Holistic Performance Analysis of Multi-layer I/O in Parallel Scientific Applications

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Ronny Tschüter
geboren am 21. Dezember 1983 in Görlitz

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Wolfgang E. Nagel

Tag der Abgabe: 06. Oktober 2020
Tag der Verteidigung: 16. Februar 2021

Acknowledgments

I want to thank Prof. Dr. Wolfgang E. Nagel for giving me the opportunity to work on this thesis. Also, I would like to thank my reviewer Prof. Dr. Thomas Ludwig for his advise and feedback.

Furthermore, special thanks go to Christian Herold, Sebastian Oeste, Bill Williams, and Robert Dietrich for their valuable feedback, support, and ideas. Especially, I am grateful to Matthias Weber and Franziska Kasielke for their encouraging, their time and effort spent in reviewing my dissertation, and for providing valuable criticisms and suggestions.

Finally, very special thanks go to my family for their unconditional long term support.

Abstract

Efficient usage of file systems poses a major challenge for highly scalable parallel applications. The performance of even the most sophisticated I/O subsystems lags behind the compute capabilities of current processors. To improve the utilization of I/O subsystems, several libraries, such as HDF5, facilitate the implementation of parallel I/O operations. These libraries abstract from low-level I/O interfaces (for instance, POSIX I/O) and may internally interact with additional I/O libraries. While improving usability, I/O libraries also add complexity and impede the analysis and optimization of application I/O performance.

This thesis proposes a methodology to investigate application I/O behavior in detail. In contrast to existing approaches, this methodology captures I/O activities on multiple layers of the I/O software stack, correlates these activities across all layers explicitly, and identifies interactions between multiple layers of the I/O software stack. This allows users to identify inefficiencies at individual layers of the I/O software stack as well as to detect possible conflicts in the interplay between these layers. Therefor, a monitoring infrastructure observes an application and records information about I/O activities of the application during its execution. This work describes options to monitor applications and generate event logs reflecting their behavior. Additionally, it introduces concepts to store information about I/O activities in event logs that preserve hierarchical relations between I/O operations across all layers of the I/O software stack.

In combination with the introduced methodology for multi-layer I/O performance analysis, this work provides the foundation for application I/O tuning by exposing patterns in the usage of I/O routines. This contribution includes the definition of I/O access patterns observable in the event logs of parallel scientific applications. These access patterns originate either directly from the application or from utilized I/O libraries. The introduced patterns reflect inefficiencies in the usage of I/O routines or reveal optimization strategies for I/O accesses. Software developers can use these patterns as a guideline for performance analysis to investigate the I/O behavior of their applications and verify the effectiveness of internal optimizations applied by high-level I/O libraries.

After focusing on the analysis of individual applications, this work widens the scope to investigations of coordinated sequences of applications by introducing a top-down approach for performance analysis of entire scientific workflows. The approach provides summarized performance metrics covering different workflow perspectives, from general overview to individual jobs and their job steps. These summaries allow users to identify inefficiencies and determine the responsible job steps. In addition, the approach utilizes the methodology for performance analysis of applications using multi-layer I/O to record detailed performance data about job steps, enabling a fine-grained analysis of the associated execution to exactly pinpoint performance issues. The introduced top-down performance analysis methodology presents a powerful tool for comprehensive performance analysis of complex workflows.

On top of their theoretical formulation, this thesis provides implementations of all proposed methodologies. For this purpose, an established performance monitoring infrastructure is enhanced by features to record I/O activities. These contributions complement existing functionality and provide a holistic performance analysis for parallel scientific applications covering computation, communication, and I/O operations. Evaluations with synthetic case studies, benchmarks, and real-world applications demonstrate the effectiveness of the proposed methodologies. The results of this work are distributed as open-source software. For instance, the measurement infrastructure including improvements introduced in this thesis is available for download and used in computing centers world-wide. Furthermore, research projects already employ the outcomes of this work.

Contents

1 Introduction

This chapter provides an introduction to the field of High Performance Computing (HPC) and motivates the relevance of performance analysis and optimization within this context. The chapter identifies open challenges for performance analysis of highly-parallel input/output (I/O) intensive applications and outlines the contributions of this thesis. Finally, it gives an overview of subsequent chapters.

This thesis proposes contributions to the holistic performance analysis of multi-layer I/O in parallel scientific applications, in particular in the field of High Performance Computing (HPC). The contributions complement established analysis techniques and therefore support users in holistic performance engineering. I/O operations are a major bottleneck with respect to performance of data intensive applications [87]. Therefore, performance analysis and optimization is critical in this field to identify inefficiencies and improve the utilization of available HPC resources. Scientific applications often employ a complex software stack including I/O libraries. Especially high-level I/O libraries abstract from details of complex storage hardware architectures, thereby facilitate the software development process, but make I/O performance analysis more challenging. Software developers require tool support to investigate complex interactions between I/O libraries and user code. This work describes methods to capture performance data related to I/O activities on multiple layers of the I/O software stack, correlate these activities across all layers explicitly, and identify interactions between multiple layers of the I/O software stack. This thesis demonstrates that the presented methods allow users to investigate the I/O behavior of individual scientific applications. Furthermore, this work proposes an approach to study multiple applications that are arranged in a workflow and coupled via their I/O activities and thereby shows that the illustrated methods lay the foundation of more complex analyses.

This chapter starts with a brief introduction to HPC and performance analysis. After describing open challenges in the analysis and optimization of the I/O performance of parallel scientific applications, the chapter lists contributions of this work and concludes with an overview of subsequent chapters.

1.1 High Performance Computing

Computing has become a substantial part of science and industry to gain knowledge, drive innovation, and preserve competitiveness [95]. Simulations based on mathematical models allow researchers of various disciplines to study effects of alternative conditions. For example, scientists build entire simulated universes, study molecular dynamics, and investigate materials designs. The complexity of such simulations demands significant computing power.

Supercomputers deliver this high level of performance. These computing machines earn their title of a supercomputer by providing significantly more processing power in comparison with the fastest personal computer or workstation at the time. The field of supercomputing has a long history. The Control Data Corporation (CDC) 6600 is often considered as the first supercomputer. In 1964 Seymour Cray and his colleagues completed the CDC 6600. In the early days of supercomputing, custom hardware was the norm to realize these machines. However by the mid-1990s, the trend had shifted towards commodity hardware as the performance of general-purpose devices such as Central Processing Units (CPUs) had improved. Nowadays, supercomputers feature tens of thousands of CPUs, often equipped with specialized accelerator hardware such as Graphic Processing Units (GPUs). On the one hand, advancements in digital electronics drive the superior performance of supercomputers. In the past decades, the number of transistors in integrated circuit chips increased according to Moore's Law [70] which also resulted in a growing computational performance. On the other hand, supercomputers provide their immense perfor-

mance by delivering a large number of compute resources. Consequently, users have to parallelize their applications in order to leverage the full potential of supercomputers.

Supercomputing, also referred to as High Performance Computing (HPC), is not only about providing powerful computing capabilities but also represents an independent research area. Among others, this field of science covers innovative methods to provide energy efficient compute infrastructures, the design of high-speed interconnects to form a powerful system based on multiple individual components, the design of reliable high-performance storage solutions to archive large data volumes, the implementation of efficient computational simulations, as well as the performance analysis and optimization of scientific parallel applications. In general, computational science profits from synergies between hardware and software development. Advances in hardware result in increased compute power and allow scientists to implement more high-fidelity simulation techniques.

1.2 Performance Analysis and Optimization

Performance analysis and optimization are essential aspects of computational science. Since the beginning of computer engineering humans have striven towards analyzing and optimizing the performance of these machines. The quote of Charles Babbage about the design of the Difference Engine, an automatic mechanical calculator designed to tabulate polynomial functions, confirms this statement.

> The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.
>
> (Charles Babbage (1791 - 1871))

The task of performance analysis and optimization often splits into two phases. The first phase, the *performance analysis*, includes measurements and delivers readings of performance relevant metrics, such as execution time or number of function calls. These readings allow users to compare performance values with their expectations and identify performance relevant components of a program, e.g., functions where most execution time is spent in. Furthermore, performance analysis assists users in detecting causes of performance insufficiencies and thereby revealing options for optimization.

In the second phase, the *performance optimization*, users adapt their applications based on the knowledge gained by the analysis. In general, the goal of performance analysis is to improve the efficiency of resource utilization.

In addition to traditional performance factors such as the choice of appropriate algorithms and compiler flags, parallel applications offer further aspects that represent potential candidates for tuning. Parallel programs apply a domain decomposition to distribute data across multiple processing elements such as processes or threads. The method used to achieve this partitioning is a performance critical factor. Furthermore, during their execution parallel applications have to communicate, e.g., to exchange intermediate results between individual processes/threads. On shared memory systems, parallel programs can transfer data between processing elements such as threads directly via memory. OpenMP [80] is the de facto standard application programming interface (API) for shared memory parallel computing [16, 42]. On distributed memory architectures, processing elements can use the message passing paradigm to communicate. The Message Passing Interface (MPI) [71] is a de facto standard API for distributed memory programming [38, 27]. Further performance relevant aspects of parallel applications are synchronization and lock operations to coordinate the execution of code on different processes/threads and protect the access to shared resources.

Due to the increasing complexity of software and hardware architectures, performance analysis and optimization on state-of-the-art systems is not trivial [48, 83]. Consequently, tools exist that assist users in both tasks. These tools aid users in conducting measurements, analyzing their applications, identifying inefficiencies in current implementations, and making decision about tuning options.

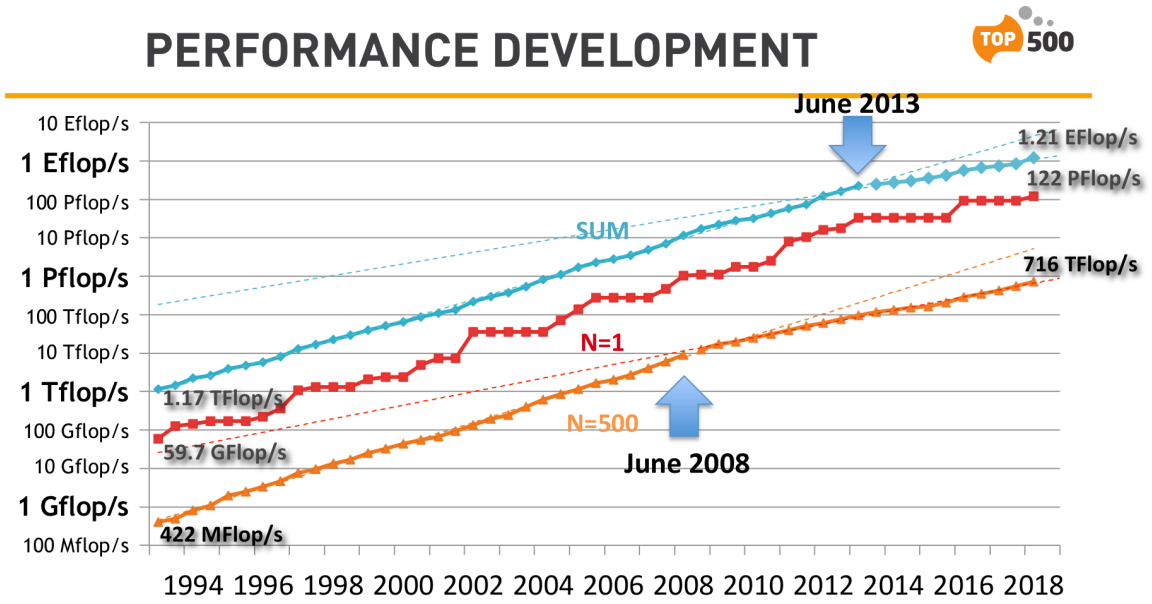


Figure 1.1: The TOP500 list ranks HPC systems by their computational performance and thereby reports general trends in the field of supercomputing. The graph illustrates a slow down in the performance development of systems ranked 500^{th} (June 2008) resp. 1^{st} (June 2013). (Taken from [109])

1.3 Challenges for Performance Analysis of The Multi-layered I/O Software Stack

Section 1.2 introduced the field of performance analysis and optimization in general. Software developers often optimize their applications with a focus on the computational performance. However, as scientific applications often process large volumes of data, the I/O behavior also represents a performance critical aspect for this kind of applications. This section discusses current trends in compute and I/O capabilities of HPC systems and thereby illustrates the importance of tuning the I/O behavior of applications. Additionally, this paragraph presents challenges in analyzing I/O operations of parallel scientific applications and motivates a holistic performance engineering approach incorporating a wide range of performance relevant aspects such as I/O activities, computation, and communication.

Based on the results of the LINPACK [85] benchmark, the TOP500 list [110] ranks general purpose computing systems and documents the trend towards exascale computing. First systems with computational capabilities of at least one exaflops (10^{18} double-precision floating point operations per second) are expected to arrive in 2020/21 [77, 76]. Figure 1.1 shows two drops in the general trend of performance development in the TOP500 list over the years. In 2008, the performance increase of the system ranked last in the list started to slow down compared with previous years. Since 2013 the same trend has applied to the system ranked 1^{st}. Consequently, there are factors that limit the performance increase. Operations to read/write data from/to storage systems are one of these limiting factors. Increasing computational capabilities of modern High Performance Computing (HPC) systems enable fine-granular simulations. The more fine-granular simulations are the more they tend to generate increasing data volumes. This fact put more pressure on I/O subsystems rendering the I/O behavior of scientific applications crucial for optimal performance. Present research topics, such as *Machine Learning* and *Big Data*, increase the trend of processing large data volumes. In 2008, the *ExaScale Computing Study* [11] identified four major challenges in order to realize exascale supercomputers. According to this study, the need to improve performance of *Memory and Storage* is one of these challenges.

> The Memory and Storage Challenge concerns the lack of currently available technology to retain data at high enough capacities, and access it at high enough rates, to support the desired application suites at the desired computational rate, and still fit within an acceptable power envelope. This information storage challenge lies in both main memory (DRAM today) and in secondary storage (rotating disks today).
>
> ([11, p. 2])

A comparison of the last two HPC systems installed at the Oak Ridge Leadership Computing Facility (OLCF) illustrates this trend. *Titan* [31] debuted in the TOP500 list in November 2012 as the world fastest supercomputer. Its successor *Summit* [30] ranked 1^{st} in the TOP500 list from June 2018 to June 2020. Table 1.1 shows that the maximum bandwidth to the file system increased by a much lower rate in comparison to compute capabilities and storage capacities. Highly optimized benchmarks evaluate the practical I/O peak performance which in best case scenarios comes close to the theoretical maximum. However, real-world applications often achieve even less I/O performance, e.g., due to sub-optimal file access patterns. Figure 1.2 depicts the *Titan* and *Summit* supercomputers.

Achieving high I/O performance is no trivial task for program developers. To leverage the potential of parallel I/O subsystems, scientific applications have to parallelize their I/O operations. I/O libraries, such as HDF5 [108], NetCDF [114], and MPI I/O [72, Chapter 13] have evolved to support software developers in implementing parallel I/O operations. On the one hand, these I/O libraries abstract from low-level I/O interfaces and, thereby, simplify the integration of parallel I/O. On the other hand, utilizing I/O libraries does not automatically guarantee efficient I/O resource utilization. Additionally, the enhanced usability accomplished by abstraction also impedes an I/O performance analysis as complex interactions between I/O libraries and user code impact each other.

Figure 1.3 illustrates the interaction of an application and multiple I/O libraries. In this example, the application directly calls NetCDF, MPI I/O, and POSIX I/O routines. Additionally, the NetCDF library issues HDF5 function calls. HDF5 in turn utilizes MPI I/O and POSIX I/O routines. Calls to I/O routines of a high-level library will also cause I/O operations at lower levels. For example, in case of writing data each I/O layer may rearrange operations. Gathering information from all involved I/O layers is essential to evaluate the effectiveness of the resulting I/O operations. Performance analysis becomes even more challenging with the increasing complexity of I/O subsystems, e.g., additional layers in the storage hierarchy like Non-Volatile Memory (NVMe) storage devices. In 2019, the *Report for the DOE ASCR Workshop on Storage Systems and I/O* [87] confirmed that I/O is still a major bottleneck with respect to application performance.

> In fact, I/O is now widely recognized as a severe performance bottleneck for both simulation and data analysis, and this bottleneck is expected to worsen with an order of magnitude increase in the disparity between computation and I/O capacity on future exascale machines.
>
> ([87, p. 19])

Table 1.1: Key features of the HPC systems *Titan* and *Summit* illustrating the widening gap between compute and I/O capabilities.

Feature	Titan (Debut in 2012)	Performance Increase	Summit (Debut in 2018)
Peak Flops (double-precision)	$27\ PF$	$7\times$	$200\ PF$
Total System Memory	$710\ TB$	$14\times$	$10\ PB$
File System Capacity	$32\ PB$	$8\times$	$250\ PB$
Maximum Bandwidth to File System	$1\ TB/s$	$2.5\times$	$2.5\ TB/s$

(a) The *Titan* supercomputer (Cray XK7 system equipped with AMD Opteron processors and NVIDIA Kepler K20X GPUs, theoretical peak performance of 27 petaflops). (Taken from [31])

(b) The *Summit* supercomputer (IBM system equipped with IBM POWER9 processors and NVIDIA Volta V100 GPUs, theoretical peak performance of 200 petaflops). (Photo by OLCF at ORNL / CC BY 2.0)

Figure 1.2: Illustration of the (a) *Titan* [31] and (b) *Summit* [30] supercomputers installed at the Oak Ridge Leadership Computing Facility (OLCF). *Titan* debuted in 2012 and *Summit* in 2018. In comparison with its predecessor, *Summit* provides an significant increase in compute performance as well as available memory and storage volume. However, the maximum bandwidth to the file system did not increase to the same extend.

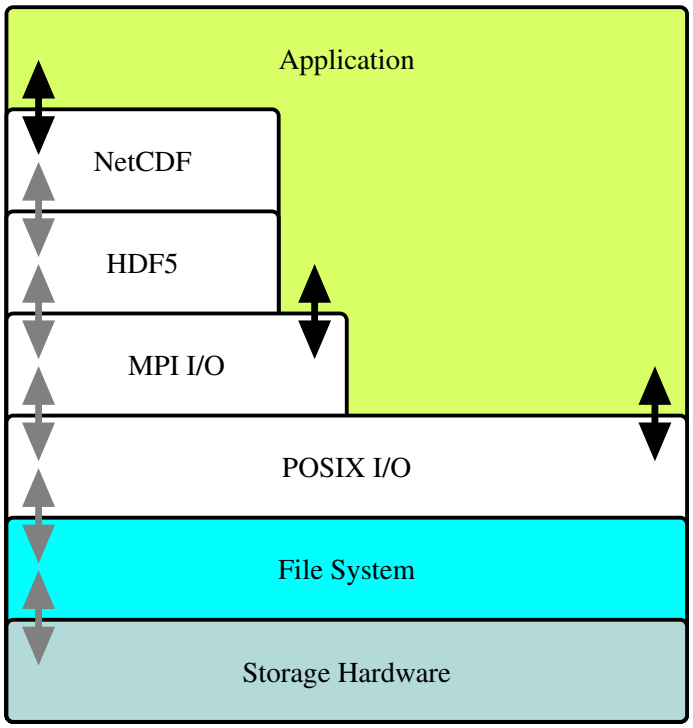


Figure 1.3: Example of an application interacting with multiple I/O libraries. Black arrows indicate calls from the application to the I/O libraries NetCDF, MPI I/O, and POSIX I/O. Grey arrows illustrate internal dependencies between the I/O libraries.

Currently, there is a deficiency in tools supporting software developers in tuning their applications for optimal I/O performance. Performance analysis cannot only focus on specific aspects of an application, e.g., communication, as this won't provide full insights. Limiting investigation to specific aspects may show an effect but often does not reveal the root cause of a performance problem. Thus, I/O performance analysis has to incorporate aspects of computation, communication, and synchronization. Therefore, this thesis proposes a methodology for holistic performance analysis of multi-layer I/O in parallel scientific applications.

1.4 Contributions of This Thesis

This thesis presents a contribution to the performance analysis of parallel scientific applications based on event logs gathering input/output (I/O) operations of these applications. It focuses on application-centric collection of runtime performance information within the scope of High Performance Computing (HPC). This work provides missing functionalities for recording and correlating activities across the entire I/O software stack. Results of this work guide software developers in identifying inefficiencies at individual layers of the I/O stack and detecting possible conflicts in the interplay between layers. Furthermore, this thesis proposes a methodology for holistic performance analysis of multi-layer I/O in parallel scientific applications. This methodology incorporates other performance relevant aspects such as computation, communication, and synchronization in the process of I/O analysis. The following contributions are made.

1.4.1 A Methodology for Performance Analysis of Applications Using Multi-layer I/O

This work proposes a methodology to investigate application I/O behavior in detail [111]. In contrast to existing approaches, this methodology captures I/O activities on multiple layers of the I/O software stack, correlates these activities across all layers explicitly, and identifies interactions between multiple layers of the I/O software stack. This allows users to identify inefficiencies at individual layers of the I/O software stack as well as to detect possible conflicts in the interplay between these layers. Therefore, a monitoring infrastructure observes an application and records information about I/O activities of the application during its execution. This work describes options to monitor applications and generate event logs reflecting the application behavior. Additionally, this thesis introduces concepts to store information about I/O activities in event logs that preserve hierarchical relations between I/O operations across all layers of the I/O software stack.

1.4.2 Definition of Multi-layer I/O Access Patterns

In combination with the methodology for multi-layer I/O performance analysis, this work provides the foundation for application I/O tuning by exposing patterns in the usage of I/O routines. This thesis defines I/O access patterns observable in the event logs of parallel scientific applications. These access patterns originate either directly from the application or from utilized I/O libraries. On the one hand, there are patterns reflecting inefficiencies in the usage of I/O routines. On the other hand, this work defines patterns that reveal optimization strategies of I/O accesses. Software developers can use these patterns as a guideline for performance analysis to investigate the I/O behavior of their applications and verify the effectiveness of internal optimizations applied by high-level I/O libraries.

1.4.3 A Top-Down Performance Analysis Methodology for Workflows

After focusing on the analysis of individual applications, this work widens the scope to investigations of coordinated sequences of applications. This work introduces a top-down approach for performance analysis of entire scientific workflows [112]. The approach provides summarized performance metrics

covering different workflow perspectives, from general overview to individual jobs and their job steps. These summaries allow users to identify inefficiencies and determine the responsible job steps. In addition, the approach utilizes the methodology for performance analysis of applications using multi-layer I/O to record detailed performance data about job steps, enabling a fine-grained analysis of the associated execution to exactly pinpoint performance issues. The introduced top-down performance analysis methodology provides a powerful tool for comprehensive performance analysis of complex workflows.

1.4.4 Implementation of the Proposed Methodologies

Additionally to their theoretical introduction, this thesis describes implementations of all proposed methodologies and thereby demonstrates their effectiveness. For this purpose, this work enhances an established performance monitoring infrastructure and realizes features to record I/O activities in this monitoring system. In this way, contributions of this thesis complement existing functionality and provide a holistic performance analysis for parallel scientific applications covering computation, communication, and I/O operations. Event logs recorded by the enhanced performance monitoring infrastructure are the foundation for the implementation of the top-down performance analysis methodology for scientific workflows. The chosen performance monitoring infrastructure Score-P is publicly available as open-source software and is used in computing centers world-wide. Since version 6.0 the official Score-P release includes features of this thesis. In addition, results of this work are already used in research projects such as “Next Generation I/O for the Exascale” (NEXTGenIO) within the European Union’s Horizon 2020 Research and Innovation programme and “Advanced Data Placement via Ad-hoc File Systems at Extreme Scales” (ADA-FS) within the German Priority Programme 1648 Software for Exascale Computing (SPPEXA).

1.5 Thesis Organization

The chapters of this thesis are organized as follows: Chapter 2 provides an introduction into the process of performance analysis as well as techniques to acquire, record, and analyze performance data of parallel applications. Additionally, this chapter presents related work in the field of application monitoring with a special focus on I/O analysis tools. Chapter 3 introduces methodologies for performance analysis of applications using multi-layer I/O activities. First, this chapter provides solutions to monitor parallel applications and acquire performance relevant information about their I/O operations. Afterwards, the chapter introduces concepts to store recorded information while preserving hierarchical relations between observed I/O activities. Finally, this chapter describes analyses based on recorded performance data. It defines detectable I/O access patterns of parallel applications and presents a methodology for a top-down performance analysis of multiple applications forming a scientific workflow. Chapter 4 details the implementation of all methodologies described in Chapter 3. Chapter 5 uses synthetic stress tests and benchmarks to evaluate the applicability of the multi-layer I/O activity recording and workflow analysis approaches. Chapter 6 summarizes this work and shows possible directions for further developments.

2 State-of-the-art and Related Work

This chapter introduces fundamental concepts of performance analysis. It provides an overview of related work and established tools in the field of I/O performance analysis. The chapter concludes with an analysis of existing work and outlines missing functionality required for a holistic performance analysis of multi-layer I/O in parallel scientific applications.

Software development of scientific applications is challenging. The software development process includes mathematical modeling of problems, converting these models into statements of a programming language, as well as generating, visualizing, and interpreting result data [49]. Especially, software developers have to ensure correctness of their simulations and need to parallelize as well as optimize their software in order to leverage resources of modern High Performance Computing (HPC) systems efficiently. Fortunately, a wide range of tools support developers in these tasks. This work briefly covers aspects of software correctness and details on performance analysis of parallel scientific applications with a special focus on their I/O behavior. Consequently, results of this thesis support developers of parallel scientific applications in performance analysis and optimization (Figure 2.1).

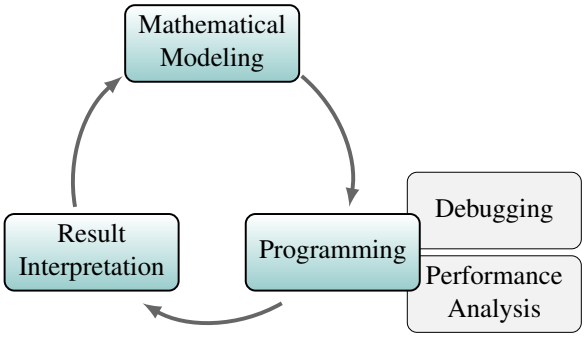


Figure 2.1: This thesis assists developers of parallel scientific applications in their programming tasks. Results of this work especially support software developers in performance analysis and optimization.

Debuggers aid software developers in testing applications and examining errors. It is not trivial to ensure correctness of software, not even for serial applications. Debuggers such as GNU debugger (gdb) [33] and LLDB [102] proofed as valuable tools in this regard. Parallel-programming paradigms introduce additional challenges. The concurrent execution of multiple threads and/or processes result in non-deterministic behavior. Parallel debuggers such as Arm DDT [5] and Rogue Wave TotalView [96] support users in debugging parallel applications. The Stack Trace Analysis Tool (STAT) [7] facilitates debugging of highly parallel applications by recording stack traces of multiple processes and identifying groups with similar behavior. Subsequently, software developers need to analyze only representatives of each group instead of each process individually. There are also more specialized tools such as ThreadSanitizer [101], Intel Inspector [24], and Archer [9] focusing on detection of data races and deadlocks in multi-threaded applications, or MUST [44] validating the correct usage of the MPI parallel-programming paradigms.

Performance analysis and optimization is another important aspect of the software development process. Because performance optimization is not feasible for faulty programs, this work assumes observed applications to be correct. Otherwise, developers can investigate errors in their applications using the debugging tools mentioned above. Parallel scientific applications have to utilize resources of HPC systems

efficiently to leverage their potential. This thesis focuses on performance analysis and optimization of parallel scientific applications.

The remainder of this chapter starts with an overview of the I/O subsystem of supercomputers (Section 2.1) and presents challenges for applications to efficiently utilize available I/O resources. Afterwards, an introduction to performance analysis (Section 2.2) provides the fundamental concepts of this thesis. The chapter details on the individual steps involved in the process of performance analysis. Section 2.2.1 presents methods to capture performance data from a running application. Section 2.2.2 describes data formats of the recorded performance data. Section 2.2.3 illustrates options to analyze and visualize performance data. Finally, Section 2.3 provides an overview of related work in the field of I/O performance analysis.

2.1 The I/O Subsystem in High Performance Computing Machines

Current HPC systems provide sophisticated compute and storage hardware. Modern supercomputers consist of thousands of compute nodes connected via a high performance network. To utilize these resources scientific applications parallelize their computations and distribute the workload across multiple compute nodes.

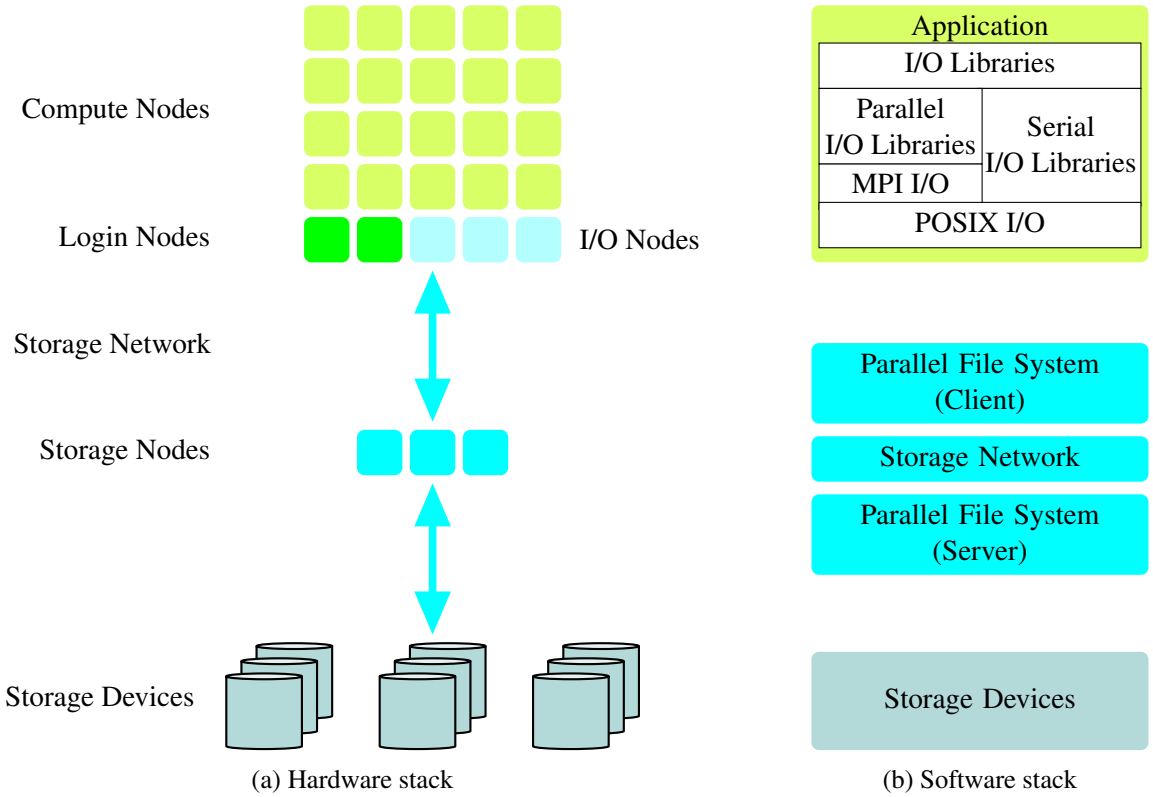
As shown in Figure 2.2a, also the I/O subsystem of typical HPC systems consists of a complex hardware stack. On some HPC systems I/O nodes collect and rearrange I/O requests of the applications running on the compute nodes. Afterwards, the I/O nodes forward the requests via a storage network to the storage nodes. Storage nodes manage the access to the storage devices. Supercomputers offer a multitude of storage devices, often of different kinds such as disk drives (HDD) or flash drives (SSD). In order to leverage the potential of these storage devices, e.g., gaining maximum bandwidth for data transfers, data must be distributed across the available devices. Therefore, similar to handling their computations, applications have to parallelize their I/O operations to utilize I/O resources efficiently. Section 2.1.2 describes common I/O strategies used by parallel applications.

At several levels software assists applications in using the I/O subsystem and thereby also constitutes a complex stack. Figure 2.2b depicts the I/O software stack and its relations to the components of the I/O hardware stack. For instance, I/O libraries offer applications an interface for data transfer operations. Some libraries focus on basic I/O operations, e.g., POSIX I/O, other libraries address challenges in implementing efficient parallel I/O routines. As already mentioned in Section 1.3 (see Figure 1.3) I/O libraries can depend on each other and form complex interactions. Parallel file systems handle the distribution of data across storage devices. Section 2.1.1 describes the concepts of parallel file systems. The implementation of parallel file systems can follow different approaches. On the one hand, some parallel file systems are entirely implemented as a module of the operating system running on the compute nodes. On the other hand, parallel file systems can employ a client-server architecture. This kind of parallel file systems split their service into distinguished components. A module of the operating system implements the client. The server-side components run on the I/O subsystem. Clients and servers communicate via network. Therefore, Figure 2.2b shows two boxes representing parallel file systems.

Each layer of the I/O hardware and software stack affects the I/O performance of applications. For instance, routing algorithms and packet sizes used by the storage network, block sizes of the physical storage devices, or the efficient mapping of high-level I/O APIs onto low-level I/O library routines are critical aspects for achieving best I/O performance. This thesis focuses on the analysis at the application and library level. However, this work also shows that the proposed methodology allows users to integrate information obtained from the other levels.

2.1.1 Concepts of Parallel File Systems

A file system controls how data is stored on physical devices. Therefore, the file system manages data placement on the storage devices. Furthermore, it provides an API that abstracts from details about physical storage information and allows users to intuitively access their data. Via this API users keep their



(a) Hardware stack (b) Software stack

Figure 2.2: This figure illustrates the (a) hardware and (b) software stack of typical HPC systems focusing on components related to I/O operations.

data in files, assign names to files in order to facilitate their handling, and organize files within hierarchical directories. The next paragraphs introduce the terminology used in this thesis as the nomenclature with respect to file system techniques is ambiguous.

Distributed File Systems The development of distributed file systems (also referred to as Network File Systems) dates back to the 1980s. At this time, networks of workstations were popular and created needs to share data among these clients. Distributed file systems follow a client/server based architecture. The server hosts a local file system and offers it to clients connected via a network. An individual file is stored on a single server. Consequently, the network bandwidth of this server limits the peak I/O bandwidth for accessing the corresponding file.

Parallel File Systems In contrast to distributed file systems, parallel file systems incorporate multiple storage devices potentially located on different servers. This enables a parallel file system to split a single file up and write its portions across multiple servers. This mechanism is called *striping* and represents an important feature of parallel file systems to scale storage bandwidth and capacity. The next section takes Lustre as an example and explains the concept of a parallel file system.

Lustre The open-source parallel file system Lustre [64] is widely used on HPC systems and employs a client-server network architecture. Figure 2.3 illustrates the major components of a Lustre file system cluster.

Both clients and servers are implemented as loadable modules within the Linux kernel. Clients and servers communicate via network using the Lustre Network protocol (LNet). The servers provide a POSIX-compliant file system which is mounted by the **Lustre Clients**. Applications running on the clients use Lustre I/O features via standard POSIX system calls. However, the Lustre client forwards

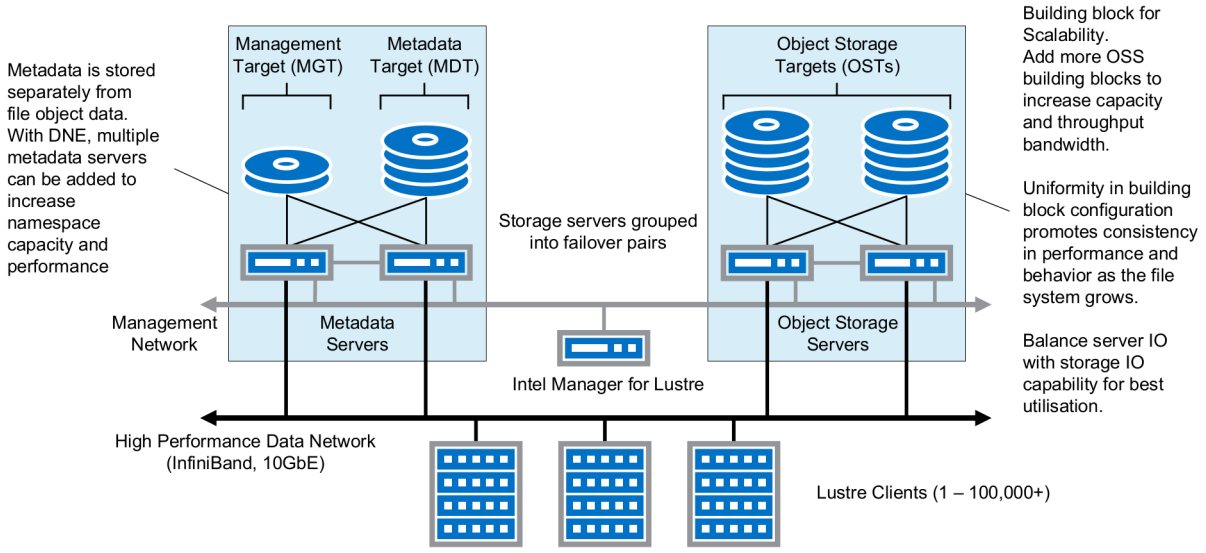


Figure 2.3: Architecture of the Lustre storage platform. (Taken from [63])

the I/O requests to the servers via network. Therefore, the client translates the POSIX system calls into Remote Procedure Calls (RPCs). The servers process the requests and send their responses to the clients.

The server components are responsible for three major tasks: general service management, metadata handling, and payload storage. **Management Servers (MGS)** maintain a registry of active Lustre servers and clients. **Management Targets (MGT)** store associated configuration information.

Metadata Servers (MDS) provide the file system name space and handle metadata operations of the clients. For instance, whenever a client creates, opens, closes, or deletes a file or manipulates its permissions, a MDS handles the associated request. **Metadata Targets (MDT)** store information with respect to metadata management. Lustre allows users to install multiple Management Servers and Targets and thereby scale the metadata management service.

Object Storage Servers (OSS) handle data transfer operations and manage data storage. **Object Storage Targets (OST)** represent physical devices providing storage space. Lustre supports striping of files. A file is split into several stripes and these stripes are distributed across multiple OSTs. Hence, users can adjust throughput and capacity of the Lustre file system by scaling the number of Object Storage Servers and Targets.

2.1.2 I/O Strategies of Parallel Applications

Parallel applications may apply different strategies to perform their I/O operations. This section introduces two common strategies: performing I/O operations in *serial* or in *parallel*.

The Serial I/O Strategy As shown in Figure 2.4, in the serial I/O strategy a single process acts as proxy for all I/O operations of an application. For instance, if a parallel application wants to write its results to a file, only the proxy process opens the file. As the proxy process is responsible for all file operations, it has to collect data from the remaining processes. Either the proxy process has direct access to the data in memory or the data needs to be exchanged, e.g., via message transfers. Dotted lines in Figure 2.4 illustrate this data exchange from $Process\ 1-3$ to $Process\ 0$. Afterwards, the proxy process starts writing results to the I/O subsystem. Therefore, the proxy interacts with the parallel file system via API routines.

In the serial I/O strategy, the capabilities of the single proxy process limit the I/O performance. For instance, as the proxy process runs on a single compute node, the bandwidth of data transfers is limited by the network bandwidth of this node. Consequently, this strategy does not scale for highly parallel

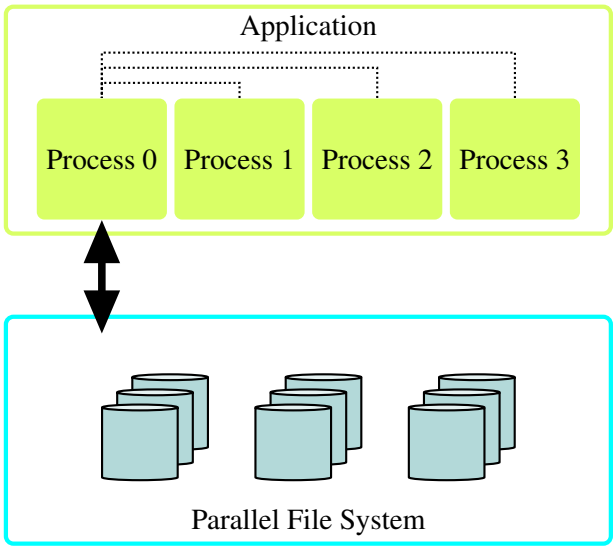


Figure 2.4: A parallel application performing its I/O operations in serial. A single process collects the I/O requests of an application and forwards them to the I/O subsystem.

applications. In order to handle large data volumes and leverage full potential of HPC systems, scalable applications not only have to parallelize their computations but also their I/O activities. However, the serial I/O strategy is often used by applications due to its ease of use.

The Parallel I/O Strategy Figure 2.5 illustrates the concept of a parallel I/O strategy. Multiple processes of an application issue their I/O requests in parallel to the the file system. With this strategy an application profits from improved resource utilization and performance and leverages the potential of parallel I/O subsystems. For instance, an application can increase its bandwidth to the I/O subsystem by utilizing multiple links to the storage backend.

During program execution an application keeps its data structures in main memory. From time to time the application may transfer data to the I/O subsystem for persistent storage. However, on storage devices data is typically organized as a stream of bytes. Therefore, the application has to map its data layout in main memory and the file layout on the storage devices while writing/reading data to/from the I/O subsystem. Furthermore, in a parallel application data is distributed across multiple processes. Therefore, a parallel application has also several options to collate data into a file representation. Figure 2.6 illustrates two common approaches to address both tasks.

Figure 2.6a shows the *file-per-process* approach. Each process of a parallel application operates on its individual file. Therefore, each process can perform its I/O operations independent of other processes. However, the number of files scale with the number of processes which increases the demands on the I/O subsystem. For instance, as each process opens/closes its corresponding file the parallel file system has to handle a large number of metadata operations. The file-per-process approach often involves a post-processing step to merge multiple files for later analysis.

