CAPIO: a Middleware for Transparent I/O Streaming in Data-Intensive Workflows

CAPIO: a Middleware for Transparent I/O Streaming in Data-Intensive Workflows

1st Alberto Riccardo Martinelli
University of Turin, Italy
albertoriccardo.martinelli@unito.it

2nd Massimo Torquati
University of Pisa, Italy
massimo.torquati@unipi.it

3rd Marco Aldinucci
University of Turin, Italy
marco.aldinucci@unito.it

4th Iacopo Colonnelli
University of Turin, Italy
iacopo.colonnelli@unito.it

5th Barbara Cantalupo
University of Turin, Italy
barbara.cantalupo@unito.it

***Abstract*—With the increasing amount of digital data available for analysis and simulation, the class of I/O-intensive HPC workflows is fated to quickly expand, further exacerbating the performance gap between computing, memory, and storage technologies. This paper introduces CAPIO (Cross-Application Programmable I/O), a middleware capable of injecting I/O streaming capabilities into file-based workflows, improving the computation-I/O overlap *without the need to change the application code*. The contribution is twofold: 1) at design time, a new I/O coordination language allows users to annotate workflow data dependencies with synchronization semantics; 2) at run time, a user-space middleware automatically and transparently to the user turns a workflow batch execution into a streaming execution according to the semantics expressed in the configuration file. CAPIO has been tested on synthetic benchmarks simulating typical workflow I/O patterns and two real-world workflows. Experiments show that CAPIO reduces the execution time by 10% to 66% for data-intensive workflows that use the file system as a communication medium.**

***Index Terms*—Workflow, In situ model, I/O coordination**

I. Introduction

A workflow describes a sequence of application steps and their control/data dependencies. Traditionally, in the HPC context, data dependencies are usually streamlined by storing data files in the distributed storage system by producer steps and afterward read out by consumer steps. However, with the growing gap between processing and storage system speeds coupled with the ever-increasing amount of data produced in scientific applications, sharing files between workflow steps through the central file system is costly [1], [2].

Burst buffers [3] and user-space ad-hoc file systems [4] have been proposed as solutions to increase the available I/O bandwidth and reduce the contention on the shared file system by leveraging fast local storage. However, workflow steps need to be executed orderly according to the data dependency graph, and it could be challenging to exploit pipeline parallelism among them. Nevertheless, in many scientific simulation analysis workflows, the core simulation steps producing data might be co-executed with the data analysis steps for almost real-time evaluation of results [5]. Unless the analysis steps were developed to support such coupled execution with the simulation steps through explicit synchronizations, the analysis steps need to wait for all data results to be produced by simulation phases into the storage before starting their analysis.

In situ workflows [2], [6], [7] were proposed to mitigate or avoid the cost of relying on the distributed file system as communication media and enable temporal parallelism between steps. Multiple steps are executed concurrently; data dependencies are accomplished by sidestepping the file system through *explicit coordination mechanisms* among workflow steps (e.g., via message-passing or coordinated files access through suitable APIs) [8]. The ADIOS [9] framework provides applications with a general I/O API to switch among multiple file-based or streaming-based transport backends without paying the cost of rewriting the application code for each one. However, it is not always desirable, or even possible (e.g., because of legacy components in the workflow), to rewrite or patch existing workflow steps to enable in situ orchestration by using ADIOS API or equivalent frameworks. For this reason, we propose CAPIO (Cross-Application Programmable I/O)[1], a new 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 modules in scientific workflows where the file system is used to allocate files used as communication buffer 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). It also shifts I/O coordination in workflows toward a declarative approach through a new, I/O-tailored coordination language based on the JSON syntax. Users can specify input/output file dependencies of steps and annotate them with synchronization semantics information to enable (or improve) pipeline execution between steps. In doing that, CAPIO avoids changing the existing codebase. It intercepts POSIX SCs of processes composing the different workflow steps and bents their execution according to the user-provided

[1]CAPIO: https://github.com/High-Performance-IO/capio

Authors copy of Alberto R. Martinelli, Massimo Torquati, Marco Aldinucci, Iacopo Colonnelli, and Barbara Cantalupo, CAPIO: a Middleware for Transparent I/O Streaming in Data-Intensive Workflows, in Proc. of the IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC), IEEE, 2023. DOI: 10.1109/HiPC58850.2023.00031

semantics information. The aim is to transparently overlap the execution of I/O phases and computation-I/O phases in file-based workflow steps.

The paper makes the following contributions:

- We propose a set of *commit and firing rules* defining file synchronization semantics in consecutive workflow steps.
- We propose a new *JSON-based language to coordinate I/O operations* in traditional and in situ workflows.
- We describe the internals of the CAPIO middleware.
- We demonstrate through a set of benchmarks and two realistic applications the feasibility of the proposed approach and its performance benefits.

The paper's outline is as follows: Sec. II presents the background and related works. Sec. III discusses CAPIO I/O optimizations. Sec. IV introduces the CAPIO middleware with the coordination semantics, the JSON-based language, and the runtime. Sec. V presents the experimental evaluation conducted. Sec. VI concludes the paper.

II. Background and Related Work

A. Workflows

Workflow graphs are widely used to model and execute complex scientific applications on large-scale distributed architectures. A workflow specification incorporates two different classes of semantics [10]: the *host semantics*, which defines the subprogram in each workflow step, and the *coordination semantics*, which defines the interactions between steps. Tools in charge of exposing coordination semantics to the users and orchestrating workflow executions are called *Workflow Management Systems* (WMSs) [11].

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) [12], whereas others adopt a general-purpose programming language (e.g., Pegasus [13]) or a GUI (e.g., Jupyter Workflow [14]). This approach is very flexible, as it does not impose constraints on the host application code and does 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 tokens are propagated to consumer steps 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) of the directly connected steps.

In scientific workflows, tokens carry data, and in most implementations, such data are *files*[2] [2].

B. Optimize I/O behavior

In HPC facilities, the contention for the I/O bandwidth of the shared file system is often the main obstacle to scalability [15]. Consider the two steps pipeline workflow in the left-hand side of Fig. 1. Step S produces k files, consumed as input tokens by the second step Q. The files are stored on a shared file system providing 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:

$$\max\left(T_C, T_{I/O}\right) \leq T_T \leq T_C + T_{I/O} \tag{1}$$

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

$$T_{I/O} = k \cdot \left(\frac{N}{wB} + \frac{M}{rB}\right) \tag{2}$$

According to Eq. (2), two classes of techniques aiming to mitigate the I/O overhead can be identified in the literature.

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. Burst buffers [3] and ad hoc file systems [4] rely on high-end storage technologies (e.g., SSD or NVMe) or store intermediate results in memory to increase the available I/O bandwidth [16]. Plus, they transfer data directly among the HPC nodes, reducing the bandwidth contention on the shared file system. On the other hand, parallel I/O interfaces (e.g., OrangeFS/PVFS [17], MPI-IO [18]) aim to better utilize the available I/O bandwidth by allowing multiple processes to read/write different portions of a file in parallel. Libraries such as HDF5 [19] and ADIOS [9] 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. Some ad-hoc file systems, such as GekkoFS [20], embraced relaxed POSIX semantics to enhance performances [21].

