H2020-JTI-EuroHPC-2019-1

Project no. 956748

ADAPTIVE MULTI-TIER INTELLIGENT DATA MANAGER FOR EXASCALE

D6.4
Coordination language for intelligent I/O systems

Version 1.0

Date: March 4, 2024

Type: Deliverable
WP number: WP6

Editor: Massimo Torquati
Institution: CINI

Project co-funded by the European Union Horizon 2020 JTI-EuroHPC research and innovation programme and Spain, Germany, France, Italy, Poland, and Sweden		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Change Log

Rev.	Date	Who	Site	What
1	15/12/23	Jesus Carretero	UC3M	Document created.
2	11/01/24	Massimo Torquati	CINI	Draft structure of the document.
3	29/01/24	Massimo Torquati	CINI	First version of Chapter 2 and 3.
4	02/02/24	Alberto R. Martinelli	CINI	Improvement of Language syntax and semantics.
5	02/02/24	Marco E. Santimaria	CINI	Improvement of Chapters 2 and 3.
6	02/02/24	Iacopo Colonnelli	CINI	Home node section sketch.
7	05/02/24	Massimo Torquati	CINI	First version of the Introduction chapter.
8	09/02/24	Alberto R. Martinelli	CINI	Improved home node policies description.
9	15/02/24	Massimo Torquati	CINI	Chapter 3 improved description.
10	18/02/24	Marco E. Santimaria	CINI	Improvement to Chapter 4.
11	23/02/24	Barbara Cantalupo	CINI	Improved the Introduction Chapter.
12	23/02/24	Massimo Torquati	CINI	Conclusion Chapter and Executive Summary.
13	01/03/24	Taylan Özden	TUD	Reviewed the document.
14	01/03/24	Rocco Sedona	FZJ	Reviewed the document.
15	04/03/24	Massimo Torquati	CINI	Final version.

Executive Summary

The ADMIRE project is dedicated to designing and implementing a platform where an active I/O stack can dynamically adjust computation and storage requirements through intelligent global coordination, the malleability of computation and I/O, and the scheduling of storage resources along all levels of the storage hierarchy.

In this deliverable, we present a complementary approach to the ad-hoc storage solution developed in the ADMIRE project, aiming to define an I/O coordination language and the related middleware component (CAPIO – Cross-Application Programmable I/O) with the primary objective of transparently injecting I/O streaming capabilities into file-based loosely-coupled workflows. By leveraging user insights provided through a JSON-based I/O coordination language (CLIO – Coordination Language for I/O), the CAPIO middleware improves computation-I/O overlap without the need to change the application code of the workflow modules. We first describe the declarative I/O coordination language as a primary component of the CAPIO infrastructure, focusing both on syntax and semantics features; then, we describe the CAPIO's runtime system. We also present two use cases, 1000 Genome and WRF-visualization workflows, to validate the expressivity of the I/O coordination language and the performance of the CAPIO runtime system.

Contents

Chapter 1

Introduction

Challenges in the current HPC scenario are strongly related to the growing amount of data to manage from the application side and a variety of new computing (e.g., accelerators) and storage (multi-tier hierarchy) solutions from the architectural one. The main objective of the ADMIRE project is to create an I/O stack that can dynamically fulfill computation and storage requirements through data analytics, runtime coordination, computation and I/O malleability, and scheduling of storage resources at all levels. The I/O coordination language proposed in this deliverable, with its related implementation, adds a new component to the ADMIRE dynamic storage management, offering a complementary solution exploiting unique users' knowledge.

1.1 ADMIRE Architecture

ADMIRE architecture comprises several active modules working collaboratively to execute multiple applications on an HPC Platform. It is designed to dynamically provide computational and I/O resources by integrating information coming from very different sources, including user requirements, resource availability, and application characteristics. The Intelligent Controller (IC) serves as the central orchestrator within the ADMIRE

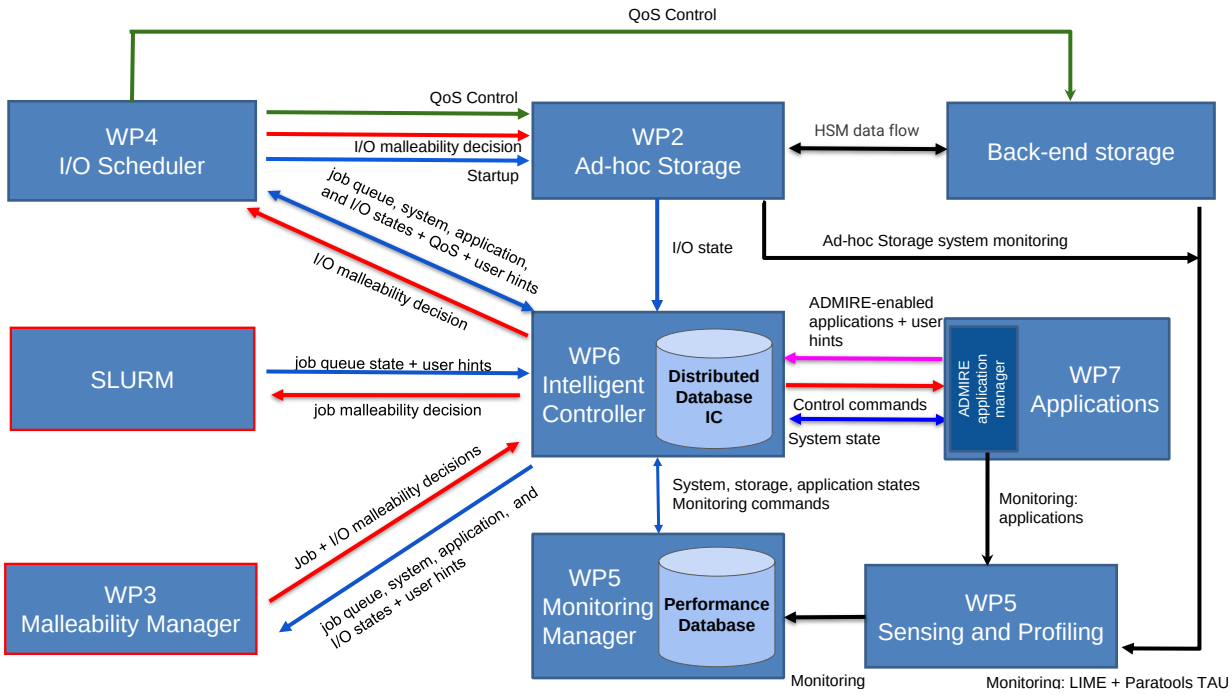

Figure 1.1: ADMIRE architecture overview, including for each component the related workpackage (WP).

architecture, facilitating communication and coordination among its various components. Figure 1.1 illustrates the main interactions between ADMIRE components and the information, data, and controls exchanged, also specifying the ADMIRE work packages (WP) where the components are developed. The backend storage, typically a parallel system such as BeeGFS[1] and Lustre[2], is accompanied by an ad-hoc storage system (WP2), that provides each application with a high-performance storage system tailored to the application's characteristics. Both storage tiers are coordinated by the I/O scheduler (WP4) responsible for the ad-hoc storage deployment and configuration, the specification of quality-of-service metrics, and the implementation of I/O scheduling policies. ADMIRE-enabled applications are executed on the platform using Slurm commands given by the IC, also considering suggestions from the malleability manager (WP3). Applications can provide user-defined application-specific information to the system to aid the identification of I/O patterns and reconfiguration-safe states in which malleability commands can be executed.

ADMIRE ad-hoc storage system [1] is a dynamic module that accelerates the I/O performance of scientific applications, significantly reducing I/O traffic to the general-purpose backend storage. Different components are integrated into the ad-hoc storage module, such as GekkoFS [2], Expand [3], and Hercules [4], and they are all mainly focused on optimizing the management of the general purpose storage backends. A complementary approach to the ad-hoc storage solution in ADMIRE has been investigated and implemented in task 6.4, leading to the definition of an I/O coordination language and the related middleware component, CAPIO [5]. This new approach capitalizes on the information provided by users to inject I/O streaming capabilities into file-based workflows, improving the computation-I/O overlap without needing to change the application code. In practice, CAPIO enables the transparent transformation of POSIX I/O system calls related to files (e.g., `read` and `write`) into data communications between producers and consumers of those files. Therefore, it enables efficient handling of data streaming between different applications (or components) of the applications workflow. By leveraging user-provided insights, such as data dependencies and characteristics of the data production (i.e., `writes`) and usage (`reads`), the system can tailor communication protocols to optimize performance and efficiency without changing the application code. This personalized approach enables the system to adapt to the unique demands of each workflow, exploiting unique users' knowledge and guaranteeing smooth and efficient data streaming processes throughout the application lifecycle.

From the ADMIRE perspective, this means adding a new component between the ad-hoc storage and the applications, which, in specific cases, can convert conventional I/O operations into streamlined communications, thereby mitigating traffic congestion and alleviating the burden of storage management. While ad-hoc storage remains essential for cases requiring persistent storage or when no additional hints are provided by the users, the complementary availability of this component enhances overall system performance and resource utilization.

1.2 I/O coordination in Workflow

Efficient I/O coordination is crucial in workflow management as it ensures seamless and efficient execution of application modules comprising the workflow while respecting input and output data dependencies. Input dependencies, typically in the form of data files in traditional file-based workflow, are the needed input data for executing the processing of workflow modules, while output dependencies include the resultant files generated upon processing the inputs. In situ workflow refers to a workflow, i.e., a collection of programs with their data dependencies, executing concurrently in a cluster system and communicating through the memory and interconnect of the cluster system instead of physical storage through the parallel/distributed filesystem. A classic example of in situ workflow is a simulation task coupled with the data analysis task so that the data analysis task starts while data produced by the simulation is still resident in memory, rather than after having saved all the output data to disk for post-processing.

