International Journal of Parallel Programming (2025) 53:10
https://doi.org/10.1007/s10766-025-00789-0

CAPIO-CL: The CAPIO Coordination Language

Marco Edoardo Santimaria[1] · Alberto Riccardo Martinelli[1] · Iacopo Colonnelli[1] · Barbara Cantalupo[1] · Massimo Torquati[2] · Marco Aldinucci[1]

Received: 29 September 2024 / Accepted: 13 February 2025
© The Author(s) 2025

Abstract

The performance bottleneck in file-based workflows remains a pressing issue in the realm of I/O-based workflows. To address this challenge, a novel annotation language has been developed. CAPIO-CL is positioned as an innovative I/O coordination language, enabling users to annotate data dependencies within file-based workflows with synchronization semantics pertinent to the involved files and directories. Through the information provided by the language, optimization opportunities arise in streaming and preemptive data movement. This paper serves to illustrate the semantics and syntax enabling CAPIO-CL to enhance the performance of in situ workflows without necessitating the rewriting or modification of the original workflow application steps. Finally, an analysis of CAPIO-CL is provided, taking into consideration both language expressiveness and application performance enhancement.

Keywords I/O · Coordination language · In-situ workflow · Streaming · CAPIO

✉ Marco Edoardo Santimaria
marcoedoardo.santimaria@unito.it

✉ Iacopo Colonnelli
iacopo.colonnelli@unito.it

Alberto Riccardo Martinelli
albertoriccardo.martinelli@unito.it

Barbara Cantalupo
barbara.cantalupo@unito.it

Massimo Torquati
massimo.torquati@unipi.it

Marco Aldinucci
marco.aldinucci@unito.it

[1] Computer Science Department, Università degli studi di Torino, Via Pessinetto 12, 10149 Torino, Piemonte, Italy

[2] Computer Science Department, Università degli studi di Pisa, Largo B. Pontecorvo, 3, 56127 Pisa, Toscana, Italy

Published online: 25 February 2025

1 Introduction

The I/O performance gap is still a significant bottleneck in large-scale HPC workflows [1]. Due to network contention and metadata servers' overloading, parallel file systems struggle to scale linearly with the aggregate disk bandwidth in real scenarios. With the advent of Exascale systems, this gap is fated to increase. In this setting, overlapping computation and I/O is crucial to reduce the makespan of data-intensive workflows. However, legacy workflow management systems heavily rely on shared data spaces (parallel file systems and object storages) to store and load intermediate data of file-based workflows, injecting significant delays between subsequent producer and consumer steps [2]. Streaming workflows reduce this delay by relaxing fireable semantics between subsequent steps, allowing them to be co-scheduled and to perform inter-step communications through token streams [3, 4]. However, injecting streaming communication semantics into a file-based legacy application is laborious and error-prone, especially when complex combinations of in-situ and in-transit computing optimizations come into play. On the hardware plane, high-performance burst buffers are becoming increasingly popular to release the pressure on the shared file system [5]. However, their characteristics and configuration vary among different facilities, hindering workflows' portability. Multi-backend I/O libraries that hide local storage complexities behind agnostic I/O APIs are flourishing in the HPC ecosystem [6]. Nevertheless, they require developers to modify the business code, tying it to a vendor-specific I/O API schema. Plus, most of them are unaware of I/O dependencies between subsequent steps of a complex application, requiring users to encode streaming communications in the application codebase explicitly. This work introduces a novel Cross-Application Programmable I/O Coordination Language (CAPIO-CL), a fully declarative and vendor-agnostic language aiming to describe the data plane of a complex workflow in terms of producer/consumer communications, read/write patterns, and file ownership. Different from I/O libraries, CAPIO-CL is fully decoupled from the host code, allowing workflow orchestrators to improve cross-application I/O communications transparently and also to apply other classes of optimizations, such as proactive data staging and node-local burst buffers exploitation. This work extends previous work on the CAPIO runtime library [7], the reference implementation of the CAPIO-CL, detailing the syntax and semantics of the coordination language. In detail, Sect. 2 describes the concepts behind the CAPIO-CL design and the other I/O coordination languages in the literature. Sections 3 and 4 detail the semantics and the syntax of CAPIO-CL, respectively. Section 5 shows some tests to validate the reason for CAPIO-CL to exist performance and expressiveness. Finally, Sect. 6 summarizes the benefits of introducing CAPIO-CL and illustrates how we plan to continue its development.

2 Background and Related Work

A workflow can generally be represented as a directed bipartite graph $W = (S, P, D)$, where S is the set of steps, P is the set of ports, and $D \subseteq (S \times P) \cup (P \times S)$ is the set of dependency links [8]. Workflow execution is then modeled using token-pushing

semantics [9]: steps are enabled by the presence of tokens in their input ports, and their execution produces tokens in their output ports. Let $T(p)$ be the set of tokens that transit through port $p \in P$ during the execution of a workflow W, and assume that each token is a binary variable $t \in \{0, 1\}$. More formally, a step becomes fireable when the set of tokens in its input ports meets a specific configuration called firing rule, which for any $s \in S$ s.t. $In(s) = \{p \in P | (p, s) \in D\}$ specifies how many tokens should reside in each input port of s, i.e.,

$$R(s) = \bigwedge_{p \in In(s)} \left(\sum_{t \in T(p)} t = x_p \right), \quad x_p \in [0, |T(p)|] \cap \mathbb{N} \tag{1}$$

This work focuses on those workflow representations in which tokens carry data (e.g., Coloured Petri Nets [10] or Dataflow Graphs [11]) and, in particular, workflows in which tokens represent files and directories. Given their application-agnostic nature, files represent one of the most powerful abstractions for exchanging data between independent applications, i.e., between different workflow steps. Consequently, an efficient I/O coordination strategy is a crucial component of every WMS on the market.

3 Semantics

The CAPIO-CL semantics enriches workflow tokens representing files with attributes describing their state, read/write pattern, and ownership. For the sake of simplicity, we assume that all tokens in $T(p)$ relate to file dependencies for any $p \in P$. When dealing with distributed workflows, we should also consider the execution environment. In this work, we assume that each workflow runs on top of a set of locations L (e.g., the nodes of an HPC cluster) and that the mapping function $\mathcal{M}(s) \subseteq L$ returns the set of locations on which step $s \in S$ is offloaded. Then, a distributed workflow can be written as $DW = (S, P, D, L, \mathcal{M})$ [8].

3.1 Streaming Semantics

Generalizing token-pushing semantics, CAPIO-CL adds two additional classes of rules: *commit rules (CR)*, which state when the data carried by a token have been fully propagated to an output port of a step s_i, and *firing rules (FR)*, which state when it is possible to begin to consume these data from an input port of a step s_j. In practice, given a file f, a user can define a committed rule $CR(s_i, f) = \{0, 1\}$ to express when there will be no further updates to file f and a firing rule $FR(s_j, f, CR(s_i, f)) = \{0, 1\}$ to identify when a step can safely start consuming a portion of data written in the file.