In contrast, Figure 2.6b depicts the *shared-file* approach. Multiple processes work on a single logical file. On the one hand, this approach reduces the number of used files. On the other hand, processes of a parallel application have to coordinate their I/O operations. For instance, when multiple processes write to a single file, each process has to determine its individual offset within the file. Otherwise, a process might overwrite data written by another one.

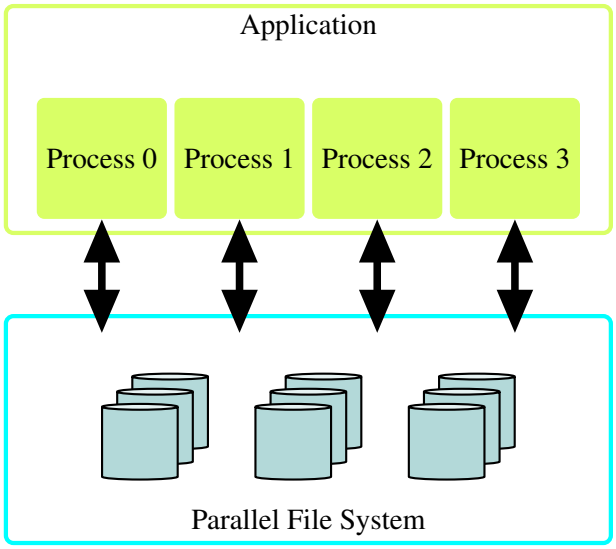


Figure 2.5: A parallel application performing its I/O operations in parallel. Multiple processes of an application issue I/O requests to the I/O subsystem.

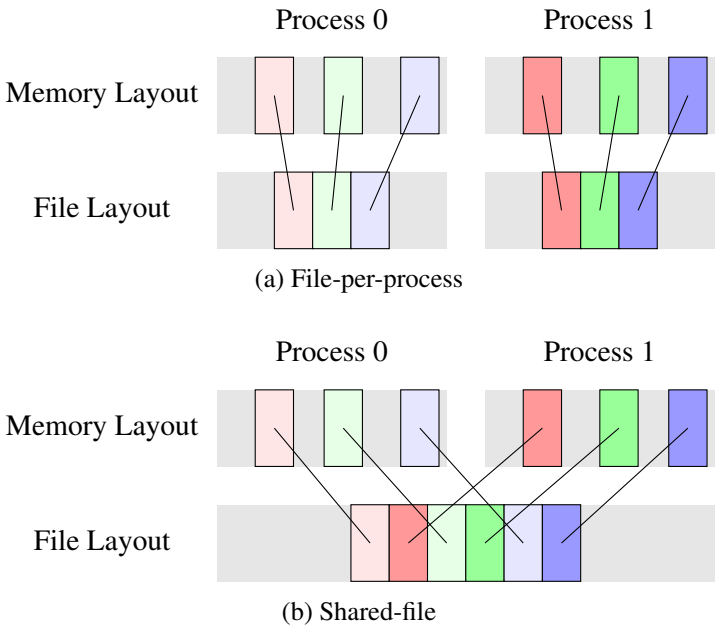


Figure 2.6: This figure illustrates two approaches to manage file accesses in parallel applications: the (a) *file-per-process* and (b) *shared-file* approach. In the file-per-process approach each process of a parallel application operates on its own file. In contrast, in the shared-file approach multiple processes work on a single logical file. Parallel applications often use both approaches during their execution.

2.2 Performance Analysis

Performance analysis is an essential part of the software development process. This analysis builds upon information about the behavior of an application and allows users to identify performance critical aspects of their programs. Therefore, performance analysis of parallel scientific applications includes a *monitor* software component that observes activities of the application during its execution. Figure 2.7 illustrates typical steps of the performance analysis process.

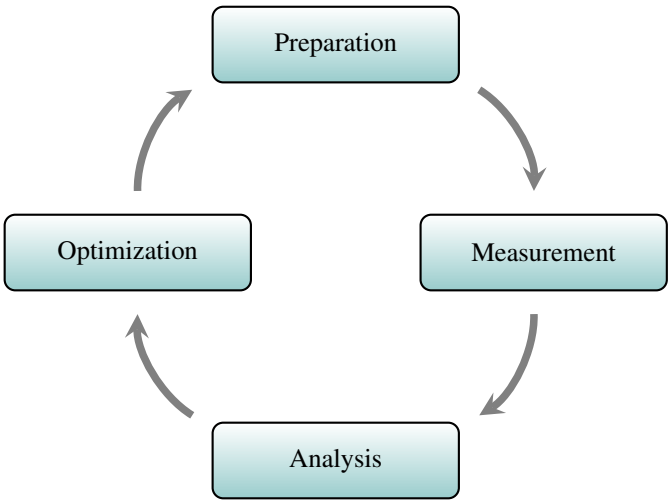


Figure 2.7: An illustration of the performance analysis workflow. First, a monitor prepares the application for observation. Then, the user executes the application and the monitor records application's performance data. Analysis of performance data is the foundation for subsequent application optimization.

Preparation This first step prepares the application for observation by inserting additional instructions to each event of interest. These instructions will invoke the monitor whenever such an event occurs during application execution. Section 2.2.1 details on methods to prepare an application for monitoring.

Measurement This step executes the prepared application. Whenever an event of interest occurs (e.g., function entries and exits), the monitor is triggered. Then, the monitor acquires information about the event, current application status, and collects performance data. Section 2.2.2 presents methods used to record collected performance data.

Analysis The analysis step uses the collected performance data and calculates metrics (e.g., rates derived from recorded counts). Intuitive presentations of the results facilitate identification of performance problems. Section 2.2.3 gives an overview on analysis methods for performance data.

Optimization Based on the knowledge gained from the previous step, developers make decisions about changes to eliminate or at least reduce performance issues. Typically, the optimization step includes source code modifications, e.g., to increase data locality of computationally intensive parts of a program, or changes to the execution setup such as improved process placement to arrange communication partners close to each other. After each optimization, software developers should conduct measurements to validate the effectiveness of applied modifications.

2.2.1 Data Acquisition

This section describes methods used to prepare an application for performance monitoring. There are two major methods to gather information about an application run: *sampling* and *instrumentation*. Section 2.2.1.1 describes sampling in more detail. Section 2.2.1.2 explains the instrumentation-based approach. The following examples describe how a monitor can collect information about function entries and exits. Without limiting the generality of the shown example, the principles also apply to other types of events.

2.2.1.1 Sampling

Performance analysis implies a monitoring infrastructure observing an application during its execution. With sampling, the monitor periodically interrupts the application during its execution. An interrupt is triggered whenever a threshold is reached. Common performance monitors allow users to select interrupts from various sources such as timers or cache misses and specify a threshold value.

Listing 2.1 shows the unmodified version of a source code example. Figure 2.8 illustrates corresponding function calls during the execution of this unmodified application. Figure 2.9 depicts the same sequence of function calls with a performance monitor attached that samples the application execution. In this example, the monitor interrupts the application in equal time intervals and records its current status (e.g., function call stack). Additional workload induced by monitor activities result in a runtime prolongation. As the figure indicates, this runtime overhead depends on the sampling interval and, therefore, is adjustable by the user. However, the figure also illustrates that the sampling method does not record all function invocations. Sampling does not recognize changes to the application status between sampling points. Additionally, sampling cannot provide exact timing information for function entries and exits.

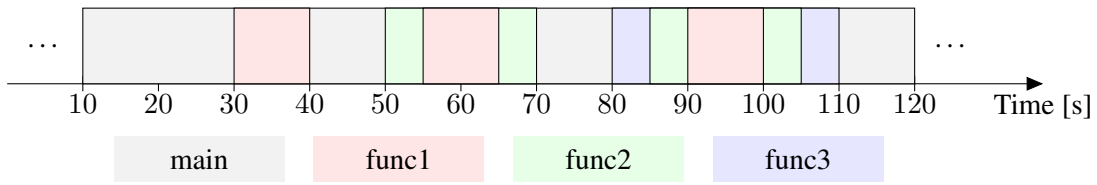


Figure 2.8: This figure depicts function calls during the execution of an unmodified application. The illustrated sequence of function calls corresponds to the pseudo code shown in Listing 2.1.

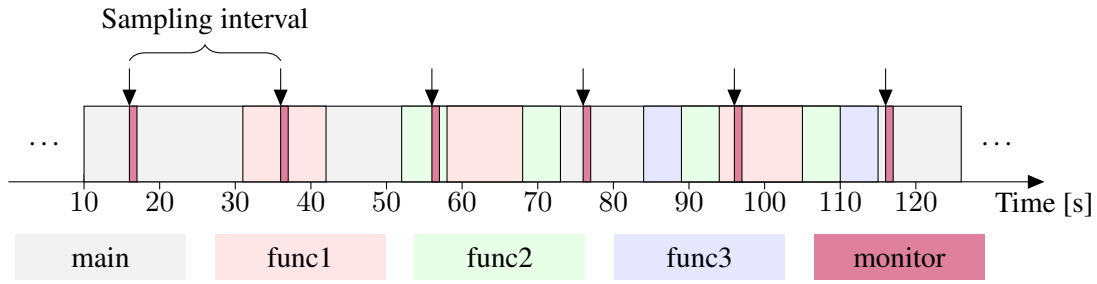


Figure 2.9: This figure depicts function calls and monitor activities executed while sampling the execution of an application. It illustrates the same sequence of function calls as shown in Figure 2.8. However, in this figure a performance monitor is attached to the application and samples its execution.

Listing 2.1: Original source code

```
int main()
{

        func1();
        func2();
        func3();

        return 0;
}

void func1()
{

        ...

}

void func2()
{

        ...
        func1();

}

void func3()
{

        ...
        func2();

}
```

Listing 2.2: Annotated source code

```
int main()
{
        ENTER("main");
        func1();
        func2();
        func3();
        EXIT("main");
        return 0;
}

void func1()
{
        ENTER("func1");
        ...
        EXIT("func1");
}

void func2()
{
        ENTER("func2");
        ...
        func1();
        EXIT("func2");
}

void func3()
{
        ENTER("func3");
        ...
        func2();
        EXIT("func3");
}
```

2.2.1.2 Instrumentation

With this method, the monitor annotates the source code with *hooks* (sometimes also called probes or tracepoints) to record activities such as function entries and exits. These hooks invoke the monitor whenever such an event occurs. This process is called *instrumentation*. Listings 2.1 and 2.2 demonstrate the concept of instrumentation. Listing 2.1 shows the unmodified version of the source code. Listing 2.2 illustrates the annotated version. This example contains calls (hooks) to the monitor after each function entry and before each function exit. The monitor has to implement the corresponding functions `ENTER` and `EXIT`. During application execution, the monitor is invoked whenever the application enters or leaves an annotated function. The control flow passes to the monitor which records relevant information such as timestamp and function name. Afterwards the monitor returns the control flow back to the application and program execution continues. Different strategies exist to insert hooks into source code.

Manual source code modification to insert these hooks is a labor-intensive and error-prone task. Hence, manual instrumentation of real-world applications with thousands of lines of code is not feasible. Furthermore, humans often base their decision whether to instrument a function or not on assumptions instead of profound knowledge. Consequently, a manual instrumentation tends to be biased.

Automatic instrumentation represents another option. Almost all current compilers such as GNU, IBM, Intel, LLVM/Clang, and PGI provide flags to enable automatic code instrumentation during compilation. In addition, some compilers (e.g., LLVM/Clang and GNU) also support a plug-in interface to dynamically extend compiler features. Monitors can utilize this interface and provide their own plug-ins to realize automatic code instrumentation at compile-time [113].

Figure 2.10 depicts the event sequence of Listing 2.2 with function instrumentation and an attached monitor. Similar to sampling, additional instructions induced by function instrumentation cause a runtime prolongation. In contrast to sampling, this runtime overhead depends on the frequency of instrumented events. As the figure shows, the instrumentation method ensures recording of each instrumented event during application execution. With instrumentation, the performance monitor can correlate exact timing information with events because the monitor is immediately called when an event occurs.

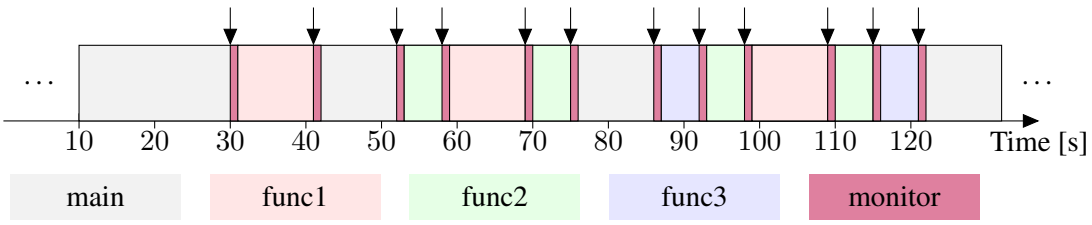


Figure 2.10: This figure depicts function calls and monitor activities executed during the execution of an instrumented application. It illustrates the same sequence of function calls as shown in Figure 2.8. However, the executed application was prepared with instrumentation hooks (see Listing 2.2).

2.2.2 Data Recording

The previous section introduced methods to acquire performance data. This section describes options to store the obtained information. This thesis distinguishes the two approaches *profiling* (Section 2.2.2.1) and *tracing* (Section 2.2.2.2).

2.2.2.1 Profiling

Profiles represent a statistical overview of an application's runtime behavior by summarizing information from individual events. Various kinds of profiles exist, differing in the way data is aggregated [43]. *Flat profiles* summarize data for each function ignoring its context, e.g., the callee of the function. For example, they collect number of invocations of or time spent within a specific function. Listing 2.3 shows a flat profile derived from the example shown in Figure 2.10. In contrast, *callpath profiles* maintain separate statistics with respect to the call stack of a function. Further aggregation is possible for parallel applications, e.g., across threads and processes. On the one hand, data aggregation reduces the storage space required to save profiles. On the other hand, information about individual function calls get lost due to this aggregation. Consequently, performance issues that evolve during runtime are hard to identify with profiles.

Listing 2.3: Example of flat profile data (data derived from Figure 2.10)

Function	Count	Exclusive Time [s]	Inclusive Time [s]
main	1	50	110
func1	3	30	30
func2	2	20	40
func3	1	10	30

2.2.2.2 Tracing

Trace data represents a log of individual events. Consequently, the term *event log* is often used as a synonym for trace data. For each event, the trace records the time when the event occurred, the process/thread where the event was triggered, the event type (e.g., function entry or exit), and meta information such as function names or additional performance metrics. Therefore, event logs retain temporal information about the application behavior. This enables detection of performance problems with changing characteristics over application runtime. Listing 2.4 illustrates an example of event log data. Traces can demand a significant amount of storage space because each event needs to be stored.

Listing 2.4: Example of event log data (data derived from Figure 2.10)

Timestamp	Process	Operation
...		
87	2	ENTER func3
93	2	ENTER func2
99	2	ENTER func1
109	2	EXIT func1
115	2	EXIT func2
121	2	EXIT func3
...		

2.2.3 Data Analysis

An intuitive presentation of collected performance data allows users to gain knowledge about the behavior of their applications, detect performance bottlenecks, and decide about possible improvements. In general, analysis tools can be distinguished into two categories focusing on either *visualization* or *automatic analysis*.

Tools of the first category provide sophisticated visualizations of result data to facilitate their interpretation. Profile data is usually shown as statistical charts such as bar plots or histograms. Based on the profile data shown in Listing 2.3, Figure 2.11 illustrates a bar chart visualization of the exclusive function time.

Timeline charts are commonly used to visualize event log data. This kind of chart depicts application activities (e.g., function entries and exits) with processes/threads on one axis and time on the other axis. Figure 2.12 depicts a timeline visualization of the event log data shown in Listing 2.4.

Tools of the second category perform automatic analysis on result data to detect and mark potential issues. For instance, analysis methods exist to identify I/O intensive phases (I/O bursts) or critical I/O access patterns of an application execution. Based on the result of these automatic analyses some tools also tune parameters (e.g., compiler optimization flags or number of threads) to improve application performance. Thereby, these kind of tools guide users in the process of performance analysis and optimization.

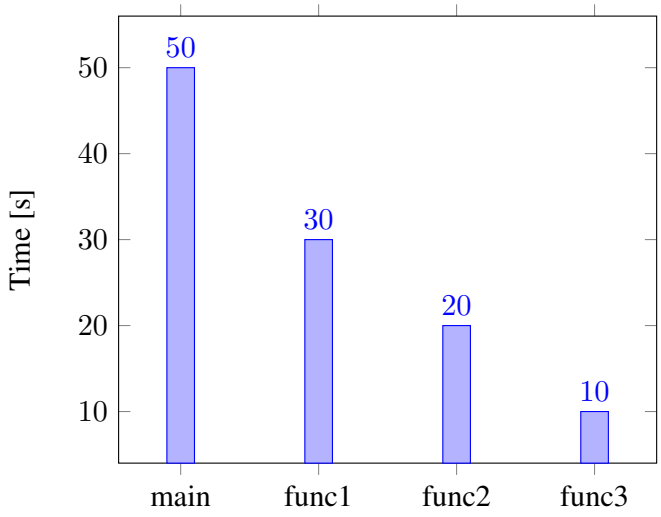


Figure 2.11: This figure illustrates a bar chart visualization of profile data. It depicts exclusive function times as shown in Listing 2.3.

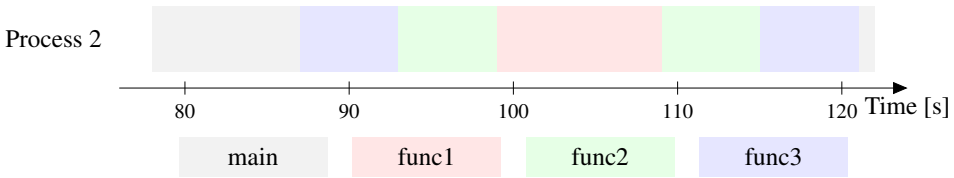


Figure 2.12: This figure illustrates a timeline visualization of event log data. The depicted event sequence is derived from Listing 2.4.

2.3 Performance Analysis Tools for I/O Monitoring

This sections provides an overview of established I/O analysis tools. Table 2.1 provides an overview of monitoring tools and their scope of data acquisition. Table 2.2 focuses especially on tools appropriate for application monitoring. Tools relevant for the subsequent analysis of recorded data are shown in Table 2.3.

OS Monitoring Linux and Unix-like variants are the de-facto standard operating systems (OS) on HPC systems. A wide range of monitoring tools exist for these operating systems. *iotop* [47], *iostat* [46], and *sar* [88] especially focus on monitoring of I/O resource utilization. These tools use a sampling-based approach to obtain data and collect statistics per device, partition, or network file system as well as a global view of the whole system.

In contrast, *blktrace* [12], *Extended Berkeley Packet Filter (eBPF)* [32], *ftrace* [35], *ltrace* [60], *strace* [97], *sysdig* [98], *SystemTap* [99], *Tracefs* [4] use instrumentation and record event logs. blktrace interacts with the OS kernel and generates event logs of the I/O traffic on block devices. Using the eBPF subsystem in the Linux kernel users can attach their own routines to hooks in order to collect information about I/O on block devices, I/O functions of the C standard library, and other operations related to file systems. SystemTap and sysdig follow similar approaches to analyze Linux kernel events. ftrace, ltrace, and strace monitor calls to functions, routines of shared libraries, and system calls. Tracefs works as a layer between the Virtual File System (VFS) and any other file system. Thereby, it records event logs of activities on file systems.

In comparison with the tools mentioned above, *perf* is a flexible framework. It supports sampling as well as instrumentation and records performance data as statistics or event logs. The perf toolset collects information from hardware features (e.g., Performance Monitoring Unit (PMU) of recent processors) as well as software features (software counters, hooks). For example, perf can monitor syscalls invoked by I/O operations.

All these tools focus on monitoring an individual OS instance. Scientific parallel applications typically spawn processes among multiple compute nodes, where each node runs its own OS instance. However, performance analysis of parallel applications has to allow correlations between processes by adding information about communication and synchronization between processes running on different nodes. OS monitoring tools do not include this kind of information and, therefore, lack support for parallel applications. In addition, some of the mentioned OS monitoring tools collect their data on the kernel level. This usually requires root privileges which is prohibitive for common users of HPC systems.

File System Monitoring Tools of this category specialize on observing the behavior of file systems. Established parallel file systems such as IBM Spectrum Scale (formerly known as GPFS) [45] and Lustre provide dedicated monitoring tools. The following paragraph lists corresponding tools for the Lustre file system as an example.

LIOProf [126] instruments I/O activities on the file servers, whereas *Lustre Monitoring Tools (LMT)* [1] use sampling to observe the status of Lustre file system servers. Both tools record statistics (e.g., I/O operations count, bytes read/written, server CPU load) to characterize file system activities.

This kind of tools focus investigations on file systems and servers. Consequently, details about applications are missing in the collected information.

Application Monitoring Table 2.2 provides an overview of tools for monitoring individual applications. Besides events, such as function entries and exits, these tools also record information about I/O activities.

Arm MAP [6] and *HPCToolkit* [3] use sampling to monitor application behavior. In order to acquire exact information such as timestamps of I/O operations, both tools additionally instrument this kind of operations. Arm MAP intercepts calls to POSIX I/O functions as well as system calls and records

Table 2.1: Overview of monitoring levels and corresponding tools

	Data Acquisition		Data Recording		Support for Parallel Applications
	Sampling	Instrumentation	Statistics	Event Logs	
OS Monitoring					
iotop [47], iostat [46], sar [88]	✓	✗	✓	✗	✗
blktrace [12], ftrace [35], ltrace [60], strace [97], eBPF [32], sysdig [98], SystemTap [99], TraceFS [4]	✗	✓	✗	✓	✗
perf [84]	✓	✓	✓	✓	✗
File Server Monitoring					
LIOProf [126]	✗	✓	✓	✗	✗
Lustre Monitoring Tool (LMT) [1]	✓	✗	✓	✗	✗
Application Monitoring					
see Table 2.2					
System Monitoring					
collectl [22], collectd [21], Nagios [73]	✓	✗	✓	✗	✗

✓ Supported
✗ Not supported

Table 2.2: Overview of application monitoring tools

	Data Acquisition		Data Recording		Recorded I/O Activities
	Sampling	Instrumentation	Statistics	Event Logs	
Application Monitoring					
IOPin [50]	✗	✓	✗	✓	PVFS, MPI I/O
LTTng [61]	✗	✓	✗	✓	Subject to available hooks
Arm MAP [6]	✓	✓[1]	✓	✗	System calls, POSIX I/O
HPCToolkit [3]	✓	✓[1]	✓	✓[1]	POSIX I/O[1]
//Trace [69]	✗	✓	✗	✓	POSIX I/O
DUMPI [55]	✗	✓	✗	✓	MPI I/O
mpiP [118]	✗	✓	✓	✗	MPI I/O
IPM [115]	✗	✓	✓	✗	POSIX I/O, MPI I/O
Extrae [14], PIOM-PX, [41] RIOT [124], ScalaTrace [75, 120, 125], VampirTrace [52]	✗	✓	✗	✓	POSIX I/O, MPI I/O
TAU [92]	✗	✓	✓	✓	POSIX I/O, MPI I/O
Recorder [65, 10]	✗	✓	✗	✓	POSIX I/O, MPI I/O, HDF5
Darshan [18, 17]	✗	✓	✓	✓[1]	POSIX I/O, MPI I/O, HDF5[1], PNetCDF[1]
Score-P [53]	✓	✓	✓	✓	POSIX I/O[2], MPI I/O[2], HDF5[2], (P)NetCDF[2]

✓ Supported
✓,[1] Partially supported
✗ Not supported
[2] Implemented in this thesis

Lustre [64] counters. HPCToolkit wraps only a set of specific POSIX I/O functions (`read`, `write`, `fread` and `fwrite`) [67]. Both tools present statistics of their collected performance data to users.

All other tools shown in Table 2.2 use instrumentation to acquire performance data of parallel applications.

IOPin [50] captures information about MPI I/O operations of the observed application. In addition, IOPin instruments the Parallel Virtual File System (PVFS) [19] client and server. The combination of MPI I/O and PVFS logs allows users to investigate how I/O operations of their applications are mapped onto the Parallel Virtual File System. This work does not consider high-level I/O libraries such as HDF5 and PnetCDF. The *Linux Trace Toolkit: next generation (LTTng)* monitors applications and libraries in user space. Additionally, LTTng supports Linux kernel tracing which requires root privileges. The LTTng framework includes pre-built hooks to instrument function entries and exits, memory and POSIX threads functions of the C standard library, and activities of the dynamic linker. Users can extend LTTng by implementing their own tracepoints, e.g., to mark I/O operations of the application.

Many tools record event logs for a later replay. A trace replay allows users to investigate the behavior of their applications under varying conditions, such as increased bandwidth or reduced latency to the I/O subsystem. *//Trace* [69] intercepts calls to POSIX I/O functions and identifies data dependencies. In order to detect these dependencies, //Trace executes the observed application multiple times with varying delays added to the I/O operations. The *DUMPI* trace library records event logs of MPI parallel applications. Traces recorded by DUMPI contain information about MPI I/O events. DUMPI does not track I/O on any other level of the I/O software stack such as POSIX I/O. *ScalaTrace* [75], *PIOM-PX* [41], and *RIOT I/OToolkit (RIOT)* [124] record POSIX I/O and MPI I/O activities. ScalaTrace captures event logs of the MPI communication from parallel applications. In order to reduce the data size of the event log, ScalaTrace exploits the redundancy of repetitive event sequences. In scientific applications these repetitive event sequences originate from iterative or recursive algorithms or Single Program Multiple Data (SPMD) parallelization. ScalaTrace can compress repetitive event sequences within one MPI process (e.g., multiple iterations of a loop) as well as among processes (e.g., homogeneous behavior of multiple MPI processes). ScalaTrace records delta times between events instead of absolute timestamps. Histogram bins store statistical timing information. Otherwise, minimal differences in delta times could impede data compression. Resulting concise event logs preserve the application's communication structure and build the input of a trace replay tool. Vijayakumar et al. [120] extend ScalaTrace's functionality by tracing calls to MPI I/O and POSIX I/O. They employ the same data compression techniques. Wu et al. [125] implement a more aggressive compression with a user-tunable precision level. The authors apply the histogram-based compression also to iteration counts and function parameters. The choice of the precision level represents a trade-off between accuracy and gain in compression ratio. PIOM-PX also exploits the repetitive behavior of most parallel applications. The tool tracks parameters of I/O operations, and generates a model of the application's I/O behavior based on global spatial and temporal file access patterns. This model allows users to identify I/O intensive phases of the application execution, to investigate the effect of varying configurations on I/O phases and the I/O subsystem, and to reproduce the I/O behavior of an application on a different HPC platform. RIOT captures I/O activities of parallel applications. In contrast to the latter tools, it focus its analysis on presenting statistics and visualizations of the recorded performance data. RIOT tracks MPI I/O calls from the application. Additionally, the tool intercepts POSIX I/O function calls issued by the MPI I/O library. This allows users to investigate relations between MPI I/O and subsequent POSIX I/O operations. However, the recorded event logs do not explicitly correlate these I/O operations. Users can only identify relations by examining the timing information of operations stored in the logs.

Recorder [65] captures calls to POSIX I/O, MPI I/O, and HDF5. The tool does not correlate events across layers of the I/O software stack. According to [65] there is a lack of trace analysis tools for Recorder. The authors plan to implement a trace replay engine. Behzad et al. [10] use Recorder to record HDF5 operations of an application and generate I/O kernels reflecting its I/O behavior.

The *Darshan* tool focuses on characterization of applications' I/O behavior. It enables monitoring of large scale HPC applications with less overhead and reasonable memory demands. To keep performance

data of the observed application in a memory buffer of a fixed size, Darshan collects compact statistics instead of detailed information about each individual I/O operation. Darshan maintains the statistics for each opened file in a separate record. By default Darshan tracks up to 1024 files, however, users can adjust this limit. If the limit is exceeded, Darshan aggregates the statistics into a single record. The Darshan tool comes with software modules supporting the instrumentation of MPI I/O, POSIX I/O, and standard I/O library function calls. For these I/O libraries, Darshan collects information, for example, about access patterns within files, access sizes, time spent within I/O operations, or the number of bytes read and written. Additional software modules provide basic information about HDF5 and PnetCDF library function calls, including the number of open operations (for PnetCDF differentiated between collective and independent operations) as well as the timestamps of the first and the last open/close operation. Further software modules collect Lustre file system information or trace calls to MPI I/O and POSIX I/O library functions (Darshan eXtended Tracing (DXT)).

In contrast to the tools mentioned above, there are monitoring infrastructures focusing on traditional aspects of performance analysis such as computation, communication, and synchronization. In addition, these monitors also record information about I/O operations. *mpiP* [118] assists users in profiling their MPI applications. The mpiP tool intercepts MPI library calls via the PMPI interface and captures statistics per process. Although mpiP does not focus on I/O analysis, the tool collects information about MPI I/O routines including the number of invocations as well as the maximum, mean, and minimum number of bytes transferred. mpiP does not capture any information about POSIX I/O. The *Integrated Performance Monitoring (IPM)* [115] infrastructure provides a similar feature set. In contrast to mpiP, IPM also records POSIX I/O operations of an application. The *Tuning and Analysis Utilities* (TAU) [92] are a versatile toolkit for performance analysis of parallel applications. The TAU framework instruments applications and records profiles and traces. TAU captures POSIX I/O and MPI I/O operations of the monitored program. *Extrae* [14] and *VampirTrace* [52] instrument parallel applications and capture their POSIX I/O as well as MPI I/O activities. VampirTrace is no longer supported. *Score-P* [53], the successor of VampirTrace and the common measurement infrastructure of analysis tools such as Scalasca and Vampir, utilizes sampling and instrumentation to acquire performance data of parallel applications. However, prior to this thesis Score-P did not support I/O monitoring. This work implements its proposed methodology in Score-P and thereby extends this measurement infrastructure by sophisticated capabilities for I/O activity recording.

There are many more vendor specific tools, e.g., *Intel VTune Profiler* [26], *Intel Trace Analyzer and Collector* [25], *CrayPat* [28], and *IBM HPCT-IO* [23]. Because these tools share their approach with one of the tools described in this paragraph, they are not covered separately.

System Monitoring Tools such as *collectl* [22], *collectd* [21] or *Nagios* [73] regularly check system characteristics and collect statistics describing the current status. These tools report the system health and alert administrators in case of service degradation. Due to their scope these tools are less suitable for application performance analysis.

Analysis of Performance Data Table 2.3 lists established tools and their field of analysis. This paragraph starts with the presentation of analysis tools focusing on intuitive visualizations of collected performance data. *Cube* [37] and *ParaProf* [92] use charts to visualize profile data. Both tools are designed for general performance analysis, presenting information such as number of function invocations or time spent in individual functions. Figure 2.13 shows a visualization of profile data in Cube. Figure 2.14 to Figure 2.16 give an impression of ParaProf's visualization features. In contrast, *Darshan* [18, 17] focuses on I/O characterization of applications. Therefore, Darshan shows specialized graphs, e.g., bar charts for I/O operation counts split by the operation type. Figure 2.17 exemplifies a Darshan report.

Jumpshot [56], *Paraver* [15], and *Vampir* [52] visualize event logs as timeline charts. As shown in Figure 2.18 this kind of chart illustrates the dynamic behavior of observed applications. Additionally, Vampir provides charts that highlight utilization of I/O resources.

Table 2.3: Overview of performance analysis tools

	Data Analysis			
	Visualization		Automatic Analysis	
	Statistics	Event Logs	Wait State Detection	Replay
Cube [37], ParaProf [92]	✓	✗	✗	✗
Jumpshot [56], Paraver [15], Vampir [52]	✗	✓	✗	✗
Scalasca [37]	✗	✗	✓	✗
DUMPI [55], //Trace [69], ScalaTrace [75, 120, 125]	✗	✓	✗	✓

Another set of tools apply automatic analyses on recorded performance data. *Scalasca* [37] processes event logs of MPI/OpenMP parallel applications, analyzes their communication and synchronization, and detects patterns that represent potential performance bottlenecks. Figure 2.19 illustrates the event sequence of typical inefficiency patterns observable in MPI parallel applications. For instance, Figure 2.19b depicts the *Late Sender* pattern. One process sends a message to another process. In this example, both processes use blocking operations to communicate (e.g., `MPI_Send`/`MPI_Recv`). The receiver has to wait until the sender actually starts the message transfer. The Scalasca analysis reveals such wait states based on event logs. The concept of detecting critical patterns in event logs is applicable to other characteristics of an application. This work defines inefficiency patterns for I/O operations of parallel applications in Section 3.3.1.

DUMPI [55], //Trace [69], and ScalaTrace [75, 120, 125] assist users in replaying the event logs with varying parameters and examining the effects.

Research Projects Many more research projects address I/O analysis in the field of HPC such as LANL-Trace and SIOX [54]. These projects will not be covered separately in this section, as they follow the same principles as one of the presented tools or lack robustness and portability.

Similar to this work, the "Total Knowledge of I/O" (TOKIO) [58] and "Unified Monitoring and Metrics Interface" (UMAMI) [59] projects emphasize the need for a holistic approach to characterize complex HPC I/O subsystems. TOKIO provides a framework that collects data on various levels of the HPC I/O hardware stack. Therefore, the framework manages data of different formats, resolutions, and scopes. UMAMI presents an approach to integrate metrics from different I/O components in order to enhance understanding of I/O performance variations. This thesis complements the work of TOKIO and UMAMI by focusing on the HPC I/O software stack, especially on the analysis of scientific parallel applications.

The Virtual Institute for I/O (VI4IO) [119] provides a collaboration platform for research groups in the field of HPC I/O. This organization informs about I/O middleware, benchmarks, and tools. In addition, VI4IO maintains the IO-500 list that ranks HPC systems based on their storage systems performance.

Summary Event logs enable detailed analysis of applications' I/O behavior including temporal information. I/O analysis in general needs to incorporate further aspects of parallel programming (computation, communication, synchronization). Existing tools focus on either traditional performance analysis providing only limited support for I/O analysis or vice versa. None of the presented tools provides a holistic view on the application. In addition, none of the tools is able to record hierarchical relations between individual layers of the HPC I/O software stack.

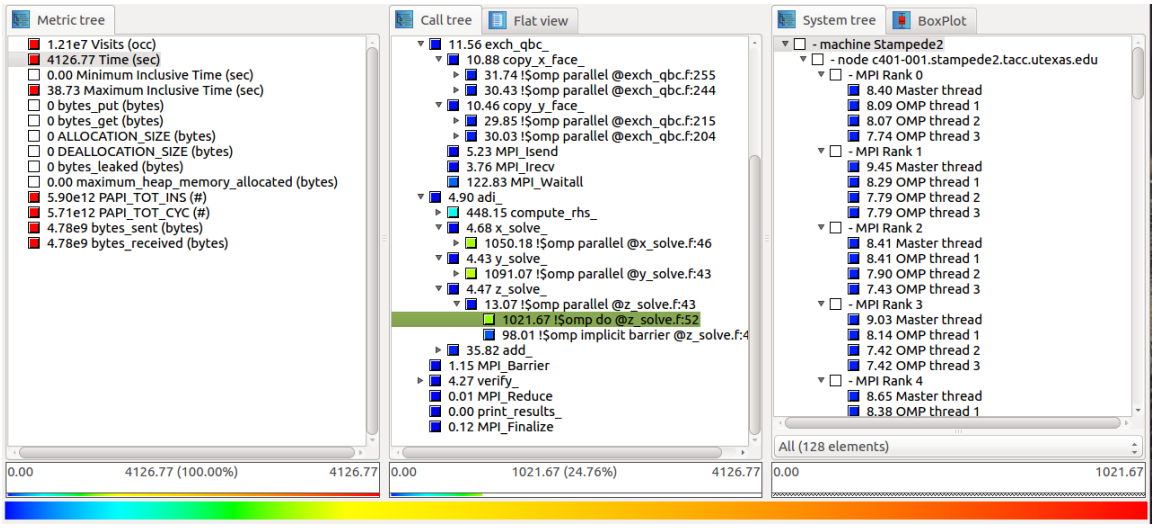


Figure 2.13: The Cube GUI provides users with an interactive visualization of performance profile data. The left pane shows recorded metrics. In this example, execution time is selected. The middle pane depicts functions with their current metric values arranged in a call tree. Color coded boxes guide users to the hotspots with respect to the currently selected metric. In this figure, the user choose an OpenMP parallelized loop for further investigations. The right pane maps metric values of the currently selected function to the system topology such as compute nodes, MPI processes, and OpenMP threads.

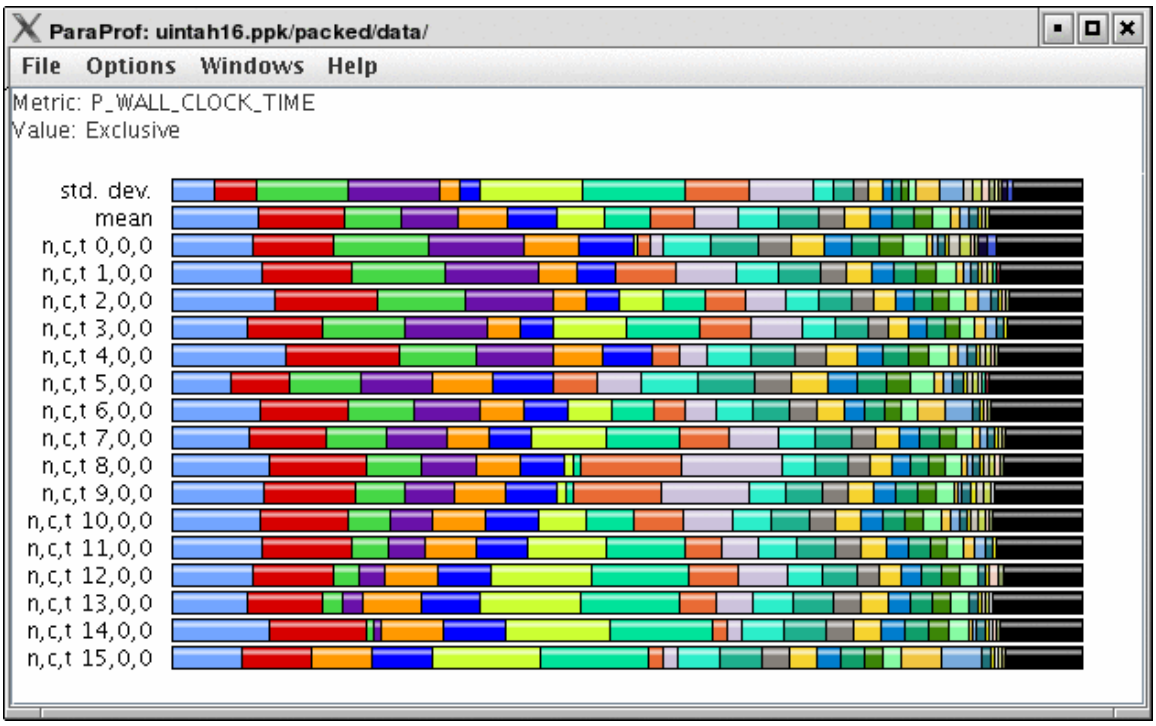


Figure 2.14: This figure depicts a visualization of profile data as a (stacked) bar graph in ParaProf. Bars represent processes and colors illustrate individual functions. The length of the colored boxes correspond to the exclusive execution time of individual functions. In this display mode ParaProf combines all functions executed by a process in one bar. (Taken from [82].)

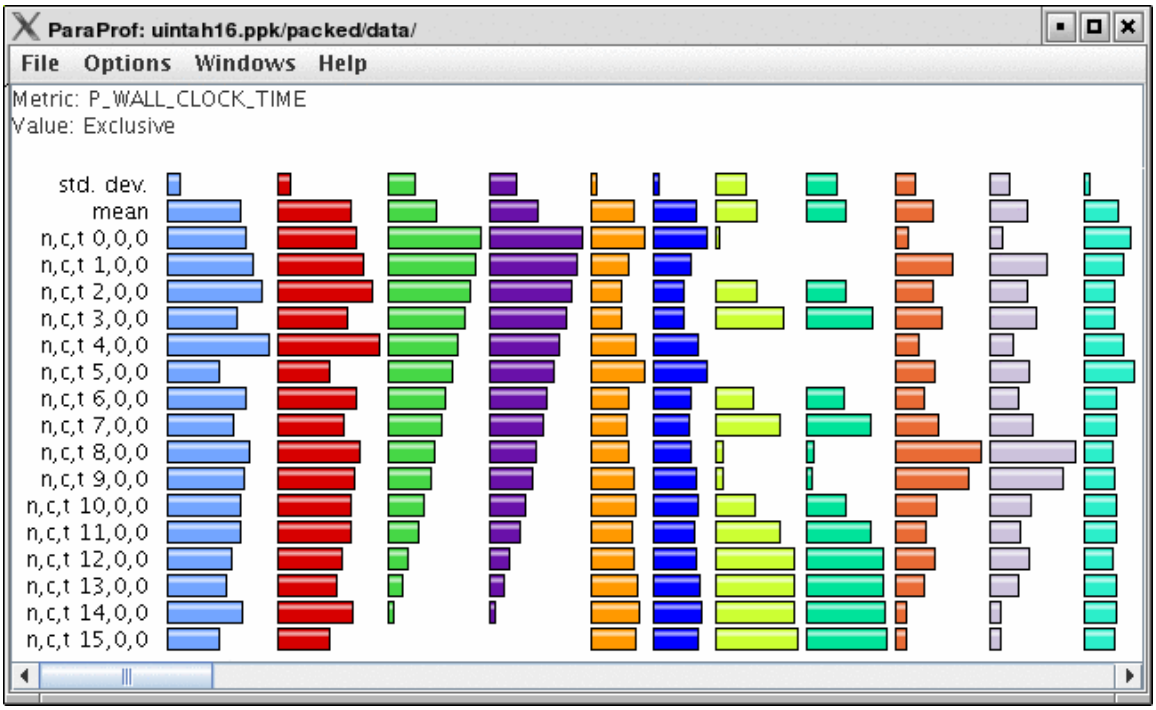


Figure 2.15: This figure depicts a visualization of profile data as a (unstacked) bar graph in ParaProf. In contrast to Figure 2.14 each function is depicted separately. This display mode facilitates comparison of individual functions across processes. (Taken from [82].)