Coordinating workflow modules requires clearly defining the source of inputs, the dependencies associated with each module working as intermediate steps of the workflow, and the sink modules, i.e., those responsible for providing the final results. The aim is to overlap the computation of the workflow modules with the com-

[1] https://www.beegfs.io/c/
[2] https://www.lustre.org/

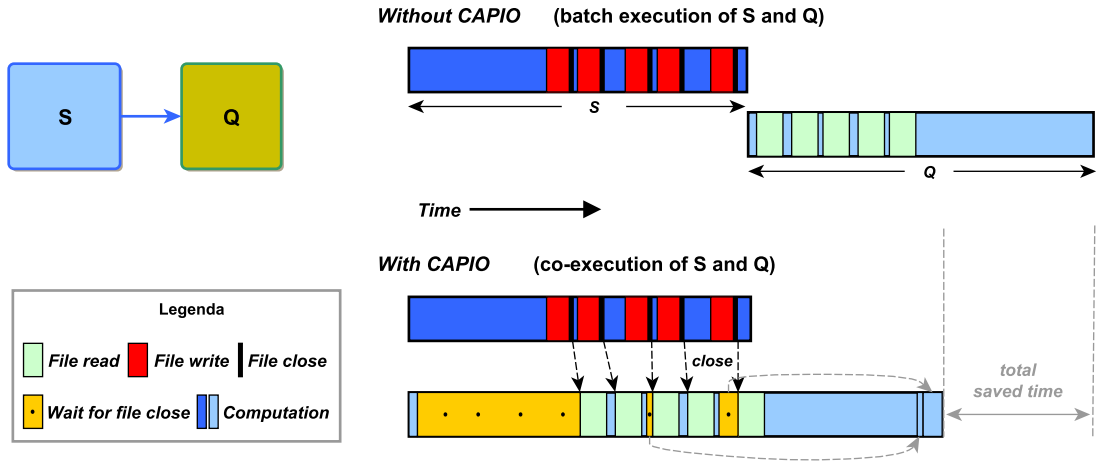

Figure 1.2: Schematic example of a two-step file-based workflow in which application modules (S and Q) communicate using files. On the right-hand side are sketched two possible executions: loosely-coupled batch execution of the S and Q vs. their concurrent execution with the CAPIO Middleware.

munication needed to exchange data among modules to optimize the makespan. This can be accomplished only by carefully coordinating I/O data dependencies among different workflow modules to start the execution of modules as soon as their input data dependencies have been produced.

Tools in charge of exposing coordination semantics to the users and orchestrating workflow executions are called *Workflow Management Systems* (WMSs) [6]. 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) [7], whereas others adopt a general-purpose programming language (e.g., Pegasus [8]) or a GUI (e.g., Jupyter Workflow [9]). These approaches are very flexible, as they do not impose constraints on the host application code and do not require any modification to the business logic (a job in a workflow is often seen as a black box from the WMS viewpoint). 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 output data dependencies are propagated to consumer modules only after the producers have terminated their execution and thus consolidated all produced data. The *on-termination* semantics forces loosely-coupled execution (i.e., batch execution) of the directly connected workflow components. We will discuss the *on-termination* semantics in the context of Workflows in Section 2.2.

In HPC facilities, the contention for the I/O bandwidth of the shared file system is often the main obstacle to scalability in traditional file-based workflows. Considering a two-step pipeline workflow sketched in Figure 1.2. Step S produces k files, consumed as input data by the second step Q. The files are stored on a shared file system, which sustains wB and rB write and read bandwidths that non-linearly depend on the file size. For the sake of simplicity, suppose that files are equally sized and bandwidth is constant. If S writes files of size N and Q reads files of M bytes, then the makespan T_T depends on the compute time $T_C = T_C^S + T_C^Q$ and the total I/O time $T_{I/O} = T_{I/O}^S + T_{I/O}^Q$, such that:

$$\max\left(T_C, T_{I/O}\right) \leq T_T \leq T_C + T_{I/O} \tag{1.1}$$

$T_{I/O}$ is the total time spent producing and consuming the tokens in the workflow model. It can be described as:

$$T_{I/O} = k \cdot \left(\frac{N}{wB} + \frac{M}{rB}\right) \tag{1.2}$$

According to Eq. (1.2), two classes of techniques aiming to mitigate the I/O overhead can be defined.

The first class of techniques aims to maximize wB and rB and is used in the so-called in situ workflow model to overcome the inherent bottleneck arising from file-based communications. Libraries such as HDF5 [10] and ADIOS [11] improve I/O by providing the application programmer with higher-level storage management APIs implemented on top of different I/O backends. For example, ADIOS enables the selection of I/O backends through an external XML-based configuration file.

The second class of I/O optimizations aims to minimize the numerator of Eq. (1.2). The so-called in-transit data processing model belongs to this second class [12]. For example, compressing and decompressing data on the fly during I/O can reduce N and M. Also, data format conversion can help when the data format consumed by Q differs from that produced by S. Note that in-transit processing can reduce $T_{I/O}$ and increase T_C.

Eventually, a different approach for optimizing I/O in data-intensive workflows involves enhancing the in situ workflow model with I/O coordination capability to exploit data streaming and overlap I/O and computation between consecutive steps. The CAPIO middleware [5] aims to do this *without modifying/patching the application code of the workflow modules* by intercepting POSIX system calls (SCs) and leveraging a declarative I/O coordination language.

The remainder of this document is organized as follows. Chapter 2 introduces the declarative I/O coordination language as the primary component of the CAPIO infrastructure. Chapter 3 describes the software architecture of the CAPIO middleware. Chapter 4 demonstrates the use of the CAPIO Middleware in two distinct use-cases, namely *1000 Genome* and *WRF-visualization* workflows. Finally, Chapter 5 draws some conclusions and potential future directions.

Chapter 2

I/O Coordination Language

Introduction

This chapter presents a new I/O coordination language that allows users to annotate file-based workflow data dependencies with synchronization semantics information related to files and directories. The coordination language, called CLIO (Coordination Language for I/O), aims to enable the transparent overlap of computation and I/O operations among distinct producer-consumer application modules.

The language used for expressing the I/O coordination language syntax and semantics is JSON (JavaScript Object Notation). JSON is not tied to any particular programming language or platform and is widely supported across various programming languages (Java, C++, Python, etc.). Although it is not the most commonly adopted language in the context of high-level coordination languages for expressing parallel computations, it provides automatic syntax validation features through the JSON schema, allowing us to focus more on the semantics aspects of the I/O coordination language. JSON is a well-recognized language in various computer science domains as it stems from simplicity, flexibility, and expressivity. Using JSON as the base syntax of the CAPIO coordination language presented in this document can make the learning curve less steep for the users.

2.1 Language syntax

In this section, we illustrate the syntax of the I/O coordination language. The syntax is expressed in JSON format. JSON syntax format is based on two primary structures: *objects* and *arrays*. An object is a collection of key/value pairs, where the key is a string identifier, usually a mnemonic name. An array is an ordered list of values.

We will refer to the JSON file written according to the I/O coordination language syntax as the *CAPIO configuration file*, as it is used by the CAPIO middleware to enforce files' synchronization semantics between consecutive workflow modules.

The CAPIO configuration file comprises five sections:

1. **Workflow Name**: identifies the application workflow composed by multiple application modules.
2. **IO_Graph** describes file data dependencies among application modules.
3. **Alias**: groups under a convenient name a set of files or directories.
4. **Permanent**: defines which files must be kept on the permanent storage at the end of the workflow execution.
5. **Exclude**: identifies the files and directories not handled by CAPIO.
6. **Home-node policy**: defines different file mapping policies to establish which CAPIO servers store which files.

We shall describe the language syntax related to each section in the following sections. Since the CAPIO language deals with I/O objects, i.e., files and directories, it supports wildcards (i.e., special characters that are used to specify unknown characters in a text), which can be used in file and directory names instead of enumerating all the files/directories that an application might produce or read (e.g., file*.dat). Currently, the language handles two wildcards: 1) '$\star$' that matches any sequence of characters of length ≥ 0; 2) '?' that matches a single character. Wildcards can be used in all values in which a file name or a directory name is expected.

Workflow name section

This section is identified by the keyword `name` (see Listing 2.1). The name is used as an identifier for the current application workflow. This is useful as it is possible to distinguish different application workflows running on the same machine.

Listing 2.1: The workflow name.

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

IO-Graph section

This section defines the dependencies between input and output streams between application modules comprising the workflow. It is identified by the keyword `IO_Graph`, and it requires an array of objects. Those objects specify the input and output streams for each application component. Each object defines the following items:

- `name`: The name of the application. This keyword is *mandatory*.
- `input_stream`: This keyword identifies the input files and directories the application module is expected to read. Its value is a vector of strings. This keyword is *optional*.
- `output_stream`: This keyword identifies a vector of files and directory names, i.e., the files and directories the application module is expected to produce. Its value is a vector of strings. This keyword is *optional*.
- `streaming`: This *optional* keyword identifies the files and directories with their streaming semantics, i.e., the "commit and firing rules" (the different semantics are described with details in Section 2.2).

  Its value is an array of objects. Each object may define the following attributes:

  - `name`: The filenames to which the rule applies. The value of this keyword is an array of filenames.
  - `dirname`: The directory names to which the rule applies. The value of this keyword is an array of directory names.
  - `committed`: This keyword defines the "commit rule" associated with the files or directories identified with the keywords `name` and `dirname`, respectively. Its value can be either `on_close:N` (where `N` is an integer ≥ 1), `on_termination`, or `on_file` if the "commit rule" applies to filenames (i.e., if the name keyword is `name`). Instead, if the "commit rule" applies to directory names, its value can be either `on_termination`, `on_file`, or `n_files:N` (where `N` is an integer ≥ 1). If the `committed` keyword is not specified, the default "commit rule" is `on_termination`. If the commit rule semantics is `on_file`, then the keyword `files_deps`, whose value is an array of filenames or directory names, defines the set of dependencies.
  - `mode`: This keyword defines the "firing rule" associated with the files and directories identified with the keys `name` and `dirname`, respectively. Its value can be either `update` or `no_update`. If the `mode` keyword is not specified, the default "firing rule" is `update`.

Listing 2.2 shows a valid `IO_Graph` section with two application modules, namely *writer* and *reader*, part of the workflow "my_workflow". The *writer* produces three files in output (file0.dat, file1.dat, and file2.dat) and one directory (dir). Each file is associated with distinct streaming semantics (i.e., distinct "commit and firing rules"). The *reader* takes in input all files produced by the *writer* module.