A commit rule then defines when a given data stream terminates, i.e., when the producer step $s \in S$ emits the end-of-file (`EOF`) token into a port $p \in Out(s)$, where $Out(s) = \{p \in P | (s, p) \in D\}$. CAPIO-CL currently supports three commit rules:

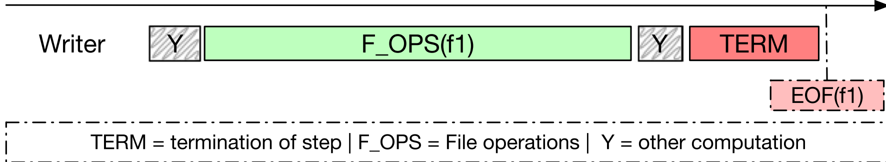

Fig. 1 A diagram of the interactions of the CoT semantic. Under the CoT semantic, several operations on a given file can occur, such as read, write open, stat, and close (as shown by the F_OPS(f1) green block). The file is committed only as soon as the producer step terminates (as shown by the TERM red block)

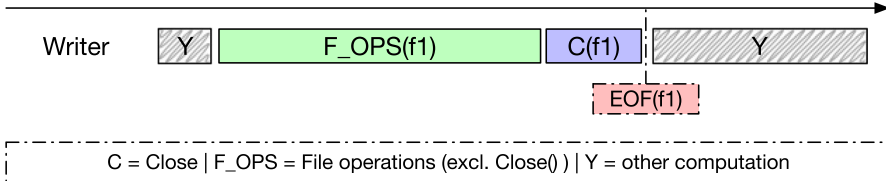

Fig. 2 A diagram of the interactions of the CoC semantic. Under the CoC semantic, a file can experience several different operations on itself, such as create, read, write, and seek (as shown in the F_OPS(f1) green box). However, it will be considered committed only as soon as the close() operation is invoked on that file, as shown in the C(f1) blue box

- *CoT: Commit on Termination (Fig. 1)*: given a step $s \in S$ and a file $f : t_f \in T(p)$ for $p \in Out(s)$, $CoT(s,f) = 1 \iff s$ is terminated. This is the traditional behavior of token-pushing workflow models. Upon termination of a workflow step, all produced data files are committed to the file system, becoming ready for consumption by all subsequent steps;
- *CoC: Commit on Close (Fig. 2)*: given a step $s \in S$ and a file $f : t_f \in T(p)$ for $p \in Out(s)$, $CoC(s,f) = 1 \iff s$ has closed file f. This behavior allows subsequent steps to initiate reading a file as soon as s invokes a `close` operation on f, signaling that all I/O operations on f are completed;
- *CoF: Commit on File (Fig. 3)*: given two steps $s_i, s_j \in S$, a file $f_i : t_{f_i} \in T(p_i)$ for $p_i \in Out(s_i)$, and a file $f_j : t_{f_j} \in T(p_j)$ for $p_j \in Out(s_j)$, $CoF(s_i,f_i) = CR(s_j,f_j)$. This behavior considers a file committed when another file has been committed, potentially by a different step, introducing temporal dependencies among commit rules.

The CoF rule proves beneficial when the number of `open` and `close` operations for a given file f_i is unknown beforehand, but users are aware that the I/O operations on f_i can be considered concluded if another file f_j has been committed according to any rule $CR(s_j,f_j)$. The CoF rule alone could lead to ill-defined I/O graphs whenever s_j, directly or indirectly, depends on s_i. Therefore, CoF always implies CoT to preserve correctness; that is, CoF has a fallback to CoT in case of unexpected or wrong behavior in the workflow step.

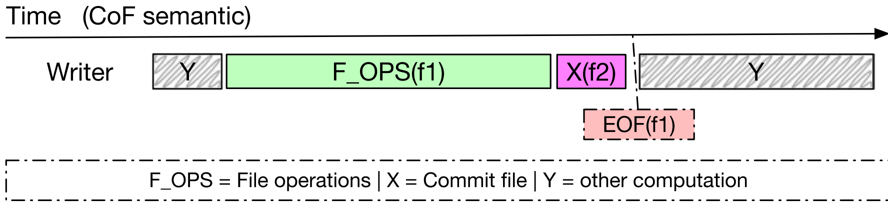

Fig. 3 A diagram of the interactions of the CoF semantic. Under the CoF semantic, a file can experience several different operations on itself, such as create, read, write, seek, and close (as shown in the F_OPS(f1) green box). However, it will be considered committed only as soon as another file is considered committed, as shown in the X(f2) magenta box

On the other hand, firing rules define when a step can begin consuming token data. Practically, a step can start to consume a stream immediately when it starts, after a specific event, or when the stream ends with the `EOF` token. CAPIO-CL supports two distinct firing rules:

- *FoC: Fire on Commit (Fig. 4)*: given a step $s \in S$ and a file $f : t_f \in T(p)$ for $p \in Out(s)$, $FoC(s,f,CR(s,f)) = CR(s,f)$. In other words, whenever the commit rule is satisfied for a given file, the file is unequivocally ready to be consumed: the commit rule implies the firing rule for the entire file;
- *FnU: Fire no Update (Fig. 5)*: given a step $s \in S$ and a file $f : t_f \in T(p)$ for $p \in Out(s)$, $FnU(s,f,CR(s,f)) = 1 \iff CR(s,f) = 1$ or f exists, is not empty and written parts of f will not be updated in the future. In other words, since s produces f data without ever modifying it, all portions of f already written by s can be consumed immediately by any subsequent step without waiting for f to be committed.

Supporting event-based firing rules that can react to application-specific and user-defined events is an interesting research direction to improve the CAPIO-CL expressive power. With this background, the value of the token $t_f \in T(p_i)$, with $p_i \in In(s_j) \cap Out(s_i)$, can be written as $t_f = FR(s_j,f,CR(s_i,f))$, thus propagating

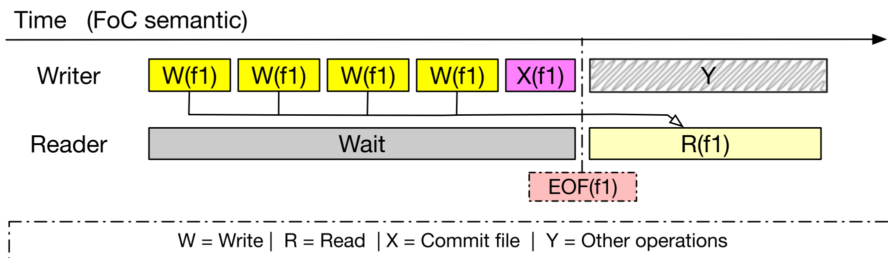

Fig. 4 A diagram of the interactions of the FoC semantic. Under the FoC, a given file can be subjected to different operations by the producer step (in the picture, depicted as Write operations on the yellow boxes). The consumer step can only consume it as soon as the file is committed (as shown by the magenta X(f1) box)