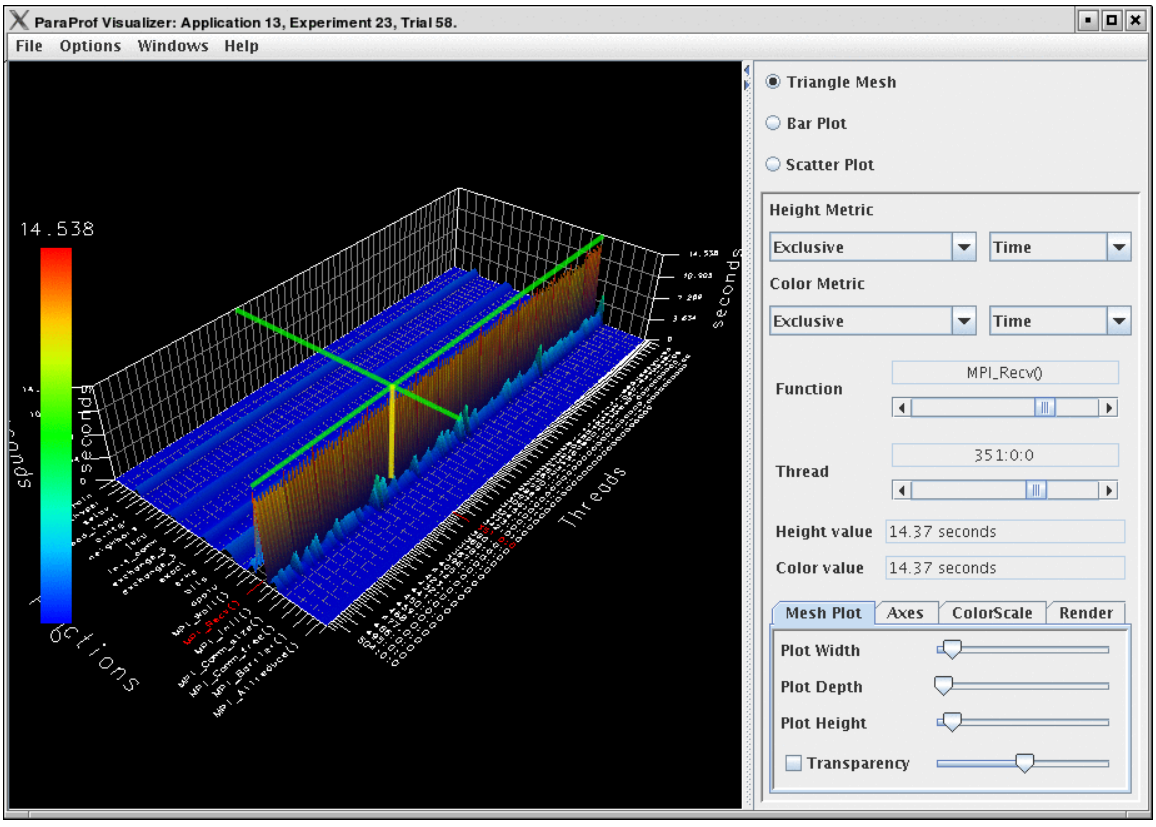
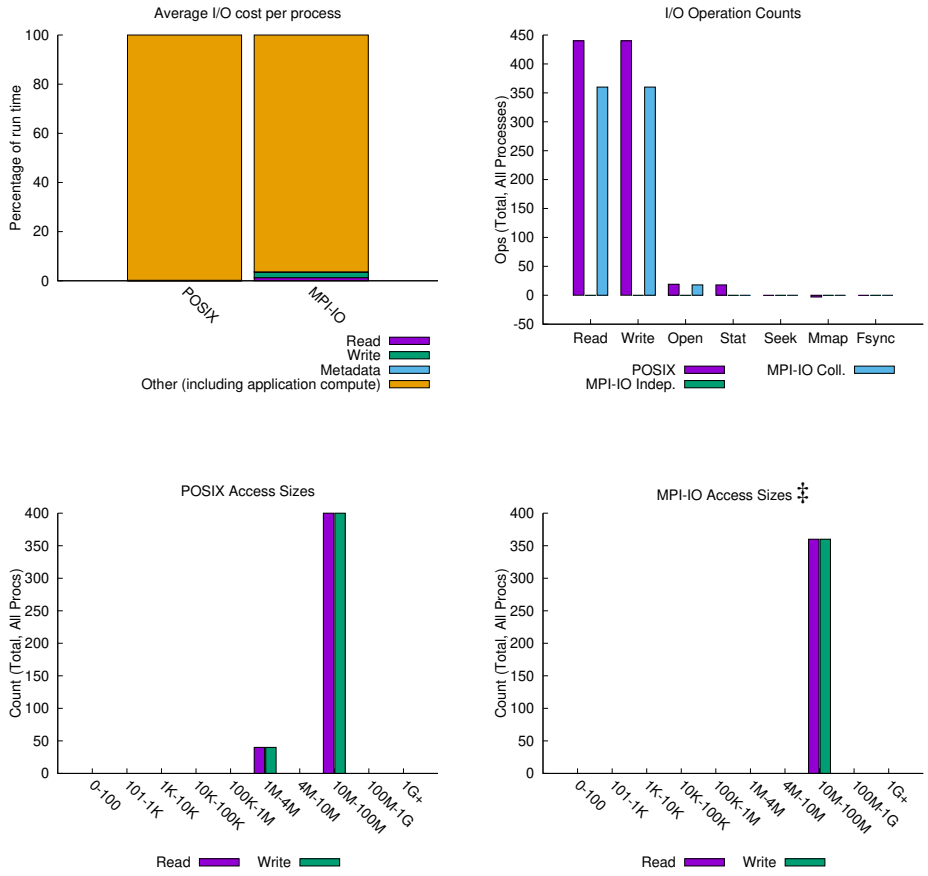


Figure 2.16: ParaProf also supports three-dimensional visualization of profile data. This example depicts the exclusive execution time of different functions across all threads. (Taken from [82].)



Most Common Access Sizes
(POSIX or MPI-IO)

	access size	count
POSIX	16777216	800
	2288960	80
MPI-IO ‡	18895680	720

‡ NOTE: MPI-IO accesses are given in terms of aggregate datatype size.

File Count Summary
(estimated by POSIX I/O access offsets)

type	number of files	avg. size	max size
total opened	1	6.4G	6.4G
read-only files	0	0	0
write-only files	0	0	0
read/write files	1	6.4G	6.4G
created files	1	6.4G	6.4G

Figure 2.17: In this example Darshan recorded statistics about POSIX I/O and MPI I/O activities of an application. Aggregated metrics inform users about the number of specific I/O operations (top right) and the time spent in these routines (top left). Additional charts categorize read-/write operations by their I/O paradigm and access sizes (middle). Finally, summaries report common access sizes and list accessed files.

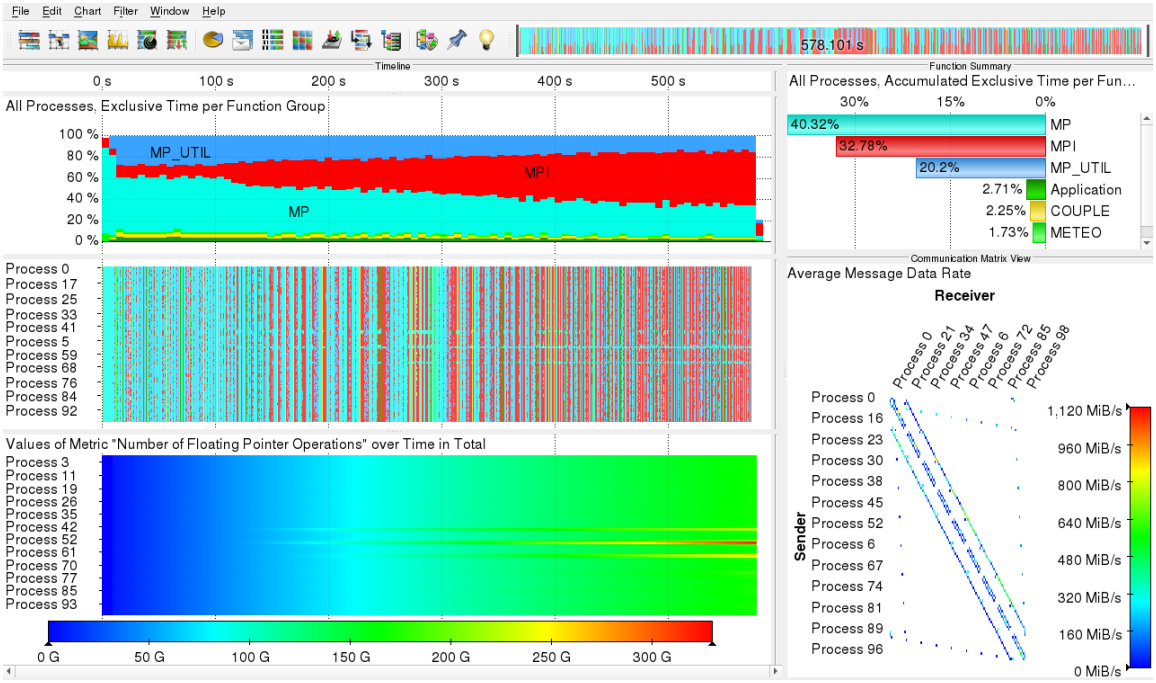


Figure 2.18: This figure illustrates the visualization of event logs in Vampir. The display in the top right corner presents an overview of the entire application run. The statistics display (top right) lists the exclusive time per function group. Different colors indicate individual function groups. The red color represents MPI functions. According to the statistics display, the application spent a lot of time in MPI routines. However, temporal information contained in trace files and timeline based visualizations reveal that the performance issue of this application evolves during its execution. The top left display illustrates the share of individual function groups (x-axis) during execution time (y-axis). The middle left display shows processes (x-axis) and execution time (y-axis). Colors indicate the group of the currently executed function. Both timeline charts depict an increase of the time spent in MPI functions over the application runtime. The metric display (bottom left) shows processes on the x-axis and time on the y-axis. The current number of floating point operations is color-coded from low (blue) to high (red) values. The display highlights processes 44, 45, 54, 55, 64, and 65, that show an increased computational load. Due to this imbalance affected processes enter MPI communication late which causes MPI wait time on other participants. In addition, the right bottom display shows the average message data rate in a communication matrix of all MPI processes.

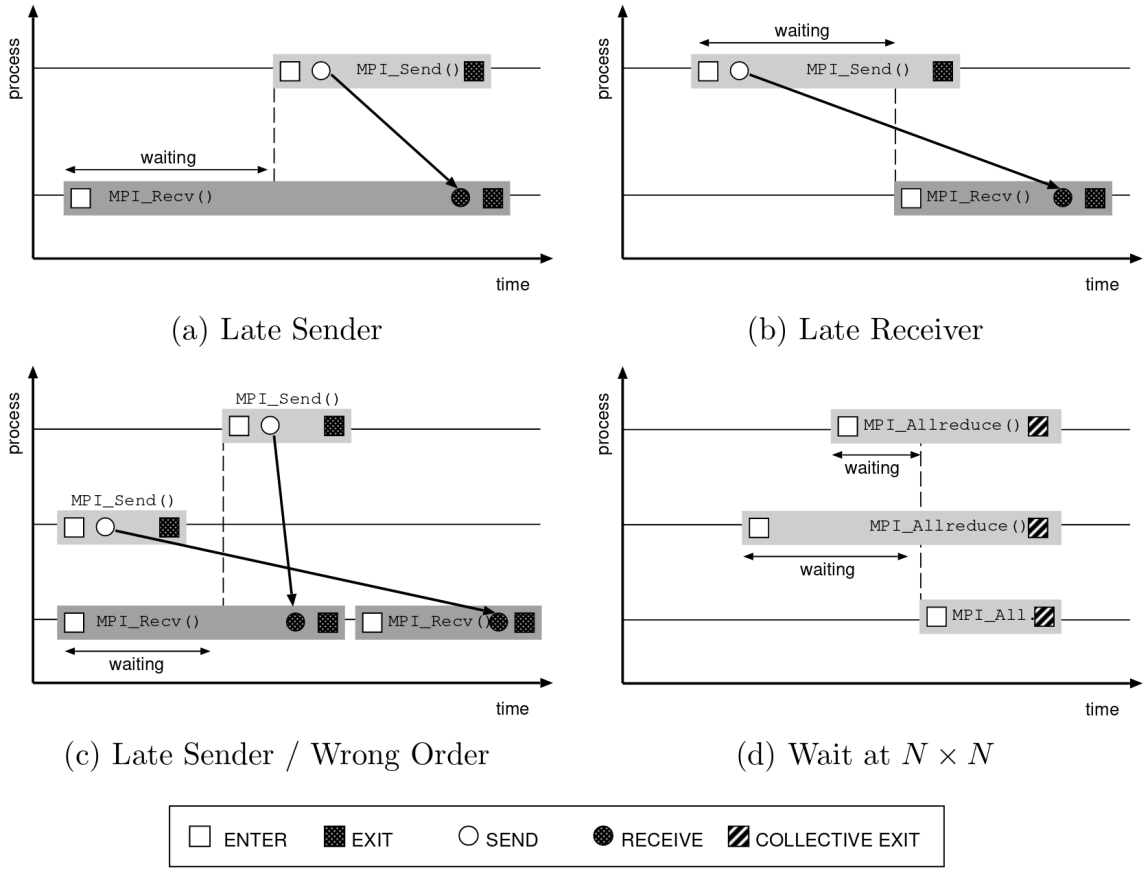


Figure 2.19: The figure illustrates typical inefficiency patterns in MPI communication: (a) *Late Sender*, (b) *Late Receiver*, (c) *Late Sender/Wrong Order*, and (d) *Wait at* $N \times N$. For instance, the *Late Sender* pattern describes an inefficiency in the communication between two processes. The receiver has to wait until the sender actually starts the message transfer. In contrast, in the *Late Receiver* pattern the sender is blocked and waits until the receiver becomes ready to receive. The *Late Sender/Wrong Order* pattern illustrates a wait state where messages are sent in a different order than the receiver expect them to receive. As the *Wait at* $N \times N$ pattern shows, wait states can also occur in collective operations. One process enters late in the collective operation and thereby causes waiting time on the other participants. (Taken from [36])

3 Methodology for a Holistic Performance Analysis of Multi-layer I/O in Parallel Scientific Applications

This chapter introduces a methodology for holistic performance analysis of multi-layer I/O in parallel scientific applications. It presents options to acquire performance relevant information about I/O operations as well as the design of data structures to store this information as event logs and model relations between observed operations. Furthermore, this chapter describes sophisticated analysis techniques based on the collected data.

The proposed methodology utilizes information about activities of an application collected during its execution. The particular focus lies on information about I/O operations of applications. Therefore, Section 3.1 explains methods used to monitor a parallel application during its execution and obtain data about its I/O operations.

After obtaining data, information needs to be stored to make it available for later analysis. Section 3.2 describes concepts developed in this thesis to store information about I/O resources and operations.

Performance data obtained and stored with the shown methodology is the foundation for enhanced analysis techniques. Section 3.3 starts with the analysis of individual applications. For example, this section specifies I/O inefficiency patterns detectable with the methodology presented in this work. Then, the scope of analysis is widen from individual applications to entire scientific workflows.

3.1 Data Acquisition

This section covers aspects of acquiring data relevant for performance analysis of parallel applications during their execution. In this work, calls to I/O libraries are a matter of particular interest. Holistic performance analysis of multi-layer I/O operations in parallel scientific applications requires accurate information, especially timing data, to detect patterns in or calculate transfer rates for these operations. Therefore, this work utilizes instrumentation (see Section 2.2.1.2) to gather information about calls from the application to I/O libraries. This section presents different methods to intercept calls to routines of a library.

Intercepting Calls to Library Functions A performance monitor can employ different techniques to obtain information about an application's behavior while it is executed. This paragraph provides an overview of options how a monitoring tool can intercept calls from an application to library functions. The process of intercepting function calls is also called *wrapping*. Although the focus of this work is on calls to I/O libraries, the presented approaches can be used for any kind of library functions.

Figure 3.1 shows the process of compiling and linking an application. An application developer writes the program logic in one or more source code files. The compiler takes source code files as input and generates an object file for each individual source code file. These object files represent a transformation of source code statements into corresponding machine instructions. Within object files, variables and functions are referenced as symbols. Modern software design splits complex functionality in separate modules and functions. Furthermore, code can be bundled as a library to make its functionality usable for other software components. For example, application developers include libraries providing implementations of mathematical routines, basic I/O operations or inter-process communication. As a consequence, an object file can also include references to symbols of other object files or libraries. The linker resolves these symbols, i.e., it maps symbols to memory addresses. There are two types of linking: *static* and *dynamic linking*. With static linking the linker copies necessary code to the executable and

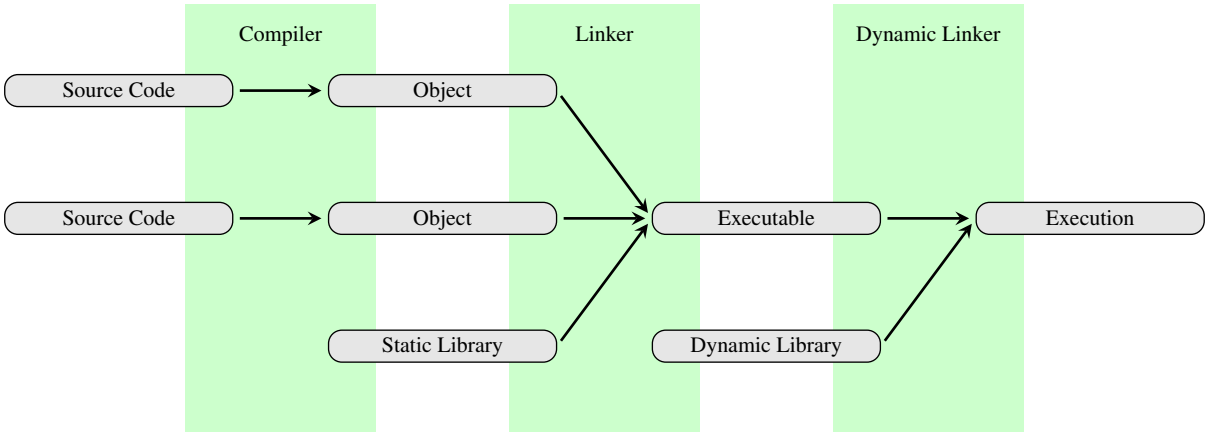


Figure 3.1: Overview of the process of compiling and linking an application. The compiler transforms source code files into object files. The linker combines these object files and additional static libraries into an executable binary. Consequently, the linked static libraries form part of the binary. Nevertheless, the resulting binary can still exhibit unresolved references to external symbols. The dynamic linker resolves these symbols at binary startup using dynamic libraries. Dynamic libraries are separate components. They are also known as shared objects or shared libraries because dynamic libraries can be used by multiple binaries individually.

resolves symbols accordingly at *link-time*. In addition, modern operating systems also support dynamic linking which defers the resolution of undefined symbols until application execution. At *execution-time* the dynamic linker inspects the executable, determines required shared libraries, loads them, and resolves undefined symbols. According to the different approaches presented above, this work distinguishes between interception of library calls at link-time (Section 3.1.1), at execution-time (Section 3.1.2), and via specific tool interfaces provided by the libraries (Section 3.1.3).

3.1.1 Intercepting Calls to Library Functions at Link-Time

This approach relies on linker capabilities to realize a name-shifting of function symbols. For example, the GNU linker provides the `--wrap` option for this purpose. A name-shifting for the function `read` takes effect, if `--wrap read` is specified in the link command. With this option set, the GNU linker resolves undefined references to `read` by shifting the name to `__wrap_read` while linking an application. A monitor has to provide an implementation of the function `__wrap_read`. Within the `__wrap_read` function the monitor can collect performance relevant information and call the original function via `__real_read`. The linker takes care of resolving `__real_read` to `read`. Figure 3.2 illustrates the process of intercepting a call to a library using capabilities of the linker. This approach utilizes the linker (vertical bar in the middle of Figure 3.1) and requires modifications to the link command. Consequently, the approach does not work for wrapping function symbols that are resolved by the dynamic linker at execution-time (vertical bar at the right hand side of Figure 3.1), e.g., functions called from a dynamic library.

3.1.2 Intercepting Calls to Library Functions at Execution-Time

This approach relies on capabilities of the dynamic linker to modify the order of linked libraries. In order to wrap the function `read`, a monitor provides its own implementation of the function `read` with the same signature in a shared library `myLibrary`. While resolving references to `read`, the dynamic linker has to consider the monitor implementation of `read` before the original one. Therefore, it is necessary to specify the path of the shared library `myLibrary` in the environment variable `LD_PRELOAD`. Libraries mentioned in `LD_PRELOAD` take precedence over other libraries in standard library paths. This ensures that the dynamic linker redirects calls to `read` from the application to the own implementation. As

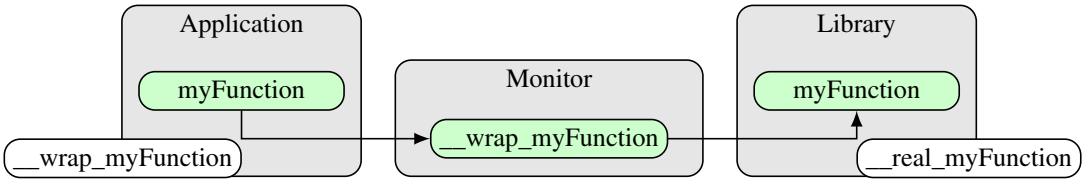


Figure 3.2: The concept of intercepting calls to library functions at link-time requires appropriate features of the linker. The linker applies a name-shifting while resolving function symbols. This mechanism allows third-party tools such as performance monitors to provide their own wrappers for selected functions.

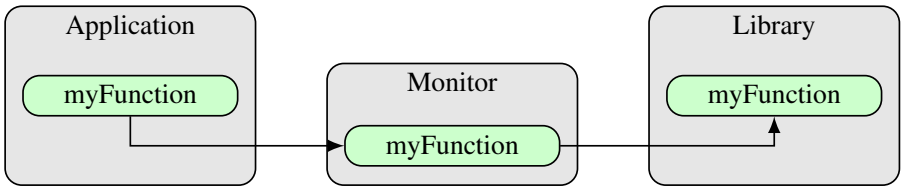


Figure 3.3: The concept of intercepting calls to library functions at execution-time is based on modifying the order of linked libraries. The dynamic linker redirects a function call if the corresponding wrapper function, e.g., provided by a performance monitor, takes precedence over other libraries.

a result, calls to `read` from the application invoke the monitor that can collect performance relevant information. In order to call the original function, the monitor uses `dlsym` to search the address of the original function symbol `read` and calls the respective implementation. Figure 3.3 depicts the process of intercepting a call to a library using capabilities of the dynamic linker. Consequently, this approach is not able to intercept function calls that were already resolved by static linkage.

3.1.3 Intercepting Calls to Library Functions via Tool Interface

Some libraries directly provide support for performance analysis tools. This section describes two established methods to implement such tool interfaces in libraries: callback-based and via weak symbols.

Callback-based Interception There are several APIs supporting callback-based interfaces for tools, e.g., OpenACC [78] (OpenACC Profiling Interface [79, Chapter 5]) or OpenMP [80] (OMPT [81, Chapter 4]). Without loss of generality the following paragraph explains the mechanism of callback-based tool interfaces using the example of OMPT.

Figure 3.4 shows the event sequence between an OpenMP parallel application, the OpenMP runtime, and a performance monitor attaching to the OpenMP library via a callback-based interface. The OpenMP runtime and the performance monitor coordinate via a defined API. This interface contains an entry point. In the example of OMPT the function `ompt_start_tool` represents this entry point. In order to make use of OMPT the performance monitor provides an implementation of this function. On the one hand, this function can be statically linked into the application. On the other hand, a separate shared library can contain this function. In this case the path to the corresponding shared library needs to be specified in the environment variable `LD_PRELOAD` or `OMP_TOOL_LIBRARIES`. During its initialization the OpenMP runtime checks for an implementation of `ompt_start_tool`. If an implementation of this function is found, it is called. Thereby, the control flow passes to the monitor. The monitor creates a `ompt_start_tool_result_t` data structure, specifies the function pointer `initialize` and `finalize` of this structure, and returns the data structure to the OpenMP runtime. After completing its own initialization the OpenMP runtime calls the `initialize` function of the monitor. The `initialize` function has a `lookup` argument. It provides a pointer to a `lookup` function. The monitor uses this `lookup` function in order to determine pointers to OMPT interface runtime entry points,

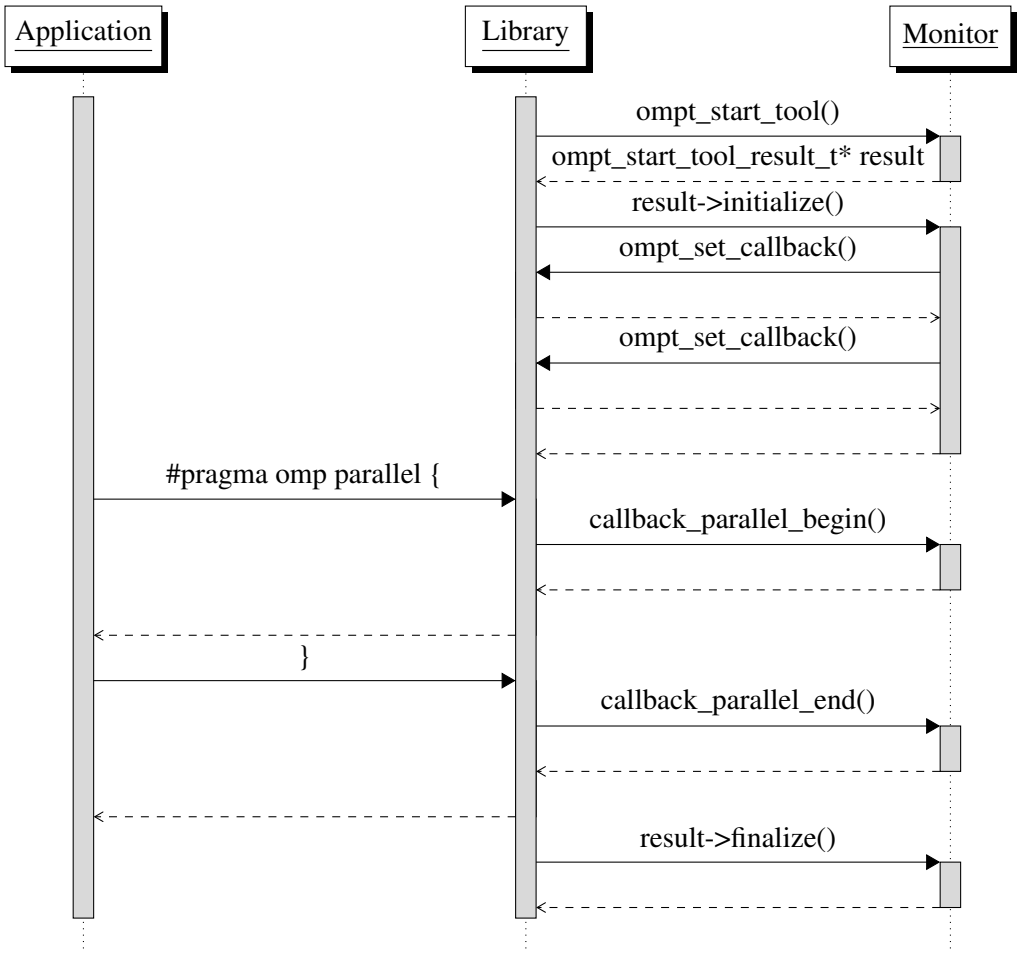
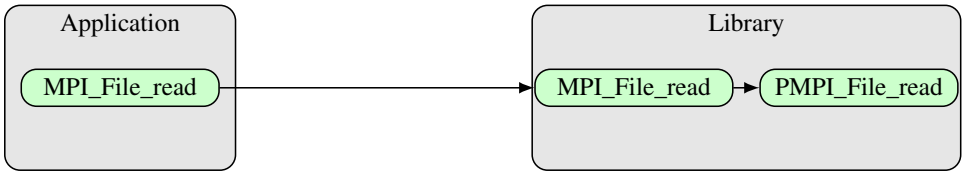
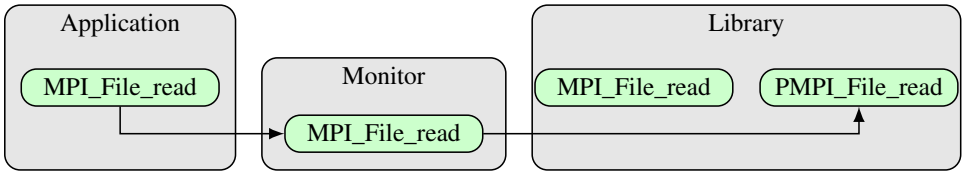


Figure 3.4: The concept of a callback-based interception of library function calls. A monitor registers its callback functions. The observed library invokes the appropriate callback function whenever a corresponding event occurs during application execution.



(a) Without monitor attached



(b) With monitor attached

Figure 3.5: The concept of intercepting calls to library functions via weak symbols. The monitor provides its own implementation of an observed function with its weak symbol name (`MPI_File_read`). In addition, the monitor has to ensure that this version is linked prior to the original one. The original function is accessible by its strong symbol name (`PMPI_File_read`).

e.g., a pointer to the `ompt_set_callback` function. Within the `initialize` function, the monitor registers all callbacks of interest using `ompt_set_callback`. Afterwards, the monitor returns to the OpenMP runtime and application execution continues. During execution the application might trigger the OpenMP runtime. Then, the OpenMP runtime checks whether a monitor has registered a callback for this specific event. If so, the OpenMP runtime invokes the callback and thereby triggers the monitor. The monitor collects performance relevant information and returns the control flow to the OpenMP runtime. When the application ends its execution, the OpenMP runtime stops. In this case, the OpenMP runtime informs the monitor about this event using the `finalize` function pointer. The monitor registered this function in the initialization phase. This function invocation allows the monitor to shut down gracefully, e.g., the monitor can free its allocated resources.

Interception via Weak Symbols Another option to implement a name-shifting of symbols is the use of strong and weak symbols. This concept requires support by the compiler and linker. For example, MPI and OpenSHMEM provide a profiling tool interface based on weak symbols. Figure 3.5 depicts the concept of using weak symbols to provide an alternative function implementation. Without loss of generality the following paragraph explains the mechanism of tool interfaces based on weak symbols using the example of MPI.

The MPI standard defines the MPI profiling interface [72, Chapter 14.2]. It allows performance analysis and debugger tools to intercept calls into the MPI library by providing their own implementation of MPI functions. According to the standard specification a compliant MPI implementation has to provide an alternative entry point (PMPI name prefix) for each MPI function (MPI name prefix). This name-shifting can be realized using weak symbols. For example, the MPI library implements the function `PMPI_File_read` and declares the symbol `MPI_File_read` as a weak alias of this implementation. If no other software component provides an implementation of `MPI_File_read`, the linker will use the weak definition and resolve references to `PMPI_File_read`. However, a performance monitor can provide its own implementation of `MPI_File_read` and has to ensure that it is linked before the MPI library. With this mechanism, the monitor can interpose MPI function calls from the application. The performance monitor itself calls the corresponding PMPI function to invoke the original implementation of the MPI library.

3.2 Data Recording

This section describes the concept to record performance data acquired by the methodology presented in Section 3.1. This recorded data is the foundation for subsequent analyses.

The shown concept handles I/O resources and activities. For instance, I/O resources reflect files, while I/O activities represent operations such as read and write. Consequently, the concept distinguishes between **definitions**, that provide detailed information about I/O resources, and **events**, that describe I/O activities during application runtime. Section 3.2.1 presents the design of definition records. Details of the event records are shown in Section 3.2.2.

3.2.1 Design of Definition Records to Represent I/O Resources

Definitions characterize resources of I/O operations. Typically, most operations of I/O APIs do not work directly on I/O resources. Instead, I/O APIs provide abstract indicators (*handles*) for I/O resources that are used by subsequent operations. The example of reading a file with POSIX I/O operations illustrates this idea. The file represents an I/O resource. First, the source code developer needs to open the file. This can be done via the `int open(const char *pathname, int flags)` routine. Therefore, the developer specifies the path to the file (`pathname`) and the access mode (`flags`). In this example, the file will be opened in read-only mode (`O_RDONLY`). Listing 3.1 illustrate a corresponding source code fragment.

Listing 3.1: POSIX I/O operation to open a file in read-only mode.

```
int fd = open( "log.txt", O_RDONLY);
```

In case of a successful operation, `open` returns a new file descriptor. This file descriptor acts as an abstract handle of the I/O resource and is used by subsequent data operations. The file descriptor holds status information such as the current positioning (offset) within the file. This method allows multiple processes/threads to access the same file independently. The design of definition records follows this principle and distinguishes between I/O resources and file descriptors. Figure 3.6 depicts the design and interactions of its components. The **IoFile** definition characterizes I/O resources, whereas the **IoHandle** definition represents file descriptors. The next paragraphs describe each definition in detail.

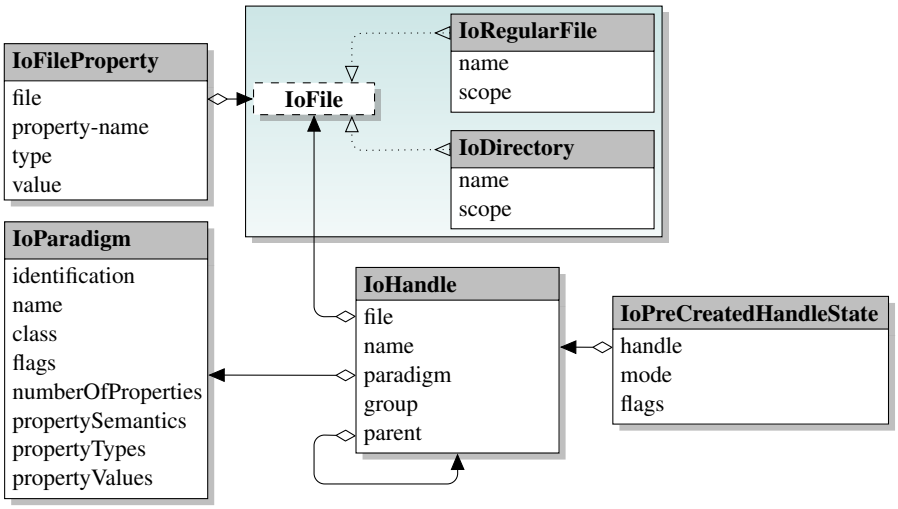
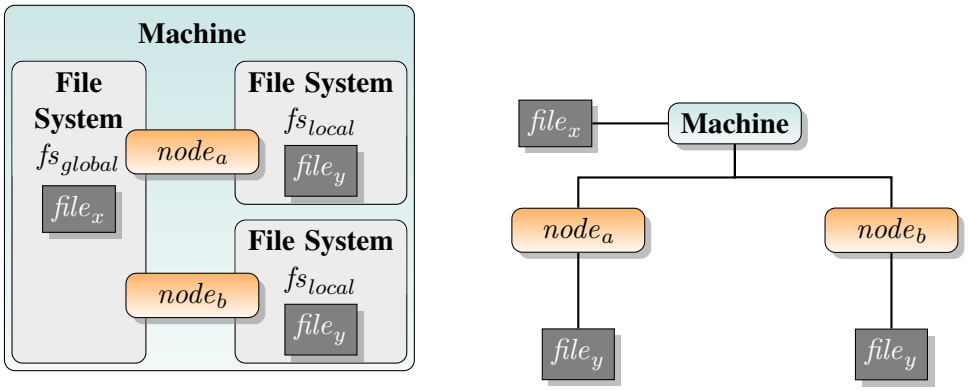


Figure 3.6: Overview of definitions to reflect I/O resources and their relations.



(a) Hardware topology (b) System tree representation

Figure 3.7: The storage location of a file determines its scope. For example, all processes can access a file stored in a global file system (e.g., $file_x$). However, a file stored in a node-local file system, such as $file_y$, can be accessed only by processes running on the same node.

Definition of I/O Resources Linux is the de-facto standard operating system on HPC machines. As a Unix-like operating system, Linux adheres to the "*Everything is a file*" philosophy. The operating systems treats a wide range of I/O resources as a file. Not only files but also directories and sockets are handled this way. The polymorphic **IoFile** definition reflects this philosophy. It provides a common namespace for objects used by I/O operations. In its current version, the design provides definitions for files (**IoRegularFile**) and directories (**IoDirectory**). The extensible design facilitates additions within this namespace. Each *IoRegularFile* and *IoDirectory* definition records the `name` of a file or directory. However, the name or path of a file or directory does not represent a unique identifier for the I/O resource. Typical HPC machines mount several file systems concurrently. These file systems differ in their accessibility. In principle, two categories of file systems can be distinguished: a) local file systems available only on a single compute node, and b) global file systems shared via network on the whole machine. Figure 3.7 shows an example. Figure 3.7a illustrates the hardware topology of a system with two compute nodes $node_a$ and $node_b$. Each node mounts two different file systems. First, both nodes use a shared network file system fs_{global}. Second, each node mounts a local scratch file system fs_{local}. The file $file_x$ in the global file system fs_{global} is accessible on the whole machine. In contrast, the file $file_y$ resides in a local file system fs_{local}. As a result, two processes, one running on $node_a$ and the other executed on $node_b$, work on distinct physical files if they access $file_y$. The `scope` attribute of the *IoRegularFile* and *IoDirectory* definition records mark the physical scope with regard to the system topology. Figure 3.7b shows the system tree representation of this example. The definitions of $file_y$ (local file system) reference the corresponding compute node in the system tree. Accordingly, the definition of $file_x$ (global file system) refers to the machine node.

In addition, the **IoFileProperty** definition enriches an *IoFile* definition by user-defined attributes. An *IoFileProperty* references an *IoFile* definition (`file`), has a specific `property-name`, declares its the data `type`, and holds the `value` of the property. The unique tuple (`file`, `property-name`) identifies a specific property. For example, this mechanism allows users to attach mount point or Lustre stripe policy data to an *IoFile* definition.

Definition of I/O Handles This thesis presents a methodology to analyze parallel scientific applications using multiple I/O libraries concurrently such as MPI I/O, HDF5, and NetCDF. This kind of analysis requires information about each I/O library. I/O operations initiated by the application propagate through the I/O software stack. For example, an `open` call will also cause open operations in lower levels of the I/O software stack. As a result, each level of the I/O software stack maintains own file descriptors to manage I/O resources. Figure 3.8 showcases an application utilizing MPI. In this example, the MPI I/O implementation maps its functionality on POSIX I/O. In case of writing data, each layer of the I/O software stack may rearrange operations or add additional meta information to the actual raw data.

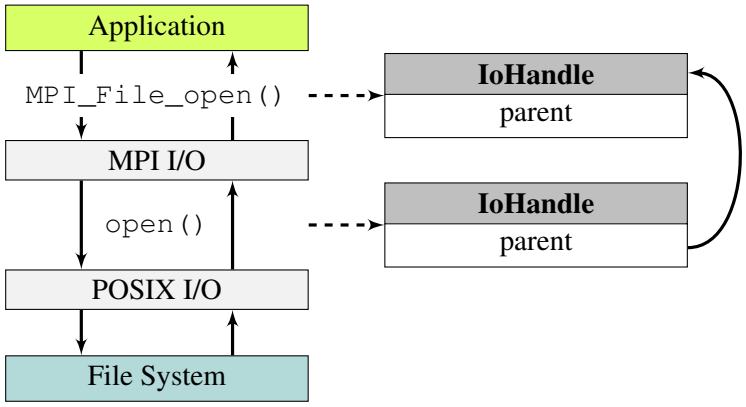


Figure 3.8: Illustration of the hierarchical relation between I/O handles. In this example, the `MPI_File_open` routine internally uses the POSIX I/O `open` function. The *parent* relation connects both corresponding I/O handles.

Consequently, the methodology presented in this work has to capture information about individual I/O libraries and assign individual operations to corresponding I/O libraries. An **IoParadigm** definition describes an I/O library utilized by the observed application during its execution. The `identification` attribute categorizes an *IoParadigm*, while the `name` distinguishes specific implementations. For example, Open MPI is an open-source implementation of Message Passing Interface (MPI). The MPI 2.0 standard defines an API for parallel I/O (MPI I/O). Open MPI provides two modules to implement MPI I/O: OMPIO [20] and ROMIO [107]). In this case, the `identification` attribute is set to "MPI I/O" and the `name` contains either "OMPIO" or "ROMIO". The `class` attribute of the *IoParadigm* definition specifies whether it is a serial or parallel I/O paradigm. Only parallel I/O paradigms permit collective I/O operations within a group of multiple processes/threads. The `flags` attribute describes further boolean characteristics for the I/O paradigm. For example, users can specify whether an I/O paradigm directly accesses the operating system or is a high-level library, i.e., it maps its functionality to other I/O paradigms such as HDF5 or NetCDF. Properties represent an extensible mechanism to specify further information of an *IoParadigm* definition. `numberOfProperties` specifies the number of stored properties. Three arrays, each with a size of `numberOfProperties` elements, contain the actual information of each property. The array `propertySemantics` describes how to interpret each property, `propertyTypes` defines the data type of each property, and `propertyValues` stores the value of each property. The following example of adding library version information to an *IoParadigm* illustrates the concept. In this case `numberOfProperties` is 1 and each array consists of one element. The element `propertySemantics[0]` contains "version information", `propertyTypes[0]` defines "string", and `propertyValues[0]` stores "7.0-alpha".

