
- Droste, A., Kuhn, M. & Ludwig, T. (2015). Mpi-checker: Static analysis for MPI. In H. Finkel (Ed.), *Proceedings of the second workshop on the LLVM compiler infrastructure in hpc, LLVM 2015, austin, texas, usa, november 15, 2015* (3:1–3:10). ACM. <https://doi.org/10.1145/2833157.2833159>. (Cit. on pp. 35, 40)
- D’Silva, V., Payer, M. & Song, D. X. (2015). The correctness-security gap in compiler optimization. *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*, 73–87. <https://doi.org/10.1109/SPW.2015.33> (cit. on p. 125)
- Dursun, H., Nomura, K.-I., Peng, L., Seymour, R., Wang, W., Kalia, R. K., Nakano, A. & Vashishta, P. (2009). A multilevel parallelization framework for high-order stencil computations. *European Conference on Parallel Processing*, 642–653 (cit. on p. 100).
- Duwe, K., Lüttgau, J., Mania, G., Squar, J., Fuchs, A., Kuhn, M., Betke, E. & Ludwig, T. (2020). State of the Art and Future Trends in Data Reduction for High-Performance Computing. *Supercomput. Front. Innov.*, 7(1), 4–36. <https://doi.org/10.14529/jsfi200101> (cit. on p. 71)

- Engelke, A. & Schulz, M. (2020a). Instrew: Leveraging LLVM for high performance dynamic binary instrumentation. In S. Nagarakatte, A. Baumann & B. Kasikci (Eds.), *VEE '20: 16th ACM SIGPLAN/SIGOPS international conference on virtual execution environments, virtual event [lausanne, switzerland], march 17, 2020* (pp. 172–184). ACM. <https://doi.org/10.1145/3381052.3381319>. (Cit. on pp. 46, 107)
- Engelke, A. & Schulz, M. (2020b). Robust practical binary optimization at run-time using LLVM. *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. <https://doi.org/10.1109/llvmhpcchipar51896.2020.00011> (cit. on p. 46)
- Esmailzadeh, H., Blem, E. R., Amant, R. S., Sankaralingam, K. & Burger, D. (2012). Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3), 122–134. <https://doi.org/10.1109/MM.2012.17> (cit. on p. 16)
- Fonseca, A., Cabral, B., Rafael, J. & Correia, I. (2016). Automatic parallelization: Executing sequential programs on a task-based parallel runtime. *Int. J. Parallel Program.*, 44(6), 1337–1358. <https://doi.org/10.1007/s10766-016-0426-5> (cit. on p. 148)
- Frigo, M., Leiserson, C. E. & Randall, K. H. (1998). The implementation of the cilk-5 multithreaded language. In J. W. Davidson, K. D. Cooper & A. M. Berman (Eds.), *Proceedings of the ACM SIGPLAN '98 conference on programming language design and implementation (pldi), montreal, canada, june 17-19, 1998* (pp. 212–223). ACM. <https://doi.org/10.1145/277650.277725>. (Cit. on p. 144)
- Furtunato, A. F. A., Georgiou, K., Eder, K. & de Souza, S. X. (2020). When parallel speedups hit the memory wall. *IEEE Access*, 8, 79225–79238. <https://doi.org/10.1109/ACCESS.2020.2990418> (cit. on p. 154)
- Gahvari, H., Schulz, M. & Yang, U. M. (2015). An approach to selecting thread + process mixes for hybrid MPI + openmp applications. *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*, 418–427. <https://doi.org/10.1109/CLUSTER.2015.64> (cit. on p. 192)
- Gerstenberger, R., Besta, M. & Hoeﬂer, T. (2013). Enabling highly-scalable remote memory access programming with MPI-3 one sided. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*, 1–12. <https://doi.org/10.1145/2503210.2503286> (cit. on p. 65)
- Einiges über RMA und Zeitkosten (abstrakte Kostenfunktion, die evtl. für die Diss praktisch werden könnte)
- Giorgetta, M. A., Sawyer, W., Lapillonne, X., Adamidis, P., Alexeev, D., Clément, V., Dietlicher, R., Engels, J. F., Esch, M., Franke, H., Frauen, C., Hannah, W. M., Hillman, B. R., Kornblueh, L., Marti, P., Norman, M. R., Pincus, R., Rast, S., Reinert, D., ... Stevens, B. (2022). The icon-a model for direct qbo simulations on gpus (version icon-cscs:baf28a514). *Geoscientific Model Development*, 15(18), 6985–7016. <https://doi.org/10.5194/gmd-15-6985-2022> (cit. on p. 29)

- Grosser, T., Größlinger, A. & Lengauer, C. (2012). Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.*, 22(4). <https://doi.org/10.1142/S0129626412500107> (cit. on p. 147)
- Guo, B., Wu, Y., Wang, C., Bridges, M. J., Ottoni, G., Vachharajani, N., Chang, J. & August, D. I. (2006). Selective runtime memory disambiguation in a dynamic binary translator. In A. Mycroft & A. Zeller (Eds.), *Compiler construction, 15th international conference, CC 2006, held as part of the joint european conferences on theory and practice of software, ETAPS 2006, vienna, austria, march 30-31, 2006, proceedings* (pp. 65–79). Springer. https://doi.org/10.1007/11688839_6. (Cit. on p. 127)
- Gustafson, J. L. (1988). Reevaluating amdahl’s law. *Commun. ACM*, 31(5), 532–533. <https://doi.org/10.1145/42411.42415> (cit. on p. 154)
- Hamidouche, K., Falcou, J. & Etiemble, D. (2011). A framework for an automatic hybrid mpi+ openmp code generation. *Proceedings of the 19th High Performance Computing Symposia*, 48–55. Retrieved April 10, 2019, from <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.461.1401&rep=rep1&type=pdf> (cit. on p. 147)
- Hammond, J. R., Ghosh, S. & Chapman, B. M. (2014). Implementing OpenSHMEM Using MPI-3 One-Sided Communication. In S. Poole, O. Hernandez & P. Shamis (Eds.), *Openshmem and related technologies. experiences, implementations, and tools: First workshop, openshmem 2014, annapolis, md, usa, march 4-6, 2014. proceedings* (pp. 44–58). Springer International Publishing. https://doi.org/10.1007/978-3-319-05215-1_4. (Cit. on p. 142)
- Hansel, G., Perrin, D. & Simon, I. (1992). Compression and entropy. In A. Finkel & M. Jantzen (Eds.), *STACS 92, 9th annual symposium on theoretical aspects of computer science, cachan, france, february 13-15, 1992, proceedings* (pp. 515–528). Springer. https://doi.org/10.1007/3-540-55210-3_209. (Cit. on p. 69)
- Hauschildt, P. H. & Baron, E. (2006). A 3D radiative transfer framework. *Astronomy and Astrophysics*, 451(1), 273–284. <https://doi.org/10.1051/0004-6361:20053846> (cit. on p. 20)
- He, H. (2015, September 28). *Using openmp at nersc* [Online]. <https://openmpcon.org/wp-content/uploads/openmpcon2015-helen-he-nersc.pdf>. (Cit. on pp. 61, 62)
- Healy, P. D., Lynn, T., Barrett, E. & Morrison, J. P. (2016). Single system image: A survey. *J. Parallel Distributed Comput.*, 90-91, 35–51. <https://doi.org/10.1016/j.jpdc.2016.01.004> (cit. on p. 142)
- Heing-Becker, M. (2019). *Verification of one-sided MPI communication code using static analysis in LLVM* (Master’s Thesis). Universität Hamburg. (Cit. on pp. 43, 44).
- Hennessy, J. L. & Patterson, D. A. (2012). *Computer architecture - A quantitative approach, 5th edition*. Morgan Kaufmann. (Cit. on p. 18).
- Hilbrich, T., Protze, J., Schulz, M., de Supinski, B. R. & Müller, M. S. (2013). MPI runtime error detection with MUST: advances in deadlock detection. *Sci. Program.*, 21(3-4), 109–121. <https://doi.org/10.3233/SPR-130368> (cit. on p. 35)
- Hind, M. (2001). Pointer analysis: Haven’t we solved this problem yet? In J. Field & G. Snelting (Eds.), *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop*

- on program analysis for software tools and engineering, paste'01, snowbird, utah, usa, june 18-19, 2001* (pp. 54–61). ACM. <https://doi.org/10.1145/379605.379665>. (Cit. on p. 127)
- Hoefler, T., Dinan, J., Thakur, R., Barrett, B., Balaji, P., Gropp, W. & Underwood, K. (2015). Remote Memory Access Programming in MPI-3. *ACM Transactions on Parallel Computing*, 2(2), 1–26. <https://doi.org/10.1145/2780584> (cit. on p. 64)
- Hoeflinger, J. P. (2006). Extending openmp to clusters. *White Paper, Intel Corporation* (cit. on p. 143).
- Hong, S., Chafi, H., Sedlar, E. & Olukotun, K. (2012). Green-marl: A DSL for easy and efficient graph analysis. In T. Harris & M. L. Scott (Eds.), *Proceedings of the 17th international conference on architectural support for programming languages and operating systems, ASPLOS 2012, london, uk, march 3-7, 2012* (pp. 349–362). ACM. <https://doi.org/10.1145/2150976.2151013>. (Cit. on pp. 145, 146)
- Hu, X., Wang, F., Li, W., Li, J. & Guan, H. (2019). QZFS: QAT accelerated compression in file system for application agnostic and cost efficient data storage. In D. Malkhi & D. Tsafir (Eds.), *2019 USENIX annual technical conference, USENIX ATC 2019, renton, wa, usa, july 10-12, 2019* (pp. 163–176). USENIX Association. <https://www.usenix.org/conference/atc19/presentation/hu-xiaokang>. (Cit. on p. 148)
- Hung, N. Q. V., Jeung, H. & Aberer, K. (2013). An evaluation of model-based approaches to sensor data compression. *IEEE Trans. Knowl. Data Eng.*, 25(11), 2434–2447. <https://doi.org/10.1109/TKDE.2012.237> (cit. on p. 148)
- Ieee standard for floating-point arithmetic. (2019). *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 1–84. <https://doi.org/10.1109/IEEESTD.2019.8766229> (cit. on p. 69)
- Inness, A., Ades, M., Agustí-Panareda, A., Barré, J., Benedictow, A., Blechschmidt, A.-M., Dominguez, J. J., Engelen, R., Eskes, H., Flemming, J., Huijnen, V., Jones, L., Kipling, Z., Massart, S., Parrington, M., Peuch, V.-H., Razinger, M., Remy, S., Schulz, M. & Suttie, M. (2019). The CAMS reanalysis of atmospheric composition. *Atmospheric Chemistry and Physics*, 19(6), 3515–3556. <https://doi.org/10.5194/acp-19-3515-2019> (cit. on p. 70)
- Jammer, T. & Bischof, C. H. (2021). Automatic partitioning of MPI operations in mpi+openmp applications. In H. Jagode, H. Anzt, H. Ltaief & P. Luszczek (Eds.), *High performance computing - ISC high performance digital 2021 international workshops, frankfurt am main, germany, june 24 - july 2, 2021, revised selected papers* (pp. 191–198). Springer. https://doi.org/10.1007/978-3-030-90539-2_12. (Cit. on p. 192)
- Jayatilaka, T., Ueno, H., Georgakoudis, G., Park, E. & Doerfert, J. (2021). Towards compile-time-reducing compiler optimization selection via machine learning. In F. Silla & O. Marques (Eds.), *ICPP workshops 2021: 50th international conference on parallel processing, virtual event / lemont (near chicago), il, usa, august 9-12, 2021* (23:1–23:6). ACM. <https://doi.org/10.1145/3458744.3473355>. (Cit. on p. 21)
- Jhala, R. & Majumdar, R. (2009). Software model checking. *ACM Comput. Surv.*, 41(4), 21:1–21:54. <https://doi.org/10.1145/1592434.1592438> (cit. on p. 44)