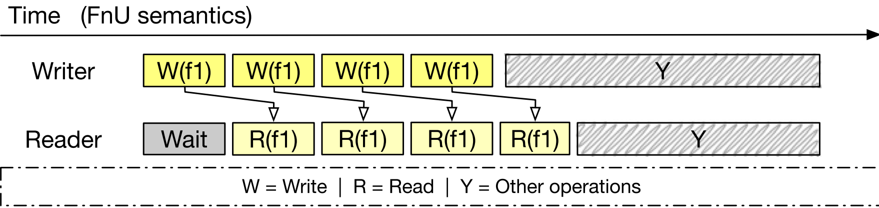

Fig. 5 A diagram of the interactions of the FnU semantic. Under the FnU semantic, the data that has been written to the file (as shown by the yellow W(f1) boxes, can be consumed instantly by the consumer step (as shown by the light yellow R(f1) boxes

token-pushing semantics to step firing semantics in the workflow execution model described in Sect. 2. Note that the combination of commit and firing rules enables streaming semantics in token-based workflows, as the creation and consumption of tokens are now related to single files instead of depending on steps' termination. In the context of the producer-consumer paradigm, a file can be seen as a data stream by combining CoC and FnU rules. Conversely, the default, more conservative behavior of token-pushing semantics can be represented by the joint CoT and FoC rules. Let us consider two real scenarios to exemplify the expressive power of the CAPIO-CL streaming semantics. Suppose a step s_i attempts to open an input file f that has not yet been created. In this case, the runtime engine should halt the workflow execution until s_i creates f through an `open` operation. This condition is met by any combination of the $CR(f)$ and $FR(f)$ mentioned above, as long as the file is listed inside the CAPIO-CL configuration file (see Sect. 4). Indeed, the token-pushing semantics described in Sect. 2 ensure that a step execution starts only when it reaches a fireable state, i.e., when the amount of tokens in its input ports satisfies the associated firing rule. In the second scenario, a consumer step tries to read a portion of a file f that has not been written yet by the producer step s_i. This behavior is nuanced as a `read` operation may return fewer data elements than the amount initially requested or even zero. In this case, the runtime library should pause the consumer step until one of the following two conditions is met. First, the requested data is entirely produced, and the `read` operation returns the total number of bytes requested. In this case, data can be simply propagated to the consumer, and the execution can proceed. Second, all producer steps trigger $CR(f)$, i.e., they either close the file (CoC rule), terminate (CoT rule), or commit a related file (CoF rule). After that, the `read` will return either the whole content of file f or the `EOF` token, signaling to the consumer step that the file has been fully processed.

3.2 Home-Node Policies

CAPIO-CL can also describe proactive data staging strategies. CAPIO-CL leverages the concept of *home-node*, borrowed from page-based software Distributed Shared-Memory implementations [12]. In CAPIO-CL, a *home-node* is a specific 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. Formally, given a distributed workflow $DW = (S, P, D, L, \mathcal{M})$, the home-node $H(f) \in L$ is the location where a file f is staged after it has been produced by a step $s \in S$. The mapping of files onto home-nodes can be customized through the *home-node policy* feature of CAPIO-CL, e.g., to maximize the I/O bandwidth in a distributed storage system. Users can define different policies for different files as long as each file has a unique rule. CAPIO-CL currently supports three possible home-node policies:

- The *create* policy maps a file f to the location in which the file has been created. In this setting, if $t_f \in T(p)$, with $p \in Out(s)$, then $H(f) \in \mathcal{M}(s)$. For example, if $\mathcal{M}(s) = \{l_1, l_2\}$, then if the file f is created on l_1 through an `open` operation, then $H(f) = l_1$. If later also l_2 performs an `open` operation on f, the home-node does not change.
- The *hashing* policy maps a file f onto a home-node according to a hash function that considers the file path and the cardinality of the set L. For example, even if the file f is created on location $l_1 \in \mathcal{M}(s)$, the file is assigned a home node $H(f) = hash(f, |L|) \in L$. Therefore, the runtime library supporting the language will move the file to the assigned location. Note that this policy does not guarantee that $H(f) \in \mathcal{M}(s)$.
- The *manual* policy allows users to manually define the home-node node for each individual or group of files handled by the distributed workflow. Note that a correct mapping requires $H(f)$ to be defined for all file tokens that appear in a workflow execution. To avoid ill-defined mapping functions, the *manual* policy always falls back to the *create* policy when $H(f)$ is not explicitly defined. Through the *manual* policy, it is possible also to express the *create* policy, though it would require more effort from a user standpoint, as all possible files produced by the workflow need to be enumerated.

The *create* policy has been chosen as the default option as it represents the default behavior in distributed workflow systems. In addition, this policy only requires every $l \in \mathcal{M}(s)$ to be active during s execution, without imposing any constraint on $L \setminus \mathcal{M}(s)$. With the information provided by the home-node policy, the CAPIO middleware may statically know the file-to-node mapping and thus retrieve the node where the producer (or consumer) process is executing at runtime. The home-node policy section is meant to work together with the information provided by the input_stream and output_stream options to optimize data transfers and preventive file movement. It is important to note that users are usually unaware of which nodes the applications will be executed on. This is particularly true in a cloud or HCP environment, and due to this, CAPIO-CL needs to abstract and use logical names for the applications and logical IDs for the application threads. By using them, 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 without the need of tailoring it to a specific cluster.

4 Language Syntax

CAPIO-CL uses JSON (JavaScript Object Notation) syntax to express the coordination semantics. JSON is not tied to any particular programming language or platform and is widely supported across various programming languages (such as 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. Moreover, compared with a custom syntax, using JSON lowers the learning curve for new users. To foster cross-system portability, CAPIO-CL interprets all file paths as relative to a single root mount point, referred to as `CAPIO_DIR`. All paths outside the `CAPIO_DIR` should be ignored by the runtime library, even if expressed as absolute paths. This strategy decouples the CAPIO-CL specification from the peculiarities of the local file system. Since CAPIO-CL deals with I/O objects, i.e., files and directories, it supports *wildcards*, special characters used to specify unknown characters in a text. *Wildcards* can be used in all values in which a file or a directory name is expected to avoid enumerating all the files/directories an application might produce or read (e.g., file*.dat). Currently, the language handles two wildcards: '`*`' that matches any sequence of characters of length ≥ 0, and '`?`' that matches a single character.

A valid CAPIO-CL file comprises six sections: *Workflow Name*, *Alias*, *IO_Graph*, *Permanent*, *Exclude* and *Home-node policy*. We shall now explain all of them and provide an incremental example of the CAPIO-CL language through the sample workflow depicted in Fig. 6.