Listing 2.2: 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"]
     }
   ]
  ...
}
```

Aliases section

The aliases section, identified by the keyword `aliases`, is useful to reduce the verbosity of the enumeration of files an application can consume or produce. It is a vector of objects composed of the following items:

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

In Listing 2.3 we report an example of how to define aliases for disjoint sets of file names.

Listing 2.3: How to define aliases.

```
{
   "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"]
     }
   ]
```

```
  ...
}
```

Permanent section

This language section, identified by the keyword `permanent`, is used to specify those files that will be stored on the filesystem at the end of the workflow execution. Its value is an array of file names (whose values can also be aliases).Listing 2.4 shows a sketch of an example of how to define the `permanent` section. At the end of the execution of "my_workflow", the file output.dat and all files belonging to group0 will be stored in the file system.

Listing 2.4: How to define which files will be stored into the filesystem at the end of workflow execution.

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

Exclude section

This section, identified by the keyword `exclude`, is used to specify those files that will not be handled by CAPIO even if they will be created inside the **CAPIO_DIR** directory. Its value is an array of file names (whose values can also be aliases).

Listing 2.5: How to define which files will not be handled by CAPIO.

```
{
   "name" :"my_workflow",
   "exclude" :["file1.dat", "group0", "*.tmp", "*~" ]
  ...
}
```

Listing 2.5 shows an example of how to define the section `exclude`. The file1.dat, all files part of the group0, all temporary files ending with ".tmp", and all files ending with the character ' ' will be ignored by CAPIO.

Home-node policy section

This section allows the CAPIO user to selectively choose the CAPIO server node where all files (or a subset of them) and their associated metadata should be stored. The syntax enabling this option is shown in listing 2.6. The user can define different policies for different files. In the current version of the CAPIO language, the home node policy options are: `create`, `manual`, and `hashing`. These three keywords are optional, i.e., the user can avoid setting them explicitly in the CAPIO configuration file. In this latter case, the default policy adopted by the CAPIO middleware for the file-to-node mapping is the `create` one. Additionally, there cannot be filename overlap among the policies specified (i.e., the intersection of the set defined by the different home-node policies must be empty). For the `hashing` and the `create` policies, the value is an array of files. For the `manual` configuration, the syntax is more verbose and requires defining, for each file or set of files, the logical identifier of the application process whose associated CAPIO server (i.e., the server process running on the same node where the application process is running) will store the file. With this information, each CAPIO server may statically know the file-to-node mapping and thus retrieve the node where this process is executing at runtime. For a detailed explanation of the semantics of all the home node policies, please refer to section 2.3.

Listing 2.6: 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"]
   }
}
```

Wildcards potential ambiguity

The CAPIO language syntax leverages wildcards to provide the user with flexibility and reduce the burden of enumerating all files and directories. However, using wildcards in the language syntax can introduce unexpected behavior due to unintended matches or undefined behavior due to multiple matches associated with different semantics rules. To clarify the point, let us consider the example reported in Listing 2.7.

Listing 2.7: 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 example, there is an overlapping match for the files file1.dat and file2.dat. For those files, it is ambiguous if the commit semantics should be “on_close” or “on_termination”. Even if, in most cases, the ambiguity can be eliminated considering the most specific match for the rules that must be used (for example, “*.dat” is more specific than “file*” if we consider the context, i.e., the `output_stream` list of files specified by the user), in the current version of CAPIO, all ambiguities are not solved, and an undefined behavior exception is raised by the CAPIO runtime. In future releases, we will increase flexibility by relaxing such constraints and automatically disambiguating the syntax expression when possible.

It is worth mentioning that, other than carefully using wildcards, proper use of the `aliases` section can help the user write a non-ambiguous and clean configuration file. For example, in Listing 2.8, we defined two distinct aliases to specify two disjoint groups of files to which we can apply two distinct semantics rules.

Listing 2.8: 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"
           }
       ]
     },
     ...
   ]
  ...
}
```

2.2 Language Semantics

In this section, we introduce the streaming semantics that allows CAPIO to transform a batch, file-based workflow into an in-situ workflow (i.e., a workflow where all steps are executed concurrently). To establish the file synchronization semantics between consecutive workflow steps, we consider two *temporal* aspects:

1. Determining when there are no further updates to the file, referred to as the *commit rule*.

2. Identifying when a consumer can safely start reading a portion of data written in the file, known as the *firing rule*.

In the following, we describe the semantics of the *commit rule* and then the semantics of the *firing rule*.

2.2.1 Commit rule

In the context of the producer-consumer paradigm, a file can be conceptualized as a data stream. The commit rule defines when a given data stream terminates, i.e., all consumers have received the so-called *end-of-stream* message, which tells them there will not be more data in input for that specific stream. The commit rule defines two distinct file commit behaviors:

1. *on-termination*. This behavior is applied in the traditional file-based execution of workflow steps. Upon termination of a workflow module, all produced data files are committed to the file system, becoming ready for consumption by all subsequent consumer modules.

2. *on-close*. This behavior allows a consumer module to initiate reading a file as soon as I/O operations from all producer modules working on that specific file are completed. The completion of these I/O operations is signaled by the `close` SC executed by all producer modules.

Beyond these two primary file commit behaviors, a third commit rule can be defined. We can consider a file committed when another file has been committed. This proves beneficial when the number of `open` and `close` SC operations for a given file is not statically known. Instead, we are aware that the I/O operations can be considered concluded on a given file if another file has been committed. This additional commit behavior introduces a dependency among files in the commit rule, expanding opportunities to leverage temporal parallelism for I/O operations across different workflow steps.

Having introduced the commit rules, we shall define more formally the commit semantics for the producer(s) of a given `fileX`:

- *Commit on-Termination* (**CoT**): `fileX` is considered complete only when all producer modules associated with it have terminated.
- *Commit on-Close* (**CoC**): `fileX` is considered complete when all producer modules have definitively closed the file. With definitively closed, we intend that the file will not be opened to be modified/updated again by any producer modules.
- *Commit on-File* (**CoF**): The completion semantics of `fileX` are contingent upon the semantics of other files, e.g., `fileY`.

2.2.2 Firing rule

The firing rule defines when consumer modules of a file are permitted to consume stream data items (i.e., the file's records) produced by producer modules. The file's data elements consumption can occur immediately or be delayed based on specific events. Practically speaking, the commit rule dictates when a particular data stream concludes, indicating that all consumers have received the so-called *end-of-stream* message, i.e., the notification that there will be no further data records for that specific stream of data.
The CAPIO coordination language defines two distinct firing rules:

- *Fire on-commit*: whenever the commit rule is satisfied for a given file, the file is unequivocally ready to be consumed. In other words, the commit rule implies the firing rule for the entire file. This rule is called *Fire on-Commit* (**FoC**) and is identified in the CAPIO coordination language file with the keyword `mode` and value `update`.
- *Fire no update*: file records already written by producer modules can be consumed immediately. This behavior is referred to as *Firing no-Update* (**FnU**): the file content is ready to be read as soon as data is written into the file. It is identified in the CAPIO coordination language with the value `no_update` for the keyword `mode`.

There are two noteworthy scenarios. The first one arises when a consumer module attempts to open a file, which was specified in the CAPIO configuration file, that has not yet been created. In this case, the CAPIO runtime will halt the process executing the `open` SC, and the `open` will not return until the file is created. The second scenario is when a consumer module tries to read a portion of a file that has not been written yet. This behavior is nuanced as the `read` SC may return fewer data elements (or even 0) than what was originally requested. In this case, the consumer process initiating the `read` will be paused by CAPIO runtime until one of the following conditions is met:

- the requested data is fully produced, and the `read` SC returns the total number of bytes requested;
- all producer modules close the file (in the case of commit semantics being **CoC**) or terminate. Consequently, the `read` SC will return the current number of bytes read, if any, or 0 to indicate the end of the file (EOF).

2.2.3 Streaming semantics

We shall now introduce how to express the commit and firing semantics using the CAPIO coordination language, providing simple examples. For the following instances, we consider a workflow comprising two applications: an application called *writer* that produces two files and an application called *reader* that reads these two files. Algorithm 1 describes the *writer* application in the general case, where it produces N files. In each iteration of the main loop, it spends some time on computation and then writes the output of this computation to a different file for each iteration. The *reader* application, outlined in Algorithm 2, is very similar. In each iteration, it reads a file produced by the *writer* application and uses the data read to perform a certain computation for a specific 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

Typically, this straightforward workflow is executed in a classic batch execution. First, the *writer* application is launched to produce the input files for the *reader* application. Then, the *reader* application can start and consume the files, reading them from the filesystem. The CAPIO middleware enables running both applications concurrently without modifying the code of the two modules. To achieve this effectively, CAPIO needs to know the kind of data streaming semantics (as described in the previous sections) it should enforce on the produced and consumed files to guarantee the correct execution.

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

The more stringent semantic in terms of streaming capabilities is expressed by *Commit on-Termination* (CoT) with *Fire On-Commit* (FoC) firing rule. When this semantics is applied to a file, the *reader* can start reading that file only after the *writer* application has terminated. This proves helpful when a section of the file can be updated multiple times, and there is no knowledge about when the writer will cease adding data records. In this case, the correct behavior is to wait for the writer to terminate. When CAPIO captures a `read` SC on a file with such stringent semantics, it will return the data only upon the termination of the *writer* application.

With the CoT-FoC semantics, there is no ongoing streaming communication. However, running both applications concurrently can still be beneficial, especially in cases where the *reader* must perform substantial computation before reading the data from the `writer` application. In Listing 2.9, we report the configuration file expressing the CoT-FoC semantics for the simple example considered.

Listing 2.9: 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"]
    }
  ]
}
```

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

In the next example, the structure of the workflow and its data dependency remain unchanged; only the producer-consumer semantics changes. From a syntax standpoint, the only section that needs to be modified is the one related to the keyword `streaming`. Listing 2.10 shows the configuration file for the workflow with the semantics *Commit on-Termination* (**CoT**), *Firing no-Update* (**FnU**). With this semantics, the *reader* may commence reading the files `file1.dat` and `file2.dat` as soon as the *writer* produces data into these files. The *reader* will receive 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 useful when the user knows that once a section of the file is written, it will not be modified, but they do not know when the *writer* will stop writing data into the file. For the workflow under consideration, this semantics involves streaming only on the first file (i.e., `file1.dat`) because the writer and reader write/read files sequentially (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. The *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 2.10: 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"]
    }
  ]
}
```

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

The listing 2.11 shows the syntax to express the semantics *Committed on-Close* (*CoC*), *Firing On-Commit* (*FoC*). This semantics allows the *reader* to start reading a file after it is closed. The mode `update` (as seen before) does not allow the reader to read the data before the file is committed. In this case, the file is considered committed as soon as it is closed. This combination of semantics is useful when the *writer* updates a file multiple times, then stops writing/updating the records and closes the file. In this case, the CAPIO middleware makes the *reader* wait for the completion of a file, and then it starts to read it 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 2.11: 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"]
    }
  ]
}
```

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

The semantics *Committed on-Close* (**CoC**), *Firing no-Update* (**FnU**) (the syntax for defining this semantics is reported in Listing 2.12) allows the user to maximize the degree of streaming between the producer and the consumer workflow modules. In this case, the *reader* can start consuming data as soon as it is produced. The *reader* will stop reading file records as soon as it reaches the end-of-file and the *writer* has closed the file. If the *reader* reaches the end of the file and the *writer* has not yet closed it, the *reader* will wait until enough data is written to satisfy the `read` SC operation or the *writer* closes the file. If the *writer* closes the file, the *reader* receives the end-of-stream from the CAPIO middleware, and the `read` SC returns fewer data or EOF.