The **IoHandle** definition reflects a file descriptor. The definition is based on a prior I/O resource definition which is referenced by the `file` attribute. A human-readable `name` facilitates users in identifying an *IoHandle*. If the associated `paradigm` supports collective I/O operations, the `group` attribute specifies the set of participating processes/threads. Explicit modeling of hierarchical relations between I/O handles and thereby reflecting the stratified nature of the I/O software stack is an important feature of the presented methodology. The `parent` attribute of an *IoHandle* expresses these relations between I/O handles. This mechanism directly connects associated I/O handles across individual layers of the I/O software stack and enables correlation of affected operations. Considering the example shown in Figure 3.8, an `MPI_File_open` results in a POSIX I/O `open` function call. Both layers define their own *IoHandle* definitions. First, the MPI operation produces an *IoHandle* definition on the MPI I/O layer. Then, it calls the POSIX I/O operation that defines its own *IoHandle*. The POSIX I/O *IoHandle* references the MPI I/O *IoHandle* as its parent.

The **IoPreCreatedHandleState** definition associates special characteristics to a previously defined `handle`. This kind of definition marks symbolic handles (e.g., `stdin`, `stdout`, `stderr`) or handles

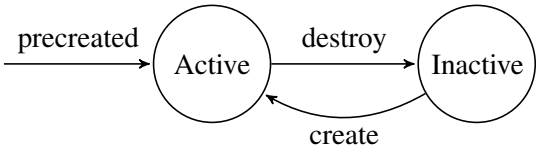


Figure 3.9: During the execution of an application runtime events record the creation and destruction of I/O handles. An I/O handle is active from its creation until its destruction. Commands to duplicate handles build a special case. The original handle remains in the active state. The newly created handle changes from the inactive to the active state.

that were inherited from a parent process/thread. In addition, the definition contains the access `mode` (e.g., read or write) and status `flags` of such a special I/O handle.

3.2.2 Design of Event Records to Represent I/O Activities

Events represent I/O activities during the execution of an application. As explained in Section 2, this thesis focuses on events relevant for performance analysis and assumes that monitored I/O operations finish successfully. Nevertheless, the presented methodology is not limited to performance analysis. Section 6.2 introduces ideas to extend the methodology, handle unsuccessful I/O operations, and thereby widens the scope of this work from performance analysis to debugging.

This work distinguishes events into two categories: metadata and data transfer operations. Metadata operations, such as `open`/`create` and `close`, create, manipulate, or close I/O handles. Data transfer operations comprise `read` and `write` activities. Irrespective of their type, all events record an accurate timestamp and information about the issuing process/thread. Additional information depends on the specific event type. The next paragraphs introduce individual events and explain their semantics using the example of corresponding POSIX I/O operations.

Event Records for Metadata Operations Events of this category indicate the creation, destruction, or manipulation of file descriptors. Especially, the creation and destruction events define the life span of an I/O handle. An I/O handle is active after its creation and before its destruction. Figure 3.9 illustrates the life cycle of tracked I/O handles.

The **IoCreateHandle** event marks the creation of a new file descriptor. For example, an `open` operation triggers this event. The event references the `handle` and records the access mode to the file descriptor such as read-only, write-only, or read-write in its `mode` attribute. Furthermore, the *IoCreateHandle* event reports optional `creationFlags` and `statusFlags`. The concept of these two attributes corresponds to the file creation and file status flags of the POSIX I/O API. For example, `creationFlags` indicate whether a file will be created if it does not already exist. The `statusFlags` mark when a file is opened in append mode. In addition, I/O APIs provide operations to duplicate an existing file descriptor, e.g., the POSIX I/O `dup` function. The **IoDuplicateHandle** event represents such a duplication. The event references the original file descriptor (`oldHandle`) and the newly created one (`newHandle`). The *IoDuplicateHandle* activates the `newHandle`. The `oldHandle` remains active. There are two options to handle status flags of the new I/O handle. Either the new I/O handle inherits status flags from the old handle or the *IoDuplicateHandle* event explicitly records the status flags of the new new I/O handle. In the first option, analysis tools would have to maintain current status flags of all I/O handles to obtain exact status information in case of a duplication event. Therefore, the design of the *IoDuplicateHandle* event records the status flags explicitly and frees analysis tools of tracking status flags on their own. An **IoDestroyHandle** marks the end of an active I/O `handle`'s lifetime. For example, a POSIX I/O `close` operation triggers this event. A pair of consecutive *IoCreateHandle* and *IoDestroyHandle* events defines the life span of an I/O handle. Within its life span a handle is active and can be used by other events.

IoSeek and **IoChangeStatusFlags** events record changes to the status of active I/O handles. Routines such as POSIX I/O `lseek` adjust the offset of read/write operations within an opened file. An **IoSeek**

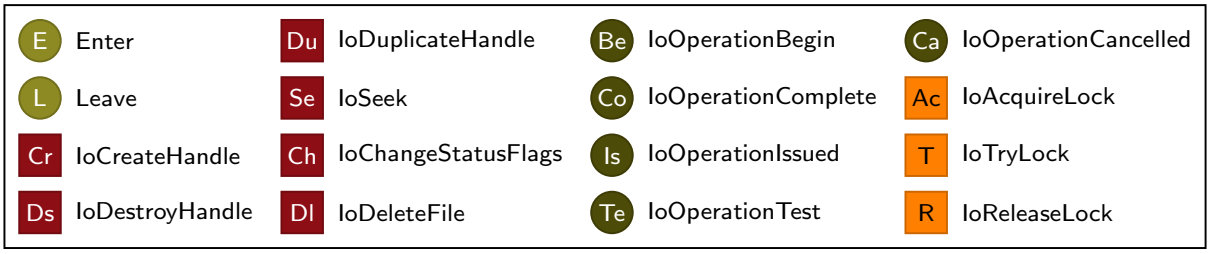


Figure 3.10: This figure gives an overview of event types and their representation in timeline charts. The two events shown as green circles describe function entries and exits. Red boxes depict events for I/O metadata operations and dark green circles illustrate I/O data transfer operations. Orange boxes represent events for file locking operations.

event reflects changes to the current positioning. The event reports the offset requested by the user in the `offsetRequest` attribute. The `whence` attribute defines how to interpret this offset. For instance, the offset can be applied as an absolute displacement from the start or end of the file or relative to the current position within the file. The `offsetResult` attribute reports the resulting offset relative to the beginning of the file. An **IoChangeStatusFlags** event tracks changes to the status flags of an active `handle` and records current status information in the `statusFlags` attribute.

Figure 3.10 illustrates the list of I/O events and their representations in the following timeline charts. The timeline chart shown in Figure 3.11 depicts a sequence of metadata operations. Enter and Leave events mark the entry and exit of each I/O routine. In this example, the sequence starts with an `open` operation that creates a new file descriptor. The *IoCreateHandle* event reflects this activity and activates a new I/O handle accordingly. Afterwards, the `lseek` operation uses and manipulates the file descriptor. Consequently, the corresponding *IoSeek* event references the active I/O handle, indicated by a dotted line, and records changes to the read/write file offset. The `close` operation finalizes this sequence of I/O operations. Thus, the *IoDestroyHandle* event marks the deactivation of the affected I/O handle.

I/O APIs also offer routines to delete file or directory names from the file system. For instance, POSIX I/O provides the `int unlink(const char *pathname)` routine. It deletes `pathname` from the file system. Additionally, if `pathname` was the last link to a file and no processes have the corresponding file open, the file associated with `pathname` will also be deleted. Consequently, a call to `unlink` might result in the deletion of an I/O resource. The **IoDeleteFile** event represents this kind of activity. In contrast to the previous events, it operates on an I/O resource referenced by the `file` attribute. The event record also logs the `paradigm` that issued the deletion.

Event Records for Data Transfer Operations Events of this category reflect data transfer operations. A complete data transfer operation consists of basic events. These events need an identifier to correlate all parts composing a complete data transfer operation. Therefore, these kind of events contain a `matchingId` attribute. This attribute identifies an I/O operation in-flight and is valid for a process including all its threads.

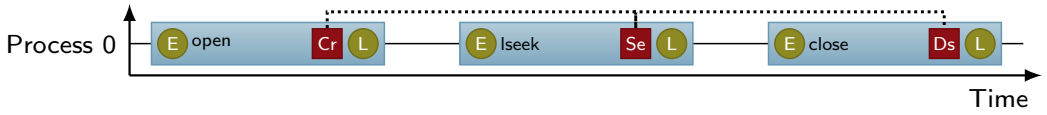
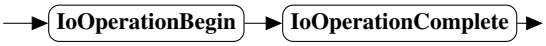
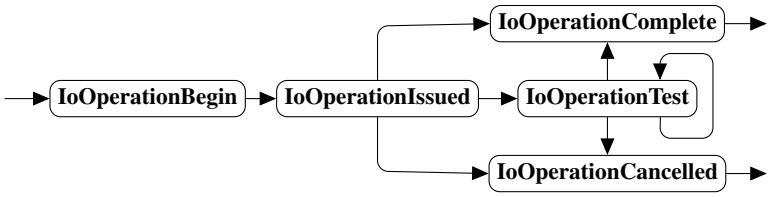


Figure 3.11: Example of a sequence of metadata operations. Blue bars represent function calls of the application. Green circles and red boxes illustrate the recorded events for this sequence of operations. The *IoCreateHandle* (Cr) and *IoDestroyHandle* (Ds) events define the life span in which the corresponding I/O handle can be used by other I/O events, such as an *IoSeek* (Se) event. The dotted line indicates that all three I/O events reference the same I/O handle.



(a) Event sequence of blocking I/O operations.



(b) Event sequence of non-blocking I/O operations.

Figure 3.12: State diagram illustrating event sequences for (a) blocking and (b) non-blocking I/O operations.

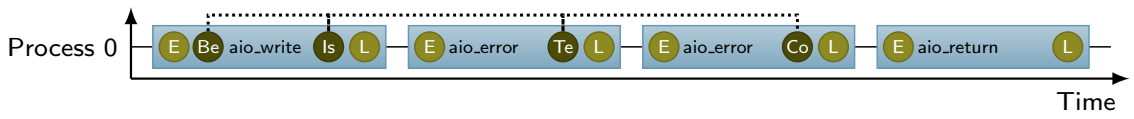
The recorded event sequence depends on the characteristics of the observed data transfer operation. This paragraph first introduces two basic events to start and finalize a data transfer operation. Then, this paragraph details different characteristics of data transfer operations and their effect on the generated event sequences.

The **IoOperationBegin** record marks the begin of a data transfer operation. The event lists the affected `handle`, the operation `mode` (e.g., reading or writing), and `operationFlags`. The last attribute provides additional semantic information. In particular, the `operationFlags` attribute determines two distinct characteristics of an operation. It specifies whether a data transfer operation is a) collective or non-collective, and b) blocking or non-blocking. The `bytesRequest` attribute logs the user defined maximum number of transferred bytes. The `matchingId` attribute defines the identifier also used by subsequent events of this data transfer operation. An **IoOperationComplete** event marks the end of a data transfer operation. It references the affected `handle`. The `bytesResult` attribute stores the actual number of transferred bytes. The `matchingId` attribute represents the identifier to correlate all associated events of an data transfer operation.

IoOperationBegin and *IoOperationComplete* records constitute basic elements of an event sequence. The difference of timestamps from corresponding *IoOperationComplete* and *IoOperationBegin* events defines the duration of the transfer operation.

The specific event sequence generated by a data transfer operation depends on whether the operation is blocking or non-blocking. Figure 3.12a illustrates the event sequence generated by blocking I/O data transfer operations. For instance, monitoring a blocking POSIX I/O `read` operation results in two I/O event records. One record represents the start of the operation and the other one represents its completion. The "blocking" bit of `operationFlags` in the *IoOperationBegin* event is set accordingly. Due to the semantics of a blocking operation, a pair of matching *IoOperationBegin* and *IoOperationComplete* events occurs within the event stream of the same thread. Figure 3.13a shows a blocking I/O operation and its corresponding events.

(a) Event sequence of a blocking I/O operation.



(b) Event sequence of non-blocking I/O operations.

Figure 3.13: Example of (a) blocking and (b) non-blocking I/O operations and their corresponding event sequences. A common `matchingId` attribute correlates the *IoOperationBegin* (Be) and *IoOperationComplete* (Co) events.

In contrast, Figure 3.12b depicts the event sequence of a non-blocking I/O data transfer operation. A characteristic of non-blocking operations is the decoupling of issuing and completing the operation. For example, an application may use asynchronous POSIX I/O operations such as `aio_write`. The function call to initiate the operation enqueues the operation for later processing and directly returns. Even after the return of the function, the application cannot assume that the data transfer is complete. Consequently, the application has to test for completion of the operation. This arrangement can become more complex. For example, one thread might start the operation but another thread of the same process might complete it. The recorded event sequence of a non-blocking data transfer operation also starts with an *IoOperationBegin* event. According to Figure 3.12b, the remaining event sequence diverges depending on the result of the *IoOperationBegin* event. In case of a successful initiation an **IoOperationIssued** event follows. The *IoOperationIssued* event references the affected `handle` and `matchingId`. *IoOperationBegin* and its corresponding *IoOperationIssued* event must occur on the same thread. Next, applications can test active non-blocking operations to ensure their completion. If the I/O operation was not finished, the test returns without success. An **IoOperationTest** event represents such an unsuccessful test. This event references the affected `handle` and `matchingId`. In contrast, an *IoOperationComplete* event indicates a successful test, i.e., the I/O operation has finished. The **IoOperationCancelled** event records the successful cancellation of a non-blocking I/O operation. The event references the affected `handle` and `matchingId`. Any thread of the same process can test, cancel, or complete a non-blocking I/O operation in-flight.

Figure 3.13b shows a sequence of non-blocking I/O operations and its corresponding events. In this example, the application starts a non-blocking I/O data transfer via the `aio_write` routine. The *IoOperationBegin* (Be) event reflects the begin of the operation. The *IoOperationIssued* (Is) event marks when the operation is enqueued, i.e., handed over to the operating system to be processed. Afterwards, the application can test the operation for its completion. In Figure 3.13b, the first call to `aio_error` returns with the exit code `EINPROGRESS` to signal that the operation has not completed yet. Consequently, the recorded event sequence contains an *IoOperationTest* (Te) event. The second call to `aio_error` reports a successful completion of the asynchronous data transfer, reflected by an *IoOperationComplete* (Co) event. Subsequently, the applications calls `aio_return` to check the return status of the operation.

Another important characteristic of data transfer operations is their collective or non-collective nature. I/O operations are called *collective* if they involve a group of processes/threads. The "collective" bit in the `operationFlags` attribute of the *IoOperationBegin* event marks the special semantic of such operations. The `handle` referenced by the I/O operation defines a group of processes/threads.

Figure 3.14 shows four MPI ranks (Process 0 - Process 3) performing a collective blocking I/O operation. The dotted line indicates two distinct types of correlation between the recorded I/O events. First, the *IoOperationBegin* (Be) and *IoOperationComplete* (Co) events on each process are correlated by a

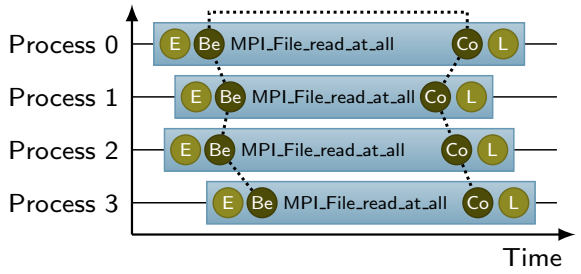


Figure 3.14: Illustration of four processes executing a collective blocking I/O operation. The `matchingId` correlates the *IoOperationBegin* (Be) and *IoOperationComplete* (Co) events on each process. The group information of the I/O handle associates the I/O data transfers across the four processes.

common `matchingId`. Second, the I/O handle referenced by the shown I/O events contains group information that associates the I/O data transfers across the four processes.

Event Records for Locking Operations Events of this category deal with obtaining, testing and releasing locks on I/O resources. The event sequence starts with a process requesting a lock. The recorded event depends on the kind and result of this operation. On the one hand, lock operations can block until the requested lock is granted. On the other hand, non-blocking lock operations return immediately and report about the status of the operation. The return code of a non-blocking lock operation informs whether the lock was granted or not, e.g., because the lock was held by another process. An **IoAcquireLock** event marks the acquisition of an I/O lock, whereas **IoTryLock** indicates that the lock was not granted. An **IoReleaseLock** event marks the release of a previously granted lock. All events of this category reference the affected `handle` and specify the `type` of the lock. A lock is either *shared* or *exclusive*. Multiple processes may hold a shared lock at the same time. For instance, several processes can concurrently acquire a read lock for a file without the risk of mutual data corruption. Only one process can hold an exclusive lock at a time, e.g., to protect write operations on a shared file.

Figure 3.15 shows an example of locking operations. In the first call to `flock` the requested lock is not granted. Therefore, the event sequence contains an *IoTryLock* (T) event. Afterwards, the application executes a second call to `flock` and requests another lock. In this case, the lock is granted and an *IoAcquireLock* (Ac) event is recorded. Finally, the application releases this lock in the third call to `flock`. This results in the recording of an *IoReleaseLock* event.

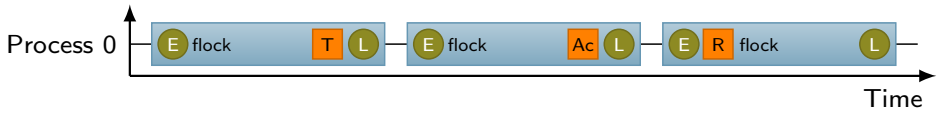


Figure 3.15: Example of a sequence of file locking operations and their corresponding I/O events.

3.3 Data Analysis

Section 3.2 introduced the methodology to record performance data including hierarchical relations between I/O operations. This data is the foundation for analysis techniques as described in this section. The captured trace data contains event sequences of I/O operations that reflect access patterns of the observed application. Analyses of these patterns identify potential inefficiencies in the I/O behavior of applications and guide users in the process of I/O performance optimization.

Section 3.3.1 focuses on the analysis of an individual application. It introduces access patterns detectable with the multi-layer I/O analysis approach presented in this work.

Nowadays, scientific applications are often embedded in complex workflows. Therefore, Section 3.3.2 widens the scope and presents an approach to analyze entire scientific workflows.

3.3.1 Definition of Multi-layer I/O Access Patterns in Applications

In order to leverage the potential of sophisticated parallel I/O subsystems applications have to access their data in an appropriate manner. As several studies [66, 17] revealed this is not the case for many current applications. Often applications perform a lot of accesses to small, non-contiguous chunks of data. This kind of access pattern induces a high load on the I/O subsystem. In addition, parallel applications issue I/O requests by multiple processes. For strong scaling experiments, where the problem size stays constant while the number of processes increases, the average data size of an I/O request tends to decrease. These trends intensify contention of I/O subsystems and result in suboptimal I/O performance of applications.

Therefore, best practices recommend to perform I/O operations in few and large chunks [68]. For instance, an aggregation of multiple small I/O requests into few larger ones will minimize the number of requests and thereby reduce the load on the I/O subsystem. I/O libraries such as MPI I/O or HDF5 provide users options to specify hints about the file layout and data access. Based on this information the libraries apply internal optimizations, e.g, data prefetching and aggregation. Because these internal optimizations are not directly visible to users, but influence the I/O performance of an application, analysts need information about both I/O access patterns of applications and internal optimizations of I/O libraries to successfully evaluate application performance.

This work is designed to observe operations across individual layers of the I/O software stack. As a consequence, it offers the potential to reveal effects of internal optimizations, e.g., applied by I/O libraries. Section 3.3.1.1 showcases how internal optimizations alter I/O characteristics from the user's API perspective. In addition, two techniques to reduce the number of requests to the parallel file system are discussed: *Data Sieving* (Section 3.3.1.1) and *Collective Buffering* (Section 3.3.1.2). The identification of these patterns allows analysts to prove the effectiveness of internal optimizations in I/O libraries.

3.3.1.1 I/O Access Patterns On Individual Processes/Threads

This section introduces I/O access patterns observable in the event log of an individual process or thread.

Varying Number of Function Calls Within the complex I/O software stack, high-level libraries map their functionality to low-level libraries. Often this is not a direct mapping. In some cases, a single function call to the high-level library results in multiple calls to a low-level library.

In the example shown in Figure 3.16 an application writes a large portion of data to the file system. Therefore, the application uses an appropriate API call provided by a high-level I/O library (`hl_write`, blue bar). The high-level library realizes the data transfer by utilizing a low-level I/O library (`ll_write`, light red bars). Additionally, the high-level library splits the operation into multiple write calls to the low-level library each with a block size of $10\ MB$. Such an effect is not directly visible from a user's perspective and requires information collected across multiple layers of the software stack. The increased number of I/O requests might stress the I/O subsystem. Therefore, this pattern guides developers and users of high-level I/O libraries in investigating inefficiencies in the usage of high-level I/O libraries.

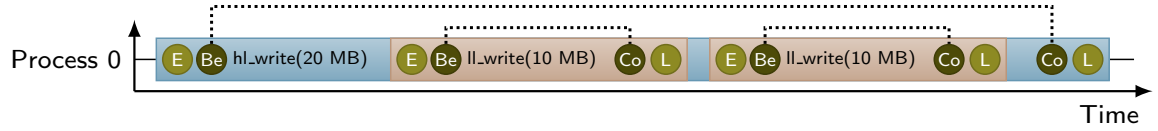


Figure 3.16: Illustration of a high-level I/O library mapping its functionality to a low-level library. In this example, a single function call to the high-level library (`hl_write`) results in multiple calls to the low-level library (`ll_write`).

Varying Number of Transferred Bytes Similar to a varying number of function calls across the software stack, internal optimizations within the I/O libraries might alter the number of transferred bytes. A well-known optimization is data sieving, where an I/O library rearranges I/O requests internally [105, 106].

Data Sieving The concept of data sieving aims at combining small I/O accesses made by an application into larger ones. For instance, ROMIO, an implementation of MPI I/O, realizes data sieving. MPI provides developers the option to create file views [72, Chapter 13.3] assigning regions of a file to individual processes. However, a file view might assign non-contiguous portions of data to a process. Figure 3.17 illustrates an example where a MPI process reads data from a file using `MPI_File_read()`. Due to the process' file view, the read request maps to six independent, non-contiguous chunks of data within the file. The top chart of Figure 3.17 depicts the file layout. As a consequence, the MPI implementation has to gather data from the individual chunks. A naive approach would generate six requests to the I/O subsystem, one for each chunk. Data sieving combines these I/O request into a single request ranging from the first requested to the last requested byte. As a result, it reads a contiguous chunk of data into an intermediate buffer (middle chart of Figure 3.17). The intermediate buffer may hold more data than requested by the user. Afterwards, only requested chunks of data are copied to the application's buffer.

On the one hand, data sieving reduces the number of requests to and thereby the load on the I/O subsystem due to the combination of I/O accesses. Because I/O requests are high latency operations, applications also benefit from a decreased number of these operations. On the other hand, data sieving transfers more data than requested by the user.

Effects of optimization strategies such as data sieving manifest in recorded event logs. Comparing correlated I/O events from a high-level (e.g., MPI I/O) and a low-level library (e.g., POSIX I/O) reveals that the amount of transferred data differs on the individual layers. Figure 3.18 shows an illustration.

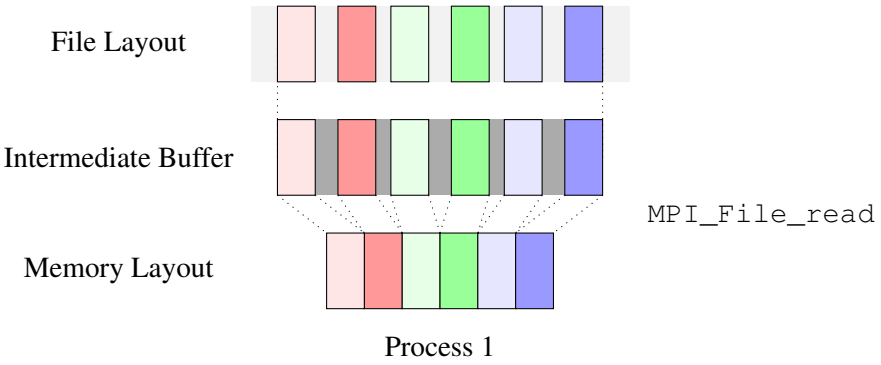


Figure 3.17: Data sieving combines multiple small I/O accesses of a process into one larger request. This technique reads a large contiguous portion of data, including the requested chunks (top), into an intermediate buffer (middle). Only requested chunks are moved to process' memory (bottom).

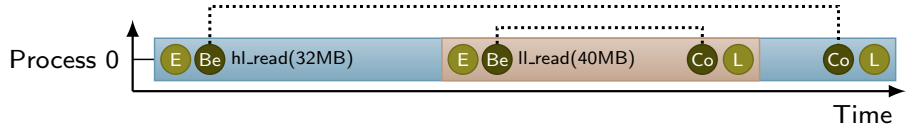


Figure 3.18: Illustration of a high-level I/O library mapping its functionality to a low-level library. In this example, the high-level library applies optimizations such as data sieving. Consequently, the total amount of transferred data differs on the individual layers of the I/O software stack. From the high-level perspective (`hl_read`), the user reads 32 MB. However, from the low-level perspective (`ll_read`) a total of 40 MB is read.

3.3.1.2 I/O Access Patterns Across Multiple Processes/Threads

In contrast to Section 3.3.1.1, this sections presents I/O access patterns involving multiple processes/threads.

Collective Buffering Collective buffering is a strategy for optimized handling of I/O requests made by several clients [104, 106]. It can be accomplished by storage devices, servers, or directly by clients themselves. This section explains collective buffering at the client level which is also known as two-phase I/O.

Figure 3.19 depicts an example where three processes read chunks of data from a file. The top chart illustrates the file layout and the distribution of data blocks to processes. While all processes together read the entire file, data blocks for individual processes are scattered across the file. If each process would read its data chunks individually, a lot of small non-contiguous I/O requests would be generated which results in an inefficient I/O behavior. In such situations collective buffering can improve the efficiency of I/O accesses. Therefore, collective buffering collects access information from all processes, rearranges I/O requests in a way that each process accesses a large contiguous data block, and splits the data transfer into two phases. Irrespective of the distribution of data chunks the file is segmented into domains and each process is responsible for a specific domain of the file (top chart Figure 3.19). The domain represents a large contiguous block within the file. In the first phase, each process reads data of its domain into an intermediate buffer (middle chart). In the second phase, processes communicate and redistribute data to corresponding destinations (bottom chart).

The concept of collective buffering prefers the I/O access to a large contiguous block of data over multiple accesses to small non-contiguous blocks. This idea assumes that a large contiguous file access reduces time spent in I/O operations and amortizes communication costs.

Aggregation Based on the concept of collective buffering aggregation utilizes some processes as proxies (aggregators) for I/O requests. As shown in Figure 3.20, instead of all processes accessing the I/O subsystem individually, $Process\ 0$ acts as a representative for all three processes and issues a large I/O requests to the I/O subsystem. $Process\ 0$ reads a large chunk of data into its intermediate buffer. After retrieving the data $Process\ 0$ redistributes data to all other processes. This concept reduces the number of requests to and lowers the load on the I/O subsystem.

Figure 3.21 showcases the concept of aggregation and its effect on event logs using MPI I/O as an example. For all processes the event log contains a call to the routine `MPI_File_write_at_all`. In this example, the high-level MPI I/O library internally uses POSIX I/O to realize data transfers. As the figure shows, $Process\ 0$ acts as an aggregator for I/O requests. Only the event stream of $Process\ 0$ reports a call to the POSIX I/O routine `pwrite`.

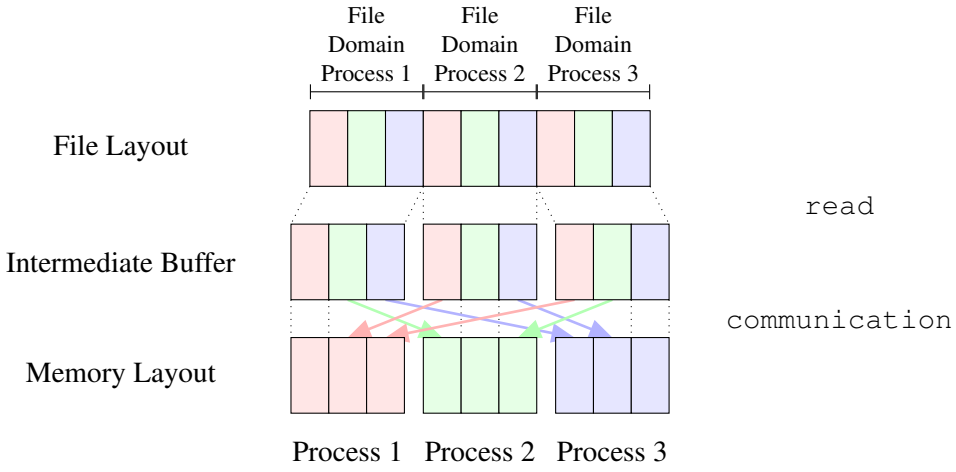


Figure 3.19: Collective buffering reorganizes I/O requests to generate accesses to large contiguous chunks of data (top). Participating processes communicate data from their intermediate buffer (middle) to achieve final distribution of data (bottom).

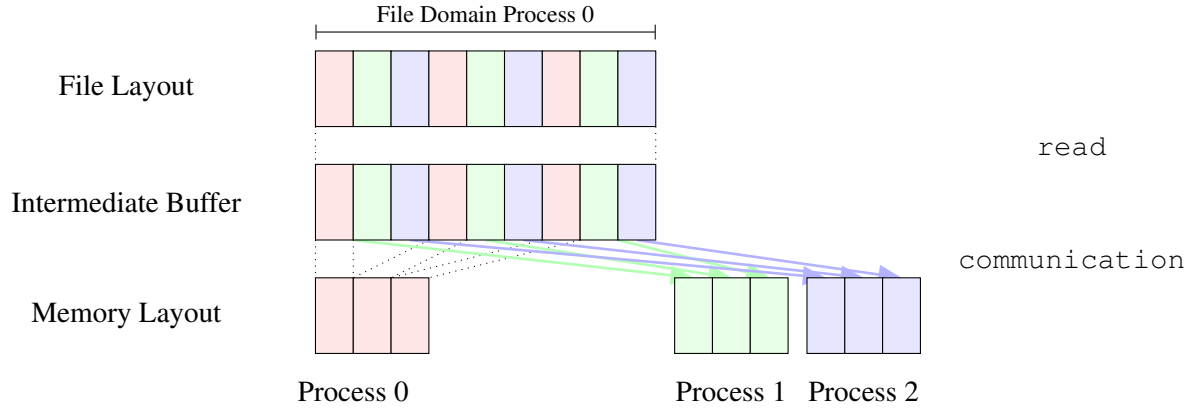


Figure 3.20: Aggregation utilizes selected processes as proxies for requests to the I/O subsystem. In this example, *Process* 0 acts as an aggregator. Communication between processes realizes the final data distribution.

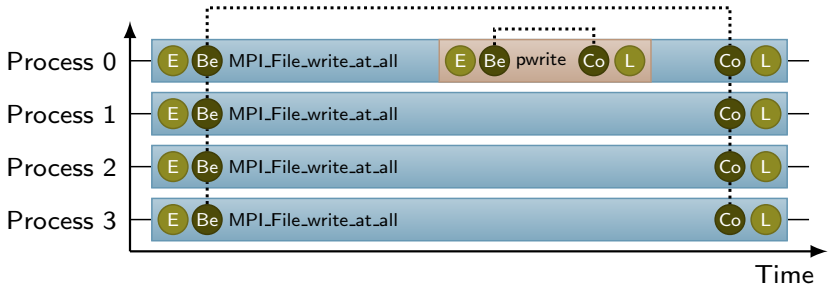


Figure 3.21: Illustration of an event log highlighting the effect of aggregation. Despite all processes making function calls to the high-level I/O library (`MPI_File_write_at_all`), only *Process* 0 issues an I/O request via a call to the low-level `pwrite` routine.

3.3.2 Analysis of Scientific Workflows

The previous section focused on the analysis of individual parallel scientific applications. This section widens the scope and presents an approach to analyze entire scientific workflows. Current research is usually characterized by collaborations of scientists from various domains, large-scale simulations on parallel and distributed platforms as well as analysis of large data volumes (data analytics). This composes multiple applications and requires well-defined coordination to configure appropriate series of operations. Scientific workflows have been established for this task [8]. Within a workflow scientists systematically describe tasks, express dependencies between tasks, allocate compute resources to execute tasks, and manage data. Workflow Management Systems (WMS) control task executions on distributed hardware resources.

In this thesis a workflow represents a coordinated sequence of interdependent applications. A *Workflow* consists of one or more *Jobs*. For example, a *Job* reflects a single submission to the scheduling system. Each *Job* comprises one or more *Job Steps*. For instance, a *Job Step* executes a single application. Figure 3.22 shows an example of a workflow with three individual jobs. *Jobs* and *Job Steps* can depend on each other, e.g., if a *Job Step* reads the output of another preceding one (data dependency) or if users specify a "happens-before" relation between *Jobs* or *Job Steps* in their workflow configuration (control dependency). In Figure 3.22 arrows illustrate control dependencies. In this example *Job Step* $J_{A.2}$ of *Job* J_A depends on *Job Step* $J_{A.1}$ of the same job, i.e., $J_{A.2}$ can start its execution only after $J_{A.1}$ has finished. The same applies to *Job* J_B. In addition, *Job* J_C depends on *Job* J_A and *Job* J_B. Consequently, inefficiencies of a *Job Step* do not only delay the affected *Job Step* but also propagate to depending *Job Steps* and delay execution of dependent *Jobs*. As a result, the overall workflow runtime increases. Identification of bottlenecks within a complex workflow requires tool support. Subsequent performance optimization necessitates detailed information about the workflow configuration and the runtime behavior of its components. Event logs provide this level of detail and characterize the behavior of applications. The presented features to observe I/O activities enhance the event logs and allow users to track data dependencies between *Jobs* or *Job Steps*. Therefore, based on the presented methods for data acquisition and recording, this thesis proposes a top-down approach for performance analysis of scientific workflows. This section presents the conceptual overview. Section 4.4 focuses on details of the implementation.

Methodology of the Workflow Analysis Following Shneiderman's Visual Information Seeking Mantra [93], the approach scales performance data from a global (the entire workflow) to a detailed (application level) view. For each level of the workflow the proposed approach provides relevant performance information at a suitable level of detail. Figure 3.23 illustrates the typical hierarchical structure of a workflow. The *Workflow* W_1 of this example consists of the two *Jobs* J_X and J_Y. Each *Job* com-

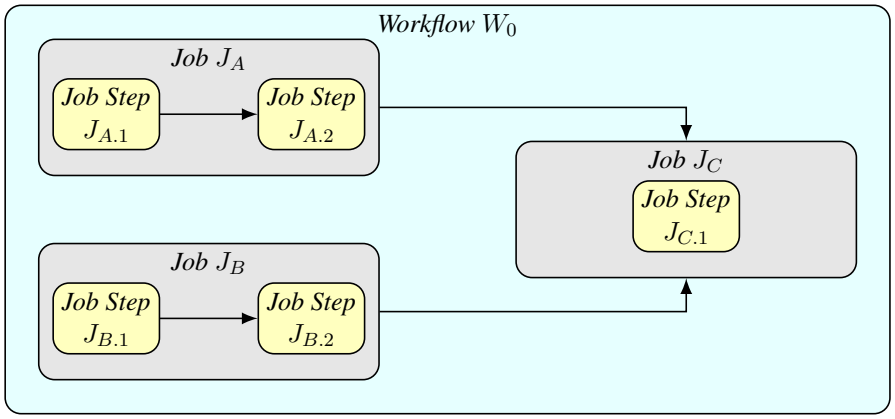


Figure 3.22: Example of a *Workflow* consisting of three *Jobs*. Each *Job Step* represents command executions, e.g., `mpirun`. Arrows indicate control dependencies.

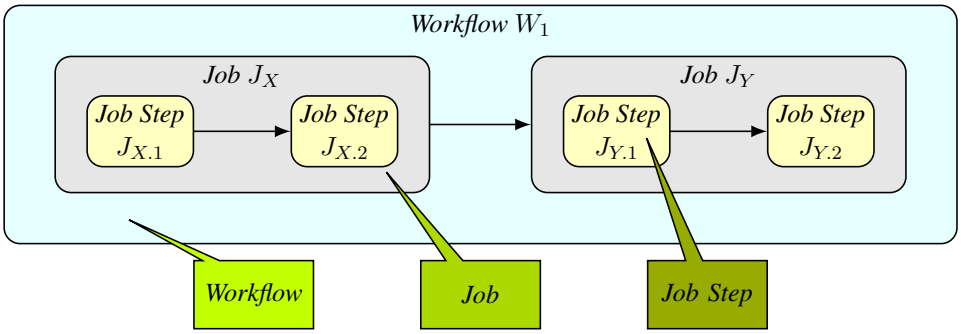


Figure 3.23: The typical hierarchical structure of a *Workflow* includes *Jobs* and *Job Steps*.

prises two *Job Steps* ($J_{X.1}$, $J_{X.2}$, $J_{Y.1}$, and $J_{Y.2}$). As shown in Figure 3.24, the analysis starts with an overview of the entire workflow. In accordance with the hierarchical structure of workflows, the analysis increases the level of detail while progressing to fine granular levels of a workflow. In the initial analysis view users investigate aggregated performance metrics to get an overview of the entire workflow. Additionally, users inspect the workflow graph illustrating job dependencies and the order of job executions. These information directs users to problematic *Jobs*. The job level analysis provides detailed performance information which precisely characterizes the *Job*. The increased level of detail enables users to identify the performance critical parts of the workflow. Finally, users examine individual *Job Steps* of the selected *Job*. This last analysis step provides fully detailed event logs for *Job Steps* to facilitate users in revealing the root cause of any performance issues detected.

The next paragraphs describe the workflow analysis procedure in detail by following the individual analysis levels.

3.3.2.1 *Workflow* Level

The goal of this initial level of workflow analysis is to give users a general overview of the entire *Workflow*, illustrate dependencies between *Jobs*, and thereby expose inefficiencies as well as unused parallelization potential.

Therefore, the proposed approach presents performance summaries of all *Jobs*. Additional runtime statistics categorize workflow time into three groups: *computation*, *communication*, and *I/O*. The analysis at this level assists users in diagnosing performance problems within the workflow, localizing affected *Jobs* and *Job Steps*, and assessing general performance characteristics. For instance, users can verify whether a workflow is communication bound, if particular *Jobs* dominate the workflow's time to completion, and estimate the impact of performance problems on the entire workflow. Section 4.4.3 details on the implementation of the *Workflow* analysis level.

At the end of this analysis level, users identify particular jobs that appear to contain performance issues. With these selected *Jobs* users continue to the next analysis level, the *Job* level.

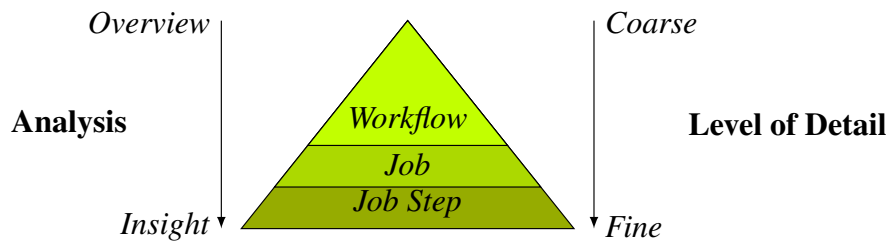


Figure 3.24: Corresponding to the hierarchical structure of workflows the top-down workflow analysis approach starts with a general overview of the entire *Workflow*. Subsequent levels provide more detailed information about individual parts of a workflow.

3.3.2.2 *Job* Level

In contrast to the *Workflow* level, analysis at the *Job* level provides statistics characterizing individual *Jobs* and their *Job Steps*. A categorization of the statistics into *computation*, *communication/synchronization*, and *I/O* guides users through the analysis process. *Job Step* statistics are the foundation for derived summaries about the total *Runtime Share* of a *Job*. The *Runtime Share* indicates the contribution of individual *Job Steps* and categories to the overall execution time of a *Job*. An overview of accessed I/O resources amends analysis options at the *Job* level. Section 4.4.2 introduces the tools used to generate these statistics.

At the end of this analysis level, users identify the most time consuming *Job Steps* or function categories. Based on this knowledge, users select individual *Job Steps* for further analysis in the next level, the *Job Step* level.

3.3.2.3 *Job Step* Level

This level features the most detailed performance information. Based on event logs the *Job Step* level analysis provides statistics about individual events such as function calls or communication operations. A performance monitor instruments applications and records event logs. Users can adjust the measurement process and thereby control the type of recorded events. For instance, users can decide to record function calls, parallelization constructs, and calls to I/O routines. With this configuration the analysis can provide a detailed outline about the computation, communication, and I/O behavior of a specific *Job Step*. Because event logs preserve the temporal application behavior, timeline views complement statistics. This allows users to ultimately diagnose root causes of performance bottlenecks, such as inefficiencies in inter-process communication or synchronization. Section 4.4.1 discusses the implementation of performance monitoring at the *Job Step* level.

4 Implementation of the Methodology for a Holistic Performance Analysis of Multi-layer I/O in Parallel Scientific Applications

The previous chapter formally introduced the methodology for recording information about multi-layer I/O operations. This chapter is dedicated to the realization of the proposed methodology. It shows the integration into a monitoring infrastructure, highlights the extensions to the trace data format, and presents the implementation of an analysis toolset.