4.1 Workflow Name

The workflow name section is identified by the keyword `name` (Listing 1). The name is used as an identifier for the current application workflow. This is needed to distinguish different application workflows running on the same machine.

Listing 1: The workflow name.

```
1  {
2    "name" :"my_workflow",
3    ...
4  }
```

4.2 Aliases

The alias section is identified by the keyword `aliases` (Listing 2). Aliases are used to group a set of files or directories under a convenient name, reducing the

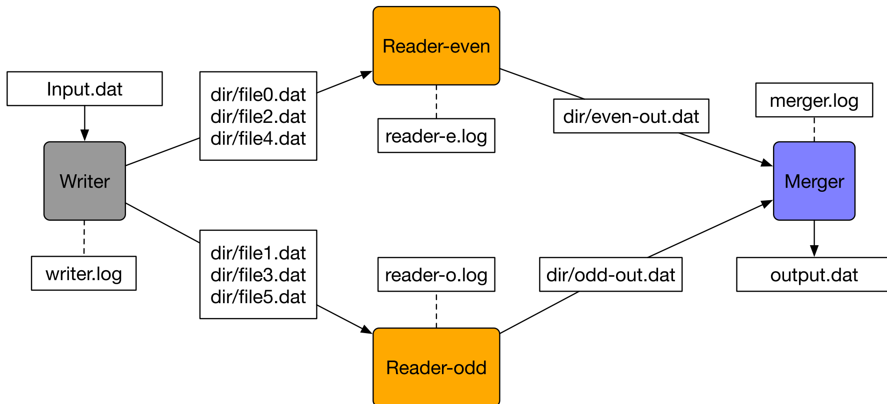

Fig. 6 A sample workflow used to create an incremental example of the CAPIO-CL language. Initially, the input.dat file is accessed by the Writer module, where it undergoes processing and is divided among multiple files numbered and split between even and odd. These files are then forwarded to two subsequent stages for further refinement before ultimately being merged into an output file by a final stage. While logs are generated during this process, they are disregarded by the CAPIO implementation

verbosity of enumerating files an application can consume or produce. It is a vector of objects composed of the following items:

- `group_name`: This keyword identifies the alias name.
- `files`: This keyword identifies an array of strings representing file names.

Listing 2: Example of alias section.

```
1  {
2    "name" :"my_workflow",
3    "aliases" :[
4      { "group_name" :"group-even", "files" :["dir/file0.dat", "dir/file2.dat", "dir/file4.
           dat"] },
5      { "group_name" :"group-odd", "files" :["dir/file1.dat", "dir/file3.dat", "dir/file5.dat
           "] }
6    ],
7    ...
8  }
```

Aliases can improve the readability of the CAPIO-CL file by simplifying the definition of the IO_Graph and avoiding ambiguities when using wildcards.

4.3 Permanent and Exclude

The permanent section is identified by the keyword `permanent` (Listing 3). It is used to specify which files must be kept in the permanent storage at the end of the workflow execution. Its value is an array of file names (whose values can also be aliases). The permanent section only guarantees that the enumerated files in this section will be kept

on the file system at the end of the workflow. Still, it does not provide any guarantees whatsoever about the fact that other files could be present on the file system. The exclude section is identified by the keyword `exclude` (Listing 3). It is used to specify those files that will not be handled by CAPIO even if they will be created inside the `CAPIO_DIR`. Its value is an array of file names (whose values can also be aliases).

Listing 3: Example of exclude section.

```
1  {
2     "name" :"my_workflow",
3     "aliases" :[
4        { "group_name" :"group-even", "files" :["dir/file0.dat", "dir/file2.dat", "dir/file4
             .dat"] },
5        { "group_name" :"group-odd", "files" :["dir/file1.dat", "dir/file3.dat", "dir/file5.
             dat"] }
6     ],
7     "permanent" :["output.dat"],
8     "exclude" :["source.dat", "*.log"],
9     ...
10 }
```

4.4 IO_Graph

The IO_Graph section is identified by the keyword `IO_Graph` (Listing 4). It defines the producer and consumer dependencies between the application modules comprising the workflow, specifying respectively the output and input streams. It is composed of an array of objects, each one composed of the following items:

- `name` (**mandatory**): This keyword identifies the application's name.
- `input_stream`: This keyword identifies the input files and directories the application module is expected to read. Wildcards might be used in this field. Its value is a vector of strings, and it is optional.
- `output_stream`: This keyword identifies the files and directories the application module is expected to produce. Wildcards might be used in this field. Its value is a vector of strings, and it is optional.
- `streaming`: This keyword is used to specify the streaming semantics, i.e., the *commit* and *firing rules* for a given file or directory that the application might produce or consume. It is optional, and if it is omitted, then the *default* streaming semantics (as described in Sect. 3) is applied to all the files inside the input stream and output stream.

Listing 4: Example of IO_Graph section.

```
1  {
2     "name" :"my_workflow",
3     "aliases" :[
4        { "group_name" :"group-even", "files" :["dir/file0.dat", "dir/file2.dat", "dir/file4
           .dat"] },
5        { "group_name" :"group-odd", "files" :["dir/file1.dat", "dir/file3.dat", "dir/file5.
           dat"] }
6     ],
7     "permanent" :["output.dat"],
8     "exclude" :["input.dat", "*.log"],
9     "IO_Graph": [
10      {
11        "name": "writer",
12        "input_stream" :["input.dat"],
13        "output_stream": ["group-even","group-odd","writer.log","dir"],
14        "streaming": [
15           { "name": ["group-even" ],"committed": "on_termination", "mode": "update" },
16           { "name": ["group-odd" ],"committed": "on_close", "mode": "update" },
17           { "dirname": ["dir" ],  "committed": "n_files:6", "mode": "no_update"} ]
18      },{
19        "name":"reader-even",
20        "input_stream": ["group-even"],
21        "output_stream": ["even-out.dat", "reader-e.log"],
22        "streaming" :[{"name":["even-out.dat"], "committed":"on_close", "mode":"update"}]
23      },{
24        "name":"reader-odd",
25        "input_stream": ["group-odd"],
26        "output-stream": ["odd-out.dat", "reader-o.log"],
27        "streaming" :[{"name":["odd-out.dat"], "committed":"on_file", "file_deps":"even-out.
           dat", "mode":"no_update"}]
28      },{
29        "name": "merger",
30        "input_stream": ["odd-out.dat", "even-out.dat"],
31        "output_stream": ["output.dat", "merger.log"]
32      }
33    ],
34    ...
35 }
```

The streaming section is an array of objects, and each object may define the following attributes:

- `name` / `dirname` (**mandatory**): This keyword is used to specify the files/directories to which the rule applies. If `dirname` is specified instead of name, the rule will apply to the directory and all of its content (unless a more specific rule for a file inside the `dirname` is specified. In that case, the most specific rule prevails over the less specific rule). In both cases, the value of this keyword is an array of names.
- `committed` (**mandatory**): This keyword is used to specify the *commit rule* for a given file or directory. Depending on the case, whether a file or a directory is involved, two different options are available:
  - In case a `file` is involved, then its value can be either `on_close` if the semantics is `CoC`, `on_termination` if the semantic is `CoT`, or `on_file` if the commit rule is `CoF`. In the event, the value is `on_close`, the modifier `:N` is allowed. This modifier considers the file committed when the

close() operation has been invoked *N* times on the target file. By default, writing `"committed":"on_close"`, and `"committed":"on_close:1"` is to be considered equal.
  - In case a `directory` is involved, its value can be either `on_termination` or `on_file`. The commit rule can also be `n_files:N` (defined only for directories), which allows to set a directory to be committed after *N* files have been created inside that directory.