- Johnson, A., Coti, C., Malony, A. D. & Doerfert, J. (2022). MARTINI: the little match and replace tool for automatic application rewriting with code examples. In J. Cano & P. Trinder (Eds.), *Euro-par 2022: Parallel processing - 28th international conference on parallel and distributed computing, glasgow, uk, august 22-26, 2022, proceedings* (pp. 19–34). Springer. https://doi.org/10.1007/978-3-031-12597-3_2. (Cit. on pp. 36, 193)
- Jordan, H., Pellegrini, S., Thoman, P., Kofler, K. & Fahringer, T. (2013). INSPIRE: the insieme parallel intermediate representation. In C. Fensch, M. F. P. O’Boyle, A. Sez nec & F. Bodin (Eds.), *Proceedings of the 22nd international conference on parallel architectures and compilation techniques, edinburgh, united kingdom, september 7-11, 2013* (pp. 7–17). IEEE Computer Society. <https://doi.org/10.1109/PACT.2013.6618799>. (Cit. on p. 56)
- Jr., G. L. S. (2006). Parallel programming and parallel abstractions in fortress. In M. Hagiya & P. Wadler (Eds.), *Functional and logic programming, 8th international symposium, FLOPS 2006, fuji-susono, japan, april 24-26, 2006, proceedings* (p. 1). Springer. https://doi.org/10.1007/11737414_1. (Cit. on p. 144)
- Kapre, N. & DeHon, A. (2007). Optimistic parallelization of floating-point accumulation. *18th IEEE Symposium on Computer Arithmetic (ARITH-18 2007), 25-27 June 2007, Montpellier, France*, 205–216. <https://doi.org/10.1109/ARITH.2007.25> (cit. on p. 131)
- Keiff, M., Voigt, F., Fuchs, A., Kuhn, M., Squar, J. & Ludwig, T. (2022). Automated performance analysis tools framework for HPC programs. In M. Cristani, C. Toro, C. Zanni-Merk, R. J. Howlett & L. C. Jain (Eds.), *Knowledge-based and intelligent information & engineering systems: Proceedings of the 26th international conference kes-2022, verona, italy and virtual event, 7-9 september 2022* (pp. 1067–1076). Elsevier. <https://doi.org/10.1016/j.procs.2022.09.162>
- Kirchner, K. & Rosenthaler, S. (2017). Bin2llvm: Analysis of binary programs using LLVM intermediate representation. *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*, 45:1–45:7. <https://doi.org/10.1145/3098954.3103152> (cit. on p. 107)
- Klöwer, M., Razing er, M., Dominguez, J. J., Düben, P. D. & Palmer, T. N. (2021). Compressing atmospheric data into its real information content. *Nature Computational Science*, 1(11), 713–724. <https://doi.org/10.1038/s43588-021-00156-2> (cit. on p. 70)
- Komatsu, K., Gomi, A., Egawa, R., Takahashi, D., Suda, R. & Takizawa, H. (2020). Xevolver: A code transformation framework for separation of system-awareness from application codes. *Concurr. Comput. Pract. Exp.*, 32(7). <https://doi.org/10.1002/cpe.5577> (cit. on p. 56)
- Křoustek, J. (2014, November 17). *Retargetable analysis of machine code* (Doctoral dissertation). Brno, FIT BUT. (Cit. on p. 107).
- Kruse, M. (2014). Introducing molly: Distributed memory parallelization with LLVM. *CoRR*, abs/1409.2088. <http://arxiv.org/abs/1409.2088> (cit. on pp. 147, 191)

- Kuhn, M., Kunkel, J. M. & Ludwig, T. (2016). Data compression for climate data. *Supercomput. Front. Innov.*, 3(1), 75–94. <https://doi.org/10.14529/jsfi160105> (cit. on pp. 23, 148)
- Kuhn, M., Plehn, J., Alforov, Y. & Ludwig, T. (2020). Improving Energy Efficiency of Scientific Data Compression with Decision Trees. *ENERGY 2020: The Tenth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, 17–23. https://www.thinkmind.org/index.php?view=article&articleid=energy_2020_1_40_30038 (cit. on p. 24)
- Kumar, S. & Blocksome, M. (2014). Scalable mpi-3.0 rma on the blue gene/q supercomputer. *Proceedings of the 21st European MPI Users' Group Meeting*, 7:7–7:12. <https://doi.org/10.1145/2642769.2642778> (cit. on p. 100)
- Kusano, K., Sato, M., Hosomi, T. & Seo, Y. (2001). The omni openmp compiler on the distributed shared memory of cenju-4. In R. Eigenmann & M. Voss (Eds.), *Openmp shared memory parallel programming, international workshop on openmp applications and tools, WOMPAT 2001, west lafayette, in, usa, july 30-31, 2001 proceedings* (pp. 20–30). Springer. https://doi.org/10.1007/3-540-44587-0_3. (Cit. on p. 142)
- Landi, W. & Ryder, B. G. (1991). Pointer-induced aliasing: A problem classification. In D. S. Wise (Ed.), *Conference record of the eighteenth annual ACM symposium on principles of programming languages, orlando, florida, usa, january 21-23, 1991* (pp. 93–103). ACM Press. <https://doi.org/10.1145/99583.99599>. (Cit. on pp. 37, 127)
- Lattner, C. & Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, 75–88. <https://doi.org/10.1109/CGO.2004.1281665> (cit. on p. 117)
- Lawrencea, B., Maynardb, C., Turnerc, A., Guoc, X. & Sloan-Murphyc, D. (2017). Parallel i/o performance benchmarking and investigation on multiple hpc architectures. (Cit. on p. 104).
- Lee, J., Sato, M. & Boku, T. (2007). Design and implementation of openmpd: An openmp-like programming language for distributed memory systems. In B. M. Chapman, W. Zheng, G. R. Gao, M. Sato, E. Ayguadé & D. Wang (Eds.), *A practical programming model for the multi-core era, 3rd international workshop on openmp, IWOMP 2007, beijing, china, june 3-7, 2007, proceedings* (pp. 143–147). Springer. https://doi.org/10.1007/978-3-540-69303-1_15. (Cit. on p. 147)
- Lee, S. & Vetter, J. S. (2014). Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In B. Plale, M. Ripeanu, F. Cappello & D. Xu (Eds.), *The 23rd international symposium on high-performance parallel and distributed computing, hpdc'14, vancouver, bc, canada - june 23 - 27, 2014* (pp. 115–120). ACM. <https://doi.org/10.1145/2600212.2600704>. (Cit. on p. 56)
- Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W., Gupta, A., Hennessy, J. L., Horowitz, M. & Lam, M. S. (1992). The stanford dash multiprocessor. *Computer*, 25(3), 63–79. <https://doi.org/10.1109/2.121510> (cit. on p. 142)