Listing 2.12: 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"]
    }
  ]
}
```

Commit on File, Fire update (CoF-FU)

The listing 2.13 shows the syntax for the semantics *Commit on-File* (*CoF*), *Firing no-update* (*FnU*). In this case, the `file2.dat` is considered committed when the `file1.dat` is committed. In our example workflow, the `file1.dat` is committed when closed; thus, `file2.dat` is committed with the semantics `on_close`, and the firing semantics is `update`. Since the *Firing update* is the default policy, the keyword `mode` can be omitted or can be set to force the *no_update* semantics.

Listing 2.13: 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"]
    }
  ]
}
```

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 the same (see Algorithm 1), but now the *reader* does not statically know the names of the files to read, but just the name of the directory containing them. Therefore, the *reader* iterates through the directory entries and reads all the files present in the directory. The pseudo-code of the reader is reported in Algorithm 3. Listing 2.14 shows one possible configuration file for this application scenario.

Algorithm 3: Reading all files in a directory.

```
Data: dir_path ; secs ≥ 0
F ← next_file(dir_path) ;                    /* Returns a file in the directory */
while F ≠ EOS do
    buffer ← read_file(F);
    compute(secs, buffer) F ← next_file(dir_path);
end
```

Streaming semantics can also be used for directories. For directories, it is possible to define the value `n_files:N` for the keyword "committed". This value indicates how many files will be produced within the directory. With this optional information, it is possible to read the entries of the directory while the *writer* is creating them. The CAPIO middleware, knowing the total number of files that will be produced in the directory, can determine when to return the EOS signal to the *reader* application. Of course, this information is useful only when the firing rule for the directory is `no_update` (as in the example provided). If not stated differently,

the default directory commits semantics is `on_termination` (the same default value applies to files). This means that the total number of files in the directory will be known when all producers writing in the directory terminate. It is correct to set the firing rule for a directory to `no_update` only if the new directory entries created (i.e., new files in the directory) will not be removed.

Listing 2.14: 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"]
      }
   ]
}
```

2.2.4 Data dependencies

In the previous examples, we always considered a simple pipeline composed of a producer and a consumer. In the IO-Graph section of the configuration file, it is possible to represent any DAG representing a workflow with its data dependencies. In this paragraph, we provide an example that uses more producers and consumers. Figure 2.1 represents a workflow composed by an Application S that produces some files that are needed by three applications: W, X, and Z. Each of these applications reads different files, and they write the result of their computations in the same file file-out.dat. Each application writes a different portion of this file that will be read by the last application named Y. Listing 2.17 shows how to write the IO-Graph with the CAPIO coordination language.

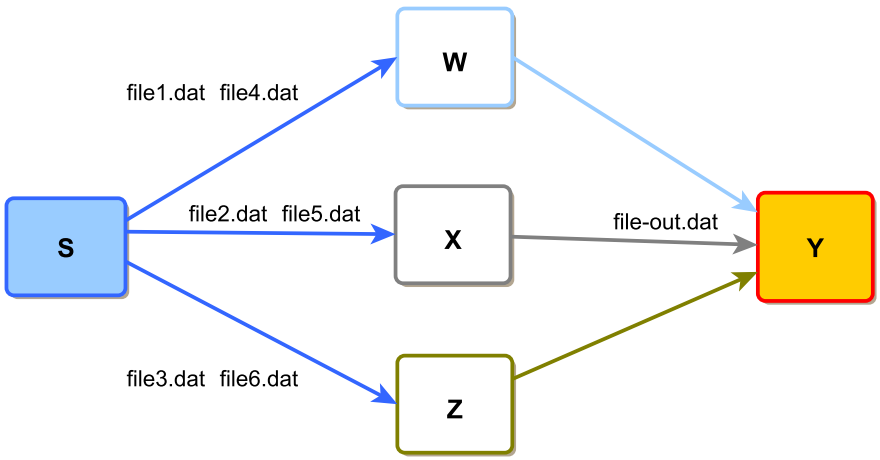

Figure 2.1: Example workflow showing files dependencies.

Listing 2.15: 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"]
      }
   ]
}
```

2.3 Hints for the run-time

In this section, we describe the features offered by the coordination languages to pass hints to the CAPIO's runtime system to optimize file data placement. We leverage the concept of *home-node*, borrowed from page-based software Distributed Shared-Memory implementations [13]. An *home-node* is a specific CAPIO node within the CAPIO ecosystem that serves as the reference node for storing information about data and metadata of a given set of files or directories.

As introduced in Section 2.1, the CAPIO language offers three distinct policies to set the *home-node*(s):

- *create*
- *manual*
- *hashing*

The user may define one or more of these policies for a disjoint set of files and directories. A given file or directory cannot have more than one *home-node* policy.

More details about the implementation of the *home-node* policies in the current version of the CAPIO middleware are described in Section 3.3.

Create home-node policy

The `create` policy is the default policy for the CAPIO coordination language. It means that this policy applies both when the `home-node-policy` language section is not present at all and when the `policy-name` object is explicitly set to `create`. If the `home-node-policy` language section is specified but not all files have the home-node policy defined, then for those files, the policy is `create`. The semantics behavior of the `create` policy is to assign as home-node the CAPIO server in which the file has been first created. As an example, if there are two CAPIO servers, namely C1 and C2, and one workflow module has two processes (P1 and P2) running on C1 and C2, respectively, then if C1 creates the file "file.dat", the data and metadata of that file will be stored in the C1 server. C1 is automatically elected as *home-node* of the "file.dat" file.

Manual home-node policy

When the `home-node-policy` keyword equals to `manual`, the users can explicitly set the reference node for each individual or group of files by choosing the *home-node* node where a specific application module is running. This is accomplished by specifying the workflow module's name used in the `IO_Graph` section. For instance, in the Listing 2.16, 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, whereas for the file "file2.dat" and "file4.dat" the *home-node* is the *writer*'s process with *logic id* 1. If the application is single-process, it is unnecessary to set its *logic id*, i.e., the module name is enough. The *logic id* is a unique identifier (integer type) that identifies a process of an application module. In the JSON file, it must be set according to this syntax: `module_name:id`. This `id` is the number passed through the environmental variable `CAPIO_APP_NAME` to the CAPIO runtime when launching the application. For example, let us assume that there are two CAPIO servers, namely C1 and C2, and the *writer* application module is executed using two processes. The process of the *writer* module running on the node C1 node will be launched with `CAPIO_APP_NAME="writer:0"` environmental variable set, whereas the *writer* module process running on the node C2 will be launched with the environmental variable `CAPIO_APP_NAME="writer:1"`.

Listing 2.16: 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 the `hashing`. The reference node for a given file is selected by hashing its pathname modulo the number of CAPIO nodes running for the given workflow. For example, the *logical id* of the *home-node* for the file "file.dat" for a workflow in which 4 CAPIO servers are used is obtained as `hash-function("$CAPIO_DIR/file.dat")%4`. The `hash-function` is fixed for all CAPIO servers and can be any good hashing function for strings as, for example, the `std::hash` method of modern C++. To use this policy, the `home_node_policy` must be set to `hashing`, and the keyword configuration must be omitted.

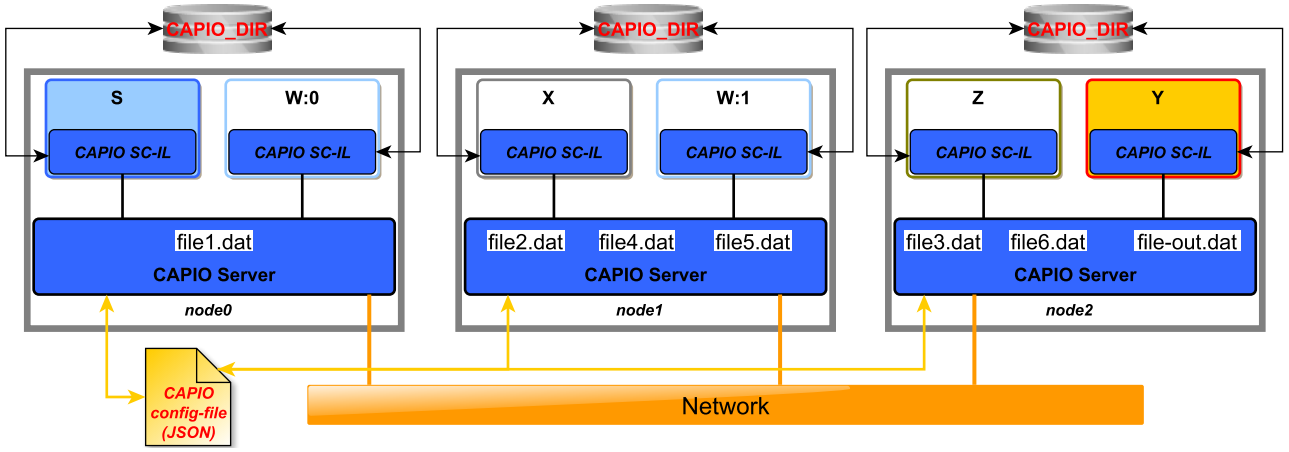

Figure 2.2: One possible deployment for the example workflow in Figure 2.1.

