CAPIO: Cross-Application Programmable I/O

by

Alberto Riccardo Martinelli

Ph.D. Thesis

University of Turin
Computer Science Department
Cycle XXXVI

Supervisor: Prof. Marco Aldinucci
Co-Supervisor: Prof. Massimo Torquati

To my family.

"The real cycle you're working on is a cycle called yourself. The machine that appears to be 'out there' and the person that appears to be 'in here' are not two separate things. They grow toward Quality or fall away from Quality together."

- Robert M. Pirsig, *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values*

Abstract

As the volume of digital data available for analysis and simulation continues to surge, the realm of I/O-intensive HPC workflows is poised for rapid expansion. This trend, however, threatens to widen the performance gap between computing, memory, and storage technologies, underscoring the criticality of our research.

A workflow describes a sequence of application steps and their control/-data dependencies. In the HPC context, data dependencies are usually streamlined by storing data files in the distributed storage system in producer steps and read out by consumer steps afterward. However, with the increasing gap between the computation speed and the speed of the central storage system and the ever-increasing amount of data produced in scientific applications, sharing files between workflow steps through the file system is costly. Burst buffers and user-space ad-hoc file systems have been proposed to increase the available I/O bandwidth and reduce the contention on the shared file system by leveraging fast local storage. However, workflow steps need to be executed orderly according to the data dependency graph, and it could be difficult, or even impossible, to exploit pipeline parallelism among them. In-situ workflows were proposed to mitigate or avoid the cost of profoundly relying on the file system as communication media and enable temporal parallelism between workflow steps. In in-situ workflows, multiple steps are executed concurrently; data dependencies are accomplished by sidestepping the file system through explicit coordination mechanisms among workflow steps.

However, it is not always desirable, or even possible (e.g., legacy code), to rewrite or patch existing workflows to enable in-situ orchestration by using specific frameworks. For this reason, we propose CAPIO (Cross-Application Programmable I/O), a middleware capable of transparently injecting I/O streaming capabilities into file-based workflows, improving the

computation-I/O overlap without modifying the business code. The contribution is twofold: at design time, a new I/O coordination language allows users to annotate workflow data dependencies with synchronization semantics; at run time, a user-space software layer automatically turns a batch execution into a streaming execution according to the semantics expressed in the configuration file. CAPIO has been tested on synthetic benchmarks simulating typical I/O workflow patterns and three real-world workflows. The results show how CAPIO provides performance improvements in data-intensive workflows that extensively use the file system as a communication medium.

Looking ahead, tools like CAPIO could reshape HPC workflow orchestration strategies. For instance, they might enable the distribution of pre-and post-processing I/O-intensive phases across computation phases, fostering better overlap between steps by reducing applications' peak I/O demands. This prospect underscores the transformative potential of our research.

Acknowledgements

I would like to express my gratitude to my supervisor, Professor Marco Aldinucci, and my co-supervisor, Professor Massimo Torquati, for their guidance, support, and valuable suggestions throughout my Ph.D. journey. I also extend my thanks to my senior colleagues, Iacopo Colonelli and Barbara Cantalupo, for their assistance.

The help provided by all the aforementioned persons, along with the countless hours of discussions, have been of fundamental importance in shaping the outcomes of this work.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The fundamental shift in computational dynamics, marked by the cessation of Moore's Law [1] in recent years, has underscored the indispensability of parallel and distributed computing for addressing complex computational challenges. Leveraging a high number of nodes, along with the multiple processors and the numerous cores across processors, is pivotal for tackling forthcoming scientific and engineering challenges.

High-performance computing (HPC) systems comprise many machines connected to a high-speed network and equipped with multicore CPUs, often supplemented with accelerators like FPGA and GPUs. These systems, characterized by their substantial cost, are designed for concurrent use by multiple users. Users rely on job schedulers, such as Slurm [2], to execute their programs across a specified number of nodes. Given the heterogeneous nature of hardware within HPC systems, using programming languages and libraries that offer portability across different architectures is imperative.

Thanks to distributed file systems, HPC systems endeavor to optimize parallel access to data, thereby surpassing conventional file systems in efficiency. Notwithstanding advancements in processor speeds, data access rates on disks continue to lag, perpetuating I/O as a bottleneck in applications with intensive I/O demands.

The gap between computational speed and I/O speed is poised to widen, posing challenges for data-intensive workflows on HPC platforms, spanning scientific workflows [3, 4] to AI applications [5, 6, 7]. Sharing files between workflow steps through the central file system has become a costly operation [8]. Various tools and technologies have emerged to address these issues. For instance, Burst Buffers [9], fast secondary memories like SSDs in com-

putation nodes, offer a local disk solution to enhance workflow performance beyond reliance on distributed file systems. Ad-hoc filesystems [10], temporary user-space filesystems with a lifespan tied to workflow execution, aim to overcome distributed filesystem limitations by relaxing POSIX semantics and leveraging burst buffers or the shared memory of the nodes.

From the application perspective, different APIs have been developed to enhance I/O parallelism, including MPI I/O [11], Dataspaces [12], Damaris [13], ADIOS2 [14], and more. A shift in paradigm from classic batch workflows, where each program executes after the termination of programs producing its input, to in-situ workflows [15, 16, 17], where workflow steps run concurrently and communicate data in a streaming fashion, has become common. Adapting legacy code for concurrent execution with synchronization mechanisms, especially with low-level APIs like POSIX, can be challenging. ADIOS2 addresses this issue by providing a high-level API, but even this can be complex when modifying legacy code.

To simplify this process, we introduce a new tool called CAPIO (Cross-Application Programmable I/O) [18]. CAPIO transforms a batch workflow into a streaming workflow without requiring code modifications. The user only needs to write, using the CAPIO coordination language, a configuration file describing the workflow's data dependencies (which programs read/write which files) and the required synchronization using the CAPIO coordination language. The CAPIO runtime captures application system calls and enforces synchronization between workflow applications described in the configuration file, allowing users to execute a workflow initially designed for batch execution concurrently by merely writing a configuration file instead of modifying the I/O code.

Overlapping computation and I/O can significantly improve the performance of entire workflows. This will be demonstrated in the experimental chapter (6) using real-world workflows. For example, the 1000 Genomes workflow consists of several steps where each step must wait for the previous step to finish to ensure that all the input data has been produced. By using middleware such as ADIOS2 and CAPIO, we can overlap the execution of these steps; as soon as a token of input data is produced by one step, it can be consumed by the following step. While ADIOS2 (and POSIX) rely on a computational approach, CAPIO relies on a declarative approach. Another common situation where overlapping I/O and computation is useful is when a scientific simulation is followed by an analysis or visualization step. Instead of waiting for the scientific simulation to complete, the analysis step

can be executed concurrently with the simulation, starting to analyze a piece of information as soon as it is produced. In this thesis, a real-world workflow named the Weather Forecast Workflow, which falls into this category, will be discussed. In the original workflow, the simulation was followed by the analysis and visualization tasks, and the user could only start seeing the weather predictions after the simulation was completed. Instead, with CAPIO, as soon as the simulation produces data for an hour, that output can be visualized by the user.

1.1 Coordination languages

Coordination languages express and coordinate different computations using communication and synchronization primitives. In endogenous coordination languages, coordination is expressed within the computation, while in exogenous coordination, computation and coordination are separated [19]. Coordination is expressed through communication primitives such as send and receive for message passing or through synchronization mechanisms such as semaphores, monitors, and barriers.

Coordination languages allow the user to communicate and synchronize during computations, while others define coordination rules triggered by specific actions or events. These languages offer syntactic constructs to facilitate synchronization and data communication. While some languages abstract communication and synchronization details, relying on predefined patterns, others take a more low-level approach, providing a variety of mechanisms for synchronization and communication.

Linda [20] is among the most influential coordination languages, abstracting synchronization through shared tuple spaces for communication between processes. In recent years, higher-level languages based on the skeleton approach [21, 22] have gained popularity. These languages allow users to program using parallel patterns without requiring synchronization and communication details. Examples of such languages include P3L [23], FastFlow [24], GRPPI [25] SkePU [26] and more [27, 28, 29]. Coordination languages are not only used for parallel computation but also for coordinating input/output operations. For instance, Common Workflow Language (CWL) [30] specifies the data dependencies among workflow modules and is utilized by workflow management systems. The CAPIO coordination language is the first language that transforms a batch workflow into an in-situ workflow using a

declarative approach. Chapter 2 delves into coordination languages in greater detail.

1.2 Research questions

This work aims to answer the following research questions:

- **Q1:** Given a workflow composed of steps that communicate using files and are normally executed sequentially based on data dependencies, is it possible to execute these steps concurrently, enabling streaming communication without modifying the code, while ensuring the correctness of the results by externally imposing synchronization mechanisms?
- **Q2:** If the answer to the previous question is yes, how does the user express the synchronization mechanism for streaming communication without modifying the original code?
- **Q3:** Is it possible to implement a runtime in the current HPC hardware and software stack to provide the features required by the previous questions?

The next section provides an overview of the contributions of this thesis that address these research questions.

1.3 Contributions and results

The contribution of this work is twofold:

1. An I/O coordination language named CAPIO language enables users to transform a conventional workflow (where producer-consumer steps are executed one after the other) into an in-situ workflow (where steps are executed concurrently), exploiting streaming communication.
2. The development of a runtime system that leverages information from the coordination language to facilitate transparent streaming communication.

The CAPIO coordination language delineates two fundamental aspects:

Firing-Rule. Determining when the content of a file can be read.

Commit-Rule. Establishing when a file is deemed complete.

With this information, a process can ascertain when it is permissible to access a file, even amid its ongoing generation. Consequently, the CAPIO language empowers users to specify the data dependencies within workflows, encompassing identifying files utilized and generated by individual applications. To facilitate streaming communication between specific applications for designated files, users must also define the commit and firing rules for those files.

The CAPIO language currently uses a JSON configuration file, which is then conveyed to the CAPIO runtime. The CAPIO runtime guarantees the accurate execution of the workflow by enforcing the streaming semantics as stipulated by the commit and firing rules.

The *Firing-Rule* defines when the content of a file can be read. The file's content can be read when it will not be modified because the CAPIO language ensures that an application reads the same data as if it were executed after all the previous application steps in the workflow have concluded. Therefore, the CAPIO language defines two Firing-Rules: "no-update" and "update". The "no-update" Firing-Rule ensures that when a process writes data into a file section, that section will not be modified. With this rule, a process can start reading a section of a file as soon as it is written by another application, ensuring maximum streaming communication. The "update" Firing-Rule expresses the situation when a process can modify the content of a file multiple times. In this case, a process can read the file only when it is considered completed (committed), i.e., when no more data is written.

The *Commit-Rule* defines when a file is considered committed. This information is necessary to determine when a process can stop reading a file because it receives an end-of-stream (EOS) signal. The CAPIO language supports three Commit-Rules: 1) on-termination, 2) on-close, and 3) on-file. The Commit-Rule "on-termination" for a file specifies that the file is considered committed when all the writers of the file have terminated. It is necessary when the user lacks knowledge about when no more data will be added to a file by its writers, and therefore, the reader process has to wait for their termination to be sure that the file is completed. The Commit-Rule "on-close" ensures that a file is committed when a process closes it, which is useful when a writer process opens a file, writes in it, and then closes it without writing to it again. In this case, a reader can consider the file completed after the writer closes it and stops reading it. It is also possible

to define how often a file must be closed to be considered complete, which is necessary when a process opens and closes a file multiple times while writing to it. The last Commit-Rule is the "on-file" rule, which specifies that a file is considered committed when another file is committed.

Both the Commit-Rule and the Firing-Rule must be defined for a file, and every combination is allowed. For example, if the Commit-Rule for a file is "on-close" and the Firing-Rule is "no-update", it means that a reader process can start reading the file as soon as data is written into it and will receive an end-of-stream (EOS) signal when it reaches the end of it, and the writer process has closed it. In this way, we achieve streaming communication even if the workflow was written to execute the reader and the writer processes sequentially. With the Commit-Rule "on-termination" and Firing-Rule "no-update", a process can start reading a section of the file as soon as it is written. At the same time, it will receive the EOS signal only after all writers have concluded their execution. There are cases where streaming communication is not possible due to the nature of the problem. For example, if a file is updated multiple times, and the user does not know when it can be considered committed because it is opened and closed an unknown number of times, no streaming communication is possible. In this case, the correct rules are the Commit-Rule "on-termination" and the Firing-Rule "update". These rules are guaranteed to provide correct results since they impose batch execution. With the definition of the Commit-Rule and the Firing-Rule, it is possible to concurrently run two programs that communicate using files without the need to implement synchronization mechanisms inside the applications' code.

The Commit-Rules for directories work slightly differently, and it will be discussed in Chapter 4, where the syntax and semantics of the CAPIO language are presented in detail.

The second contribution of this work is a runtime (referred to as CAPIO runtime) that can respect the synchronization semantics expressed with the CAPIO Language. The CAPIO runtime proposed in this work leverages the configuration file written with the CAPIO language to execute concurrently the steps of a workflow that communicates using files and that was designed to be executed in a batch fashion; that is, a step can be executed when all the steps that produce its input files are terminated. The CAPIO runtime only needs the configuration file to inject streaming capabilities into a batch workflow; the code must not be modified. The transparent injection of streaming capabilities is achieved by the CAPIO runtime, which captures the system

calls of the workflow's applications. When a process wants to read a file, the POSIX syscalls such as "open", "read", and "write" are intercepted, and if the file is one that CAPIO must handle, then CAPIO executes its code; otherwise, it passes the control to the kernel.

The CAPIO runtime is composed of a shared library called libcapio_posix, which must be linked by setting the LD_PRELOAD environment variable to the path where the shared library resides and by CAPIO servers that run on the nodes where the workflow is executed. There is one CAPIO server per node.

The CAPIO shared library captures system calls of the target application and interacts with the local CAPIO server to respond to requests made by the application with the syscall. For example, if a program executes a "read" syscall on a file managed by CAPIO, the CAPIO shared library intercepts this syscall and requests the required data from the local CAPIO server. The CAPIO server checks the Commit-Rule and the Firing-Rule for that file, and if they are satisfied, it retrieves the requested data. The data could be stored in the local shared memory or another node's shared memory. If the latter is true, the CAPIO server will contact the other CAPIO server on the node where the data is stored to retrieve it. This process is entirely transparent to the application. The user only needs to write the configuration file, deploy the CAPIO servers on the nodes where the workflow will be executed, and link (without the need to recompile) the CAPIO shared library. The CAPIO servers communicate using MPI, and the CAPIO shared library communicates with the local CAPIO server using a circular buffer in shared memory. The files are stored in the main memory of the nodes where the CAPIO servers are running. Using the CAPIO language, it is possible to decide to save files on the file system at the end of the workflow run. It is helpful to avoid copying temporary files to the file system used only during the workflow execution. It is possible to extend CAPIO to use multiple backends, such as ADIOS2 and the file system.

It is worth noting that the runtime can respect the Commit-Rules and the Firing-Rules because it captures the system calls. When a process closes a file, it can trigger the Commit-Rule "on-close" if the user has specified so. After this close operation, the CAPIO servers know the file is considered completed when another process wants to read it. The same thing applies to the semantics "on-termination", which can be triggered (if the user chooses this semantics for a specific file) by the POSIX system calls *exit* and *exit_group*.

To showcase the effectiveness of this solution, we conducted a perfor-

mance comparison between CAPIO, POSIX, and ADIOS2 on two clusters using two different distributed filesystems (Lustre and BeeGFS). Our evaluation included a set of synthetic benchmarks replicating common I/O patterns in HPC workflows, as well as three workflows: a bioinformatics workflow for computing overlaps in human genome mutations called "1000 Genomes", a DAG-based weather-forecasting workflow actively employed for weather prediction in Italy, and a MapReduce workflow that replicates the common MapReduce pattern. Furthermore, we re-implemented the synthetic benchmarks using ADIOS2 and conducted an extensive performance comparison among ADIOS2, POSIX I/O, and CAPIO.

We compared workflows that used POSIX without streaming communication with CAPIO to demonstrate the performance gain achieved by running a batch workflow enabling streaming communication through the CAPIO language. The results of these tests show, in most cases, a reduction in execution time using CAPIO ranging from 20% to 30%. In some extreme cases, the difference can be even greater. For example, in a synthetic benchmark with 10,000 files of 10MB each, the speed-up is equal to 8 ($\sim$ 630s vs. $\sim$ 75s). Another aspect to consider, in addition to total execution time, is when the user can start to see the output produced by a workflow. In simulations, the user can start to visualize the output while the simulation is still running and producing output. In the weather forecasting workflow used in the tests, the user starts to see the forecast images after the simulation has ended, and the post-processing application begins reading its binary output files. With CAPIO, the simulation and the post-processing application can run concurrently. As soon as the simulation produces an output file, it is read by the post-processing step, allowing the user to start seeing the forecast earlier. In this workflow, the simulation writes an output file every 2/3 minutes. Without CAPIO, the user would have to wait for the simulation to terminate before being able to start viewing the forecast by the hour. In contrast, with CAPIO, it can be viewed every 2/3 minutes during the simulation.

The comparison also included ADIOS2 to understand the performance gap between introducing streaming capabilities by modifying the code using a library (ADIOS2) and doing it with the CAPIO language without changing the original code. Adding layers of abstraction can introduce overhead, but it simplifies the use of some tools. The tests were used to determine if manually implementing synchronization for streaming communication improves performance better than CAPIO with more effort. The impact of the manual implementation of synchronization mechanism in the code could be further

explored by directly using POSIX and implementing streaming communication on files using file-locking mechanisms. The decision to invest more effort for better performance is a trade-off that the user must decide. The results of the synthetic benchmarks show that CAPIO and ADIOS2 are almost always faster than POSIX with batch execution. CAPIO and ADIOS2 exhibit similar times; sometimes, one is faster than the other and vice versa. The results could depend on numerous factors such as hardware configuration, the backend used for ADIOS2, the parameters configuration of ADIOS2, or the current state of the current implementation of the CAPIO runtime.

1.4 Organization

This thesis is organized with the following structure:

- Chapter 2 : Discusses the background, state of the art, identifies gaps in the state of the art, and explains how CAPIO is designed to address these gaps.
- Chapter 3 : Introduces CAPIO, outlining its goals and principal features without delving into excessive technical details.
- Chapter 4 : Presents the semantics and syntax of the CAPIO coordination language and demonstrates how it enables streaming communication in batch workflows.
- Chapter 5 : Describes the CAPIO runtime, its implementation, and its integration with the coordination language.
- Chapter 6 : Validates this work by evaluating the performance of CAPIO through synthetic benchmarks and real-world workflows.
- Chapter 7 : Summarizes key points and outlines areas for future work.

1.5 How to read this thesis

The thesis may be approached entirely or selectively, depending on the reader's background and objectives. Readers with a comprehensive understanding of parallel computing and HPC may bypass Chapter 2, which furnishes foundational and contemporary insights into these subjects. Those

interested in CAPIO's core concept may proceed directly to Chapter 3. Chapter 4 delineates the CAPIO coordination language, offering a deeper elucidation of the principles expounded in the preceding chapter. Chapter 5 holds particular significance for those interested in understanding how middlewares can be implemented to support the CAPIO coordination language, enabling the transparent injection of streaming capabilities into file-based workflows. Chapter 6 provides a practical exploration of CAPIO's application in real-world scenarios, illustrating the composition of configuration files employing the CAPIO coordination language. Finally, readers interested in the current limitations and potential future work regarding CAPIO will find Chapter 7, which concludes the thesis, particularly relevant.

1.6 Research Impact

CAPIO is a foreground or background technology of three European Projects:

- **ADMIRE** a European-funded project with a budget of €7.9M that started on 1st April 2021. The primary goal of the ADMIRE project is to improve I/O performance in HPC systems by developing a software stack that can dynamically adapt to changing computation and storage needs through malleability in computation and I/O and efficient resource scheduling across all levels of the storage hierarchy. CAPIO foreground technology.

- **ACROSS** a three-year European project with a substantial budget of €8.8 million. The project primarily focuses on harnessing emerging pre-exascale infrastructures, designed as a stepping stone toward exascale systems. It will effectively employ sophisticated methods to describe and efficiently manage complex workflows. The project strongly emphasises achieving energy efficiency, involving extensive deployment of specialized hardware accelerators, continuous system monitoring, and intelligent job scheduling mechanisms. CAPIO foreground technology.

- **EUPEX** a European project with a total budget of €40,760,065.93 to deploy a hardware and software platform integrating the full spectrum of European technologies. CAPIO foreground technology.

1.6.1 List of publications

Below is the list of publications by the author that directly relate to the topic of this thesis or have influenced or inspired it:

- Alberto Riccardo Martinelli, Massimo Torquati, Marco Aldinucci, Iacopo Colonnelli, and Barbara Cantalupo. "CAPIO: a Middleware for Transparent I/O Streaming in Data-Intensive Workflows". In: *2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. Goa, India: IEEE, Dec. 2023. DOI: 10.1109/HiPC58850.2023.00031

- Giorgio Audrito, Alberto Riccardo Martinelli, and Gianluca Torta. "Parallelising an Aggregate Programming Framework with Message-Passing Interface". In: *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. 2023, pp. 140–145. DOI: 10.1109/ACSOS-C58168.2023.00054

- Javier Garcia-Blas, Genaro Sanchez-Gallegos, Cosmin Petre, Alberto Riccardo Martinelli, Marco Aldinucci, and Jesus Carretero. "Hercules: Scalable and Network Portable In-Memory Ad-Hoc File System for Data-Centric and High-Performance Applications". In: *Euro-Par 2023: Parallel Processing*. Ed. by José Cano, Marios D. Dikaiakos, George A. Papadopoulos, Miquel Pericàs, and Rizos Sakellariou. Cham: Springer Nature Switzerland, 2023, pp. 679–693. ISBN: 978-3-031-39698-4

- Iacopo Colonnelli et al. "Federated Learning Meets HPC and Cloud". In: *Machine Learning for Astrophysics*. Ed. by Filomena Bufano, Simone Riggi, Eva Sciacca, and Francesco Schilliro. Cham: Springer International Publishing, 2023, pp. 193–199. ISBN: 978-3-031-34167-0

- Giovanni Agosta et al. "Towards EXtreme scale technologies and accelerators for euROhpc hw/Sw supercomputing applications for exascale: The TEXTAROSSA approach". In: *Microprocessors and Microsystems* 95 (2022), p. 104679. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2022.104679

- Giovanni Agosta et al. "TEXTAROSSA: Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale". In: *2021 24th Euromicro Conference on Digital*

System Design (DSD). 2021, pp. 286–294. DOI: 10.1109/DSD53832.2021.00051

- Marco Aldinucci, Valentina Cesare, Iacopo Colonnelli, Alberto Riccardo Martinelli, Gianluca Mittone, and Barbara Cantalupo. "Practical Parallelizazion of a Laplace Solver with MPI". in: *ENEA CRESCO in the fight against COVID-19*. Ed. by Francesco Iannone. ENEA, 2021, pp. 21–24

- Marco Aldinucci et al. "Practical Parallelization of Scientific Applications with OpenMP, OpenACC and MPI". in: *Journal of Parallel and Distributed Computing* 157 (Jan. 1, 2021), 13–29. DOI: 10.1016/j.jpdc.2021.05.017. published

- Claudia Misale, Maurizio Drocco, Guy Tremblay, Alberto R. Martinelli, and Marco Aldinucci. "PiCo: High-performance data analytics pipelines in modern C++". In: *Future Generation Computer Systems* 87 (2018), pp. 392–403. DOI: 10.1016/j.future.2018.05.030

Chapter 2

Background

In this chapter, an exploration of the background and current state of the art is presented. Initially, the concept of workflows and the distinctions between traditional workflows and in-situ workflows are expounded upon. Following this, a comprehensive analysis of the contemporary landscape of I/O in HPC systems is provided, covering topics such as parallel I/O APIs, ad-hoc filesystems, and distributed filesystems. The chapter concludes by highlighting an identified gap within the current state of the art and elucidating how CAPIO functions to address and bridge this gap.

2.1 Parallel Computing

In 1965, Gordon Moore predicted that roughly every two years, the number of transistors in an integrated circuit would double [1]. Initially, this projection held true, but since the early 2000s, it has ceased to be accurate. Consequently, the importance of utilizing multiple CPUs and computers has increased for computational tasks that demand significant processing power. Modern CPUs are comprised of multiple cores, and high-speed connections like InfiniBand [32] are being developed to facilitate rapid communication between CPUs in different machines.

As explained in the next section, the necessity of designing programs capable of exploiting parallelism has spurred the development of languages and programming models aimed at simplifying the programming of such applications. However, writing parallel or distributed code is inherently more complex than writing sequential code, and it introduces potential issues. For

instance, errors in design can lead to situations of deadlock and starvation.

Deadlock occurs when a group of processes cannot continue their execution because each process is waiting for another process in the group to release a resource, typically a lock. Coffman [33] defined four conditions that must all be true for a system to experience deadlock: 1) a resource can only be used by one process at a time, 2) at least one process is holding a resource and waiting for another, 3) a process cannot be forced to release a resource, and 4) circular waiting: given a set of processes $\{P1, P2, P3, ..., Pn\}$, P1 is waiting for P2, P2 for P3, and so on, with Pn waiting for P1.

Starvation occurs when a process remains blocked, waiting for a resource without ever obtaining access to it.

Using a distributed system, composed of multiple computers, introduces additional challenges, with fault tolerance being among the most critical. With a large number of machines in a system, the likelihood of a node failure increases. A program is considered fault-tolerant if it can handle hardware failures without significant downtime. From a user perspective, having to rerun a lengthy program due to the failure of a single machine can be problematic.

HPC systems are systems with an high number of nodes equipped with multicores CPUs and often also with hardware accelerators such as GPUs, FPGAs, and DPUs, optimized for specific classes of computations. To leverage the capabilities of these accelerators, programmers utilize libraries that interface with them. It is crucial to exploit parallelism across all layers of HPC system architecture.

At the instruction level, parallelism is achieved through Instruction-Level Parallelism (ILP), where multiple instructions of a single-threaded programs are executed in parallel. Typically, this is facilitated by vector units in the CPU, which can be leveraged manually through specific vector instructions in assembly code or automatically by the compiler [34].

Within a single node, CPUs are composed of multiple cores, allowing programs to be divided into multiple threads that execute concurrently (Thread-Level Parallelism). Accelerators can speed up specific classes of computations compared to general-purpose CPUs. Programs can be designed to offload these kinds of computations to accelerators, allowing CPUs to focus on other tasks.

Another layer where parallelism can be exploited is the network. Processes and threads of an application can execute on different nodes of an HPC system, increasing parallelism beyond the confines of a single node.

More recently, it has become possible to further enhance parallelism by distributing computations across multiple HPC systems.

The concept of running computations across different HPC systems is akin to grid computing, as pioneered by Ian Foster et al. [35], where multiple computations are distributed across heterogeneous machines and local networks over the globe to solve complex problems. The main difference is that in grid computing the systems used for the computations are heterogeneous, loosley coupled, dynamic and therefore more distributed.

HPC systems employ distributed file systems (such as BeeGFS[36], Lustre[37], and GPFS[38]) that utilize multiple nodes to store files and their metadata. These file systems, prevalent in HPC environments, prioritize low latency and high throughput through parallel access across multiple machines, as discussed in greater detail in Section 2.4.4.

The network plays a crucial role in HPC systems, facilitating communication between processors located in different machines. Given the significant disparity between computational and data access speeds, designing networks for HPC aims to achieve low latency and high-speed communication. Consequently, new hardware and protocols, such as InfiniBand [32] and Omni-Path [39], have been developed to surpass the limitations of the TCP protocol, originally designed for the internet, not HPC.

One notable feature of these new protocols is Remote Direct Memory Access (RDMA) [40], enabling processes in a node to read data directly from the primary memory of another node without involving the operating system. RDMA offers greater efficiency than standard TCP communication by bypassing OS context switches and costly data copying between user and kernel spaces. It is noteworthy that RDMA capabilities have also become available in TCP in recent years.

Users interact with the HPC system by connecting to a login node via SSH rather than directly accessing the computation nodes. Upon login, users can execute their programs on the computation nodes in a batch manner using a job scheduler that adheres to a First-In-First-Out (FIFO) policy, taking into account user or project priorities. If an insufficient number of nodes are available, the job will wait until resources become available, prioritizing jobs with higher precedence at the given moment. Job schedulers are responsible for implementing policies to prevent job starvation. Examples of widely used

job schedulers include Slurm [2], PBS[1], and LSF[2].

Given the diverse range of users and their requirements in utilizing an HPC system, it is essential to provide different versions of libraries and compilers. This is typically achieved through the use of modules, allowing users to activate or deactivate specific versions of installed software. In recent years, more user-friendly package managers such as Spack [41] have become increasingly prevalent. If users need to install new software, they must do so in their home directory or seek assistance from system administrators, as users typically lack root permissions for security reasons. Currently, the absence of a graphical user interface (GUI) for node interaction, the requisite knowledge for installing or activating software dependencies, and the importance of understanding the available hardware and software stack, including features like RDMA, can pose considerable barriers to entry for scientists lacking a background in computer science. These challenges may impede their ability to fully exploit the capabilities of an HPC system.

Programming software to fully utilize HPC systems can be challenging, as programmers must exploit parallel I/O with the file system, parallel network communication between nodes, various types of accelerators, and different CPU architectures (e.g., ARM vs. X86). Consequently, various libraries have been developed to assist programmers, including I/O libraries such as MPI I/O [11], Damaris [13], and ADIOS2 [14], as well as libraries for utilizing accelerators such as CUDA [42] and OpenACC [43].

Given the importance of writing distributed and parallel programs in HPC, coordination languages play a crucial role in enabling developers to focus on business logic without delving into low-level details. The subsequent section will explore such languages in detail.

2.2 Coordination Languages

Coordination languages are employed to coordinate various computations aimed at solving a problem. They serve to synchronize computations and facilitate data communication. In cases where a problem exhibits embarrassingly parallel characteristics, computations can be replicated to address subsets of the problem with little or no need of synchronization and comunication. A simple example is adding a costant value to all integers in an

[1]PBS: https://www.openpbs.org
[2]LSF: https://www.ibm.com/docs/en/spectrum-lsf

array. Every process can add the costant to its own subset of the array with out need to worry about the other processes.