  In both cases, 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 `name` and `dirname`, respectively. Its value can be either `update` if the semantics is `FoC` or `no_update` if the semantics is `FnU`.

4.5 Home-Node Policy

This optional section is identified by the `home_node_policy` keyword. Users may declare the use of one or more of these policies using the keyword create, hashing and (or) manual, specifying a disjoint set of files and directories for each of them (Listing 5). The `create` policy is the default policy for CAPIO-CL. This means that it 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`. Consequently, if some files are not specified in the home-node policy section, they will be treated with the create policy. For the `hashing` policy, a solid and collision-resistant hashing function needs to be provided by the runtime implementation 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`. Finally, when the `home-node-policy` keyword, defined for a subset of files, equals to `manual`, the users can explicitly set the reference node for each individual or group of files by choosing as *home-node* the node where a specific application module is running. This is accomplished by specifying the workflow module's name used in the `IO_Graph` section and its logical id[1]. The syntax is the following: `app_node:id`. 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 too many files are 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

[1] The *logical id* is a unique identifier (integer type) that identifies a process of an application module. If the application is single-process, it is redundant and not required.

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.

Listing 5: Complete example of the workflow depicted in Fig. 6.

```
1  {
2     "name" :"my_workflow",
3     "aliases" :[
4        { "group_name" :"group-even", "files" :["dir/file0.dat", "dir/file2.dat", "dir/file4
            .dat"] },
5        { "group_name" :"group-odd", "files" :["dir/file1.dat", "dir/file3.dat", "dir/file5.
            dat"] }
6     ],
7     "permanent" :["output.dat"],
8     "exclude" :["source.dat", "*.tmp"],
9     "IO_Graph": [
10    {
11       "name": "writer",
12       "input_stream" :["input.dat"],
13       "output_stream": ["group-even","group-odd","logs.tmp","dir"],
14       "streaming": [
15          { "name": ["group-even" ],"committed": "on_termination", "mode": "update" },
16          { "name": ["group-odd" ],"committed": "on_close", "mode": "update" },
17          { "dirname": ["dir" ],  "committed": "n_files:6", "mode": "no_update"} ]
18     },{
19       "name":"reader-even",
20       "input_stream": ["group-even"],
21       "output-stream": ["even-out.dat"],
22       "streaming" :[{"name":["even-out.dat"], "committed":"on_close", "mode":"update"}]
23     },{
24       "name":"reader-odd",
25       "input_stream": ["group-odd"],
26       "output-stream": ["odd-out.dat"],
27       "streaming" :[{"name":["odd-out.dat"], "committed":"on_file:even-out.dat", "mode":"
            no_update"}]
28     },{
29       "name": "merger",
30       "input_stream": ["odd-out.dat", "even-out.dat"],
31       "output_stream" :["output.dat"]
32     }],
33    "home_node_policy": {
34       "create": ["dir/file0.dat", "dir/file1.dat"],
35       "hashing": ["dir/file2.dat", "dir/file3.dat"],
36       "manual": [
37          {"name": ["dir/file4.dat", "dir/even-out.dat" ],"app_node": "Reader-even:0"},
38          {"name": ["dir/file5.dat", "dir/even-out.dat" ],"app_node": "Reader-odd:0" }
39       ]
40    }
41 }
```

5 Evaluation

The effectiveness of the CAPIO-CL language can be evaluated considering two main aspects: its expressiveness, intended as the ability to define and combine advanced semantic behavior of I/O patterns easily, and the performance improvements that can be achieved by implementing a middleware leveraging the CAPIO-CL language. In this section, we evaluate the language's expressiveness by comparing CAPIO-CL

to another coordination language and then describe the results obtained through the CAPIO middleware on both synthetic benchmarks and an actual workflow. Note that performance measures are strictly related to the implementation, i.e., to the runtime library that translates CAPIO-CL semantics into actual optimization of the I/O plane orchestration. Different implementations may differ in several aspects: the programming language (e.g., Python vs. C++), the target infrastructure (e.g., cloud VMs or Kubernetes vs. queue-based HPC), and the optimization strategies they rely on (e.g., proactive data staging vs. data-locality scheduling). The results reported in this article are based on the reference CAPIO-CL implementation, called CAPIO [7].[2]

5.1 Expressiveness

Listing 6: Wilkins example.

```
1   tasks:
2   - func: producer
3     nprocs: 3
4     outports:
5     - filename: outfile.h5
6       dsets:
7       - name: /group1/grid
8         file: 0
9         memory: 1
10      - name: /group1/particles
11        file: 0
12        memory: 1
13  - func: consumer1
14    nprocs: 5
15    inports:
16    - filename: outfile.h5
17      dsets:
18      - name: /group1/grid
19        file: 0
20        memory: 1
21  - func: consumer2
22    nprocs: 2
23    inports:
24    - filename: outfile.h5
25      dsets:
26      - name: /group1/particles
27        file: 0
28        memory: 1
```

Listing 7: CAPIO-CL example.

```
1   {
2    "name": "producer-consumer",
3    "IO_Graph":[
4     {
5      "name": "Producer",
6      "output_stream": ["/group1/
           particles", "group1/grid"],
7      "streaming": [{
8        "name": ["/group1/particles"],
9        "committed":"on_close"
10      },{
11       "name": ["/group1/grid"],
12       "committed": "on_close"
13       }
14     ],
15    },{
16     "name": "consumer1",
17     "input_stream": ["group1/grid"]
18    },{
19     "name": "consumer2",
20     "input_stream": ["group1/
           particles"]
21    }
22   ],
23   "home_node_policy": {
24      "create": ["/group1/particles"],
25      "hashing": ["group1/grid"],
26      }
27  }
```

Listing 5 is a complete example showing how CAPIO-CL allows for defining a data dependency graph and modeling the I/O behavior of an application annotated with streaming rules and data staging policies, targeting the effective handling of specific files in the producer-consumer life cycle. It means that users can specify

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

the I/O behavior of the application at the file level, combining, if needed, different semantics. It is worth noting that using JSON reduces the users' learning curve and guarantees the possibility of quickly extending the language with further semantic behavior, as it is already planned in future work (Sect. 6).