4.1 Realization of the Methodology Within a Monitoring Infrastructure

This work does not start the development of a completely new monitoring infrastructure. Instead, this thesis provides extensions to established performance monitors, data formats, and analysis tools. This decision allows continued use of existing functionality. The thesis enhances the established performance measurement infrastructure *Scalable Performance Measurement Infrastructure for Parallel Codes* (Score-P) [53]. Score-P provides a measurement infrastructure for profiling, event trace recording, and online analysis of highly parallel applications. Figure 4.1 depicts Score-P's modular software architecture. *Adapter* components interact with the application and record information about the behavior of the application during execution time. There are adapters supporting instrumentation of process-level parallelism (MPI, SHMEM), thread-level parallelism (OpenMP, Pthreads), accelerator-based parallelism (CUDA, OpenCL, OpenACC), and source code (automatic compiler instrumentation, manual user instrumentation, PDT). In addition to instrumentation, Score-P supports sampling as an alternative method for data acquisition. After collecting information from the application, adapters provide their data to the *measurement core*. The Score-P measurement core provides internal buffers for common data management and holds collected performance data in main memory during application execution. Furthermore, the measurement core can augment the data with metric information, e.g., collected from hardware performance counters. The *online interface* allows users to access performance data while the application is running. Score-P writes collected performance data to permanent storage when the application finishes

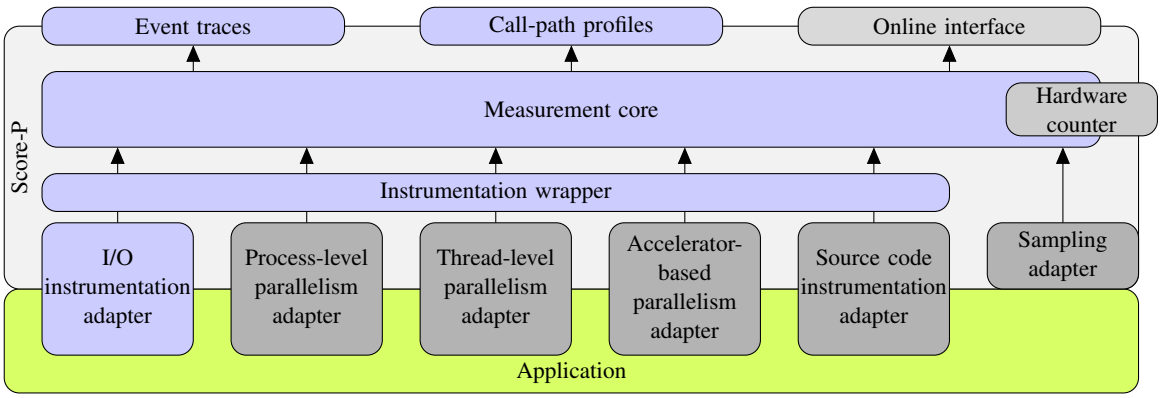


Figure 4.1: Software architecture of the Score-P measurement infrastructure. Components implemented or enhanced in this thesis are highlighted in blue.

its execution or if internal buffers are fully utilized. The *Open Trace Format Version 2 (OTF2)* [29] is the persistent data format for event logs and *Cube4* [89] for profiling data. Several analysis tools work with these data formats and interfaces. For example, Periscope [94] uses the online interface for auto-tuning by investigating different parameter settings for, e.g., compilers flags or number of OpenMP threads. The Cube report explorer tool provides an intuitive graphical user interface to visualize profile data. Scalasca and Vampir use event log data for automatic identification of common inefficiency patterns and scalable visualization of the application behavior.

The following sections detail on the individual aspects of the enhancements of the Score-P measurement infrastructure implemented in this thesis. Section 4.2 describes the additional adapters responsible for the instrumentation of calls to the I/O libraries NetCDF, HDF5, MPI I/O, and POSIX I/O. This work also enhances the Score-P measurement core to efficiently manage I/O definitions and events (Section 4.2). Furthermore, the thesis realizes options to record the collected I/O performance data persistently. Therefore, it integrates I/O records in the profile and trace data formats (Section 4.3). Figure 4.1 highlights affected components of the Score-P measurement system in blue color.

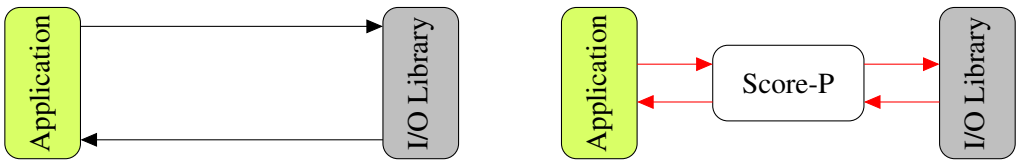
4.2 Implementation of the Data Acquisition Methods

In order to identify I/O bottlenecks in the application behavior, the data acquisition method has to provide detailed information about I/O operations, e.g., exact timings. Each individual operation might be of interest to users, especially independently of its duration. A sampling-based approach cannot guarantee to capture all I/O operations of an application, especially short-running instances (Section 2.2.1.1 on page 18). Therefore, the implementation presented in this thesis uses an instrumentation-based approach (Section 2.2.1.2 on page 20). Figure 4.2 demonstrates the interception of calls from an application to an I/O library by Score-P. Score-P acts as an additional layer interposing the call from an application to a library. This thesis implements adapters that realize the instrumentation of calls to the I/O libraries NetCDF, HDF5, MPI I/O, and POSIX I/O.

Intercepting Calls to Library Functions The interception of MPI library calls builds upon the existing MPI profiling interface (PMPI) [72, Section 14.2]. This interface is typically implemented using weak symbols (Section 3.1.3 on page 39).

The interception of calls to NetCDF, HDF5, and POSIX I/O uses a generic interception method [13]. This method parses the header files of a library in order to obtain functions provided by the library, generates wrapper source code for each function, and builds wrapper libraries. The generated wrapper libraries support both interception at link-time (Section 3.1.1 on page 36) as well as interception at execution-time (Section 3.1.2 on page 36).

When an application issues an I/O function call, Score-P intercepts this call. As a consequence, the control flow passes to the Score-P measurement system. The measurement system has access to function parameters, records performance relevant data, and calls the original function. After the original function returns, the control flow passes back to the application and the program continues its execution.



(a) Original function call sequence (b) Interception of function calls by Score-P

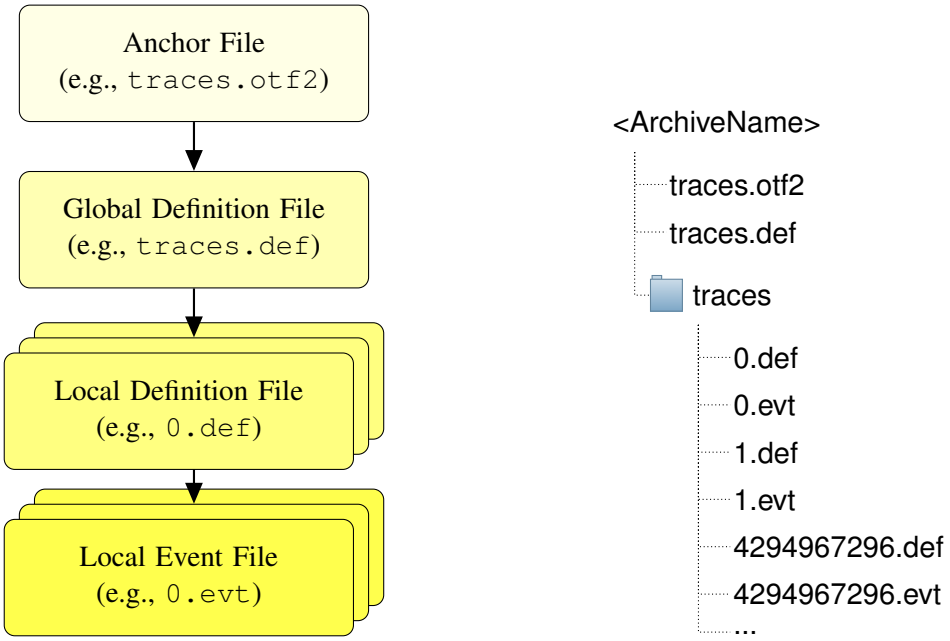
Figure 4.2: The figure illustrates the principle of library call interception with Score-P. Black arrows show the original function calls of an application into an I/O library. Red arrows illustrate the function call interception by Score-P.

Internal Data Management This work enhances Score-P by adapters for NetCDF, HDF5, MPI I/O, and POSIX I/O. According to the presented methodology, recording interactions between these adapters, e.g., if a high-level I/O paradigm realizes its functionality by utilizing another I/O paradigm, is a fundamental requirement. Furthermore, extensions of the Score-P measurement core to manage I/O definitions and events should not be limited to specific I/O paradigms. Instead, the implementation introduced in this work has to support a flexible list of I/O paradigms. This requires a generic handling of interactions between I/O adapters. A shared per-thread I/O management stack in Score-P maintains the status of current I/O operations and allows I/O adapters to communicate. The example of MPI I/O implemented on-top of ISO-C illustrates this approach. If the MPI I/O adapter of Score-P intercepts a call to `MPI_File_open`, it creates a new *IoHandle* ($handle_1$) and pushes it to the I/O management stack. Afterwards, the `PMPI_File_open` function is invoked via the MPI profiling interface. The MPI implementation may than call `fopen`, which is subsequently intercepted by Score-P as well. Score-P's ISO-C adapter inspects the top element of the I/O management stack to determine whether a potential higher-level I/O paradigm is active. If a handle is available on the stack, such as $handle_1$ in this example, this handle is used as parent for the newly created *IoHandle* ($handle_2$). After leaving `fopen` and `MPI_File_open`, the top element from the I/O management stack is removed for each involved paradigm.

In summary, a priori it is unknown whether lower-level paradigms will create new *IoHandle*s. Therefore, each I/O component must push and pop its current active handle onto the I/O management stack. This ensures appropriate references to controlling higher-level I/O paradigms in individual handles. As a result, all occurring *IoHandle*s create a root-directed tree. This generic approach offers an extendable mechanism to support I/O paradigms in Score-P.

Capturing Supplementary Information Metadata information complements recorded events of the observed application and assists users in a holistic performance analysis. Score-P already provides several options to amend recorded event data. For instance, users can configure Score-P to record performance counters using the Performance Application Programming Interface (PAPI) [103] or perf [84]. PAPI and perf provide an interface to access performance counters implemented in hardware or software. Hardware performance counters read registers of a device that count the number of specific events executed by the hardware. For instance, many modern CPUs provide performance counters reporting about the number of executed floating point operations, cache misses, or reference cycles when a processor core is not in a halt state. The number of page faults or context switches are examples of performance counters implemented in software. Furthermore, Score-P provides a metric plugin API [90]. This interface allows users to load software modules that implement additional performance counters. A variety of plugins is available for download at the Score-P repository [91].

In addition to recording I/O events, this work provides complementary file system information for later analysis. For instance, on Lustre file systems users can include information about the file system type, the mount point and source, as well as the stripe count and size of a file in their event logs. The mount point and source information is retrieved from the Linux process information pseudo-filesystem [86]. For reasons of portability, the implementation queries `/proc/self/mounts` to get the mount information. However, recent Linux versions also provide the `/proc/self/mountinfo` file. In comparison with `/proc/self/mounts`, this file supplies additional information and can also be used to obtain the information about mount points and sources. Furthermore, the implementation presented in this work uses the llapi library [62] to obtain stripe count and size information. This library allows applications to request or set Lustre properties of a file or directory. For instance, the `llapi_file_get_stripe` routine returns striping information for a given file or directory.



(a) Structure of an OTF2 archive

(b) File system representation of an OTF2 archive

Figure 4.3: The figure depicts (a) the structure of an OTF2 archive and (b) its corresponding file system representation.

4.3 Implementation of the Data Recording Methods Within a Trace Format

This thesis implements the methodology presented in Section 3.2 on page 40 as an extension of the Open Trace Format Version 2 (OTF2) [29]. OTF2 is a memory efficient event trace data format. With its compact binary format and internal compression techniques, OTF2 is able to record event logs of highly scalable applications. The OTF2 library provides interfaces for reading and writing trace data. According to the OTF2 file format specification an event log always consists of an *anchor file*, a *global definition file*, and one or more *local definition* as well as *local event files*. The number of local definition and local event files depends on the number of event streams. An event stream is the basic entity of recording and represents a unit of execution such as a process or thread. OTF2 also uses the term *location* to refer to an event stream. The entire set of files is also called an OTF2 archive. The following paragraphs describe the purpose of individual file types.

Anchor file The anchor file provides metadata about the event log such as the OTF2 version used to create this archive or an unique trace identifier.

Global definition file The global definition file contains all definitions that are equal among all locations.

Local definition file In contrast to the global definition file, a local definition file contains only definitions that apply to a certain location.

Local event file The local event file stores all events recorded on a location. Within this local log file all events are stored in chronological order.

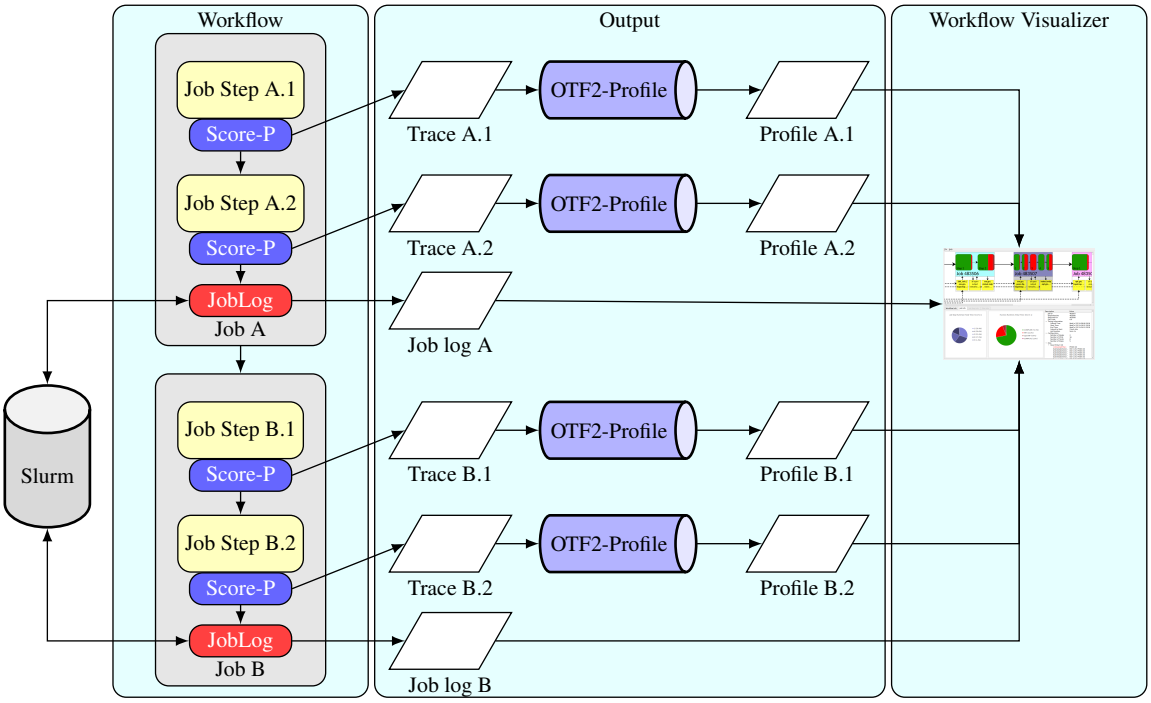


Figure 4.4: The figure depicts the workflow measurement infrastructure and illustrates interactions between individual tools. Measurements record event logs and scheduling information. A conversion of the event logs to profile data provides a statistical overview. A graphical user interface presents a user-friendly visualization of the collected data.

OTF2 already supported events for function entry and exit, parallelization constructs, and communication. This thesis extends OTF2 with definitions and events to represent I/O operations. The record design of OTF2 also distinguishes between definitions and events. Therefore, the proposed methodology translates almost identically to the implementation in OTF2. Appendix A.1 (page 111) details new OTF2 I/O definition records, Appendix A.2 (page 113) lists OTF2 I/O event records.

OTF2 maintains enumerated types to represent various categories. For example, the list of parallelization paradigms such as MPI, OpenMP, or Pthreads is implemented as a C-enumeration in the OTF2 application programming interface. Adding support for new parallelization paradigms requires an extension of the corresponding enumeration. Preserving compatibility of API versions is challenging in the presence of such extensions. As a result, inconsistencies due to unknown enumeration members would occur if older OTF2 versions read event logs written by a newer OTF2 version. There is a wide range of available I/O interfaces and an recording of I/O operations should not be limited to a fixed list of representatives. Therefore, reflecting I/O paradigms as enumeration types in OTF2 is unsuitable. This work implements the *IoParadigm* definition record using a self-describing mechanism. For the sake of convenience, the OTF2 library textually describes common I/O paradigms such as HDF5, MPI I/O, or POSIX I/O and their expected definition. Users are encouraged to follow these suggestions when generating their own event logs.

4.4 Implementation of a Toolset for the Analysis of Scientific Workflows

This section explains the implementation of the top-down workflow analysis approach presented in Section 3.3.2. It details on implementation aspects at each individual level of the workflow hierarchy.

Recording trace data at the *Job Step* level is a crucial aspect of the methodology. Detailed event logs are the foundation for the entire analysis process. Aggregated statistics are always derived from more de-

tailed information provided by the level below. Therefore, contrary to the methodology, data processing internally operates in a bottom-up fashion. Accordingly, this section initially presents implementation details at the *Job Step* level (Section 4.4.1), continues with the *Job* level (Section 4.4.2), and concludes with the *Workflow* level (Section 4.4.3).

The implementation incorporates several independent tools that acquire and process performance data at the different analysis levels. Figure 4.4 gives an overview of the implementation and its associated tools using the example of Figure 3.23.

4.4.1 Data Processing at the *Job Step* Level

The current implementation conducts performance measurements at the *Job Step* level. As shown in Figure 4.4, Score-P is used to instrument each *Job Step*, monitor the execution of individual *Job Steps*, and collect detailed performance data. The implementation configures Score-P to store obtained performance data as event logs in the OTF2 format. Established tools such as Vampir or Scalasca assist users with scalable visualizations or automatic analyses of OTF2 traces. Each *Job Step* of the observed workflow produces an individual trace. For instance, in Figure 4.4 *Job Step A.1* of *Job A* creates *Trace.A.1*. Fully detailed event logs include all information necessary to diagnose performance problems in affected *Job Steps*. Furthermore, event logs are the foundation for all aggregated statistics at the *Job* and *Workflow* level.

4.4.2 Data Processing at the *Job* Level

At this level, the implementation of the top-down workflow analysis provides a general performance overview of entire *Jobs*. Therefore, combined statistics are derived from associated *Job Step* traces. The tool OTF2-Profile [40] processes recorded event logs and generates statistics for each trace file. A subsequent procedure combines the values for all *Job Steps* belonging to one *Job*. The following paragraph presents OTF2-Profile in detail.

Job dependencies are another performance relevant aspect of the analysis at the *Job* level. These dependencies express relations between *Job Steps* such as "happens-before" and therefore affect *Job Step* ordering and execution. For example, a data dependency between two *Job Steps* results in a serialized execution of these *Job Steps*. The analysis of performance data of a single *Job Step* alone is insufficient and would not reveal this kind of inefficiency. Therefore, methods to study dependencies between *Job Steps* complement options to inspect performance details at the *Job* level. The implementation utilizes additional sources of information such as the scheduling system to provide insight into the workflow composition. This work showcases an implementation of the proposed methodology for the Slurm Workload Manager [127] that is commonly used on HPC systems worldwide. However, according to this example users can conveniently apply the methodology to other scheduling system. The paragraph after next details on querying scheduling information.

OTF2-Profile The command-line tool OTF2-Profile reads OTF/OTF2 trace files and produces statistics based on the information contained in the event logs. Because trace files can become large in size, OTF2-Profile processes traces in parallel. With this parallel execution feature users can run OTF2-Profile as a post-processing step of their workflows reusing already allocated resources.

Figure 4.4 depicts OTF2-Profile reading recorded traces and generating corresponding profiles. Each event log of a *Job Step* results in its own profile, see *Profile.A.1*, *Profile.A.2*, *Profile.B.1*, and *Profile.B.2* in Figure 4.4. OTF2-Profile already supported the output of profiles in the Cube data format suitable for the Scalasca framework and other compatible tools. Cube profiles focus on performance data of an individual application. However, this work requires additional information about characteristics of a job. Therefore, this work extends OTF2-Profile by a new high-level JSON output format. This JSON format comprises four main parts: a) metadata about the monitored job, b) a breakdown of the job's Runtime Share, c) a summary of function call information, and d) a summary of I/O handles accessed by the job.

In addition to the Cube data format, the JSON output includes additional *Job* information, such as the job ID, the number of nodes, processes and threads present in the trace, the path to the trace file, and the measurement clock resolution.

The JSON format classifies the time share of each *Job Step* into the categories *I/O*, *communication*, and *computation*. In accordance with the OTF2 data format, the category I/O comprises all I/O operations recorded in the event logs, i.e., the current implementation considers the I/O paradigms ISO C, POSIX I/O, MPI I/O, NetCDF, and HDF5. Each I/O paradigm maintains its individual statistics. *Communication* consists of MPI [71] and OpenMP [80] routines. Again, each paradigm is listed separately in the statistics. All remaining CPU time is attributed to the *computation* category. The JSON output additionally lists the serial (only one thread of execution) and parallel (more than one thread or process active) time share of the *Job*. The current implementation does not distinguish between single-node and multi-node parallelism. Based on I/O operations contained in the event log OTF2-Profile generates an additional summary about I/O handles and a list of accessed files. Each I/O handle entry includes details, such as accessing process, associated I/O paradigms, access modes, and the name of the parent file. For instance POSIX I/O file entries may point to an associated HDF5 parent file. Subsequent analysis steps utilize this information to reconstruct data dependencies.

Although OTF2-Profile produces one JSON profile for each *Job Step*, the Workflow Visualizer (see Section 4.4.3) aggregates *Job Step* profiles to generate summarized *Job* statistics at analysis time. This allows users to investigate dependencies and identify parallelization potential.

JobLog In addition to performance data, the implementation of the workflow analysis includes job scheduling information. For instance, information obtained from the job scheduling system captures the execution order of *Jobs* and their *Job Steps*. This work demonstrates an implementation for the Slurm Workload Manager. However, this approach can be easily adapted to other job scheduling systems.

A python script, called JobLog [39], queries job-specific information from the Slurm job scheduler. JobLog collects information about a *Job* after all its *Job Steps* have finished and stores the result in a JSON file. For example in Figure 4.4, JobLog creates two job log files, one file (`Job log A`) for *Job A* and another one (`Job log B`) for *Job B*.

Table 4.1 lists metrics that JobLog queries once for a running job. Since this information is defined once per job, it applies to all *Job Steps* within the job. In contrast, *Job Step* related information is queried for each job step individually. Table 4.2 lists metrics recorded for individual *Job Steps*. If a *Job* consists of various *Job Steps*, *Job Step* related information may occur multiple times in one job log file.

At this point, the collected scheduling information along with recorded data dependencies are sufficient to exactly reconstruct the execution of the workflow.

4.4.3 Data Processing at the *Workflow* Level

The analysis at the *Workflow* level combines all information gathered on the other levels. The *Workflow Visualizer* GUI provides an intuitive visualization of the data. It processes the job log files created by JobLog and the JSON profiles created by OTF2-Profile. The GUI guides users through the analysis of their workflows, starting with an overview of the entire workflow, then diving into details by selecting individual jobs and job steps.

Workflow Visualizer The graphical user interface of the Workflow Visualizer allows users to analyze individual jobs or entire workflows. Depending on that choice, users open a single job log file or a folder that contains all workflow job log and profile data files. Figure 4.5 illustrates the graphical user interface. Initially, the GUI displays the *Workflow View* main window consisting of the *Workflow Graph* (top) and the *Info View* (bottom). The Info View provides different displays focusing on individual analysis aspects. The next paragraphs introduce the Workflow Graph and the Info View in more detail.

Table 4.1: *Job* metrics queried from the scheduling system.

Field	Description
JobId	Job identifier
JobName	Name of the job
StartTime	Time the job starts running
EndTime	Time the job terminates
SubmitTime	Time of the job submission
NumNodes	Number of allocated nodes
NumCPUs	Number of allocated CPUs
NumTasks	Number of running tasks
Dependency	A list of jobs referenced by ids which must be finished before the current job can run
ExitCode	Job's exit code

Table 4.2: *Job Step* metrics queried from the scheduling system.

Field	Description
JobID	Job step identifier
NNodes	Number of nodes allocated for the step
NCPUS	Number of CPUs allocated for the step
Start	Start time of step
End	End time of the step
Elapsed	Duration in seconds of the step
JobName	Name of the executable
NodeList	List of nodes that were used
ExitCode	Exit code of the command
State	State of the step

Workflow Graph The Workflow Graph depicts jobs, job steps, and accessed files arranged in a graph to visualize job dependencies.

A *Job Box* with an unique background color represents an individual job. Each Job Box is labeled with its corresponding job identifier (`Job <job id>`). The upper part of a Job Box contains *Job Step Boxes* for associated job steps. The background coloring of a Job Step Box indicates the Runtime Share of executed function groups. For example, the almost exclusively red-colored Job Step Box (`Job 483507`, `Step 3`) in Figure 4.6 indicates that this job step spent most of its time in MPI functions. *File Boxes* in the bottom part of a Job Step Box present information about the I/O operations of a job step. There is one box for each access mode (read-only, read/write, write-only; from left to right).

Additionally, the Workflow Graph illustrates dependencies between jobs as arrows. Solid arrows represent job dependencies as deduced from the scheduling system. This information is included in the job log files. For example, the job log in Figure 4.6 lists that `Job 483507` depends on `Job 483506`. Therefore, a solid arrow between both jobs visualizes this dependency. Dotted arrows represent job dependencies derived from job start and end times. Additionally, dashed arrows indicate dependencies based on file access information.

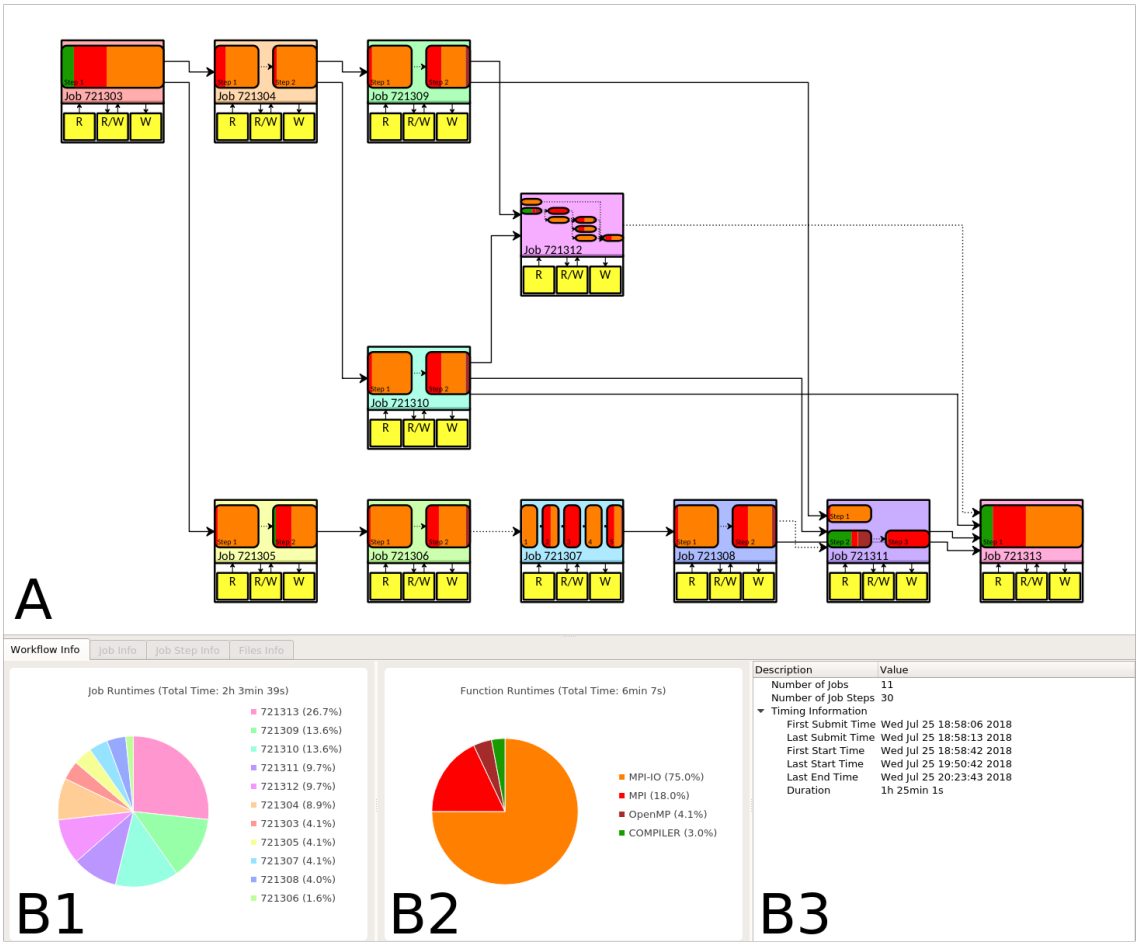


Figure 4.5: The Workflow Visualizer main window presents the Workflow Graph (A) at the top and the Info View at the bottom. In the Info View section users can switch between tabs to select information on the entire workflow, individual jobs and job steps, or accessed files. In this figure, the Info View illustrates workflow statistics. Job Runtimes (B1) and Function Runtimes (B2) are visualized as pie charts. Furthermore, an Info Table (B3) lists general workflow information.

In order to facilitate different analysis goals, the Workflow Graph offers various display modes to arrange Job (Step) Boxes. Figure 4.7 illustrates each of the three modes. The display modes are:

- *Dependency Graph*: In this mode all Job (Step) Boxes are equally sized and arranged at fixed grid positions. This kind of visualization represents the default mode and facilitates dependency analyses (Figure 4.7 A).

- *Duration Scaled Dependency Graph*: This mode adjusts the width of Job (Step) Boxes proportional to the duration of corresponding jobs and job steps. Similar to the Dependency Graph display mode, the Duration Scaled Dependency Graph places boxes using fixed size horizontal gaps. This display mode intuitively visualizes the execution time of jobs and job steps while ignoring waiting times within the batch scheduling system (Figure 4.7 B).

- *Timeline*: This mode scales widths and positions of Job (Step) Boxes based on their start and end times. The horizontal positioning of boxes follows a time axis. Consequently, multiple rows indicate parallel running jobs. This display mode is useful for performance analysis (Figure 4.7 C).

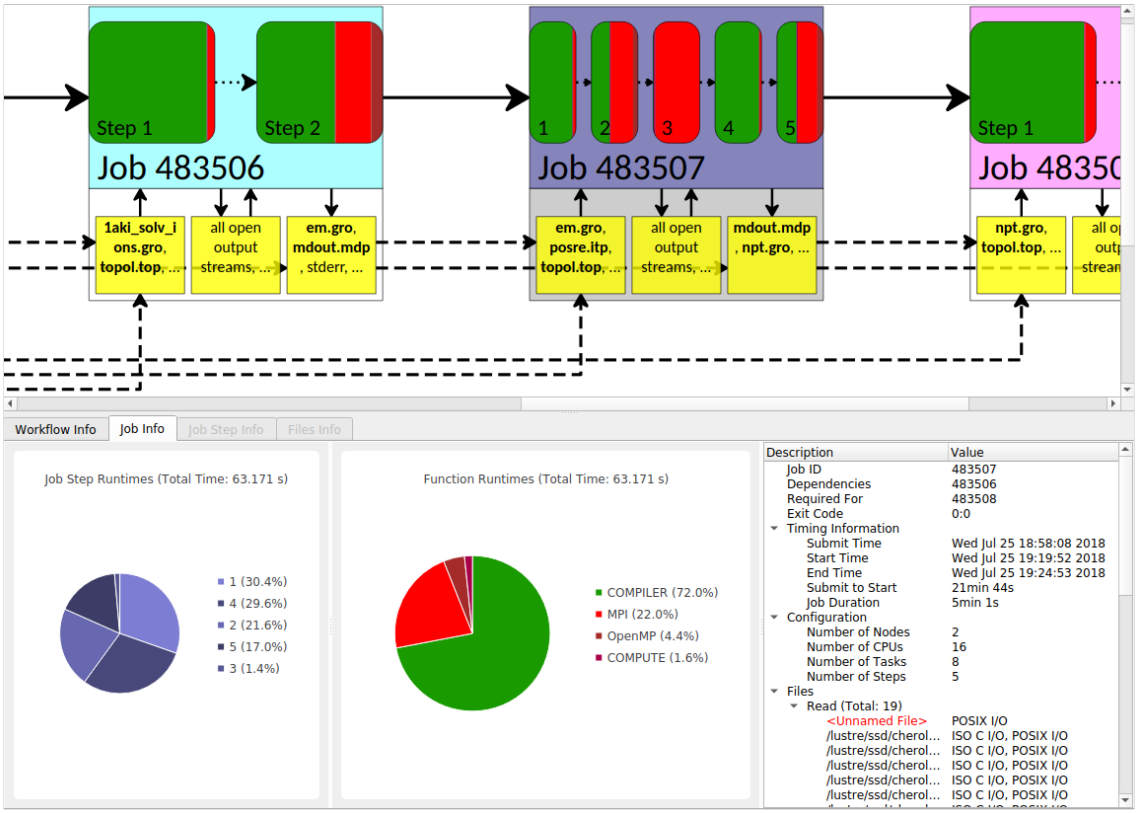


Figure 4.6: Zoom to a Job Box in the Workflow Graph. The Info View shows summaries about the selected `Job 483507`.

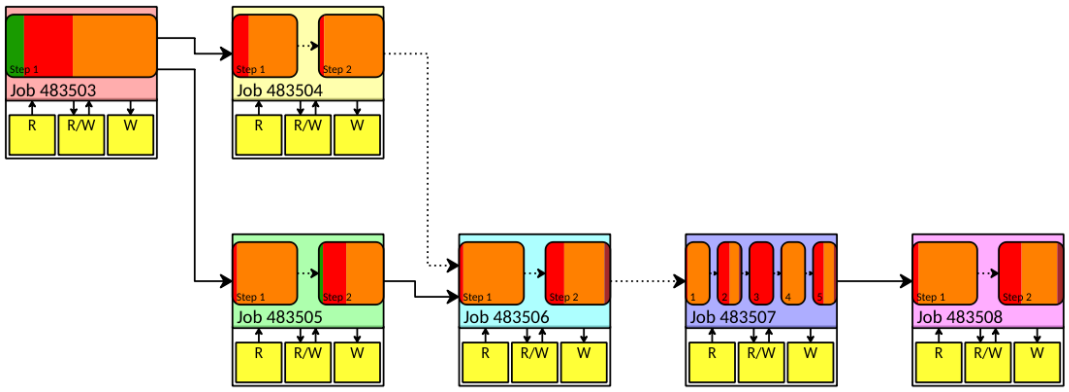
Info View The Info View (Figure 4.5, bottom) shows statistics for the item currently selected in the Workflow Graph. This display visualizes profile data for each level of the workflow. Users can choose statistics about the entire workflow, a job, a job step, or file operations by selecting the respective tab.

Figure 4.5 shows an activated *Workflow Info* tab. Therefore, this figure depicts statistics for the workflow. In this example, the Info View shows runtimes of jobs and function categories as well as scheduling information. In addition to absolute values for runtimes, pie charts visualize their respective share of the overall runtime.

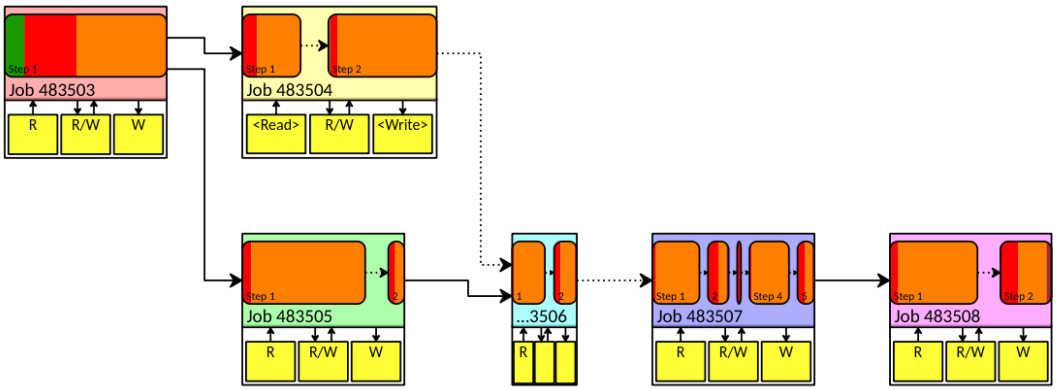
The *Job Info* tab as depicted in Figure 4.6 visualizes statistics derived from the performance data of an individual job. For instance, users can investigate the share of each job step in the overall runtime of a job, the function runtimes spent in the individual categories, start/end time of the selected job, details of the job configuration, and the list of files accessed by the job.

The *Job Step Info* tab allows the user to open related trace data in Vampir (if available on the system) to visualize the fully detailed event log.

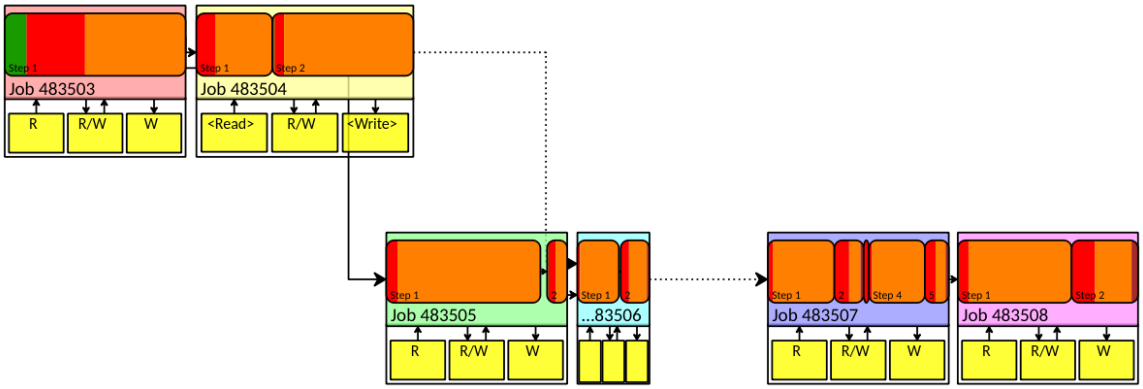
The *Files Info* tab lists access information for the currently selected file. Based on file access information, the tool derives dependencies. For example, users can easily determine all jobs which opened a particular file for writing previously, or which will open the file later for reading or writing.



(a) Dependency Graph



(b) Duration Scaled Dependency Graph



(c) Timeline

Figure 4.7: Different modes to arrange Job (Step) Boxes in the Workflow Graph: (a) Dependency Graph, (b) Duration Scaled Dependency Graph, (c) Timeline.

5 Evaluation

This chapter presents a theoretical and practical evaluation of both methodologies presented in this thesis: the multi-layer I/O monitoring of parallel applications and the analysis of scientific workflows. First, the chapter introduces the experiment design. The theoretical part contains a discussion of the overhead induced by instrumentation and presents options to control this overhead. Furthermore, results obtained from synthetic experiments evaluate specific overheads of the I/O monitoring approach with respect to execution time and memory consumption. Finally, experiments with real-world applications and benchmarks prove the applicability of this work.

5.1 Experiment Design

All experiments (except the MONC test case) shown in this chapter were conducted on the *Taurus* HPC cluster [100] at Technische Univerität Dresden.

Compute Resources The Taurus system consists of 2,117 compute nodes. The Slurm batch system is responsible for allocating and scheduling compute resources to users. Slurm groups these nodes into logical sets (*partitions*) depending on the micro-architecture of installed Intel Xeon processors. Experiments shown in this chapter use partitions equipped with Intel Haswell processors. Figure 5.1 illustrates the design of these compute nodes. They feature two sockets with each socket hosting a 12-core Intel Xeon E5-2680 v3 processor.

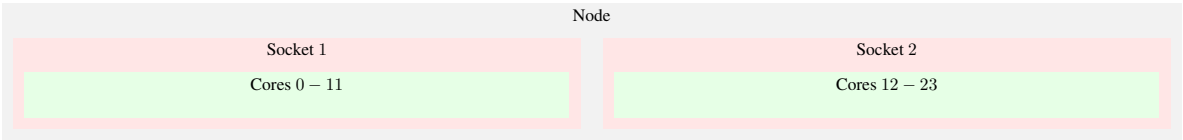


Figure 5.1: Schematic diagram of a Haswell compute node on *Taurus*.

In all tests the processors run without hyper-threading. The batch scripts submitted to the Slurm job scheduler use the option `--cpu-bind=cores` of the `srun` command to bind each compute task such as an OpenMP thread or MPI process to an individual core of the processors. Figure 5.2 illustrates the resulting task bindings. Furthermore, the use of `srun`'s `--cpu-freq` option ensures that all processor cores run at a fixed frequency of 2.50 GHz during experiments.

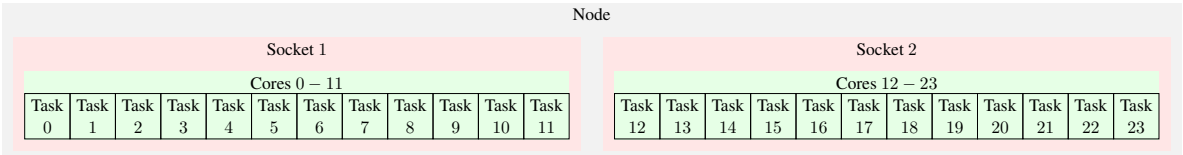


Figure 5.2: Task bindings according to the *bind-to-core* allocation method.

Software Environment Taurus uses a Bullx Linux operating system based on Red Hat Enterprise Linux Server release 7.4. The list of software modules loaded for all experiments is shown in Listing 5.1.

Listing 5.1: List of loaded software modules

```
Currently Loaded Modules:
 1) modenv/scs5
 2) GCCcore/8.2.0
 3) binutils/2.31.1-GCCcore-8.2.0
 4) GCC/8.2.0-2.31.1
 5) zlib/1.2.11-GCCcore-8.2.0
 6) numactl/2.0.12-GCCcore-8.2.0
 7) XZ/5.2.4-GCCcore-8.2.0
 8) libxml2/2.9.8-GCCcore-8.2.0
 9) libpciaccess/0.14-GCCcore-8.2.0
10) hwloc/1.11.11-GCCcore-8.2.0
11) OpenMPI/3.1.3-GCC-8.2.0-2.31.1
12) OpenBLAS/0.3.5-GCC-8.2.0-2.31.1
13) gompi/2019a
14) FFTW/3.3.8-gompi-2019a
15) ScaLAPACK/2.0.2-gompi-2019a-OpenBLAS-0.3.5
16) foss/2019a
17) Szip/2.1.1-GCCcore-8.2.0
18) HDF5/1.10.5-foss-2019a
19) ScoreP/try-io
```

File Systems Taurus provides *home* and *project* directories to store source code, share data within a group of researchers, or archive experiment results. However, compute nodes mount these directories in read-only mode. Therefore, two additional file systems host working directories.