The second class of I/O optimizations aims to minimize the numerator of Eq. (2). The so-called in-transit data processing model belongs to this second class [6]. 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 $T_{I/O}$ but also increase T_C. For example, some I/O libraries (e.g., Damaris [22]) dedicate some of the cores of an HPC worker node to perform asynchronous I/O operations and processing.

Eventually, a different approach for optimizing I/O in data-intensive workflows involves enhancing the in situ workflow model with I/O streaming behavior to overlap I/O and computation between consecutive steps. As we will discuss in Sec. III, the CAPIO middleware aims to do this *without modifying/patching the application code*. To our knowledge, no other tools adopt this approach using a declarative I/O coordination model.

[2]We will use the metonym *tokens* to indicate *files* when they are used to coordinate successive steps in a workflow.

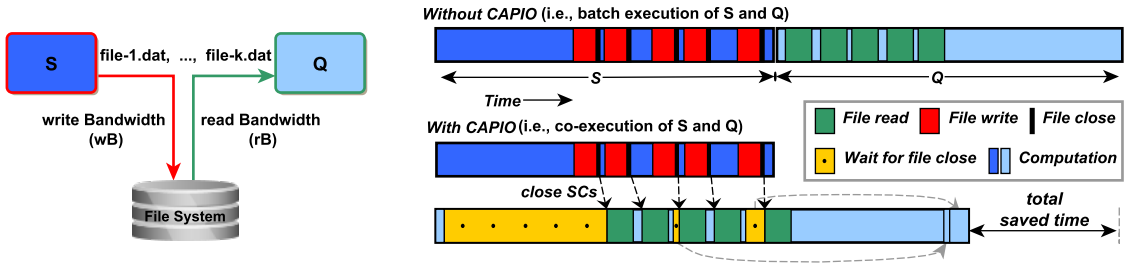


Fig. 1: `Left`: A two steps workflow whose tokens are files. `Right`: Loosely-coupled batch execution of the S and Q steps vs. their concurrent execution with CAPIO and the "*on-close file commit*" file synchronization semantics.

III. I/O OPTIMIZATIONS WITH CAPIO

Let us consider the two-step pipeline in Fig. 1 (left). A possible technique to shorten $T_{I/O}$ is to overlap the I/O phases of the two stages.

In the ideal case of full overlap, the Eq. (2) can be rewritten as follows:

$$T_{I/O} \approx \max\left(T^S_{I/O}, T^Q_{I/O}\right) = \max\left(\frac{kN}{wB}, \frac{kM}{rB}\right) \quad (3)$$

The challenge is introducing such streaming optimizations *without modifying the business code of the workflow steps* involved, which means reinterpreting the semantics of existing file access primitives rather than substituting them with semantically richer I/O calls (e.g., ADIOS).

The CAPIO user-space I/O middleware aims to enable these optimizations through a) concurrent execution rather than batch execution of workflow steps for which the I/O operations have to be optimized; b) by enabling a more relaxed synchronization semantics for tokens propagation than the standard *on-termination* one (cf. Sec II-A). Concurrent execution of workflow steps enables exploiting temporal parallelism (i.e., pipelining). Relaxing the I/O synchronization semantics enables anticipating consumer operations on a given file without introducing side effects on program execution. CAPIO introduces the so-called '*on-close file commit*' semantics, i.e., the consumers of a file f may start reading its content not before all producers of data in f have closed the file. This semantics gives the following information: a) the file is ready to be read (i.e., the corresponding token is *fireable*) as soon as the file is closed by all producers; b) the file is completed (i.e., *committed* into the file storage) when it is closed by all producers. Also, it implicitly states that the producers will not re-open the file.

The *on-close file commit* semantics is more relaxed than the standard *on-termination* semantics, which requires that all producer steps terminate before the consumer steps may open the file and start reading its content, i.e., previous points *a)* and *b)* are both associated to the termination of the producer steps. Additionally, it enables the temporal overlap of distinct I/O phases between two consecutive steps, i.e., producers and consumers work on distinct tokens. For example, if Q in our example can be executed according to the new semantics, its `open` (and `read`) SCs to the file f can be paused until all `close` SCs have been completed at S. Therefore, the writing of data into the file f_{i+1} by S can be overlapped (or partially overlapped) with the reading of data from file f_i by Q. CAPIO can transparently enable this behavior by intercepting POSIX SCs issued by S and Q and forcing them to execute according to the user-provided file synchronization semantics. Fig. 1 (right-hand side) exemplifies the traditional executions of S and Q and their execution with CAPIO and the *on-close file commit* semantics.

Notice that the makespan gain of the model described by Eq. (3) over the one described by Eq. (2) is not only the part related to I/O phases overlap (i.e., data movement). We should also consider possible computation and I/O overlap typical of pipeline computations (as exemplified in Fig. 1). The degree of CAPIO optimizations, and thus the amount of different phases overlapping, strongly depends on how relaxed the file commit semantics in the workflow steps could be. A common case is when a producer writes chunks of data into a file in *append-only* mode (i.e., there is no data update), and the consumer continuously reads and processes such data chunks. The data movement granularity between producer and consumer steps is not the entire file as in the on-close file commit semantics but it could be as small as the data chunk of a single read SC. Provided the data chunk is not too small (which would increase the SC handling overhead), this enables pipeline parallelism between the producer and consumer steps at chunk granularity. In this notable case, producers and consumers can overlap even within a single I/O phase (i.e., the file is accessed concurrently).

Introducing all mentioned optimizations and thus moving from a traditional file-based loosely-coupled model to a streaming model without modifying the application code is challenging because common I/O APIs (e.g., POSIX, MPI-IO, HDF5 Async API) do not carry enough information on the whole I/O phase beyond the single operation. Observing the execution of producers and consumers while it happens in sequential order (i.e. the default behavior) is not generally possible to state that a given coordination relaxation will be correct. For this CAPIO adopts a different approach. The user must provide extra-functional information to suggest the correct coordination relaxation to enable transparent streaming. For this reason, the CAPIO middleware provides the workflow programmers with an *I/O coordination language*

to annotate the semantics (coordination relaxation) of data movement between different workflow steps.

IV. CAPIO MIDDLEWARE

The CAPIO middleware is made of two logical tiers. The higher tier defines a *coordination model* allowing the user to express relaxed token synchronization semantics between producer-consumer workflow steps via an *I/O coordination language* (currently in the JSON format) 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 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 mount point*).

In the rest of this section we introduce the token synchronization semantics (Sec. IV-A); we describe how these semantics can be expressed by the user through the JSON configuration file (Sec. IV-B), and we describe the CAPIO runtime (Sec. IV-C).