- Li, W., Luo, X., Zhang, Y., Meng, Q. & Ren, F. (2022). Crossdbt: An llvm-based user-level dynamic binary translation emulator. In J. Cano & P. Trinder (Eds.), *Euro-par 2022: Parallel processing - 28th international conference on parallel and distributed computing, glasgow, uk, august 22-26, 2022, proceedings* (pp. 3–18). Springer. https://doi.org/10.1007/978-3-031-12597-3_1. (Cit. on p. 46)
- Lindstrom, P. (2014). Fixed-rate compressed floating-point arrays. *IEEE Trans. Vis. Comput. Graph.*, 20(12), 2674–2683. <https://doi.org/10.1109/TVCG.2014.2346458> (cit. on p. 105)
- Lockwood, G., Hazen, D., Koziol, Q., Canon, R., Antypas, K., Balewski, J., Balthaser, N., Bhimji, W., Botts, J., Broughton, J., Butler, T., Butler, G., Cheema, R., Daley, C., Declerck, T., Gerhardt, L., Hurlbert, W., Kallback-Rose, K., Leak, S., ... Wright, N. (2017, October). *Storage 2020: A vision for the future of HPC storage* (tech. rep.). Office of Scientific; Technical Information (OSTI). <https://doi.org/10.2172/1632124>. (Cit. on p. 17)
- Lottiaux, R., Gallard, P., Vallée, G., Morin, C. & Boissinot, B. (2005). Openmosix, openssi and kerrighed: A comparative study. *5th International Symposium on Cluster Computing and the Grid (CCGrid 2005), 9-12 May, 2005, Cardiff, UK*, 1016–1023. <https://doi.org/10.1109/CCGRID.2005.1558672> (cit. on p. 142)
- Lu, W., Shan, B., Raut, E., Meng, J., Araya-Polo, M., Doerfert, J., Malik, A. M. & Chapman, B. M. (2022). Towards efficient remote openmp offloading. In M. Klemm, B. R. de Supinski, J. Klinkenberg & B. Neth (Eds.), *Openmp in a modern world: From multi-device support to meta programming - 18th international workshop on openmp, IWOMP 2022, chattanooga, tn, usa, september 27-30, 2022, proceedings* (pp. 17–31). Springer. https://doi.org/10.1007/978-3-031-15922-0_2. (Cit. on p. 143)
- Ludwig, T. & Squar, J. (2023). Editorial Themenheft Digitale Landwirtschaft. *Informatik Spektrum*, 46(1), 1–2. <https://doi.org/10.1007/s00287-023-01521-3>
- Lüttgau, J., Kuhn, M., Duwe, K., Alforov, Y., Betke, E., Kunkel, J. M. & Ludwig, T. (2018). Survey of storage systems for high-performance computing. *Supercomput. Front. Innov.*, 5(1), 31–58. <https://doi.org/10.14529/jsfi180103> (cit. on p. 17)
- MacArthur, P., Liu, Q., Russell, R. D., Mizero, F., Veeraraghavan, M. & Dennis, J. M. (2017). An integrated tutorial on infiniband, verbs, and MPI. *IEEE Commun. Surv. Tutorials*, 19(4), 2894–2926. <https://doi.org/10.1109/COMST.2017.2746083> (cit. on p. 65)
- MacLachlan, G., Hurlburt, J., Suarez, M., Wong, K. L., Burke, W., Lewis, T., Gallo, A., Flidr, J., Gabiam, R., Nicholas, J. & Ensorgauging, B. (2020). Building a shared resource HPC center across university schools and institutes: A case study. *CoRR, abs/2003.13629*. <https://arxiv.org/abs/2003.13629> (cit. on p. 15)
- Mena, J. A., Shaaban, O., Beltran, V., Carpenter, P. M., Ayguadé, E. & Mancho, J. L. (2022). Ompss-2@cluster: Distributed memory execution of nested openmp-style tasks. In J. Cano & P. Trinder (Eds.), *Euro-par 2022: Parallel processing - 28th international conference on parallel and distributed computing, glasgow, uk, august 22-26, 2022, proceedings* (pp. 319–334). Springer. https://doi.org/10.1007/978-3-031-12597-3_20. (Cit. on p. 143)

- Mendez, S. & Lührs, S. (2019, February 7). *Best practice guide - parallel i/o* (D. Sloan-Murphy, A. Turner & V. Weinberg, Eds.). https://prace-ri.eu/wp-content/uploads/Best-Practice-Guide_Parallel-IO.pdf. (Cit. on p. 22)
- Milewicz, R., Pirkelbauer, P., Soundararajan, P., Ahmed, H. & Skjellum, A. (2021). Negative perceptions about the applicability of source-to-source compilers in HPC: A literature review. In H. Jagode, H. Anzt, H. Ltaief & P. Luszczek (Eds.), *High performance computing - ISC high performance digital 2021 international workshops, frankfurt am main, germany, june 24 - july 2, 2021, revised selected papers* (pp. 233–246). Springer. https://doi.org/10.1007/978-3-030-90539-2_16. (Cit. on p. 56)
- Millot, D., Muller, A., Parrot, C. & Silber-Chaussumier, F. (2008). STEP: A distributed openmp for coarse-grain parallelism tool. In R. Eigenmann & B. R. de Supinski (Eds.), *Openmp in a new era of parallelism, 4th international workshop, IWOMP 2008, west lafayette, in, usa, may 12-14, 2008, proceedings* (pp. 83–99). Springer. https://doi.org/10.1007/978-3-540-79561-2_8. (Cit. on p. 148)
- Mishra, A., Malik, A. M. & Chapman, B. (2020). Extending the LLVM/clang framework for OpenMP metadirective support. *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. <https://doi.org/10.1109/llvmhpcchipar51896.2020.00009> (cit. on pp. 51, 192)
- Miyoshi, A., Lefurgy, C., Hensbergen, E. V., Rajamony, R. & Rajkumar, R. (2002). Critical power slope: Understanding the runtime effects of frequency scaling. In K. Ebcioglu, K. Pingali & A. Nicolau (Eds.), *Proceedings of the 16th international conference on supercomputing, ICS 2002, new york city, ny, usa, june 22-26, 2002* (pp. 35–44). ACM. <https://doi.org/10.1145/514191.514200>. (Cit. on p. 24)
- Mock, M. (2003). Dynamic analysis from the bottom up. *WODA 2003 ICSE Workshop on Dynamic Analysis*, 13 (cit. on pp. 44, 127).
- MPI Forum. (2021, June). *MPI : A Message-Passing Interface Standard : Version 4.0* (W. Gropp, Ed.). MPI Forum. <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>. (Cit. on pp. 63, 127, 131)
Last accessed: 2023-03-07.
- Mudalige, G., Giles, M., Regulý, I., Bertolli, C. & Kelly, P. (2012). Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. *2012 Innovative Parallel Computing (InPar)*, 1–12. <https://doi.org/10.1109/InPar.2012.6339594> (cit. on p. 146)
- Nakao, M., Murai, H. & Sato, M. (2019). Multi-accelerator extension in openmp based on PGAS model. *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2019, Guangzhou, China, January 14-16, 2019*, 18–25. <https://doi.org/10.1145/3293320.3293324> (cit. on p. 143)
- Nethercote, N. & Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. In J. Ferrante & K. S. McKinley (Eds.), *Proceedings of the ACM SIGPLAN 2007 conference on programming language design and implementation, san diego, california, usa, june 10-13, 2007* (pp. 89–100). ACM. <https://doi.org/10.1145/1250734.1250746>. (Cit. on p. 48)

- OpenMP Architecture Review Board. (2021). OpenMP Application Programming Interface 5.2 (cit. on pp. 60, 86, 87).
- Palfner, S. (2012). Das deutsche klimarechenzentrum. kartographie eines rechenraumes. *Zur Geschichte von Forschungstechnologien. Generizität – Interstitialität – Transfer* (cit. on p. 20).
- Patel, A. & Doerfert, J. (2022). Remote openmp offloading. In A. L. Varbanescu, A. Bhatele, P. Luszczek & M. Baboulin (Eds.), *High performance computing - 37th international conference, ISC high performance 2022, hamburg, germany, may 29 - june 2, 2022, proceedings* (pp. 315–333). Springer. https://doi.org/10.1007/978-3-031-07312-0_16. (Cit. on pp. 143, 192)
- Paul, A. K., Faaland, O., Moody, A., Gonsiorowski, E., Mohror, K. & Butt, A. R. (2020). Understanding HPC application I/O behavior using system level statistics. *27th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2020, Pune, India, December 16-19, 2020*, 202–211. <https://doi.org/10.1109/HiPC50609.2020.00034> (cit. on p. 23)
- Petrank, E. & Rawitz, D. (2002). The hardness of cache conscious data placement. In J. Launchbury & J. C. Mitchell (Eds.), *Conference record of POPL 2002: The 29th SIGPLAN-SIGACT symposium on principles of programming languages, portland, or, usa, january 16-18, 2002* (pp. 101–112). ACM. <https://doi.org/10.1145/503272.503283>. (Cit. on p. 100)
- Plehn, J., Fuchs, A., Kuhn, M., Lüttgau, J. & Ludwig, T. (2022). Data-aware compression for HPC using machine learning. *ACM SIGOPS Oper. Syst. Rev.*, 56(1), 62–69. <https://doi.org/10.1145/3544497.3544508> (cit. on p. 105)
- Priyadarshan, S. (2019, August 30). *A study of Binary Instrumentation techniques* (research rep.). Stony Brook University. (Cit. on p. 48).
- Quinlan, D. & Liao, C. (2011). The ROSE source-to-source compiler infrastructure. *Cetus users and compiler infrastructure workshop, in conjunction with PACT, 2011*, 1 (cit. on p. 56).
- Rabenseifner, R., Hager, G. & Jost, G. (2009). Hybrid mpi/openmp parallel programming on clusters of multi-core SMP nodes. In D. E. Baz, F. Spies & T. Gross (Eds.), *Proceedings of the 17th euromicro international conference on parallel, distributed and network-based processing, PDP 2009, weimar, germany, 18-20 february 2009* (pp. 427–436). IEEE Computer Society. <https://doi.org/10.1109/PDP.2009.43>. (Cit. on p. 115)
- Ramalingam, G. (1994). The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5), 1467–1471. <https://doi.org/10.1145/186025.186041> (cit. on pp. 37, 127)
- Reps, T. W., Horwitz, S. & Sagiv, S. (1995). Precise interprocedural dataflow analysis via graph reachability. In R. K. Cytron & P. Lee (Eds.), *Conference record of popl’95: 22nd ACM SIGPLAN-SIGACT symposium on principles of programming languages, san francisco, california, usa, january 23-25, 1995* (pp. 49–61). ACM Press. <https://doi.org/10.1145/199448.199462>. (Cit. on p. 43)
- Reussner, R. H., Sanders, P., Prechelt, L. & Müller, M. (1998). Skampi: A detailed, accurate MPI benchmark. In V. N. Alexandrov & J. J. Dongarra (Eds.), *Recent*