Usage example of the home-node policies

Here, we present an example where the correct use of the *home-node* policies may significantly improve the performance of a workflow. Figure 2.2 shows a possible deployment of the workflow represented in Figure 2.1. In this example workflow, the workflow modules are deployed on three nodes. The application module *S* runs on `node0`, the modules *X* and *V* on `node1`, and the modules *Z* and *Y* on `node2`. The workflow module *W* is the only multi-process application, and the process with logic id 0 runs on `node0` while the process with logic id 1 runs on `node1`. Listing 2.17 sketches the CAPIO configuration file with the *home-node* policies set to minimize the number of communications between producer-consumer nodes. The idea is to set as *home-node* the node where the process that needs to read it is running. In the configuration file presented, we use the `manual` configuration for the *home-nodes* as we can enforce a specific mapping of files to nodes. In more complex workflows with intricated file dependencies where files are written/read by multiple processes, the `hashing` policy can be a simpler solution for the user.

Listing 2.17: Files to nodes mapping using the *create* and `manual` policies for the workflow in Fig. 2.1.

```
{
 "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"
           }
     ]
  }
}
```

Home-node policies considerations

In HPC clusters, the users are usually not aware of which nodes the applications will be executed. By using logical names for the workflow application modules and logical IDs for the processes composing the single application module, the users may define data placement without knowing the actual configuration of nodes in which the workflow will be executed. Additionally, the same configuration file can be used in different deployments. This is possible because the *home-node* policy depends on the applications, not on the physical resources used.

It is worth noting that the `create` policy is essentially just syntactic sugar to facilitate the users. The same data placement of the `create` policy can be expressed with the `manual` policy with more effort by the user. Both the `create` and `hashing` policies are best suited when the user lacks deep knowledge about the workflow and its deployment. The `create` policy has the advantage of being very fast during the writing of a file by a single process because it is stored in the memory where the file is created. However, file creation may not be well-balanced among processes or application modules, potentially using a lot more memory in some nodes. Furthermore, if there are too many files stored on a single node, it may quickly become a bottleneck, especially in the case of a high number of reads from different nodes.

The `hashing` policy may help overcome these issues by balancing the data across all nodes where the workflow is running. However, the writing of a file can be slower because the data might be placed on a "distant" node. In the worst-case scenario, the hashing policy might create a situation in which a file is stored in neither the producer nor the consumer of the file, hence slowing down both read and write operations.

The `create` policy has been chosen as the default option as it is more accessible from a user standpoint to work with. This policy does not require the user to explicitly state the relationship between application modules and files produced or read. It is also simple to implement collecting the files-to-home-node mapping in a database shared by all CAPIO servers. As discussed in subsection 3.1, the current CAPIO middleware implements the *home-node* database as a set of files stored in the cluster filesystem.

Chapter 3

The CAPIO Middleware

Introduction

CAPIO (Cross-Application Programmable I/O)[1], is an open-source middleware capable of transparently injecting streaming executions of I/O operations into traditional or in situ workflows to enable temporal parallelism among workflow steps and reduce the contention on the shared file system through memory-to-memory data transfer. CAPIO seeks to optimize the I/O used to implement the communications among distinct application steps in scientific workflows where the file system allocates files used as communication buffers among steps. It promotes portability by supporting the POSIX standard and targeting all workflows whose I/O backend uses POSIX I/O system calls (SCs). Thanks to the new I/O-tailored coordination language[2] whose syntax and semantics were presented in Chapter 2, the CAPIO middleware shifts I/O coordination in workflow steps toward a declarative approach.

CAPIO users specify input/output file dependencies of steps and annotate them with synchronization semantics information to enable (or improve) pipeline execution between workflow steps. In doing that, *CAPIO avoids changing the existing codebase*. In many application scenarios, this is a very desirable feature because it is not always possible (e.g., because of legacy components in the workflow or closed-source applications) to rewrite or patch existing workflow steps to enable in situ orchestration by using explicit communication among steps. CAPIO transparently intercepts POSIX SCs of processes composing the different workflow steps and bends their execution according to the user-provided semantics information contained in a JSON-based configuration file, which is written by the users according to the I/O coordination language syntax and semantics described in Chapter 2. The aim is to transparently overlap the execution of I/O phases and computation-I/O phases in file-based workflow steps.

The CAPIO middleware is made of two logical tiers. The higher tier defines a *coordination model* allowing the user to express relaxed synchronization semantics between producer-consumer workflow steps via an *I/O coordination language* describing when a file is *fireable* (i.e., when its content can be accessed) and when a file is *committed* (i.e., when it is completed). The lower tier implements the CAPIO *runtime system*, which comprises a set of per-node user-space servers implementing the distributed data and metadata storage for the files and directories and the SCs intercept library. This library should be dynamically linked (through the `LD_PRELOAD` environment variable) to the application steps to intercept file system POSIX SCs executed on a pre-defined CAPIO directory (i.e., the CAPIO *local node mount point*).

In the rest of this chapter, we introduce the CAPIO software architecture and its runtime system (Sec. 3.1), how to deploy and run a CAPIO-based workflow in a SLURM-based cluster (Sec.3.2), and how to tune important internal parameters of the CAPIO runtime system (Sec.3.3).

[1] CAPIO: https://github.com/High-Performance-IO/capio
[2] CLIO: Coordination Language for I/O. https://github.com/High-Performance-IO/clio.

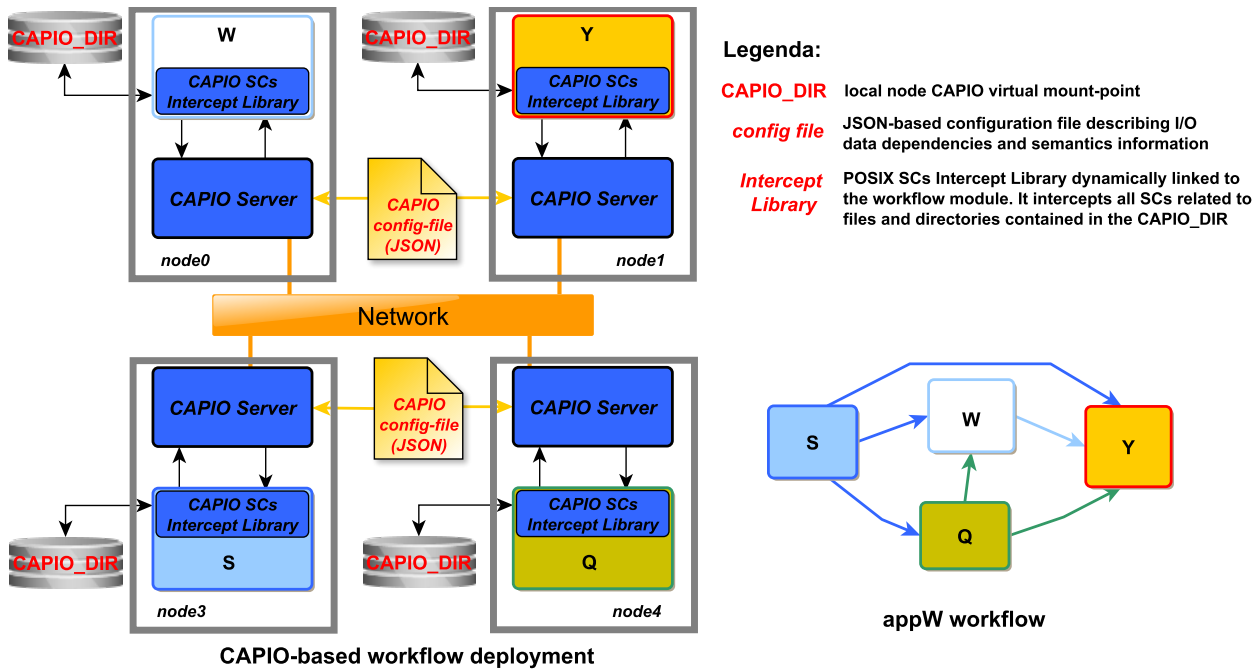

Figure 3.1: High-level view of the CAPIO middleware when used to coordinate I/O in a workflow application (the example workflow *appW* is composed of 4 steps *S*, *W*, *Q*, and *Y*) deployed on 4 cluster nodes (one workflow step for each cluster node).

3.1 Software Architecture

CAPIO's runtime comprises a set of per-node user-space servers (simply *CAPIO server*) implementing a distributed data storage for a single application workflow. Each CAPIO Server uses the information contained in the CAPIO configuration file provided as an input argument to enforce file exchange coordination and data streaming according to the producer-consumer semantics described in Chapter 2.

Figure 3.1 sketches a schematic view of a possible deployment of the CAPIO Servers in a 4-node cluster (*node*0-4) for the application workflow example *appW*. In each node where the workflow steps are deployed, a CAPIO server handles all files and directories referenced by the step in the local node CAPIO virtual mount-point. All SCs targeting files and directories outside the CAPIO local virtual mount point are not directly handled by the CAPIO server and they are forwarded to the kernel.

The internals of the CAPIO server are written in C++, whereas server-to-server communications are currently implemented using MPI[3] (specifically, the OpenMPI [14] implementation of the standard). The user specifies a CAPIO local node virtual mount point for each node through 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`[4] is a shared library dynamically linked to the steps of the application workflow, through the `LD_PRELOAD` dynamic linker environment variable, to capture the I/O SCs executed on files and directories inside the local node virtual mount point. 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 3.2 shows the internal primary components of the CAPIO server running on a single node. In the following, we briefly report the main functionalities of each component.