A. Commit and Firing rules

To define the file synchronization semantics between consecutive workflow steps, we should consider two temporal aspects: *a)* when there are no more updates to the file; *b)* when a consumer can safely start reading (portion of) data written in the file. We refer to the first as the *commit rule* and to the second as the *firing rule*. From the producer-consumer paradigm perspective, a file can be seen as a data stream. The *firing rule* defines when consumer steps may consume stream data items produced by producers' steps. It can be immediately or later based on some events. Instead, 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* allows us to define two distinct file commit behaviors: 1) *on-termination* and 2) *on-close*. The *on-termination* behavior is used in traditional file-based execution of workflow steps. When a step terminates, all data files produced are committed to the file system and ready to be read by all consumer steps. Instead, the *on-close* behavior enables the consumer to start reading a file as soon as I/O operations from all producers on that specific file are completed. Such I/O operations completion is notified with the `close` SC by each producer. In addition to these two primary file commit behaviors, we can consider a file committed when another is committed. This is useful when the number of opens and closes operations for a given file is not known statically, but we know that the I/O operations are completed if another file is committed. These additional commit behavior creates a commit rule dependency among files and widens the opportunities

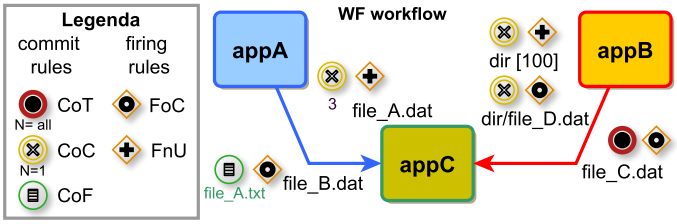


Fig. 2: Workflow example whose files dependencies between steps are annotated with the CAPIO commit and firing rules.

to exploit temporal parallelism for I/O operations in distinct workflow steps.

The commit semantics of producer(s) of a `fileX` can defined as follows:

- *Commit on-Termination* (CoT): the `fileX` is completed *iff* all producers have terminated.
- *Commit on-Close* (CoC): the `fileX` is completed *iff* all producers have definitely closed the file.
- *Commit on-File* (CoF): the commit semantics of `fileX` depends on the semantics of another file (`fileY`).

The *firing rule* states when data are ready to be consumed: if the commit rule holds for a given file, the file is undoubtedly ready to be consumed, i.e., the commit rule implies the firing rule of the entire file. We call this base firing rule *Firing on-Commit* (*FoC*). However, portions of the file (i.e., those parts already written by the producers) could be immediately ready to be consumed (i.e., fireable), provided the producers do not update them. We call this behavior *Firing no-Update* (*FnU*), i.e., the file content is ready to be read as soon as data is written into the file.

Two scenarios are noteworthy. Firstly, when a consumer step attempts to open a file (present in the CAPIO config file) that has yet to be created. The CAPIO runtime will pause the process executing the `open` SC, i.e., the `open` will not return until the file is created. Secondly, when a consumer step tries to read a portion of a file that has not been already written. The behavior is more nuanced because the `read` SC may return fewer data elements (or even 0) than what was requested. The consumer process issuing the `read` will be paused by CAPIO until one of the following conditions is met: *a*) the requested data is entirely produced, and the read SC returns the total number of bytes requested; *b*) all producers close the file (in the case of commit semantics being *CoC*) or terminate, thus the `read` SC will return the current number of bytes read if any, or 0 to indicate the end of the file (`EOF`).

B. The CAPIO coordination language

This section describes how the synchronization semantics introduced in Sec. IV-A can be expressed through a JSON configuration file. The JSON keywords and their syntax rules define the *CAPIO coordination language* of I/O operations in workflows.

Fig. 2 shows a simple 3-step workflow (named *WF*) whose CAPIO configuration file is reported in Fig. 3. The three

```
1 { "name" : "WF",
2   "IO_Graph" : [
3    { "name" : "appA",
4      "output_stream": [{
5      "group_name": "group0", "files": ["file_A.dat","file_B.dat"] }],
6      "streaming": [
7        { "name" : "file_A.dat", "committed":"on_close:3", "mode":"no_update"},
8        { "name" : "file_B.dat", "committed":"file_A.dat", "mode":"update"} ]
9    },
10   { "name" : "appB",
11     "output_stream" : ["file_C.dat","dir"],
12     "streaming" : [
13       { "name":"file_C.dat", "committed" : "on_termination", "mode":"update"},
14       { "name":"dir", "type":"d", "nfiles":100,
15         "committed": "on_close", "mode":"no_update" }
16       { "name" : "dir/file_D.dat", "committed":"on_close", "mode":"update"} ]
17   },
18   { "name" : "appC",   "input_stream" : ["group0", "dir", "file_C.dat"] },
19  ],
20  "permanent" : [ "dir/*", "file_C.dat"],
21  "home_node" : [ { "files": [ "group0", "file_C.dat" ], "node": "appC" } ]
22 }
```

Fig. 3: CAPIO configuration file of the workflow in Fig. 2. It describes the files' dependencies and semantics annotations.

workflow steps in the figure produce and consume 3 files (`file_A.dat`, `file_B.dat`, and `file_C.dat`), and one directory (`dir`) containing 100 files. Each I/O token (i.e., file or directory) is annotated to declare its commit and firing rules (the symbols close to the token name in Fig. 2).

Fig. 3 shows the syntax to declare the workflow steps, their token dependencies, and semantics annotations for each token. The keyword `IO_Graph` denotes a directed graph whose nodes are workflow steps, and arcs are data dependencies (i.e., files and directories tokens). Each application step is identified by a `name`. It declares the tokens consumed and those produced using the keywords `input_stream` and `output_stream`, respectively. The user can also define an alias for a set of files with the keyword `group_name`. Then, the group name can be used elsewhere to refer to all files and directories. The language also supports `Bash`-style wildcards (e.g., file_*.???, dir/*).

With the keyword `streaming`, each token (or a group of tokens) of the `output_stream` can be annotated with the commit rule (keyword `committed`) and with the firing rule (keyword `mode`). For example, at line 13 of Fig. 3, the token `file_C.dat` produced in output by the step `appB` has `on_termination` (CoT) as commit rule, and the *Firing on-Commit* (*FoC*) as firing rule (expressed with the keyword value `update`). FoC models the default batch behavior in traditional file-based workflows. It means that the file data can be read by `appC` only when step `appB` has terminated. Conversely, the token `dir` (the keyword *type* tells the CAPIO runtime that the token is a directory) is annotated with the commit rule `on_close` (CoC) and the firing rule *Firing no-Update* (FnU) set with the value `no_update` (line 15), which means that tokens contained in the directory `dir` can be read as soon as the data is produced, and, for each token, the data stream ends when it is closed by all producers. The semantics behavior of tokens in a directory can be specialized, providing specific rules for some of the files in the directory. For example, the file `file_D.dat` in the directory `dir` has a different firing rule (line 16).