To provide a comparison with an existing I/O coordination language, we took a listing from the original Wilkins article [13] (Listing 6) and rewrote it using CAPIO-CL (Listing 7). Wilkins and CAPIO-CL are comparable in terms of language verbosity, as both of them adopt a fully declarative approach. In addition, JSON and YAML are interchangeable when no advanced features are used, as in the case of Wilkins and CAPIO-CL. However, CAPIO-CL is capable of encoding more information. For example, home-node policies carry information on the home node of any given file. This feature is a fundamental building block for several advanced performance optimizations, like proactive data transfers and locality-based scheduling. In Listing 7 we added a `home_node_policy` section to notify the runtime library on where the files will be located. It is possible to convert any Wilkins configuration file to CAPIO-CL, and this shows how CAPIO-CL can address the same model without limitation and express various configurations. In detail, all the examples shown in the paper can be converted, but for the sake of brevity, we did not include them in this paper.

5.2 CAPIO Runtime

The CAPIO reference runtime [7] comprises a set of per-node user-space servers implementing distributed data storage and I/O coordination (trough the information provided by the CAPIO-CL configuration file) for a given workflow. CAPIO is implemented in C++ and uses MPI only for server-to-server communications. It supports all the features defined in the CAPIO-CL specification, and can be considered the CAPIO-CL reference implementation. For each node, the user specifies a CAPIO local-node mount point through the `CAPIO_DIR` environment variable. CAPIO captures all the I/O system calls (SCs) executed by the targeted process. If the SC targets a file or directory inside the `CAPIO_DIR` directory, then CAPIO handles the system call directly. Otherwise, it forwards it to the kernel.

The *CAPIO Intercept Library*, based on the `syscall_intercept` library,[3] can be dynamically linked to each workflow step through the `LD_PRELOAD` mechanism. The CAPIO intercept library supports both multi-process and multi-threaded applications. The intercept library communicates with the local CAPIO server through the POSIX shared memory component. If both consumer and producer steps are co-located on the same node, the produced and consumed files are passed between the two steps over shared memory. Otherwise, the files are transferred using MPI. The following commands launch a generic program (in this example *appA*) with CAPIO, where `capioServer.sh` runs the CAPIO server passing the CAPIO-CL configuration file `Config.json`.

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

Table 1 Execution time of synthetic benchmarks with POSIX and with CAPIO different file sizes and different streaming semantics. *Lustre* is the execution without CAPIO, *FoC* is the execution with CAPIO and with firing rule *Fire on Commit* and *FnU* is the execution with CAPIO with firing rule *Fire no Update*. For all the CAPIO results, the commit rule is *Commit on Close*. The granularity of read and write operations is 1MB

Test case	10 files of 1GB			100 files of 1GB			1 file of 100GB		
	Lustre	CoC FoC	CoC FnU	Lustre	CoC FoC	CoC FnU	Lustre	CoC FoC	CoC FnU
1-to-1	26.43	21.04	9.93	–	–	–	–	–	–
1-to-N	–	–	–	74.12	36.01	37	102	45.44	35.85
N-to-1	–	–	–	112.91	70.19	69.4	–	–	–

```bash
$ ./capioServer.sh Config.json &
$ LD_PRELOAD=libcapio_posix.so CAPIO_DIR="/capio_dir"
    CAPIO_APP_NAME="appA" ./appA
```

5.3 Performance

The CAPIO reference runtime has been proven to successfully improve performance of data-oriented workflows by transparently injecting streaming capabilities into them [7]. In detail, CAPIO has been evaluated on top of synthetic, file-based benchmarks and real-world use cases. The synthetic benchmarks were designed to replicate three common I/O patterns:

- *1-to-1*: one producer and one consumer steps (with 10 files of 1 GB each);
- *1-to-N*: one producer and 20 consumer steps (with 100 files of 1 GB each and with a single 100GB file);
- *N-to-1*: 20 producers and one consumer steps (with 100 files of 1 GB each).

All experiments were conducted on the GALILEO100 supercomputer,[4] which has compute nodes with 2 Intel Cascade Lake 8260 CPUs (24 cores, 2.4 GHz each) and 384GB RAM and a Lustre parallel shared file system. A subset of the experimental results is presented in Table 1 for the reader's convenience. The table displays the computing time of the benchmarks under different conditions: without the CAPIO runtime, with CAPIO using the CoC and FoC rules, and with CAPIO utilizing the

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

joint CoC and FnU rules, enabling full streaming capabilities. The CoC rule is the most frequently applied in real-world scenarios, as high-performance scientific codes often use files as persistent memory buffers to store output data, deallocating (closing) them only at the end of the computation. In detail, the reported test cases are: the *1-to-1* benchmark on 2 nodes with a dataset of 10 files of 1GB; the *1-to-N* benchmark on 20 nodes with a dataset of 100 files of 1GB; the *N-to-1* benchmark on 20 nodes with a dataset of 100 files of 1GB. It is important to note that the CAPIO runtime consistently improves benchmark execution across all cases. The lack of difference in execution time between the *1-to-N* and *N-to-1* benchmarks can be attributed to how the producer and consumer steps handle file production and consumption. In the *1-to-N* benchmark, the producer step generates files sequentially before sending them to the consumer steps. Even with streaming enabled (i.e., using the CoC-FnU semantics), the application's I/O pattern, combined with the overhead introduced by CAPIO's network communication, results in no significant difference between the two semantics. The *N-to-1* benchmark exhibits similar behavior, as the consumer step processes all files sequentially. However, when the *1-to-N* test is executed with a single 100GB sparse file whose data are written after seek operations, the benefits of CAPIO-CL's CoC-FnU semantics become more evident, significantly improving execution times. It is important to emphasize that, despite no significant difference in these specific synthetic benchmark configurations for the two semantics, the overall advantage of using CAPIO and CAPIO-CL over not using CAPIO at all ranges from approximately 30% to 50%. All previous experiments were conducted without explicitly configuring a home-node policy, instead relying on the default *create* strategy. To demonstrate the potential of home-node policies, we compared two different strategies for the workflow shown in Fig. 6: the *create* policy and the *manual* policy, configured as shown in Listing 6. The test for the home-node policies was conducted on the HPC4AI Cluster,[5] which consists of 68 compute nodes, each equipped with two Intel(R) Xeon(R) E5-2697 v4 processors (18 cores, 2.3GHz each) and 128GB RAM, connected via an OPA 100Gbit/s network and utilizing a BeeGFS parallel shared file system. With the *create* policy, the sample workflow takes 34 s to execute. In contrast, with the *manual* policy, where pre-moving data optimization is enabled, it completes in 27 seconds, resulting in a runtime reduction of approximately 20 As this is a preliminary test on a prototype implementation, no statistical analysis has been conducted yet. However, such analysis is planned for the future development of the CAPIO middleware.