However, more complex problems necessitate additional coordination between processes. Different synchronization mechanisms have been developed for this purpose. Dijkstra's seminal work [44] introduced semaphores as one of the simplest forms of synchronization. More sophisticated mechanisms, such as monitors, as described by Hoare [45], provide higher-level coordination capabilities. Coordinating processes across different machines presents additional challenges, as demonstrated in the influential work of Leslie Lamport. His contributions include addressing issues such as the ordering of events in distributed systems [46], resilience in scenarios like the Byzantine Generals problem [47], where one or more components of a distributed system start malfunctioning and sending erroneous and conflicting information, determination of a global state [48], consensus algorithms for decision-making among proposed values [49], and more. These works provided the foundation for developing coordination languages and designing algorithms within these languages.

Coordination languages provide syntax to express synchronization and data comunication(I/O). While some languages abstract away communication and synchronization details and rely on predefined patterns, others adopt a more low-level approach, offering various mechanisms for synchronization and communication.

For instance, high-level coordination languages, such as those utilizing algorithmic skeletons [21, 22], encode parallel patterns (e.g., pipeline, farm, etc.) in a functional form. These languages do not require explicit coding of synchronization and communication mechanisms as they are implicit in the pattern.

In [19], coordination languages are categorized as endogenous and exogenous. Endogenous languages utilize constructs for coordination within the computation, while exogenous languages separate computation from coordination. Examples of endogenous coordination languages include Linda [20] and MPI [50], whereas Reo [51] is exogenous. Coordination languages provide mechanisms for defining coordination rules. Some languages offer primitives for specifying rules, such as Linda and Reo, while others activate rules when specific actions are performed by one or more participants in the computation. Another aspect of coordination languages is the medium used for data communication and coordination. For instance, Linda defines a single shared data space called tuple-space, while other languages allow the definition of

multiple shared data spaces through which processes can interact. In the following paragraphs, the most influential coordination languages employing a high-level approach will be briefly outlined, followed by the most influential languages with a low-level approach.

Linda [20] is a coordination language that defines a shared data space, called tuple-space, which serves as a medium for data exchange. Processes communicate through tuples, and Linda provides synchronization mechanisms for coordination. A process can publish a tuple in the tuple-space, read a tuple from the shared space that matches a pattern, or copy a tuple from the tuple-space that follows a given pattern. Both reading and copying are available in two versions: blocking and non-blocking. Another built-in mechanism is the creation of processes for evaluating tuples. Linda is not a complete programming language; rather, it is designed to coordinate programs written in other programming languages such as C. It is one of the most widely-used coordination languages, with multiple implementations available in common programming languages such as C, C++, Python, Java, and Go, among others.

The Bulk Synchronous Parallel (BSP) model [52] consists of a collection of computational units with their own local memory, a network facilitating point-to-point communication between these units, and a facility capable of enforcing global synchronizations (barriers) among all computational units. Each unit is executed independently and concurrently, and the computation is organized into supersteps, with a barrier enforced at the end of each superstep. A barrier serves as a synchronization point, preventing units from progressing beyond it until all other units have also reached the barrier. At the the end of a superstep and prior to the barrier, units may communicate by exchanging point-to-point messages. The BSP model comes with a cost model for performance prediction of BSP algorithms. The cost depends on following factors:

- p, the number of processors
- N, the number of supersteps
- g, the time spent to deliver a message (assumed costant for every superstep)

- L, the costant overhead for the barrier and the startup of the communication (assumed costant for every superstep)
- h-relation, a superstep where the p processors send and receive at most h messages each.
- h_i, the maximum number of messages sent and received by a processor during the superstep i.
- w_i, the maximum time spent by a processor for local computation in a superstep i.

Then the cost of a superstep i is given by $T_{step_i} = w_i + h_i g + L$ where $h_i g$ is the biggest time spent for the communication by a processor during the superstep i (g, the constant time required to send/receive a message, multiplied by h_i, the maximum number of messages received and sent by a processor during i). The cost of the entire computation is obtained by summing the cost of all supersteps: $\sum_{i=1}^{N} T_{step_i}$.

The actor model was first published in 1973 by Carl Hewitt et al. [53]. In this model, computation is divided into actors that communicate by sending and receiving messages. When an actor receives a message, it can send messages to other actors, potentially spawning new actors and modeling behavior for subsequent messages. There is no particular order to these actions, and they can be executed in parallel. Communication is asynchronous, and it does not rely on channels; instead, an actor sends a message specifying an address that identifies another actor. With the use of addresses, it is possible to simulate named channels. An actor knows the addresses of actors it created, but it can also obtain addresses of other actors through messages. Each actor functions autonomously, maintaining its individual state and executing its behavior independently, ensuring that modifications to one actor's state do not affect others unless communicated through message exchanges. Actors may reside either locally or on remote machines within a distributed system, with the location of other actors transparent to each actor, facilitating seamless communication regardless of physical location. Notably, the actor model inherently supports scalability and fault tolerance. By distributing actors across multiple nodes, the model enables parallelism and load balancing. Furthermore, the isolation of actors facilitates fault tolerance, as

failures in one actor do not compromise the overall system. The actor model has inspired the development of process calculus languages.

Process calculus denotes a collection of formal mathematical languages employed for the modeling and analysis of concurrent systems developed from the Communicating Sequential Processes (CSP) model [54]. These systems encompass multiple interdependent processes whose behavior evolves over time. Process calculus furnishes a structured framework for articulating the dynamics of these processes, encompassing their behaviors, interactions, and synchronization mechanisms. This formalism enables precise delineation of concurrency and communication patterns, thereby facilitating rigorous reasoning about distributed and parallel systems. Notable instances of process calculi are π-calculus and Join calculus.

π-calculus [55, 56, 57] is a process calculus employed for coordinating processes through message passing across named channels. These channels possess the capability to be dynamically created, thereby affording flexibility in the construction of concurrent systems. As a Turing complete formalism, π-calculus holds the capacity to emulate any Turing machine. Its formal semantics serve as a valuable tool for the systematic analysis and verification of concurrent systems, enabling precise reasoning regarding their behavior. π-calculus has undergone numerous extensions, one notable example being the asynchronous π-calculus, which introduces the capability of asynchronous message passing.

Join calculus [58] constitutes an additional process calculus which expands upon the principles of π-calculus and asynchronous π-calculus by introducing the concept of "joins", which serve as synchronization points between concurrent processes. Join calculus furnishes a formalized framework for the representation of distributed systems and concurrent computations, accentuating the compositional nature of processes and the adaptable nature of synchronization mechanisms. It presents a succinct and expressive linguistic structure for the precise delineation of intricate coordination patterns, rendering it apt for the modeling of diverse concurrent systems and protocols

Reo [51] is a coordination language that defines a set of components interconnected in a circuit to facilitate communication between them. These components include nodes, which encompass boundary nodes, and channels

with assigned types. Components can perform I/O operations on the boundary nodes to which they are linked. There are two types of I/O operations: one for dispatching data items to a node and another for fetching data items from a node. All I/O operations are blocking. Nodes propagate the data from one of the incoming channels to all outgoing channels without storing or modifying it. If multiple incoming channels can provide data, the node makes a nondeterministic choice among them. In contrast, channels have user-defined behavior defined by their type. This means that channels may store or modify data items passing through them. The behavior of a channel is determined by its type, which can specify actions such as acting as a FIFO queue of dimension N or filtering data based on a given condition.

OpenMP [59] offers an application programming interface designed to parallelize code written in C, C++, or FORTRAN within a shared memory system. Through the use of preprocessor directives known as pragmas, OpenMP facilitates the parallelization of specific sections of code and provides mechanisms for process synchronization. One of the advantages of OpenMP-modified code is its portability; if a compiler lacks support for these directives, it simply disregards them. In contrast to low-level tools where the programmer must explicitly define the behavior of each thread (creation, destruction, scheduling, and synchronization), OpenMP delegates these responsibilities to the runtime environment.

OpenACC [43] shares similarities with OpenMP in its utilization of preprocessor directives to delineate the parallelization of sequential code. However, OpenACC is specifically tailored to harness the capabilities of hardware accelerators like GPUs, with an emphasis on minimizing target-specific directives. Currently, OpenMP also supports GPU offloading, blurring the line between the two interfaces.

P3L [23] provides constructs for common parallel programming pattern that can be combined to implement parallel programs. For this reason P3L is considered a structured language. P3L is designed to be used with an host language employed for writing the sequential parts. The P3L constructs are combined together to applying the desider parallelism to the sequential part written in the host language. Examples of these constructs are *farm*, *map* and *pipe*. The *farm* construct represents a set of identical workers that

execute in parallel tasks coming from an input stream and then produce the output in an output stream. The *map* feature expresses as set of identical workers that apply a function to a partition of an input stream without any communication between the workers. Then their output is combined and written into an output stream. The *pipe* construct is used for modelling a list of computations that are executed sequentially over an input stream. P3L comes with a set of compiling tools that provide efficient portability of applications written in P3L.

SkePU [26] stands out as a skeleton programming framework designed for both multicore CPUs and multi-GPU systems. Operating within a C++ environment, SkePU offers a range of data-parallel skeletons, including map, reduce, and scan functionalities. Notably, each skeleton boasts multiple implementations (such as OpenMP, OpenCL, and CUDA), ensuring portability across CPUs and multi-GPU setups. The latest iteration, SkePU 3, introduces an MPI implementation, extending its usability to distributed systems.

Intel TBB (Threading Building Blocks) [60] is a C++ template library crafted by Intel, offering high-level abstractions such as "parallel for" and "parallel for each" to parallelize independent loops. Employing an algorithmic skeletons approach, TBB furnishes containers that support concurrent operations alongside low-level synchronization mechanisms like mutexes and atomic operations (e.g., fetch and add, compare and swap, etc.). To maximize core utilization, it adopts the work-stealing scheduling strategy [61].

SkeTo (Skeletons in Tokyo) [62] is a C++ framework built on MPI, offering various distributed structures and algorithmics skeletons that leverage these distributed structures. The core structures in SkeTo include lists, matrices, and trees. Programmers develop sequential programs, which are then parallelized using these structures and the skeletons provided by SkeTo.

GrPPI (Generic Parallel Pattern Interface) [25] is a C++ programming interface tailored for streaming applications. GrPPI enables users to parallelize sequential streaming code using well-known parallel frameworks such as OpenMP and TBB, which were not originally designed for streaming processing. It offers high-level streaming patterns like pipeline and farm. While

refactoring sequential code with GrPPI can be challenging, there are techniques available for semi-automatically expressing GrPPI patterns in sequential C++ code [63].

Fastflow [24] is an open-source parallel programming framework designed for both shared-memory and distributed systems [64]. Fastflow offers both stream parallel skeletons (such as pipelines and farms) and data parallel skeletons (including map and reduce operations). Fastflow programs are composed by combining these skeletons. Fastflow is structured into different layers, each building upon the previous one to provide a higher level of abstraction. The bottom layer, known as "building blocks", provides ff_node constructs and collective channels beetween them. An ff_node represents a node in a streaming processing graph, where users define sequential code (kernel) utilized in parallel streaming execution. The collective channel serves as a communication link between multiple ff_nodes, implemented using state-of-the-art lock-free queues to avoid unnecessary data copies. The next layer, termed "core patterns", offers streaming skeletons like pipelines and farms. Finally, the top layer, named "High-level patterns", facilitates the exploitation of data parallelism akin to OpenMP. FastFlow also facilitates the utilization of GPU accelerators, enabling the integration of GPU-specific code written in CUDA or OpenCL.

2.2.1 Low level coordination languages

This class of languages is considered to operate at a lower level of abstraction compared to the previously mentioned languages. They provide fundamental communication and synchronization mechanisms, offering a high degree of flexibility while increasing the complexity involved in avoiding issues such as deadlocks and process starvation. In some cases, like with CUDA, users require deep knowledge of hardware details to achieve desired performance levels. It is noteworthy that despite their lower-level nature, these languages remain crucial as they are often employed to implement higher-level abstraction models outlined in the previous section.

MPI (Message Passing Interface) [50] is a specification interface for the message passing paradigm. Initially, MPI adhered to the Single Program Multiple Data (SPMD) paradigm [65], but recent versions also incorporate

support for Multiple Programs Multiple Data (MPMD) paradigms. Various implementations of MPI exist. One of the primary advantages of MPI is its portability, as it defines a standardized API for inter-process communication via messages. The point-to-point functions within MPI dictate how messages are sent and received between processes. Additionally, MPI provides collective functions, which are built upon the point-to-point functions, enabling complex and beneficial group communication operations. The purpose of MPI is to offer a standardized API for communication, facilitating optimization across diverse hardware architectures. Notably, MPI is not tied to any specific programming language; it is language-independent. However, MPI does specify bindings for C and FORTRAN, outlining how the functions defined in MPI can be utilized within these languages.

POSIX threads [66], commonly referred to as *Pthreads*, offer an application programming interface for multithreaded programming within a shared memory environment. Pthreads provide low-level primitives for creating and synchronizing threads. However, Pthreads do not include built-in mechanisms for communication between threads. It is the responsibility of the programmer to implement communication using shared data structures, such as multi-producer, multi-consumer queues. Therefore, programmers must also handle the implementation of these data structures and the necessary synchronization methods to ensure their proper utilization.

CUDA (Compute Unified Device Architecture) [42] is a thread-centric programming model that adheres to the GPGPU (General-Purpose Computing on Graphics Processing Units) paradigm, allowing programmers to write programs for Nvidia GPUs. GPGPU extends the use of GPUs beyond traditional computer graphics tasks for which they were initially designed. CUDA is intricately linked to the execution model of Nvidia GPUs, which exemplify the SIMT (Single Instruction, Multiple Threads) architecture. Considered a low-level programming model, CUDA requires users to possess knowledge of the underlying architecture to fully exploit its capabilities. For optimal performance, programmers must understand concepts such as memory coalescing, where multiple memory accesses are combined into a single operation. Efficient memory coalescing relies on threads with neighboring IDs accessing adjacent memory locations. In CUDA, computations are organized into groups of threads known as thread blocks, which can be executed either seri-

ally or in parallel. The size of each block is determined by the user, and the optimal number depends on the specific architecture. Another example of low-level optimization in CUDA is the subdivision of data transfer between the CPU and GPU into streams. This approach allows for the overlap of data transfer and computation, enhancing overall performance.

OpenCL (Open Computing Language) [67] is a framework that enhances the capabilities of the C and C++ programming languages for heterogeneous systems. It enables code execution across various hardware platforms, including CPUs, GPUs, and FPGAs. Developed collaboratively by multiple vendors, OpenCL ensures portability of code across a diverse range of platforms. For instance, if a system lacks specific accelerators, the program can seamlessly execute on CPUs instead. This inherent flexibility makes OpenCL well-suited for heterogeneous systems comprising both CPUs and GPUs.

UPC (Unified Parallel C) [68] is a shared-memory model wherein processes interact with memory using standard functions such as read and write. Programmers are responsible for synchronizing processes using locks and barriers. UPC follows a PGAS (Partitioned Global Address Space) approach, where each process is associated with both a shared and a local portion of memory. Processes have exclusive access to their respective local portions, while the shared portion can be accessed by all processes. Other notable PGAS libraries include UPC++ [69], X10 [70], Chapel [71] and Global Arrays [72].

At the end of this chapter, in subsection 2.5, the gap in the state of the art is identified, and it is explained how the CAPIO language addresses it. Additionally, a table will be presented comparing the coordination languages discussed in this section, including the CAPIO language (table 2.2).

2.3 Workflows

Workflow graphs are widely used to model and execute complex scientific applications on large-scale distributed architectures such as supercomputers and cloud infrastructures. A *workflow* can be defined as a directed bipartite graph $W = (S, P, D)$, where S is the set of steps, P is the set of ports, and $D \subseteq (S \times P) \cup (P \times S)$ is the set of dependency links. Let $In(s), Out(s) \subseteq P$

be the sets of input and output ports of a step s. The behavior of s can be described with a function f_s, taking arguments in $In(s)$ and returning values in $Out(s)$. In this setting, a path connecting step s to step t introduces a partial execution order $s \prec t$. Workflow can be modeled with well-known formalization tools such as Petri Nets [73] and dataflow graphs [74]. Both models come with *token-pushing* semantics [75], i.e., steps are enabled by the presence of *tokens* in their input ports. An example of a workflow graph is sketched on Fig. 2.1. It has 4 distinct steps, one of them (W) replicated n-times. A workflow specification incorporates two different classes of semantics [74]: the *host semantics*, which define the subprogram in each workflow step (i.e., the body of f_s), and the *coordination semantics*, which defines the interactions between *steps*. Tools in charge of exposing coordination semantics to the users and orchestrating workflow executions are called Workflow Management Systems (WMSs). The coordination semantics can interleave with host code into a single program, as in task-based programming libraries like Dask [76], COMPSs [77], and Ray [78]. Steps are functions that can be executed as asynchronous remote tasks, while tokens are implemented with the *futures* mechanism [79]. The workflow execution plan, typically a layered dataflow model [80], is automatically built just-in-time by the runtime layer of the framework. These libraries are often used to model HPC workflows, being a middle ground between *high-level WMSs* and explicit message-passing libraries for complexity and performance. However, the host application must be entirely (re-)written in one of the supported languages.

Conversely, *high-level WMSs* express coordination semantics using a host-independent medium. Some WMSs rely on a Domain Specific Language (DSL), like the Common Workflow Language (CWL) [30] or the Snakemake DSL [81], whereas others adopt a general-purpose programming language (e.g., Pegasus [82] and DagOnStar [83, 84]) or a Graphical User Interface (e.g., Kepler [85] and Jupyter Workflow [86]). This approach is more flexible, as it does not impose constraints on the host application code and does not require any modification to the business logic. However, the fact that the host application cannot communicate with the coordination layer imposes specific semantics that we define as "*on-termination*", i.e., produced tokens are propagated to consumer steps only after the producer has terminated its execution and thus consolidated all produced data. The *on-termination* semantics forces batch execution of the steps, i.e. a step can be executed

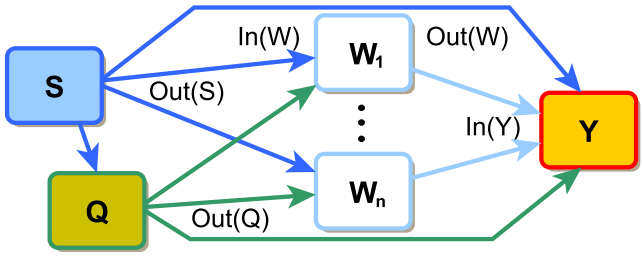

Figure 2.1: Example of a workflow graph.

when all the steps producing its input data have concluded their run.

In dataflow models typically used to express scientific workflows, tokens carry data. In most implementations, such data can be files or instances of primary types (e.g., integers, strings, floats). Some WMSs also support composite types and containers, e.g., arrays or maps. In this work, we are interested in I/O-bound scientific workflows that extensively use files and directories as dataflow tokens (e.g., the 1000-genome workflow [87]). Consequently, throughout this document, the terms "token" and "file" are synonymous.

2.3.1 Traditional workflows vs in-situ workflows

Traditional workflows[3] steps need to be executed orderly according to the data dependency graph, and it could be challenging to exploit pipeline parallelism among them. Nevertheless, in many scientific simulation workflows, the core simulation steps producing data might be co-executed with the data analysis steps for almost real-time evaluation of results. Unless the analysis steps were developed to support such coupled execution with the simulation steps through explicit synchronizations, the analysis steps need to wait for all data results to be produced by simulation phases into the storage before starting their analysis. The benefit of batch workflows is the modularity and simple design. Infact every application can be designed programmed in stand-alone without worrying about the other part of the worklfows. The price to pay is the potential worst performance in cases where applications could run concurrently exploiting streaming communication.

[3]Through the document traditional workflows are also referred as batch workflows.

In-situ workflows [88] were introduced to alleviate the limitations associated with relying on distributed file systems as communication media, with the goal of facilitating temporal parallelism between steps. In in-situ workflows, multiple steps can execute concurrently, and data communication occurs promptly upon data production, facilitated by explicit coordination mechanisms like message-passing or coordinated file access through suitable APIs. This eliminates the necessity to await the termination of the producer and can significantly improve the performance of the entire workflow.

In-situ systems can be categorized as integrated or connected based on their workflow structure. Integrated in-situ systems incorporate in-situ capabilities within a shared execution context, whereas connected in-situ systems treat each step of the workflow as a separate component [89]. For instance, Catalyst [90] and libsim [91] offer in-situ analysis within the simulation, falling into the integrated category, while Dataspaces [12] allow applications to connect and subscribe to a channel, representing the connected category. Connected in-situ systems offer a more dynamic execution environment and increased fault tolerance.

2.3.2 Workflow management systems

A workflow management system (WMS) allows users to express the coordination between the steps of a workflow. These systems are not concerned with the host semantics, i.e., the specific computation carried out by each step. Various workflow management systems have been developed, and several surveys exist to compare their functionalities. The primary purpose of a WMS is to assist the user during the modeling phase, where the workflow structure is defined, and during the runtime phase, where the WMS deploys the workflow and manages the available resources.

Numerous scientific WMSs, including Kepler [85], Pegasus [82], Taverna [92], and Galaxy [93], were developed in response to the growing demand for more efficient Grid computing infrastructures. These systems provide researchers with enhanced capabilities to manage complex workflows across diverse computing environments. However, each WMS often comes with its own specialized communication framework, typically relying on low-level libraries and tools that need to be meticulously installed and configured on every node participating in the workflow. For example, Pegasus utilizes a Grid-specific technology named HTCondor [94], to handle task distribution and data transfer. While effective, this reliance on specialized stacks can

restrict the flexibility and portability of these WMSs across different computational environments.

Traditional product-specific DSLs often bind workflows to a single software platform, limiting their portability and reusability across different environments. To address these challenges, the development of standardized workflow languages has become a priority. One prominent example is the Common Workflow Language (CWL), an open standard that uses JSON or YAML syntax, or a combination of both, to define workflow DAGs in a flexible and interoperable manner. Another alternative is DagonStar, which offers a straightforward Python API designed specifically for coordinating workflows in computational environmental science. A key innovation of DagonStar is its workflow:// schema, which unifies all task-specific directories into a shared virtual file system. This approach ensures consistent data handling, even when workflows are executed across multiple, geographically dispersed resources.

2.4 I/O in HPC Systems

In HPC systems, I/O operations can often become a bottleneck for data-intensive workflows, particularly those involving the processing of terabytes of data. Distributed file systems typically distribute files and their metadata across multiple machines, some of which may be dedicated for this purpose. Accessing files may entail multiple steps across various nodes, potentially leading to performance challenges. Moreover, not only the file content but also its metadata can contribute to bottlenecks. The presence of a large number of files in a directory can exacerbate performance issues in a distributed file system. To address such challenges, various solutions have been developed. These range from libraries like MPI I/O [11], which facilitate parallel reading and writing, to temporary ad-hoc file systems such as GekkoFS [95] and Hercules [31], which leverage local burst buffers or main memory resources on nodes avoiding the distributed filesystem. Additionally, tools like ADIOS2 [14] offer streaming parallelism for reading and writing data in in-situ workflows while Damaris [13] enforce asynchronous I/O dedicating a core or an entrire node that writes data to the filesystem asynchornous while the main program can continue with its computation. These solutions will be further discussed in the following sections. Here is presented a possible classification of tools designed to improve I/O performance of HPC workflows:

- API-Driven Approach: In this approach, the tool provides an API, and applications need to be modified to take advantage of these tools.
- Transparent Approach: I/O improvement is achieved by an external tool without user intervention, except for the installation of such tools. Examples of this approach include distributed filesystems, most ad-hoc filesystems, and distributed shared memories [96, 97, 98].
- Hybrid Approach: Users provide specific I/O hints for an HPC workflow to the tool before executing the workflow. This approach is considered hybrid because, like the Transparent Approach, it doesn't require modification of the original code but does require some high-level I/O information specific to the application before running it.

The API approach potentially offers higher efficiency but demands more effort from the user. Conversely, tools built with a Transparent Approach are easy to use but may yield inferior performance. The Hybrid Approach strikes a balance between the two.

2.4.1 API driven I/O

With API-driven I/O tools, users must instruct them through an API, which requires more user effort but offers the potential for greater flexibility and improved performance.

There are various methods for achieving parallel I/O in POSIX, each with its own advantages and disadvantages.

The *sequential I/O* approach involves sending all data from all processes to a master process, which then writes the data into a file. Similarly, when reading a file, a master process reads the entire file and then distributes parts of it to each process. While this technique is straightforward to implement, it is inefficient because it fails to exploit parallelism when reading and writing data to the distributed file system.

Another approach is the *independent parallel I/O* approach, where each process reads and writes its own file or different parts of a single file. This approach increases parallelism, as each process can read and write data independently without the need for communication or synchronization. However, a drawback arises when multiple processes write different files, potentially leading to a high number of accessed files and increased metadata, which can become a bottleneck in a distributed file system [99, 100]. Additionally, each

process independently makes requests to the distributed file system, which can be optimized by aggregating requests from different processes to reduce network traffic and minimize disturbance to other processes and users.

The *collective parallel I/O* approach involves parallel reading and writing of different sections of a file, with processes cooperating to reduce the number of requests sent to the file system. For example, MPI I/O provides collective I/O functions to simplify programming for these non-trivial I/O operations. This approach allows for *data sieving* [101], where numerous small and non-contiguous data accesses are aggregated into a few large contiguous accesses. Aggregating I/O requests into large and contiguous data accesses also improves file system data prefetching and caching.

in the following paragraphs, some notable I/O libraries used in HPC will be discussed.

Damaris [13] is a framework that improves the I/O performance of an application dedicating cores or machines for the I/O tasks in order to transform synchronous I/O operations in asynchronous I/O operations. Damaris provides an API designed to modify the existing application as little as possible. It provides in-situ visualization for scientific simulations integrating VizIt. It is possible to execute C++ (or Python) user code in the dedicated cores/nodes for I/O using the plugin system of Damaris; this is useful for in-transit processing. With Damaris is possible to maintain an high level data abstraction because modifying the existing code with the Damaris API, the semantic of the data is preserved. The authors claim that this feature is necessary to allow Damaris to write the data in an high-level format as HDF5 or NetCDF and to be integrated in an in-situ visualization pipeline. Therefore, to preserve the semantic of the data, the user must provide an XML file with the description of the data to the Damaris run-time.

Apache Arrow [102] is an open-source, cross-language platform designed for in-memory data processing. It offers a standardized, language-independent specification for representing structured data. The primary objective of Apache Arrow is to facilitate efficient data interchange among diverse systems and programming languages. It defines a universal data format compatible with languages such as Python, Java, C++, and more, eliminating the need for costly data serialization and deserialization. Apache Arrow has gained substantial recognition within the data processing and analytics community

and enjoys widespread adoption in numerous projects and applications, including prominent data processing frameworks like Apache Spark

Dataspaces [12] is a decentralized framework that provides a shared and distributed space that can be accessed by coupled applications for communication. It allows to access live data produced by an application using the semantic operator provided. The data used for the communication with Dataspaces must be specified as key-value pairs. Dataspaces provides the application the capabilitites to express the publisher-subscriber pattern and monitoring a region of the shared spaces for data vaiabilty or the use of a custom filter. The programmer must only express the type of communication and dataspaces wuold take care of the technical details transparently.

MPI I/O [11]: is a library that provides collective I/O operations to MPI applications. These functions exploit parallelism and minimize the comunications for retrieving the data, assembling multiple I/O requests in one request.

Like MPI I/O, *ADIOS2* [14] and HDF5 [103] improve the I/O of HPC scientific applications providing a library at the user level. ADIOS2 is a complex and an highly modular framework that provides data management on different layers, from the desktop to a powerful High Performance Computing system. It provides three APIs: a public Low-Level APIs for HPC applications and workflows, a private High-Level API for scientic data analytics and visualization and a private API for ADIOS2 internal modules. ADIOS2 can be used both for parrallel I/O on files and parallel streaming communication between processes. ADIOS2 is designed in a modular way allowing it to use differnt components (called engines) that support differnt backends and use cases (UCX, POSIX I/O, etc..).

2.4.2 Transparent I/O

With the transparent I/O approach, users only need to install and, if necessary, launch the tool to benefit from it. The drawback of this approach is that if the user programmed the I/O of their applications poorly, there is only so much that transparent I/O can do without additional information. API-driven I/O tools and transparent I/O tools are not mutually exclusive;

they can be used together to provide overall better I/O performance. For example, a workflow could improve its performance using an ad-hoc filesystem, but if one of its data-intensive application writes and reads files sequentially using a master process, the improvement would not be significant. In this case, the user can consider using a parallel I/O API and ad-hoc filesystem together.

2.4.3 Ad-Hoc File Systems

Ad-hoc file systems [10] are temporary file systems with a lifespan limited to a single application or an entire workflow. They aim to manage the diverse types of storage available in HPC systems, leveraging main memory, especially for temporary files. Typically, ad-hoc file systems operate as Transparent I/O tools, although there are exceptions that also provide an API.

Ad-hoc file systems attempt to address the limitations of distributed file systems for specific use cases. Some achieve this by utilizing local storage on computational nodes or by relaxing POSIX semantics. For example, some disable support for file locking, an operation that can be costly in distributed file systems under certain conditions.

Ad-hoc file systems can be implemented in kernel space, like classic file systems, or in user-space. User-space file systems are well-suited for HPC environments because users do not need to install kernel modules with the assistance of system administrators. Most ad-hoc file systems can be used without modifying the target application. They intercept system calls to the distributed file systems and execute their code instead of passing control to the kernel. From the application's perspective, this process remains transparent.

There are multiple methods for intercepting I/O system calls and injecting code to be executed in user space. One such method is the use of FUSE [104], but it is generally avoided in HPC due to performance issues caused by context switches between kernel and user space, as well as overhead in data communication between the application and FUSE[105]. Using FUSE file systems requires mounting them before use, which necessitates root permissions that users typically lack on HPC systems for security reasons. Therefore, other methods are preferred for implementing ad-hoc user-space file systems.