If not differently stated, the default directory commit semantics is `on_termination`. It means the total number of files in the directory will be known when all producers terminate. If the number of files is instead known statically, it can be provided to the CAPIO runtime through the `nfiles` keyword (line 14). This way, for our workflow WF, after 100 files are produced, CAPIO will notify the consumer step that no more files will be produced in the directory *dir*, and the standard C-style 'readir-loop' through all files in a directory executed by the consumer may terminate.

At line 7, the token `file_A.dat` is annotated with `CoC` and `FnU`. However, if the file is opened and closed multiple times, CAPIO must consider the file committed only at the last `close`. The number 3 associated with the `on_close` keyword (line 7), specifies the number of `close` SCs CAPIO has to wait for before considering the file committed.

If there is more than one writer for a given file, the behavior does not change. However, for the *CoC* semantics, it is paramount to set the correct number value in the `on_close` keyword equal to the number of writers by the number of `close` SCs performed by each writer. Similarly, for the *CoT* semantics, it is possible to specify how many writers must terminate before considering the file committed (e.g., `on_termination:3`). If the number value is not provided, it means that *all* producers must terminate.

Line 8 in Fig. 3 shows how to specify the *CoF* semantics for the file `file_B.dat`, whose commit rule is associated to the one of `file_A.dat`. Specifically, the *file_B.dat* is considered committed when the `file_A.dat` is committed. The firing rule for the `file_B.dat` is FoC (keyword `no_update`).

With the keyword `permanent` (line 20), the user states which files must be permanently saved into the storage after the termination of the workflow steps. All files are considered temporary if the `permanent` keyword is not defined.

At line 21, the keyword `home_nodes` allows the user to specify the node where a set of files with their metadata will be stored. The default policy (i.e., when the keyword is not provided) is to store the file data and metadata in the node where the file will be created. The home-node concept was inspired by page-based software Distributed Shared-Memory implementations [23]. The user may explicitly set the reference node of each single (or group) of a file(s) by choosing as home-node the node where a given application step is running. This is done by setting in the config file the name of the step. For example, line 21 shows that for the files contained in the `group0` and the file `file_C.dat` the home-node is the node where the `appC` application step will be running. In this case, we assume that `appC` is a single-process application. Differently, with the notation `appC:N`, we state that the home-node is the node where the *appC*'s process with logical id N will be running. In this way, the user can fully define a static mapping between files and computing nodes. Another supported policy is the `hash` one, in which the reference node

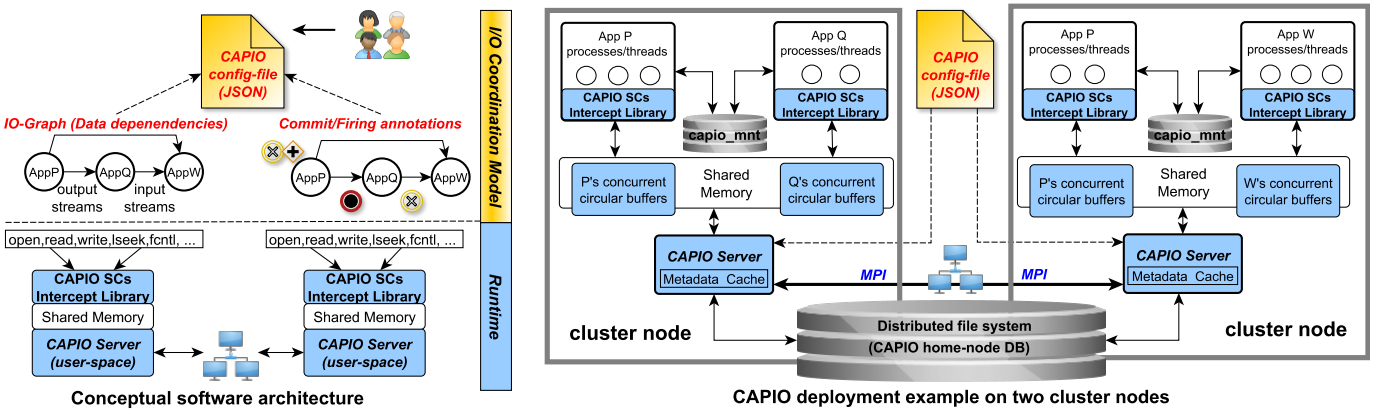


Fig. 4: `Left`: The CAPIO software layers: the JSON-based *configuration file* embeds I/O data dependencies and files' annotations describing commit and firing rules; the *CAPIO SCs Intercept Library* (SC-IL) coupled with the per-node *CAPIO Server* enable in-memory files store by intercepting POSIX I/O SCs. They enforce synchronized access to files. `Right`: CAPIO deployment on 2 cluster nodes: AppP executes on both nodes; the local node `capio_mnt` directory is the CAPIO FS entry point for SC-IL. The distributed FS is used only to store *home-node* information of the files.

for a file with a given *pathname* is dynamically selected among all the compute nodes where the workflow is running whose logical id results from the hashing of the pathname through the `std::hash` function.

Notice that the CAPIO configuration file could be directly generated by the WMS describing the entire application workflow. This is particularly useful for complex workflows. In particular, the part related to the I/O data dependencies (i.e., the IO-Graph) can be entirely generated starting from the application description. Conversely, the tokens synchronization semantics requires adding explicit annotations at the workflow description language level, such as, for example, the `streamable` keyword used in the CWL standard indicating that the given file is read or written sequentially without seeking [24].

C. The CAPIO runtime

CAPIO's runtime is composed of a set of per-node user-space servers implementing distributed data storage for a given workflow. It uses the information in the JSON configuration file provided as an input argument to enforce data streaming of file content between workflow steps.

The software architecture of CAPIO is sketched in Fig. 4. It is implemented in C++ and uses MPI only for server-to-server communications. For each node, the user specifies a CAPIO local-node mount point through the `CAPIO_DIR` environment variable (i.e., `capio_mnt`). CAPIO captures all I/O SCs executed on files and directories inside the `capio_mnt` directory. All SCs targeting files outside the CAPIO local mount point are forwarded to the kernel. CAPIO implementation supports both multi-process and multi-threaded applications.

The *CAPIO Intercept Library* (SC-IL) implemented using the Linux-x86_64 *system call intercepting library* `syscall_intercept`[3] is a shared library dynamically linked to the steps of the workflow (through the `LD_PRELOAD` dynamic linker environment variable) to capture the I/O SCs executed on files and directories inside the local-node mount point. SC-IL communicates with the local CAPIO server through a *concurrent circular buffer stored in a POSIX shared-memory segment*. The CAPIO server identifies workflow's application steps by their names to match the semantics information in the configuration file (i.e., the keyword `name` in Fig 3). The name is set at the launch time of the workflow step by using the environment variable `CAPIO_APP_NAME`.