5.4 1000 Genome Workflow

Martinelli et al. [7] have demonstrated how a real-life workflow can benefit from a CAPIO-CL middleware implementation in its performance. The 1000 Genome workflow [22], whose structure is represented in Fig. 7, comprises five steps. Using a data-parallel approach, the *individuals* step can be split into multiple independent

[5] HPC4AI Cluster: https://hpc4ai.unito.it

instances. In detail, each instance analyzes a partition of the input file and generates a directory containing 2504 temporary small files (1 - 15KB with 16 instances). The *individuals_merge* is a *reduce* operation. It reads all the files in the directories produced by the previous step and concatenates them, producing a single directory with 2504 files. The *sifting* step can be executed in parallel with the *individuals* and *individual_merge* steps. The last two steps, *mutation_overlap* and *frequency*, are composed of independent instances that process individuals from different populations in parallel, starting from the data produced by *individuals_merge* and *sifting* steps. Listing 8 shows the configuration required for each workflow step.

Listing 8: The 1000-Genome workflow CAPIO-CL configuration file

```
1   {
2     "name": "1000_genome",
3     "IO_Graph": [
4       {
5         "name": "individuals",
6         "input_stream": ["data/20130502/columns.txt","data/20130502/ALL.chr1.250000.vcf"],
7         "output_stream": ["chr1n-1-250000/*"],
8         "streaming": [{
9             "name": ["chr1n-1-250000/*"],
10            "committed": "on_close",
11            "mode": "no_update"
12          }]
13      },{
14        "name": "individuals_merge",
15        "input_stream": ["chr1n-1-250000/*"],
16        "output_stream": ["chr1n/*"],
17        "streaming": [{
18            "dirname": ["chr1n/*"],
19            "committed": "on_close",
20            "mode": "no_update",
21            "n_files": 2504
22          }]
23      },{
24        "name": "sifting",
25        "input_stream": ["data/20130502/sifting/*"],
26        "output_stream": ["sifted.SIFT.chr1.txt"],
27        "streaming": [{
28            "name": ["sifted.SIFT.chr1.txt"],
29            "committed": "on_close",
30            "mode": "no_update"
31          }]
32      },{
33        "name": "mutation_overlap",
34        "input_stream": ["data/populations/ALL","data/20130502/columns.txt"],
35        "output_stream": [],
36        "streaming": []
37      },{
38        "name": "frequency",
39        "input_stream": ["data/populations/ALL","data/20130502/columns.txt"],
40        "output_stream": [],
41        "streaming": []
42      }
43    ]
44  }
```

Table 2 Execution time (seconds) of the 1000 Genome workflow with an increasing number of replicas of the *individuals* steps

Configuration	File system	CAPIO
4 Individuals	730	662
8 Individuals	427	338
16 Individuals	293	180

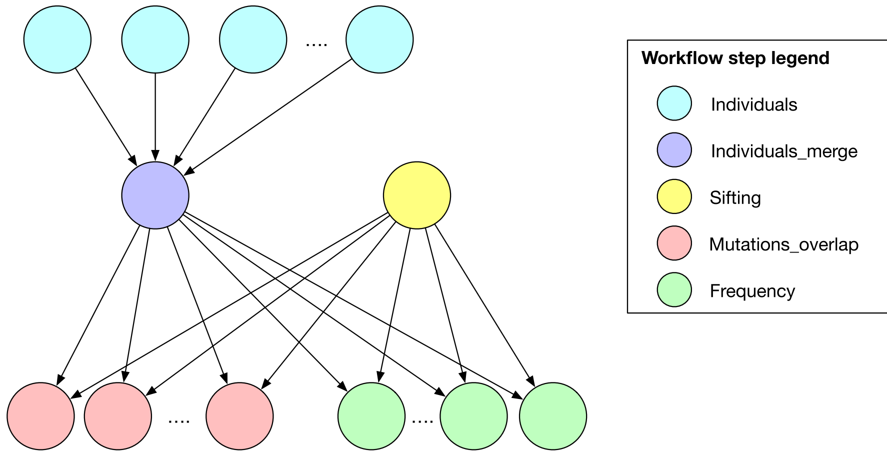

Fig. 7 1000-genome workflow structure [22]

The *individuals* and *individuals_merge* involve the vast majority of I/O operations of the entire workflow. The high number of small intermediate files causes a performance bottleneck when the workflow is scheduled on an HPC cluster mounting a parallel shared file system [14]. The CAPIO runtime mitigates this issue by replacing the file system with in-memory file handling and network-based communications between the two steps. In addition, the *individuals* step supports CoC-FnU synchronization semantics, allowing CAPIO to inject transparent stream processing logic effectively and increase the computation-communication overlap. Table 2 reports the results of 1000 Genome workflow running on top of 8, 12, and 20 nodes of GALILEO100 supercomputer. All workflow steps are deployed on a different node, meaning no allocated node executes two different workflow steps. All workflow steps are deployed as a single process, except for the *individuals* step, which is replicated. In the 8-node configuration, 4 nodes execute the *individuals* step, while the remaining nodes each execute a different workflow step. The same applies to the 12 node configuration, where 8 nodes execute the *individuals* step, and the remaining nodes handle the other workflow steps. Similarly, in the 20 node configuration, 16 nodes execute the *individuals* step. The speedup achieved by varying the number of *individuals* instances aligns with the synthetic benchmark, showing an improvement ranging from 10% to approximately 40%.

6 Conclusion and Future Work

High-performance computing workloads are rapidly shifting from monolithic applications to workflows that integrate both co-engineered and legacy components. These components communicate through a portable, file-based interface, using the file system as a communication medium. Efficiently coordinating I/O behavior across different workflow modules, increasing the overlap between computation and communication, optimizing data staging, and enabling in-situ and in-transit optimizations are essential for performance.

In this paper, we introduce CAPIO-CL, a novel I/O coordination language designed to enhance scientific workflows by incorporating I/O information without requiring modifications or patches to the application code. We defined multiple synchronization semantics to model file dependencies among workflow modules, along with data staging policies that enable preemptive data movement. By leveraging these rules, I/O streaming capabilities can be transparently injected into file-based workflows, utilizing user-provided directives and hints through a CAPIO-CL JSON-based file. CAPIO-CL is flexible enough to map onto other existing languages with minimal effort and has already demonstrated substantial performance improvements across various application scenarios. Additionally, by offering a formal definition of CAPIO-CL's semantics, we lay the groundwork for formal model checking and system verification of file-based workflows, ensuring correctness and efficiency in scientific computing environments.