Intra-node Shared-Memory Layer. The data exchange between the CAPIO SC-IL component running within the application process and the CAPIO server is coordinated in this layer by leveraging the POSIX Shared-Memory standard interface. For each workflow step running in the same node of the CAPIO

[3]Support for other protocols is already present, but as of now, no other libraries have been implemented into CAPIO.
[4]syscall_intercept: https://github.com/pmem/syscall_intercept

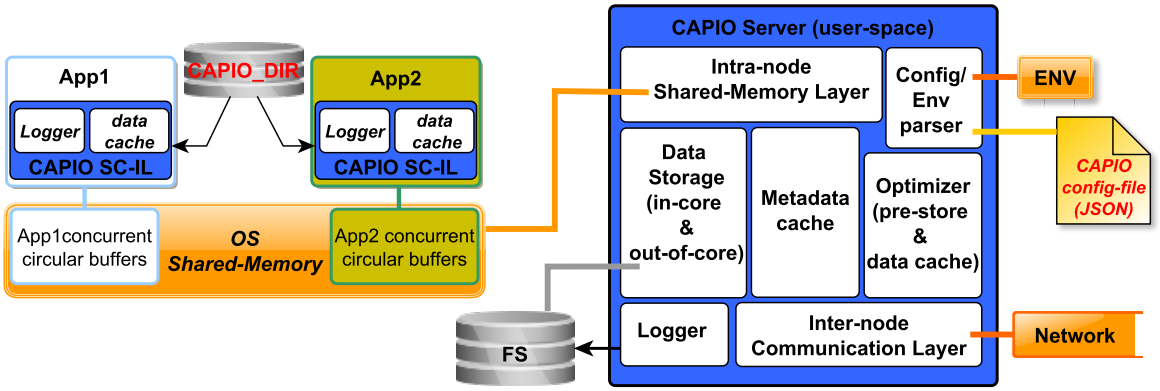

Figure 3.2: The internal software components of the CAPIO Middleware running on a single node. It comprises two distinct parts: the CAPIO server and the CAPIO System Calls Intercept Library (CAPIO SC-IL).

server, a set of shared-memory segments are allocated to store buffers and semaphores used for the synchronization. Such segments will be destroyed once the application step exits.

Config/Env parser. The CAPIO Server leverages a JSON-based config file storing information related to file coordination and streaming semantics. It also uses a set of process environmental variables to obtain information related to the performance tuning of internal components and to know the name of the CAPIO_DIR. Such pieces of information are extracted by the Config and Env parser from the config file and from the process's env and then adequately organized to be used by other CAPIO components.

Data Storage. This component stores the contents of files for which the CAPIO server is the home node, that is, the server that stores the master copy of the file. Portions of a given file may also be stored in other CAPIO servers for performance reasons in order to reduce network traffic, but, however, in the current implementation of the CAPIO middleware, a single master node is elected as the one that stores the most recent metadata and data information of a given file or directory. The other CAPIO servers always refer to the home node to know if a given file has updated data and metadata. By default, each file is stored in the main memory of the CAPIO Server. Upon request, the user may select to store the file in the filesystem through the `permanent` keyword in the JSON config file. In this way, the file will be preserved after the termination of the CAPIO server. This feature is currently implemented by leveraging the *mmap* system call.

Metadata cache. This component implements a cache of metadata information about files and directories whose home node is not the current CAPIO server. A simple invalidation protocol among CAPIO servers ensures the metadata cache stores only up-to-date information.

Optimizer. The Optimizer is a CAPIO component that implements file data caching in non-home node CAPIO servers and enables the optimization of sending more data than requested to reduce the number of request-reply message exchanges among CAPIO servers.

Logger. The CAPIO Logger is divided into 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 separately and are not usually compiled into CAPIO unless explicitly told to do so, as logfiles can usually be made of several gigabytes of data (and might leak internal information about the data that CAPIO treats). The logger component is also capable of logging on different, increasing levels (from 0 = do not log, to a number n, where n is the level of the reentrant function call. If the log level is < 0, then everything is logged).

Inter-node Communication Layer. This component is responsible for managing communications between CAPIO servers for exchanging data and metadata information. The software layer is designed to be independent of the actual data transport method used for the communications. The abstract interface currently uses the Message Passing Interface (MPI) library. The future releases of the CAPIO middleware

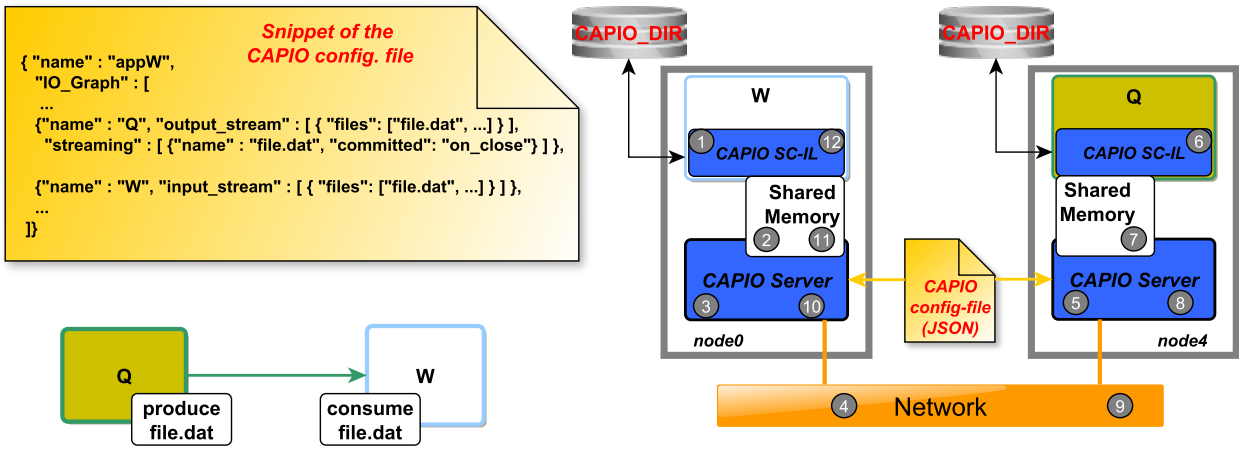

Figure 3.3: Schema of a simple producer-consumer workflow deployed on two nodes. The picture reports the steps executed in the request-reply protocol handling `read` and `write` operations on the "file.dat".

will allow users to select from multiple transport back-ends, as the infrastructure required to support other libraries has already been implemented and tested.

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

CAPIO implementation supports both multi-process and multi-threaded applications. By default, the file data (and metadata) are 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 in the same producer node, the communications go through the shared memory buffer mediated by the local CAPIO server. Instead, if the consumer steps are deployed in different cluster nodes, the requested data is transferred by 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.

Concerning the files' metadata information, not all metadata are 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, thus allowing CAPIO to deal with *sparse files*, a technique often used to write different partitions of a single file in parallel.

Request-Reply steps between CAPIO servers

Here, we present a simple producer-consumer workflow example that involves two application steps, namely *Q* (the producer of the *file.dat* file) and *W* (the consumer of that file) running on two different computing nodes. The aim is to illustrate the producer-consumer protocol between the two CAPIO servers by describing the steps taken to handle two possible cases: 1) *W* reads the *file.dat* before *Q* has produced it; 2) *W* reads the file when *Q* has already started to produce its content. The CAPIO server where *Q* is deployed is the *home_node* of the file. The snippet of the JSON configuration file describing the example workflow is sketched in Figure 3.3.

For the first scenario, we suppose the step *W* opens the file *file.dat* for reading, but the data for this file has yet to be produced by the step *Q*. The CAPIO SC-IL intercepts the `read` system call (step 1) and passes the request to the CAPIO server through a shared-memory segment containing the requests' buffer (step 2). The CAPIO server checks if the requested data is already present in the local data cache. If it is not present, the CAPIO server retrieves the CAPIO home node server (step 3) and sends a data request containing the amount of data to read to the selected server (step 4). After that, the CAPIO server continues serving any other requests that may arrive from local workflow steps or other remote CAPIO servers. The requesting process in the *W* step that issued the `read` SC request waits for the operation to be completed. The home node CAPIO server

receives the read request and checks if the data is available in the local storage (step 5). In our example, the producer *Q* has yet to produce the data; hence, the home node CAPIO server cannot immediately reply to the request. When the step *Q* starts writing the data into the *file.dat* file, the CAPIO SC-IL intercepts all `write` system calls (step 6), stores the data into its local cache, and then passes the data to the CAPIO Server through a shared-memory buffer once the cache buffer is full or flushed when the file is closed (step 7). The CAPIO Server stores the data in the local storage (step 8), updates the metadata information, and simultaneously serves all the pending remote requests waiting for a reply (step 9). The amount of data sent as a reply message can be greater than the initially requested size if the CAPIO Server's Optimizer component has been configured to perform pre-store operations aggressively. Once a reply message is received, the CAPIO Server stores the data in the local cache (step 10) and puts the data requested by the initial `read` SC into the replies' shared-memory buffer associated with the request (step 11). Finally, the CAPIO-IL completes the `read` request by copying the data into the SC buffer (step 12).

The second scenario (i.e., *W* starts reading the file content after that *Q* has already started to produce its content) is similar to the one just described but for a few aspects. When the home node CAPIO Server receives the `read` request from the CAPIO Server running on the same node of the *W* step, it immediately replies to the request. It sends more data than the amount requested if it is available in the data storage up to a given threshold that can be defined by the user through the environment variable CAPIO_PREFETCH_DATA_SIZE (see section 3.3). We called this optimization *pre-store*. The aim is to optimize the data transmission and potentially reduce the number of request-reply message exchanges.

CAPIO SC-IL

The CAPIO System Calls Intercept Library (CAPIO SC-IL) is a crucial part of the CAPIO Middleware. Its primary function is intercepting system calls related to file and directory management. However, to guarantee proper application behavior, it must intercept all system calls of the application workflow step with which it is dynamically linked. Then, only a subset of all POSIX SCs will be forwarded to the CAPIO Server, specifically only those related to files and directories contained in the CAPIO_DIR. Therefore, it is essential to keep its overhead as low as possible.

To understand how much overhead the CAPIO SC-IL introduces, we considered the *lmbench* benchmarks [15], a series of micro-benchmarks measuring OS and HW system metrics. In particular, we considered the `lat_syscall` benchmark measuring the latency of some SCs largely used when dealing with files and directories.