Scratch file system This global Lustre file system provides a capacity of $4\ PB$ and is designed to provide storage space for large amounts of data. It is available under */lustre/scratch*.

SSD file system This global Lustre file system is designed to serve a large number of small I/O requests issued by large parallel applications. It is available under */lustre/ssd* and provides a capacity of $40\ TB$.

5.2 Theoretical and Synthetic Evaluation

Instrumentation is a critical aspect with respect to overhead as it might prolong the execution time of observed applications and generate large amounts of event log data. Depending on the type of the subsequent performance analysis, users might also pay attention to further aspects such as disturbance of the cache behavior of an application due to instrumentation. This section studies the execution time and memory overhead of the I/O instrumentation capabilities introduced to the Score-P measurement infrastructure in this work. Overhead induced by instrumentation highly depends on the kind of instrumented events and the frequency of their occurrence. Score-P already provided several options to instrument applications such as manual source code annotation, automatic compiler instrumentation, or interception of calls to the MPI library. The measurement system addresses overhead by offering users the possibility to selectively instrument applications. Users can select categories of recorded events by adjusting the measurement process, e.g., only enabling instrumentation of MPI routines and disabling manual as well as compiler instrumentation. In addition, Score-P supports filtering options to further refine the set of recorded events within a category. The implementation of I/O instrumentation capabilities in Score-P

adheres to this philosophy. Users can individually instrument each I/O paradigm (POSIX I/O, MPI I/O, NetCDF, and HDF5).

To efficiently use I/O operations in large-scale applications developers are recommended to avoid unnecessary output such as printing debug information to the prompt, organize their I/O operations to access data in few and large chunks, and avoid excessive calls to metadata operations such as `open`/`text` routines [68]. As a result of this recommendations, I/O operations should not appear as high-frequent operations within an application. Instrumentation benefits from this characteristic. The overhead induced by I/O instrumentation can be expected to be lower than for frequently called functions such as getter/setter routines.

There is no general rule for an acceptable overhead as this threshold depends on the goal of the analysis. For instance, 10% runtime overhead might be acceptable in order to obtain results reasonable for performance analysis and identify most time consuming parts of an application. However, users may want to decrease overhead to study fine-granular synchronization between threads.

The following results of synthetic experiments evaluate the execution time overhead and memory demands of the I/O instrumentation approach. Therefore, these experiments instrument POSIX I/O operations with Score-P.

Execution Time Overhead A worst-case scenario experiment examines the execution time prolongation caused by the I/O instrumentation. This experiment calls a pair of `open`/`close` operations within a loop. In contrast to real-world applications, it issues I/O operations frequently without using the file to perform useful work such as reading input data from or writing results to disk. Listing 5.2 shows a pseudo code skeleton of the main loop.

Listing 5.2: Pseudo code of the worst-case scenario experiment examining metadata operations.

```
int i, ret;
for ( i = 0; i < NUM_ITERATIONS; ++i )
{
    ret = open( MY_FILE, O_RDONLY|O_CREAT, S_IRUSR|S_IWUSR );
    ret = close( ret );
}
```

The evaluation comprises experiments with 10^i $(i = 2, \ldots, 6)$ iterations. For each number of iterations the measurement is repeated 10 times. The storage target of these experiments is Taurus' SSD file system. The accessed file has a stripe size of 1 MB and the stripe count is 1. Figure 5.3 contrasts execution times of two experiment setups: running an uninstrumented and an instrumented binary of the same application. The blue bars represent results obtained from experiments without any instrumentation. The yellow bars depict execution times of experiments executed with I/O instrumentation. Black errors bars indicate variability of the measurement results due to fluctuations in the execution times. In the experiments the execution times are relatively stable showing only single outliers. The figure illustrates that in both experiment setups the execution time scales linearly with the number of executed iterations, i.e., the number of executed I/O operations. Table 5.1 presents the exact measured values and thereby facilitates the interpretation of the results. According to the values shown in this table, the relative overhead in execution time is independent of the number of recorded operations. The I/O instrumentation increases the median execution time by a maximum of 11%.

Because metadata operations such as `open` and `close` are latency bound, additional experiments slightly vary the kind of I/O operations and examine the behavior of data transfers such as `read`/`write` routines. Listing 5.3 illustrates a pseudo code skeleton for the `read` test case. The `write` test follows an analog approach and substitutes the data transfer operation.

Table 5.1: Instrumentation of `open`/`close` operations within a loop and the corresponding prolongation of execution times.

Iterations[1]	Runtime [ms]										Overhead [%][2]
	Without Instrumentation					With Instrumentation					
	Minimum	Lower Quartile	Median	Upper Quartile	Maximum	Minimum	Lower Quartile	Median	Upper Quartile	Maximum	
10^2	3.20×10^1	3.30×10^1	3.40×10^1	3.50×10^1	3.70×10^1	3.50×10^1	3.60×10^1	3.70×10^1	3.70×10^1	3.80×10^1	9.48
10^3	3.52×10^2	3.55×10^2	3.62×10^2	3.63×10^2	3.66×10^2	3.78×10^2	3.81×10^2	3.86×10^2	3.94×10^2	4.03×10^2	6.51
10^4	3.58×10^3	3.59×10^3	3.61×10^3	3.61×10^3	3.62×10^3	3.93×10^3	3.96×10^3	3.97×10^3	3.99×10^3	4.06×10^3	10.17
10^5	3.59×10^4	3.60×10^4	3.61×10^4	3.65×10^4	3.75×10^4	3.94×10^4	3.98×10^4	3.99×10^4	4.02×10^4	4.08×10^4	10.48
10^6	3.62×10^5	3.63×10^5	3.64×10^5	3.66×10^5	3.67×10^5	4.00×10^5	4.01×10^5	4.05×10^5	4.07×10^5	4.09×10^5	11.04

[1] Number of recorded events = 4×number of iterations.
[2] Increase (in percent) based on the comparison of the median execution times without and with instrumentation.

Table 5.2: Instrumentation of `read` operations within a loop and the corresponding prolongation of execution times.

Iterations[1]	Runtime [ms]										Overhead [%][2]
	Without Instrumentation					With Instrumentation					
	Minimum	Lower Quartile	Median	Upper Quartile	Maximum	Minimum	Lower Quartile	Median	Upper Quartile	Maximum	
10^2	1.90×10^1	1.90×10^1	1.90×10^1	1.90×10^1	1.62×10^2	1.90×10^1	1.90×10^1	1.90×10^1	1.90×10^1	2.30×10^1	0.47
10^3	1.82×10^2	1.83×10^2	1.83×10^2	1.83×10^2	6.85×10^2	1.82×10^2	1.83×10^2	1.83×10^2	1.84×10^2	2.10×10^2	0.16
10^4	1.80×10^3	1.80×10^3	1.81×10^3	1.81×10^3	5.44×10^3	1.80×10^3	1.81×10^3	1.81×10^3	1.82×10^3	2.02×10^3	0.17
10^5	3.72×10^3	3.73×10^3	3.73×10^3	3.74×10^3	3.93×10^3	3.78×10^3	3.79×10^3	3.79×10^3	3.81×10^3	3.85×10^3	1.66
10^6	2.24×10^4	2.25×10^4	2.25×10^4	2.26×10^4	2.26×10^4	2.30×10^4	2.30×10^4	2.31×10^4	2.32×10^4	2.33×10^4	2.72

[1] Number of recorded events = 2×number of iterations.
[2] Increase (in percent) based on the comparison of the median execution times without and with instrumentation.

Table 5.3: Instrumentation of `write` operations within a loop and the corresponding prolongation of execution times.

Iterations[1]	Runtime [ms]										Overhead [%][2]
	Without Instrumentation					With Instrumentation					
	Minimum	Lower Quartile	Median	Upper Quartile	Maximum	Minimum	Lower Quartile	Median	Upper Quartile	Maximum	
10^2	8.10×10^1	8.20×10^1	8.20×10^1	8.30×10^1	1.17×10^2	8.30×10^1	8.30×10^1	8.40×10^1	8.40×10^1	8.60×10^1	2.63
10^3	8.24×10^2	8.27×10^2	8.31×10^2	8.36×10^2	8.79×10^2	8.51×10^2	8.52×10^2	8.59×10^2	8.63×10^2	1.05×10^3	3.36
10^4	8.38×10^3	8.42×10^3	8.50×10^3	8.53×10^3	8.57×10^3	8.50×10^3	8.63×10^3	8.68×10^3	8.70×10^3	1.14×10^4	2.07
10^5	8.93×10^4	9.23×10^4	9.29×10^4	9.68×10^4	9.99×10^4	8.94×10^4	9.06×10^4	9.45×10^4	9.65×10^4	1.12×10^5	1.71
10^6	1.30×10^6	1.31×10^6	1.31×10^6	1.41×10^6	1.58×10^6	1.29×10^6	1.30×10^6	1.31×10^6	1.36×10^6	1.55×10^6	0.15

[1] Number of recorded events = 2×number of iterations.
[2] Increase (in percent) based on the comparison of the median execution times without and with instrumentation.

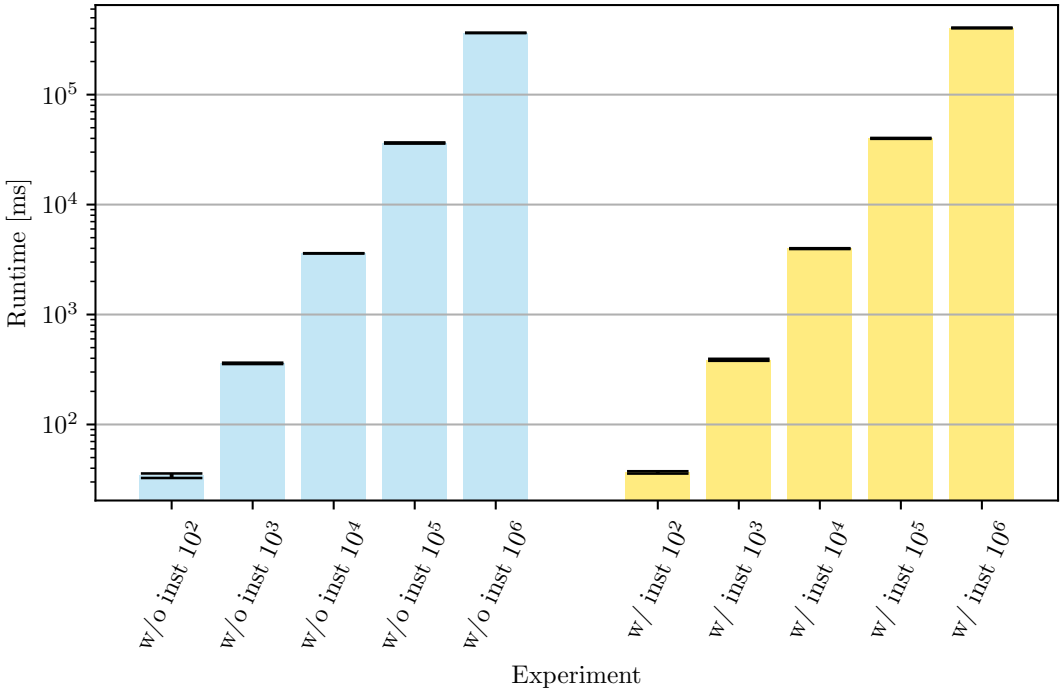


Figure 5.3: The figure shows the execution times obtained from experiments investigating `open`/`close` operations within a loop. The number of executed loop iterations varies from 10^2 to 10^6. The results shown in this figure are obtained from experiments without (blue bars) and with I/O instrumentation (yellow bars). In both setups the execution time scales linearly with the number of executed iterations . The error bars document relatively stable execution times in the experiments. The results from the experiments with I/O instrumentation applied show a prolongation of the execution time by a maximum of 11%.

Listing 5.3: Pseudo code of the worst-case scenario experiment examining data transfer operations.

```
int i, ret, bytes;
void* buffer;
ret = open( MY_FILE, O_RDONLY|O_CREAT, S_IRUSR|S_IWUSR );
for ( i = 0; i < NUM_ITERATIONS; ++i )
{
        bytes = read( ret, buffer, CHUNK_SIZE );
}
ret = close( ret );
```

Table 5.2 lists the overhead results of the `read` test case. Measurement results of the `write` test case are shown in Table 5.3. Both `read` and `write` test cases show a smaller overhead in comparison with the `open`/`close` experiment. The median execution time increases by less than 3% in the results of the `read` as well as the `write` experiment.

These evaluations prove that the presented approach for I/O instrumentation induces an acceptable execution time overhead. Even in the worst-case scenario I/O instrumentation increases the median execution time by a maximum of 11%. Therefore, event logs collected by this approach preserve the application behavior and represent valuable input for subsequent performance analyses.

Size of Event Logs In addition to the execution time overhead, the amount of memory required to store event logs during application execution is an important aspect of monitoring solutions. If events cannot be stored in memory during an application run, collected data needs to be flushed to disk. This data transfer potentially disturbs the observed application and renders acquired performance data useless for later analysis. OTF2 stores event traces in a compact binary format and uses several efficient encoding strategies to reduce the size of event logs. For instance, it automatically compresses leading zeros. Additionally, if multiple events of a sequence have an identical time stamp, OTF2 combines this chronological information and stores the corresponding time stamp only once. The size of event logs correlates with the number of recorded events. Especially for I/O recording, it is important to note that the actual size of event logs is independent of the payload of the monitored I/O operations. The event record of an I/O data transfer operation contains an attribute informing about the number of transferred bytes. However, the record does not hold the actual data of the transfer operation. Consequently, the size of the event record does not scale with the amount of data transferred by the observed I/O operation.

Varying the Amount of Transferred Data in a Fixed Sequence of Observed I/O Operations

The first experiment investigates an application that transfers a varying amount of data. This test instruments the POSIX I/O operations of an application, executes this application, and records the resulting OTF2 event log. During its execution the application opens a file and reads data from this file. Individual experiment setups vary the amount of requested data ($2^j\ MiB, j = 0, \ldots, 10$). Measurements for each individual data volume comprises 10 repetitions. The subsequent experiment evaluation examines the recorded event logs. For instance, Listing 5.4 exemplifies information contained in the event log of the test case reading 1 GiB of data from the file. As the event sequence does not change while scaling the amount of transferred data, the instrumented binary generates event logs of constant size in all setups. Table 5.4 lists the memory requirements of the individual components of the OTF2 archive. Because this test executes a serial application, the OTF2 archive consists of a global definition, a local definition, and a local event file. Slight differences in the timing and the binary name account for minor fluctuations of the event log size.

Listing 5.4: Excerpt of the information contained in the event log of the test case reading 1 GiB of data from a file.

```
Event                Location   Timestamp  Attributes
-------------------------------------------------------------------------
ENTER                       0  1060692936  Region: "open"
IO_CREATE_HANDLE            0  1062528600  Handle: "[POSIX I/O] my_file.txt",
                                           Access Mode: READ_ONLY,
                                           Creation Flags: {CREATE},
                                           Status Flags: NONE
LEAVE                       0  1062537696  Region: "open" <15>
ENTER                       0  1062559280  Region: "read" <25>
IO_OPERATION_BEGIN          0  1062566800  Handle: "[POSIX I/O] my_file.txt",
                                           Mode: READ,
                                           Operation Flags: NONE,
                                           Bytes Request: 1073741824,
                                           Matching Id: 5
IO_OPERATION_COMPLETE       0  2761088896  Handle: "[POSIX I/O] my_file.txt",
                                           Bytes Result: 1073741824,
                                           Matching Id: 5
LEAVE                       0  2761106856  Region: "read"
ENTER                       0  2761117928  Region: "close"
IO_DESTROY_HANDLE           0  2761149992  Handle: "[POSIX I/O] my_file.txt"
LEAVE                       0  2761155528  Region: "close"
```

Table 5.4: Size of the event log obtained from an application reading 1 MiB to 1 GiB of data (10 repetitions for each data volume). The size of the event log is independent of the payload of data transfers and therefore stays constant in this experiment.

Entity	Size [B]
Global Definition File	3729 - 3735[a]
Local Definition File	58
Local Event File	254 - 256[b]

[a] Varying size due to differences in ticks per second, global offset, duration, binary name.
[b] Varying size due to differences in timestamps.

Scaling the Number of Observed I/O Operations A second experiment investigates how the size of OTF2 event logs scales with the number of recorded I/O events. Similar to the first experiment, this test instruments the POSIX I/O operations of an application, executes this application, and records the resulting OTF2 event log. During its execution the application opens a file and issues `read` calls to a file within a loop. Individual experiment configurations vary the number of loop iterations and comprise experiments with 10^i $(i = 2, \ldots, 6)$ iterations. For each number of iterations the measurement is repeated 10 times. The subsequent evaluation examines the recorded event logs. As shown in Listing 5.5 each loop iteration creates 4 events. The begin and end of the execution as well as operations to open and close the file add another 12 events to the trace. Table 5.5 depicts the size of the event logs for different numbers of recorded events. The results confirm the expected behavior of the implementation in the OTF2 format. Definitions contain information about clock properties, the name of the executable, and the number of events per location. This information slightly differs between individual experiment setups as listed in Table 5.5. Therefore, the size of the global as well as the local definition files stays almost constant, except for the aspects mentioned above. The memory demand of the event file scales linearly with the number of events.

Literature documents approaches that could further reduce memory demands of event logs. Wagner [122] proposes enhanced encoding techniques to increase memory efficiency of trace data formats. Knüpfer [51] presents an advanced memory data structures that exploits redundancies in the program structure for compression. Noeth et al. [75] focuses on preserving the communication structure of MPI programs while relaxing the chronological level of detail.

Listing 5.5: The event sequence generated by a POSIX I/O `read` operation.

```
Event                Location   Timestamp  Attributes
-------------------------------------------------------------------------
ENTER                       0  3616830259  Region: "read"
IO_OPERATION_BEGIN          0  3616837583  Handle: "[POSIX I/O] my_file.txt",
                                           Mode: READ,
                                           Operation Flags: NONE,
                                           Bytes Request: 1048576,
                                           Matching Id: 5
IO_OPERATION_COMPLETE       0  3618806647  Handle: "[POSIX I/O] my_file.txt"
                                           Bytes Result: 1048576,
                                           Matching Id: 5
LEAVE                       0  3618811715  Region: "read"
```

Table 5.5: Size of the event log obtained from an application reading 1 MiB of data per loop iteration (10 repetitions for each number of iterations). The size of the local and global definition file is almost constant. The size of the local event file increases linearly with the number of recorded events.

Number of Iterations	100	1000	10000	100000	1000000
Number of Events	412	4012	40012	400012	4000012
Size of Global Definition File [B][a]	3739	3741	3744	3747	3749
Size of Local Definition File [B]	58	58	58	58	58
Size of Local Event File [B]	6591	64191	640191	6131143	61033634

[a] Varying size due to differences in ticks per second, global offset, length, binary name.

5.3 Holistic Performance Analysis of Real World Multi-layer I/O Applications

After investigating the I/O instrumentation approach in synthetic test cases, this section focuses on the performance analysis of real world applications. Therefore, event logs recorded in this section are not limited to I/O operations. Besides highlighting performance issues related to I/O activities, the following case studies demonstrate a holistic performance analysis taking computation, communication, and I/O activities into account. This section comprises three case studies. The case study of a heat transfer simulation introduces into I/O performance analysis and optimization. In addition, the NAS BT-IO and MONC experiments describe the holistic performance analysis of multi-layer I/O in parallel scientific applications in detail. NAS BT-IO and MONC use different strategies to realize efficient I/O operations. The NAS BT-IO benchmark utilizes libraries such as MPI I/O to transfer data to/from the I/O subsystem. The MONC application implements a client-server-architecture to manage its I/O operations. An event log analysis conducted for both case studies reveals the effects of these different strategies.

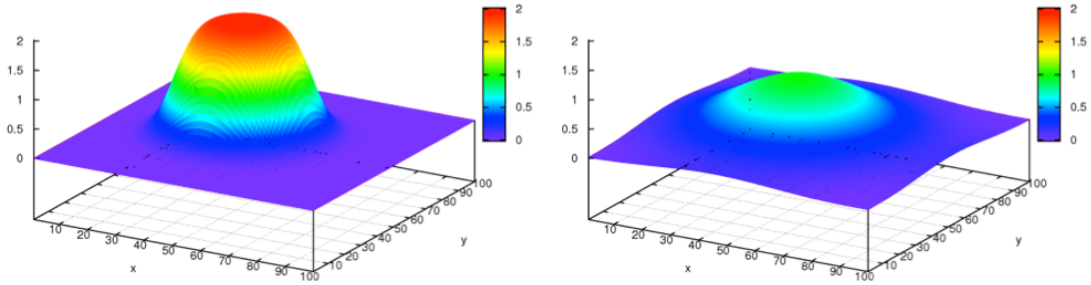
Heat Transfer Simulation This case study illustrates a basic performance analysis and optimization utilizing enhanced I/O recording options introduced in this work. The application observed in this experiment simulates heat transfer in a two-dimensional space. The following parabolic partial differential equation describes the propagation of thermal energy in this case

$$\frac{\partial}{\partial t}u(x,y,t) = a \cdot \left(\frac{\partial^2}{\partial x^2}u(x,y,t) + \frac{\partial^2}{\partial y^2}u(x,y,t) \right), \tag{5.1}$$

where a denotes the thermal diffusivity. Figure 5.4 depicts a visualization of the heat distribution in a two-dimensional space with a source of heat located in the center of the region.

The numerical solution of Equation 5.1 uses finite difference methods (FDM). These numerical methods approximate a continuous partial differential equation with a discrete equation. The heat distribution u is determined on a grid $\Omega = \{(x_i, y_j, t_k)\}$, with $x_i := i \cdot \Delta x$ $(i = 1, \ldots, n_x)$, $y_j := j \cdot \Delta y$ $(j = 1, \ldots, n_y)$, and $t_k := k \cdot \Delta t$ $(k = 1, \ldots, n_t)$, where Δx, Δy, and Δt denote the increments in x-, y-, and t-direction. The heat distribution in a given cell (x_i, y_j) of the grid at a given time step t_k is denoted as $u(x_i, y_j, t_k) := u|^k_{i,j}$. Approximating the time derivative by the forward differencing scheme and the space derivatives by the 2nd order central differencing scheme yields

$$\frac{u|^{t+1}_{i,j} - u|^t_{i,j}}{\Delta t} = a \cdot \left(\frac{u|^t_{i+1,j} - 2u|^t_{i,j} + u|^t_{i-1,j}}{\Delta x^2} + \frac{u|^t_{i,j+1} - 2u|^t_{i,j} + u|^t_{i,j-1}}{\Delta y^2} \right). \tag{5.2}$$



(a) Initial state of the simulation

(b) Intermediate state of the simulation

Figure 5.4: Illustration of the heat distribution in a two-dimensional space. The heat source is located in the center of the region. (Taken from [49].)

The solution of Equation 5.2 requires the specification of boundary conditions. These experiments use periodic boundary conditions, given by

$$\begin{aligned} u|^t_{0,j} &= u|^t_{n_x,j}, & u|^t_{n_x+1,j} &= u|^t_{1,j} \quad (\forall j = 1, \ldots, n_y), \\ u|^t_{i,0} &= u|^t_{i,n_y}, & u|^t_{i,n_y+1} &= u|^t_{i,1} \quad (\forall i = 1, \ldots, n_x). \end{aligned} \tag{5.3}$$

The implementation in the C programming language uses MPI for parallelization and I/O operations. Listing 5.6 illustrates the basic program structure. As shown in this code fragment, every 20 iterations the heat simulation writes its intermediate results (snapshot) to a file. The final result is also stored in the file at the end of the program execution.

Listing 5.6: Pseudo code illustrating the basic components of the heat distribution simulation

```
/* initialize grid from file */
loadGridFromFile();

/* initialize temporary grid */
initializeTempGrid();

/* heat distribution calculation phase */
for ( count = 0; count < max_steps; count++ )
{
        /* save intermediate result to file */
        if ( count % 20 )
        {
                saveGridToFile();
        }
        heatCalculation();
}

/* save result to file */
saveGridToFile();
```

In the experiments the application runs with 24 MPI processes on one Taurus node. All processes write to the same shared file. During the application execution, all processes generate an entire data volume of about 550 MiB. An examination of the execution times recorded during 10 repetitions of the experiment reveals large deviations. Individual program executions of the unmodified application run between 47 s and 78 s. The following performance analysis reveals the causes of these deviations. Therefore, the application is instrumented with Score-P. Vampir visualizes the trace data collected during the execution

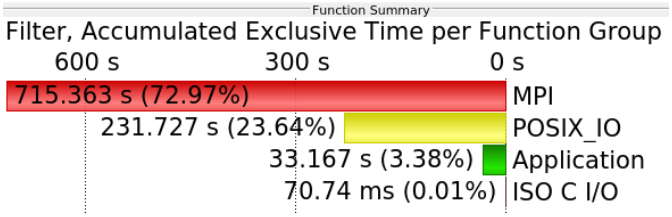


Figure 5.5: The figure depicts summary information about the accumulated time per function group. MPI function calls (red bar) consume about 73% of the execution time within the computation phase. I/O routines (yellow and brown bars) account for about 24%. Only about 3% of the time is spent within user functions (green bar).

of the instrumented application. The experiments with the instrumented version of the application show the same behavior with respect to execution time and its deviations.

Similar to other MPI programs, the heat simulation consists of an initialization (`MPI_Init`), computation, and finalization phase (`MPI_Finalize`). The performance analysis reveals that most of the execution time is spent in MPI routines. As depicted in Figure 5.5, MPI function calls consume about 73% of the execution time within the computation phase. Therefore, further analysis examines application's MPI usage in detail. Figure 5.6 shows the exclusive time of individual MPI functions accumulated over all processes. Especially, `MPI_File_open` has a large share in the execution time.

On the one hand, this performance issue stems from inefficient usage of MPI routines. For each snapshot, the application opens the result file, determines the overall number of processes as well as the rank of each process, and creates a corresponding MPI data type. Consequently, some functions are called very often, such as `MPI_File_[open|close]`, `MPI_Dims_create`, `MPI_Cart_shift`, `MPI_Type_[free|commit|vector|create_subarray]`, and `MPI_Comm_[rank|size]`.

`MPI_File_open` and `MPI_File_close` include metadata handling of the file system and therefore represent expensive operations. Event logs enhanced by multi-layer I/O events introduced in this thesis, clearly document this performance issue. At the beginning of writing a snapshot, all processes issue an `MPI_File_open` call. This behavior results in a burst of requests and induces load on the file system that cannot handle all requests at the same time. Consequently, some processes have to wait until their request is processed. As shown in Figure 5.7, MPI internally uses the `open64` POSIX I/O function. The figure also depicts the described imbalance in the execution time of `open64` calls across all processes. `MPI_File_close` shows an analog behavior.

Insights provided by the presented performance analysis result in recommendations for optimizations. Program developers should focus on improving the I/O performance as this aspect of the application is critical. For example, the result file is opened and closed for each snapshot. Developers might change the code to open the result file once at the beginning and close the file at the end of the application execution. Such a behavior diminishes the number of requests to the file system and thereby reduces the load on the file system. Furthermore, users can check whether the HPC system has access to multiple file systems. Users should select a file system that fits their needs best. In this example, the application would benefit from a file system that is able to handle a large amount of I/O requests efficiently. In addition to I/O related optimizations, the performance analysis identifies further potential for improvement. The ratio between communication and computation can be improved as well. As mentioned above, the current version of the application uses some MPI routines inefficiently. Functions to determine the number of processes and the own rank identifier are called unnecessarily often. However, this information does not change during the execution of this heat transfer simulation. Consequently, information about the number of processes and the own rank identifier should be queried once and reused later on. Additionally, the handling of MPI data types can be optimized similarly.

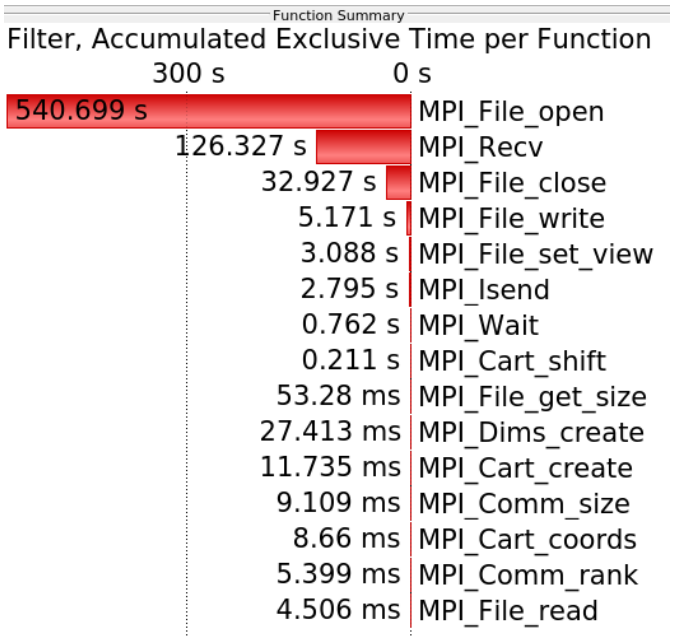


Figure 5.6: The figure shows time spent exclusively in individual MPI functions accumulated over all processes. The share of `MPI_File_open` (top bar) dominates the statistic.

Figure 5.7: This figure illustrates the application behavior during the time interval of writing a snapshot to the result file. The upper chart gives an overview of current activities across all processes. Red bars represent MPI routines, yellow bars indicate I/O operations, and green bars reflect user functions. The middle and lower chart depict the sequence of function calls for MPI Rank 0 (*Master thread:0*, top) and 14 (*Master thread:14*, bottom). The `saveGrid` routine (green bar, call stack level 2) generates a snapshot. This routine utilizes MPI I/O to transfer data to the result file, e.g., `MPI_File_open`, `MPI_File_write`, and `MPI_File_close` (red bars, call stack level 3). The call stack visualization reveals internal I/O library calls within the MPI I/O routines. For instance, the middle chart depicting *Master thread:0* shows that `MPI_File_open` calls the POSIX I/O routines `open64` and `lseek64` (yellow bars, call stack level 4). Additionally, the upper chart indicates an imbalance in the execution time of `open64` calls across all processes (variable length of the yellow bars).

Figure 5.8: This figure compares the I/O modes in NAS BT-IO (problem size *Class A*, 16 processes). It contrasts all four modes from top to bottom: *epio* (white background), *full* (purple background), *fortran* (green background), and *simple* (azure background).

NAS BT-IO This case study investigates event logs including I/O records as proposed by this thesis and thereby demonstrates the value of such event logs for a comprehensive performance analysis. Tracking the multi-level I/O behavior of applications reveals internals of I/O libraries in use.

The NASA Advanced Supercomputing Division (NAS) offers a benchmark suite to evaluate the performance of HPC machines. Kernels of the NAS Parallel Benchmarks (NPB) [1] reflect computation and data movement in Computational Fluid Dynamics (CFD) applications. This paragraph studies the Block-Tridiagonal (BT) benchmark, especially the BT-IO version [123] to analyze parallel I/O. The BT-IO benchmark applies a domain decomposition (diagonal multi-partitioning [117]) to distribute work across all MPI processes. Regulations of the benchmark require that intermediate results are written to one or more files after every five time steps. However, at the end of the benchmark, snapshot data of a single time step has to be stored in the same file. Users can chose between several problem sizes that scale the grid size in each dimension and the volume of written data. Additionally, users can select different modes to realize I/O operations. Available I/O modes are *simple* (blocking noncollective MPI I/O operations), *full* (blocking collective MPI I/O operations), *fortran* (Fortran file operations), and *epio* (each process writes its data to an individual file). The *epio* mode does not comply with the benchmark requirements. Snapshot data of a single time step is scattered across files of all processes. Therefore, a post-processing step is necessary to rearrange data. However, this step is not part of the benchmark. Furthermore, the execution of the BT-IO benchmark requires a square number of processes. At the end of each run, the NAS benchmarks provide a summary listing information about the experiment setup, execution time, and the status of the result verification.

The analysis starts with a comparison of individual I/O modes at small scale. The first set of experiments runs BT-IO with problem size *Class A* and 16 processes on one node. Configured with this problem size, NAS BT-IO calculates 200 time steps and writes about $420\ MB$ of data. Figure 5.8 visualizes event logs for each I/O mode. The modes *epio* (white background) and *full* (purple background) perform best. The modes *fortran* (green background) and *simple* (azure background) perform poorly. As the statistic chart on the right hand side of this figure shows, both modes spent most of their execution

[1] https://www.nas.nasa.gov/publications/npb.html

time in write operations. As the mode *epio* represents a special case and *fortran* does not perform well, further analysis focuses on identifying differences between the two modes using MPI I/O *full* and *simple*.

Figure 5.9 illustrates the writing of a snapshot in detail. Corresponding to Figure 5.8, the event sequence of the *full* mode is depicted with a purple background (top) and the *simple* mode with an azure background (bottom). Within the interval shown in Figure 5.9 the *full* mode writes two snapshots whereas the creation of a snapshot takes considerably longer in the *simple* mode. The *simple* mode writes only one snapshot within this interval. Both modes utilize different strategies in the `output_timestep` routine to realize snapshots. Figure 5.9 illustrates invocations of the `output_timestep` routine as black horizontal bars. In the *simple* mode (bottom of Figure 5.9) the implementation of `output_timestep` calls the `MPI_File_write_at` routine within a loop. On each MPI rank one instance of the routine `output_timestep` issues 1024 calls to the `MPI_File_write_at` routine (red horizontal bars on call stack level 3). `MPI_File_write_at` is a blocking, noncollective MPI I/O operation that uses an explicit offset to write data at the specified position of a file. The application writes $640\ B$ with each call to the `MPI_File_write_at` routine. As the figure shows, `MPI_File_write_at` internally invokes the `pwrite64` routine (yellow horizontal bars on call stack level 4). This routine writes data ($640\ B$) to the snapshot file. In the *simple* mode the call stack of all MPI processes is similar. The figure exemplifies this fact using the example of MPI Rank 0 (*Master thread:0*) and Rank 1 (*Master thread:1*). In summary, the *simple* mode realizes its data transfers by a series of multiple small I/O operations. As this example shows, such a behavior often results in poor I/O performance.

The event log recorded for the *full* mode (top of Figure 5.9) reveals major differences in handling I/O operations. First, in contrast to the *simple* mode, an invocation of `output_timestep` in the *full* mode makes a single call to `MPI_File_write_at_all` (red horizontal bars on call stack level 3). `MPI_File_write_at_all` is a blocking, collective MPI I/O operation that uses an explicit offset to write data at the specified position of a file. Second, the call stack differs between individual MPI processes. Figure 5.9 shows that `MPI_File_write_at_all` internally uses `pwrite64` on MPI Rank 0 (*Master thread:0*). However, this internal function call is missing on MPI Rank 1 (*Master thread:1*). Third, the data volume differs between the MPI I/O and POSIX I/O layers of the software stack. According to the event logs, the MPI I/O operation transfers $640\ KiB$ of data. This data volume corresponds to the sum of 1024 transfers of $640\ B$ each as observed in the *simple* mode. However, the `pwrite64` routine transfers $10\ MiB$ of data per instance.

An analysis of BT-IO experiments run at a larger scale facilitates the case study of the differing data volume between the MPI I/O and POSIX I/O layers. The next test case examines the problem size *Class D* in which BT-IO calculates 250 time steps and writes about $126.5\ GiB$ of data. Figure 5.10 depicts an overview of the application behavior during its execution. The application runs about $423\ s$ with 144 MPI processes distributed across 6 nodes. MPI processes *Master thread:0* to *Master thread:23* run on node *taurusi4145*, *Master thread:24* to *Master thread:47* on *taurusi4147*, *Master thread:48* to *Master thread:71* on *taurusi4150*, *Master thread:72* to *Master thread:95* on *taurusi4151*, *Master thread:96* to *Master thread:119* on *taurusi4157*, and *Master thread:120* to *Master thread:143* on *taurusi4158*. Furthermore, Figure 5.10 exhibits three phases within the execution: program initialization, calculation, and result validation. `MPI_Init` dominates the first phase ($0 - 4\ s$). In the calculation phase ($4 - 320\ s$) BT-IO executes user code depicted in green color to solve the BT problem. This phase also contains MPI communications (red color) and MPI I/O operations to write snapshots (purple). The third phase ($320 - 423\ s$) uses MPI I/O operations to read all previously written snapshots (blue) and validate results. Additionally, the figure also confirms that some processes behave differently (six yellow horizontal bars). In accordance with Figure 5.9 these yellow bars represent POSIX I/O operations that often appear during MPI I/O operations. The performance analysis of multi-layer I/O operations in the recorded event logs reveals effects of collective buffering within MPI I/O routines. MPI I/O applies this technique to improve the I/O performance. The analysis of the trace data clearly shows that the first MPI process of each node (*Master thread:0*, *Master thread:24*, *Master thread:48*, *Master thread:96*, and *Master thread:120*) acts as a proxy for data transfers. In contrast to other MPI ranks of a node, these processes map their MPI I/O routine to actual POSIX I/O data transfers. Figure 5.11 illustrates this behavior using *Master thread:0*

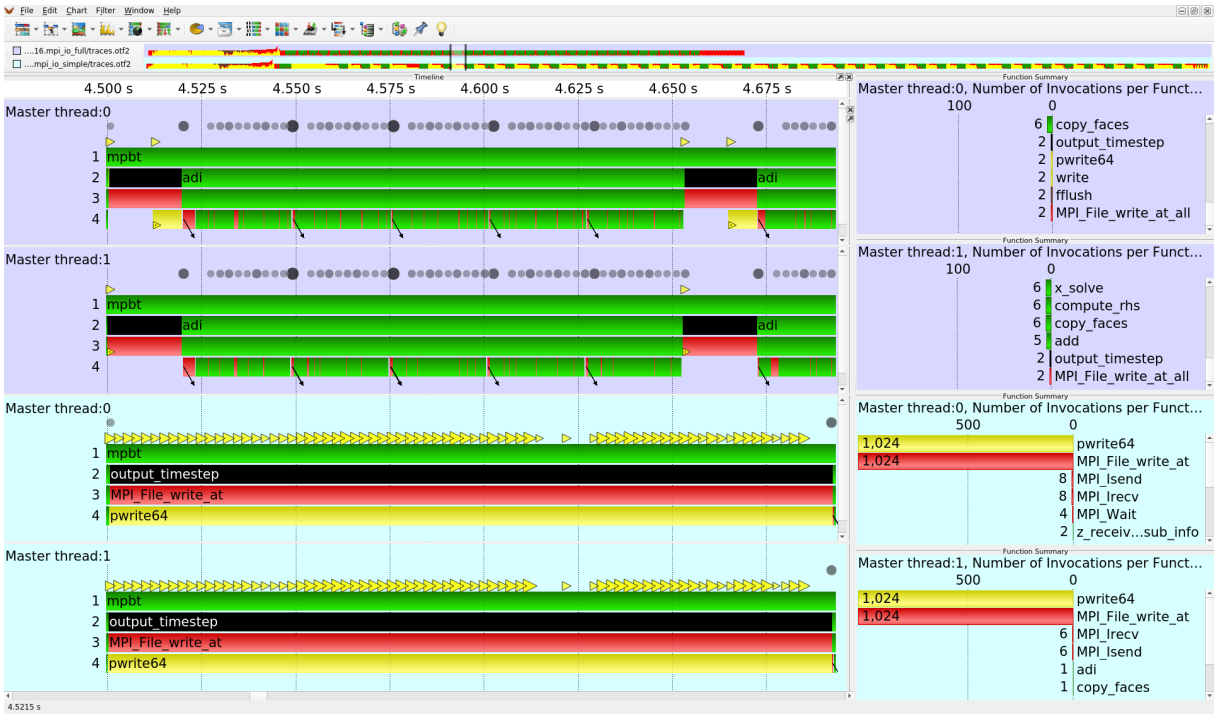


Figure 5.9: This figure illustrates a detailed comparison of the functions executed by MPI Rank 0 (*Master thread:0*) and 1 (*Master thread:0*) while writing a snapshot in the *full* (purple background) versus *simple* (azure background) I/O mode of NAS BT-IO (problem size *Class A*, 16 processes in total). The figure depicts the interval of one `output_timestep` function call in the simple I/O mode. Calls to the `output_timestep` routine are shown as black horizontal bars on the call stack level 2 of each MPI rank. As shown in the two charts at the top, the application writes two snapshots in the *full* mode within the depicted interval. The two charts at the bottom illustrate that the creation of a snapshot takes considerably longer in the *simple* mode and the application writes only one snapshot within the same interval. Both modes show a varying behavior because they utilize different strategies in the `output_timestep` routine to realize snapshots. The bar charts on the right hand side of this figure contrast the number of function calls in both modes within the selected interval. In the full mode, a call to `output_timestep` maps to one call to `MPI_File_write_at_all` (red horizontal bar at call stack level 3). Furthermore, only selected MPI ranks issue calls to POSIX I/O functions (yellow horizontal bar at call stack level 4) within the MPI I/O routine (e.g., *Master thread:0*). In the simple mode, a call to `output_timestep` maps to 1024 calls to `MPI_File_write_at` and each MPI rank calls a POSIX I/O function within this MPI I/O routine.