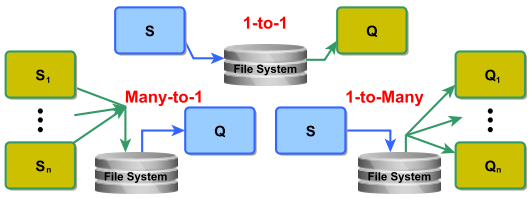
For example, to launch the *appA* step with CAPIO on the cluster node `nX` the commands are:

```
nX> capioServer.sh Config.json
nX> LD_PRELOAD=<libcapio_posix.so> \
    CAPIO_DIR="/capio_mnt" CAPIO_APP_NAME="appA" \
    appAexe <application arguments>
```

where `capioServer.sh` runs the CAPIO server in background passing the configuration file `Config.json`.

By default, the file data (and metadata) are stored in the main memory of the node where the file is created (the default *home_node* file mapping policy). 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 using MPI. However, as described in the previous section, file data placement can be controlled by setting the `home_nodes` keyword in the CAPIO configuration file. In the current implementation, CAPIO supports two per-file home-node policies: *local home-node* (i.e., the node that creates the file – this is the default policy), *single remote home-node* using both a static mapping or a dynamic mapping through pathname hashing. Let us consider the default policy: the first process that creates a file in the application workflow elects the node that runs as the file's home-node. This informa-

[3]syscall_intercept: https://github.com/pmem/syscall_intercept




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

Fig. 5: `Left`: I/O pattern benchmarks tested with CAPIO. `Right`: The CAPIO configuration file used for all benchmarks.

tion is shared with all other CAPIO servers by writing a record in a *home-node system DB* stored in the distributed file system (see Fig. 4). Such DB, implemented using standard POSIX files, 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 dynamically. The same implementation schema is followed for the *single remote home-node* with pathname hashing. Conversely, for the policy *single remote home-node* with static assignment, the home-node of a file is statically specified in the CAPIO configuration file. Thus all CAPIO servers statically know the mapping and do not access the *home-node system database* to know the home-node for those files. The policies that elect the home-node at runtime are more costly, but they are more flexible and easier to use for the user that does not need to set an explicit mapping in the config file. For this reason, in the experimental section, we will study the performance of the CAPIO runtime configured with the default policy.

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.

V. Evaluation

Here, first we present some tests measuring the overhead introduced by the CAPIO intercept library, and then we present the results obtained using the CAPIO middleware on synthetic benchmarks simulating typical I/O workflow patterns and on two scientific workflows. The first synthetic benchmark (MapReduce) emulates standard in-memory MapReduce computations fed by a large input dataset [25]. The second one (1000 Genomes [26]) is a DAG-based bioinformatics workflow computing human gnome mutation overlaps.

A. System Configuration

We performed our experiments deploying the CAPIO middleware on the GALILEO100[4] Tier-1 supercomputer hosted by CINECA supercomputing center[5]. 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 OS is Centos 8.3.2011, and the Linux kernel version is 4.18.0-240. The storage system is based on *Lustre* open source parallel files system [27]. 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 entire workflow is submitted as a single *SLURM* job using a Bash script that spawns the CAPIO servers (via mpirun) on the reserved nodes and then starts all workflow steps. The CAPIO servers were configured to use the default *local home-node policy* (cf. IV-C). We executed 10 runs for each test and then computed the average value and standard deviation.

[4]GALILEO100: https://www.hpc.cineca.it/hardware/galileo100
[5]CINECA: https://www.cineca.it/en

B. The SC's intercept overhead

The first tests aim to understand how much overhead the CAPIO SC intercept library introduces. We used the *lmbench* benchmarks [28], a series of micro benchmarks measuring OS and HW system metrics. In particular, we considered the `lat_syscall` benchmark measuring the latency of some simple SCs.

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

We tested two cases: 1) no LD_PRELOAD, that means no SC intercept; and 2) the CAPIO intercept library by setting `LD_PRELOAD=libcapio_posix.so`. The results obtained by averaging 5 repetitions are reported in Table I. Overall, the overhead introduced by the CAPIO intercept library is relatively small.

C. Synthetic Benchmarks

The synthetic benchmarks provide relatively simple scenarios highlighting the potential impact of CAPIO optimizations on real use cases. These benchmarks are designed to mimic three recurrent I/O patterns in workflows: *a)* one producer and one consumer steps (*1-to-1*); *b)* one producer

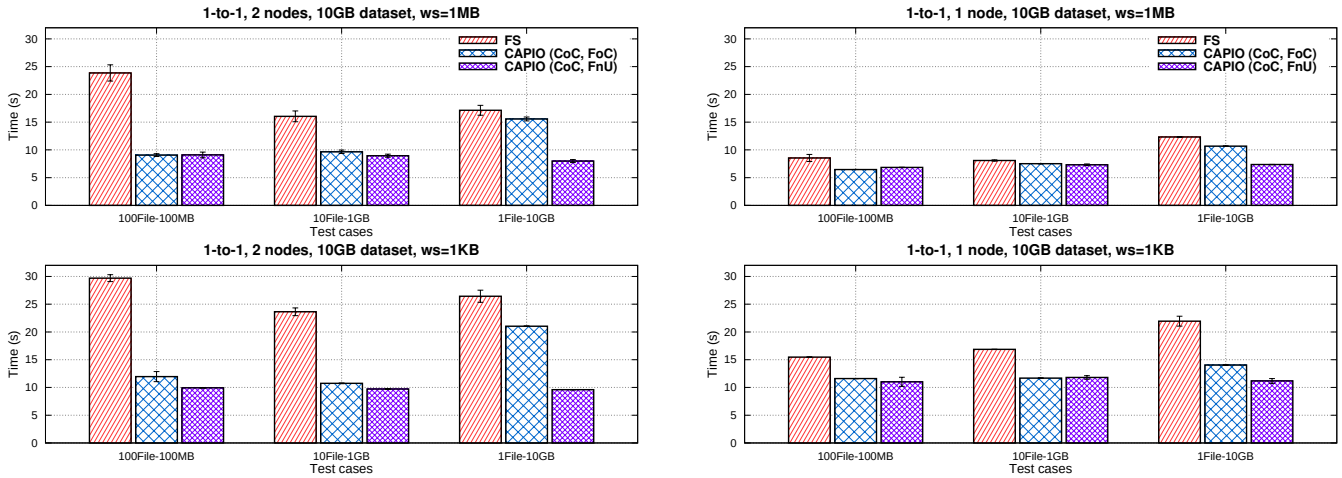


Fig. 6: *1-to-1* benchmark test with 10GB dataset and two different sizes for the read/write buffer ws: large (1MB) in the top plot; small (1KB) in the bottom plot. `Left`: *S* and *Q* executed on 2 cluster nodes. `Right`: *S* and *Q* executed on 1 node.