- advances in parallel virtual machine and message passing interface, 5th european PVM/MPI users' group meeting, liverpool, uk, september 7-9, 1998, proceedings* (pp. 52–59). Springer. <https://doi.org/10.1007/BFb0056559>. (Cit. on p. 78)
- Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2), 358–366. <https://doi.org/10.1090/s0002-9947-1953-0053041-6> (cit. on p. 44)
- Saà-Garriga, A., Castells-Rufas, D. & Carrabina, J. (2015). OMP2MPI: automatic MPI code generation from openmp programs. *CoRR*, abs/1502.02921. <http://arxiv.org/abs/1502.02921> (cit. on p. 147)
- Sato, M., Satoh, S., Kusano, K. & Tanaka, Y. (1999). Design of openmp compiler for an smp cluster. *Proc. of the 1st European Workshop on OpenMP*, 32–39 (cit. on pp. 56, 142).
- Savchenko, V. V., Sorokin, K. S., Pankratenko, G. A., Markov, S., Spiridonov, A., Alexandrov, I., Volkov, A. S. & Sun, K. (2019). Nobrainer: An example-driven framework for C/C++ code transformations. In N. S. Bjørner, I. B. Virbitskaite & A. Voronkov (Eds.), *Perspectives of system informatics - 12th international andrei p. ershov informatics conference, PSI 2019, novosibirsk, russia, july 2-5, 2019, revised selected papers* (pp. 140–155). Springer. https://doi.org/10.1007/978-3-030-37487-7_12. (Cit. on p. 36)
- Schmidt, B., Gonzalez-Dominguez, J., Hundt, C. & Schlarb, M. (2017). *Parallel programming: Concepts and practice*. Morgan Kaufmann. (Cit. on pp. 58, 144).
- Schneider, N., Kadosh, T., Hasabnis, N., Mattson, T., Pinter, Y. & Oren, G. (2023). Mpi-ricol: Data-driven mpi distributed parallelism assistance with transformers. <https://doi.org/10.48550/ARXIV.2305.09438> (cit. on p. 147)
- Schubert, P. D., Hermann, B. & Bodden, E. (2019). Phasar: An inter-procedural static analysis framework for c/c++. In T. Vojnar & L. Zhang (Eds.), *Tools and algorithms for the construction and analysis of systems* (pp. 393–410). Springer International Publishing. (Cit. on p. 43).
- Sensi, D. D., Girolamo, S. D., McMahon, K. H., Roweth, D. & Hoefler, T. (2020). An in-depth analysis of the slingshot interconnect. In C. Cuicchi, I. Qualters & W. T. Kramer (Eds.), *Proceedings of the international conference for high performance computing, networking, storage and analysis, SC 2020, virtual event / atlanta, georgia, usa, november 9-19, 2020* (p. 35). IEEE/ACM. <https://doi.org/10.1109/SC41405.2020.00039>. (Cit. on p. 65)
- Shan, B., Araya-Polo, M., Malik, A. M. & Chapman, B. M. (2023). Mpi-based remote openmp offloading: A more efficient and easy-to-use implementation. In Q. Chen, Z. Huang & M. Si (Eds.), *Proceedings of the 14th international workshop on programming models and applications for multicores and manycores, pmam@ppopp 2023, montreal, qc, canada, 25 february 2023 - 1 march 2023* (pp. 50–59). ACM. <https://doi.org/10.1145/3582514.3582519>. (Cit. on p. 143)
- Shan, H., Wright, N. J., Shalf, J., Yelick, K. A., Wagner, M. & Wichmann, N. (2012). A preliminary evaluation of the hardware acceleration of the cray gemini interconnect for PGAS languages and comparison with MPI. *SIGMETRICS Perform.*

- Evaluation Rev.*, 40(2), 92–98. <https://doi.org/10.1145/2381056.2381077> (cit. on pp. 142, 144)
- Shirako, J., Hayashi, A., Paul, S. R., Tumanov, A. & Sarkar, V. (2022). Automatic parallelization of python programs for distributed heterogeneous computing. In J. Cano & P. Trinder (Eds.), *Euro-par 2022: Parallel processing - 28th international conference on parallel and distributed computing, glasgow, uk, august 22-26, 2022, proceedings* (pp. 350–366). Springer. https://doi.org/10.1007/978-3-031-12597-3_22. (Cit. on p. 148)
- Si, M., Fu, H., Hammond, J. R. & Balaji, P. (2021). Openshmem over MPI as a performance contender: Thorough analysis and optimizations. In S. W. Poole, O. R. Hernandez, M. B. Baker & T. Curtis (Eds.), *Openshmem and related technologies. openshmem in the era of exascale and smart networks - 8th workshop on openshmem and related technologies, openshmem 2021, virtual event, september 14-16, 2021, revised selected papers* (pp. 39–60). Springer. https://doi.org/10.1007/978-3-031-04888-3_3. (Cit. on p. 142)
- Squar, J., Jammer, T., Blesel, M., Kuhn, M. & Ludwig, T. (2020). Compiler Assisted Source Transformation of OpenMP Kernels. *2020 19th International Symposium on Parallel and Distributed Computing (ISPD)*, 44–51. <https://doi.org/10.1109/ispd51135.2020.00016> (cit. on pp. 167, 168)
- Squar, J., Schroeter, N., Fuchs, A., Kuhn, M. & Ludwig, T. (2022). Content queries and in-depth analysis on version-controlled software. In M. Cristani, C. Toro, C. Zanni-Merk, R. J. Howlett & L. C. Jain (Eds.), *Knowledge-based and intelligent information & engineering systems: Proceedings of the 26th international conference kes-2022, verona, italy and virtual event, 7-9 september 2022* (pp. 1261–1270). Elsevier. <https://doi.org/10.1016/j.procs.2022.09.182>. (Cit. on p. 55)
- Sterling, T., Anderson, M. & Brodowicz, M. (2017, December 6). *High Performance Computing*. Elsevier LTD, Oxford. (Cit. on pp. 16, 29, 142, 184).
- Stpiczynski, P. (2018). Language-based vectorization and parallelization using intrinsics, openmp, TBB and cilk plus. *J. Supercomput.*, 74(4), 1461–1472. <https://doi.org/10.1007/s11227-017-2231-3> (cit. on p. 60)
- Sun, X. & Ni, L. M. (1993). Scalable problems and memory-bounded speedup. *J. Parallel Distributed Comput.*, 19(1), 27–37. <https://doi.org/10.1006/jpdc.1993.1087> (cit. on pp. 151, 152)
- Süßkraut, M., Knauth, T., Weigert, S., Schiffel, U., Meinhold, M. & Fetzner, C. (2010). Prospect: A compiler framework for speculative parallelization. In A. Moshovos, J. G. Steffan, K. M. Hazelwood & D. R. Kaeli (Eds.), *Proceedings of the CGO 2010, the 8th international symposium on code generation and optimization, toronto, ontario, canada, april 24-28, 2010* (pp. 131–140). ACM. <https://doi.org/10.1145/1772954.1772974>. (Cit. on p. 56)
- Tanenbaum, A. S. (2009). *Modern operating systems, 3rd edition*. Pearson Prentice-Hall. <https://www.worldcat.org/oclc/254320777>. (Cit. on p. 84)
- Terboven, C., an Mey, D., Schmidl, D. & Wagner, M. (2008). First experiences with intel cluster openmp. In R. Eigenmann & B. R. de Supinski (Eds.), *Openmp in a new era of parallelism, 4th international workshop, IWOMP 2008, west*

- lafayette, in, usa, may 12-14, 2008, proceedings* (pp. 48–59). Springer. https://doi.org/10.1007/978-3-540-79561-2_5. (Cit. on p. 143)
- Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V. Q., Ellingwood, N. D., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., Liber, N., Madsen, J. R., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcsin, B. & Wilke, J. J. (2022). Kokkos 3: Programming model extensions for the exascale era. *IEEE Trans. Parallel Distributed Syst.*, 33(4), 805–817. <https://doi.org/10.1109/TPDS.2021.3097283> (cit. on pp. 32, 34, 144)
- van der Pas, R., Stotzer, E. & Terboven, C. (2017). *Using openmp - the next step: Affinity, accelerators, tasking, and SIMD*. MIT Press. <https://mitpress.mit.edu/books/using-openmp-next-step>. (Cit. on p. 131)
- Wahlgren, J., Gokhale, M. B. & Peng, I. B. (2022). Evaluating emerging cxl-enabled memory pooling for HPC systems. *IEEE/ACM Workshop on Memory Centric High Performance Computing, MCHPC@SC 2022, Dallas, TX, USA, November 13-18, 2022*, 11–20. <https://doi.org/10.1109/MCHPC56545.2022.00007> (cit. on p. 192)
- Wendland, S., Hankers, B., Bock, M., Böhner, J., Squar, J., Lembrich, D. & Conrad, O. (2023). Ein Simulationsmodell zur Erfassung von Abflussrisiken in der Landwirtschaft. *Inform. Spektrum*, 46(1), 15–23. <https://doi.org/10.1007/s00287-023-01522-2> (cit. on p. 18)
- Williams, S., Waterman, A. & Patterson, D. (2009). Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Communications of the ACM*, 52(4), 65. <https://doi.org/10.1145/1498765.1498785> (cit. on p. 78)
- Wong, H. J., Cai, J., Rendell, A. P. & Strazdins, P. E. (2008). Micro-benchmarks for cluster openmp implementations: Memory consistency costs. In R. Eigenmann & B. R. de Supinski (Eds.), *Openmp in a new era of parallelism, 4th international workshop, IWOMP 2008, west lafayette, in, usa, may 12-14, 2008, proceedings* (pp. 60–70). Springer. https://doi.org/10.1007/978-3-540-79561-2_6. (Cit. on p. 143)
- Wu, X., Deshpande, V. & Mueller, F. (2012). ScalaBenchGen: Auto-generation of communication benchmarks traces. *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. <https://doi.org/10.1109/ipdps.2012.114> (cit. on p. 78)
- Wulf, W. A. & McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1), 20–24. <https://doi.org/10.1145/216585.216588> (cit. on pp. 18, 71)
- XcalableMP Specification Working Group. (2018). *Xcalablemp language specification*. Retrieved May 11, 2020, from <https://xcalablemp.org/download/spec/xmp-spec-1.4.pdf>. (Cit. on p. 147)
- Yadavalli, S. B. & Smith, A. (2019). Raising binaries to LLVM IR with MCTOLL (WIP paper). In J. Chen & A. Shrivastava (Eds.), *Proceedings of the 20th ACM SIGPLAN/SIGBED international conference on languages, compilers, and tools for embedded systems, LCTES 2019, phoenix, az, usa, june 23-23, 2019* (pp. 213–218). ACM. <https://doi.org/10.1145/3316482.3326354>. (Cit. on p. 107)