The LD_PRELOAD method in Linux operating systems allows for capturing libc functions by exploiting the linker's capability to intercept stan-

dard library calls with user-space wrappers, either statically during linking time or dynamically by setting the LD_PRELOAD environment variable. While avoiding kernel involvement during I/O can yield performance benefits, the LD_PRELOAD method requires designers to handle all libc functions that the ad-hoc file system must manage. To simplify this task, the *syscall_intercept library* [106] was developed. Although it still relies on the LD_PRELOAD method, it only needs to capture system calls, reducing the effort required for creating ad-hoc user-space file systems. From the user's standpoint, defining the environment variable LD_PRELOAD to point to the path of the ad-hoc file system shared library suffices. Users do not need to modify or recompile applications.

Another alternative is to use libraries such as libsysio[4] and Gotcha[5], which function similarly to the *syscall_intercept library*.

Ad-hoc user-space file systems aim to improve I/O performance by leveraging the parallelism of HPC systems. However, to achieve this goal, they must introduce minimal overhead when interacting with user applications, as they share computational units, communication channels, storage, and memory.

One important emergent concept in HPC systems is the ability to run programs that are malleable. Malleability refers to the capability of adjusting the number of nodes during the execution of a program or workflow and is becoming an increasingly important feature in HPC environments to accommodate changes in computational and I/O requirements during application execution [107, 108, 109]. An ad-hoc filesystem designed to support malleability should offer mechanisms to seamlessly increase or decrease the number of nodes at runtime without data loss, while minimizing the time required for data transportation and redistribution.

GekkoFS [95] is a file system in user space that provides a relaxed POSIX semantic to improve specific I/O patterns used by scientific application exploiting the local burst buffers on the nodes of a cluster. It is decentralized and the data is distributed on local SSDs of the nodes in order to balance the I/O operations. It also divide large files in data chunks that are distributed on the cluster. GekkoFS relaxes the POSIX semantic to overcome the problems of the parrallel filesystems. For example, GekkoFS does not provide a

[4] https://github.com/hpc/lustre/tree/master/libsysio
[5] https://github.com/LLNL/GOTCHA

global locking mechanism to avoid the relative overhead. Therefore, it is the application or the libraries used by the application that must implement a locking mechanism if it needs to access files concurrently. Without locking mechanism GekkoFS can't gurantee to return the current state of a directory. For designs choiches, GekkoFS does not allows the use of certain operations such as rename because they can be very inefficient. The authors argue that, fortunely, these kind of operations are rare in HPC environments. Another way in which GekkoFS relaxes the POSIX semantic to gain a performance improvement is the option to disabilitate some metadata. GekkoFS divides the metadata in three categories: rarely used, redundant and mandatory. The metadata belonging to the first two categories can be disabilitated to improve the performance of the application. In a recent version of GekkoFS, the architecture was updated to include a proxy on each node, aiming to support malleable executions in the future [110] .

Hercules [31] is an ad-hoc file system that fully supports POSIX semantics. Hercules offers two primary modes of operation: using its API or capturing I/O system calls without the need to modify the original code. It leverages a client-server architecture and stores data in the nodes' main memory. Additionally, it can utilize NVMe memory for persistency when the main memory is full, implementing an LRU (Least Recently Used) policy for replacements. Users can set the replica factor for data to provide a degree of fault tolerance. From an implementation standpoint, Hercules employs a memory pool divided into blocks on each node. Files can be distributed across different blocks on various nodes, following one of the available policies. For efficient data retrieval and storage between the client and Hercules, the system relies on the UCX framework, ensuring fast communication in HPC systems. Hercules also stores metadata in memory using a distributed in-memory key-value store, organized as a balanced binary tree to facilitate rapid key lookup. It further provides an API, enabling its use as both a transparent I/O tool and an API-driven tool. Hercules supports malleability [111] to efficiently manage changing I/O requirements of applications by dynamically adjusting the number of nodes without data loss. An external agent can interface with Hercules to provide information about changes in node configuration. Alternatively, Hercules can employ internal heuristics, such as user-defined I/O throughput thresholds, to autonomously determine whether to scale the number of nodes up or down.

BeeOND [112] allows the creation of multiple instances of the distributed file system BeeGFS on the fly. It enables the utilization of local storage on computational nodes throughout the lifetime of an application to improve I/O performance. BeeOND can be seamlessly integrated with a job scheduler such as Slurm to initiate a new instance on a specific node when an application starts on that node and terminate the instance when the application ends. BeeOND facilitates the exploitation of local storage on compute nodes and facilitates data transfer to the underlying distributed file system, which can be BeeGFS or another file system, through a simple command. It establishes a shared namespace for the local storages across a given set of compute nodes and creates a mounting point with a straightforward command, effectively creating a distributed file system on the local storage of the compute nodes. BeeOND is POSIX compliant, enabling applications to interact with it like any other POSIX distributed file system without requiring additional user intervention.

Expand ad-hoc parralel file system [113] is derived from the Expand file system. While the Expand parallel file system targets heterogeneous distributed systems relying on NFS [114, 115], its new ad-hoc version is based on MPI. This ad-hoc file system does not require modifications to the code because it is based on a *syscall_interception library*, capturing the system calls of the target applications. It provides parallel access to files by dividing them into blocks and distributing these blocks across the nodes where Expand ad-hoc is running. Additionally, metadata such as size and permissions is distributed across all nodes for each file. A master node is assigned to each file, responsible for storing the distribution policy and block size for that file. The master node is selected by applying a hash function to the file's pathname. Users can customize the distribution of a subset of files (a partition) and the block size using an XML configuration file. Furthermore, Expand ad-hoc exploits data locality when two processes on a node interact through a piece of file stored on that node.

UnifyFS [116] is an ad-hoc filesystem designed to leverage local node storage to enhance the I/O performance of HPC applications. It offers configurability to provide relaxed POSIX semantics for applications that can benefit from them. When it comes to file writing, UnifyFS offers three distinct semantics:

- Read After Write Semantics: This corresponds to the standard POSIX behavior where changes to a file are immediately available after a write operation.
- Read After Sync Semantics: Data becomes visible after a synchronization operation, such as "fsync" or the related MPI I/O operation.
- Read After Laminate Semantics: Data becomes available after an explicit operation, which can either be added to the original program or configured as a specific POSIX operation like "close" or "chmod".

With these different semantics, reading data from local storage can potentially improve performance. However, it's essential to emphasize that this improvement is only achievable when the application has been designed with the appropriate synchronization logic for reading and writing data to comply with these specified semantics.

CRUISE [117] is an ad-hoc file system implemented in user-space that emulates a subset of POSIX semantics using the LD PRELOAD method. It is specifically designed for checkpointing and restart functionalities in HPC systems. To optimize performance, CRUISE does not provide full POSIX semantics, as certain information such as timestamps and file permissions are deemed unnecessary for checkpointing purposes. CRUISE operates by storing data in the main memory of the nodes where the application is running. When the data exceeds the memory allocated for CRUISE, it asynchronously moves the checkpoint data into the distributed file system. Additionally, CRUISE supports RDMA access from external processes to the data stored in the main memory of the node. This feature is particularly useful for pulling the data from CRUISE into persistent storage managed by a distributed file system.

BurstFS [118] is an ad-hoc filesystem built on top of CRUISE that exploits the local burst buffers to improve the I/O bandwidth of applications and reduce network congestion. To retrieve a specific section of a file, it implements a scalable distributed key-value store for metadata. To enhance the performance of bursty writes (common in HPC applications), it adopts a lazy synchronization approach where metadata is not updated after every write but instead after a predefined time lapse or after an explicit fsync operation. To reduce network congestion, a delegator process is spawned on

each node, and communication between nodes is facilitated through these delegators rather than involving all processes of the application. Therefore, if a process needs to read remote data, it makes a request to the delegator, which retrieves the requested data.

EchoFS [119] is a temporary file system implemented using FUSE that using hints provided by the scheduler, exploits local burst buffers managing the movement of data between different I/O layers of the HPC systems. If a job B need the data generated by the previous job A, then echofs maintains the data in the local (or shared) burst buffer. It also allows the user to provide a configuration file that expresses which file are temporary (in order to not propagate them from the burst buffer to the destination storage) and which files are used as communication between jobs. Following the categorization at the beginning of this section, this makes echofs a hybrid I/O tool. Echofs implements the POSIX API, i.e. the user can benefit from its features without change the code of the applications (but it must provide the configuration file for tagging the files produced by the jobs). Since the lifespan of the echofs file system coincides with the jobs of the user, there is an improvement on the metadata managing because the file system is only used by the owner of the jobs reducing contentions problems (while a parallel file system is used by all the user of the HPC system) and the permission to use the files must only be checked when the job use input files not generated by the workflow.

Li et al. [120] proposed the functional partional (FP) runtime environment that dedicates a part of the cores reserved to an application, to run critical I/O or data analysis task. The run-time is implemented using FUSE. The original application can benefit from the run-time without changes in the code. This tool provides an API that must be used to implement the aux-apps. An aux-app is a task that must be performed asynchronously on the data that the original application wants to write in the disk. Thanks to FUSE, the communication between the original application and the aux-apps is achieved through shared memory. The run-time does not provide a way that allow the aux-apps to communicate with each other. In the example created and used for the experiments the authors implemented the communication between aux-apps using socket programming. Therefore we can conclude that this tool is not suitable for an easy integration of independent

HPC applications.

NORNS [121] is not an ad-hoc file system but it employ an hybrid approach for improving I/O. It is a service that interfaces with the job scheduler Slurm in order to optimize the data movement between different jobs of a data driven workflow. The authors propose an extensions of slurm to express the data dependency between jobs and a service that orchestrates Slurm to exploit data locality and the differents I/O layers of HPC systems (e.g. burst buffers). For example the service can keep the data in the burst buffer of a node if the next job needed that data and will run in that node. NORNS is transparent to the applications, as the user expresses data dependencies of the workflow steps using an extension of Slurm.

At the end of this chapter, the major differences between the CAPIO runtime and the ad-hoc file systems presented in this section will be discussed.

2.4.4 Distributed File Systems

Distributed file systems are file systems deployed across multiple machines, where each machine has an identical view of the file system, providing users with a seamless experience akin to using a local file system. In HPC environments, distributed file systems are favored for their potential high bandwidth, achieved by parallelizing access to files through techniques such as data striping [122, 123]. While these file systems often adhere to POSIX semantics, they may relax them in certain cases for better performance. However, distributed file systems can become a bottleneck in certain scenarios. For instance, when a directory contains a large number of files, the metadata distributed throughout the HPC system can impede file access speed. To address such issues, ad-hoc file systems are increasingly adopted for their ability to alleviate the performance limitations encountered by POSIX distributed file systems in specific use cases. the following section will delve into some of the most commonly used distributed file systems used in HPC environments.

Lustre [124, 37] is as an open-source, parallel distributed file system tailored for high-performance computing and extensive storage setups. Its primary aim is to efficiently manage and facilitate access to massive datasets

across numerous servers and storage devices. Applied in diverse scientific, research, and enterprise environments, Lustre excels in scenarios where robust, parallel I/O performance is crucial. The file system employs a scalable architecture, distributing data across multiple storage servers in a parallel and striped manner. This design promotes concurrent file access and bolsters overall performance. Recognized for its high-bandwidth data access capabilities, Lustre is particularly well-suited for applications involving large-scale simulations, data analytics, and other data-intensive tasks prevalent in HPC settings. Key features encompass data striping, parallel I/O support, and scalable metadata handling to optimize performance. Lustre incorporates fault tolerance measures, including data replication, recovery mechanisms, and hardware redundancy support. Its flexible architecture allows seamless expansion to meet increasing storage demands by integrating additional servers and storage devices. Lustre provides users with the capability to deactivate file locking mechanisms, thereby relaxing the POSIX consistency semantics. This feature is intended to optimize performance for certain use cases.

BeeGFS [36] is a parallel file system designed to cater to the demands of high-performance computing environments. Its architecture involves the collaboration of multiple servers to manage and serve data in a distributed and parallel manner.

The system offers horizontal scalability, adapting seamlessly to the evolving needs of expansive HPC projects. Users can optimize data access and throughput by tailoring file striping policies, strategically distributing file components across multiple storage servers. BeeGFS maintains compatibility with POSIX standards, ensuring easy integration with applications relying on conventional file system semantics. BeeOND is the ad-hoc version of BeeGFS and provides a temporary on-demand deployment option leveraging burst buffers (refer to subsection 2.4.3).

BeeGFS provides a straightforward management experience through user-friendly tools and a web-based graphical user interface (GUI) for monitoring and configuring the file system. It incorporates fault tolerance features such as data replication and recovery mechanisms, ensuring data integrity even in the face of hardware failures.

GPFS (General Parallel File System) [38] is an high-performance clustered file system developed by IBM. It caters to the demands of large-scale parallel processing environments, specifically tailored for applications in HPC systems, and now bears the new identity of IBM Spectrum Scale.

This resilient file system facilitates concurrent read and write operations across multiple nodes within a computing cluster, ensuring vital parallelism essential for robust I/O performance in expansive computing environments. Demonstrating remarkable scalability, GPFS accommodates large clusters with thousands of nodes, excelling in efficiently managing substantial data volumes and delivering substantial throughput for parallel workloads.

GPFS employs a distributed architecture, distributing data across multiple servers or storage nodes. This strategic approach significantly boosts overall performance, strengthens fault tolerance, and optimizes load balancing.

Highlighting advanced features such as snapshot capabilities, GPFS empowers users to capture point-in-time copies of the file system. Additionally, it provides robust support for backup and restore functionalities, ensuring comprehensive data protection.

With a strong focus on high availability, GPFS incorporates features like failover and recovery mechanisms. These features are designed to ensure uninterrupted access to data, even in the face of hardware or network failures.

GPFS offers the capability to implement lazy metadata updates, thereby relaxing the POSIX semantics to enhance system performance.

2.5 Gap in the state of the art

In-situ workflows were introduced to leverage streaming parallelism among applications within a workflow [88]. Achieving this involves coupling applications and implementing explicit synchronization between them, a potentially time-consuming task. API-driven tools like ADIOS2 and Damaris were developed to address this challenge, but users are still required to modify existing code, transitioning from POSIX or other APIs to Damaris or ADIOS2 [125, 126]. Workflows often involve programs developed by different teams using various technologies, posing challenges for users seeking to modernize workflows into in-situ workflows due to potential knowledge gaps.

CAPIO offers a solution for users who prefer not to modify existing code or avoid using specific APIs. With CAPIO, it is possible to transform a batch

workflow into an in-situ workflow without modifying the code. CAPIO only requires a configuration file describing data dependencies and synchronization semantics between programs, using the high-level CAPIO coordination language. The CAPIO runtime captures system calls performed by applications and enforces synchronization between the workflow's steps. With CAPIO, all programs within a workflow can run together, potentially improving efficiency without the need for new code. The primary effort lies in writing the configuration file that describes the workflow's data requirements. CAPIO aims to improve the I/O performance of workflows incorporating both the Transparent Approach and the Hybrid Approach (refer to Section 2.4). It can function without any additional information, but if users provide relevant information, it can achieve improved performance.

Since the configuration file describes the data dependencies and the synchronization semantics, the computation is separated from the coordination making the CAPIO language an exogenous coordination language. Table 2.2 shows the principal differences between the coordination languages described in section 2.2, including the CAPIO language.

The CAPIO coordination language can be implemented as a connected in-situ system, where each step of a workflow is a separate component, or as an integrated in-situ system, where all the steps run in a shared execution context. The runtime discussed in this thesis currently operates as an integrated in-situ system, leveraging MPI, and does not support malleability in the number of nodes during workflow execution. Although the MPI_Comm_connect functionality could potentially address this limitation, it is not widely available on most supercomputers at present. Alternatively, other tools could be explored as alternatives to MPI, such as MTCL [127]. CAPIO could be integrated with WMSs to semi-automatically generate CAPIO configuration files. In future work, we aim to investigate the integration of CAPIO with DagonStar to leverage the workflow:// schema, which abstracts multiple (geographically) distributed file systems into one shared file system, enabling CAPIO to be used with workflows running on multiple HPC systems. In [128], the authors have already integrated an ad-hoc file system with DagonStar, demonstrating the potential for integration with CAPIO. In [129] it is shown a first integration of CAPIO with DagonStar.

Besides being used to improve the I/O performance of existing workflows without the need to modify the application's code, CAPIO can also be utilized to create new workflows. Users can design and implement workflows composed of standalone applications that produce and read files, simplify-

Ad-hoc File Systems	**User Space**	**Malleable**	**Storage Backend**	**Streaming Comm.**
GekkoFS	YES	NO	Bursts Buffers	NO
Hercules	YES	YES	Main Memory, FS	NO
BeeOND	NO	NO	Burst Buffers, FS	NO
Expand ad-hoc	YES	NO	Burst Buffers, FS	NO
UnifyFS	YES	NO	Main Memory, Burst Buffers	NO
CRUISE	YES	NO	Main Memory, Burst Buffers, FS	NO
BurstFS	YES	NO	Burst Buffers, FS	NO
EchoFS	YES	NO	Burst Buffers, FS	NO
FP runtime	YES	NO	Burst Buffers, FS	NO
CAPIO Runtime	YES	NO	Main Memory, FS	**YES**

Table 2.1: Comparison between ad-hoc file systems.

ing the initial workflow creation. CAPIO can then be applied to enhance performance by running the workflow applications in a coupled fashion. Table 2.1 shows the comparison between the ad-hoc file systems presented in subsection 2.4.3 and the CAPIO runtime.

At the best of our knowledge, no tools provide an I/O coordination model that enables the injection of streaming capabilitites into file-based token-pushing workflows.

The following chapters will present the CAPIO coordination language and how the CAPIO runtime works. Synthetic benchmarks and real workflow tests were conducted to demonstrate the effectiveness of the CAPIO approach.

Coord. Languages	**Coord. type**	**Comm. type**	**Comm. medium**	**Synch.**
Linda	Endogenous	Tuples Read/Write	Shared Tuple Space	Implicit
BSP model	Endogenous	Message passing	Network	Implicit
Actor model	Endogenous	Message passing	Network	Implicit
π-Calculus	Endogenous	Message passing	Named Channels	Explicit
Join Calculus	Endogenous	Message passing	Named Channels	Explicit
Reo	Exogenous	Message passing	Typed channels	Explicit
OpenMP	Exogenous	Memory access and Memory Transfers[6]	Shared Space and Channels	Explicit
OpenACC	Exogenous	Memory access and Memory Transfers	Shared Space and Channels	Explicit
Skeleton Based	Endogenous	Implicit	Implicit	Implicit
MPI	Endogenous	Message passing	Network	Explicit
POSIX threads	Endogenous	Memory access	Shared Space	Explicit
CUDA	Endogenous	Memory transfers	Channels	Explicit
UPC	Endogenous	Memory access	Shared Space	Explicit
CAPIO Lang.	Exogenous	Files Read/Write	Shared Space	Explicit

Table 2.2: Comparison between coordination languages.

[6]For GPU offloading.

Chapter 3

CAPIO: Goals and features

This chapter introduces the motivations, goals, and features of CAPIO while deferring technical and implementation details to subsequent chapters. It begins by discussing the context and the rationale behind the creation of CAPIO. Following this, the main features of CAPIO are outlined for clarity. Subsequently, an explanation is provided on how CAPIO enhances the I/O performance of workflows. Finally, a high-level introduction, omitting syntax details, is given for both the CAPIO language and the CAPIO runtime.

3.1 Motivations

In HPC systems, I/O often poses a bottleneck for data-intensive workflows. While distributed file systems offer parallel I/O capabilities, certain use cases can exacerbate this issue. For instance, accessing a large number of files or performing multiple small random read/write operations can lead to slowdowns [99] and even to denial of service [130]. As discussed in chapter 2, various solutions have been developed to mitigate these problems.

In-situ workflows, where producers and consumers can operate concurrently and exchange data as soon as it becomes available, were originally conceived for scientific simulations. Traditionally, simulations would first produce output files, which would then be analyzed by post-processing applications. In contrast, in-situ workflows integrate simulation and post-processing to analyze data as soon as it is generated. This approach can be extended to any workflow where one application's output must be consumed by another.

Several libraries have been developed to enable streaming workflows.

However, many workflows still rely on POSIX I/O, requiring modification with new libraries to enable in-situ execution. This can be challenging, as workflows often comprise applications written in different technologies by different teams. Consequently, users may face difficulties adapting application code to use a new I/O API.

To address this challenge, CAPIO was developed. With CAPIO, users can transform classical workflows—those relying on files for inter-step data communication—into workflows where all steps can be executed concurrently without modifying the code. CAPIO achieves this by managing synchronizations between workflow steps, ensuring consumers can access desired data as soon as it becomes available.

CAPIO provides an I/O coordination language for writing configuration files that express data dependencies and desired synchronization semantics in the workflow. The CAPIO runtime then utilizes this configuration file to execute all workflow steps concurrently, enforcing user-specified synchronizations to guarantee correct workflow execution. With CAPIO, streaming workflows yield results identical to batch execution without CAPIO.

3.2 Key Features of CAPIO

Outlined below are the main features of CAPIO, its objectives, and how it achieves them:

- CAPIO comprises an I/O coordination language and a runtime that implements it.
- CAPIO enhances I/O performance by transforming traditional workflows, which rely on file-based communication, into streaming workflows where all steps are executed concurrently.
- No modifications to the applications within the workflow are required.
- Users only need to provide the CAPIO runtime with a configuration file written in the CAPIO language expressing files synchronization semantics for streaming communication.
- The CAPIO language specifies the synchronization required for workflow steps to read and write files concurrently, even when originally programmed to do so sequentially.

3.3 From a batch execution into a streaming execution

The growing volume of digital data for analysis and simulation is leading to an inevitable expansion of I/O-bound HPC workflows. Despite recent advancements in high-speed storage technologies, a significant gap persists in latency and bandwidth between computing, memory, and storage [131].

Examining a two-step pipeline workflow depicted in Fig. 3.1, Step S generates k files, serving as input tokens for the subsequent step Q. These files are stored on a shared file system with write and read bandwidths (wB and rB), dependent on file size. Typically, bandwidth has an upper limit and decreases non-linearly with file size. Assuming equal file sizes and constant bandwidth, if Step S writes files of size N and Step Q reads files of M bytes, the makespan (T_T) is influenced by compute time $T_C = T_C^S + T_C^Q$ and total I/O time $T_{I/O} = T_{I/O}^S + T_{I/O}^Q$, such that:

$$\max\left(T_C, T_{I/O}\right) \leq T_T \leq T_C + T_{I/O} \tag{3.1}$$

$T_{I/O}$ represents the overall time dedicated to generating and consuming tokens within the workflow model. This can be articulated as follows:

$$T_{I/O} = k \cdot \left(\frac{N}{wB} + \frac{M}{rB}\right) \tag{3.2}$$

Given Eq. (3.2), two primary categories of techniques addressing the impact of I/O operations are identified in the literature. The first category aims to maximize wB and rB, with technologies like Burst buffers [9] and ad hoc file systems [10] relying on high-end storage technologies (e.g., SSD or NVMe) and intermediate storage in memory to enhance I/O bandwidth [132]. These solutions directly transfer data among HPC nodes, alleviating bandwidth contention on shared file systems. In contrast, the second category, represented by parallel I/O interfaces (e.g., OrangeFS/PVFS [133], MPI-IO [11]), seeks to optimize available I/O bandwidth by enabling multiple processes to concurrently read/write different sections of a file. Libraries such as HDF5 [103] and ADIOS [14] enhance I/O in HPC scientific applications, offering programmers higher-level storage management APIs built on various I/O backends.

Performance challenges in parallel file systems like Lustre [37] or GPFS [38] are attributed to adherence to POSIX semantics, necessitating atomicity

for multiple operations. Some parallel file systems, like UnifyFS [134] and GekkoFS [95], adopt more relaxed semantics to boost performance. However, ensuring portability among file systems with relaxed semantics remains a significant challenge [135].

The second category of I/O optimization strategies focuses on minimizing the numerator in Eq. (3.2), and it includes the in-situ/in-transit data processing model within this category [16]. For instance, performing on-the-fly compression and decompression during I/O can concurrently reduce both N and M. Format conversion proves beneficial when the data format consumed by Q differs from that produced by S. It's worth noting that while in-situ processing can decrease $T_{I/O}$, it may also lead to an increase in T_C. Some I/O libraries, like Damaris [13], allocate certain cores of an HPC worker node for asynchronous I/O operations and processing. These optimizations aim to reduce $T_C + T_{I/O}$ by enhancing the overlap between computation and I/O, preparing data for subsequent steps. However, this might diminish available computation parallelism, potentially increasing T_C. Nonetheless, in I/O-bound applications, the acceptable trade-off is increasing T_C to achieve a reduction in $T_{I/O}$.

Another approach for optimizing I/O in data-intensive workflows involves enhancing the workflow model with I/O streaming behavior to overlap I/O and computation between successive steps. As discussed in subsection 2.5, the CAPIO middleware aims to achieve this transparently. To the best of our knowledge, no other tools adopt this approach using a declarative I/O coordination model.

3.3.1 The CAPIO Approach

Let's reconsider the two-step pipeline in Fig. 3.1. An effective technique to reduce $T_{I/O}$ is to overlap the I/O phases of the two stages. In the ideal scenario of complete overlap, Eq. (3.2) can be reformulated as follows:

$$T_{I/O} \approx \max\left(T^S_{I/O}, T^Q_{I/O}\right) = \max\left(\frac{kN}{wB}, \frac{kM}{rB}\right) \tag{3.3}$$

The challenge lies in integrating such streaming optimizations into the workflow model without altering the business code of the involved steps. This involves reinterpreting the semantics of existing file system primitives rather than replacing or modifying them, with a crucial requirement: preserving correctness.

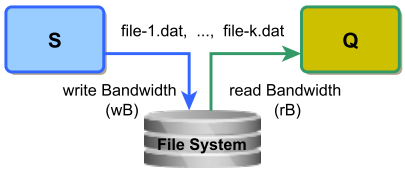

Figure 3.1: A two steps (S and Q) workflow whose tokens are files.

The CAPIO (Cross-Application Programmable I/O) user-space I/O middleware aims to facilitate these optimizations by a) adopting concurrent execution, as opposed to batch execution, for workflow steps requiring I/O optimization; and b) allowing more relaxed synchronization semantics for token propagation compared to the standard "on-termination" approach (refer to section 2.3). Concurrent execution of workflow steps provides the opportunity to exploit temporal parallelism (i.e., pipelining).

Regarding the second point, the objective is to define appropriate I/O synchronization semantics that allow anticipation of consumer operations on a given file without introducing side effects on program execution. To clarify, let's informally define the"on-close" file commit semantics: consumers of a file f may begin reading its content only after all producers of data in f have closed the file. This semantics conveys the information that a) the file is ready to be read (i.e., the corresponding token is "fireable") as soon as the file is closed by all producers; b) the file is considered completed (i.e., "committed" to the file storage) when it is closed by all producers. Additionally, it implicitly asserts that the producers will not reopen the file.

The "*on-close file commit*" semantics are more relaxed compared to the standard *on-termination* semantics, which necessitates that all producer steps terminate before the consumer steps can open the file and commence reading its content—both points *a)* and *b)* are tied to the termination of producer steps. Furthermore, this relaxed semantics facilitates the temporal overlap of (at least) distinct I/O phases between two consecutive steps. For instance, if Q in our example can be executed under the new semantics, its `open` (and `read`) system calls to the f file can be blocked until all `close` system calls have been completed at S. Consequently, the writing of data into the file f_{i+1} by S can overlap (or partially overlap) with the reading of data from file f_i by Q. CAPIO can seamlessly enforce this behavior by intercepting POSIX system calls issued by S and Q, compelling them to execute in accordance with the user-provided file commit semantics. Figure 3.2 illustrates two possible

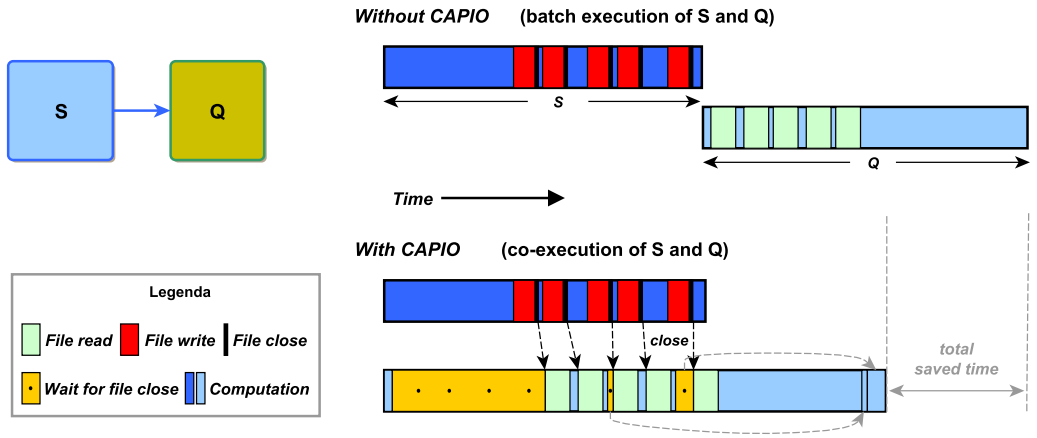

Figure 3.2: Batch execution of S and Q steps vs. their concurrent execution with CAPIO and the "*on-close file commit*" semantics.

executions of S and Q, with and without CAPIO.

To provide a preliminary estimate of the potential benefits offered by CAPIO optimizations, assuming the "*on-close file commit*" semantics, we can compare the total I/O time in the base workflow model (Eq. (3.2)) with the streaming workflow model enforced by CAPIO with k files. For the case where $N = M$ and $wB < rB$, the gain can be expressed as follows:

$$gain = \frac{T_{I/O} - (\frac{kN}{wB} + \frac{N}{rB})}{T_{I/O}} = \frac{wB(k-1)}{k(rB + wB)} \tag{3.4}$$

For example, if $k = 100$ and $rB = 2wB$, then $gain = 33\%$.