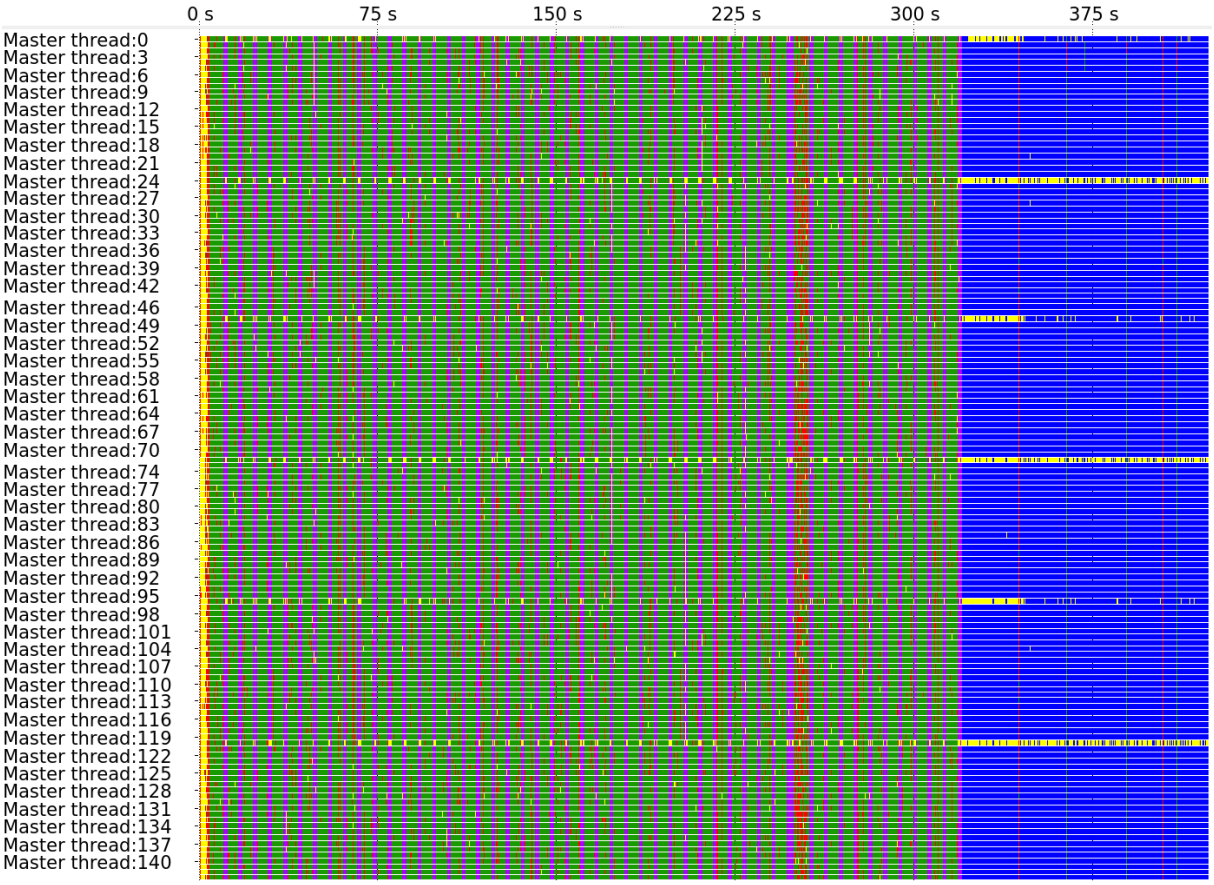


Figure 5.10: Overview of the NAS BT-IO run with 144 MPI ranks. The figure depicts three phases within the execution: program initialization, calculation, and result validation. The initialization phase at the beginning of the application execution is characterized by reading configuration and input data (yellow). In the calculation phase, the application executes user functions (green), writes intermediate results (purple), and transfers data between MPI ranks (red). After finishing its calculation, the application reads in result data and performs validation checks (blue). Furthermore, the figure illustrates differing behavior across all MPI ranks. Yellow horizontal bars indicate I/O intensive workloads on six MPI ranks (*Master thread:0*, *Master thread:24*, *Master thread:48*, *Master thread:96*, and *Master thread:120*).

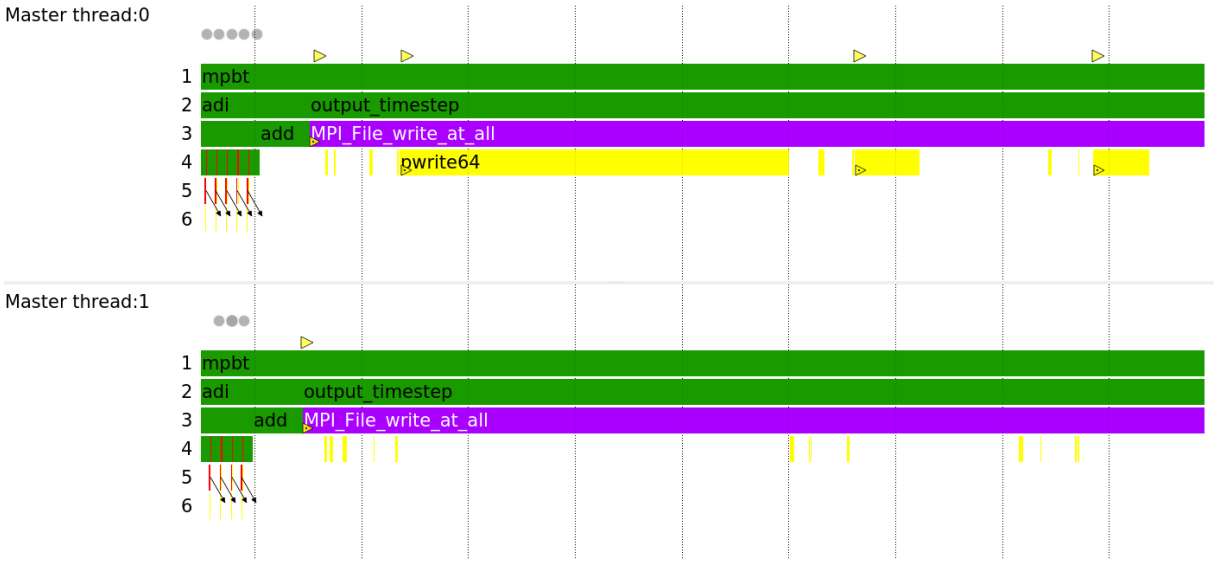


Figure 5.11: This figure compares the two MPI ranks *Master thread:0* (top) and *Master thread:1* (bottom) and their internal calls to I/O routines. Both ranks call the `output_timestamp` routine (green, call stack level 2). This routine invokes `MPI_File_write_at_all` (purple, call stack level 3). As the figure shows, only on *Master thread:0* MPI internally makes a call to the `pwrite64` POSIX I/O routine (yellow, call stack level 4).

and *Master thread:1* as representatives of the node *taurusi4145*. As shown in this figure, both processes call the `output_timestamp` routine which invokes `MPI_File_write_at_all`. However, MPI internally calls the `pwrite64` POSIX I/O routine only on *Master thread:0*. Figure 5.12 focuses on one call to `MPI_File_write_at_all` on process *Master thread:0*. Additionally, this figure highlights the invocation of `pwrite64` routines. Statistics on the right hand side of the figure illustrate that one call to `MPI_File_write_at_all` internally issues 27 calls to the `pwrite64` routine. Data of the other processes on the same node is transferred via MPI communication to the proxy that handles actual I/O data transfers. These effects of the collective buffering technique are the root cause of the differing data volume between the MPI I/O and POSIX I/O layers.

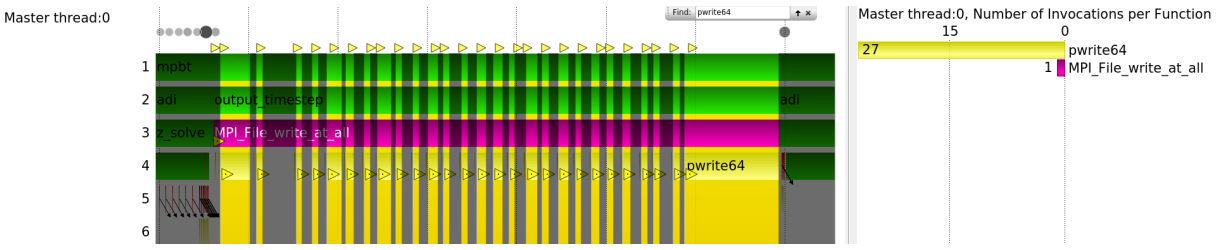


Figure 5.12: This figure illustrates effects of collective buffering within `MPI_File_write_at_all`. Process *Master thread:0* acts as a proxy for I/O requests of all MPI ranks running on the same node. Data of the other ranks is transferred via MPI communication to the proxy that handles actual I/O data transfers. Therefore, one call to `MPI_File_write_at_all` internally comprises of 27 calls to the `pwrite64` routine as the bar chart on the right hand side of the figure shows. Additionally, invocations of the `pwrite64` routine are highlighted in the left hand side of the figure.

Table 5.6: This table presents an overview of NAS BT-IO execution times (problem size *Class D*, 50 iterations) with an increasing number of MPI processes. The table reports the minimum, mean, average, and maximum execution times based on 10 repetitions of each measurement. The instrumentation overhead is within the execution time variation of the observed application.

Setup		**Time [s]**		
Number of Processes		144	576	1296
Uninstrumented	Minimum	62.49	26.16	17.90
	Mean	65.44	26.93	19.00
	Average	65.30	27.40	18.88
	Maximum	69.20	29.80	20.14
Instrumented	Minimum	62.41	25.19	17.94
	Mean	63.71	27.67	18.34
	Average	63.93	27.49	18.38
	Maximum	66.58	29.59	19.16

Scalability Study Many parallel scientific applications run at large scale, e.g., to realize detailed simulations. Often, performance problems appear or become significant at large scale. Therefore, applications cannot be analyzed only at small scale. As a consequence, performance analysis methods also have to scale. This paragraph uses the NAS BT-IO benchmark to evaluate the scalability of the proposed I/O performance analysis approach.

This experiment consists of two sets of measurements. The first set runs the unmodified binary with a varying number of MPI processes, i.e., with 144, 576, and 1296 processes. NAS BT-IO's internal timers report the corresponding execution time. The second set repeats these measurements with the performance monitor attached to the application. Based on the approach presented in this thesis, the monitor records MPI events including MPI I/O operations and stores this information in an event log. Comparing the execution times of both measurement sets reveals the overhead introduced by monitoring the application. Additionally, the experiment slightly modifies the setup used in the last analysis. In this study, BT-IO also runs with problem size *Class D*. However, the number of iterations is reduced to 50 to avoid excessive resource allocation by the measurements. The execution of the iterations shows repetitive characteristics. Consequently, the results of this scalability study report a general trend that is independent of the specific number of executed iterations.

Table 5.6 presents the execution times of the NAS BT-IO application run with 144, 576, and 1296 MPI processes. Each setup was executed in both the uninstrumented (top) and instrumented (bottom) variant. NAS BT-IO is an I/O intensive application and heavily utilizes the I/O subsystem. However, the I/O subsystem represents a resource that is shared among all users of an HPC system. As a result, the I/O workload induced by other users of the HPC system affects the performance of the observed application. The execution time of an (I/O intensive) application is expected to exhibit variations. Therefore, Table 5.6 lists the minimum, mean, average, and maximum execution times calculated from 10 repetitions of each measurement. The results show that in all test scenarios the execution time of the instrumented variant is within the runtime variation of the unmodified application. Consequently, the instrumentation does not result in a prolongation of the execution time even in large scale scenarios. The size of the event log is about $86\ MiB$ for 144 processes, $693\ MiB$ for 576 processes, and $2.3\ GiB$ for 1296 processes. According to these values, the size of the event log per process rises with an increasing number of total MPI processes. This effect is caused by the communication pattern of NAS BT-IO. With an increasing number of processes, each process issues more MPI point-to-point communication

operations. The event log includes these communication events. Therefore, the data volume of the event log increases proportional to the total number of processes. The number of issued MPI I/O operations per process stays constant. In summary, this case study proves the scalabilty of the presented approach for I/O instrumentation.

MONC This case study complements the NAS BT-IO analysis. It illustrates effects of an application that uses a deep I/O software stack and manages its I/O requests across processes manually. The analysis examines the Met Office NERC Cloud model (MONC) simulator, checks MONC for I/O related performance issues, and provides insights into operations using multiple I/O layers.

MONC is a Fortran application. It utilizes MPI for inter-process communication and NetCDF to persist its result data. The cloud simulator comprises two kinds of processes. On the one hand, there are simulation processes for computing the cloud model. On the other hand, the code implements I/O server processes for storing results to disk. Users can individually set the number of I/O server processes. During application execution, the I/O servers keep simulation results in main memory. After N simulation steps or at program termination, the I/O servers flush the result data to disk.

This experiment utilizes Score-P to instrument the source code and intercept library calls to POSIX I/O, MPI I/O, and NetCDF. In contrast to other measurements, this experiment was conducted on ARCHER. ARCHER is a Cray XC30 system and consists of 4920 compute nodes. Each compute node contains two 12-core E5-2697 v2 (Ivy Bridge) processors running at 2.7 GHz. The experiment uses a $4.4\ PB$ Lustre file system (stripe count 1, stripe size 1 GiB) to store simulation results and recorded event logs. In this measurement, MONC runs on 112 processes, distributed over 8 nodes. Each node hosts one I/O server process with a pool of 10 additional threads. The remaining 104 simulation processes compute the cloud model. In the configuration of this experiment MONC simulates 100 time steps. At the end of the application run, the I/O server processes write the data to disk via calls to NetCDF. The methodology for a holistic performance analysis of multi-layer I/O allows users to inspect internal function invocations of MPI I/O and POSIX I/O. In order to avoid interference with the I/O behavior of the observed application, all collected performance data is kept in main memory during application execution. After the application has finished, event logs are written to disk. This experiment uses a comprehensive instrumentation setup of the observed application. Nevertheless, the recording of performance data causes only a slight increase in application execution time of about 6%.

Figure 5.13 depicts the overall time exclusively spent in particular function groups. The event log contains seven groups. About 50% of the time is spent in user functions (see Application group in Figure 5.13). Furthermore, the figure shows that the simulator spends more time in MPI communication routines than in I/O operations. Although this first analysis suggests that MONC does not exhibit poor I/O performance it is worth taking a closer look at I/O operations.

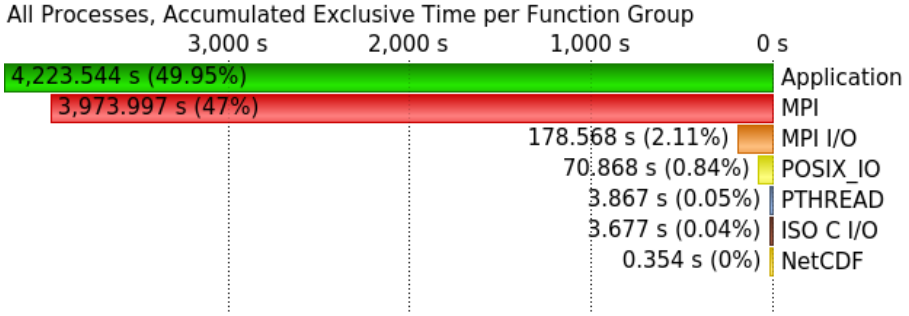
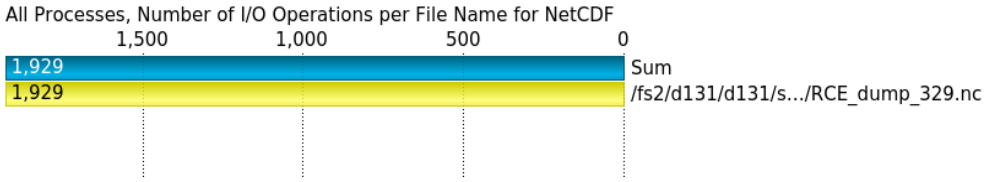
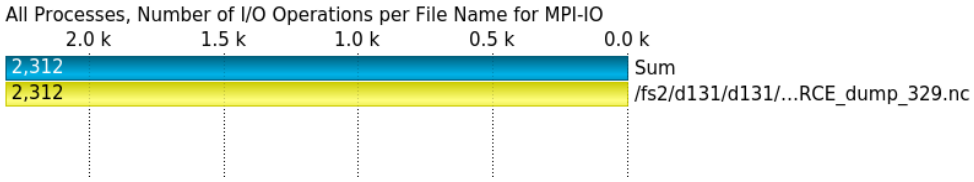


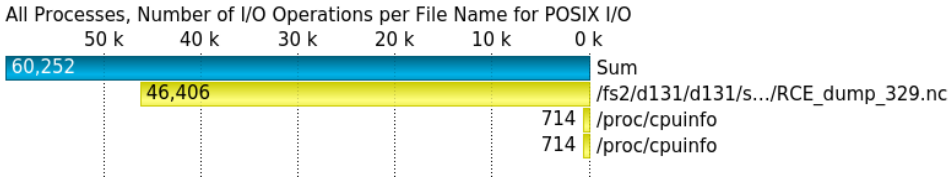
Figure 5.13: This figure depicts function statistics of the MONC experiment run. The application uses NetCDF to realize I/O data transfers. The internal I/O software stack of NetCDF utilizes MPI I/O and POSIX I/O. According to the statistic about time spent in function groups, I/O operations contribute only little to the overall execution time.



(a) Number of executed I/O operations (NetCDF)



(b) Number of executed I/O operations (MPI I/O)



(c) Number of executed I/O operations (POSIX I/O)

Figure 5.14: The figure illustrates I/O statistics of the MONC experiment run. The number of operations differs on the individual layers (a) NetCDF, (b) MPI I/O, and (c) POSIX I/O of the I/O software stack.

Figure 5.14 depicts three I/O summary charts for NetCDF (5.14a), MPI I/O (5.14b), and POSIX I/O (5.14c) to facilitate a detailed I/O performance investigation. All three layers of the I/O software stack utilize the same `RCE_dump_329.nc` file. The number of accesses to this file increases while traversing the NetCDF, MPI I/O, and POSIX I/O layers. This statistic reflects how each library abstracts functionality in order to hide complex operations. Furthermore, the figure shows that POSIX I/O also utilizes additional files. Further analyses aim on identifying the origin of these file accesses.

The recorded event logs clearly reflect the different behavior of simulation and I/O server processes. This assists users in identifying I/O server processes and focusing subsequent analysis on this subset of processes. Figure 5.15 depicts the I/O timeline (top) and the process summary (bottom) of *Pthread thread 7:0*, i.e., Thread 7 of the I/O server process Rank 0. The I/O timeline displays the performed type of I/O operations (Read (orange), Write (yellow), Open (blue), Close (green)) on the x-axis and the accessed files as well as associated handles on the y-axis. If an I/O library (e.g., NetCDF) utilizes another I/O library, the individual handles of each library are attached to each other, as represented in a tree-like hierarchy to the left of the upper chart. The hierarchical information between I/O resources recorded in the event logs are the foundation for this kind of visualization. The top chart in Figure 5.15 depicts all handles used to access the NetCDF file `RCE_dump_329.nc`. Thereby, NetCDF internally utilizes MPI I/O (see handle *MPI-IO #0*) which in turn performs POSIX I/O operations (see *POSIX I/O #20*) on *RCE_dump_329.nc*. This view also shows that MPI I/O opens (blue bars) *maps*-files from the */proc* file system using the ISO-C API. Each I/O server process reads (red boxes) its *maps*-file before transferring simulation data to the NetCDF file.

The bottom chart of Figure 5.15 depicts the process timeline for Thread 7 of Rank 0 and provides details about the calling context of I/O operations in this time slice. For instance, the execution of `nc_put_vara_double` (bottom chart, blue bar, call stack level 7) creates an I/O write event of the *NetCDF #0* handle (top chart). This operation in turn calls `MPI_File_Write_at_all` (bottom chart, orange bar, call stack level 8) which generates the I/O write event of the *MPI-IO #0* handle

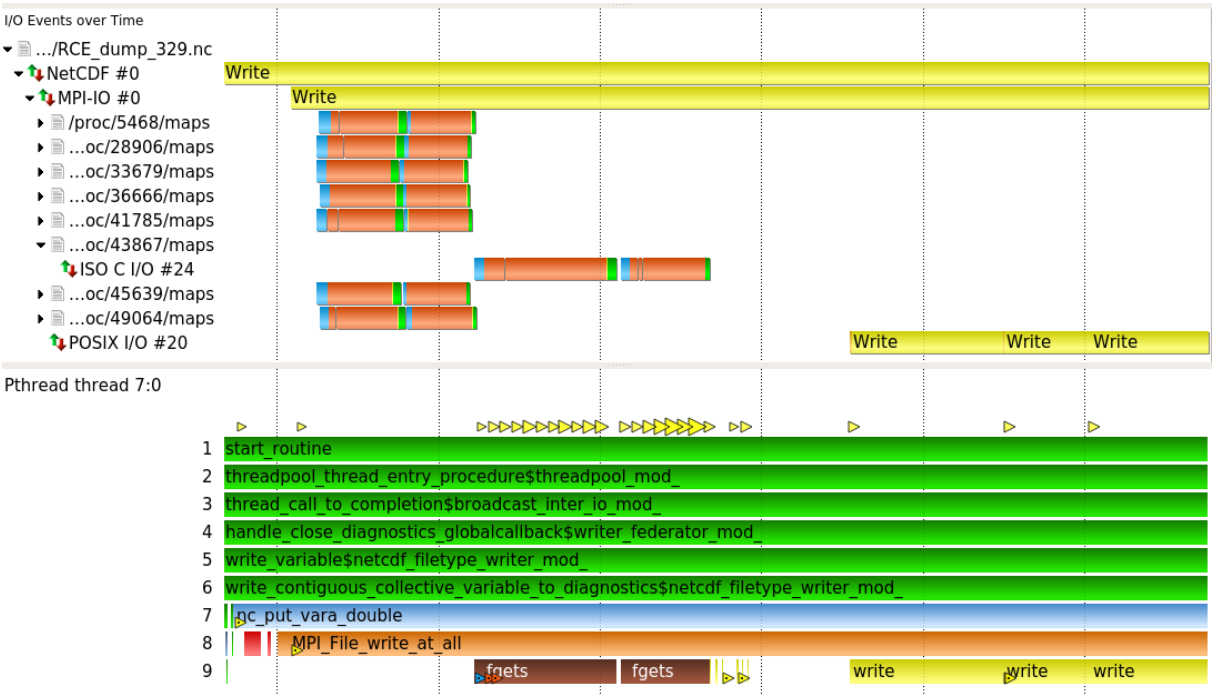


Figure 5.15: The I/O timeline (top) shows individual I/O operations of *Pthread thread 7:0* (Thread 7 of the I/O server process Rank 0) on specific files. The tree view on the right hand side reflects the recorded hierarchical relations between files. The process summary (bottom) depicts the corresponding call stack and illustrates interactions of the individual layers of the I/O software stack. For instance, the NetCDF routine `nc_put_vara_double` (blue bar, call stack level 7) internally calls the MPI I/O routine `MPI_File_write_at_all` (orange bar, call stack level 8) which utilizes POSIX I/O functions (brown and yellow bars, call stack level 9).

(top chart). Call stack level 9 in the bottom chart shows internal details of this collective MPI I/O routine. It depicts the `fgets` call to access the *maps*-file (*/proc/43867/maps*). The `write` calls to store the final data correspond to the write events of the *POSIX I/O #20* handle (top chart). Interestingly, NetCDF executes MPI communication operations (bottom chart, red bars, call stack level 8) within the `nc_put_vara_double` routine. In this time interval, these operations have a short duration compared with the `MPI_File_Write_at_all` routine and do not impede performance. However, in a different scenario, these functions may lead to a communication bottleneck or undesirable wait states.

Current analysis focused on the behavior of one I/O server process. The next analysis steps widen the scope and compare different I/O server processes. Figure 5.16 shows the process timelines of I/O server Rank 0/Thread 7 (top) and Rank 14/Thread 2 (bottom). Both servers call identical functions with similar durations until call stack level 9. On this level, both servers perform ISO-C I/O operations (brown bars) at the beginning of the `MPI_File_Write_at_all` routine. Afterwards, one server process (top) executes POSIX I/O `write` functions (yellow bars). It seems that only one I/O server process accesses the *RCE_dump_329.nc* file through the collective I/O operation. The collective operation appears to synchronize all processes (causing waiting time) except process Rank 0/Thread 7, that performs the actual I/O operations. Figure 5.17 depicts the number of system calls within MPI I/O routines. The event log provides information about the system tree topology and allows a visualization of values aggregated per compute node. Node `nid01713` performs the most system calls within MPI I/O routines. This confirms, that only one I/O server transfers data to the `RCE_dump_329.nc` file. This effect is similar to the collective buffering as presented in the NAS BT-IO case study.

This performance study proves the effectiveness of the client-server-architecture implemented by the MONC simulator. This approach aggregates I/O requests of the application and thereby takes load off the

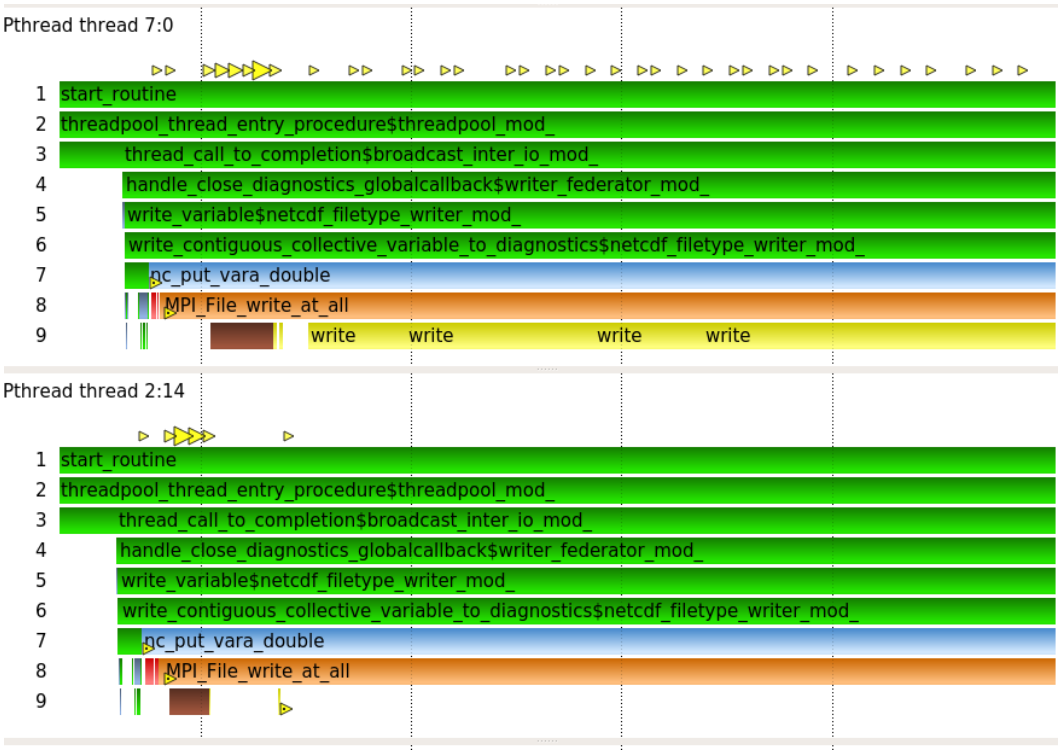


Figure 5.16: This figure compares the call stacks of two different MONC I/O server processes. Both servers show an identical call stack except level 9. Only the I/O server process shown in the top chart issues calls to the POSIX I/O `write` routine (yellow bars).

I/O subsystem. However, the study also shows that the number of simulation and I/O server processes needs to be fine-tuned to achieve optimal performance. The holistic performance analysis approach presented in this work enables users to investigate the computation and communication behavior of the application as well as internals of the I/O software stack. Based on these insights, users can calibrate the ratio between simulation and I/O server processes.

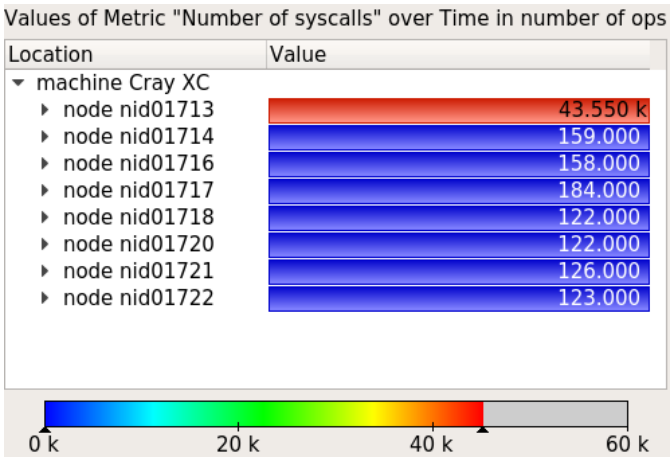


Figure 5.17: This figure depicts the number of system calls within MPI I/O routines and maps the visualization to the system tree topology. Node `nid01713` performs the most system calls within MPI I/O routines. This confirms, that only one I/O server transfers data to the `RCE_dump_329.nc` file.

5.4 Top-Down Performance Analysis of Scientific Workflows

This section presents three case studies evaluating the top-down analysis methodology of scientific workflows. The first case study illustrating a *Synthetic Workflow* demonstrates the analysis of complex workflow structures. The second case study, *GATK/Cromwell*, focuses on the integration of this methodology in workflow management systems. Finally, the third case study, *GROMACS*, describes the process of tackling performance problems from general overview to their root cause.

5.4.1 Demonstration of the Top-Down Performance Analysis Process

This first case study demonstrates the analysis process with a synthetic workflow. The analysis starts with an overview of the entire workflow and proceeds with an investigation of individual jobs and job steps. Figure 5.18 depicts the Workflow Visualizer showing the general structure of the observed workflow.

The synthetic workflow consists of five individual jobs. In this example, the workflow starts with *Job 42* which contains four job steps. After the completion of *Job 42*, a sequence of parallel jobs starts. *Job 43* and *Job 45* run in parallel to *Job 44*. Solid arrows indicate dependencies of *Job 43* and *Job 44* on *Job 42*. Each one of the *Jobs 43*, *44*, and *45* consists of three individual job steps. As Figure 5.18 shows, the workflow does not define a job dependency between *Job 43* and *Job 45*. Consequently, the job scheduler (Slurm) data contains no dependency information for these two jobs and no solid arrow is drawn in Figure 5.18. However, even in the absence of Slurm dependency information the proposed methodology for the analysis of scientific workflows is able to expose job dependencies. Based on the

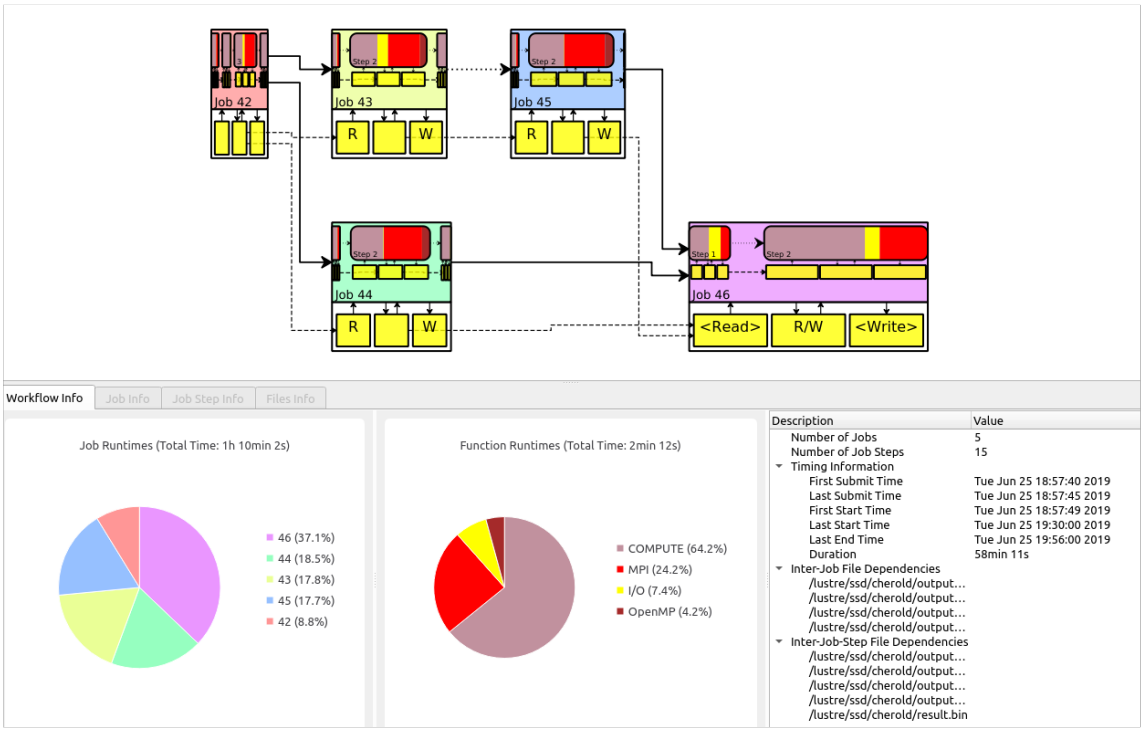


Figure 5.18: *Timeline* visualization of the synthetic workflow example. The top chart depicts the general structure of the observed workflow. Solid arrows represent scheduling dependencies between jobs, whereas the dotted arrow illustrates a data dependency between *Job 43* and *Job 45*. Pie charts at the bottom give a statistical overview of the entire workflow. The chart at the bottom left depicts the share of individual jobs in the overall runtime. The chart at the bottom center differentiates the runtime into function groups. The chart at the bottom right presents general information about the entire workflow.

I/O activity recording, the methodology tracks data dependencies and provides additional provenance data. Figure 5.18 illustrates this data dependency as a dotted arrow between *Job 43* and *Job 45*. In this example, the data dependency between *Job 43* and *Job 45* enforces a sequential execution of both jobs, resulting in inefficient resource utilization. While *Job 43* and *Job 44* run concurrently, *Job 45* represents a load imbalance in the upper execution path. Finally, after the completion of *Job 44* and *Job 45*, the workflow ends with *Job 46* including two job steps. Pie charts at the bottom of Figure 5.18 give a statistical overview of the entire workflow. The chart at the bottom left depicts the share of individual jobs in the overall runtime. According to this chart, *Job 46* (purple color) dominates and accounts for about 37% of the overall runtime. The chart at the bottom center differentiates the runtime into function groups and indicates that the workflow is compute intensive (pastel purple color).

The analysis continues with a detailed discussion of individual jobs and job steps. The investigation starts with the first job of the workflow (*Job 42*). The coloring of the Job Step Boxes of *Job 42* indicates that almost all job steps are compute intensive (pastel purple color). Only *Job Step 3* of *Job 42* spends about half of its runtime in MPI functions (red color). However, as the runtime of *Job 42* is relatively small in comparison with the total runtime of the workflow, this issue is not prioritized and the investigation proceeds. *Job 43*, *Job 44*, and *Job 45* are the next elements of the examined workflow. Each of these jobs consists of three job steps. In all three jobs, the second job step dominates the runtime and shows a significant MPI runtime share (red color). Consequently, these job steps are candidates for further analysis and optimization. Figure 5.19 depicts *Job 45* in detail. The *Job Step Info* chart indicates that *Job Step 2* spends almost half of its runtime in MPI. Further analysis of the event logs is recommended to identify the root cause of this excessive MPI time. Probably performance critical communication patterns or load imbalances lead to increased MPI wait times. In addition, *Job 46* is also a potential candidate for detailed performance analysis, as this job exhibits the longest runtime of all jobs. Especially, *Job Step 2* of *Job 46* dominates the runtime. Any optimization that reduces the runtime of this job will directly result in a reduced runtime of the overall workflow.

5.4.2 Integration of Performance Data Recording into Workflow Management Systems

Workflow management systems coordinate the execution of scientific workflows. Therefore, in order to be widely used, the proposed methodology for the analysis of scientific workflows has to integrate easily into workflow management systems. This case study demonstrates such an integration using the workflow management system Cromwell [121] as an example. Cromwell supports a Slurm back-end transparently. Additionally, many implementations of real-world workflows are available for Cromwell. These workflows use the Genome Analysis Toolkit (GATK) [121] Java application.

A parameterized extra wrapper layer to Cromwell's default Slurm back-end provider realizes the integration into Cromwell. This script wraps each job submission (`sbatch` call) and makes the approach independent of a specific workflow example. The wrapper script creates a custom environment that controls the Score-P instrumentation [34] of each job and invokes OTF2-Profile to process any generated job trace. An automatic epilogue step for all Slurm jobs invokes JobLog to collect actual wall time and other accounting information of the corresponding job. With this modification, users only have to provide two additional parameters to their workflows. First, users have to provide the location of the Score-P wrapper script. The Score-P wrapper script manages the instrumentation configurations. Because the actual instrumentation configuration is workflow specific, users should set up an appropriate configuration and provide it as an input to the workflow. Second, users have to specify the executable to use. Score-P's Java bytecode instrumentation relies on a Java wrapper (`scorep-bc-java`) that processes the Java executable. This processing step cannot simply be part of the Score-P wrapper script, as not all workflows in Cromwell are Java-based. Therefore, users need a parameter to selectively enable or disable Java bytecode instrumentation.

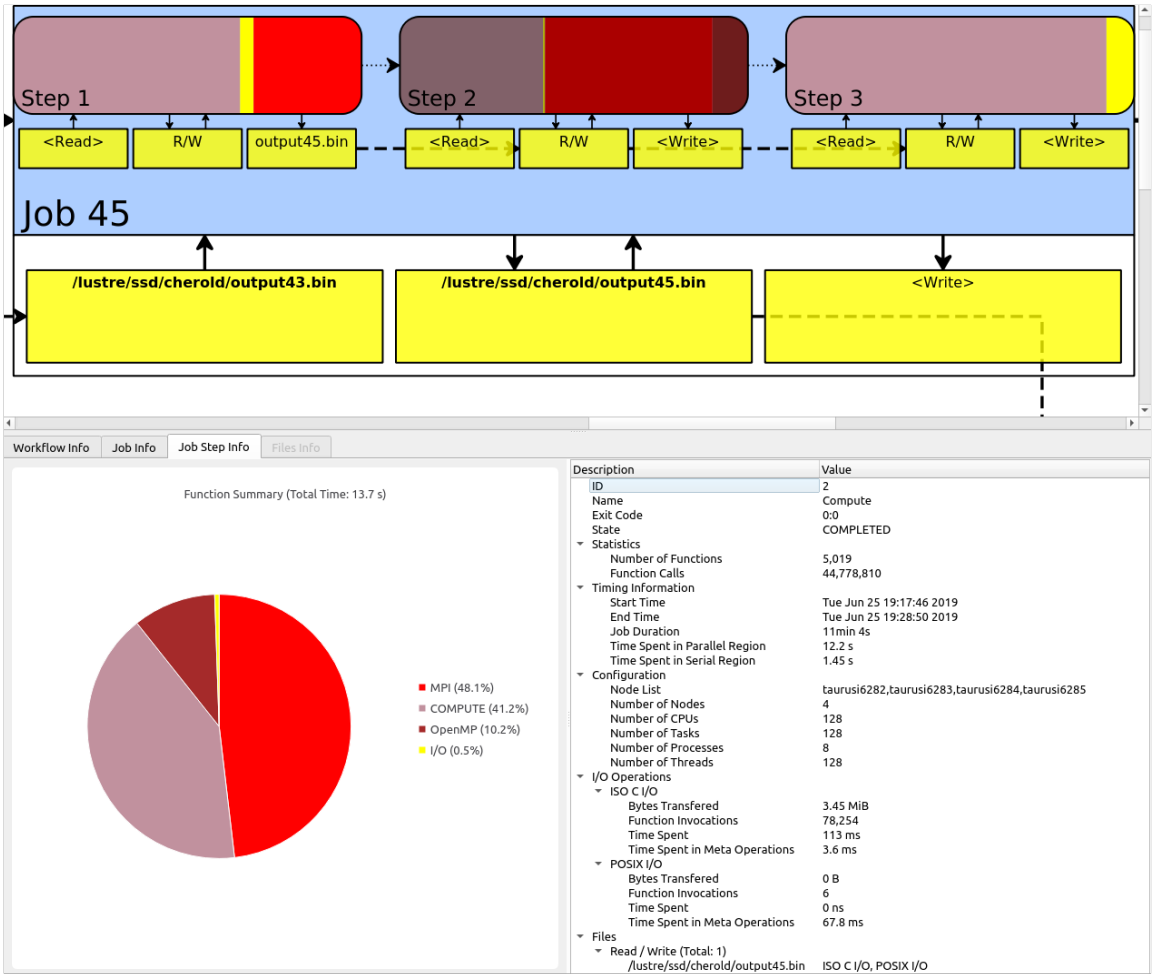


Figure 5.19: Performance profiles of *Job 45* in Figure 5.18. *Job Step 2* spends significant time in MPI routines (red color).

This test case presents the Joint Calling Genotypes (JCG) workflow from GATK. It applies user-level Java instrumentation to record the performance behavior of the (JCG) workflow. Figure 5.20 shows the results of the measurement.

The JCG workflow consists of two phases. In the first phase, JCG starts with a scatter/gather operation. This initial phase consists of a configurable number of parallel jobs. As Figure 5.20 shows, the example of this case study starts with three jobs (*Job 11943156 – 11943158*). All three jobs have well-balanced individual durations, indicating an adequate load balancing. However, not all jobs are scheduled concurrently. This suggests optimization potential for either end-to-end time or resource usage efficiency.

In the second phase, JCG performs a two-step analysis of results generated by the previous jobs. *Job 11943160* and *Job 11943161* represent this subsequent post-processing in Figure 5.20. Jobs of the second phase depend on the jobs of the first phase, as jobs of the first phase generate the input data for the post-processing step. Cromwell manages workflow dependencies by its own internal polling mechanism. Hence, dependencies are not controlled via Slurm, and thus, are not included in available Slurm information. Nevertheless, the approach presented in this thesis reveals these dependencies based on information obtained from I/O recording. Figure 5.20 illustrates these dependencies as dotted arrows between individual jobs. This allows the Workflow Visualizer tool to present the internal workflow structure for the users.