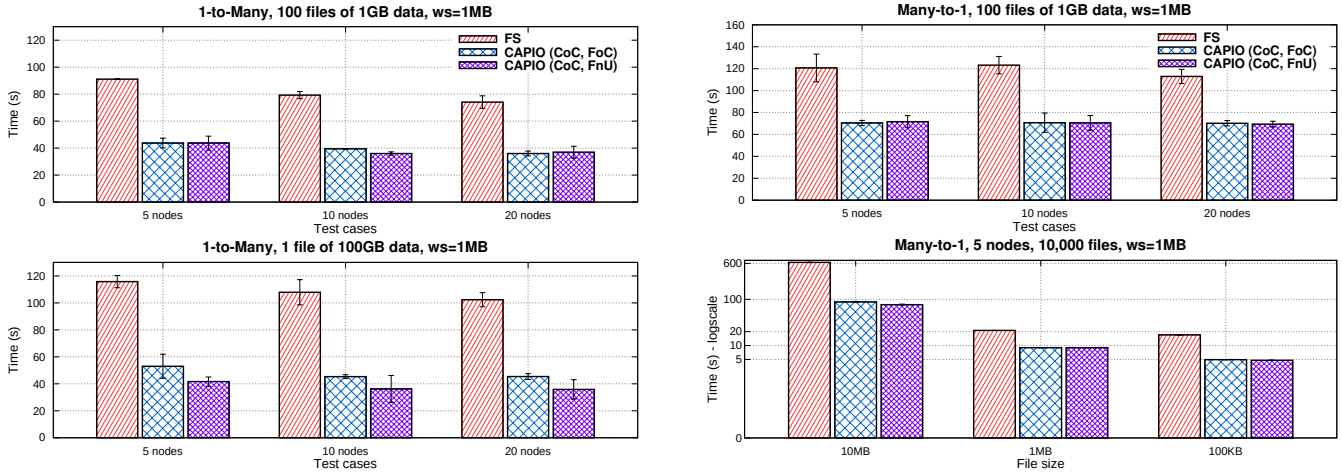


Fig. 7: `Left`: *1-to-Many* test executed with 100 files of 1GB each (top plot) and one single file of 100GB (bottom plot). `Right`: *Many-to-1* executed with 100 files of 1GB each (top plot), and $10,000$ files of different size on 5 cluster nodes (bottom plot – log. scale). For all tests, the buffer size is $ws = 1$MB.

and many consumers steps (*1-to-Many*); *c)* many producers and one consumer steps (*Many-to-1*). They are sketched in Fig. 5 (left-hand side) together with their CAPIO config file used for all tests (right-hand side). The *S* and *Q* steps of the benchmarks are written in `C` and use standard POSIX `fopen`/`fclose`/`fwrite`/`fread`/`feof`/`lseek` library calls for handling I/O. Besides simple checksum checks for verifying the results' correctness, *S* and *Q* steps perform only I/O operations. We tested the following cases:

1) *FS*: file-based execution in which producer and consumer steps are executed in sequence using *Lustre*.
2) *CAPIO-CoC-FoC*: CAPIO execution with *Commit on-Close* and *Firing on-Commit* semantics for all files (i.e., *"MODE"="update"* at line 4 in Fig. 5).
3) *CAPIO-CoC-FnU*): CAPIO execution with *Commit on-Close* and *Firing no-Update* semantics for all files (i.e., *"MODE"="no_update"* at line 4 in Fig. 5).

For CAPIO executions, first, we started the CAPIO servers on each allocated node, the consumer step(s), and then the producer step(s). The reported time is the workflow's makespan.

1-to-1: In this benchmark, the producer writes N files each of size M using a buffer size of ws KB, whereas the consumer reads all N files using the same ws buffer size. Fig. 6 (left-hand side) shows the results obtained using 2 cluster nodes and different values of N, M, and ws. Specifically, we have the following test cases: *100Files-100MB*) $N = 100$, $M =$ 100MB; *10Files-1GB*) $N = 10$, $M =$ 1GB; *1File-10GB*) $N = 1$, $M =$ 10GB. For the buffer size, we tested two cases $ws =$ 1MB (top plot) and $ws =$ 1KB (bottom plot). Overall, CAPIO-based tests exhibit consistently higher performance for all tests from 44% to 66%. Another advantage is its lower variance than *FS* since it does not depend on the file system utilization, which is typically high in large-scale production supercomputers with hundreds of users. Small buffer size (i.e., $ws =$ 1KB) in I/O operations results in higher execution times due to the higher number of SCs and `libc` internal

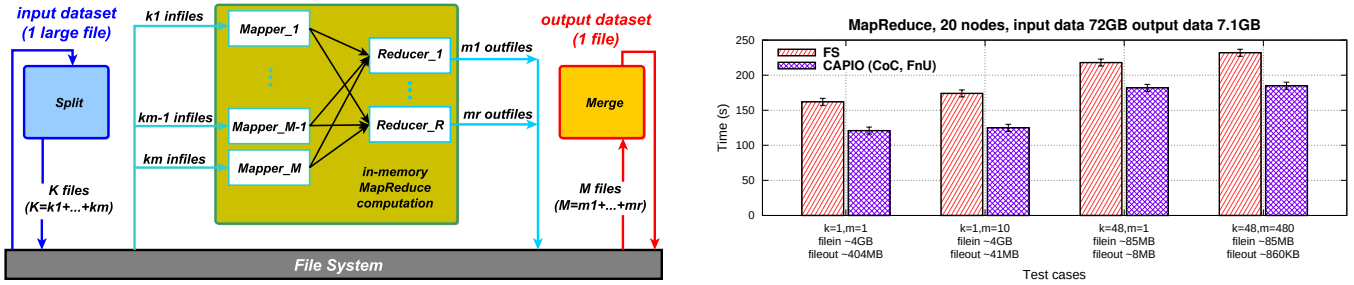


Fig. 8: *MapReduce* use case. `Left`: Split, MapReduce, and Merge workflow steps. `Right`: Execution time obtained with the standard file system deployment (FS – Lustre) and with CAPIO on 20 GALILEO100 nodes, using 20 MapReduce instances.

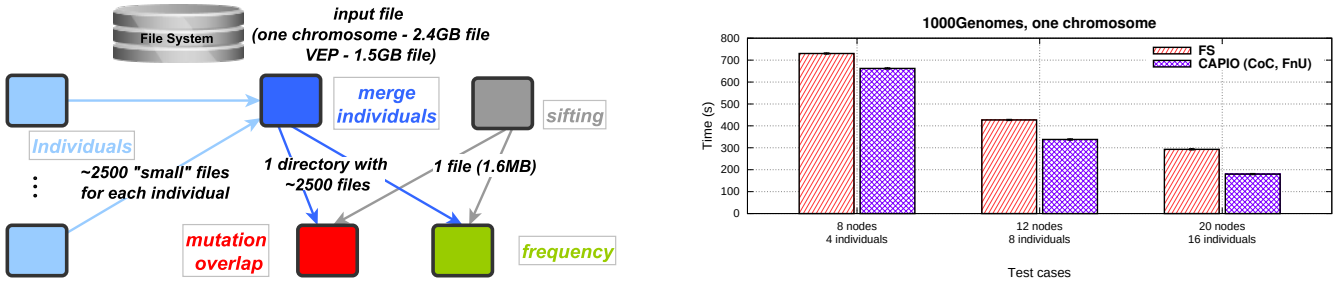


Fig. 9: *1000 Genomes* use case. `Left`: workflow steps for one chromosome. `Right`: Execution time of the C++-based version obtained with the standard deployment (file system-based – FS) and with CAPIO on 8, 12, and 20 GALILEO100 nodes.