However, the gain in Eq. 3.4 only considers I/O phase overlap (i.e., data movement) without accounting for possible computation-I/O overlap, typical in pipeline computations (see Fig. 3.2). The extent of CAPIO optimizations, and thus the degree of different I/O phase overlap, strongly depends on how relaxed the file commit semantics in the workflow steps could be. A significant scenario is when producers write portions of files only once (e.g., *append-only* writes). In this case, the data transfer granularity between producer and consumer steps is not the entire file, as in the "*on-close file commit*" semantics, but could be as small as the data chunk of a single `read` system call in the consumer steps. Assuming the data chunk is not too small (which would increase the system call overhead), this further enhances overlap opportunities.

Additionally, the CAPIO I/O optimization approach is fully compatible with the optimizations described in section 3.3; therefore, all of them can be applied together. For instance, similar to main-memory storage system implementations (e.g., IMSS [132]), CAPIO stores intermediate data in memory to enhance I/O bandwidth and transfers data directly to consumer nodes by bypassing the distributed file system in the I/O path. Moreover, it may introduce *in-transit* data processing on the fly while moving files between producer-consumer steps, leveraging technologies like SmartNICs without involving host CPU cores, as proposed in [136].

Implementing all the mentioned optimizations and transitioning from a standard workflow model to a streaming model with relaxed file commit semantics, without modifying the host code, presents a challenge. Traditional I/O operation APIs (e.g., POSIX, MPI-IO, HDF5 Async API) are only partially suitable for streaming as they lack coordination/synchronization information at the application level. Consequently, the user must provide additional non-functional information to introduce enhanced I/O optimizations. CAPIO addresses this challenge by utilizing a configuration file in JSON format. The CAPIO middleware, as detailed in section 5.2.1, utilizes this information to seamlessly enforce streaming data movement optimizations between producer-consumer steps without modifying the business code and without altering the functional semantics.

3.4 Commit and Firing Rules

To define the file synchronization semantics between consecutive workflow steps, we need to consider two temporal aspects: *a)* when there are no more updates to the file; *b)* when one consumer can safely start reading a portion of data written in the file. We term the first aspect the *commit rule* and the second aspect the *firing rule*. Drawing on terminology from the realm of Data Stream Processing [137], the *firing rule* specifies when the data items (also known as *tuples* or *stream elements*) commence production by source operators in the data flow graph, allowing consumer operators to begin consumption. On the other hand, the *commit rule* determines when a given data stream concludes—i.e., all consumer operators in the streaming data flow graph receive the *end-of-stream* token, indicating that there will be no more data in input.

The *commit rule* allows us to define two distinct file commit behaviors:

- "on-termination": This behavior is employed in the batch execution of workflow steps. When a step terminates, all data files produced are committed to the file system and become ready to be read by all consumer steps.

- "on-close": This behavior enables the consumer to start reading a file as soon as I/O operations from all producers on that specific file are completed. The completion of these I/O operations is signaled by the close system call from each producer.

In addition to these two primary file commit behaviors, we can consider a file committed when another file is committed. This is particularly useful when the number of open and close operations for a given file is not known statically, but we are aware that the I/O operations are completed if another file is committed. These additional commit behaviors create a dependency among files, expanding the opportunities to exploit temporal parallelism for I/O operations in distinct workflow steps.

We defined the following set of file commit semantics currently supported by the CAPIO middleware. Specifically, the producer(s) of a `fileX` may declare that the file is:

- "*Commit on-Termination*" (CoT): the `fileX` is completed *iff* all producers have terminated

- "*Commit on-Close*" (CoC): the `fileX` is completed *iff* all producers have closed the file, and they do not re-open it

- "*Commit on-File*" (CoF): the commit semantics of `fileX` depends on the commit semantics of another file (i.e., `fileY`)

Regarding the *firing rule*, it specifies when data is ready to be consumed. If the commit rule holds for a given file, the file is unquestionably ready to be consumed; in other words, the commit rule implies the firing rule for the entire file. We refer to this fundamental firing rule as "*Firing on-Commit* (*FoC*). However, portions of the file (i.e., those parts already written by the producers) could be immediately ready to be consumed (i.e., fireable), *provided the producers do not update them.* This behavior is termed "*Firing no-Update*" (*FnU*), signifying that the file content is ready to be read as soon as data is written into the file.

When a consumer initiates a `read` system call (SC) on a portion of the file that has not been written by producers yet, the behavior differs between the *Commit on-Termination*, *Firing no-Update* (*CoT-FnU*) semantics and the *Commit on-Close*, *Firing no-Update* (*CoC-FnU*) semantics. In both cases, CAPIO blocks the consumer until:

- The requested data is produced, and the read returns the total number of bytes requested.
- All producers close the file (for *CoC-FnU*), or terminate (for *CoT-FnU*), and the read returns the current number of bytes read, if any, or 0 to indicate the end of the file (EOF).

3.5 The CAPIO Language and its runtime

To leverage CAPIO's capabilities, users simply need to create a configuration file using the CAPIO language, specifying the synchronization requirements for the files utilized in communication between two workflow steps. Various synchronization semantics are outlined in the preceding section. Once the configuration file is authored, it is provided to a runtime capable of concurrently launching all workflow steps and ensuring adherence to the synchronization semantics delineated in the configuration file. No modifications to the original applications comprising the workflows are necessary, nor is there a need for recompilation. Chapter 4 provides a detailed explanation of the CAPIO language syntax, utilizing JSON. Additionally, future iterations may support alternative formats such as YAML and XML. With the CAPIO language, users provide at least the following information regarding a given workflow:

- The applications comprising the workflow.
- The data dependencies among the applications, specifying which files are written to and read by each application.
- The synchronization semantics for files used in communication, including:
  - The Commit rule: defining when a file is considered complete.

  - The Firing rule: defining when a process can begin accessing the file content.

Furthermore, additional information for enhanced optimization can be provided, which will be detailed in the following chapter. It is crucial to recognize that the CAPIO language and the runtime are two distinct components: there could exist multiple types of runtimes that implement the CAPIO Language. This study provides a runtime described in Chapter 5, used for experiments with synthetic benchmarks and real-world use cases explained in Chapter 6. While different runtimes supporting the CAPIO language may exist, they must possess certain key features to be effective in an HPC system:

- They must be capable of imposing the CAPIO synchronization semantics without the need to modify the original applications.
- They should be lightweight in terms of CPU usage, memory, and network usage. Since the CAPIO runtime shares resources with the workflow applications, it should not introduce excessive overhead.
- They must be able to leverage parallelism using the nodes employed for the workflow, thus avoiding bottlenecks.

The CAPIO runtime provided in this study is implemented as an ad-hoc file system based on MPI. In the future, it will offer the option to choose different backends from MPI. However, it is also possible to implement the runtime as part of a distributed file system or as part of a workflow management system.

Chapter 4

CAPIO Coordination Language

This chapter introduces the CAPIO language, a pioneering I/O coordination language crafted to annotate date dependencies within file-based workflows with synchronization semantics. Its primary aim is to seamlessly integrate computational and I/O operations across disparate producer-consumer application modules. Notably, the language operates independently from the runtime engine, allowing for the development of multiple runtimes tailored to its specifications. The user can utilize this language to compose a configuration file, instructing the runtime to execute all workflow modules concurrently, thus leveraging in-situ capabilities. Chapter 5 provides insight into the initial prototype of such a runtime.

4.1 Syntax

In this section the syntax of the CAPIO language is presented enabling the user to write the data dependencies of a workflow and the streaming semantics for in-situ computations. For each features an example is shown.

JSON (JavaScript Object Notation) serves as the language for expressing the syntax and semantics of the I/O coordination language. JSON is language-agnostic, enjoying widespread support in various programming languages such as Java, C++, Python, and more. While not the most commonly utilized language for high-level coordination languages in parallel computations, JSON offers automatic syntax validation through its schema. This feature allows a focus on the semantic aspects of the I/O coordination language. Recognized for its simplicity, flexibility, and expressivity, JSON is

well-established in various computer science domains. Choosing JSON as the foundational syntax for the CAPIO coordination language outlined in this document aims to provide users with a smoother learning curve.

In this section, we present the syntax of the I/O coordination language, which is articulated in JSON format. The JSON syntax relies on two fundamental structures: objects and arrays. An object is a grouping of key/value pairs, with the key typically being a string identifier, often represented by a mnemonic name. An array is a sequentially ordered list of values.

The JSON file adhering to the I/O coordination language syntax will be denoted as the CAPIO configuration file. This file serves as a crucial component for the CAPIO middleware, ensuring the enforcement of synchronization semantics for files across successive workflow modules.

The CAPIO configuration file encompasses six sections:

- Workflow Name: Identifies a workflow, which consists of multiple application modules.
- IO_Graph: Describes file data dependencies among application modules.
- Aliases: Groups a set of files or directories under a convenient name.
- Permanent: Specifies files to be retained in permanent storage at the end of the workflow execution.
- Exclude: Identifies files and directories not managed by CAPIO.
- Home-node Policy: Defines various file mapping policies to determine which CAPIO servers store specific files.

We will describe the syntax of the language related to each section in the subsequent parts. As the CAPIO language revolves around I/O objects—files and directories—it supports wildcards. Wildcards are special characters used to represent unknown characters in a text, providing a convenient way to specify file and directory names without enumerating all the files or directories that an application might produce or read (e.g., file*.dat). Presently, the language accommodates two wildcards: 1) * which matches any sequence of characters of length >= 0, and 2) ? which matches a single character. Wildcards can be applied in all values where a file name or a directory name is expected.

4.1.1 Workflow name section

This section is denoted by the keyword "name" (refer to Listing 4.1). The assigned name serves as a unique identifier for the current application workflow. This distinction is valuable for recognizing and differentiating between multiple application workflows concurrently executing on the same machine.

Listing 4.1: The workflow name

```
{
   "name" :"my_workflow",
   ...
}
```

4.1.2 IO-Graph section

This section delineates the relationships between input and output streams among the application modules constituting the workflow. Recognized by the keyword "IO_Graph", it necessitates an array of objects, each specifying input and output streams for an individual application component. Each object encompasses the following components:

- name: The application's name; this keyword is mandatory.
- input_stream: Identifies the input files and directories the application module is anticipated to read. It is optional and takes a vector of strings as its value.
- output_stream: Specifies a vector of file and directory names produced by the application module. It is optional and accepts a vector of strings as its value.
- streaming: An optional keyword designating files and directories with associated streaming semantics, defining *commit* and *firing* rules (details described in section 3.4). Its value is an array of objects. Each object may include the following attributes:
  - name: Filenames to which the rule applies, with values as an array of filenames.
  - dirname: Directory names to which the rule applies, with values as an array of directory names.

- committed: Defines the *commit rule* for files or directories identified with the keywords name and dirname, respectively. Acceptable values include "on_close:N" (where N is an integer >= 1), on_termination, or on_file if the *commit rule* applies to filenames. If the *commit rule* pertains to directory names, valid values are "on_termination", "on_file", or "n_files:N" (where N is an integer >= 1). If committed is unspecified, the default *commit rule* is "on_termination". For "on_file" semantics, the "files_deps" keyword, with values as an array of filenames or directory names, defines the set of dependencies.
- mode: Defines the *firing rule* associated with files and directories identified by the name and dirname keys, respectively. Valid values include "update" or "no_update". If mode is not specified, the default *firing rule* is "update".

Listing 4.2 provides a valid IO_Graph section featuring two application modules, namely "writer" and "reader", within the "my_workflow" workflow. The "writer" module generates three output files (file0.dat, file1.dat, and file2.dat) and a directory (dir). Each file is linked to unique streaming semantics, representing distinct *commit* and *firing* rules. The "reader" module reads all files produced by the "writer" module.

Listing 4.2: The I/O dependency graph

```
{
   "name" :"my_workflow",
   "IO_Graph": [
     {
       "name": "writer",
       "output_stream": ["file0.dat","file1.dat","file2.dat","dir"],
       "streaming": [
           {
               "name": ["file0.dat" ],
               "committed": "on_termination",
               "mode": "update"
           },
           {
               "name": ["file1.dat" ],
               "committed": "on_close",
               "mode": "update"
           },
           {
               "name": ["file2.dat" ],
               "committed": "on_close:10",
               "mode": "no_update"
           },
           {
```

```
            "dirname": ["dir" ],
            "committed": "n_files:1000",
            "mode": "no_update"
          }
      ]
    },
    {
      "name": "reader",
      "input_stream": ["file0.dat", "file1.dat", "file2.dat", "dir"]
    }
  ]
  ...
}
```

4.1.3 Aliases section

The aliases section, identified by the keyword "aliases", serves to mitigate the verbosity associated with enumerating files that an application may consume or produce. It consists of a vector of objects, each comprising the following items:

- "group_name": the alias identifier
- "files": an array of strings representing file names.

In Listing 4.3, an example illustrates how to define aliases for disjoint sets of file names.

Listing 4.3: How to define aliaes

```
{
   "name" :"my_workflow",
   "aliases" :[
     {
       "group_name" :"group-even",
       "files" :["file0.dat", "file2.dat", "file4.dat"]
     },
     {
       "group_name" :"group-odd",
       "files" :["file1.dat", "file3.dat", "file5.dat"]
     }
  ]
  ...
}
```

4.1.4 Permanent section

This language section, identified by the keyword "permanent", serves to designate files that will be preserved on the filesystem at the conclusion of the

workflow execution. It entails an array of file names (which may also include aliases). Listing 4.4 provides a rough example illustrating how to define the "permanent" section. Following the execution of "my_workflow", both the file "output.dat" and all files affiliated with "group0" will be retained in the filesystem.

Listing 4.4: How to define which files will be stored into the filesystem at the end of workflow execution.

```
{
    "name" :"my_workflow",
    "permanent" :["output.dat", "group0" ]
   ...
}
```

4.1.5 Exclude section

This section, identified by the keyword "exclude" is employed to specify files that will not be managed by CAPIO, even if they are generated within the CAPIO_DIR directory. It involves an array of file names (which can also include aliases).

Listing 4.5: How to define aliaes.

```
{
    "name" :"my_workflow",
    "exclude" :[ "file1.dat", "dir", "*.txt" ]
   ...
}
```

Listing 4.5 provides an example of defining the "exclude" section. CAPIO will disregard file1.dat, all files and directories within the directory "dir", and all temporary files concluding with ".txt".

4.1.6 Home-node policy section

In Listing 4.6, the syntax is presented to enable the CAPIO user to selectively designate the CAPIO server node for storing files and their associated metadata. Different policies can be defined for distinct files. In the current version of the CAPIO language, the home node policy options are "create". "manual", and "hashing". These three keywords are optional, allowing users to omit them in the CAPIO configuration file, in which case the default policy is "create". Additionally, there should be no filename overlap among the

specified policies, meaning the intersection of the sets defined by different home-node policies must be empty.

For the "hashing" and "create" policies, the value is an array of files. In the case of the "manual" configuration, the syntax is more elaborate, necessitating the definition of the logical identifier for each file or set of files. This identifier corresponds to the application process whose associated CAPIO server will store the file. This information enables each CAPIO server to statically determine the file-to-node mapping and retrieve the node where the process is executing at runtime. For a detailed explanation of the semantics of all home node policies, please refer to Section (4.3).

Listing 4.6: Home node policies.

```
{
   "name" :"my_workflow",
   "IO_Graph": [
     {
       "name": "writer",
       "output_stream": ["file*.dat" ],
       "streaming": [
           {
               "name": ["file*.dat" ],
               "committed": "on_close"
           }
       ]
     },
     {
       "name": "reader",
       "input_stream": ["file*.dat" ]
     }
   ],
   "home-node-policy": {
       "create": ["file0.dat", "file1.dat"],
       "manual": [
           {
               "name" :[ "file2.dat", "file3.dat" ],
               "app_node": "writer:0"
           },
           {
               "name": ["file4.dat", "file5.dat" ],
               "app_node": "writer:1"
           }
       ],
       "hashing": ["file6.dat", "file7.dat"]
   }
}
```

4.1.7 Wildcards potential ambiguity

The CAPIO language syntax incorporates wildcards to offer users flexibility and alleviate the need to list every file and directory explicitly. However, employing wildcards in the language syntax may introduce unexpected behavior, such as unintended matches or undefined behavior due to multiple matches associated with different semantic rules. To illustrate this, let's examine the example provided in Listing 4.7.

Listing 4.7: Example of ambiguity arising when using wildcards.

```
{
   "name" :"my_workflow",
   "IO_Graph": [
     {
       "name": "writer",
       "output_stream": ["file1.txt", "file2.txt", "file1.dat", "file2.dat"],
       "streaming": [
           {
               "name": ["file*" ],
               "committed": "on_close"
           },
           {
               "name": ["*.dat" ],
               "committed": "on_termination"
           }
       ]
     },
     ...
   ]
  ...
}
```

In the given example, there is an overlapping match for the files file1.dat and file2.dat. This creates ambiguity regarding whether the commit semantics should be "on_close" or "on_termination". While, in most cases, ambiguity can be resolved by considering the most specific match for the rules in context (e.g., ".dat" is more specific than "file" when considering the user-specified list of files in the output_stream), in the current version of CAPIO, all ambiguities remain unresolved. Consequently, the CAPIO runtime raises an exception for undefined behavior. In future releases, we aim to enhance flexibility by relaxing such constraints and automatically disambiguating the syntax expression when feasible.

It is noteworthy that, in addition to using wildcards judiciously, effective utilization of the aliases section can assist the user in creating a clear and unambiguous configuration file. For instance, in Listing 4.8, two separate aliases are defined to delineate two distinct groups of files, allowing the

application of different semantics rules to each group.

Listing 4.8: Example of using aliases to avoid ambiguity.

```
{
   "name" :"my_workflow",
   "aliases" :[
     {
       "group_name" :"group-dat",
       "files" :["file1.dat", "file2.dat"]
     },
     {
       "group_name" :"group-txt",
       "files" :["file1.txt", "file2.txt"]
     }
   ],
   "IO_Graph": [
     {
       "name": "writer",
       "output_stream": ["group-dat", "group-txt"],
       "streaming": [
           {
               "name": ["group-txt"],
               "committed": "on_close"
           },
           {
               "name": ["group-dat"],
               "committed": "on_termination"
           }
       ]
     },
     ...
   ]
  ...
}
```

Finally, it is worth mentioning that the CAPIO configuration file could be directly generated by the WMS, describing the entire application workflow. This is particularly useful for complex workflows. Specifically, the part related to I/O data dependencies (i.e., the IO-Graph) can be entirely generated starting from the application description. Conversely, the synchronization semantics of tokens require explicit annotations at the workflow description language level. For example, the `streamable` keyword in the CWL standard indicates that a given file is read or written sequentially without seeking [138].

4.2 Streaming semantics

We will now illustrate how to articulate the commit and firing semantics using the CAPIO coordination language, offering straightforward examples. For the ensuing instances, we contemplate a workflow consisting of two applications: a writer application, which generates two files, and a reader application, responsible for reading these two files. Algorithm 1 delineates the general case of the writer application, where it produces N files. In each iteration of the main loop, it engages in computational tasks, followed by writing the output of this computation to a distinct file. The reader application, as depicted in Algorithm 2, exhibits a similar pattern. In each iteration, it reads a file produced by the writer application, utilizing the read data to execute a specific computation for a designated duration.

Algorithm 1: A simple file writer application.

Data: $n\_files \geq 0$; $file\_size > 0$; $secs \geq 0$
$N \leftarrow 0$;
while $N < n\_files$ **do**
    buffer ← compute(secs, file_size);
    write_file(buffer, "fileN.dat", file_size);
    $N \leftarrow N + 1$;
end

Algorithm 2: A simple file reader application.

Data: $n\_files \geq 0$; $file\_size > 0$; $secs \geq 0$
$N \leftarrow 0$;
while $N < n\_files$ **do**
    buffer ← read_file("fileN.dat", file_size);
    compute(secs, buffer, file_size);
    $N \leftarrow N + 1$;
end

Usually, this uncomplicated workflow is executed in a conventional batch mode. Initially, the writer application is initiated to generate input files for the reader application. Subsequently, the reader application can commence its execution, consuming the files by reading them from the filesystem. The

CAPIO middleware facilitates the concurrent execution of both applications without necessitating modifications to the code of the two modules. For this seamless integration, CAPIO requires information about the specific data streaming semantics (as elucidated in earlier sections) it should enforce on the produced and consumed files, ensuring accurate execution.

4.2.1 Commit on Termination, Fire on Commit (CoT-FoC)

The more restrictive streaming capability semantic is conveyed through Commit on-Termination (CoT) paired with the Fire On-Commit (FoC) firing rule. This semantic dictates that, when applied to a file, the reader is allowed to commence reading the file only after the writer application has terminated. This proves advantageous when a section of the file can be updated multiple times, and there is uncertainty about when the writer will cease adding data records. In such cases, the correct behavior is to await the termination of the writer before reading. When CAPIO detects a read system call on a file with such stringent semantics, it will provide the data only upon the completion of the writer application. Despite the absence of ongoing streaming communication with the CoT-FoC semantics, concurrent execution of both applications can still be advantageous, particularly in scenarios where the reader must perform substantial computations before retrieving data from the writer application. In Listing 4.9, we present the configuration file that expresses the CoT-FoC semantics for the simple example under consideration.

Listing 4.9: Simple writer-reader pipeline with Commit on-Termination Firing on-Commit semantics.

```
{
  "name":"my_workflow",
  "IO_Graph":[
    {
      "name":"writer",
      "output_stream":["file1.dat", "file2.dat" ],
      "streaming":[
        {
          "name": ["file1.dat", "file2.dat" ],
          "committed":"on_termination",
          "mode":"update"
        }
      ]
    },
    {
      "name":"reader",
```

```
        "input_stream":["file1.dat","file2.dat"]
      }
   ]
}
```

4.2.2 Commit on Termination, Fire no Update (CoT-FnU)

In the following example, the workflow structure and its data dependencies remain unchanged; only the producer-consumer semantics are altered. From a syntax perspective, the only section requiring modification is the one related to the keyword "streaming". In Listing 4.10, the configuration file for the workflow with the Commit on-Termination (CoT), Firing no-Update (FnU) semantics is presented. With this semantic, the reader can initiate reading the files file1.dat and file2.dat as soon as the writer produces data into these files. The reader receives the End-of-Stream (EOS) signal upon reaching the end of the file and after the termination of the writer module. In this case, there are more opportunities for streaming communication than in the previous semantics.

This semantics is advantageous when the user knows that once a section of the file is written, it will not be modified, but they are uncertain about when the writer will stop writing data into the file. For the considered workflow, this semantics involves streaming only on the first file (i.e., file1.dat) because the writer and reader write/read files in a sequential manner (first file1.dat, then file2.dat, and so on). In this scenario, the reader will read the first file until both of these conditions are true:

1. It reaches the end of the file
2. writer terminates

When these two conditions are met, the CAPIO middleware will return the EOS signal to the reader, who will then proceed to read the second file.

Listing 4.10: Simple pipeline. Commit on-Termination Firing no-Update.

```
{
   "name":"my_workflow",
   "IO_Graph":[
      {
         "name":"writer",
         "output_stream":["file1.dat", "file2.dat"],
```

```
        "streaming":[
           {
              "name": ["file1.dat", "file2.dat" ],
              "committed":"on_termination",
              "mode":"no_update"
           },
        ]
     },
     {
        "name":"reader",
        "input_stream":["file1.dat","file2.dat"]
     }
  ]
}
```

4.2.3 Commit on Close, Fire on Commit (CoC-FoC)

Listing 4.11 presents the syntax for expressing the Committed on-Close (CoC), Firing On-Commit (FoC) semantics. This semantics enables the reader to commence reading a file after it is closed. The "update" mode, as described earlier, prevents the reader from accessing the data before the file is committed. In this scenario, the file is considered committed as soon as it is closed. This combination of semantics is beneficial when the writer updates a file multiple times, then ceases writing or updating the records and closes the file. In such cases, the CAPIO middleware makes the reader wait for the file's completion, and then it begins reading, even if the writer is still running. With this semantics, there is no streaming communication of file records but streaming at the granularity of the entire file.

Listing 4.11: Simple pipeline. Commit on-Close Firing on-Commit.

```
{
  "name":"my_workflow",
  "IO_Graph":[
     {
        "name":"writer",
        "output_stream":["file1.dat", "file2.dat"],
        "streaming" :[
           {
               "name" :[ "file1.dat" ],
               "committed" :"on_close",
               "mode" :"update"
           }
        ]
     },
     {
        "name":"reader",
        "input_stream":["file1.dat","file2.dat"]
     }
```

```
  ]
}
```

4.2.4 Commit on Close, Fire no Update (CoC-FnU)

The Committed on-Close (CoC), Firing no-Update (FnU) semantics, as defined in Listing 4.12, allows users to optimize streaming between producer and consumer workflow modules. In this scenario, the reader can start consuming data as soon as it is produced. Reading file records will cease when the reader reaches the end-of-file and the writer closes the file. If the reader reaches the end of the file and the writer has not closed it, the reader will wait until sufficient data is written to fulfill the read SC operation, or the writer closes the file. Upon the file closure by the writer, the reader receives the end-of-stream signal from the CAPIO middleware, and the read SC returns either fewer data or EOF.

Listing 4.12: Simple pipeline. Commit on-Close Firing no-Update.

```
{
  "name":"my_workflow",
  "IO_Graph":[
    {
      "name":"writer",
      "output_stream":["file1.dat", "file2.dat"],
      "streaming" :[
        {
            "name" :[ "file1.dat", "file2.dat" ],
            "committed" :"on_close",
            "mode" :"no_update"
        },
     ]
    },
    {
      "name":"reader",
      "input_stream":["file1.dat","file2.dat"]
    }
  ]
}
```

4.2.5 Commit on File, Fire update (CoF-FU)

The syntax for the Commit on-File (CoF), Firing no-update (FnU) semantics is depicted in Listing 4.13. In this scenario, file2.dat is considered committed when file1.dat is committed. In our example workflow, file1.dat is committed upon closure; therefore, file2.dat is committed with the on_close semantics,

and the firing semantics is set to update. As Firing update is the default policy, the keyword "mode" can be omitted or explicitly set to enforce the no_update semantics.

Listing 4.13: Simple pipeline. Commit on-File Firing no-Update.

```
{
   "name":"my_workflow",
   "IO_Graph":[
      {
         "name":"writer",
         "output_stream":["file1.dat", "file2.dat"],
         "streaming" :[
            {
                "name" :[ "file1.dat" ],
                "committed" :"on_close",
                "mode" :"no_update"
            },
            {
                "name" :[ "file2.dat" ],
                "committed" :"on_file",
                "files_deps": ["file1.dat" ],
                "mode" :"update"
            }
         ]
      },
      {
         "name":"reader",
         "input_stream":["file1.dat","file2.dat"]
      }
   ]
}
```

4.2.6 Streaming directory contents

Here we consider a slightly different workflow where streaming semantics is applied to a directory and its content. The writer application remains unchanged (see Algorithm 1). However, in this scenario, the reader does not have prior knowledge of the specific file names to read; instead, it only knows the name of the directory containing the files. Consequently, the reader iterates through the directory entries and reads all the files present in the directory. The pseudo-code for the reader is outlined in Algorithm 3. Listing 4.14 presents a potential configuration file for this particular application scenario. Streaming semantics can be extended to directories by utilizing the value n_files:N for the "committed" keyword. This value specifies the anticipated number of files to be produced within the directory. This optional

Algorithm 3: Reading all files in a directory.

Data: dir_path ; $secs \geq 0$
$F \leftarrow$ next_file(dir_path) ; `/* Returns a file in the directory */`
while $F \neq EOS$ **do**
    buffer $\leftarrow$ read_file(F);
    compute(secs, buffer);
    $F \leftarrow$ next_file(dir_path);
end

information allows the reader application to access and read the directory entries while the writer is still creating them. The CAPIO middleware, armed with knowledge about the total number of files expected in the directory, can determine the appropriate timing to signal the end-of-stream (EOS) to the reader.

It is crucial to note that this information becomes particularly valuable when the firing rule for the directory is set to "no_update", as exemplified in the provided example. If not explicitly specified otherwise, the default commit semantics for directories and files is "on_termination". This implies that the total number of files in the directory becomes known only when all producers writing to the directory have terminated. Setting the firing rule for a directory to "no_update" is appropriate only when the newly created directory entries (i.e., new files in the directory) are not subject to removal.

Listing 4.14: Simple pipeline with a directory.

```
{
"name" :"my_workflow",
"IO_Graph" :
   [
      {
         "name" :"writer",
         "output_stream" :["my_dir"],
         "streaming" :[
            {
               "dirname" :[ "my_dir" ],
               "committed" :"n_files:500",
               "mode" :"no_update"
            },
            {
               "name" :[ "my_dir/*" ],
               "committed" :"on_close",
               "mode" :"no_update"
            },
```

```
            ]
        },{
            "name" :"reader",
            "input_stream" :["my_dir"]
        }
    ]
}
```

4.3 Home-Node Policies

In this section, we delve into the features provided by the coordination languages to convey optimization hints to CAPIO's runtime system for the efficient placement of file data. Drawing inspiration from page-based software Distributed Shared-Memory implementations [139], we introduce the concept of a home-node within the CAPIO ecosystem. The home-node serves as the designated node for storing information related to the data and metadata of a specific set of files or directories.

As outlined in subsection (4.1.6), the CAPIO language offers three distinct policies for setting the home-node(s):

- create
- manual
- hashing

The user can specify one or more of these policies for a disjoint set of files and directories. Importantly, a given file or directory cannot have more than one home-node policy.

For a detailed understanding of the current implementation of home-node policies in the CAPIO middleware, please refer to subsection 5.2.2.

Create home-node policy

The create policy is the default setting for the CAPIO coordination language. This policy comes into effect when the home-node-policy language section is absent or when the policy-name object is explicitly set to create. If the home-node-policy language section is specified, but not all files have their home-node policy defined, the policy defaults to create for those files. The semantics of the create policy dictate that the home-node assigned is the CAPIO server where the file was initially created.