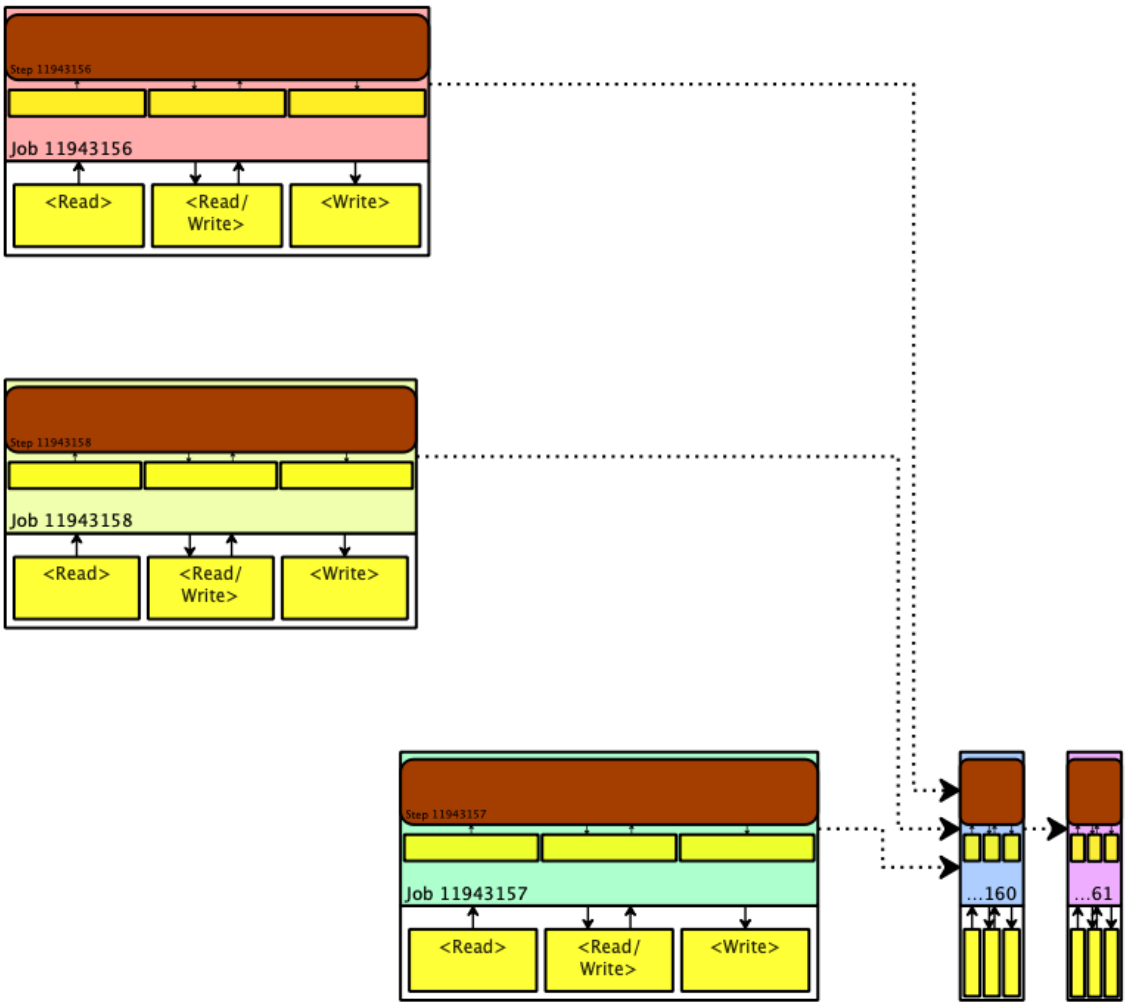
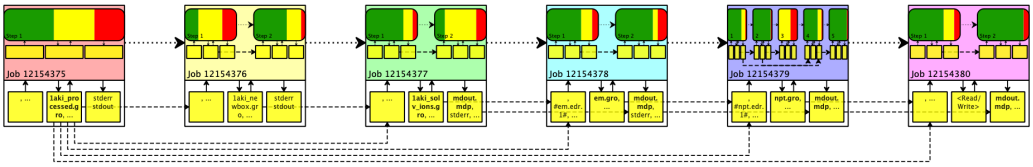


Figure 5.20: *Timeline* visualization of the Joint Calling Genotypes (JCG) workflow from GATK. The JCG workflow consists of two phases. In the first phase, three jobs run in parallel. In the second phase, two jobs run consecutively and perform a post-processing of the data generated by the previous jobs. Dotted arrows indicate data dependencies between individual jobs.

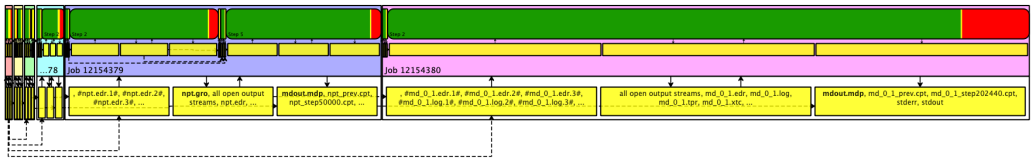
5.4.3 Optimization of a GROMACS Workflow

The software suite GROMACS (Groningen MAchine for Chemical Simulation) [116] is an open-source package for molecular dynamics. Its main purpose is the simulation of biochemical molecules like proteins, lipids, and nucleic acids. This case study examines the "Lysozyme in Water" example [57] and illustrates the effects of applied optimization.

Figure 5.21 gives an overview of the workflow topology. The observed workflow consists of six jobs in one pipeline. Figure 5.21a illustrates the topology as a dependency graph. The first three jobs prepare the simulation system. These jobs perform their work in a completely serial fashion. The last three jobs are parallelized and perform most of the simulation work. Figure 5.21a shows that the job steps of *Job 12154375* to *Job 12154377* spend a considerable amount of their time in I/O (yellow color) and MPI (red color) routines, suggesting these jobs as good candidates for optimization. However, Figure 5.21b provides additional insight by scaling the Job Boxes according to their actual runtime. This visualization indicates that the runtime share of *Job 12154375* to *Job 12154377* and also *Job 12154378* is negligible. Instead, *Job 12154379* and *Job 12154380* appear to be promising candidates for further performance analysis. Especially *Job Step 2* and *Job Step 5* of *Job 12154379* and *Job Step 2* of *Job 12154380* dominate



(a) Dependency Graph visualization



(b) Timeline visualization

Figure 5.21: The figure shows the GROMACS "Lysozyme in Water" workflow. The (a) Dependency Graph depicts the general structure of six jobs executed in one pipeline. The (b) Timeline visualization scales the Job Boxes according to their runtimes and shows that the fifth and sixth job dominate the overall runtime of the workflow.

the overall runtime. All mentioned job steps are compute intensive (green color). Further analysis of *Job Step 2* of *Job 12154380* reveals a rather high MPI runtime share of about 10 %. The next step of the analysis inspects the recorded event logs to identify the reason of this performance issue.

The upper graph of Figure 5.22 shows the Vampir visualization of the corresponding event logs. The analysis exposes a load imbalance in function *fft5d_execute*. Work is not evenly distributed across all processes. On some processes (e.g., *Master thread:11*) the execution of *fft5d_execute* takes significantly longer than on other processes. As a result, processes with less workload start their MPI communication early but have to wait for processes with longer running *fft5d_execute* instances. This increases the overall time spent in MPI for *Job 12154380*. The lower graph of Figure 5.22 illustrates the event log of an additional measurement with an improved load balancing setup. This figure shows that this setup distributes the workload evenly across all processes. As a result, the MPI time share of the job reduces from 10% to 6%. Additionally, the reduction of the MPI time share implicates a considerable reduction of the time required to execute an iteration.

5.4.4 Performance Discussion

This section discusses the overhead of the proposed workflow analysis approach. The discussion distinguishes three potential sources of overhead: application instrumentation with Score-P, recording of job scheduling information with JobLog, and data post-processing with OTF2-Profile.

Section 5.2 examines the execution time and memory overhead of the I/O instrumentation. In addition to I/O activities, recording event logs of real-world applications includes further event types such as function entries/exits or message transfers. Section 5.2 also suggests options to refine the set of recorded events and thereby control the overhead. Experiments conducted by Knüpfer et al. [53] applied a reasonable filter setup to real-world applications and reported a runtime overhead below 4%. For the workflow analysis experiments as shown in this work the overhead due to application instrumentation reads as follows. The overhead of the GROMACS case study was below 10%. In comparison with compile-time or library-wrapping based instrumentation, the Java bytecode instrumentation induces a higher overhead. Consequently, the overhead of the GATK case study was still quite significant (approximately 3× runtime), even with filtering. The event logs of the presented case studies demanded at most 10 GiB for the entire GATK workflow.

Recording of job scheduling information with JobLog is less critical. It records statistics in the range of $1 - 10$ kiB. Furthermore, the proposed approach queries scheduling information after the application has finished and thereby does not affect the application execution.

Figure 5.22: Event log visualization of GROMACS *Job 12154380*, *Job Step 2*. The upper graph (blue background) shows an event sequence of the GROMACS run with disabled load balancing. The lower graph (white background) illustrates the corresponding event sequence of a run with enabled load balancing. The event sequences depict GROMACS functions (green) and MPI communication (red). Invocations of the function *fft5d_execute* are highlighted in yellow. As the lower graph shows, improved load balancing results in a considerable reduction of the time required to execute an iteration.

Data post-processing with OTF2-Profile is also noncritical as it runs as a post-mortem task and can be executed on a local machine. In the analyses presented in this section, the post-processing shows a fixed setup time of $5 - 15$ seconds and runs for at most 10% of the original application runtime.

As this discussion shows, the proposed top-down analysis approach is able to record valuable performance data. Time-critical aspects of this approach provide options to minimize the induced overhead. The presented toolset allows users to analyze real-world scientific workflows, identify potential bottlenecks within the workflows, and determine optimization opportunities.

6 Conclusion and Outlook

This chapter gives a résumé of this thesis and presents an outlook on future work.

6.1 Summary and Conclusion

I/O operations are a performance critical aspect of data intensive applications. Consequently, in addition to traditional facets, e.g., computation and communication, performance analysis and tuning of such applications has to address I/O related issues. This dissertation proposes a novel methodology for recording calls to I/O libraries on multiple layers of the software stack. In contrast to current approaches, this methodology explicitly correlates operations between multiple levels of the complex I/O software stack. This enhanced level of detail in the recorded performance data is essential for understanding the overall I/O behavior of applications.

The first contribution of this work is a methodology to investigate application I/O behavior in detail. The key contribution is a scheme that captures I/O activities on multiple layers of the I/O software stack and correlates these activities across all layers explicitly. Recording these hierarchical relations between I/O operations arising from different layers of the I/O software stack enables analyses of the interplay between these layers. Therefore, this thesis introduces concepts to store information about I/O activities in event logs including the hierarchical relations between them. This information is the foundation for the following contributions.

The second contribution of this work defines I/O access patterns observable in the recorded event logs of parallel scientific applications. These patterns guide software developers in analyzing the I/O behavior of their applications. In addition, the hierarchical relations of I/O activities recorded in the event logs allow users to evaluate interactions between layers of the complex I/O software stack, investigate internal optimizations applied by high-level I/O libraries, and validate their effectiveness in combination with low-level I/O libraries.

The third contribution of this work consists of a top-down performance analysis methodology for scientific workflows. This methodology widens the scope of analysis from an individual application to coordinated sequences of multiple applications. It guides users through the comprehensive performance analysis of complex workflows by starting with a general overview, assisting in the identification of optimization candidates, and presenting detailed performance data as needed. The methodology for workflow analysis uses the multi-layer I/O recording capability to reveal I/O dependencies between individual jobs and job steps, respectively.

The fourth contribution of this work is the implementation of all proposed methodologies. This thesis describes the enhancement of the established performance monitoring infrastructure Score-P by sophisticated I/O recording features. It presents the implementation of I/O definition and event records in the OTF2 trace format to persistently store captured information. Consequently, this thesis enhanced the mentioned performance tools significantly. Furthermore, this work provides the implementation of tools and their cooperation to realize a top-down performance analysis methodology for scientific workflows.

In summary, the proposed methodologies are viable approaches for the performance analysis of parallel applications. Software developers can now identify I/O bottlenecks and determine their root causes inside a complex I/O software and hardware stack. The presented contributions complement existing functionality and realize a holistic performance analysis for parallel scientific applications including computation, communication, and I/O operations. The techniques have been applied to real-world applications. This work shows that these novel methods expose patterns in I/O activities across multiple layers that otherwise would have been hard or even impossible to find. Introduced features such as

the recording of POSIX I/O and MPI I/O operations are included in the official Score-P releases since version 6.0. Therefore, contributions of this work are publicly available as open-source. The implementation of NetCDF and HDF5 recording is in its final stage and will be released in an upcoming Score-P release. Results of this thesis were already used in several research projects. The project "Next Generation I/O for the Exascale" (NEXTGenIO) [74] within the European Union's Horizon 2020 Research and Innovation programme explored non-volatile memory technology to bridge the gap between memory and storage. Based on the I/O recording methodology proposed in this thesis, the project extended performance analysis tools such as Score-P and Vampir to record and visualize non-volatile memory accesses of observed applications. The project "Advanced Data Placement via Ad-hoc File Systems at Extreme Scales" (ADA-FS) [2] within the German Priority Programme 1648 Software for Exascale Computing (SPPEXA) investigated the use of distributed ad-hoc overlay file systems to improve I/O performance for highly-parallel applications. Therefore, this project used the work presented in this thesis to investigate access patterns of applications to and within files.

6.2 Outlook

This thesis focuses on performance analysis of file I/O operations. However, it can be easily extended to monitor I/O operations on sockets. This use case would only require new definitions for representing sockets as an I/O resource besides files and directories as shown in this thesis. Furthermore, current event records can be augmented with information about failed operations. This would extend their scope of application from performance analysis to debugging and correctness. In addition, future work could include support for I/O activities on object storage architectures. The extensible design of the definition and event records as well as the measurement system facilitates the implementation of novel features and the adaption to upcoming system architectures.

Bibliography

[1] LMT3 - Lustre Monitoring Tool. `https://github.com/LLNL/lmt`. Last accessed on 2020-09-30.

[2] ADA-FS – Advanced Data Placement via Ad-hoc File Systems at Extreme Scales. `http://ada-fs.github.io/`. Last accessed on 2020-09-30.

[3] Laksono Adhianto, Shisagnee Banerjee, Mike W. Fagan, Mark Krentel, Gabriel Marin, John M. Mellor-Crummey, and Nathan R. Tallent. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 2010.

[4] Akshat Aranya, Charles P. Wright, and Erez Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 129–145, Berkeley, CA, USA, 2004. USENIX Association.

[5] Arm. Arm DDT – Debugger for C, C++ and Fortran Threaded and Parallel Code. `https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt`. Last accessed on 2020-09-30.

[6] Arm. Arm MAP – Low-Overhead Profiling to Optimize C, C++, Fortran and F90 Codes. `https://www.arm.com/products/development-tools/server-and-hpc/forge/map`. Last accessed on 2020-09-30.

[7] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. Stack Trace Analysis for Large Scale Debugging. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, March 2007.

[8] Malcolm Atkinson, Sandra Gesing, Johan Montagnat, and Ian Taylor. Scientific workflows: Past, present and future. *Future Generation Computer Systems*, 75:216 – 227, 2017.

[9] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 53–62, May 2016.

[10] Babak Behzad, Hoang-Vu Dang, Farah Hariri, Weizhe Zhang, and Marc Snir. Automatic Generation of I/O Kernels for HPC Applications. In *2014 9th Parallel Data Storage Workshop*, pages 31–36, November 2014.

[11] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Peter Kogge, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO)*, 15, January 2008.

[12] blktrace(8) - Linux man page. `https://linux.die.net/man/8/blktrace`. Last accessed on 2020-09-30.

[13] Ronny Brendel, Bert Wesarg, Ronny Tschüter, Matthias Weber, Thomas Ilsche, and Sebastian Oeste. Generic Library Interception for Improved Performance Measurement and Insight. In Abhinav Bhatele, David Boehme, Joshua A. Levine, Allen D. Malony, and Martin Schulz, editors, *Programming and Performance Visualization Tools*, pages 21–37. Springer International Publishing, 2019.

[14] Barcelona Supercomputing Center (BSC). Extrae instrumentation package. `http://tools.bsc.es/extrae`. Last accessed on 2020-09-30.

[15] Barcelona Supercomputing Center (BSC). Paraver: a flexible performance analysis tool. `http://tools.bsc.es/paraver`. Last accessed on 2020-09-30.

[16] Antoine Capra, Patrick Carribault, Jean-Baptiste Besnard, Allen D. Malony, Marc Pérache, and Julien Jaeger. User Co-scheduling for MPI+OpenMP Applications Using OpenMP Semantics. In Bronis R. de Supinski, Stephen L. Olivier, Christian Terboven, Barbara M. Chapman, and Matthias S. Müller, editors, *Scaling OpenMP for Exascale Performance and Portability*, pages 203–216. Springer International Publishing, 2017.

[17] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and Improving Computational Science Storage Access Through Continuous Characterization. *ACM Transactions on Storage*, 7(3):8:1–8:26, October 2011.

[18] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 Characterization of Petascale I/O Workloads. In *Proceedings of 2009 Workshop on Interfaces and Architectures for Scientific Data Storage*, pages 1–10, 2009.

[19] Philip Carns, Walter Ligon, Robert Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the Extreme Linux Track: 4th Annual Linux Showcase and Conference*, volume 4, page 11, October 2000.

[20] Mohamad Chaarawi, Edgar Gabriel, Rainer Keller, Richard L. Graham, George Bosilca, and Jack J. Dongarra. OMPIO: A Modular Software Architecture for MPI I/O. In Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 81–89, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[21] collectd – The system statistics collection daemon. `https://collectd.org/`. Last accessed on 2020-09-30.

[22] Collectl. `http://collectl.sourceforge.net/`. Last accessed on 2020-09-30.

[23] IBM Corporation. IBM Parallel Performance Toolkit Version 2.4 documentation. `https://www.ibm.com/support/knowledgecenter/SSFK5S_2.4/doc.pdf`. Last accessed on 2020-09-30.

[24] Intel Corporation. Intel Inspector. `https://software.intel.com/en-us/inspector`. Last accessed on 2020-09-30.

[25] Intel Corporation. Intel Trace Analyzer and Collector. `https://software.intel.com/content/www/us/en/develop/tools/trace-analyzer.html`. Last accessed on 2020-09-30.

[26] Intel Corporation. Intel VTune Profiler. `https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html`. Last accessed on 2020-09-30.

[27] Anthony Danalis. MPI and Compiler Technology: A Love-Hate Relationship. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 12–13, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[28] Hewlett Packard Enterprise Development. Cray Performance Measurement and Analysis Tools User Guide. https://pubs.cray.com/bundle/Cray_Performance_Measurement_and_Analysis_Tools_User_Guide/page/CrayPat.html. Last accessed on 2020-09-30.

[29] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Gerd Eugen Wolf. Open Trace Format 2 - The Next Generation of Scalable Trace Formats and Support Libraries. In *Applications, Tools and Techniques on the Road to Exascale Computing: Proc. of the 14th biennial ParCo conference*, volume 22 of *Advances in Parallel Computing*, pages 481–490, 2012.

[30] Oak Ridge Leadership Computing Facility. Summit - Oak Ridge National Laboratory's 200 petaflop supercomputer. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/. Last accessed on 2020-09-30.

[31] Oak Ridge Leadership Computing Facility. Titan - Advancing the Era of Accelerated Computing. https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/. Last accessed on 2020-09-30.

[32] Matt Fleming. A thorough introduction to eBPF. https://lwn.net/Articles/740157/. Last accessed on 2020-09-30.

[33] Free Software Foundation. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/. Last accessed on 2020-09-30.

[34] Jan Frenzel, Kim Feldhoff, Rene Jäkel, and Ralph Müller-Pfefferkorn. Tracing of Multi-Threaded Java Applications in Score-P Using Bytecode Instrumentation. In *ARCS Workshop 2018; 31th International Conference on Architecture of Computing Systems*, pages 1–8, April 2018.

[35] ftrace(1) - Linux man page. https://linux.die.net/man/1/ftrace. Last accessed on 2020-09-30.

[36] Markus Geimer, Felix Wolf, Brian Wylie, and Bernd Mohr. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel computing*, 35:375 – 388, 2009.

[37] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[38] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. October 2018.

[39] JobLog. https://github.com/harryherold/JobLog. Last accessed on 2020-09-30.

[40] OTF/OTF2 Profile. https://github.com/score-p/otf2_cli_profile. Last accessed on 2020-09-30.

[41] Pilar Gomez-Sanchez, Sandra Mendez, Dolores Rexachs, and Emilio Luque. PIOM-PX: A Framework for Modeling the I/O Behavior of Parallel Scientific Applications. In Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf, editors, *High Performance Computing*, pages 160–173. Springer International Publishing, 2017.

[42] Max Grossman, Jun Shirako, and Vivek Sarkar. OpenMP as a High-Level Specification Language for Parallelism. In Naoya Maruyama, Bronis R. de Supinski, and Mohamed Wahib, editors, *OpenMP: Memory, Devices, and Tasks*, pages 141–155. Springer International Publishing, 2016.

[43] Robert J. Hall. Call Path Profiling. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 296–306, New York, NY, USA, 1992. ACM.

[44] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. MPI runtime error detection with MUST: Advances in deadlock detection. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, November 2012.

[45] IBM Corporation. IBM Spectrum Scale Version 5.0.3 - Concepts, Planning, and Installation Guide. `https://www.ibm.com/support/knowledgecenter/STXKQY_5.0.3/com.ibm.spectrum.scale.v5r03.doc/pdf/scale_ins.pdf?view=kc`, 2019. Last accessed on 2020-09-30.

[46] sysstat - System performance tools for the Linux operating system. `https://github.com/sysstat/sysstat`. Last accessed on 2020-09-30.

[47] iotop. `http://guichaz.free.fr/iotop/`. Last accessed on 2020-09-30.

[48] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley Professional Computing. John Wiley & Sons, April 1991.

[49] Franziska Kasielke and Ronny Tschüter. From Mathematical Model to Parallel Execution to Performance Improvement: Introducing Students to a Workflow for Scientific Computing. In Gabriele Mencagli, Dora B. Heras, Valeria Cardellini, Emiliano Casalicchio, Emmanuel Jeannot, Felix Wolf, Antonio Salis, Claudio Schifanella, Ravi Reddy Manumachu, Laura Ricci, Marco Beccuti, Laura Antonelli, José Daniel Garcia Sanchez, and Stephen L. Scott, editors, *Euro-Par 2018: Parallel Processing Workshops*, pages 211–221. Springer International Publishing, 2019.

[50] Seong J. Kim, Seung W. Son, Wei-keng Liao, Mahmut Kandemir, Rajeev Thakur, and Alok Choudhary. IOPin: Runtime Profiling of Parallel I/O in HPC Systems. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 18–23, November 2012.

[51] Andreas Knüpfer. *Advanced Memory Data Structures for Scalable Event Trace Analysis*. PhD thesis, December 2008.

[52] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[53] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[54] Julian M. Kunkel, Michaela Zimmer, Nathanael Hübbe, Alvaro Aguilera, Holger Mickler, Xuan Wang, Andriy Chut, Thomas Bönisch, Jakob Lüttgau, Roman Michel, and Johann Weging. The SIOX Architecture – Coupling Automatic Monitoring and Optimization of Parallel I/O. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing*, pages 245–260. Springer International Publishing, 2014.

[55] Sandia National Laboratories. Structural Simulation Toolkit (SST) DUMPI Trace Library. https://github.com/sstsimulator/sst-dumpi. Last accessed on 2020-09-30.

[56] Argonne National Laboratory. Jumpshot: Performance Visualization Tool. https://www.anl.gov/mcs/jumpshot-performance-visualization-tool. Last accessed on 2020-09-30.

[57] Justin Lemkul. GROMACS Tutorial "Lysozyme In Water". http://www.mdtutorials.com/gmx/complex/01_pdb2gmx.html. Last accessed on 2020-09-30.

[58] Glenn K. Lockwood, Shane Snyder, George Brown, Kevin Harms, Philip Carns, and Nicholas J. Wright. TOKIO on ClusterStor: Connecting Standard Tools to Enable Holistic I/O Performance Analysis. In *Proceedings of the 2018 Cray User Group*, May.

[59] Glenn K. Lockwood, Wucherl Yoo, Suren Byna, Nicholas J. Wright, Shane Snyder, Kevin Harms, Zachary Nault, and Philip Carns. UMAMI: A Recipe for Generating Meaningful Metrics Through Holistic I/O Performance Analysis. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, PDSW-DISCS '17, pages 55–60, New York, NY, USA, 2017. ACM.

[60] ltrace(1) - Linux man page. https://linux.die.net/man/1/ltrace. Last accessed on 2020-09-30.

[61] LTTng: an open source tracing framework for Linux. https://lttng.org/. Last accessed on 2020-09-30.

[62] The Lustre llapi library. http://doc.lustre.org/lustre_manual.xhtml#settinglustreproperties. Last accessed on 2020-09-30.

[63] Introduction to Lustre Architecture - Lustre systems and network administration. http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf, October 2017. Last accessed on 2020-09-30.

[64] The Lustre file system. http://lustre.org/, February 2018. Last accessed on 2020-09-30.

[65] Huong Luu, Babak Behzad, Ruth Aydt, and Marianne Winslett. A Multi-Level Approach for Understanding I/O Activity in HPC Applications. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–5, September 2013.

[66] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. A Multiplatform Study of I/O Behavior on Petascale Supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 33–44, New York, NY, USA, 2015. ACM.

[67] John M. Mellor-Crummey, Laksono Adhianto, Mike W. Fagan, Mark Krentel, and Nathan R. Tallent. HPCTOOLKIT User's Manual, Version 2020.07. http://hpctoolkit.org/manual/HPCToolkit-users-manual.pdf, July 2020. Last accessed on 2020-09-30.

[68] Sandra Mendez, Sebastian Lührs, Dominic Sloan-Murphy, Andrew Turner, and Volker Weinberg. Best Practice Guide - Parallel I/O. https://prace-ri.eu/training-support/best-practice-guides/best-practice-guide-parallel-io/, July 2019. Last accessed on 2020-09-30.

[69] Michael P. Mesnier, Matthew Wachs, Raja R. Simbasivan, Julio Lopez, James Hendricks, Gregory R. Ganger, and David O'Hallaron. //TRACE: Parallel trace replay with approximate causal events. In *In Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 153–167, 2007.

[70] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114, April 1965.

[71] MPI Forum. The standardization forum for the Message Passing Interface (MPI). `http://www.mpi-forum.org/`. Last accessed on 2020-09-30.

[72] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.1. `https://www.mpi-forum.org/docs/mpi-3.1/`, June 2015. Last accessed on 2020-09-30.

[73] LLC Nagios Enterprises. Nagios - The Industry Standard In IT Infrastructure Monitoring. `https://www.nagios.com/`. Last accessed on 2020-09-30.

[74] NEXTGenIO – Next Generation I/O for the Exascale. `http://www.nextgenio.eu/`. Last accessed on 2020-09-30.

[75] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. ScalaTrace: Scalable Compression and Replay of Communication Traces for High Performance Computing. *Journal of Parallel and Distributed Computing*, 69(8), May 2008.

[76] U.S. Department of Energy. U.S. Department of Energy and Intel to deliver first exascale supercomputer. `https://www.anl.gov/article/us-department-of-energy-and-intel-to-deliver-first-exascale-supercomputer`. Last accessed on 2020-09-30.

[77] Chinese Acadamey of Sciences. China's Exascale Supercomputer Operational by 2020. `http://english.cas.cn/newsroom/archive/china_archive/cn2016/201606/t20160616_164450.shtml`. Last accessed on 2020-09-30.

[78] OpenACC. `http://openacc.org/`. Last accessed on 2020-09-30.

[79] The OpenACC Application Programming Interface Version 3.0. `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf`, November 2019. Last accessed on 2020-09-30.

[80] OpenMP. The OpenMP API specification for parallel programming. `http://openmp.org/`. Last accessed on 2020-09-30.

[81] OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 5.0 November 2018. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`, November 2018. Last accessed on 2020-09-30.

[82] ParaProf - User's Manual. `https://www.cs.uoregon.edu/research/tau/docs/newguide/bk01pt02.html`. Last accessed on 2020-09-30.

[83] Scott Parker, John Mellor-Crummey, Dong H. Ahn, Heike Jagode, Holger Brunst, Sameer Shende, Allen D. Malony, David DelSignore, Ronny Tschüter, Ralph Castain, Kevin Harms, Philip Carns, Ray Loy, and Kalyan Kumaran. *Chapter 2: Performance Analysis and Debugging Tools at Scale*. Chapman & Hall/CRC Computational Science. CRC Press, 2017.

[84] perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`. Last accessed on 2020-09-30.

[85] Antoine Petitet, Clint Whaley, Jack Dongarra, and Andy Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/. Last accessed on 2020-09-30.

[86] proc - process information pseudo-filesystem. http://man7.org/linux/man-pages/man5/proc.5.html. Last accessed on 2020-09-30.

[87] Robert Ross, Lee Ward, Philip Carns, Gary Grider, Scott Klasky, Quincey Koziol, Glenn K. Lockwood, Kathryn Mohror, Bradley Settlemyer, and Matthew Wolf. Storage Systems and I/O: Organizing, Storing, and Accessing Data for Scientific Discovery. May 2019.

[88] sar(1) - Linux man page. https://linux.die.net/man/1/sar. Last accessed on 2020-09-30.

[89] Pavel Saviankou, Michael Knobloch, Anke Visser, and Bernd Mohr. Cube v4: From Performance Report Explorer to Performance Analysis Tool. *Procedia Computer Science*, 51:1343 – 1352, 2015. International Conference On Computational Science, ICCS 2015.

[90] Robert Schöne, Ronny Tschüter, Thomas Ilsche, Joseph Schuchart, Daniel Hackenberg, and Wolfgang E. Nagel. Extending the Functionality of Score-P Through Plugins: Interfaces and Use Cases. In Christoph Niethammer, José Gracia, Tobias Hilbrich, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2016*, pages 59–82. Springer International Publishing, 2017.

[91] Score-P GitHub Repository. https://github.com/score-p/. Last accessed on 2020-09-30.

[92] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, May 2006.

[93] Ben Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, pages 336–343, September 1996.

[94] Anna Sikora, Eduardo César, Isaías Comprés, and Michael Gerndt. Autotuning of MPI Applications Using PTF. In *Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications*, SEM4HPC '16, page 31–38, New York, NY, USA, 2016. Association for Computing Machinery.

[95] Benjamin Skuse. The third pillar. *Physics World*, 32(3):40–43, March 2019.

[96] Perforce Software. TotalView HPC Debugging Software. https://totalview.io/products/totalview. Last accessed on 2020-09-30.

[97] strace(1) - Linux man page. https://linux.die.net/man/1/strace. Last accessed on 2020-09-30.

[98] sysdig. https://sysdig.com/. Last accessed on 2020-09-30.

[99] SystemTap. https://sourceware.org/systemtap/. Last accessed on 2020-09-30.

[100] Taurus HPC-Cluster at Technische Universität Dresden. https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/SystemTaurus. Last accessed on 2020-09-30.

[101] The Clang Team. ThreadSanitizer. https://clang.llvm.org/docs/ThreadSanitizer.html. Last accessed on 2020-09-30.

[102] The LLDB Team. LLDB. https://lldb.llvm.org/. Last accessed on 2020-09-30.

[103] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting Performance Data with PAPI-C. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[104] Rajeev Thakur and Alok Choudhary. An Extended Two-phase Method for Accessing Sections of Out-of-core Arrays. *Scientific Programming*, 5(4):301–317, December 1996.

[105] Rajeev Thakur, Alok Choudhary, Rajesh Bordawekar, Sachin More, and Sivaramakrishna Kuditipudi. Passion: Optimized I/O for Parallel Applications. *Computer*, 29(6):70–78, June 1996.

[106] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, February 1999.

[107] Rajeev Thakur, William Gropp, and Ewing Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, IOPADS '99, pages 23–32, New York, NY, USA, 1999. ACM.

[108] The HDF Group. Hierarchical Data Format, Version 5. https://www.hdfgroup.org/solutions/hdf5/. Last accessed on 2020-09-30.

[109] TOP500.org. Highlights of the 51st TOP500 List. https://www.top500.org/static/media/uploads/top500_ppt_201806.pdf. Last accessed on 2020-09-30.

[110] TOP500.org. The TOP 500 Supercomputers List. https://www.top500.org. Last accessed on 2020-09-30.

[111] Ronny Tschüter, Christian Herold, Bert Wesarg, and Matthias Weber. A Methodology for Performance Analysis of Applications Using Multi-layer I/O. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 16–30. Springer International Publishing, 2018.

[112] Ronny Tschüter, Christian Herold, William Williams, Maximilian Knespel, and Matthias Weber. A Top-Down Performance Analysis Methodology for Workflows: Tracking Performance Issues from Overview to Individual Operations. In *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 1–10, November 2019.

[113] Ronny Tschüter, Johannes Ziegenbalg, Bert Wesarg, Matthias Weber, Christian Herold, Sebastian Döbel, and Ronny Brendel. An LLVM Instrumentation Plug-in for Score-P. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM-HPC'17, pages 2:1–2:8, New York, NY, USA, 2017. ACM.

[114] Unidata. Network Common Data Form (NetCDF) [software]. https://doi.org/10.5065/D6H70CW6. Last accessed on 2020-09-30.

[115] Lawrence Berkeley National Laboratory University of California. Integrated Performance Monitoring for High Performance Computing. https://github.com/nerscadmin/IPM/. Last accessed on 2020-09-30.

[116] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E. Mark, and Herman J. C. Berendsen. GROMACS: fast, flexible, and free. *Journal of computational chemistry*, 26(16):1701–1718, 2005.

[117] Rob F. Van der Wijngaart. Charon Message-Passing Toolkit for Scientific Computations. In Mateo Valero, Viktor K. Prasanna, and Sriram Vajapeyam, editors, *High Performance Computing — HiPC 2000*, pages 3–14, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[118] Jeffrey S. Vetter and Michael O. McCracken. Statistical Scalability Analysis of Communication Operations in Distributed Applications. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 123–132, New York, NY, USA, 2001. ACM.

[119] Virtual Institute for I/O. `https://www.vi4io.org/start`, February 2018. Last accessed on 2020-09-30.

[120] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. Scalable I/O Tracing and Analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 26–31, New York, NY, USA, 2009. ACM.

[121] Kate Voss, Geraldine Van Der Auwera, and Jeff Gentry. Full-stack genomics pipelining with GATK4 + WDL + Cromwell [version 1; not peer reviewed]. *ISCB Comm J*, 6(1381), 2017.

[122] Michael Wagner. *Concepts for In-memory Event Tracing: Runtime Event Reduction with Hierarchical Memory Buffers*. PhD thesis, March 2015.

[123] Parkson Wong, Rob F. Van der Wijngaart, and Bryan Biegel. NAS Parallel Benchmarks I/O Version 2.4. December 2002.

[124] Steven A. Wright, Simon D. Hammond, John Pennycook, Robert F. Bird, Andy Herdman, Iain Miller, Ash Vadgama, Abhir Bhalerao, and Stephen A. Jarvis. Parallel File System Analysis Through Application I/O Tracing. *Computer Journal*, 56(2):141–155, February 2013.

[125] Xing Wu, Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. Probabilistic Communication and I/O Tracing with Deterministic Replay at Scale. In *2011 International Conference on Parallel Processing*, pages 196–205, September 2011.

[126] Cong Xu, Suren Byna, Vishwanath Venkatesan, Robert Sisneros, Omkar Kulkarni, Mohamad Chaarawi, and Kalyana Chadalavada. LIOProf: Exposing Lustre File System Behavior for I/O Middleware. *Cray User Group Conference*, 2016.

[127] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

List of Figures

List of Tables

A Appendix

A.1 Definition Records

A.1.1 Definition of I/O Resources

IoRegularFile		
String	name	Name of the file. References a String definition.
SystemTreeNode	scope	Defines the physical scope of this IoRegularFile in the system tree. E.g., two IoRegularFile definitions with the same name but different scope values are physically different, thus I/O operations through IoHandles do not operate on the same file. References a SystemTreeNode definition.

IoDirectory		
String	name	Name of the directory. References a String definition.
SystemTreeNode	scope	Defines the physical scope of this IoDirectory in the system tree. E.g., two IoDirectory definitions with the same name but different scope values are physically different, thus I/O operations through IoHandles do not operate on the same directory. References a SystemTreeNode definition.

IoFileProperty		
IoFile	ioFile	Parent IoRegularFile definition to which this one is a supplementary definition. References a IoRegularFile definition.
String	property-name	Property name. References a String definition.
Type	type	The type of this property.
AttributeValue	value	The value of this property.

A.1.2 Definition of I/O Handles

IoParadigm		
String	identification	The I/O paradigm identification. This should be used programmatically to identify a specific I/O paradigm. For a human-readable name use the name attribute. If this identification matches one of the known I/O paradigms listed in the OTF2 documentation Known OTF2 I/O paradigms, then the attributes of this definition must match those specified there. References a String definition.
String	name	The name of the I/O paradigm. This should be presented to humans as the name of this I/O paradigm. References a String definition.
IoParadigmClass	class	The class of this I/O paradigm.
IoParadigmFlag	flags	Boolean properties of this I/O paradigm.
uint8_t	numberOfProperties	Number of properties.
IoParadigmProperty	properties [numberOfProperties]	The property.
Type	types [numberOfProperties]	The type of this property. Must match with the defined type of the property.
AttributeValue	values [numberOfProperties]	The value of this property.

IoHandle		
String	name	Handle name. References a String definition.
IoFile	file	File identifier. References a IoRegularFile, or a IoDirectory definition.
IoParadigm	paradigm	The I/O paradigm. References a IoParadigm definition.
IoHandleFlag	flags	Special characteristics of this handle.
Comm	group	Scope of the file handle. This scope defines which process can access this file via this handle and also defines the collective context for this handle. References a Comm definition.
IoHandle	parent	Parent, in case this I/O handle was created and operated by an higher- level I/O paradigm. References a IoHandle definition.

IoPreCreatedHandleState		
IoHandle	handle	Parent IoHandle definition to which this one is a supplementary definition. References a IoHandle definition.
IoAccessMode	mode	The access mode of the pre-created IoHandle.
IoStatusFlag	fags	The status flags of the pre-created IoHandle.

A.2 Event Records

A.2.1 Metadata Operations

IoCreateHandle		
Location	location	The location where this event happened.
TimeStamp	timestamp	The time when this event happened.
IoHandle	handle	A previously inactive I/O handle which will be activated by this record. References a IoHandle definition.
IoAccessMode	mode	Determines which I/O operations can be applied to this I/O handle (e.g., read-only, write-only, read-write).
IoCreationFlag	creationFlags	Requested I/O handle creation flags (e.g., create, exclusive, etc.).
IoStatusFlag	statusFlags	I/O handle status flags which will be associated with the handle attribute (e.g., append, create, close-on-exec, async, etc).

IoDestroyHandle		
Location	location	The location where this event happened.
TimeStamp	timestamp	The time when this event happened.
IoHandle	handle	An active I/O handle which will be inactivated by this records. References a IoHandle definition.

IoDuplicateHandle		
Location	location	The location where this event happened.
TimeStamp	timestamp	The time when this event happened.
IoHandle	oldHandle	An active I/O handle. References a IoHandle definition.
IoHandle	newHandle	A previously inactive I/O handle which will be activated by this record. References a IoHandle definition.
IoStatusFlag	statusFlags	The status flag for the new I/O handle newHandle. No status flags will be inherited from the I/O handle oldHandle.

IoSeek		
Location	location	The location where this event happened.
TimeStamp	timestamp	The time when this event happened.
IoHandle	handle	An active I/O handle. References a IoHandle definition.
int64_t	offsetRequest	Requested offset.
IoSeekOption	whence	Position inside the file from where offsetRequest should be applied (e.g., absolute from the start or end, relative to the current position).
uint64_t	offsetResult	Resulting offset, e.g., within the file relative to the beginning of the file.

IoChangeStatusFlags		
Location	location	The location where this event happened.
TimeStamp	timestamp	The time when this event happened.
IoHandle	handle	An active I/O handle. References a IoHandle definition.
IoStatusFlag	statusFlags	Set flags (e.g., close-on-exec, append, etc.).

IoDeleteFile		
Location	location	The location where this event happened.
TimeStamp	timestamp	The time when this event happened.
IoParadigm	paradigm	The I/O paradigm which induced the deletion. References a IoParadigm definition.
IoFile	file	File identifier. References a IoRegularFile, or a IoDirectory definition.

A.2.2 Data Transfer Operations

IoOperationBegin		
Location	location	The location where this event happened.
TimeStamp	timestamp	The time when this event happened.
IoHandle	handle	An active I/O handle. References a IoHandle definition.
IoOperationMode	mode	Mode of an I/O handle operation (e.g., read or write).
IoOperationFlag	operationFlags	Special semantic of this operation.
uint64_t	bytesRequest	equested bytes to write/read.
uint64_t	matchingId	Identifier used to correlate associated event records of an I/O operation. This identifier is unique for the referenced IoHandle.

IoOperationComplete		
Location	location	The location where this event happened.
TimeStamp	timestamp	The time when this event happened.
IoHandle	handle	An active I/O handle. References a IoHandle definition.
uint64_t	bytesResult	Number of actual transferred bytes.
uint64_t	matchingId	Identifier used to correlate associated event records of an I/O operation. This identifier is unique for the referenced IoHandle.

IoOperation(Issued|Test|Cancelled)		
Location	location	The location where this event happened.
TimeStamp	timestamp	The time when this event happened.
IoHandle	handle	An active I/O handle. References a IoHandle definition.
uint64_t	matchingId	Identifier used to correlate associated event records of an I/O operation. This identifier is unique for the referenced IoHandle.

A.2.3 Locking Operations

Io(Acquire|Try|Release)Lock		
LocationRef	location	The location where this event happened.
TimeStamp	timestamp	The time when this event happened.
IoHandle	handle	An active I/O handle. References a IoHandle definition.
LockType	type	Type of the lock.

Glossary

API Application Programming Interface.

CFD Computational Fluid Dynamics.

CPU Central Processing Unit.

GPU Graphics Processing Unit.

HPC High Performance Computing.

MPI Message Passing Interface.

SPMD Single Program Multiple Data.