buffer flush operations. However, CAPIO is less sensitive to this aspect than the file system. Finally, as expected, the *CAPIO-CoC-FnU* synchronization semantics performs better than *CAPIO-CoC-FoC* when there are a few large files (e.g., the test *1File-10GB*). The right-hand side of Fig. 6 shows the results of the same set of tests when both *S* and *Q* steps are deployed on the same cluster node. The CAPIO benefits are capped by aggressive OS file system caching in this case, except for the case *1File-10GB* where the maximum gain is about 49% with $ws = 1$KB. Additionally, the deployment on the same node might have significant effects depending on the kind of workflow steps executed. Strict I/O-bound workflow steps can be effectively co-executed with CAPIO on the same node because the potentially overwhelmed file system is completely removed from the I/O path. On the contrary, mixed I/O-/CPU-bound workloads might exhibit different behaviors due to the CPU cores and memory bandwidth sharing.

1-to-Many: In this benchmark, we tested two cases shown in Fig. 7 (left-hand side): *1)* the producer writes $N = 100$ files, each of size $M = 1$GB and each consumer reads a disjoint set of files (top plot); *2)* the producer writes one file ($N = 1$) of $M = 100$GB and the consumers read disjoint partitions of the file (bottom plot). The number of consumer steps tested is 5, 10, and 20. As for the *1-to-1* test, the producer step is faster because of the *local home-node policy*; thus, the execution time is dominated by the consumers' reads. In the first test, there is almost no difference between the two firing rules tested as in the *1-to-1* test for file sizes of 1GB. Here, the writing time for producing the file with id k is overlapped with the reading time of the file with id $k - 1$ by one of the consumers. Since files are consumed in increasing order of their ids, there is no appreciable difference in the execution time when increasing the number of consumer steps (i.e., nodes). The second test is logically analogous to the first one (both mimic a scatter communication pattern) but differs in the implementation since it uses a single large sparse file written after seeking. Again, CAPIO demonstrates consistent results with those of the first test. Specifically, the *CAPIO-CoC-FnU* synchronization semantics performs better than *CAPIO-CoC-FoC* due to higher I/O phase overlap between producer/consumer steps. The CAPIO improvement in the two tests ranges from 50% to 65%.

Many-to-1: In this benchmark, we tested two cases shown in Fig. 7 (right-hand side): *1)* the producers write a total number of 100 files, each of size 1GB (top plot); *2)* the producers write a large number of small files (10,000); we used three different file sizes: 10MB, 1MB, and 100KB. In the first test, the number of producer steps tested is 5, 10, and 20, whereas in the second case, the total number of workflow steps is fixed to 6 (i.e., 5 *S*s and 1 *Q*). From the qualitative standpoint, the first test presents results similar to those in the first test of the *1-to-Many* benchmark even if they emulate different communication patterns, i.e., scatter vs gather. However, the absolute execution time is higher for both the FS and CAPIO cases. In the second test (bottom plot in Fig. 7), the FS is hard-pressed when there are many files, and the dataset is relatively large (up to $100GB$). CAPIO introduces less overhead in all cases tested. With $10,000$ files of $10M$ the speed-up is more than 8 ($\sim$ 630s for FS vs. $\sim$ 75s for CAPIO).

D. Use cases

MapReduce: We have implemented a simple workflow in C that captures the typical I/O pattern found in Map-Reduce computation in which the intermediate results are stored in the main memory of the cluster nodes [29]. It consists of three steps, as sketched in Fig. 8. The sequential 'Split' step gets as input a large file and generates K smaller files. The parallel 'MapReduce' step (composed of several instances of Mappers and Reducers) reads the K files and produces M output files. Each Mapper instance reads a partition (of size k) of the K files and produces a subset (of size m) of the output files (e.g., suppose 4 Mappers and 3 Reducers, $K = 20$ and $M = 9$, then each Mapper reads $k = 5$ files and each Reducer produces $m = 3$ files). The final step, 'Merge' reads all M files and produces a single output file. The communication between the first and second steps is a *1-to-Many* I/O pattern, while the communication between the second and third steps, where files from multiple sources are consolidated into a single destination, is a *Many-to-1* I/O pattern. Fig. 8 (right-hand side) shows the execution time obtained when running the workflow using *Lustre* (FS) and CAPIO configured with *Commit on-Close* and *Firing no-Update* synchronization semantics. The input dataset is 72GB large. The final output file is about 7GB. The number of files created by the 'Split' and 'MapReduce' steps are the parameters k and m, respectively. We tested values for k and m are reported in the plot. Increasing the number of files produced reduces their size. The CAPIO deployment reduces the execution time by 22% to 33%.

1000 Genomes: Fig. 9 (left-hand side) shows the five steps of the workflow and their dependencies. The 'individuals' step 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 files ($1 - 15$KB with 16 instances). The 'sifting' step runs in parallel with all 'individuals' steps. The 'individuals_merge' step 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 steps, 'mutation_overlap' and 'frequency', are independent. They read the input dataset and the data produced by previous steps. We tested CAPIO with *CoC-FnU* synchronization semantics to exploit pipeline parallelism among the steps. 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 is unattainable with the traditional workflow execution model* (i.e., FS). The 1000 Genomes workflow was originally implemented using Bash and Python scripts. We re-implemented the entire workflow using C++ and the Boost library, which reduces the total execution time by more than $3\times$. We tested CAPIO with the fastest version using one single chromosome simulation (different simulations on different chromosomes generate independent workflows that can be executed in parallel). We tested it with three configuration 8,12, and 20 cluster nodes, thus 4, 8, and 16 'individuals' instances, respectively (see Fig. 9, right-hand side). With 16 individuals, the files produced are $16 \times 2,504 = 40,064$. Overall, CAPIO reduces the execution time over FS in the range from 10% to 39%.

VI. Conclusions and future work

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. This paper proposes the CAPIO middleware. It transparently injects I/O streaming capabilities into file-based workflows. CAPIO leverages an *I/O coordination language* (based on JSON syntax), which allows the user to annotate workflow data dependencies with synchronization semantics information to enrich the POSIX-based I/O system calls semantics and enable temporal overlap of distinct workflow steps. We presented the synchronization semantics currently supported by CAPIO and the corresponding language annotations to be provided to CAPIO through a JSON configuration file. Using synthetic benchmarks and two workflow use cases, we validated the performance benefit of using the CAPIO middleware vs. the Lustre parallel file system on a production supercomputer. The results demonstrate the approach's feasibility and performance improvements in the range from 10% to 66%. We believe that CAPIO-alike tools might influence novel workflow orchestration strategies to improve temporal overlap between steps and thus reduce the peak I/O file system demand.