To illustrate, consider two CAPIO servers, C1 and C2, and a workflow module with two processes (P1 and P2) running on C1 and C2, respectively. If C1 creates the file "file.dat", the data and metadata of that file will be stored in the C1 server, automatically designating C1 as the home-node for the "file.dat" file.

Manual home-node policy

When the home-node policy keyword is set to manual, users have the flexibility to explicitly designate the home-node for individual or groups of files by specifying the home-node where a particular application module is executing. This is achieved by specifying the name of the workflow module used in the IO_Graph section.

For example, in Listing 4.15, the home-node for the files "file1.dat" and "file3.dat" is the node where the process of the writer application with logic id 0 will be executed. Similarly, for the files "file2.dat" and "file4.dat", the home-node is the writer's process with logic id 1. In cases where the application is single-process, setting the logic id is unnecessary, and the module name alone is sufficient.

The logic id serves as a unique identifier (integer type) for identifying a process of an application module. In the JSON file, it must be set according to the syntax: module_name:id. The id is the number passed through the environmental variable CAPIO_APP_NAME to the CAPIO runtime when launching the application. For example, assuming two CAPIO servers, C1 and C2, and the writer application module executed with two processes, the process of the writer module running on node C1 will have CAPIO_APP_NAME="writer:0" as the environmental variable, and the writer module process on node C2 will have CAPIO_APP_NAME="writer:1".

Listing 4.15: Manual home-node policy example.

```
{
   "name" :"my_workflow",
   "IO_Graph": [
     {
       "name": "writer",
       "output_stream": ["file*.dat" ],
       "streaming": [
           {
               "name": ["file*.dat" ],
               "committed": "on_close"
           }
       ]
     },
```

```
    {
      "name": "reader",
      "input_stream": ["file*.dat" ]
    }
  ],
  "home-node-policy": {
      "manual": [
          {
              "name" :[ "file1.dat", "file3.dat" ],
              "app_node": "writer:0"
          },
          {
              "name": ["file2.dat", "file4.dat" ],
              "app_node": "writer:1"
          }
      ],
  }
}
```

Hashing home-node policy

Another home-node policy supported by the CAPIO language is hashing. The reference node for a given file is determined by hashing its pathname modulo the number of CAPIO nodes running in the given workflow. For instance, the logical id of the home-node for the file "file.dat" in a workflow with 4 CAPIO servers is calculated as `hash-function("$CAPIO_DIR/file.dat")%4`. The hash-function is consistent across all CAPIO servers and can be any effective string hashing function, such as the std::hash method in modern C++.

4.3.1 More complex examples

In this subsection, more complex examples will be explored to show the potential of CAPIO's features and how the user can exploit them.

Data dependencies

In the previous examples, we focused on simple pipelines consisting of a producer and a consumer. However, the IO-Graph section of the configuration file allows the representation of any Directed Acyclic Graph (DAG) that captures a workflow along with its data dependencies. In this context, we present an example that involves multiple producers and consumers. Consider a workflow depicted in Figure 4.1, consisting of an Application S producing files required by three applications: W, X, and Z. Each of these applications

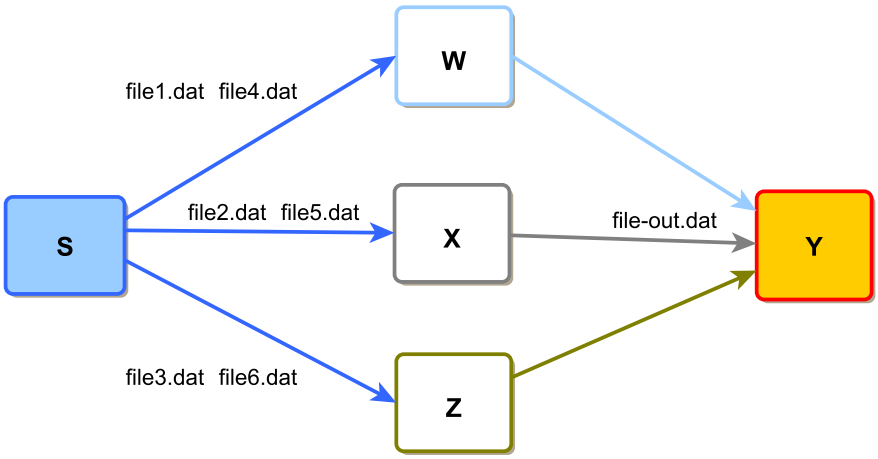

Figure 4.1: Example workflow showing files dependencies.

reads distinct files, processes the data, and writes the results to the same file, "file-out.dat". Each application contributes a unique portion to this file, which is then read by the final application, Y. Listing 4.17 illustrates how to articulate the IO-Graph using the CAPIO coordination language.

Listing 4.16: A more complex workflow.

```
{
"name" :"my_workflow",
"IO_Graph" :
    [
        {
            "name" :"S",
            "output_stream" :["file*.dat"]
        },
        {
            "name" :"W",
            "input_stream" :["file1.dat", "file4.dat"],
            "output_stream": :["file-out.dat"]
        },
        {
            "name" :"X",
            "input_stream" :["file2.dat", "file5.dat"],
            "output_stream" :["file-out.dat"]
        },
        {
            "name" :"Z",
            "input_stream" :["file3.dat", "file6.dat"],
            "output_stream": :["file-out.dat"]
        },
        {
            "name" :"Y",
            "input_stream" :["file-out.dat"]
```

```
        }
    ]
}
```

Usage example of the home-node policies

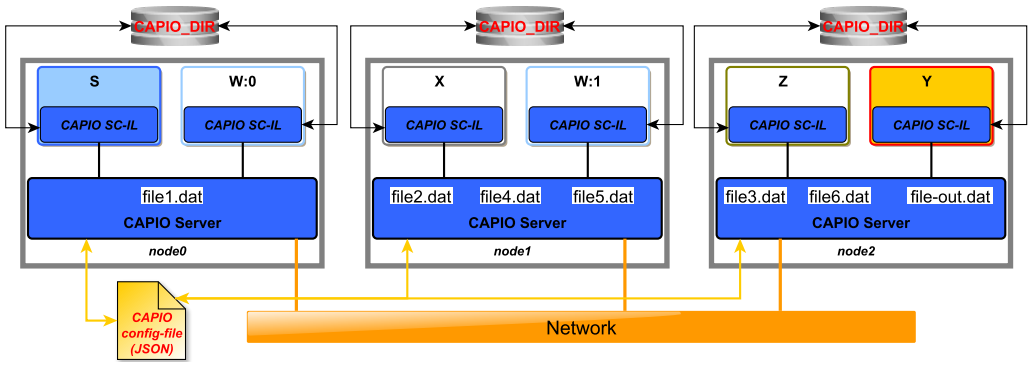

Figure 4.2: One possible deployment for the example workflow in Figure 4.1.

Here we present an example where the judicious use of home-node policies can significantly enhance the performance of a workflow. Figure 4.2 illustrates a potential deployment of the workflow depicted in Figure 4.1. In this instance, the workflow modules are distributed across three nodes, with application module S running on node0, modules X and V on node1, and modules Z and Y on node2. The workflow module W is the only multi-process application, with the process having logic id 0 running on node0, and the process with logic id 1 on node1. Listing 4.17 outlines the CAPIO configuration file with home-node policies configured to minimize the number of communications between producer-consumer nodes. The strategy is to set the home-node as the node where the process that needs to read the file is running. In this presented configuration file, we utilize manual configuration for home-nodes, enabling the enforcement of a specific mapping of files to nodes. In more intricate workflows with complex file dependencies, where files are written/read by multiple processes, the hashing policy can be a simpler solution for the user.

Listing 4.17: Files to nodes mapping using the *create* and `manual` policies for the workflow in Fig. 4.1.

```
{
```

```
"name" :"my_workflow",
"IO_Graph" :
   [
       {
           "name" :"S",
           "output_stream" :["file*.dat"]
       },
       {
           "name" :"W",
           "input_stream" :["file1.dat", "file4.dat"],
           "output_stream": :["file-out.dat"]
       },
       {
           "name" :"X",
           "input_stream" :["file2.dat", "file5.dat"],
           "output_stream" :["file-out.dat"]
       },
       {
           "name" :"Z",
           "input_stream" :["file3.dat", "file6.dat"],
           "output_stream": :["file-out.dat"]
       },
       {
           "name" :"Y",
           "input_stream" :["file-out.dat"]
       }
   ],
   "home_node_policy": {
       "create" :["file1.dat"],
       "manual" :
           [
               {
                   "name": ["file2.dat", "file4.dat"],
                   "app_node": "X"
               },
               {
                        "name": ["file5.dat"],
                        "app_node": "W:1"
               },
               {
                   "name": ["file3.dat", "file6.dat"],
                   "app_node": "Z"
               },
               {
                   "name": ["file-out.dat"],
                   "app_node": "Y"
               }
           ]
   }
}
```

In HPC clusters, users are typically unaware of the nodes where applications will be executed. By utilizing logical names for workflow application modules and logical ids for the processes within a single application module, users can define data placement without detailed knowledge of the actual

node configuration. Moreover, the same configuration file can be applied in different deployments, as the home-node policy relies on applications rather than the specific physical resources. It's important to note that the create policy is essentially syntactic sugar designed for user convenience. The same data placement achieved with the create policy can be expressed using the manual policy with more user effort. Both the create and hashing policies are well-suited when users lack deep insights into the workflow and its deployment. The create policy offers the advantage of fast file writing by a single process, storing it in the memory where the file is created. However, this approach may lead to uneven file creation among processes or application modules, potentially causing memory imbalances across nodes. In cases with too many files stored on a single node, it may become a bottleneck, especially with a high number of reads from different nodes. On the other hand, the hashing policy helps overcome these issues by balancing data across all nodes in the workflow. However, writing a file may be slower as the data might be placed on a "distant" node. In the worst-case scenario, the hashing policy might result in a situation where a file is stored in neither the producer nor the consumer of the file, thus slowing down both read and write operations. The create policy has been chosen as the default option due to its user-friendly nature. It does not require users to explicitly state the relationship between application modules and files produced or read. It is also straightforward to implement by collecting the files-to-home-node mapping in a database shared by all CAPIO servers, as discussed in subsection 5.2.2. The current CAPIO middleware implements the home-node database as a set of files stored in the cluster filesystem.

4.4 JSON Schema of the I/O coordination language

Here we report the JSON schema of the I/O coordination language implemented in the CAPIO middleware.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "name": {
      "description": "Name of the workflow",
      "type": "string"
    },
```

```
  "aliases": {"$ref": "#/$defs/aliases" },
  "IO_Graph": {
    "description": "Representation of data dependencies between applications and streaming
        semantics",
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "name": {"$ref": "#/$defs/application_name" },
        "input_stream": {
          "description": "Files read by the application",
          "$ref": "#/$defs/list_of_files"
        },
        "output_stream": {
          "description": "Files produced by the application",
          "$ref": "#/$defs/list_of_files"
        },
        "streaming": {
          "description": "Streaming semantics of files produced by the application",
          "$ref": "#/$defs/streaming"
        }
      },
      "required": ["name"],
      "dependentRequired": {
        "streaming": ["output_stream"]
      }
    }
  },
  "permanent": {
    "description": "Files that will be stored in the filesystem at the end of the workflow
        execution",
    "$ref": "#/$defs/list_of_files"
  },
  "exclude": {
    "description": "Files that will not be handled by CAPIO even if they are in the CAPIODIR
        ",
    "$ref": "#/$defs/list_of_files"
  },
  "home_node_policy": {"$ref": "#/$defs/home_node_policy" }
},
  "required": ["name", "IO_Graph"],

  "$defs": {

    "aliases": {
      "description": "Defines aliases for groups of files",
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "group_name": {"type": "string" },
          "files": {"$ref": "#/$defs/list_of_files" }
        }
      }
    },

    "application_name": {
```

```
63        "description": "Name of the application",
64        "type": "string"
65      },
66
67      "list_of_files": {
68        "type": "array",
69        "items": {
70          "type": "string"
71        },
72        "minItems": 1
73      },
74
75      "streaming": {
76        "description": "Streaming semantics of files produced by the application",
77        "type" :"array",
78        "items": {
79          "type": "object",
80          "oneOf" :[
81          {
82            "properties": {
83              "name": {"$ref": "#/$defs/list_of_files" },
84              "committed" :{
85                "description": "Commit rule",
86                "type" :"string",
87                "pattern": "^(on_termination)$|^(on_close(:([1-9]+))?)$|on_file"
88              }
89            },
90            "required" :["name", "committed"]
91          },
92          {
93            "properties": {
94              "dirname": {"$ref": "#/$defs/list_of_files" },
95              "committed" :{
96                "description": "Commit rule",
97                "type" :"string",
98                "pattern": "^(on_termination)$|^(n_files(:([1-9]+)([0-9]*))?)$|on_file"
99              }
100           },
101           "required" :["dirname", "committed"]
102         }
103         ],
104         "properties": {
105           "mode" :{
106           "description": "Firing rule",
107           "enum": ["update", "no_update"]
108           }
109         },
110         "if": {
111           "type": "object",
112           "properties": {
113             "committed": {"const": "on_file" }
114           }
115         },
116         "then": {
117           "properties": {
118             "files_deps": {
119               "$ref": "#/$defs/list_of_files"
```

```
          }
        },
        "required": ["files_deps"]
      },
      "unevaluatedProperties": "false"
    },
    "minItems": 1
  },

  "home_node_policy": {
    "description": "In which nodes the files are stored during the workflow execution",
    "type": "object",
    "properties": {
      "manual": {
        "description": "Name of the policy",
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "name": {"$ref": "#/$defs/list_of_files" },
            "app_node" :{
            "description": "Appname:LogicID",
            "type": "string",
            "pattern": "^([a-z]+(:([0-9]+))?)$"
            }
          },
          "required": ["name", "app_node"]
        },
        "minItems": 1
      },
      "hashing" :{ "$ref": "#/$defs/list_of_files" },
      "create" :{ "$ref": "#/$defs/list_of_files" }
    },
    "additionalProperties": "false"
  }
 },
 "additionalProperties": "false"
}
```

Chapter 5

CAPIO Runtime

This chapter presents the architecture of the initial version of a runtime capable of adhering to the specifications outlined in the CAPIO coordination language. Subsequently, implementation details are explored, providing insights into how the runtime transforms a batch workflow into an in-situ workflow using a configuration file written in the CAPIO language by the user. Additionally, an example is provided on deploying the CAPIO runtime and a workflow onto an HPC system using the SLURM job scheduler.

5.1 Architecture

In this section, we introduce the software architecture of CAPIO and its runtime. The CAPIO runtime consists of a collection of user-space servers per node, which implement an ad hoc distributed data storage system for a specified workflow application. Utilizing information from the input configuration file, CAPIO enforces streaming data movements of files between different workflow steps (figure 5.1).

The software architecture of CAPIO is depicted in Figure 5.2 and is implemented in modern C++, leveraging MPI [50] for inter-node communications. CAPIO server processes are currently deployed on a node cluster using the mpirun launcher. Users define a CAPIO local-node mount point (`capio_mnt`) for each cluster node through the `CAPIO_DIR` environment variable. CAPIO captures all I/O system calls executed on files and directories created within the capio_mnt directory.

System calls directed at files outside the CAPIO local mount point are

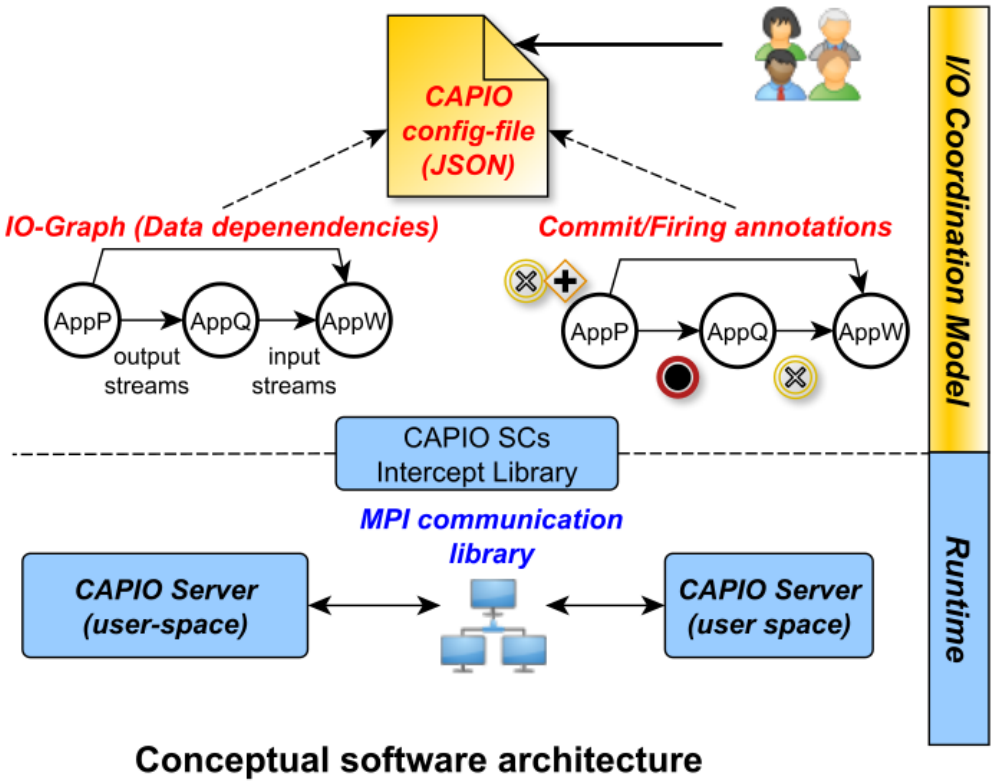

Figure 5.1: The CAPIO I/O coordination layers: the *config. file* embeds both I/O data dependencies and files' annotations describing commit and firing rules; the *CAPIO SCs intercept library* coupled with the local-node *CAPIO Server* transparently enable in-memory store of files and directories, and enforce synchronized accesses to them.

disregarded by CAPIO and forwarded to the kernel. The CAPIO implementation accommodates both multi-process and multi-threaded applications, allowing I/O calls to the same or different files to be executed by different processes or threads within the same application process.

The *CAPIO intercept library*, implemented using the Linux-x86_64 *system call intercepting library* `syscall_intercept` [106], functions as a shared library dynamically linked to the steps of the workflow. This linking occurs through the `LD_PRELOAD` dynamic linker environment variable, enabling the library to capture I/O system calls executed on files and directories within the CAPIO local-node mount point. Another alternative is to use FUSE or

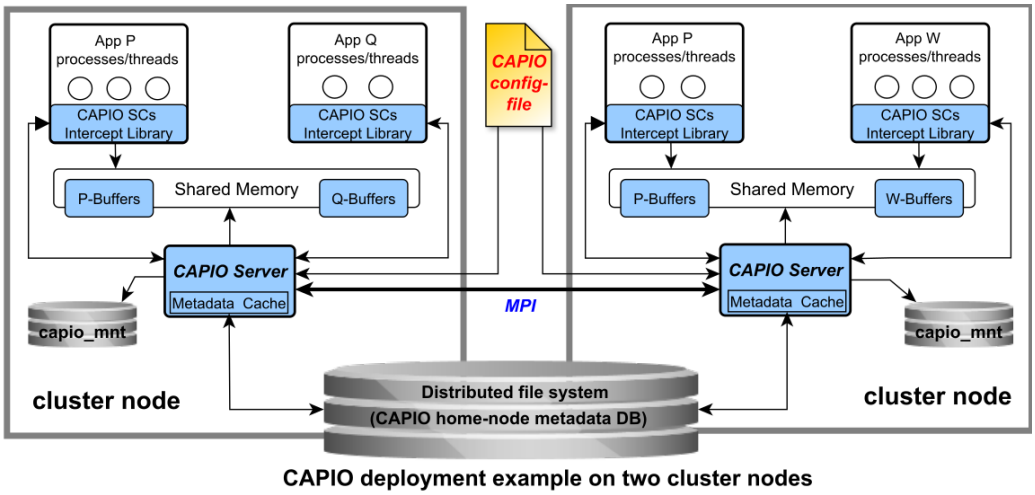

Figure 5.2: Deployment example of the CAPIO middleware on 2 cluster nodes: AppP is executed on both nodes; the local `capio_mnt` directory is the CAPIO FS entry point; the distributed FS is used to store *home-node* metadata information of the files.

to write a shared library that redefine the C lIbrary functions and that is loaded before GLIBC (using LD_PRELOAD) in order to capture the execution of those functions. FUSE can have performance issues ([140]) and it can be unpratical to be used in HPC systems because it require root access to be mounted. Therefore, as discussed in [10] the LD_PRELOAD method is to be generally prefered in HPC systems. We used the library syscall intercept that allows to capture directly the system calls instead of capturing the standard C library functions. This is a more general approach because it can used with programs or library that directly use the system calls and we have only to reimplement the system call and not all the C I/O functions. The CAPIO intercept library seamlessly communicates with the local CAPIO server through POSIX shared-memory APIs. By default, file data (along with metadata) is held in the primary memory of the producer. When the consumer step resides on the same node as the producer, communication takes place via local-node shared memory, overseen by the local CAPIO server. Conversely, if consumer steps are distributed across multiple cluster nodes, the requisite data is directly transferred from memory to memory between CAPIO servers.

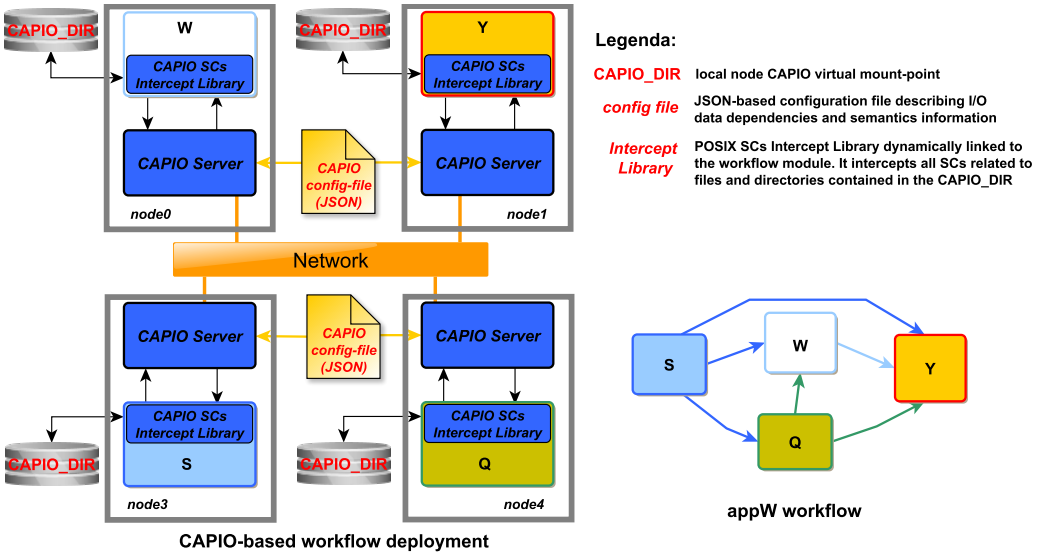

Figure 5.3: High-level view of the CAPIO middleware when used to coordinate I/O in a workflow application (the example workflow *appW* is composed of 4 steps *S*, *W*, *Q*, and *Y*) deployed on 4 cluster nodes (one workflow step for each cluster node).

5.2 Implementation

CAPIO's runtime comprises a collection of per-node user-space servers, referred to as CAPIO servers, that implement a distributed data storage system for a single application workflow. Each CAPIO server utilizes information from the provided CAPIO configuration file, serving as an input argument to ensure coordination of file exchanges and data streaming based on the producer-consumer semantics outlined in chapter 4.

Figure 5.3 illustrates a schematic deployment of CAPIO servers in a 4-node cluster (node0-4) for the application workflow example appW. In each node where the workflow steps are deployed, a CAPIO server manages all files and directories referenced by the step within the local-node CAPIO virtual mount-point. Any system calls (SCs) directed at files and directories outside the CAPIO local virtual mount-point are not directly handled by the CAPIO server; instead, they are forwarded to the kernel.

The internal workings of the CAPIO server are coded in C++, while server-to-server communications are currently implemented using MPI. Users

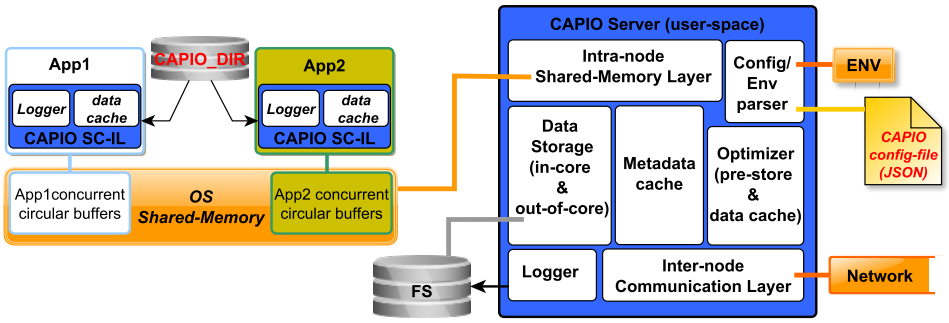

Figure 5.4: The internal software components of the CAPIO runtime running on a single node. It comprises two distinct parts: the CAPIO server and the CAPIO System Calls Intercept Library (CAPIO SC-IL).

specify a CAPIO local-node virtual mount-point for each node via the CAPIO_DIR environment variable. The CAPIO System Calls Intercept Library (CAPIO SC-IL), implemented using the Linux-x86_64 system call intercepting library syscall_intercept, is a shared library dynamically linked to the steps of the application workflow using the LD_PRELOAD dynamic linker environment variable. This library captures the I/O SCs executed on files and directories inside the local-node virtual mount-point. The SC-IL communicates with the local CAPIO server through a concurrent circular buffer stored in a POSIX shared-memory segment protected by POSIX Semaphores.

Figure 5.4 depicts the internal primary components of the CAPIO server running on a single node. Below, we provide a brief overview of the main functionalities of each component.

- Intra-node Shared-Memory Layer. The coordination of data exchange between the CAPIO SC-IL component, operating within the application process, and the CAPIO server occurs in this layer through the utilization of the POSIX Shared-Memory standard interface. Dedicated shared-memory segments are allocated for each workflow step that runs on the same node as the CAPIO server. These segments store buffers and semaphores essential for synchronization, and they are promptly dismantled upon the exit of the corresponding application step.

- Config/Env parser. The CAPIO server utilizes a JSON-based configuration file to store information related to file coordination and streaming semantics. It also relies on a set of environmental variables to

obtain details for tuning the performance of internal components and to determine the name of the CAPIO_DIR. The Config and Env parser extracts this information from the configuration file and the process's environment variables, organizing it for use by other CAPIO components.

- Data Storage. This component stores the contents of files for which the CAPIO server acts as the home-node, meaning it stores the master copy of the file. While portions of a file may also be stored in other CAPIO servers to enhance performance and reduce network traffic, the current CAPIO middleware implementation designates a single master node responsible for storing the most recent metadata and data information for a given file or directory. Other CAPIO servers always refer to the home-node to determine if a file has updated data and metadata. By default, each file is stored in the main memory of the CAPIO server. However, upon request, the user can choose to store the file in the filesystem using the 'permanent' keyword in the JSON config file. This ensures that the file is preserved even after the termination of the CAPIO server. This feature is currently implemented by utilizing the mmap system call.

- Metadata cache. This component functions as a metadata cache for files and directories, specifically those for which the current CAPIO server is not the home-node. To maintain data coherence, a straightforward invalidation protocol is implemented among CAPIO servers, ensuring that the metadata cache only retains up-to-date information.

- Optimizer. The Optimizer is responsible for implementing file data caching in CAPIO servers that are not the home-node. Its role includes optimizing the transmission of data by allowing the sending of more data than requested, aiming to minimize the number of request-reply message exchanges among CAPIO servers.

- Logger. The CAPIO Logger consists of two sections—one for storing information related to the CAPIO server and the other for storing information related to the CAPIO SC-IL component. These two loggers can be activated independently and are not typically compiled into CAPIO, unless explicitly instructed to do so. The logger component is also capable of logging at different levels, ranging from 0 (no logging)

to "n", where "n" represents the maximum depth of the function on the stack. If the log level is less than 0, then everything is logged.

- Inter-node Communication Layer. This component oversees the coordination of communications between CAPIO servers, facilitating the exchange of both data and metadata information. The software layer is intentionally designed to be agnostic to the specific data transport method employed for these communications. Currently, the abstract interface utilizes the Message Passing Interface (MPI) library. In future releases of the CAPIO middleware, users will have the flexibility to choose from various transport back-end options.

The CAPIO server utilizes the names of workflow application steps to associate them with their corresponding semantic information in the configuration file. The application step name is specified at the launch time of each workflow step by utilizing the environment variable CAPIO_APP_NAME. During the start-up handshake protocol, the CAPIO SC-IL transmits this information to the local CAPIO server. The name of the application step is identified by the JSON file's language keyword "name".

CAPIO's implementation supports both multi-process and multi-threaded applications. By default, file data (and metadata) is stored in the main memory of the node where the file is created (this is the default home_node policy adopted by the CAPIO middleware). If the consumer step is deployed on the same producer node, communications occur through the shared memory buffer mediated by the local CAPIO server. Conversely, if consumer steps are deployed on different cluster nodes, the requested data is transferred through direct memory-to-memory communications between CAPIO servers. However, file data placement can be controlled by setting the "home_nodes" keyword in the CAPIO configuration file.

Regarding file metadata information, not all metadata is kept consistent for each data and metadata access, such as the timestamp fields, for performance reasons. Instead, the file size is always kept consistent in the home-node, allowing CAPIO to handle sparse files—a technique often employed to write different partitions of a single file in parallel.

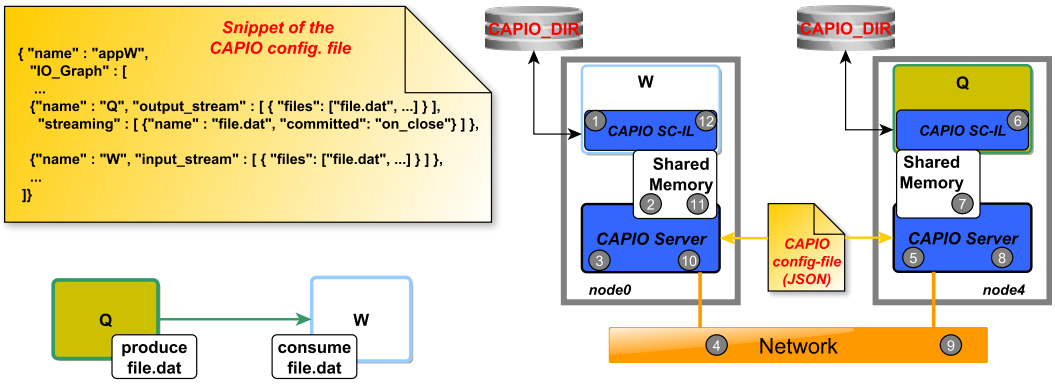

Figure 5.5: Schema depicting a straightforward producer-consumer workflow deployed across two nodes. The diagram illustrates the steps involved in the request-reply protocol for handling read and write operations on the "file.dat".

5.2.1 CAPIO servers interaction for writing and reading data

Here, we showcase a straightforward example of a producer-consumer workflow involving two application steps—Q (the producer of the file.dat file) and W (the consumer of that file)—executing on separate computing nodes. The objective is to elucidate the producer-consumer protocol between the two CAPIO servers by outlining the steps taken to address two distinct scenarios: 1) W attempts to read the file.dat before Q has produced its content; 2) W reads the file when Q has already started to produce its content. The CAPIO server where Q is deployed serves as the home_node for the file. A snippet of the JSON configuration file outlining the example workflow is provided in Figure 5.5.

In the first scenario, we assume that step W opens the file file.dat for reading, even though the data for this file has not yet been produced by step Q.

The CAPIO SC-IL intercepts the read system call (step 1) and forwards the request to the CAPIO server through a shared-memory segment containing the requests' buffer (step 2). The CAPIO server checks whether the requested data is already present in the local data cache. If not, the CAPIO server retrieves the CAPIO home-node server (step 3) and dispatches a data request, specifying the amount of data to read, to the selected server

(step 4). Subsequently, the CAPIO server resumes handling other requests from local workflow steps or other remote CAPIO servers. Meanwhile, the requesting process in the W step, which initiated the read SC request, awaits the completion of the operation.

The home-node CAPIO server receives the read request and verifies the availability of the data in the local storage (step 5). In our example, the producer Q has not yet produced the data; therefore, the home-node CAPIO server cannot immediately respond to the request. When the step Q begins writing the data into the file.dat file, the CAPIO SC-IL intercepts all write system calls (step 6), stores the data in its local cache, and then forwards the data to the CAPIO server through a shared-memory buffer, either when the cache buffer is full or when the file is closed (step 7). The CAPIO server stores the data in the local storage (step 8), updates the metadata information, and concurrently handles all pending remote requests awaiting a reply (step 9).

The amount of data sent as a reply message may exceed the initially requested size if the CAPIO server's Optimizer component is configured to perform pre-store operations aggressively. Upon receiving a reply message, the CAPIO server stores the data in the local cache (step 10) and places the requested data from the initial read SC into the shared-memory buffer associated with the replies (step 11). Finally, the CAPIO-IL completes the read request by transferring the data into the SC buffer (step 12).

The second scenario (i.e., W initiates reading the file content after Q has already begun producing its content) closely resembles the previously described scenario, but with a few distinctions. When the home-node CAPIO server receives the read request from the CAPIO server running on the same node as the W step, it promptly responds to the request. It transmits more data than the amount requested, provided it is available in the data storage, up to a specified threshold defined by the user through the environment variable CAPIO_PREFETCH_DATA_SIZE (refer to section 5.2.4). This optimization is termed pre-store. The objective is to enhance data transmission efficiency and potentially decrease the number of request-reply message exchanges.

5.2.2 Home-node Implementation

The home-node DB is currently implemented using POSIX files in the distributed filesystem, and it is updated/accessed by the CAPIO servers through POSIX file locking to prevent race conditions. Additionally, it is cached in

the main memory of each CAPIO server for performance reasons, as file-to-home-node mappings remain static. Looking ahead, we plan to explore alternative solutions, such as in-memory distributed databases. When a process seeks to read a file, it needs to determine which node serves as the home-node for that file. If the home-node is known statically, the process is straightforward. However, in the case of a dynamic home-node policy, the only way to retrieve the home-node is through a query to the home-node database. Currently, there are three home-node policies: "create", "manual", and "hashing". "Create" and "manual" are dynamic policies as the home-node is determined only at runtime. In contrast, the hashing policy is static because retrieving the node requires applying the hashing function to the file's path. Presently, a file is stored in the memory of the home node. However, our future plans include offering users the option to distribute the content of a file across multiple nodes. This technique, known as "Data Stripping", is commonly employed in distributed file systems to alleviate the bottleneck associated with a single node. Distributing the content of a file across multiple locations enhances performance, particularly for multiple parallel read requests to different parts of a file.

5.2.3 Deployment

To deploy a workflow using CAPIO, it is necessary to launch a CAPIO server on each node designated for workflow execution. Additionally, for every application step within the workflow, the CAPIO SC-IL library must be linked by setting the LD_PRELOAD environment variable to the path of the CAPIO shared library (libcapio_posix.so).

In listing 5.1, an example script for the Slurm job scheduler [2] is presented for a straightforward workflow leveraging the CAPIO middleware. The workflow consists of two application components: a writer and a reader. Notably, CAPIO enables the concurrent execution of writer and reader components. In this simple scenario, the workflow operates on two cluster nodes, with the writer initiated on the first node and the reader on the second. The CAPIO server is active on both nodes.

Looking ahead, for enhanced CAPIO deployment convenience, we plan to integrate the CAPIO middleware with workflow management systems like StreamFlow [141] and DagOnStar [84]. This integration aims to automate the CAPIO deployment process.

Listing 5.1: SLURM script example to execute the *reader-writer* example workflow with CAPIO.