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 3.1: Execution time (in microseconds) of the `lat_syscall` test from *lmbech* benchmark suite considering some relevant SCs. The tests were executed on an Intel(R) CascadeLake 8260 CPU running at 2.4 GHz (Centos 8.3.2011, Linux 4.18.0-240 version).

We tested two cases: 1) no LD_PRELOAD, which means no SC intercept; and 2) the CAPIO intercept library by setting `LD_PRELOAD=libcapio_posix.so`. In the latter case, we set as CAPIO_DIR the /dev/null directory to measure only the extra cost of using the CAPIO SC-IL library. The results obtained by

averaging multiple repetitions of the tests are reported in Table 3.1. Overall, the extra overhead introduced by the CAPIO intercept library is relatively small.

Home node implementation

The home node DB is currently implemented with POSIX files in the distributed filesystem and is updated/accessed by the CAPIO servers using POSIX *file locking* to avoid race conditions, and it is cached in the main memory of each CAPIO server for performance reasons since file-to-home node mappings do not change. However, in the future, we plan to explore alternative solutions such as in-memory distributed DBs. When a process wants to read a file, it must know which node serves as the home node for that file. If the home node is known statically, then it is straightforward. However, if the home node policy is dynamic, 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 because the home node is determined only at runtime. In contrast, the hashing policy is static because retrieving the node only requires applying the hashing function to the path of the file.

At the moment, a file is stored in the memory of the home node, but in the future, we plan to provide users with the option to distribute the content of a file across multiple nodes. This technique is called 'Data Stripping' and is commonly used in distributed file systems to mitigate the bottleneck of a single node. Distributing the content of a file across multiple locations is beneficial for improving the performance of multiple parallel read requests to different parts of a file.

3.2 Deployment

To deploy a workflow with CAPIO, a CAPIO server must be launched on each node where the workflow will be executed. Additionally, for each application step in the workflow, the CAPIO SC-IL library must be linked by setting the environment variable LD_PRELOAD to the path of the CAPIO shared library (libcapio_posix.so). In listing 3.1 is presented an example of a SLURM script for a simple workflow using the CAPIO middleware. The workflow comprises two application components: a *writer* and a *reader*. It is worth remarking that with CAPIO, the *writer* and *reader* components of the workflow are executed concurrently. In this simple scenario, the workflow runs on two cluster nodes, with the *writer* launched on the first node and the *reader* on the second node. The CAPIO server is present on both nodes. In the future, to facilitate the CAPIO deployment, we planned to integrate the CAPIO middleware with some workflow management systems such as StreamFlow [16] and Dagon* [17] to automate the deployment of CAPIO.

Listing 3.1: 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=$!

# RUN A READER PROCESS IN THE NODE 1
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

wait $WRITER_PID
kill $SERVER_PID
```

3.3 Configuration options

In the following, we report a set of environmental variables to be used to tune both the performance of the CAPIO middleware and the logging configuration and verbosity. We distinguish between "server-side", "client-side", and "both" to refer to who uses the variable, the CAPIO server, the CAPIO SC-IL, or both.

- **CAPIO_FILE_INIT_SIZE** (server-side): The default space in RAM reserved for files stored by the CAPIO server is 1MB. It can be beneficial to set this variable with a larger value for efficiency because when a file exceeds the reserved space, memory must be reallocated to a larger size.
- **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. It means that only the requested data is retrieved. Setting this variable to a larger value is helpful 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 of 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 environment 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 tune the verbosity level of the CAPIO logging.

Chapter 4

Use-Cases

Introduction

In this chapter, we describe two workflows used to validate the expressivity of the I/O coordination language and the performance of the CAPIO runtime system.

The first workflow, called 1000 Genomes [18], is a relatively complex DAG-based bioinformatics workflow computing human genome mutation overlap. The original 1000 Genomes workflow was implemented using Bash and Python scripts. It was later re-implemented using C++ and the Boost library, which reduced the total execution time by more than $3\times$. We selected this workflow because of the non-trivial interactions (i.e., files and directories dependencies) between the different components implementing the entire 1000 Genomes workflow. We tested CAPIO language synchronization semantics using the *commit-on-Close Firing-no-Update* (CoC-FnU) to transparently enable pipeline parallelism among the workflow application modules. We performed our experiments deploying the CAPIO middleware on the GALILEO100[1] Tier-1 supercomputer hosted by CINECA supercomputing center[2]. Each computing node we used for the experiments has 2 Intel CascadeLake 8260 CPUs, with 24 cores, each running at 2.4 GHz and equipped with 384GB RAM. The distribution is Centos 8.3.2011 with the Linux kernel version 4.18.0-240. The storage system is based on *Lustre* open source parallel files system [19]. Each cluster node is connected through a switched 100 Gb/s Infiniband interconnect. The *scratch* directory (mounted on the Lustre file system under `/g100_scratch`) is connected to the storage with a 100Gb/s Infiniband interconnect. In our experiments, when using the file system, we always used the *scratch* directory to store the files and directories.

The second workflow tested is the WRF-visualization workflow, which is a small part of a more complex Environmental Applications (ENV) ADMIRE use-case [20]. We considered this workflow because the WRF part is a parallel MPI-based application written in Fortran, which is challenging to modify to implement a tightly coupled interaction with the visualization module of the workflow that is implemented as a Python script. We performed our experiments deploying the CAPIO middleware on the HPC4AI cluster (a.k.a. UNITO cluster)[3] comprising 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.

4.1 1000 Genome Workflow

The workflow called *1000 Genome* refers to a DAG-based bioinformatics workflow computing human genome mutation overlaps.

It comprises file application components (also called modules): *individuals*, *merge individuals*, *sifting*, *mutation overlap*, and *frequency*. The *individuals* module can be replicated in multiple independent instances. Each instance analyzes a partition of the input file and generates a directory containing $2,504$ temporary small

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

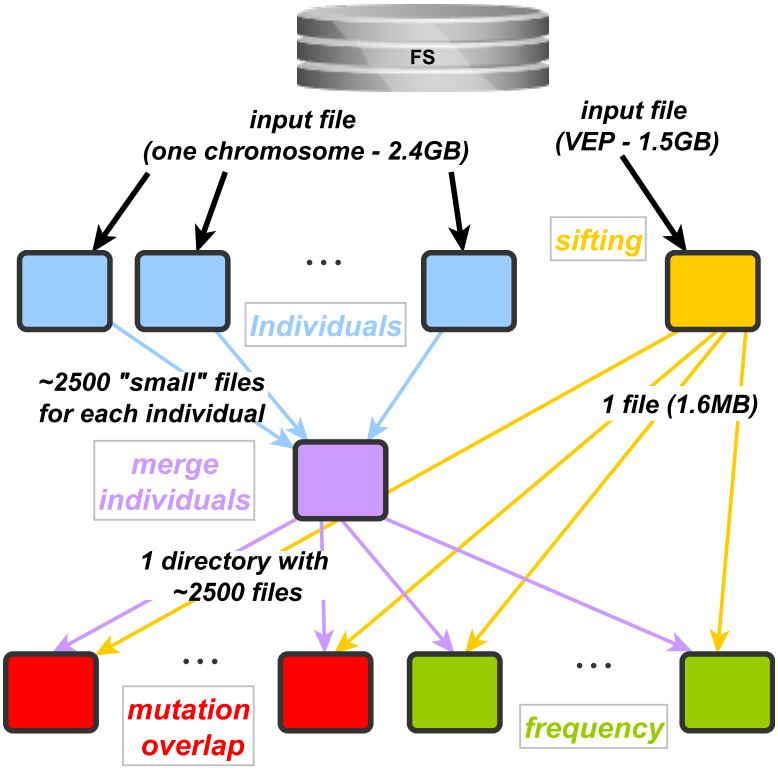

Figure 4.1: 1000 Genome workflow structure.

files (whose sizes depend on the number of individuals, – in the range $1 - 15$KB with 16 instances). The *sifting* module runs in parallel with all *individuals* components. The *individuals_merge* module reads all the files in the directories produced by all *individuals* and combines them into one single directory with $2,504$ files, where each file is a merge of all files with the same name produced by the *individuals*. The last two workflow components, *mutation_overlap* and *frequency*, are independent. They read the input dataset and the data produced by previous steps.

Figure 4.1 shows the five components of the workflow and their non-trivial dependencies.

Listing 4.1: 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"],
           "committed" :"n_files:2504",
           "mode" :"no_update"
        },
        {
           "name" :["chr1n/*"],
           "committed" :"on_close",
           "mode" :"no_update"
        }
     ]
  },

  {
     "name" :"sifting",
     "input_stream" :["data/20130502/sifting/ALL.chr1.phase3_shapeit2_mvncall_integrated_v5
        (cont.).20130502.sites.annotation.vcf"],
     "output_stream" :["sifted.SIFT.chr1.txt"],
     "streaming" :[
        {
           "name" :["sifted.SIFT.chr1.txt"],
           "committed" :"on_close",
           "mode" :"no_update"
        }
     ]
  },

  {
     "name" :"mutations_overlap",
     "input_stream" :["sifted.SIFT.chr1.txt", "chr1n", "data/populations/ALL", "data/20130502/
        (cont.)columns.txt"],
     "output_stream" :["chr1-ALL.tar.gz"]
  },

  {
     "name" :"frequency",
     "input_stream" :["sifted.SIFT.chr1.txt", "chr1n", "data/populations/ALL", "data/20130502/
        (cont.)columns.txt"],
     "output_stream" :["chr1-ALL-freq.tar.gz"]
  }
]
}
```

We tested CAPIO with *CoC-FnU* synchronization semantics (see Section 2.2) to exploit temporal parallelism among the workflow components. The CAPIO configuration file is reported in Listing 4.1. For example, *mutation_overlap* and *frequency* start reading the input dataset while *individuals_merge* and *sifting* are still running. Additionally, they start reading files produced by *merge_individuals* as soon as a file is available without waiting for all $2,504$ files to be produced. Such overlap among distinct workflow modules is unattainable with the traditional file-based workflow execution model. The 1000 Genomes workflow was originally implemented using Bash and Python scripts. We tested CAPIO with the fastest version (i.e., the C++-based one) using one single chromosome simulation. Different simulations on different chromosomes generate independent workflows that can be executed in parallel on distinct cluster nodes.