As future directions, we plan: a) to extend CAPIO using different communication backends to target heterogeneous distributed systems, for example, by leveraging ADIOS in the CAPIO run-time or by using multi-backend communication libraries, such as MTCL [30], which abstracts both MPI and TCP; b) to integrate CAPIO with existing WMS (e.g., StreamFlow [31]) to enhance I/O coordination among workflow steps and simplify the CAPIO deployment; c) to test CAPIO on application workflows using high-level I/O interfaces, such as MPI-I/O. Finally, we intend to test CAPIO on dedicated in-house clusters using different parallel file systems (e.g., Lustre, GPFS, BeeGFS).

Acknowledgment

This work was partially funded by:

- Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 Componente 2 Investimento 1.4: Potenziamento strutture di ricerca e creazione di "campioni nazionali di R&S (M4C2-19)" - Next Generation EU (NGEU)
- the ADMIRE EU's Horizon 2020 JTI-EuroHPC research and innovation programme project under the grant agreement No 956748

- the EUPEX EU's Horizon 2020 JTI-EuroHPC research and innovation programme project under grant agreement No 101033975.

We also thankfully acknowledge the CINECA award under the ISCRA initiative, for the availability of high-performance computing resources and support.

REFERENCES

[1] R. Ferreira da Silva, R. Filgueira, I. Pietri, M. Jiang *et al.*, "A characterization of workflow management systems for extreme-scale applications," *FGCS journal*, vol. 75, 2017.

[2] T. M. A. Do, L. Pottier, O. Yildiz, K. Vahi *et al.*, "Accelerating scientific workflows on hpc platforms with in situ processing," in *CCGrid'22: 22nd IEEE Intern. Symp. on Cluster, Cloud and Internet Comp.*, 2022.

[3] J. Bent, G. Grider, B. Kettering, A. Manzanares *et al.*, "Storage challenges at los alamos national lab," in *MSST '12: Proc. of the 28th Symp. on Mass Storage Systems and Technologies*. IEEE Computer Society, 2012.

[4] A. Brinkmann, K. Mohror, W. Yu, P. H. Carns *et al.*, "Ad hoc file systems for high-performance computing," *J. Comput. Sci. Technol.*, vol. 35, no. 1, 2020.

[5] I. Foster, M. Ainsworth, B. Allen, J. Bessac *et al.*, "Computing just what you need: Online data analysis and reduction at extreme scales," in *Euro-Par '17: Proc. of the 23rd Inter. Conf. on Parallel and Distributed Comp.*, 2017.

[6] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout *et al.*, "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.

[7] C. Sewell, K. Heitmann, H. Finkel, G. Zagaris *et al.*, "Large-scale compute-intensive analysis via a combined in-situ and co-scheduling workflow approach," in *SC '15: Proc. of the ACM/IEEE Conf. on Supercomputing*, 2015.

[8] J. Chen, Q. Guan, Z. Zhang, X. Liang *et al.*, "Beeflow: A workflow management system for in situ processing across hpc and cloud systems," in *ICDCS '18: Proc. of the 38th Inter. Conf. on Distributed Comp. Systems*, 2018.

[9] W. Godoy, N. Podhorszki, R. Wang, C. Atkins *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, 2020.

[10] E. Lee and T. Parks, "Dataflow process networks," *Proc. of the IEEE*, vol. 83, no. 5, May 1995.

[11] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso, "A survey of data-intensive scientific workflow management," *Journal of Grid Computing*, vol. 13, 2015.

[12] M. R. Crusoe, S. Abeln, A. Iosup, P. Amstutz *et al.*, "Methods included: standardizing computational reuse and portability with the Common Workflow Language," *Communications of the ACM*, vol. 65, no. 6, 2022.

[13] E. Deelman, K. Vahi, G. Juve, M. Rynge *et al.*, "Pegasus, a workflow management system for science automation," *FGCS journal*, vol. 46, 2015.

[14] I. Colonnelli, M. Aldinucci, B. Cantalupo, L. Padovani *et al.*, "Distributed workflows with jupyter," *FGCS journal*, vol. 128, 2022.

[15] J. Lofstead, F. Zheng, Q. Liu, S. Klasky *et al.*, "Managing variability in the io performance of petascale storage systems," in *SC '10: Proc. of the ACM/IEEE Conf. on Supercomputing*, 2010.

[16] J. Garcia-Blas, D. E. Singh, and J. Carretero, "IMSS: In-Memory Storage System for Data Intensive Applications," in *HPCMALL 2022. ISC High Performance 2022*, June 2022, conference.

[17] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *4th Annual Linux Showcase & Conference (ALS 2000)*. Atlanta, GA: USENIX Assoc., Oct. 2000.

[18] R. Thakur, W. Gropp, and E. L. Lusk, "A case for using mpi's derived datatypes to improve I/O performance," in *SC '98: Proc. of the ACM/IEEE Conf. on Supercomputing*, 1998.

[19] M. Folk, A. Cheng, and K. 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*, vol. 99, 1999.

[20] M.-A. Vef, N. Moti, T. Süß, M. Tacke *et al.*, "Gekkofs — a temporary burst buffer file system for hpc applications," *Journal of Computer Science and Technology*, vol. 35, 01 2020.

[21] C. Wang, K. Mohror, and M. Snir, "File system semantics requirements of hpc applications," in *Proc. of the 30th Inter. Symp. on High-Performance Parallel and Distributed Computing*, ser. HPDC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[22] M. Dorier, G. Antoniu, F. Cappello, M. Snir *et al.*, "Damaris: Addressing performance variability in data management for post-petascale simulations," *ACM Trans. Parallel Comput.*, vol. 3, no. 3, Oct. 2016.

[23] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and performance of munin," in *SOSP '91: Proc. of the 13th ACM Symp. on Operating Systems Principles*, New York, 1991.

[24] "Common workflow language user guide v1.2," http://www.commonwl.org/user_guide, accessed on 2023-05-15.

[25] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards Efficient MapReduce Using MPI," in *Recent Advances in PVM and MPI*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Springer, 2009.

[26] R. F. da Silva, R. Filgueira, E. Deelman, E. Pairo-Castineira *et al.*, "Using simple pid-inspired controllers for online resilient resource management of distributed scientific workflows," *FGCS journal*, vol. 95, 2019.

[27] P. J. Braam and P. Schwan, "Lustre: The intergalactic file system," in *Ottawa Linux Symp.*, vol. 8, no. 11, 2002.

[28] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *ATEC '96: Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*. USENIX Association, 1996.

[29] S. Shahrivari, "Beyond batch processing: Towards real-time and streaming big data," *Computers*, vol. 3, no. 4, 2014.

[30] F. Finocchio, N. Tonci, and M. Torquati, "MTCL: a Multi-Transport Communication Library," in *Euro-Par 2023: Parallel Processing Workshops. WSCC '23: International Workshop on Scalable Compute Continuum*. Springer, 2023.

[31] I. Colonnelli, B. Cantalupo, I. Merelli, and M. Aldinucci, "StreamFlow: cross-breeding cloud with HPC," *IEEE Trans. on Emerging Topics in Computing*, vol. 9, no. 4, 2021.