```
#!/bin/bash
#SBATCH --exclusive
#SBATCH --job-name=my_workflow
#SBATCH --error=myJob%j.err    # standard error file
#SBATCH --output=myJob%j.out   # standard output file
#SBATCH --nodes=2

# GET THE LIST OF NODES
read -d ' ' -a nodelist <<< "$(scontrol show hostnames
    $SLURM_NODELIST)"

if [ $# -ne 3 ]
then
        echo "Usage: $0 CAPIO_HOME CAPIO_DIR CONF_FILE"
        exit 1
fi

capio_home=$1    # the install directory of CAPIO
capio_dir=$2     # the CAPIO virtual mount-point
conf_file=$3     # the name of the JSON configuration
    file

# RUN ONE CAPIO SERVER IN EACH NODE
srun --exact -N $SLURM_NNODES -n $SLURM_NNODES --ntasks
    -per-node=1 $capio_home/src/capio_server server.log
     $conf_file &
SERVER_PID=$!

# RUN A WRITER PROCCES IN THE NODE 0
srun -N 1 -n 1 -w ${nodelist[0]} --exact --export=ALL,
    LD_PRELOAD="$capio_home/libcapio_posix.so",
    CAPIO_DIR="$capio_dir", CAPIO_APP_NAME="writer" ./
    writer &
WRITER_PID=$!

```

```
29 # RUN A READER PROCESS IN THE NODE 1
30 srun -N 1 -n 1 -w ${nodelist[1]} --exact --export=ALL,
       LD_PRELOAD="$capio_home/libcapio_posix.so",
       CAPIO_DIR="$capio_dir", CAPIO_APP_NAME="reader" ./
       reader
31
32 wait $WRITER_PID
33 kill $SERVER_PID
```

5.2.4 Configuration options

This section outlines a set of environmental variables tailored to fine-tune both the performance of the CAPIO middleware and logging configuration with varying levels of verbosity. The categorization distinguishes between "server-side", "client-side", and "both", indicating whether the variable is utilized by the CAPIO server, the CAPIO SC-IL, or both components.

- CAPIO_FILE_INIT_SIZE (Server-side): The default RAM space reserved for files stored by the CAPIO server is 1MB. Adjusting this variable to a larger value can enhance efficiency. When a file surpasses the reserved space, memory reallocation to a larger size becomes necessary.

- CAPIO_PREFETCH_DATA_SIZE (Server-side): The number of bytes to prefetch from a remote CAPIO server when a remote read is requested. The default is 0, indicating that only the requested data is retrieved. Setting this variable to a larger value is beneficial for aggressively caching data between nodes.

- CAPIO_WRITER_CACHE_SIZE (Client-side): The number of bytes for the cache between the application and the local CAPIO server for write operations. The default value is 0, indicating no cache. Configuring this cache enhances performance when a file undergoes numerous sequential small write operations.

- CAPIO_READER_CACHE_SIZE (Client-side): The number of bytes for the cache between the application and the local CAPIO server for

read operations. The default value is 0, indicating no cache. Configuring this cache may enhance performance when a file undergoes numerous sequential small read operations.

- CAPIO_N_ELEMS_DATA_BUFS (Both): The number of elements in the shared circular buffer used for data communication between the CAPIO SC-IL and the local CAPIO server.
- CAPIO_WINDOW_DATA_BUFS_SIZE (Both): The size in bytes of the elements of the shared circular buffer used for data communication between the CAPIO SC-IL and the local CAPIO server.
- CAPIO_LOG_DIR (Both): This environmental variable redefines the default directory in which log files are stored.
- CAPIO_LOG_PREFIX (Both): This environmental variable redefines the names of the log files CAPIO uses. By default, the CAPIO SC-IL and the CAPIO server create log files into:
  "CAPIO_LOG_DIR/posix/<machine_hostname>/<thread_id>.log"
  "CAPIO_LOG_DIR/server/<machine_hostname>/<thread_id>.log"
- CAPIO_MAX_LOG_LEVEL (Both): This environmental variable is used to adjust the verbosity level of the CAPIO logging.

Chapter 6

Evaluation

In this section, we present the outcomes derived from employing the CAPIO middleware across a series of synthetic benchmarks designed to replicate typical I/O workflow patterns, as well as on three distinct scientific workflows. The initial workflow, following a MapReduce paradigm, simulates common in-memory MapReduce computations using a sizable input dataset. The second workflow, centered around the 1000 Genomes Project [142], adopts a DAG-based bioinformatics approach for data parsing. The third workflow is based on a mesoscale numerical weather prediction, leveraging the Weather Research and Forecasting (WRF) Model [143].

6.1 System configuration

Our experiments were conducted by deploying the CAPIO middleware on the GALILEO100[1] and on the HPC4AI cluster (a.k.a. UNITO cluster)[2]. GALIELO100 is a Tier-1 supercomputer hosted at the CINECA supercomputing center. Each computing node utilized in the experiments is equipped with 2 Intel CascadeLake 8260 CPUs, featuring 24 cores running at 2.4 GHz, and is outfitted with 384 GB RAM. The operating system is Centos 8.3.2011, and the Linux kernel version is 4.18.0-240. The storage system is built on the *Lustre* open-source parallel file system [37]. Each cluster node is interconnected through a switched 100 Gb/s Infiniband. The *scratch* directory, mounted on the Lustre file system under /g100_scratch, is connected to stor-

[1]GALILEO100: https://www.hpc.cineca.it/hardware/galileo100
[2]HPC4AI Cluster: https://hpc4ai.unito.it

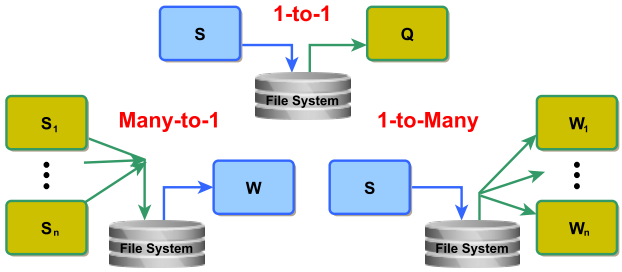

Figure 6.1: Common workflow patterns tested with CAPIO. An outgoing arrow indicates some files are produced while an incoming arrow indicates that some files are consumed.

age with a 100 Gb/s Infiniband interconnect. Throughout our experiments, we consistently utilized the scratch directory to store files and directories when interacting with the file system.

The UNITO cluster comprises 68 Broadwell nodes connected through an OPA 100Gbit/s network. Each node is equipped with 2 Intel(R) Xeon(R) E5-2697 v4 running at 2.3GHz, 18 cores each. The distribution is Ubuntu 20.04.5 LTS, and the Linux kernel version is 5.4.0-137. The filesystem used for the tests is BeeGFS.

Job submissions were facilitated through Bash scripts employing the Slurm cluster resource manager. CAPIO was configured to use the default home-node policy. For each test, we executed 10 runs, excluded the highest and lowest values obtained, and then computed the average value from the remaining results.

6.2 The system calls' intercept overhead

The initial tests were conducted to assess the overhead introduced by the CAPIO SC intercept library. We utilized the *lmbench* benchmarks [144], which consist of micro benchmarks measuring OS and hardware system metrics. Specifically, we focused on the `lat_syscall` benchmark, which measures the latency of certain simple system calls.

Two scenarios were tested:

- No LD_PRELOAD defined, indicating no system call interception.
- using the CAPIO intercept library by setting LD_PRELOAD=libcapio_posix.so.

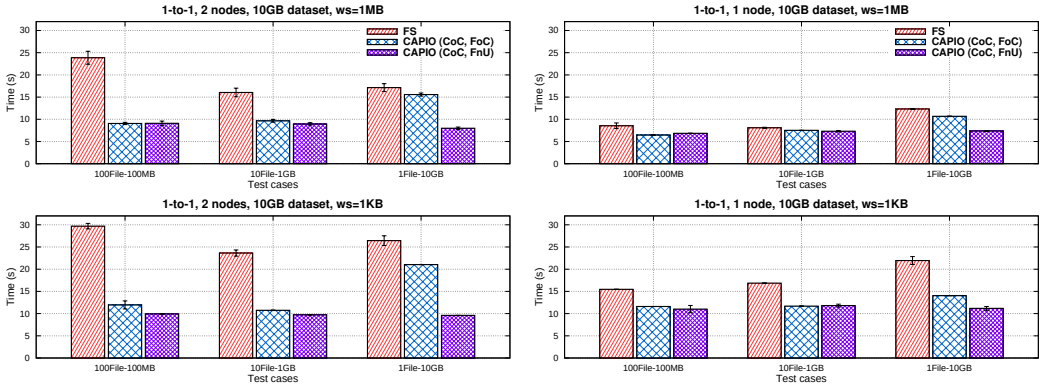

Figure 6.2: *1-to-1* benchmark test with 10GB dataset and two different sizes for the read/write buffer *ws*: large (1MB) in the top plot; small (1KB) in the bottom plot. *Left*): test executed on one cluster node. *Right*): test executed on two cluster nodes.

The results, averaged over 5 repetitions, are presented in Table 6.1. In summary, the overhead introduced by the CAPIO intercept library is relatively minimal.

SCs	**no intercept**	**CAPIO intercept**
open	1.35	1.49
read	0.18	0.23
write	0.13	0.18
stat	0.45	0.52
fstat	0.19	0.24

Table 6.1: Execution time (in microseconds) of the `lat_syscall` test from *lmbech* benchmark suite considering some relevant SCs.

6.3 Synthetic banchmarks

The synthetic benchmarks are designed to portray relatively straightforward scenarios, showcasing the potential impact of CAPIO optimizations on real-

use cases. These benchmarks emulate three recurring application I/O patterns: *a)* one producer and one consumer steps (*1-to-1*); *b)* one producer and multiple consumer steps (*1-to-Many*); *c)* multiple producers and one consumer steps (*Many-to-1*). These patterns are illustrated in Fig. 6.1.

All benchmarks are implemented in C and utilize standard POSIX library calls such as `fopen`, `fclose`, `fwrite`, `fread`, `feof`, and `lseek` for managing I/O operations.

6.3.1 Synthetic benchmarks on GALILEO100

These tests were performed on the GALILEO100 supercomputer. Apart from basic checksum checks to verify result correctness, the workflow steps exclusively carry out I/O operations. The tested cases include:

1. Default batch execution, where producers and consumers are executed sequentially, utilizing the file system for file sharing (referred to as *FS*).

2. Execution with CAPIO, employing *Commit on-Close* and *Firing on-Commit* semantics for all files (referred to as *CAPIO-CoC-FoC*, i.e. "MODE"="update" at line 4 in listing 6.1).

3. Execution with CAPIO, employing *Commit on-Close* and *Firing no-Update* semantics for all files (referred to as *CAPIO-CoC-FnU*, i.e. "MODE"="no_update" at line 4 in listing 6.1).

In CAPIO executions, we initiated the CAPIO daemon(s), followed by the producers and then the consumers that are exeeceuted concurrently. The reported execution time represents the maximum values across all workflow steps.

Listing 6.1: CAPIO configuration file for the synthetic benchmarks. "MODE" can be equal to "update" or "no_update".

```
1 { "name" :"benchmarks",
2   "IO_Graph" :[
3   {"name" :"S", "output_stream": ["file*.dat"],
4   "streaming": [{"name" :["file*.dat"], "committed":"on_close", "mode":"MODE"}] },
5   {"name" :"Q", "input_stream" :["file*.dat"] }
6   ]
7 }
```

1-to-1

In this benchmark, the producer generates N files, each of size M, using a buffer size of ws KB, while the consumer reads all N files with the same buffer size ws. The results depicted in Figure 6.2 (left-hand side) showcase the outcomes with 2 cluster nodes and varying values of N, M, and ws. The specific test cases include: *100Files-100MB* ($N = 100$, $M = 100MB$), *10Files-1GB* ($N = 10$, $M = 1$GB), and *1File-10GB* ($N = 1$, $M = 10$GB). For the buffer size, two cases were tested: $ws = 1$MB (top plot) and $ws = 1$KB (bottom plot).

Overall, CAPIO-based tests consistently exhibit higher performance across all scenarios. An additional advantage is the lower variance compared to FS, as it is independent of file system utilization, which tends to be high in large-scale production supercomputers with numerous users. With a small buffer size (i.e., $ws = 1$KB) in I/O operations, execution times increase due to the higher number of system calls and `libc` internal buffer flush operations. However, CAPIO proves to be less sensitive to this aspect compared to the file system.

As expected, *CAPIO-CoC-FnU* synchronization semantics outperform *CAPIO-CoC-FoC* when dealing with a few large files (e.g., the test *1File-10GB*). Moving to the right-hand side of Figure 6.2, the results for the same set of tests with both producer and consumer deployed on the same cluster node are presented. In this scenario, the benefits of CAPIO are less pronounced, except for the case 1File-10GB. This can be attributed to aggressive in-memory local node file system caching of data blocks. Additionally, deployment on the same node may have significant effects depending on the nature of workflow steps executed. Strict I/O-bound workflow steps can be effectively co-executed on the same node, removing the potentially overwhelmed file system from the I/O path. Conversely, mixed I/O-/CPU-bound workloads may exhibit different behaviors due to the sharing of CPU cores and memory bandwidth.

1-to-Many

In this benchmark, we examined two scenarios presented in Figure 6.3 (left-hand side): *1)* the producer writes 100 files, each of size 1GB (top plot); *2)* the producer writes one file of 100GB, and the consumers read disjoint partitions of the file (bottom plot). We tested the number of consumer steps

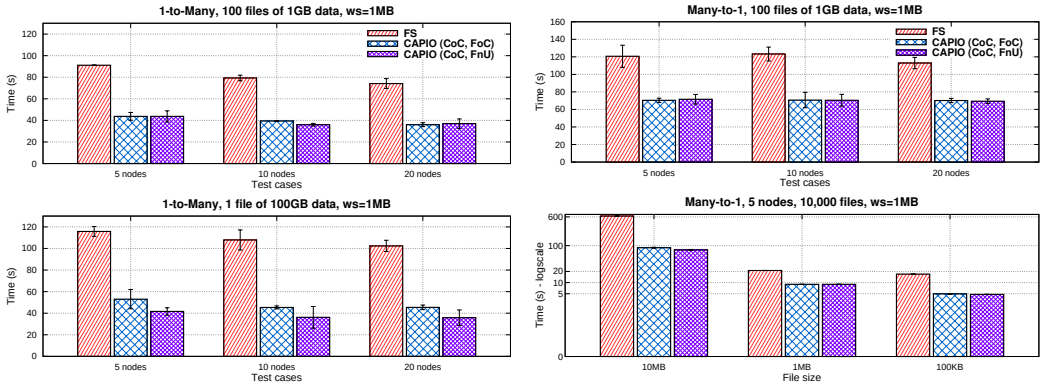

Figure 6.3: `Left)`: *1-to-Many* test executed with 100 files of 1GB each (top plot), and one single file of 100GB (bottom plot). `Right)`: *Many-to-1* executed with 100 files of 1GB each (top plot), and 10,000 files of different size on 5 cluster nodes (bottom plot – log. scale). For all tests the buffer size is $ws = 1$MB.

with values of 5, 10, and 20.

As for the *1-to-1* test, the producer step is more efficient as data is written to the local CAPIO server through a shared-memory segment. Consequently, the execution time is primarily influenced by the consumers' reading operations. In the first test, there is minimal difference between the two firing rules tested, similar to the *1-to-1* test for file sizes of 1GB. In this case, the writing time for producing the file with id k overlaps with the reading time of the file with id $k-1$ by one of the consumers. Since files are consumed in increasing order of their ids, there is no significant variation in execution time when increasing the number of consumer steps (i.e., nodes).

The second test mirrors the first one in its scatter communication pattern but differs in implementation by using a sparse file. Once again, CAPIO exhibits consistent results with those of the first test. Specifically, the *CAPIO-CoC-FnU* synchronization semantics outperforms *CAPIO-CoC-FoC* due to increased overlap in the I/O phase between producer and consumer steps. In contrast, the file system execution performs less optimally when dealing with a large sparse file created and written after seeking.

Many-to-1

In this benchmark, we explored two scenarios depicted in Figure 6.3 (right-hand side): *1)* the producers collectively generate 100 files, each of size 1GB (top plot); *2)* the producers generate a large number of small files (10,000), with three different file sizes: 10MB, 1MB, and 100KB. In the first test, the number of producer steps tested is 5, 10, and 20. In the second case, the total number of workflow steps is fixed at 6 (i.e., 5 producers and 1 consumer).

Qualitatively, the first test exhibits results similar to those in the initial test of the *1-to-Many* benchmark, even though they emulate different communication patterns—scatter versus gather. However, the absolute execution time is higher for both FS and CAPIO cases. For CAPIO, this is partially attributed to its more intricate communication protocol.

In the second test (bottom plot in Figure 6.3), FS encounters challenges when dealing with numerous files and a relatively large dataset (up to 100GB). In all tested cases, CAPIO introduces less overhead. With 10,000 files of 10MB, the speed-up is more than 8 ($\sim$ 630s vs. $\sim$ 75s).

6.3.2 Synthetic Benchmarks with ADIOS

We have implemented the synthetic benchamarks with ADIOS and compared them with CAPIO (CoC-FnU semantics) and the filesystem on the UNITO cluster. For translating the synthetic bechmarks described in Listing 1 and 2 in ADIOS2 variable is defined for each file and an explicit synchronization instructions (begin_step and end_step) are added for reading/writing in a streaming fashion. The pseudocode for the ADIOS version can be found in listing 4 and 5. This pseudocode was presented to avoid to be lost in C++ and ADIOS2 details, but the real code used for the tests can be found on the CAPIO repository. For these experiments we did not do any finetuning on the ADIOS2 parameters because it is outside the scope of these experiments. We used the filesystem as backend for ADIOS2, because in this cluster we obtained the best results with this backend.

For these experiments, we tested different computation times for producers and consumers to simulate more realistic scenarios. In fact, the synthetic benchmarks in the previous section represent the worst-case scenario for streaming communication because there is no opportunity to overlap communication and computation. In these tests, the writer processes perform a sleep of N seconds before starting a file, where N represents the computation

Algorithm 4: A simple file writer application with ADIOS2.

```
Data: n_files ≥ 0 ; file_size > 0 ; secs ≥ 0
N ← 0;
while N < n_files do
    adios_variable ← define_variable("fileN.dat", file_size);
    buffer ← compute(secs, file_size);
    begin_step();
    write_variable(buffer, adios_variable);
    end_step();
    N ← N + 1;
end
```

Algorithm 5: A simple file reader application with ADIOS2.

```
Data: n_files ≥ 0 ; file_size > 0 ; secs ≥ 0
N ← 0;
while N < n_files do
    adios_variable ← define_variable("fileN.dat", file_size);
    begin_step();
    buffer ← read_variable(adios_variable);
    end_step();
    compute(secs, buffer, file_size);
    N ← N + 1;
end
```

time before writing a file. $N = 0$ implies no computation, similar to the previous tests. When $N > 0$, it allows for better exploitation of streaming execution because, during the writer's sleep, the reading process can read the data produced in the previous iteration. However, if N is too large, the improvement in I/O communication may be less impactful because the workflow becomes CPU-bound rather than I/O-bound. If N is significant enough, after the writer's sleep, the consumer may have already completed the read of the file generated in the previous iteration. By the end of the writer's execution, the consumer will have read all the files except the last one. Increasing the value of N in this case may not lead to an improvement in streaming communication because the consumer was already fast enough with the previous value of N. Therefore, before converting a batch workflow

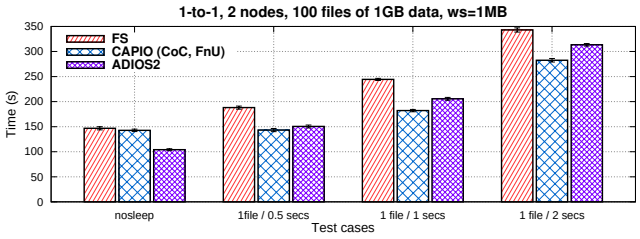

Figure 6.4: *1-to-1* tests for different times of computation with the filesystem, CAPIO and ADIOS2 on 2 nodes.

into an in situ workflow, the user should assess the possible gains through the use of a profiling tool. In the next subsections are reported the results of these tests for window size of 1 MB and a dataset of 100 files of 1 GB.

1 to 1

In Figure 6.4, the results for the *1-to-1* tests are presented. When no sleep is performed, CAPIO performs slightly better than the filesystem, while ADIOS2 is faster than both. As expected, in this case, the effort of rewriting the application with ADIOS2 proves beneficial. When we increase the time spent on sleep to 0.5 seconds, the execution time for the filesystem increases, while CAPIO performs slightly better than in the previous case, outperforming the filesystem. In this scenario, the execution time with CAPIO has not increased because it can utilize the time spent on sleep for asynchronous communication. On the other hand, ADIOS2 is still better than the filesystem but slightly worse than CAPIO. This is likely due to using the filesystem as the backend for ADIOS2, and no fine-tuning of its parameters was done. Increasing the time spent on computation (N) to more than 0.5 also increases the execution time for CAPIO because the workflow becomes CPU-bound. In fact, with $N = 0$, the computation time for the filesystem is approximately 150 seconds, while with $N = 2$, the execution time is approximately 340 seconds. With a sleep before writing every file, the total time spent sleeping is $100 * 2 = 200$ seconds because the writer produces 100 files.

1 to many

The results for the *1-to-many* test case are presented in figure 6.5. With 5 nodes, CAPIO and ADIOS2 outperform the filesystem due to their ability

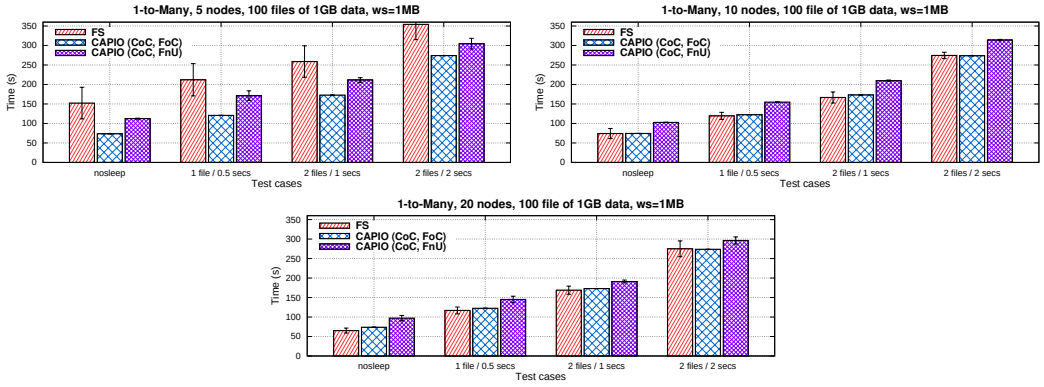

Figure 6.5: *1-to-Many* tests for different times of computation with the filesystem, CAPIO and ADIOS2 using 5, 10 and 20 nodes.

to exploit streaming communication. With 10 and 20 nodes, the filesystem performs slightly better than CAPIO and ADIOS2. For ADIOS2, this can be explained by the lack of fine-tuning or the choice of using the filesystem as a backend. For CAPIO, this depends on the use of the home-node policy. Currently, only the "create" policy is implemented, and in this case, it creates a bottleneck because all 100GB of data is stored on the node of the writer. Instead, the filesystem leverages data stripping to avoid this problem. With CAPIO, this could be addressed using the "hashing" policy or the "manual" setting, explicitly mapping the file-node relationship in the configuration file. As future work, we plan to explore the impact of different home-node policies.

many-to-1

The results for the *many-to-1* test case are presented in figure 6.6. With no time spent on sleep, the filesystem is faster than CAPIO. This is likely caused by the fact that the implementation of CAPIO is still in a prototype phase, while BeeGFS is a mature distributed filesystem. Our goal here is not to showcase the maturity of the CAPIO implementation in every aspect but to demonstrate that using the CAPIO language can improve performance in some cases. This is evident in this test case as well because when $N > 0$, the CAPIO time is lower than the filesystem, and up to $N = 1$, its execution time increases more slowly. This is because with streaming communication, CAPIO is able to exploit the time while the writer is sleeping to pass the file

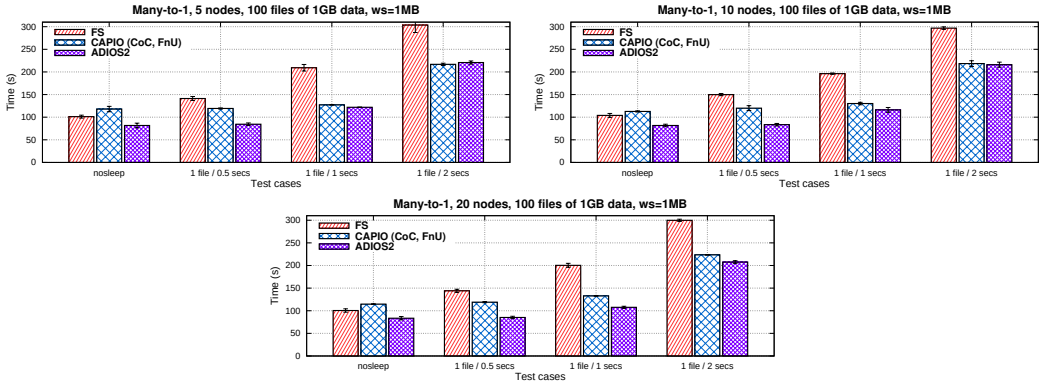

Figure 6.6: *Many-to-1* tests for different times of computation with the filesystem, CAPIO and ADIOS2 using 5, 10 and 20 nodes.

of the previous iteration to the consumer, which can start reading it.

6.3.3 Home-node policies impact

As discussed in section 4.3, the CAPIO language provides a mechanism for defining how data is distributed on the nodes used by a workflow. At the moment, the runtime presented in this work only supports the "create" policy, where a file is stored on the node where the file is created. Home-node policies can significantly impact workflow performance, as the distribution of data plays a key role, especially in HPC systems. For example, the "hashing" policy, where files are stored in one of the nodes available to the running workflow by applying a hash function to their pathname, is well-suited for parallel reads, while the "manual" policy is useful for pre-moving the data to the node where a process that needs to read a specific file will be executed.

Considering the example depicted in Figure 4.1 and assuming, for the sake of simplicity, the on-termination semantics, the workflows must be executed respecting the I/O dependency graph. Therefore, first, the application S must be run, then the applications W, X, and Z can be run in parallel. After they have terminated, the application Y can start and read the file produced by the previous applications. With the default home-node policy "create", the file "file-out.dat" is on the node that first created it, which could be different from the node where Y is running. Therefore, the protocol discussed in 5.2.1 is used to retrieve "file-out.dat" from a remote node. If, instead of the default home-node policy, the "manual" one is used, it is possible to pre-move the

data to the node where Y will be executed. In this way, when Y starts reading "file-out.dat", the file is already in the main memory of the node, resulting in a performance improvement in reading the file.

To demonstrate the potential of the home-node policies, we conducted a simple test by implementing the example discussed earlier and a prototype version of the "manual" policy. We compared the two different policies. With the default policy, the application Y takes 34 seconds, while with the "manual" policy, where pre-moving data optimization is enabled, it takes 27 seconds, resulting in a decrease of circa 20%.

6.4 Real-world applications

6.4.1 1000 Genome

The 1000 Genomes use case [87] is a DAG-based bioinformatics workflow for computing overlaps in human genome mutations. Figure 6.7 illustrates the five steps of the workflow and their interdependencies. The "individuals" step can be replicated in multiple independent instances, with each instance analyzing a partition of the input file and generating a directory containing 2,504 temporary small files (ranging from 1KBto 15KB, with 16 instances). The "sifting" step runs concurrently with all "individuals" steps. The "individuals_merge" step reads directories produced by all "individuals" and combines them into one directory with 2,504 files, where each file is a merge of all files with the same name produced by "individuals". The last two steps, "mutation_overlap" and "frequency", are independent, reading the input dataset and data produced by previous steps.

We tested CAPIO with *CoC-FnU* synchronization semantics to leverage pipeline parallelism among steps whenever possible. In Listing 6.2 is shown the configuration file for this use case. For instance, "mutation_overlap" and "frequency" may commence reading the input dataset while "individuals merge" and "sifting" are still running. Such overlap is unattainable with the traditional workflow execution model. Originally implemented using Bash and Python scripts, the 1000 Genomes workflow underwent reimplementation using C++ and the Boost library, resulting in a more than 3X reduction in total execution time. We opted to test CAPIO using the fastest version and a single chromosome simulation (distinct simulations on various chromosomes generate separate and independent workflows that can

be executed in parallel). Testing it with three configurations: 8, 12, and 20 cluster nodes, corresponding to 4, 8, and 16 "individuals" instances. 16 individuals produce $16 \times 2,504 = 40,064$ temporary files.

As expected, CAPIO accelerates the execution compared to FS (refer to Figure 6.8). Demonstrated by the benchmarks, CAPIO efficiently handles a high number of files with less overhead. Furthermore, the files generated by the "individuals" steps are temporary and consumed by the "individuals_merge" step; they do not need to be stored in the file system when the steps conclude, saving time.

Listing 6.2: CAPIO configuration file for the 1000 genome workflow using the CoC-FnU semantics.