We tested the workflow execution with CAPIO on the GALILEO100 cluster using 20 cluster nodes. Specifically, 16 *individuals* instances (one for each node), and 1 instance of *mutation_overlap* and *frequency*. With this configuration, the files produced are $16 \times 2,504 = 40,064$.

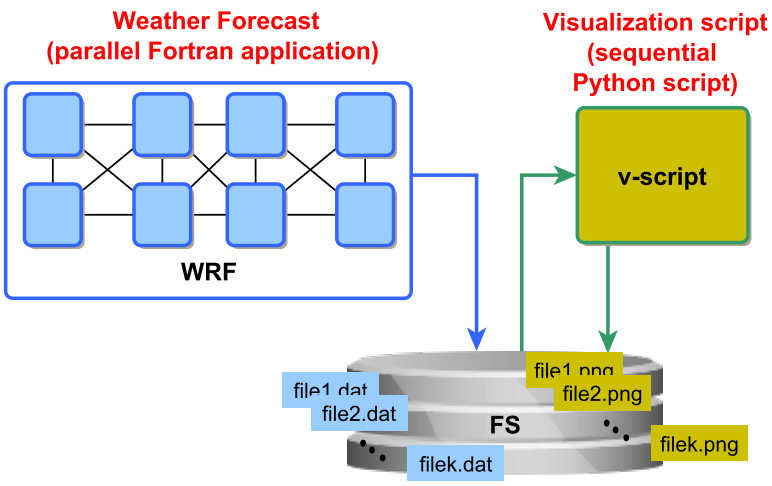

Figure 4.2: WRF-visualization workflow schema. This is a part of the ADMIRE ENV application workflow [20].

Overall, CAPIO reduces the execution time over the filesystem-based execution by about 39% (from about 393s to about 179s). The interested reader may find a more detailed discussion about the performance results of the 1000 Genome workflow in [5].

4.2 WRF-visualization Workflow

The WRF-visualization workflow is part of the Environmental Applications (ENV) ADMIRE use-case [20] [21]. The entire workflow comprises several steps. We focused on the post-processing part and, specifically, the production of PNG images for visualization purposes, executed after the WRF (Weather Research and Forecasting model [22] component. In fact, these two components currently work in batch, i.e., first, the WRF module produces all the files resulting from the simulation, then the visualization script reads all the files produced and creates the PNG output files. By using the CAPIO middleware, we aim to run these two applications together, exploiting pipeline parallelism between the two components, without modifying the existing code.

The WRF-visualization workflow is sketched in Figure 4.2. The CAPIO configuration file for the workflow is shown in Listing 4.2. The WRF application produces the output binary files of the simulation in the working directory ('.') one after the other. The post-processing application reads all files ('./*') to produce PNG images depicting the status of the simulation at a certain point in time.

The number of output binary files produced by the WRF application depends on an input parameter (the number of hours simulated). Therefore, it is not possible to set the `n_files` commit value for the working directory in the configuration file. This is why the commit rule for the working directory is `on_termination`. A file is never deleted after it is created in the current directory; therefore, the post-processing application can read the directory entries of the working directory concurrently with the WRF application. To achieve this behavior, the `mode` keyword (i.e., defining the *firing rule*) is set to `no_update`. The streaming semantics of the files produced by WRF is CoC (*committed on close*), whereas the keyword `mode` is set to `update` because a file can be updated multiple times by the WRF module (the files will not be re-opened once closed). With the simple configuration file shown in Listing 4.2, we can achieve the desired streaming parallelism behavior.

Listing 4.2: CAPIO configuration file for the WRF-visualization workflow using the CoC-FU semantics.

```
{
"name" :"WRF_WORKFLOW",
"IO_Graph" :
[
   {
      "name" :"WRF",
      "output_stream" :[".", "./*"],
```

```
      "streaming" :[
         {
            "dirname" :["."],
            "committed" :"on_termination",
            "mode" :"no_update"
         },
         {
            "name" :["./*"],
            "committed" :"on_close"
         }
      ]
   },
   {
      "name" :"visualization",
      "input_stream" :["./*"]
   }

]
}
```

We tested the WRF-visualization workflow execution with CAPIO using 96 MPI processes on 4 cluster nodes of the UNITO cluster. The test considered 2-domains and 25-hours simulation. Overall, CAPIO reduced the execution time over the filesystem-based execution by about 17% (from 2538s to 2098s). The first output file was produced after 761s in the filesystem-based workflow execution and after about 80s in the CAPIO-based workflow execution. In the ADMIRE deliverable "D7.4 - Validation of project outcomes through applications", due by the end of the project, we will provide more extensive performance results.

Chapter 5

Conclusion

HPC workloads are moving fast from monolithic applications to workflows with a mix of co-engineered and legacy components communicating through the portable file-based interface and using the file system as a communication media.

In this document, we presented a novel I/O coordination language developed as one of the components of the ADMIRE software stack. We described the syntax and semantics of the I/O coordination language, as well as its current implementation within the CAPIO middleware. CAPIO can transparently inject I/O streaming capabilities into file-based workflows, leveraging the directives and hints the user provides through a JSON-based configuration file describing the I/O synchronization semantics of file dependencies among the workflow's application modules according to the I/O coordination language rules. By transparently intercepting POSIX-based I/O system calls, CAPIO enables temporal overlap of distinct workflow application modules. We described the synchronization semantics currently supported by the I/O coordination language and its associated CAPIO runtime.

We plan to finalize the integration of the CAPIO middleware with the WRF-visualization workflow in the ENV application use case where the initial tests have already demonstrated the benefits of the temporal overlap of distinct application components without the need to rewrite entirely or part of their code. A thorough performance analysis and more comprehensive tests will be provided as part of the project validation outcomes.

Finally, the I/O coordination language and the CAPIO middleware were both released as open-source and freely available online.

Appendix A

Acronyms

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

- **HPC**: High Performance Computing.
- **WP**: Workpackage.
- **IC**: Intelligent Controller.
- **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.

Appendix B

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. The most up-to-date version can be found at the following web link:

https://github.com/High-Performance-IO/clio.

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

     "list_of_files": {
        "type": "array",
        "items": {
           "type": "string"
        },
        "minItems": 1
     },

     "streaming": {
        "description": "Streaming semantics of files produced by the application",
        "type" :"array",
        "items": {
           "type": "object",
           "oneOf" :[
              {
                 "properties": {
                    "name": {"$ref": "#/$defs/list_of_files" },
                    "committed" :{
                       "description": "Commit rule",
                       "type" :"string",
                       "pattern": "^(on_termination)$|^(on_close(:([1-9]+))?)$|on_file"
                    }
                 },
                 "required" :["name", "committed"]
              },
              {
                 "properties": {
                    "dirname": {"$ref": "#/$defs/list_of_files" },
                    "committed" :{
                       "description": "Commit rule",
                       "type" :"string",
                       "pattern": "^(on_termination)$|^(n_files(:([1-9]+)([0-9]*))?)$|on_file"
                    }
                 },
                 "required" :["dirname", "committed"]
              }
           ],
           "properties": {
              "mode" :{
                 "description": "Firing rule",
                 "enum": ["update", "no_update"]
              }
           },
           "if": {
              "type": "object",
              "properties": {
                 "committed": {"const": "on_file" }
              }
           },
           "then": {
              "properties": {
                 "files_deps": {
                    "$ref": "#/$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
}
```

Bibliography

[1] ADMIRE project deliverable (M24). D2.3 - Short-lived ad-hoc storage system, March 2023.

[2] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. Gekkofs-a temporary distributed file system for hpc applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 319–324. IEEE, 2018.

[3] Felix Garcia-Carballeira, Alejandro Calderón, Jesus Carretero, Javier Fernández, and Jose Maria Perez Menor. The design of the expand parallel file system. *International Journal of High Performance Computing Applications*, 17:21–37, 03 2003.

[4] Francisco Rodrigo Duro, Javier Garcia Blas, Florin Isaila, Jesus Carretero, JM Wozniak, and Rob Ross. Exploiting data locality in swift/t workflows using hercules. In *Proc. NESUS Workshop*, 2014.

[5] 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, December 2023. IEEE.

[6] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13, 2015.

[7] 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. *Communications of the ACM*, 65(6), 2022.

[8] 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. *FGCS journal*, 46, 2015.

[9] 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. *FGCS journal*, 128, 2022.

[10] Mike Folk, Albert Cheng, and Kim Yates. Hdf5: A file format and i/o library for high performance computing applications. In *SC '99: Proc. of the ACM/IEEE Conf. on Supercomputing*, volume 99, 1999.

[11] 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. *SoftwareX*, 12, 2020.

[12] 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 ACM/IEEE Conf. on Supercomputing*, 2012.

[13] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. *SIGOPS Oper. Syst. Rev.*, 25(5):152–164, sep 1991.

[14] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, 2004.

[15] 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.

[16] Iacopo Colonnelli, Barbara Cantalupo, Ivan Merelli, and Marco Aldinucci. StreamFlow: cross-breeding cloud with HPC. *IEEE Transactions on Emerging Topics in Computing*, 9(4):1723–1737, 2021.

[17] Dante Domizzi Sánchez-Gallegos, Diana Di Luccio, Sokol Kosta, J.L. Gonzalez-Compean, and Raffaele Montella. An efficient pattern-based approach for workflow supporting large-scale science: The dagonstar experience. *Future Generation Computer Systems*, 122:187–203, 2021.

[18] 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. *FGCS journal*, 95, 2019.

[19] Peter J Braam and Philip Schwan. Lustre: The intergalactic file system. In *Ottawa Linux Symp.*, volume 8, 2002.

[20] ADMIRE project deliverable (M19). D7.2 - Application Co-design Preliminary Report, December 2022.

[21] ADMIRE project deliverable (M35). D7.3 - Application Co-design Report. Due at M35.

[22] William C Skamarock, Joseph B Klemp, Jimy Dudhia, David O Gill, Zhiquan Liu, Judith Berner, Wei Wang, Jordan G Powers, Michael G Duda, Dale M Barker, et al. A description of the advanced research wrf model version 4. *National Center for Atmospheric Research: Boulder, CO, USA*, 145:145, 2019.