- Yan, B. & Regueiro, R. A. (2018). Comparison between pure MPI and hybrid MPI-OpenMP parallelism for discrete element method (DEM) of ellipsoidal and poly-ellipsoidal particles. *Computational Particle Mechanics*, 6(2), 271–295. <https://doi.org/10.1007/s40571-018-0213-8> (cit. on p. 128)
- Yviquel, H., Pereira, M., Franceschini, E., Valarini, G., Leite, G., Rosso, P. H. D. F., Ceccato, R., Cusiualpa, C., Dias, V., Rigo, S., Souza, A. & Araujo, G. (2022). The openmp cluster programming model. *Workshop Proceedings of the 51st International Conference on Parallel Processing, ICPP Workshops 2022, Bordeaux, France, 29 August 2022 - 1 September 2022*, 17:1–17:11. <https://doi.org/10.1145/3547276.3548444> (cit. on p. 143)
- Zängl, G., Reinert, D., Rípodas, P. & Baldauf, M. (2014). The ICON (ICOsahedral non-hydrostatic) modelling framework of DWD and MPI-m: Description of the non-hydrostatic dynamical core. *Quarterly Journal of the Royal Meteorological Society*, 141(687), 563–579. <https://doi.org/10.1002/qj.2378> (cit. on p. 27)

Cited Websites

- Blagodarenko, A. & Faber, C. (2023). Client-Side Data Compression [Last accessed: 2023-06-21]. https://wiki.lustre.org/images/0/06/LUG2023-Client_Side_Data_Compression-Blagodarenko.pdf. (Cit. on p. 148)
- Bougacha, A. (2017). Dagger - Binary Translator to LLVM IR [Last accessed: 2023-05-07]. <https://github.com/repzret/dagger>. (Cit. on p. 107)
- Cocoacrumbs. (2021). Experimenting with the LLVM/Clang toolchain for the Zilog eZ80 [Last accessed: 2023-02-28]. <https://www.cocoacrumbs.com/blog/2021-12-28-experimenting-with-the-llvm-toolchain-for-the-ez80/>. (Cit. on p. 120)
- Collet, Y. (n.d.). Zstandard [Last accessed: 2023-05-05]. <https://facebook.github.io/zstd/>. (Cit. on p. 106)
- Dask core developers. (2022). Dask [Last accessed: 2023-06-25]. <https://www.dask.org/>. (Cit. on p. 144)
- Desaulniers, N. (2017). GCC vs LLVM Q3 2017 Commit Rates and Active Developer Counts [Last accessed: 2023-06-10]. <https://nickdesaulniers.github.io/blog/2017/09/05/gcc-vs-llvm-q3-2017-commit-rates-and-active-developer-counts/>. (Cit. on p. 57)
- Deutsches Klimarechenzentrum GmbH. (n.d.-a). HLRE-3 - der Supercomputer Mistral [Last accessed: 2023-06-08]. <https://www.dkrz.de/de/kommunikation/galerie/Media-DKRZ/hlre-3>. (Cit. on p. 27)
- Deutsches Klimarechenzentrum GmbH. (n.d.-b). HLRE-4 Levante [Last accessed: 2023-06-08]. <https://www.dkrz.de/en/systems/hpc/hlre-4-levante>. (Cit. on p. 155)
- Deutsches Klimarechenzentrum GmbH. (2021). Levante Configuration [Last accessed: 2023-06-18]. <https://docs.dkrz.de/doc/levante/configuration.html>. (Cit. on p. 155)
- Droste, A. (2020). Mpi-checker [Last accessed: 2023-03-02]. <https://github.com/0ax1/MPI-Checker>. (Cit. on p. 40)

- Gailly, J.-l. & Adler, M. (2023). Zlib [Last accessed: 2023-05-05]. <https://www.zlib.net/>. (Cit. on p. 106)
- GCC Team. (n.d.). Declaring Attributes of Functions [Last accessed: 2023-05-10]. <https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Function-Attributes.html>. (Cit. on p. 113)
- GNU. (n.d.). Generic [Last accessed: 2023-06-10]. <https://gcc.gnu.org/onlinedocs/gccint/GENERIC.html>. (Cit. on p. 57)
- Google. (2023). Googletest - google testing and mocking framework [Last accessed: 2023-05-29]. <https://github.com/google/googletest>. (Cit. on p. 125)
- GWDG. (n.d.). Hardwarebeschreibung SGI Altix 4700 [Last accessed: 2023-06-21]. https://wwwuser.gwdg.de/~parallel/parallelrechner/altix_beschreibung.html. (Cit. on p. 142)
- HDF Group. (n.d.). Szip compression in hdf products [Last accessed: 2023-02-19]. https://support.hdfgroup.org/doc_resource/SZIP/index.html. (Cit. on p. 106)
- HDF Group. (2018). Release of HDF5 1.10.2 – Newsletter #160 [Last accessed: 2023-05-09]. <https://www.hdfgroup.org/2018/03/release-of-hdf5-1-10-2-newsletter-160/>. (Cit. on p. 104)
- HDF Group. (2023). Registered filter plugins [Last accessed: 2023-05-05]. <https://portal.hdfgroup.org/display/support/Registered+Filter+Plugins>. (Cit. on p. 104)
- HPCG Benchmark. (2022). HPCG Benchmark [Last accessed: 2023-06-26]. <https://www.hpcg-benchmark.org/>. (Cit. on p. 78)
- IBM. (2021). File compression [Last accessed: 2023-06-21]. <https://www.ibm.com/docs/en/storage-scale/4.2.0?topic=systems-file-compression>. (Cit. on p. 148)
- Intel. (n.d.). Intel oneAPI Threading Building Blocks [Last accessed: 2023-06-25]. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html>. (Cit. on p. 144)
- Intel. (2018). Introducing Intel MPI Benchmarks [Last accessed: 2023-06-26]. <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-mpi-benchmarks.html>. (Cit. on p. 78)
- Iza-Teran, R. & Lorentz, R. (2005). Lossless Compression of Meteorological Data [Last accessed: 2023-06-21]. https://www.ercim.eu/publication/Ercim_News/enw61/lorentz.html. (Cit. on p. 149)
- JuliaHubOSS. (2022). Resurrected LLVM "C Backend", with improvements [Last accessed: 2023-02-28]. <https://github.com/JuliaHubOSS/llvm-cbe>. (Cit. on p. 108)
- Knox, L. & Pourmal, E. (2018). Overcoming Pitfalls When Using HDF5 Compression [Last accessed: 2023-06-21]. <https://ntrs.nasa.gov/api/citations/20180008456/downloads/20180008456.pdf>. (Cit. on p. 149)
- Křoustek, J. (2023). Retdec [Last accessed: 2023-05-07]. <https://github.com/avast/retdec>. (Cit. on p. 107)
- K.Rupp, Horowitz, M., Labonte, F., Shacham, O., Olukotun, K., Hammond, L. & Batten, C. (2022). Microprocessor Trend Data [Last accessed: 2023-06-06]. <https://github.com/karlrupp/microprocessor-trend-data>. (Cit. on p. 17)
- Lattner, C. (n.d.). The architecture of open source applications [Last accessed: 2023-05-12]. <http://www.aosabook.org/en/llvm.html>. (Cit. on pp. 117, 118)

- LLNL. (2020). MACSio [Last accessed: 2023-06-26]. <https://github.com/LLNL/MACSio>. (Cit. on p. 78)
- LLVM Project. (n.d.-a). DragonEgg - Using LLVM as a GCC backend [Last accessed: 2023-06-09]. <https://dragonegg.llvm.org/>. (Cit. on p. 57)
- LLVM Project. (n.d.-b). Immutability [last accesses 2023-03-02]. <https://clang.llvm.org/docs/InternalsManual.html#immutability>. (Cit. on p. 40)
- LLVM Project. (n.d.-c). Projects built with LLVM [Last accessed: 2023-05-12]. <https://llvm.org/ProjectsWithLLVM/>. (Cit. on p. 117)
- LLVM Project. (2012). Llvm 3.1 release notes [Last accessed: 2023-05-07]. <https://releases.llvm.org/3.1/docs/ReleaseNotes.html>. (Cit. on p. 108)
- LLVM Project. (2023a). Attributes in Clang [Last accessed: 2023-05-10]. <https://clang.llvm.org/docs/AttributeReference.html#nodiscard-warn-unused-result>. (Cit. on p. 113)
- LLVM Project. (2023b). Choosing the Right Interface for Your Application [Last accessed: 2023-05-12]. <https://clang.llvm.org/docs/Tooling.html>. (Cit. on p. 118)
- LLVM Project. (2023c). Flang [Last accessed: 2023-06-10]. <https://github.com/llvm/llvm-project/tree/main/flang>. (Cit. on p. 57)
- LLVM Project. (2023d). How To Release LLVM To The Public [Last accessed: 2023-03-24]. <https://llvm.org/docs/HowToReleaseLLVM.html>. (Cit. on p. 57)
- LLVM Project. (2023e). libclang: C Interface to Clang [Last accessed: 2023-05-12]. https://clang.llvm.org/doxygen/group___CINDEX.html. (Cit. on p. 118)
- LLVM Project. (2023f). Libtooling [Last accessed: 2023-05-12]. <https://clang.llvm.org/docs/LibTooling.html>. (Cit. on p. 118)
- LLVM Project. (2023g). lit - LLVM Integrated Tester [Last accessed: 2023-05-29]. <https://llvm.org/docs/CommandGuide/lit.html>. (Cit. on p. 49)
- LLVM Project. (2023h). Llvm alias analysis infrastructure [Last accessed: 2023-01-09]. <https://llvm.org/docs/AliasAnalysis.html>. (Cit. on p. 127)
- LLVM Project. (2023i). LLVM Block Frequency Terminology [Last accessed: 2023-03-02]. <https://llvm.org/docs/BlockFrequencyTerminology.html>. (Cit. on p. 43)
- LLVM Project. (2023j). LLVM Language Reference Manual [Last accessed: 2023-03-02]. <https://llvm.org/docs/LangRef.html>. (Cit. on pp. 42, 57, 120)
- LLVM Project. (2023k). `llvm::BlockFrequencyInfoImpl` Class Template Reference [Last accessed: 2023-03-02]. https://llvm.org/doxygen/classllvm_1_1BlockFrequencyInfoImpl.html#details. (Cit. on p. 43)
- LLVM Project. (2023l). LLVM's Analysis and Transform Passes [Last accessed: 2023-05-13]. <https://llvm.org/docs/Passes.html>. (Cit. on p. 121)
- LLVM Project. (2023m). Writing an llvm pass [Last accessed: 2023-05-15]. <https://llvm.org/docs/WritingAnLLVMPass.html>. (Cit. on p. 121)
- LLVM Project. (2023n). Writing an llvm pass [Last accessed: 2023-05-29]. <https://llvm.org/docs/WritingAnLLVMNewPMPass.html>. (Cit. on p. 122)
- Microway. (2023). DDN EXAScaler [Last accessed: 2023-06-17]. <https://www.microway.com/products/storage/ddn-exascalr/>. (Cit. on p. 155)
- MPI Forum. (n.d.). Implementation Status [Last accessed: 2023-06-11]. <https://www.mpi-forum.org/implementation-status/>. (Cit. on p. 63)