```
{
"name" :"1000_genome",
"IO_Graph" :
[
        {
                "name" :"download",
                "output_stream" :["data"],
                "streaming" :[
                        {
                                "name" :["data*"],
                                "committed" :"on_close",
                                "mode" :"no_update"
                        }
                ]
        },
        {
                "name" :"individuals",
                "input_stream" :["data/20130502/ALL.chr1.250000.vcf"],
                "output_stream" :["chr1n-*"],
                "streaming" :[
                        {
                                "dirname" :["chr1n-*"],
                                "committed" :"n_files:2504",
                                "mode" :"no_update"
                        },
                        {
                                "name" :["chr1n-*/*"],
                                "committed" :"on_close",
                                "mode" :"no_update"
                        }
                ]
        },

        {
                "name" :"individuals_merge",
                "input_stream" :["chr1n-*"],
                "output_stream" :["chr1n"],
                "streaming" :[
                        {
                                "dirname" :["chr1n"],
```

```
41                              "committed" :"n_files:2504",
42                              "mode" :"no_update"
43                      },
44                      {
45                              "name" :["chr1n/*"],
46                              "committed" :"on_close",
47                              "mode" :"no_update"
48                      }
49              ]
50      },
51
52      {
53              "name" :"sifting",
54              "input_stream" :["data/20130502/sifting/ALL.chr1.
                    phase3_shapeit2_mvncall_integrated_v5.20130502.sites.annotation.vcf"],
55              "output_stream" :["sifted.SIFT.chr1.txt"],
56              "streaming" :[
57                      {
58                              "name" :["sifted.SIFT.chr1.txt"],
59                              "committed" :"on_close",
60                              "mode" :"no_update"
61                      }
62              ]
63      },
64
65      {
66              "name" :"mutations_overlap",
67              "input_stream" :["sifted.SIFT.chr1.txt", "chr1n", "data/populations/ALL", "
                    data/20130502/columns.txt"],
68              "output_stream" :["chr1-ALL.tar.gz"]
69      },
70
71      {
72              "name" :"frequency",
73              "input_stream" :["sifted.SIFT.chr1.txt", "chr1n", "data/populations/ALL", "
                    data/20130502/columns.txt"],
74              "output_stream" :["chr1-ALL-freq.tar.gz"]
75      }
76  ]
77  }
```

6.4.2 Map-Reduce workflow

We have implemented a straightforward workflow in C that replicates the typical I/O pattern observed in Map-Reduce computations, where intermediate results are stored in the main memory of cluster nodes [145]. This workflow consists of three steps, as illustrated in Figure 6.9. The sequential "Split" step takes a large file as input and generates K smaller files. The parallel "MapReduce" step, comprising multiple instances of Mappers and Reducers, reads the K files and produces M output files. Each Mapper instance reads a partition (of size k) of the K files and generates a subset (of

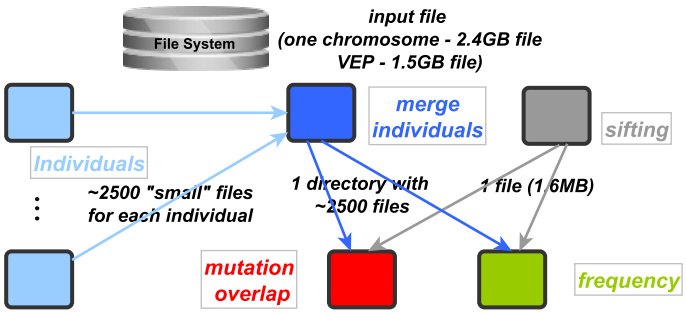

Figure 6.7: High-level view of the*1000 Genomes* use case. Workflow steps for one chromosome.

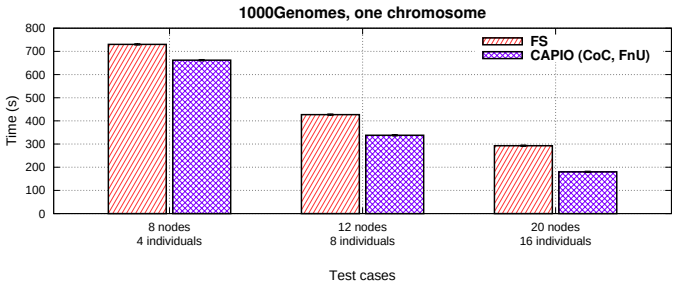

Figure 6.8: Results for the *1000 Genomes* use case. Execution time of the C++-based version obtained with the standard deployment (file system-based – FS) and with CAPIO on 8, 12, and 20 GALILEO100 nodes.

size m) of the output files. For example, with 4 Mappers and 3 Reducers, $K = 20$, and $M = 9$, each Mapper reads $k = 5$ files, and each Reducer produces $m = 3$ files. The final "Merge" step reads all M files and generates a single output file. The communication pattern between the first and second steps is a *1-to-Many* pattern, while the pattern between the second and third steps, involving consolidating data from multiple sources into a single destination, is a *Many-to-1* pattern.

In Figure 6.10, the execution time results are presented when running the workflow using the file system (FS) and CAPIO configured with *Commit on-Close* and *Firing no-Update* synchronization semantics. The input dataset is 72GB, and the output file is approximately 7GB. The parameters k and m, representing the number of files created by the "Split" and "MapReduce" steps, were varied in several configurations, as depicted in the charts on

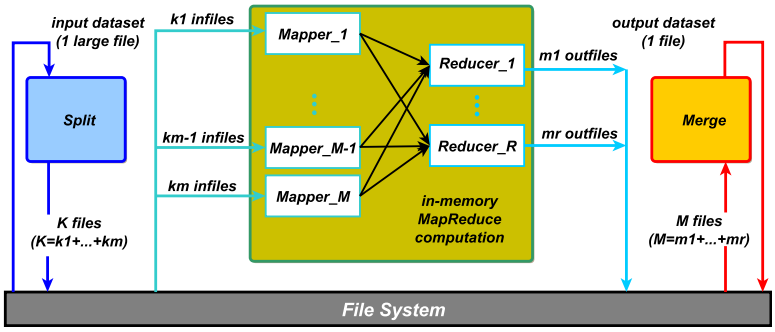

Figure 6.9: *MapReduce* use case. The Workflow is composed by the steps: Split, MapReduce and Merge.

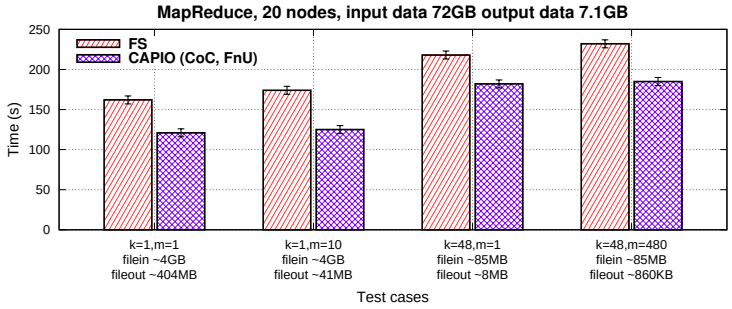

Figure 6.10: Results for the *MapReduce* use case. Execution time obtained with the standard deployment (file system-based – FS) and with CAPIO on 20 GALILEO100 nodes.

Figure 6.10. Increasing the number of files produced leads to a reduction in their size. Across all tests, the deployment of CAPIO consistently results in an execution time reduction ranging from 22-33%.

6.4.3 Weather Forecast Workflow

The Weahter Forecast workflow consists of two applications (Figure 6.11). The first application is the WRF model, which simulates and predicts the weather, and the second application (written in Python) reads the files produced by WRF and generates images representing the weather (Figure 6.12). Without CAPIO, the images can only be created after WRF has completed its execution. This is inefficient because WRF produces output for each simulated hour every 2/3 minutes of real execution. Users have to wait for the

entire WRF execution before starting to view the weather images. However, with CAPIO, we can run the two applications concurrently, and every 2/3 minutes, the user can see a new image of the simulation. In [125], the authors successfully integrated WRF with ADIOS2, resulting in improved performance compared to previous implementations, attributed to ADIOS2's in-situ capabilities. Interested readers are encouraged to refer to the paper for detailed integration procedures. The I/O module underwent rewriting to utilize the ADIOS2 API, and fine-tuning of ADIOS2 parameters was necessary. Notably, for the workflow used in this test, transitioning to the ADIOS2 version of WRF would have entailed rewriting the application responsible for the visualization, as it would require the use of ADIOS2 APIs to read the data produced by the modified version of WRF. In contrast, CAPIO obviates the need for such code modifications.

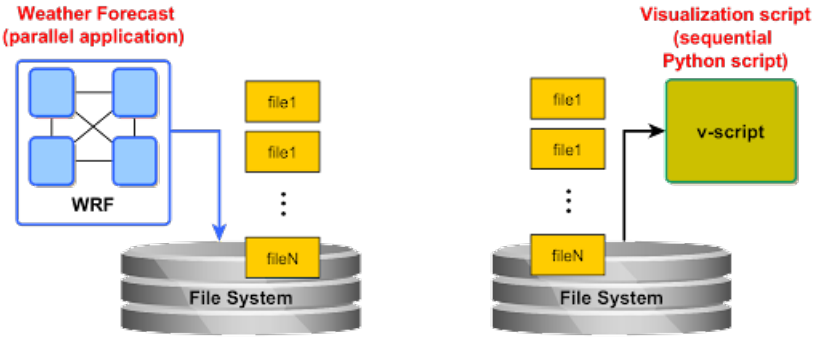

Figure 6.11: Weather Forecast workflow using the filesystem. Without CAPIO, the user has to wait for the WRF application to finish before being able to view the first image of the simulation.

The Listing 6.3 shows the CAPIO configuration file for this use case. The producer application, named "wrf" (line 6), generates an unspecified number of files in the working directory, depending on the runtime parameter related to the simulated hours. Given this uncertainty, the streaming commit rule for the working directory is set to "on termination" (line 8). Files created by the producer are not deleted; hence, the "mode" keyword is configured as "no update" (line 12). The produced files adhere to the committed rule of "on close" (line 14). An observation using the strace tool revealed that the initial bytes of the files undergoes 2-3 updates, making the firing rule inappropriate for "no update" (line 16). The consumer application, denoted

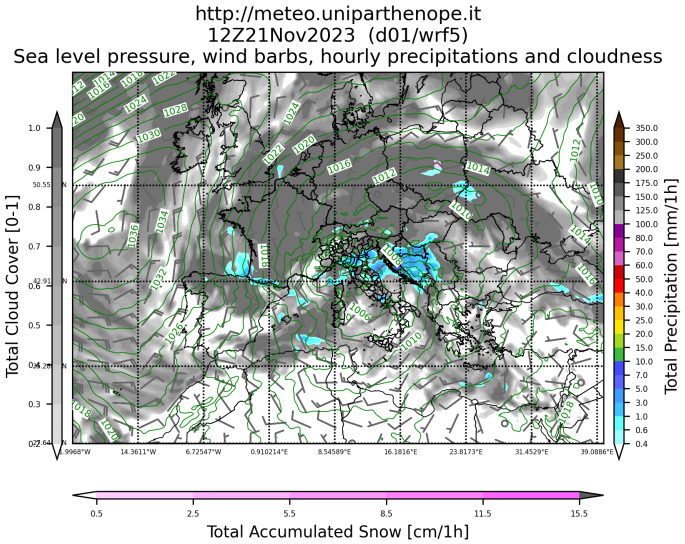

Figure 6.12: One of the image produced by the Weather Forecast workflow.

as "post processor" (line 21), lacks prior knowledge of the total number of files to read. Its source code resembles the pseudocode in listing 3, where it iterates over the entries in the working directory and reads the content of each file. CAPIO facilitates a seamless reading process, allowing the consumer to start reading a file as soon as the producer writes it. The consumer will receive the end-of-stream token after reading all directory entries and the producer's termination, as determined by the "on termination" commit rule for the working directory (line 26).

Listing 6.3: CAPIO configuration file for the WRF workflow using the CoC-FnU semantics.

```
1  {
2  "name" :"WRF_WORKFLOW",
3  "IO_Graph" :
4  [
5          {
6                  "name" :"WRF",
7                  "output_stream" :[".", "./*"],
8                  "streaming" :[
9                          {
10                                 "dirname" :["."],
11                                 "committed" :"on_termination",
12                                 "mode" :"no_update"
13                         },
14                         {
```

```
                        "name" :["./*"],
                        "committed" :"on_close"
                    }
                ]
        },
        {
                "name" :"visualization",
                "input_stream" :["./*"]
        }