- NASA. (2022). NAS Parallel Benchmarks [Last accessed: 2023-04-17]. <https://www.nasa.gov/software/npb.html>. (Cit. on p. 78)
- netlib. (2018). HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers [Last accessed: 2023-06-26]. <https://netlib.org/benchmark/hpl/>. (Cit. on p. 78)
- Peter, D. (2023). Binocle [Last accessed: 2023-06-17]. <https://github.com/sharkdp/binocle>. (Cit. on p. 178)
- Pophale, S. & Curtis, T. (n.d.). OpenSHMEM TUTORIAL [Last accessed: 2023-06-25]. OpenSHMEM%20TUTORIAL. (Cit. on p. 145)
- Pourmal, E. & Knox, L. (2017). HDF5 Data Compression Demystified #2: Performance Tuning [Last accessed: 2023-06-21]. <https://www.hdfgroup.org/2017/05/hdf5-data-compression-demystified-2-performance-tuning>. (Cit. on p. 149)
- Rew, R. (2013a). Chunking data: Choosing shapes [Last accessed: 2023-05-17]. https://www.unidata.ucar.edu/blogs/developer/en/entry/chunking_data_choosing_shapes. (Cit. on p. 74)
- Rew, R. (2013b). Chunking data: Why it matters [Last accessed: 2023-05-17]. https://www.unidata.ucar.edu/blogs/developer/en/entry/chunking_data_why_it_matters. (Cit. on pp. 72, 74)
- Sampson, A. (2015). LLVM for Grad Students [Last accessed: 2023-05-12]. <https://www.cs.cornell.edu/~asampson/blog/llvm.html>. (Cit. on p. 124)
- Seward, J. (2019). Bzip2 [Last accessed: 2023-05-05]. <https://sourceware.org/bzip2/>. (Cit. on p. 106)
- Shah, A. (2022). RISC-V Is Far from Being an Alternative to x86 and Arm in HPC [Last accessed: 2023-02-27]. <https://www.hpcwire.com/2022/11/18/risc-v-is-far-from-being-an-alternative-to-x86-and-arm-in-hpc/>. (Cit. on p. 49)
- Siebenmann, C. (2012). Understanding Resident Set Size and the RSS problem on modern Unixes [Last accessed: 2023-06-23]. <https://utcc.utoronto.ca/~cks/space/blog/unix/UnderstandingRSS>. (Cit. on p. 138)
- Silicon Graphics International Corp. (2022). Openshmem [Last accessed: 2023-06-25]. <http://openshmem.org/site/>. (Cit. on p. 144)
- Strohmaier, E., Dongarra, J., Simon, H. & Meuer, M. (2023). Top500 - the list [Last accessed: 2023-04-13]. <https://www.top500.org/>. (Cit. on pp. 17, 63, 119, 155)
- The Blosc Developers. (2023). Blosc [Last accessed: 2023-05-05]. <https://www.blosc.org/>. (Cit. on p. 106)
- The European Technology Platform For High-Performance Computing. (2018). Why is HPC important? (The European Technology Platform For High-Performance Computing, Ed.) [Last accessed: 2023-04-13]. <http://www.etp4hpc.eu/the-importance-of-hpc.html>. (Cit. on p. 16)
- Unidata. (n.d.-a). Lossy compression with quantize [Last accessed: 2023-05-05]. https://docs.unidata.ucar.edu/netcdf-c/current/md__media_psf_Home_Desktop__netcdf_releases_v4_9_2_release__netcdf_c_docs_quantize.html#quantize. (Cit. on p. 105)
- Unidata. (n.d.-b). NetCDF Utilities [Last accessed: 2023-06-02]. https://docs.unidata.ucar.edu/nug/current/netcdf_utilities_guide.html. (Cit. on p. 134)

- Unidata. (2020). Allow user to turn on zlib, shuffle, and/or fletcher32 filters with parallel I/O for HDF5-1.10.2+ [Last accessed: 2023-06-02]. <https://github.com/Unidata/netcdf-c/commit/8771d0bdf44e26472a98a25f5f833630ffea4edb>. (Cit. on p. 134)
- Unidata. (2023a). NetCDF File and Data I/O [Last accessed: 2023-06-02]. https://docs.unidata.ucar.edu/netcdf-c/current/group__datasets.html. (Cit. on pp. 133, 134)
- Unidata. (2023b). Netcdf variables [Last accessed: 2023-06-02]. https://docs.unidata.ucar.edu/netcdf-c/current/group__variables.html. (Cit. on p. 134)
- Unidata News Comments. (2020). Netcdf 4.7.4 [Last accessed: 2023-02-19]. <https://www.unidata.ucar.edu/blogs/news/entry/netcdf-4-7-4>. (Cit. on p. 104)
- University of Illinois. (2021). Charm++ [Last accessed: 2023-06-22]. <https://charmplusplus.org/>. (Cit. on p. 144)
- Vader, G. (2016). NetCDF Compression [Last accessed: 2023-02-19]. https://www.unidata.ucar.edu/blogs/developer/entry/netcdf_compression. (Cit. on pp. 74, 105)
- Winkler, H. (2022). Clang/llvm overview [Last accessed: 2023-06-10]. <https://blog.parcio.de/posts/2022/07/clang-llvm-overview/>. (Cit. on p. 58)
- WR. (2023). Scientific Computing // Wissenschaftliches Rechnen [Last accessed: 2023-06-18]. <https://wr.informatik.uni-hamburg.de/>. (Cit. on p. 155)
- Yadavalli, S. B. & Smith, A. (2022). llvm-mctoll [Last accessed: 2023-05-07]. <https://github.com/microsoft/llvm-mctoll>. (Cit. on p. 107)

Own Publications

- Blesel, M., Kuhn, M. & Squar, J. (2021). heimdallr: Improving Compile Time Correctness Checking for Message Passing with Rust. In H. Jagode, H. Anzt, H. Ltaief & P. Luszczek (Eds.), *High performance computing - ISC high performance digital 2021 international workshops, frankfurt am main, germany, june 24 - july 2, 2021, revised selected papers* (pp. 199–211). Springer. https://doi.org/10.1007/978-3-030-90539-2_13
- Duwe, K., Lüttgau, J., Mania, G., Squar, J., Fuchs, A., Kuhn, M., Betke, E. & Ludwig, T. (2020). State of the Art and Future Trends in Data Reduction for High-Performance Computing. *Supercomput. Front. Innov.*, 7(1), 4–36. <https://doi.org/10.14529/jsfi200101> (cit. on p. 71)
- Keiff, M., Voigt, F., Fuchs, A., Kuhn, M., Squar, J. & Ludwig, T. (2022). Automated performance analysis tools framework for HPC programs. In M. Cristani, C. Toro, C. Zanni-Merk, R. J. Howlett & L. C. Jain (Eds.), *Knowledge-based and intelligent information & engineering systems: Proceedings of the 26th international conference kes-2022, verona, italy and virtual event, 7-9 september 2022* (pp. 1067–1076). Elsevier. <https://doi.org/10.1016/j.procs.2022.09.162>
- Ludwig, T. & Squar, J. (2023). Editorial Themenheft Digitale Landwirtschaft. *Informatik Spektrum*, 46(1), 1–2. <https://doi.org/10.1007/s00287-023-01521-3>
- Squar, J., Jammer, T., Blesel, M., Kuhn, M. & Ludwig, T. (2020). Compiler Assisted Source Transformation of OpenMP Kernels. *2020 19th International Symposium*

- on Parallel and Distributed Computing (ISPD)*, 44–51. <https://doi.org/10.1109/ispdc51135.2020.00016> (cit. on pp. 167, 168)
- Squar, J., Schroeter, N., Fuchs, A., Kuhn, M. & Ludwig, T. (2022). Content queries and in-depth analysis on version-controlled software. In M. Cristani, C. Toro, C. Zanni-Merk, R. J. Howlett & L. C. Jain (Eds.), *Knowledge-based and intelligent information & engineering systems: Proceedings of the 26th international conference kes-2022, verona, italy and virtual event, 7-9 september 2022* (pp. 1261–1270). Elsevier. <https://doi.org/10.1016/j.procs.2022.09.182>. (Cit. on p. 55)
- Wendland, S., Hankers, B., Bock, M., Böhner, J., Squar, J., Lembrich, D. & Conrad, O. (2023). Ein Simulationsmodell zur Erfassung von Abflussrisiken in der Landwirtschaft. *Inform. Spektrum*, 46(1), 15–23. <https://doi.org/10.1007/s00287-023-01522-2> (cit. on p. 18)

Appendices

A. Appendix

A.1. LLVM Visualisation Tools

- Create AST output: `clang-check -ast-dump -ast-dump-filter=main source.c`
(add `-extra-arg="-fno-color-diagnostics"` flag to suppress colours, if output shall be redirected into a file)
- Create AST visualisation:
 - Get output from `clang -cc1 -ast-view source.c` (requires debug build of llvm)
 - Convert graph description with graphviz: `dot -Tpdf /tmp/TMPNAME -o source_ast.pdf`
- Create CFG visualisation:
 - Create IR without `optnone` attribute: `clang -emit-llvm -S -Xclang -disable-O0-optnone source.c`
 - Create dot file: `opt -dot-cfg source.ll`
 - Convert graph description with graphviz: `dot -Tpdf -o source_cfg.pdf .main.dot`

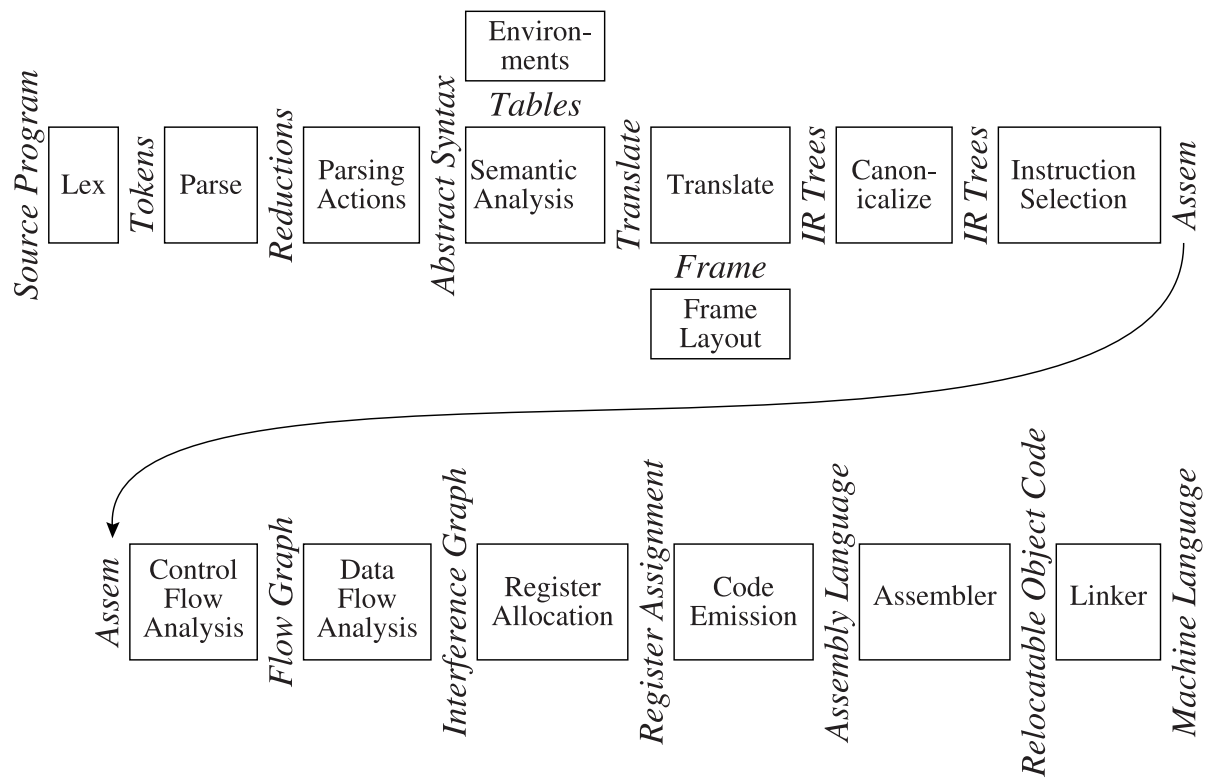


Figure A.1.: General composition of a compiler including all phases a high-level source code it passed through to be transformed into the final binary. Figure is from Appel, 1998.

```
1  #include "llvm/Pass.h"
2  #include "llvm/IR/Function.h"
3  #include "llvm/Support/raw_ostream.h"
4
5  using namespace llvm;
6  namespace
7  {
8      struct printFunction : public FunctionPass
9      {
10         static char ID;
11         printFunction() : FunctionPass(ID) {}
12
13         bool runOnFunction(Function &F) override
14         {
15             errs() << "Hello: ";
16             errs().write_escaped(F.getName()) << '\n';
17             return false;
18         }
19     };
20 }
21
22 char printFunction::ID = 0;
23 static RegisterPass<printFunction> X("hello", "Hello World Pass",
↪ false, false);
```

Listing A.1.: Simple LLVM function pass

```
1  #Compile pass
2  $ clang++ -fno-rtti -fPIC -shared -o pass.so pass.cpp
3  #Link pass into application
4  $ mpicxx -cxx=clang++ -fopenmp -Xclang -load -Xclang pass.so -o app.x
↪ app.cpp
```

Listing A.2.: Linking an LLVM module pass

```
1  #include <stdio.h>
2  #include "eingabeTrafo.h"
3
4
5  int main()
6  {
7      const int xDimension = 160;
8      const int yDimension = 40;
9      // Male einen schönen Rundbogen
10     for (int y = 0; y < yDimension; ++y)
11     {
12         for (int x = 0; x < xDimension; ++x)
13         {
14             if(checkLocationPrint(x,y,xDimension,yDimension))
15             {
16                 printf("#");
17             }
18             else
19             {
20                 printf(" ");
21             }
22         }
23         printf("\n");
24     }
25 }
```

Listing A.3.: Main application, which adds elements to the ‘canvas’.

```
1  #include <math.h>
2  #include <stdlib.h>
3  #include "eingabeTrafo.h"
4
5  const float PI = 3.14;
6
7  // Gib Werte zurück, ob an der gegebenen Position ein Zeichen
   ↪ ausgegeben werden soll
8  int checkLocationPrint(int x, int y, int xDim, int yDim)
9  {
10     // x und y auf bildfläche bzgl. pi skalieren, damit kurve bild
   ↪ ausfüllt
11     float scaledX = (double)x/(double)xDim;
12     float scaledY = (double)y/(double)yDim*2.0 - 1 ;
13
14     float sinValue = sin(scaledX*4*PI);
15     if(fabs(scaledY-sinValue) < 0.075)
16     {
17         return 1;
18     }
19     else
20     {
21         return 0;
22     }
23 }
```

Listing A.4.: Helper function, which decides based on the value of `sin` if the main application shall draw a sign.

```
1  double sin(double input)
2  {
3      return 0;
4  }
```

Listing A.5.: Replacement code, which redefines `sin` and overshadows the definition from `math.h` respectively the math library.



Figure A.2.: CFG of netCDF parallel I/O benchmark, after it has been modified by CATO, created by *LLVis*.

Glossary

AGSearch *Advanced GitHub Search* is a tool to traverse through GitHub repository matching a user-defined set of filters and analyses them locally (e.g. if specific functions are called).

API Application Programming Interface, a set of defined rules and protocols that allows different software applications to communicate and interact with each other, facilitating the exchange of data and functionality.

assembly Low-level programming language that represents machine instructions in a human-readable form and is directly understandable by the computer's processor.

atmosphere Gaseous envelope surrounding the Earth, composed of various layers and responsible for weather patterns, climate dynamics, and the exchange of gases between the Earth's surface and outer space.

awk Flexible programming language and text processing tool used for extracting and manipulating data, primarily in UNIX and Linux environments.

binocle Graphical tool to visualise binary data on disk.

BinOpt LLVM library to perform binary optimisation and specialisation.

biosphere Zone on Earth where living organisms exist, encompassing all ecosystems and interactions between organisms and their environment.

Blosc A blocking, shuffling and lossless compression library.

Bulk synchronous parallel Bridging model between hardware and software to design parallel algorithms.

bzip2 Widely used data compression algorithm and software tool for lossless data compression.

C General-purpose, imperative programming language known for its efficiency, low-level control, and portability, widely used for developing operating systems, embedded systems, and performance-critical applications.

C++ Object-oriented powerful programming language that extends the capabilities of C with features such as classes, templates, and polymorphism, enabling efficient and modular development of a wide range of applications.

Cetus S2S compiler for automatic parallelisation on multicore systems with focus on research capabilities.

Charm++ Parallel programming framework in C++ with focus on portability.

- Chunk** Hyper-rectangle of arbitrary shape, which defines strong adjacencies between single data.
- clang** Compiler front-end for the C, C++, and Objective-C programming languages, providing robust and efficient compilation, supporting various platforms and architectures.
- clang-tidy** A C++ linter tool based on **clang**, offering a flexible framework for identifying and correcting common programming errors through static analysis and enabling easy creation of new checks.
- Clava** S2S compiler based on **clang** to translate JavaScript based DSL into C.
- CMake** Tool suite to build and test software.
- CPU** *Central Processing Unit.*
- Cray Slingshot** High-performance network designed for HPE Cray supercomputers and HPE HPC clusters, integrating various simulation, modeling, AI, and analytics workloads into a single system for supercomputing in the era of Exascale computing.
- CrossDBT** Dynamic binary translation emulator.
- CUDA** Parallel computing platform and programming model developed by NVIDIA that enables efficient utilisation of GPU resources for accelerated computing tasks.
- Dagger** LLVM tool to decompile a binary into IR.
- decompiler** Tool to reverse the usual compilation process by generating high-level code from low-level code.
- DragonEgg** gcc plugin to use LLVM's optimisation and code generation components.
- du** Linux command line tool to estimate a file's space usage.
- Dwarf** Classes of numerical communication patterns.
- epoch** Execution span between two synchronisation calls, in which .
- ext4** Widely-used, scalable, and robust FS for Linux-based operating systems, offering features such as support for large file sizes, journaling, and backward compatibility with its predecessor, ext3.
- flang** Fortran language compiler based on the LLVM infrastructure, designed to compile and optimise Fortran programs for HPC environments.

Fortran Programming language widely used in the field of HPC due to its strong numerical and scientific computing capabilities, making it well-suited for computationally intensive applications such as simulations, modeling, and data processing in scientific and engineering domains.

Frontier Exascale supercomputer at *Oak Ridge National Laboratory*.

Fujitsu Tofu Specialised network topology developed by Fujitsu for supercomputers like the K computer and Fugaku, featuring a six-dimensional mesh structure, exceptional scalability of over 100000 nodes, and high-speed, bidirectional links with a peak bandwidth of 10 Gb/s per link.

GASPI PGAS specification, an alternative to MPI.