]
}
```

Table 6.2 reports the execution time of the workflow with and without CAPIO. With CAPIO, there is an improvement in performance. The most noteworthy aspect is that when using 4 nodes, with CAPIO, the user can start viewing the first image after 80 seconds from the start of the workflow, while without CAPIO, it will take 761 seconds to see the first image.

Nodes	Weather Workflow POSIX (secs)	Weather Workflow CAPIO (secs)
1	4215.623	3737.944
2	3161.513	2315.045
4	2542.668	2099.501

Table 6.2: Execution times of Weather workflow without and with CAPIO.

Chapter 7

Conclusion

7.1 Final remarks

HPC workloads are experiencing a significant transition, shifting from monolithic applications towards workflows that incorporate a blend of co-engineered and legacy components. These components communicate through a portable file-based interface, utilizing the file system as a means of communication. This work introduced the CAPIO middleware, designed to integrate I/O streaming capabilities into file-based workflows seamlessly. CAPIO relies on an I/O coordination language, employing JSON syntax, empowering users to annotate workflow data dependencies with synchronization semantics, which extends the semantics of POSIX-based I/O system calls, enabling the temporal overlap of distinct workflow steps. We detailed the synchronization semantics currently supported by CAPIO and the corresponding language annotations specified in a JSON configuration file. Through synthetic benchmarks and examination of three workflow use cases, we substantiated the performance advantages of CAPIO compared to Lustre, BeeGFS, and ADSIO2 on two different production clusters. The results underscore the approach's feasibility, showcasing performance enhancements ranging (circa) from 20% to 30%, and in an edge case with 10000 files, CAPIO achieves a speedup of 8. We anticipate that tools similar to CAPIO may influence innovative workflow orchestration strategies, fostering improved temporal overlap between steps and reducing the peak I/O file system demand.

7.2 Limitations and Future work

As part of future work, the development of the CAPIO runtime will continue, with the addition of multiple backends, providing users the option to choose the backend for CAPIO. Currently, only MPI is supported, but in the future, the use of distributed filesystems and other communication layers such as ADIOS2 [14], UCX [146], or MTCL [127] will be incorporated. The language will be extended to allow users to specify the backend type in the JSON configuration file. Plans also include implementing home-node policies such as "hashing" and "manual". Multiple tests will be conducted to determine the optimal policy choices in everyday situations.

As part of maturing the CAPIO runtime, extensive testing on application workflows utilizing common high-level I/O interfaces like MPI-I/O, HDF5, and NETCDF, which are widely employed in scientific workflows, is essential. Another development direction for CAPIO is integration with Workflow Management Systems (WMS) such as StreamFlow [141] and DagOnStar[84]. A WMS can seamlessly link the CAPIO intercept library with the applications in a workflow and deploy CAPIO servers on the nodes required. The WMS may generate an initial version of the CAPIO configuration file, which users can then fine-tune for optimal data communication between applications. Typically, WMS uses a configuration file to describe applications and their data dependencies, which could be leveraged to create the IO-Graph section of the CAPIO configuration file automatically.

Another approach is to extend the language used by the WMS (for example, the Workflow Common Language) with the CAPIO streaming semantics. This way, the WMS can automatically add the streaming semantics to the CAPIO configuration file.

In recent years, malleability (i.e., the capacity to dynamically change the computational units assigned to an application [107, 147]) has become a nice-to-have feature of the HPC ecosystem [108, 109, 110, 111]. Currently, CAPIO does not support malleability because all the CAPIO servers must be launched in all the nodes where the applications will run. If one application dynamically obtains more nodes, it is not possible at the moment to extend CAPIO to the new nodes. For this reason, we plan to implement the capability of creating new CAPIO servers for a running workflow and to be resilient when a node is removed from the pool of nodes assigned to a workflow. In order to do that, integration with malleability tools such as FlexMPI [148] will be explored.

Acronyms

The following is the list of acronyms used throughout the document.

- **HPC**: High Performance Computing.
- **CAPIO**: Cross-Application Programmable IO.
- **CLIO**: Coordination Language for IO.
- **WMS**: Workflow Management System.
- **DSL**: Domain Specific Lnaguage.
- **CWL**: Common Workflow Language.
- **GUI**: Graphical User Interface.
- **API**: Application Programming Interface.
- **XML**: eXtensible Markup Language.
- **POSIX**: Portable Operating System Interface for Unix.
- **SC**: System Call.
- **CAPIO SC-IL**: CAPIO System Call Intercept Library.
- **JSON**: JavaScript Object Notation.
- **MPI**: Message Passing Interface.
- **EOF**: End Of File.
- **EOS**: End Of Stream.

- **CAPIO_DIR**: CAPIO Directory. The virtual mount-mount of the CAPIO infrastructure.
- **CAPIO conf. file**: It is the JSON file written according to the I/O coordination language syntax. It is get as an input file by each CAPIO server.
- **WRF**: Weather Research and Forecasting model.
- **CoC**: Commit on Close semantics.
- **CoT**: Commit on Termination semantics
- **CoF**: Commit on File semantics.
- **FoC**: Firing on Commit semantics.
- **FnU**: Firing on Update semantics.

Bibliography

[1] Gordon E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860.

[2] Morris A. Jette and Tim Wickberg. "Architecture of the Slurm Workload Manager". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dalibor Klusáček, Julita Corbalán, and Gonzalo P. Rodrigo. Cham: Springer Nature Switzerland, 2023, pp. 3–23. ISBN: 978-3-031-43943-8.

[3] A Grannan, K Sood, B Norris, and A Dubey. "Understanding the landscape of scientific software used on high-performance computing platforms". In: *The International Journal of High Performance Computing Applications* 34.4 (2020), pp. 465–477. DOI: 10.1177/1094342019899451.

[4] Marco Aldinucci, Valentina Cesare, Iacopo Colonnelli, Alberto Riccardo Martinelli, Gianluca Mittone, Barbara Cantalupo, Carlo Cavazzoni, and Maurizio Drocco. "Practical Parallelization of Scientific Applications with OpenMP, OpenACC and MPI". In: *Journal of Parallel and Distributed Computing* 157 (Jan. 1, 2021), 13–29. DOI: 10.1016/j.jpdc.2021.05.017. published.

[5] Gangman Yi and Vincenzo Loia. "High-performance computing systems and applications for AI". In: *J. Supercomput.* 75.8 (2019), 4248–4251. ISSN: 0920-8542. DOI: 10.1007/s11227-019-02937-z.

[6] E. A. Huerta, Asad Khan, Edward Davis, Colleen Bushell, William D. Gropp, Daniel S. Katz, Volodymyr Kindratenko, Seid Koric, William T. C. Kramer, Brendan McGinty, Kenton McHenry, and Aaron Saxton. "Convergence of artificial intelligence and high performance com-

puting on NSF-supported cyberinfrastructure". In: *Journal of Big Data* 7.1 (2020). ISSN: 2196-1115. DOI: 10.1186/s40537-020-00361-2.

[7] Iacopo Colonnelli, Bruno Casella, Gianluca Mittone, Yasir Arfat, Barbara Cantalupo, Roberto Esposito, Alberto Riccardo Martinelli, Doriana Medić, and Marco Aldinucci. "Federated Learning Meets HPC and Cloud". In: *Machine Learning for Astrophysics*. Ed. by Filomena Bufano, Simone Riggi, Eva Sciacca, and Francesco Schilliro. Cham: Springer International Publishing, 2023, pp. 193–199. ISBN: 978-3-031-34167-0.

[8] Rafael Ferreira da Silva, Rosa Filgueira, Ilia Pietri, Ming Jiang, Rizos Sakellariou, and Ewa Deelman. "A characterization of workflow management systems for extreme-scale applications". In: *Future Generation Comp. Syst.* 75 (2017). ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.02.026.

[9] John Bent, Gary Grider, Brett Kettering, Adam Manzanares, Meghan McClelland, Aaron Torres, and Alfred Torrez. "Storage challenges at Los Alamos National Lab". In: *MSST '12: Proc. of the 28th Symp. on Mass Storage Systems and Technologies*. IEEE Computer Society, 2012. DOI: 10.1109/MSST.2012.6232376.

[10] André Brinkmann, Kathryn Mohror, Weikuan Yu, Philip H. Carns, Toni Cortes, Scott Klasky, Alberto Miranda, Franz-Josef Pfreundt, Robert B. Ross, and Marc-Andre Vef. "Ad Hoc File Systems for High-Performance Computing". In: *J. Comput. Sci. Technol.* 35.1 (2020). DOI: 10.1007/s11390-020-9801-1.

[11] Rajeev Thakur, William Gropp, and Ewing L. Lusk. "A Case for Using MPI's Derived Datatypes to Improve I/O Performance". In: *Proc. of the ACM/IEEE Conference on Supercomputing, SC 1998, November 7-13, 1998, Orlando, FL, USA*. 1998. DOI: 10.1109/SC.1998.10006.

[12] Ciprian Docan, Manish Parashar, and Scott Klasky. "DataSpaces: An interaction and coordination framework for coupled simulation workflows". In: vol. 15. Jan. 2010. DOI: 10.1145/1851476.1851481.

[13] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Robert Sisneros, Orcun Yildiz, Shadi Ibrahim, Tom Peterka, and Leigh Orf. "Damaris: Addressing Performance Variability in Data Management

for Post-Petascale Simulations". In: *ACM Trans. Parallel Comput.* 3.3 (Oct. 2016). ISSN: 2329-4949. DOI: 10.1145/2987371.

[14] William Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Youl Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, and Scott Klasky. "ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management". In: *SoftwareX* 12 (2020). DOI: 10.1016/j.softx.2020.100561.

[15] Tu Mai Anh Do, Loïc Pottier, Orcun Yildiz, Karan Vahi, Patrycja Krawczuk, Tom Peterka, and Ewa Deelman. "Accelerating Scientific Workflows on HPC Platforms with In Situ Processing". In: *CCGrid'22: 22nd IEEE Intern. Symp. on Cluster, Cloud and Internet Comp.* 2022. DOI: 10.1109/CCGrid54584.2022.00009.

[16] Janine C. Bennett, Hasan Abbasi, Peer-Timo Bremer, Ray Grout, Attila Gyulassy, Tong Jin, Scott Klasky, Hemanth Kolla, Manish Parashar, Valerio Pascucci, Philippe Pebay, David Thompson, Hongfeng Yu, Fan Zhang, and Jacqueline Chen. "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis". In: *SC '12: Proc. of the Intern. Conf. on High Performance Computing, Networking, Storage and Analysis*. 2012. DOI: 10.1109/SC.2012.31.

[17] Christopher Sewell, Katrin Heitmann, Hal Finkel, George Zagaris, Suzanne T. Parete-Koon, Patricia K. Fasel, Adrian Pope, Nicholas Frontiere, Li-ta Lo, Bronson Messer, Salman Habib, and James Ahrens. "Large-scale compute-intensive analysis via a combined in-situ and co-scheduling workflow approach". In: *SC '15: Proc. of the Inter. Conf. for High Performance Computing, Networking, Storage and Analysis*. 2015. DOI: 10.1145/2807591.2807663.

[18] Alberto Riccardo Martinelli, Massimo Torquati, Marco Aldinucci, Iacopo Colonnelli, and Barbara Cantalupo. "CAPIO: a Middleware for Transparent I/O Streaming in Data-Intensive Workflows". In: *2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. Goa, India: IEEE, Dec. 2023. DOI: 10.1109/HiPC58850.2023.00031.

[19] Farhad Arbab. "Composition of Interacting Computations". In: *Interactive Computation: The New Paradigm*. Ed. by Dina Goldin, Scott A. Smolka, and Peter Wegner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 277–321. ISBN: 978-3-540-34874-0. DOI: 10.1007/3-540-34874-3_12.

[20] Ahuja, Carriero, and Gelernter. "Linda and Friends". In: *Computer* 19.8 (1986), pp. 26–34. DOI: 10.1109/MC.1986.1663305.

[21] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0262530864.

[22] Murray Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming". In: *Parallel Computing* 30.3 (2004), pp. 389–406. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2003.12.002.

[23] Marco Danelutto, Salvatore Orlando, and Susanna Pelagatti. "P 3 L : a Structured High-level Parallel Language , and itsStructured". In: 1993. URL: https://api.semanticscholar.org/CorpusID:8577376.

[24] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. "Fastflow: High-Level and Efficient Streaming on Multicore". In: Mar. 2014. ISBN: 9780470936900. DOI: 10.1002/9781119332015.ch13.

[25] David Astorga, Manuel F. Dolz, Javier Fernández, and José García. "A generic parallel pattern interface for stream and data processing". In: *Concurrency and Computation: Practice and Experience* 29 (May 2017), e4175–n/a. DOI: 10.1002/cpe.4175.

[26] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. "SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters". In: *Int. J. Parallel Program.* 49.6 (2021), 846–866. ISSN: 0885-7458. DOI: 10.1007/s10766-021-00704-3.

[27] Marco Vanneschi. "The programming model of ASSIST, an environment for parallel and distributed portable applications". In: *Parallel Computing* 28.12 (2002), pp. 1709–1732. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(02)00188-6.

[28] Jocelyn Serot. “TAGGED-TOKEN DATA-FLOW FOR SKELETONS”. In: *Parallel Processing Letters* 11.04 (2001), pp. 377–392. DOI: 10.1142/S0129626401000671.

[29] Holger Bishof, Sergei Gorlatch, and Roman Leshchinksiy. “DatTeL: A DATA-PARALLEL C++ TEMPLATE LIBRARY”. In: *Parallel Processing Letters* 13.03 (2003), pp. 461–472. DOI: 10.1142/S0129626403001422.

[30] Michael R. Crusoe, Sanne Abeln, Alexandru Iosup, Peter Amstutz, John Chilton, Nebojsa Tijanic, Hervé Ménager, Stian Soiland-Reyes, Bogdan Gavrilovic, Carole A. Goble, and The Cwl Community. “Methods included: standardizing computational reuse and portability with the Common Workflow Language”. In: *Communications of the ACM* 65.6 (2022). DOI: 10.1145/3486897.

[31] Javier Garcia-Blas, Genaro Sanchez-Gallegos, Cosmin Petre, Alberto Riccardo Martinelli, Marco Aldinucci, and Jesus Carretero. “Hercules: Scalable and Network Portable In-Memory Ad-Hoc File System for Data-Centric and High-Performance Applications”. In: *Euro-Par 2023: Parallel Processing*. Ed. by José Cano, Marios D. Dikaiakos, George A. Papadopoulos, Miquel Pericàs, and Rizos Sakellariou. Cham: Springer Nature Switzerland, 2023, pp. 679–693. ISBN: 978-3-031-39698-4.

[32] Rajkumar Buyya, Toni Cortes, and Hai Jin. “An Introduction to the InfiniBand Architecture”. In: *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. 2002, pp. 616–632. DOI: 10.1109/9780470544839.ch42.

[33] E. G. Coffman, M. Elphick, and A. Shoshani. “System Deadlocks”. In: *ACM Comput. Surv.* 3.2 (1971), 67–78. ISSN: 0360-0300. DOI: 10.1145/356586.356588.

[34] Saeed Maleki, Yaoqing Gao, Maria J. Garzar´n, Tommy Wong, and David A. Padua. “An Evaluation of Vectorizing Compilers”. In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. 2011, pp. 372–382. DOI: 10.1109/PACT.2011.68.

[35] Ian Foster and Carl Kesselman, eds. *The grid: blueprint for a new computing infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN: 1558604758.

[36] Jan Heichler. "An introduction to BeeGFS". In: *Introductionto BeeGFS by ThinkParQ. pdf* (2014).

[37] Peter J Braam and Philip Schwan. "Lustre: The intergalactic file system". In: *Ottawa Linux Symp.* Vol. 8. 11. 2002.

[38] Frank Schmuck and Roger Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters". In: *Proc. of the 1st USENIX Conference on File and Storage Technologies*. FAST '02. Monterey, CA: USENIX Assoc., 2002.

[39] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. "Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015, pp. 1–9. DOI: 10.1109/HOTI.2015.22.

[40] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Design Guidelines for High Performance RDMA Systems". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 437–450. ISBN: 978-1-931971-30-0. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[41] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. "The Spack package manager: bringing order to HPC software chaos". In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12. DOI: 10.1145/2807591.2807623.

[42] David Luebke. "CUDA: Scalable parallel programming for high-performance scientific computing". In: *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. 2008, pp. 836–838. DOI: 10.1109/ISBI.2008.4541126.

[43] Nvidia Cray CAPS and PGI. *openacc*. https://www.openacc.org/. Accessed on 2024-04-21.

[44] Edsger Wybe Dijkstra. *Cooperating Sequential Processes, Technical Report EWD-123*. Tech. rep. Technological University Eindhoven, 1965. URL: https://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF.

[45] C. A. R. Hoare. "Monitors: an operating system structuring concept". In: *Commun. ACM* 17.10 (1974), 549–557. ISSN: 0001-0782. DOI: 10.1145/355620.361161.

[46] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Commun. ACM* 21.7 (1978), 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563.

[47] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176.

[48] K. Mani Chandy and Leslie Lamport. "Distributed snapshots: determining global states of distributed systems". In: *ACM Trans. Comput. Syst.* 3.1 (1985), 63–75. ISSN: 0734-2071. DOI: 10.1145/214451.214456.

[49] Leslie Lamport. "The part-time parliament". In: *ACM Trans. Comput. Syst.* 16.2 (1998), 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229.

[50] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. 2nd. (Revised). Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262692155.

[51] FARHAD ARBAB. "Reo: a channel-based coordination model for component composition". In: *Mathematical Structures in Computer Science* 14.3 (2004), 329–366. DOI: 10.1017/S0960129504004153.

[52] Leslie G. Valiant. "A bridging model for parallel computation". In: *Commun. ACM* 33.8 (1990), 103–111. ISSN: 0001-0782. DOI: 10.1145/79173.79181.

[53] Carl Hewitt, Peter Bishop, and Richard Steiger. "A universal modular ACTOR formalism for artificial intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, 235–245.

[54] C. A. R. Hoare. "Communicating sequential processes". In: *Commun. ACM* 26.1 (1983), 100–106. ISSN: 0001-0782. DOI: 10.1145/357980.358021.

[55] Robin Milner. "Functions as processes". In: *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*. Warwick University, England: Springer-Verlag, 1990, 167–180. ISBN: 0387528261.

[56] Robin Milner, Joachim Parrow, and David Walker. "A calculus of mobile processes, I". In: *Information and Computation* 100.1 (1992), pp. 1–40. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90008-4.

[57] Robin Milner, Joachim Parrow, and David Walker. "A calculus of mobile processes, II". In: *Information and Computation* 100.1 (1992), pp. 41–77. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90009-5.

[58] Robin Milner, Joachim Parrow, and David Walker. "A calculus of mobile processes, II". In: *Information and Computation* 100.1 (1992), pp. 41–77. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90009-5.

[59] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[60] Chuck Pheatt. "Intel® threading building blocks". In: *J. Comput. Sci. Coll.* 23.4 (2008), p. 298. ISSN: 1937-4771.

[61] Robert D. Blumofe and Charles E. Leiserson. "Scheduling multithreaded computations by work stealing". In: *J. ACM* 46.5 (1999), 720–748. ISSN: 0004-5411. DOI: 10.1145/324133.324234.

[62] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. "A library of constructive skeletons for sequential style of parallel programming". In: *Proceedings of the 1st International Conference on Scalable Information Systems*. InfoScale '06. Hong Kong: Association for Computing Machinery, 2006, 13–es. ISBN: 1595934286. DOI: 10.1145/1146847.1146860.

[63] Christopher Brown, Vladimir Janjic, Adam Barwell, J. Garcia, and Kenneth MacKenzie. "Refactoring GrPPI: Generic Refactoring for Generic Parallelism in C++". In: *International Journal of Parallel Programming* 48 (Aug. 2020). DOI: 10.1007/s10766-020-00667-x.

[64] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. "Targeting distributed systems in fastflow". In: *Proceedings of the 18th International Conference on Parallel Processing Workshops*. Euro-Par'12. Rhodes Island, Greece: Springer-Verlag, 2012, 47–56. ISBN: 9783642369483. DOI: 10.1007/978-3-642-36949-0_7.

[65] Frederica Darema. "The SPMD Model: Past, Present and Future". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Yiannis Cotronis and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–1. ISBN: 978-3-540-45417-5.

[66] David R. Butenhof. *Programming with POSIX Threads*. Professional Computing Series. Addison-Wesley, 1997. ISBN: 0201633922.

[67] Aaftab Munshi. "The OpenCL specification". In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. 2009, pp. 1–314. DOI: 10.1109/HOTCHIPS.2009.7478342.

[68] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. USA: Wiley-Interscience, 2003. ISBN: 0471220485.

[69] Yili Zheng, Amir Kamil, Michael Driscoll, Hongzhang Shan, and Katherine Yelick. "UPC++: A PGAS extension for C++". In: May 2014, pp. 1105–1114. ISBN: 978-1-4799-3800-1. DOI: 10.1109/IPDPS.2014.115.

[70] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: Association for Computing Machinery, 2005, 519–538. ISBN: 1595930310. DOI: 10.1145/1094811.1094852.

[71] D. Callahan, B.L. Chamberlain, and H.P. Zima. "The cascade high productivity language". In: *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* 2004, pp. 52–60. DOI: 10.1109/HIPS.2004.1299190.

[72] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit". In: *The International Journal of High Performance Computing Applications* 20.2 (2006), pp. 203–231. DOI: 10.1177/1094342006064503. eprint: https://doi.org/10.1177/1094342006064503. URL: https://doi.org/10.1177/1094342006064503.

[73] Wolfgang Reisig and Grzegorz Rozenberg, eds. *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996.* Vol. 1491. Lecture Notes in Computer Science. Springer, 1998. ISBN: 3-540-65306-6. DOI: 10.1007/3-540-65306-6.

[74] E.A. Lee and T.M. Parks. "Dataflow Process Networks". In: *Proc. of the IEEE* 83.5 (May 1995).

[75] J. Dean Brock. "A formal model of non-determinate dataflow computation". PhD thesis. Massachusetts Institute of Technology, 1983.

[76] Dask Development Team. *Dask: Library for dynamic task scheduling.* 2016. URL: https://dask.org.

[77] Fabrizio Marozzo, Francesc Lordan, Roger Rafanell, Daniele Lezzi, Domenico Talia, and Rosa M. Badia. "Enabling Cloud Interoperability with COMPSs". In: *Euro-Par 2012 Parallel Processing.* Ed. by Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-32820-6.

[78] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. "Ray: A Distributed Framework for Emerging AI Applications". In: *13th USENIX Symp. on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.* 2018.

[79] Henry G. Baker and Carl Hewitt. "The incremental garbage collection of processes". In: *Proc. of the 1977 Symp. on Artificial Intelligence and Programming Languages, USA, August 15-17, 1977*. ACM, 1977. DOI: 10.1145/800228.806932.

[80] Claudia Misale, Maurizio Drocco, Marco Aldinucci, and Guy Tremblay. "A Comparison of Big Data Frameworks on a Layered Dataflow Model". In: *Parallel Processing Letters* 27.01 (2017). DOI: 10.1142/S0129626417400035.

[81] Johannes Köster and Sven Rahmann. "Snakemake - a scalable bioinformatics workflow engine". In: *Bioinformatics* 28.19 (2012). DOI: 10.1093/bioinformatics/bts480.

[82] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and R. Kent Wenger. "Pegasus, a workflow management system for science automation". In: *Future Generation Comp. Syst.* 46 (2015). DOI: 10.1016/j.future.2014.10.008.

[83] Raffaele Montella, Diana Di Luccio, and Sokol Kosta. "DagOn*: Executing Direct Acyclic Graphs as Parallel Jobs on Anything". In: *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. 2018, pp. 64–73. DOI: 10.1109/WORKS.2018.00012.

[84] Dante Domizzi Sánchez-Gallegos, Diana Di Luccio, Sokol Kosta, José Luis González Compeán, and Raffaele Montella. "An efficient pattern-based approach for workflow supporting large-scale science: The DagOnStar experience". In: *Future Gener. Comput. Syst.* 122 (2021). DOI: 10.1016/j.future.2021.03.017.

[85] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew B. Jones, Edward A. Lee, Jing Tao, and Yang Zhao. "Scientific workflow management and the Kepler system". In: *Concurrency and Computation: Practice and Experience* 18.10 (2006). DOI: 10.1002/cpe.994.

[86] Iacopo Colonnelli, Marco Aldinucci, Barbara Cantalupo, Luca Padovani, Sergio Rabellino, Concetto Spampinato, Roberto Morelli, Rosario Di Carlo, Nicolò Magini, and Carlo Cavazzoni. "Distributed workflows with Jupyter". In: *Future Generation Comp. Syst.* 128 (2022). ISSN: 0167-739X. DOI: 10.1016/j.future.2021.10.007.

[87] Rafael Ferreira da Silva, Rosa Filgueira, Ewa Deelman, Erola Pairo-Castineira, Ian Michael Overton, and Malcolm P. Atkinson. "Using simple PID-inspired controllers for online resilient resource management of distributed scientific workflows". In: *Future Generation Comp. Syst.* 95 (2019). DOI: 10.1016/j.future.2019.01.015.

[88] Christopher Sewell, Katrin Heitmann, Hal Finkel, George Zagaris, Suzanne T. Parete-Koon, Patricia K. Fasel, Adrian Pope, Nicholas Frontiere, Li-ta Lo, Bronson Messer, Salman Habib, and James Ahrens. "Large-scale compute-intensive analysis via a combined in-situ and co-scheduling workflow approach". In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–11. DOI: 10.1145/2807591.2807663.

[89] Matthieu Dreher, Swann Perarnau, Tom Peterka, Kamil Iskra, and Pete Beckman. "In Situ Workflows at Exascale: System Software to the Rescue". In: *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*. ISAV'17. Denver, CO, USA: Association for Computing Machinery, 2017, 22–26. ISBN: 9781450351393. DOI: 10.1145/3144769.3144774.

[90] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C. Bauer, Pat Marion, Berk Gevecik, Michel Rasquin, and Kenneth E. Jansen. "The ParaView Coprocessing Library: A scalable, general purpose in situ visualization library". In: *2011 IEEE Symposium on Large Data Analysis and Visualization*. 2011, pp. 89–96. DOI: 10.1109/LDAV.2011.6092322.

[91] Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. "Parallel in situ coupling of simulation with a fully featured visualization system". In: *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*. EGPGV '11. Llandudno, UK: Eurographics Association, 2011, 101–109. ISBN: 9783905674323.

[92] Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. "Taverna: lessons in creating a workflow environment for the life sciences". In: *Concurrency and Computation: Practice and Experience* 18.10 (2006),

pp. 1067–1100. DOI: https://doi.org/10.1002/cpe.993. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.993. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.993.

[93] Enis Afgan, Dannon Baker, Marius van den Beek, Daniel Blankenberg, Dave Bouvier, Martin Čech, John Chilton, Dave Clements, Nate Coraor, Carl Eberhard, Björn Grüning, Aysam Guerler, Jennifer Hillman-Jackson, Greg Von Kuster, Eric Rasche, Nicola Soranzo, Nitesh Turaga, James Taylor, Anton Nekrutenko, and Jeremy Goecks. "The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update". In: *Nucleic Acids Research* 44.W1 (May 2016), W3–W10. ISSN: 0305-1048. DOI: 10.1093/nar/gkw343. eprint: https://academic.oup.com/nar/article-pdf/44/W1/W3/7633114/gkw343.pdf. URL: https://doi.org/10.1093/nar/gkw343.

[94] Douglas Thain, Todd Tannenbaum, and Miron Livny. "Distributed computing in practice: the Condor experience". In: *Concurrency and Computation: Practice and Experience* 17.2-4 (2005), pp. 323–356. DOI: https://doi.org/10.1002/cpe.938. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.938. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.938.

[95] Marc-André Vef, Nafiseh Moti, Tim Süß, Markus Tacke, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. "GekkoFS — A Temporary Burst Buffer File System for HPC Applications". In: *Journal of Computer Science and Technology* 35 (Jan. 2020). DOI: 10.1007/s11390-020-9797-6.

[96] B. Nitzberg and V. Lo. "Distributed shared memory: a survey of issues and algorithms". In: *Computer* 24.8 (1991), pp. 52–60. DOI: 10.1109/2.84877.

[97] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, Honghui Lu, R. Rajamony, Weimin Yu, and W. Zwaenepoel. "TreadMarks: shared memory computing on networks of workstations". In: *Computer* 29.2 (1996), pp. 18–28. DOI: 10.1109/2.485843.

[98] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. "Latency-Tolerant Software Distributed Shared Memory". In: *2015 USENIX Annual Technical*

Conference (USENIX ATC 15). Santa Clara, CA: USENIX Association, 2015, pp. 291–305. ISBN: 978-1-931971-225. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/nelson.

[99] 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. Portland, Oregon, USA: Association for Computing Machinery, 2015, 33–44. ISBN: 9781450335508. DOI: 10.1145/2749246.2749269.

[100] Glenn K. Lockwood, Shane Snyder, Teng Wang, Suren Byna, Philip Carns, and Nicholas J. Wright. "A Year in the Life of a Parallel File System". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2018, pp. 931–943. DOI: 10.1109/SC.2018.00077.

[101] Rajeev Thakur, Alok Choudhary, Rajesh Bordawekar, Sachin More, and Sivaramakrishna Kuditipudi. "Passion: Optimized I/O for Parallel Applications". In: *Computer* 29.6 (1996), 70–78. ISSN: 0018-9162.

[102] *Apache Arrow*. https://arrow.apache.org. Accessed on 2024-04-21.

[103] Mike Folk, Albert Cheng, and Kim Yates. "HDF5: A file format and I/O library for high performance computing applications". In: *Proc. of supercomputing*. Vol. 99. 1999.

[104] Aditya Rajgarhia and Ashish Gehani. "Performance and extension of user space file systems". In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. Sierre, Switzerland: Association for Computing Machinery, 2010, 206–213. ISBN: 9781605586397. DOI: 10.1145/1774088.1774130.

[105] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. "To FUSE or Not to FUSE: Performance of User-Space File Systems". In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, Feb. 2017, pp. 59–72. ISBN: 978-1-931971-36-2. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor.

[106] *syscall_intercept*. https://github.com/pmem/syscall_intercept. Accessed on 2024-04-21.

[107] Dror G. Feitelson and Larry Rudolph. "Toward convergence in job schedulers for parallel supercomputers". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror G. Feitelson and Larry Rudolph. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–26. ISBN: 978-3-540-70710-3.

[108] Eishi Arima, A. Isaías Comprés, and Martin Schulz. "On the Convergence of Malleability and the HPC PowerStack: Exploiting Dynamism in Over-Provisioned and Power-Constrained HPC Systems". In: *High Performance Computing. ISC High Performance 2022 International Workshops*. Ed. by Hartwig Anzt, Amanda Bienz, Piotr Luszczek, and Marc Baboulin. Cham: Springer International Publishing, 2022, pp. 206–217. ISBN: 978-3-031-23220-6.

[109] Jose I. Aliaga, Maribel Castillo, Sergio Iserte, Iker Martín-Álvarez, and Rafael Mayo. "A Survey on Malleability Solutions for High-Performance Distributed Computing". In: *Applied Sciences* 12.10 (2022). ISSN: 2076-3417. DOI: 10.3390/app12105231. URL: https://www.mdpi.com/2076-3417/12/10/5231.

[110] Marc-André Vef, Alberto Miranda, Ramon Nou, and André Brinkmann. "From Static to Malleable: Improving Flexibility and Compatibility in Burst Buffer File Systems". In: *High Performance Computing*. Ed. by Amanda Bienz, Michèle Weiland, Marc Baboulin, and Carola Kruse. Cham: Springer Nature Switzerland, 2023, pp. 3–15. ISBN: 978-3-031-40843-4.

[111] Genaro Sanchez-Gallegos, Javier Garcia-Blas, Cosmin Petre, and Jesus Carretero. "Malleable and Adaptive Ad-Hoc File System for Data Intensive Workloads in HPC Applications". In: *High Performance Computing*. Ed. by Amanda Bienz, Michèle Weiland, Marc Baboulin, and Carola Kruse. Cham: Springer Nature Switzerland, 2023, pp. 56–67. ISBN: 978-3-031-40843-4.

[112] *BeeOND - BeeGFS On Demand*. https://doc.beegfs.io/latest/advanced_topics/beeond.html. Accessed on 2024-04-21.

[113] Felix Garcia-Carballeira, Diego Camarmas-Alonso, Alejandro Calderon-Mateos, and Jesus Carretero. "A new Ad-Hoc parallel file system for HPC environments based on the Expand parallel file system". In: *2023 22nd International Symposium on Parallel and Distributed Computing (ISPDC)*. 2023, pp. 69–76. DOI: 10.1109/ISPDC59212.2023.00015.

[114] Alejandro Calderón, Félix García, Jesús Carretero, Jose M Pérez, and Javier Fernández. "An implementation of MPI-IO on Expand: A parallel file system based on NFS servers". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2002, pp. 306–313.

[115] Félix Garcia-Carballeira, Alejandro Calderon, Jesus Carretero, Javier Fernandez, and Jose M Perez. "The design of the Expand parallel file system". In: *The International Journal of High Performance Computing Applications* 17.1 (2003), pp. 21–37.

[116] Michael J. Brim, Adam T. Moody, Seung-Hwan Lim, Ross Miller, Swen Boehm, Cameron Stanavige, Kathryn M. Mohror, and Sarp Oral. "UnifyFS: A User-level Shared File System for Unified Access to Distributed Local Storage". In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2023, pp. 290–300. DOI: 10.1109/IPDPS54959.2023.00037.

[117] Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror, and Dhabaleswar K. (DK) Panda. "A 1 PB/s file system to checkpoint three million MPI tasks". In: *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '13. New York, New York, USA: Association for Computing Machinery, 2018, 143–154. ISBN: 9781450319102. DOI: 10.1145/2462902.2462908.

[118] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. "An Ephemeral Burst-Buffer File System for Scientific Applications". In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 807–818. DOI: 10.1109/SC.2016.68.

[119] A. Miranda, R. Nou, and T. Cortes. "ECHOFS: A Scheduler-Guided Temporary Filesystem to Leverage Node-Local NVMS". In: *2018 30th Inter. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2018. DOI: 10.1109/CAHPC.2018.8645894.

[120] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman. "Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures". In: *SC '10: Proc. of the 2010 ACM/IEEE Inter. Conf. for High Performance Computing, Networking, Storage and Analysis*. 2010. DOI: 10.1109/SC.2010.28.

[121] A. Miranda, A. Jackson, T. Tocci, I. Panourgias, and R. Nou. "NORNS: Extending Slurm to Support Data-Driven Workflows through Asynchronous Data Staging". In: *2019 IEEE Inter. Conf. on Cluster Computing (CLUSTER)*. 2019. DOI: 10.1109/CLUSTER.2019.8891014.

[122] Viacheslav Dubeyko. *Comparative Analysis of Distributed and Parallel File Systems' Internal Techniques*. 2019. arXiv: 1904.03997 [cs.DC].

[123] Priyam Shah, Jie Ye, and Xian-He Sun. *Survey the storage systems used in HPC and BDA ecosystems*. 2021. arXiv: 2112.12142 [cs.DC].

[124] "Lustre : A Scalable , High-Performance File System Cluster". In: 2003. URL: https://api.semanticscholar.org/CorpusID:16120094.

[125] Erick Fredj, Yann Delorme, Sameeh Jubran, Mark Wasserman, Zhaohui Ding, and Michael Laufer. *accelerating wrf i/o performance with adios2 and network-based streaming*. 2023. arXiv: 2304.06603 [cs.DC].

[126] François Mazen, Lucas Givord, and Charles Gueunet. "Catalyst-ADIOS2: In Transit Analysis for Numerical Simulations Using Catalyst 2 API". In: *High Performance Computing*. Ed. by Amanda Bienz, Michèle Weiland, Marc Baboulin, and Carola Kruse. Cham: Springer Nature Switzerland, 2023, pp. 269–276. ISBN: 978-3-031-40843-4.

[127] Federico Finocchio, Nicolo Tonci, and Massimo Torquati. "MTCL: a Multi-Transport Communication Library". In: *1st International Workshop on Scalable Compute Continuum (WSCC 2023), co-located with Euro-Par '23, Cyprus*. LNCS 14351, ISBN 978-3-031-48802-3. Springer Verlag, 2023.

[128] Ciro Giuseppe De Vita, Dario Caramiello, Gennaro Mellone, Genaro Sánchez-Gallegos, Dante Domizzi Sánchez-Gallegos, Valeria Mele, Stefania Cavallo, and Diana Di Luccio. "An ad-hoc file system accelerated workflow application for accidental fire fast response". In: *Proceedings of the 2nd Workshop on Workflows in Distributed Environments*. WiDE '24. Athens, Greece: Association for Computing Machinery,

2024, 21–27. ISBN: 9798400705465. DOI: 10.1145/3642978.3652836. URL: https://doi.org/10.1145/3642978.3652836.

[129] Simone Perrotta, Giuseppe De Vita, Gennaro Mellone, Marco Edoardo Santimaria, Giuseppe Salvi, Marco Lapenga, Massimo Torquati, and Angelo Ciaramella. "Extending a scientific workflow engine with streaming I/O capabilities: DAGonStar and CAPIO". In: *1st Workshop on High-Performance eScience (HiPES), Workshop co-located with the Euro-Par 2024 conference*. Madrid, Spain, 2024.

[130] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. "A Configurable Rule based Classful Token Bucket Filter Network Request Scheduler for the Lustre File System". In: *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pp. 1–12.

[131] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. "Managing Variability in the IO Performance of Petascale Storage Systems". In: *SC '10: Proc. of the 2010 ACM/IEEE Inter. Conf. for High Performance Computing, Networking, Storage and Analysis*. 2010. DOI: 10.1109/SC.2010.32.

[132] Javier Garcia-Blas, David E. Singh, and Jesus Carretero. "IMSS: In-Memory Storage System for Data Intensive Applications". In: *HPC-MALL 2022. ISC High Performance 2022*. 2022.

[133] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. "PVFS: A Parallel File System for Linux Clusters". In: *4th Annual Linux Showcase & Conference (ALS 2000)*. Atlanta, GA: USENIX Assoc., Oct. 2000.

[134] *UnifyFS: A Distributed Burst Buffer File System*. https://github.com/LLNL/UnifyFS. Accessed on 2024-04-21.

[135] Chen Wang, Kathryn Mohror, and Marc Snir. "File System Semantics Requirements of HPC Applications". In: *Proc. of the 30th Inter. Symp. on High-Performance Parallel and Distributed Computing*. HPDC '21. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450382175. DOI: 10.1145/3431379.3460637.

[136] Salvatore Di Girolamo, Daniele De Sensi, Konstantin Taranov, Milos Malesevic, Maciej Besta, Timo Schneider, Severin Kistler, and Torsten Hoefler. "Building Blocks for Network-Accelerated Distributed File Systems". In: *Proc. of the Inter. Conf. on High Performance Computing, Networking, Storage and Analysis*. SC '22. Dallas, Texas: IEEE Press, 2022. ISBN: 9784665454445.

[137] Henrique Andrade, Buğra Gedik, and Deepak Turaga. *Fundamentals of Stream Processing*. Cambridge Books. Cambridge University Press, 2014. ISBN: 9781139058940.

[138] *Common Workflow Language User Guide v1.2*. http://www.commonwl.org/user_guide. Accessed on 2024-04-21.

[139] John B. Carter, John K. Bennett, and Willy Zwaenepoel. "Implementation and performance of Munin". In: *SIGOPS Oper. Syst. Rev.* 25.5 (1991), 152–164. ISSN: 0163-5980. DOI: 10.1145/121133.121159. URL: https://doi.org/10.1145/121133.121159.

[140] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. "To FUSE or Not to FUSE: Performance of User-Space File Systems". In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Assoc., Feb. 2017. ISBN: 978-1-931971-36-2.

[141] Iacopo Colonnelli, Barbara Cantalupo, Ivan Merelli, and Marco Aldinucci. "StreamFlow: cross-breeding cloud with HPC". In: *IEEE Trans. on Emerging Topics in Computing* 9.4 (2021). DOI: 10.1109/TETC.2020.3019202.

[142] *The 1000 Genomes Project*. https://www.internationalgenome.org/1000-genomes-summary. Accessed on 2024-04-21.

[143] Jordan Powers, Joseph Klemp, William Skamarock, Christopher Davis, Jimy Dudhia, David Gill, Janice Coen, David Gochis, Ravan Ahmadov, Steven Peckham, Georg Grell, John Michalakes, Samuel Trahan, Stanley Benjamin, Curtis Alexander, Geoffrey Dimego, Wei Wang, Craig Schwartz, Glen Romine, and Michael Duda. "The Weather Research and Forecasting (WRF) Model: Overview, System Efforts, and Future Directions". In: *Bulletin of the American Meteorological Society* 98 (Jan. 2017). DOI: 10.1175/BAMS-D-15-00308.1.

[144] Larry McVoy and Carl Staelin. "Lmbench: Portable Tools for Performance Analysis". In: *ATEC '96: Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*. USENIX Association, 1996.

[145] Saeed Shahrivari. "Beyond Batch Processing: Towards Real-Time and Streaming Big Data". In: *Computers* 3.4 (2014). ISSN: 2073-431X. DOI: 10.3390/computers3040117.

[146] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. "UCX: An Open Source Framework for HPC Network APIs and Beyond". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015, pp. 40–43. DOI: 10.1109/HOTI.2015.13.

[147] Jesus Carretero, Javier Garcia-Blas, André Brinkmann, Marc Vef, Jean-Baptiste Besnard, Massimo Torquati, Yi Ju, and Raffaele Montella. "Adaptive HPC Input/Output Systems". In: *Euro-Par 2023: Parallel Processing Workshops*. Ed. by Demetris Zeinalipour, Dora Blanco Heras, George Pallis, Herodotos Herodotou, Demetris Trihinas, Daniel Balouek, Patrick Diehl, Terry Cojean, Karl Fürlinger, Maja Hanne Kirkeby, Matteo Nardellli, and Pierangelo Di Sanzo. Cham: Springer Nature Switzerland, 2024, pp. 199–202. ISBN: 978-3-031-48803-0.

[148] Jesus Carretero, David Exposito, Alberto Cascajo, and Raffaele Montella. "Malleability Techniques for HPC Systems". In: *Parallel Processing and Applied Mathematics: 14th International Conference, PPAM 2022, Gdansk, Poland, September 11–14, 2022, Revised Selected Papers, Part II*. Gdansk, Poland: Springer-Verlag, 2023, 77–88. ISBN: 978-3-031-30444-6. DOI: 10.1007/978-3-031-30445-3_7.