GCC GNU Compiler Collection.

gcc GCC compiler frontend for C.

GENERIC GNU IR.

gfortran GCC compiler frontend for Fortran.

GitHub Web-based platform that provides version control, collaboration tools, and hosting for software development projects, allowing developers to work together and track changes to code repositories.

GNU Comprehensive collection of free software that can function as an operating system or be used alongside other operating systems, and it is closely associated with the Linux family of operating systems.

GoogleTest Test framework by Google, which supports many kind of tests and mockups.

GPCNeT Benchmark suite to induce and measure network congestion on HPC systems.

GPU *Graphics Processing Unit*.

graphviz Open-source graph visualisation software that allows users to create, analyse, and render graphs and diagrams programmatically or through a simple textual description.

Green-Marl DSL to ease the parallelisation of graph analysis algorithms.

HDF5 Hierarchical Data Format version 5 is a flexible and efficient self describing file format designed for storing and managing large and complex scientific data, providing high-performance data storage and organisation capabilities.

HPCG Creates a new ranking metric for high-performance computing systems, complementing HPL benchmark, by testing patterns that match a broader set of applications and incentivising system designers to enhance performance.

HPL Benchmarking tool used to measure the computational power and performance of computer systems by solving a system of linear equations, often used to rank the world's fastest supercomputers in the TOP500 list.

hydrosphere Includes all forms of water found on Earth, such as oceans, lakes, rivers, groundwater, and atmospheric moisture, contributing significantly to global climate regulation, the water cycle, and the sustenance of diverse ecosystems.

InfiniBand High-speed networking standard designed for interconnecting computer systems, storage devices, and communication infrastructure; latencies of less than one microsecond and data rates of up to 100 Gb/s per link.

Insieme S2S compiler infrastructure for researching parallel languages within the HPC context.

Instrew Client-Server framework for dynamic binary instrumentation.

Intel MPI Stands out for its optimised performance, portability across multiple operating systems, adherence to MPI standards, and seamless integration with Intel tools, making it a preferred choice for high-performance message passing and parallel computing on Intel architectures.

Intel MPI Benchmarks Collection of elementary benchmarks adhering to MPI-1, MPI-2, and MPI-3 standards, enabling users to run the supported benchmarks, either in their entirety or selectively, by specifying different settings through command-line parameters.

Java Widely-used, object-oriented programming language known for its platform independence, extensive library support, and ability to build robust and scalable applications for various domains such as enterprise software, mobile development, and embedded systems.

JavaScript Flexible, interpreted programming language used primarily for web development, allowing dynamic and interactive behavior on websites through client-side scripting.

Kokkos Library for programming with portable performance.

ld GNU linker.

ldiskfs FS based on ext4 with improved performance.

Levante Supercomputer for ESS research, which is located at the *Deutsches Klimarechenzentrum GmbH* in Hamburg, Germany. It is operational since 2022.

Linux Popular open-source operating system kernel that forms the foundation of various Linux distributions, providing a robust and customisable platform for running computer systems ranging from servers to personal computers and embedded devices.

lit LLVM Integrated Tester.

llc LLVM static compiler, which translates LLVM IR into machine code for a specific target architecture, enabling efficient compilation and optimisation of programming languages.

LLVis LLVM pass to create CFG with mapping of IR onto high-level code.

LLVM Modular compiler framework, first developed by Chris Lattner, which allows to easily step into every component of the compilation workflow. High-level code is translated into a assembly-like but machine-independent IR, which can be easily analysed and transformed before translated by the backend into the final machine code.

LLVM-CBE LLVM backend to compile IR into C code.

llvm-mctoll LLVM tool to decompile a binary into IR.

llvm2c Tool for converting LLVM bitcode into C code.

ls Linux command line tool to list files and their attributes.

Lustre High-performance parallel distributed client-server file system designed for large-scale computing clusters, offering scalable storage capacity and high-speed data access for demanding workloads.

MACSio Application-focused and scalable I/O proxy application that aims to address the longstanding need for evaluating I/O performance, data models, library interfaces, and parallel I/O paradigms in multi-physics, high-performance computing applications.

MARTINI Tool prototype based on `clang` to match and replace code automatically using rules.

Mercurium S2S compiler offering a testbed for new OpenMP features.

Mistral Supercomputer for ESS research, which was located at the *Deutsches Klimarechenzentrum GmbH* in Hamburg, Germany. It has been used from 2015 up to 2022.

Molly Further development of Polly, utilising MPI.

MPI-checker Static analysis tool that utilises `clang`’s Static Analyser to validate proper usage of the MPI API in C and C++ code, supporting both path-sensitive and non-path-sensitive analysis by leveraging the abstract syntax tree representation of the source code.

MPI-IO MPI interface, which focuses on file I/O. A well known implementation is *ROMIO*, which is used by major MPI implementations like *MPICH*, *MVAPICH* or *Intel MPI*.

MPI-rical Tool for automatic parallelisation of serial code using MPI based on machine learning functionality.

MPICH High-performance implementation of the MPI standard developed by the Argonne National Laboratory and the Mississippi State University.

MUST Tool for efficient runtime MPI error checking.

MVAPICH High-performance implementation of the MPI standard, developed by the Ohio State University and the University of Tennessee.

NAS Parallel Benchmarks Set of benchmarks to mimic the computational and data access behaviour of large scale computational fluid dynamics developed by NASA Ames Research Center.

nccopy Command-line tool to copy and compress netCDF files.

ncdump Linux command line tool to print metadata and data of a netCDF file.

netCDF Network Common Data Form.

Nobrainier Matches and replaces code snippets using an AST matcher.

NP-hard Refers to a class of computational problems that are at least as challenging as the hardest problems in the class NP (nondeterministic polynomial time), indicating they are difficult to solve efficiently.

Omni OpenMP S2S compiler intended to be used on SMP cluster with focus on the AST.

OmpSs-2 Programming model with an OpenMP-like approach using a thread pool instead of a fork-join model.

OP2 DSL specialised for unstructured grids.

Open MPI Open-source implementation of the MPI standard, developed and maintained by a community of contributors from academia, research institutions.

OpenACC Open Accelerators, a programming standard that simplifies the process of parallelising code for heterogeneous computing systems, such as GPUs, by providing directives to offload computations to accelerators.

OpenARC S2S research compiler framework with focus on OpenACC.

OpenCL Open Computing Language, an open standard for programming heterogeneous computing platforms, enabling developers to write code that efficiently utilises CPUs, GPUs, and other accelerators for parallel computation tasks.

OpenMP Open Multi-Processing, an compiler based API specification for parallel programming that allows developers to write multi-threaded and shared-memory parallel applications, enabling efficient utilisation of multi-core processors.

OpenMPD Language extension prototype to provide compiler directives, which are translated into MPI code.

OpenSHMEM PGAS standard, an alternative to MPI.

OSU Microbenchmarks Created by Ohio State University, encompasses independent MPI performance benchmarks, evaluating latency, bandwidth, and host overhead for both traditional and GPU-enhanced nodes.

partdiff Partial differential equation solver using iterative methods like Jacobi and Gauss-Seidel.

pass Program transformation or analysis performed on the IR of a program, enabling optimisations or collecting information for further compilation stages.

Pass Manager An LLVM compiler infrastructure component that orchestrates the execution of a sequence of LLVM passes for optimising or analysing the IR of a program.

PATUS DSL specialised for structured grids.

pedosphere Layer of the Earth composed of soil, including its formation, composition, and interactions with other spheres.

Perl Flexible, high-level programming language known for its text processing capabilities, regular expression support, and extensive collection of modules, making it suitable for a wide range of tasks including system administration, web development, and data manipulation.

PhASAR Static Analysis Framework based on LLVM.

PLuTo S2S framework for high-level loop and data-locality optimisations.

Polly LLVM framework for high-level loop and data-locality optimisations.

Prospect S2S compiler framework to build two-phase applications: a slow and robust version and a faster but more instable version.

Python High-level, dynamically-typed programming language known for its simplicity, readability, and vast collection of libraries, making it popular for a wide range of applications from web development to data analysis.

QAT *Intel QuickAssist* is a hardware-based technology to accelerate compression and encryption.

Rellume Decompiler from binary to IR.

RetDec LLVM decompiler to transform a binary into high-level code.

Rodinia Benchmark suite for heterogeneous computing consisting of compute kernels inspired by Dwarves.

ROSE S2S compiler with focus on large scale, robustness, and automatic parallelisation, which is funded by the US Department of Energy.

ScalaBenchGen Tool for auto-generation of communication benchmarks traces.

sed Stream Editor, command-line text manipulation tool used for performing automated editing operations on streams of text, including file processing, search and replace, and line-by-line transformations.

SKaMPI Benchmark of MPI implementations.

Slurm Open source job scheduler for HPC systems.

spack Package management tool to build various versions and configurations of HPC software on a wide variety of platforms and environments.

SPar DSL to exploit stream parallelism in C++.

Spectrum Scale Parallel FS from IBM to support parallel I/O on HPC systems.

StressBench Configurable full system network and I/O benchmark framework.

SZ Lossy compressor using a multidimensional prediction model.

Szip Lossless compressor.

TBB *Threading Building Blocks* is a C++ library developed by Intel for parallelisation on shared memory.

Top500 A list that ranks and benchmarks the world's most powerful supercomputers based on their performance on the LINPACK benchmark, providing insights into the state of high-performance computing worldwide.

TreadMarks Software implementation of an SSI, to provide a virtual shared memory of all nodes.

Unified Parallel C PGAS implementation.

Valgrind Framework for dynamic binary instrumentation.

WR-Cluster Research cluster of the research group *Scientific Computing* at University of Hamburg.

XcalableMP Language extension of C and Fortran offering OpenMP-like compiler directives to utilise distributed memory.

Xevolver S2S compiler framework based on ROSE to work on an XML representation of the AST.

ZFP Compressor performing a normalisation on chunks, to improve compressibility.

ZFS FS with advanced features like compression.

Zlib Compressor, which uses the *DEFLATE* compression algorithm.

Zstandard Very fast and efficient. lossless data compressor developed at Facebook

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.

Hamburg, 05.09.2023

Ort, Datum


Unterschrift