We are actively improving the CAPIO-CL language to extend its capabilities and application areas. For example, an interesting new perspective emerges from the model described in Sect. 3. Consider two workflows running in parallel on the same compute node (or, by extension, in an environment with shared resources such as the file system), both being file-based. In this scenario, all files would typically be accessible on the file system, and a step from one workflow might be able to read data from another. While file permissions usually prevent this from occurring, system administrators[6] can still perform read/write operations on these files. This presents a significant security risk, mainly if one of the workflows processes sensitive data, as specific applications require complete confidentiality for workflow-generated subproducts. It is crucial to ensure that the flow of information does not lead to unintended leaks. Several studies have explored this issue [15, 16]. With CAPIO-CL, in combination with encryption technologies such as SGX [17], we can leverage the `input_stream` and `output_stream` sections to automatically and transparently introduce an additional layer of security and confidentiality in workflow execution, ensuring that no information leakage occurs at any level [18]. To achieve this, a new policy will be introduced: one that restricts access strictly to files explicitly listed in the `input_stream` section with read-only permissions and files in the `output_stream` section with both read and write permissions. Finally, we are integrating CAPIO-CL with the StreamFlow [19] and DagOn* [20, 21] workflow

[6] In some cases, not only system administrators but also other applications, often due to configuration issues, which, in our experience, are more common than one might expect.

management systems (WMSs), enabling users to design the I/O orchestration plane for existing, production-ready scientific workflows.

Author contributions The contribution is as follows: - A.R.M., M.T. and M.A. developed the core ideas of the language - M.E.S. and I.C. Provided formal definitions as well as helped with the development of the language - B.C. oversaw the manuscript creation together with M.A. All authors equally contributed in writing the manuscript and discussing the ideas contained in the article.

Funding Open access funding provided by Università degli Studi di Torino within the CRUI-CARE Agreement. 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.

Data Availability No datasets were generated or analysed during the current study.

Competing interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., Wolf, M.: Managing variability in the io performance of petascale storage systems. In: SC '10: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis, pp. 1–12 (2010). https://doi.org/10.1109/SC.2010.32
2. Silva, R.F., Callaghan, S., Do, T.M.A., Papadimitriou, G., Deelman, E.: Measuring the impact of burst buffers on data-intensive scientific workflows. Future Gener. Comput. Syst. **101**, 208–220 (2019). https://doi.org/10.1016/J.FUTURE.2019.06.016
3. Atkinson, M.P., Liew, C.S., Galea, M., Martin, P., Krause, A., Mouat, A., Corcho, Ó., Snelling, D.: Data-intensive architecture for scientific knowledge discovery. Distrib. Parallel Datab. **30**(5–6), 307–324 (2012). https://doi.org/10.1007/S10619-012-7105-3
4. Filgueira, R., Krause, A., Atkinson, M., Klampanos, I., Spinuso, A., Sanchez-Exposito, S.: dispel4py: An agile framework for data-intensive escience. In: 2015 IEEE 11th international conference on e-science, pp. 454–464 (2015). https://doi.org/10.1109/eScience.2015.40
5. Bent, J., Grider, G., Kettering, B., Manzanares, A., McClelland, M., Torres, A., Torrez, A.: Storage challenges at Los Alamos National Lab. In: IEEE 28th symposium on mass storage systems and technologies, MSST 2012, April 16-20, 2012, asilomar conference grounds, Pacific Grove, CA, USA, pp. 1–5. IEEE computer society, New York, NY (2012). https://doi.org/10.1109/MSST.2012.6232376
6. Bez, J.L., Byna, S., Ibrahim, S.: I/O access patterns in HPC applications: A 360-degree survey. In: ACM Computing Surveys **56**(2) (2023) https://doi.org/10.1145/3611007

7. Martinelli, A.R., Torquati, M., Aldinucci, M., Colonnelli, I., Cantalupo, B.: 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). IEEE, Goa, India (2023). doi: https://doi.org/10.1109/HiPC58850.2023.00031
8. Colonnelli, I.: Workflow models for heterogeneous distributed systems. In: Bena, N., Martino, B.D., Maratea, A., Sperduti, A., Nardo, E.D., Ciaramella, A., Montella, R., Ardagna, C.A. (eds.) Proceedings of the 2nd Italian conference on big data and data science (ITADATA 2023), Naples, Italy, September 11-13, 2023. CEUR Workshop Proceedings, vol. 3606. CEUR-WS.org, Aachen, Germany (2023)
9. Brock, J.D.: A formal model of non-determinate dataflow computation. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (1983)
10. Jensen, K.: Coloured petri nets: A high level language for system design and analysis. In: Rozenberg, G. (ed.) Advances in Petri Nets 1990 [10th international conference on applications and theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]. Lecture notes in computer science, vol. 483, pp. 342–416. Springer, Berlin, Heidelberg (1989). https://doi.org/10.1007/3-540-53863-1_31
11. Kavi, K.M., Buckles, B.P., Bhat, U.N.: A formal definition of data flow graph models. IEEE Trans. Comput. **35**(11), 940–948 (1986). https://doi.org/10.1109/TC.1986.1676696
12. Carter, J.B., Bennett, J.K., Zwaenepoel, W.: Implementation and performance of Munin. SIGOPS Oper. Syst. Rev. **25**(5), 152–164 (1991). https://doi.org/10.1145/121133.121159
13. Yildiz, O., Morozov, D., Nigmetov, A., Nicolae, B., Peterka, T.: Wilkins: HPC in situ workflows made easy (2024)
14. Bent, J., Grider, G., Kettering, B., Manzanares, A., McClelland, M., Torres, A., Torrez, A.: Storage challenges at los alamos national lab. In: 2012 IEEE 28th symposium on mass storage systems and technologies (MSST), pp. 1–5 (2012). https://doi.org/10.1109/MSST.2012.6232376
15. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976). https://doi.org/10.1145/360051.360056
16. Bauereiß, Thomas, Hutter, Dieter: Information flow control for workflow management systems. Inform. Technol. **56**(6), 294–299 (2014). https://doi.org/10.1515/itit-2014-1055
17. Costan, V., Devadas, S.: Intel SGX explained. IACR Cryptol. ePrint Arch., 86 (2016)
18. Brescia, L., Aldinucci, M.: Secure generic remote workflow execution with TEEs. In: Colonnelli, I., Casanova, H., Montella, R. (eds.) Proceedings of the 2nd workshop on workflows in distributed environments, WiDE 2024, Athens, Greece, 22 April 2024, pp. 8–13. ACM, New York, NY, USA (2024). https://doi.org/10.1145/3642978.3652834
19. Colonnelli, I., Cantalupo, B., Merelli, I., Aldinucci, M.: StreamFlow: cross-breeding cloud with HPC. IEEE Trans. Emerg. Top. Comput. **9**(4), 1723–1737 (2021). https://doi.org/10.1109/TETC.2020.3019202
20. Montella, R., Di Luccio, D., Kosta, S.: DagOn*: Executing direct acyclic graphs as parallel jobs on anything. In: 2018 IEEE/ACM workflows in support of large-scale science (WORKS), pp. 64–73 (2018). https://doi.org/10.1109/WORKS.2018.00012
21. Perrotta, S., De Vita, C.G., Mellone, G., Santimaria, M.E., Salvi, G., Lapegna, M., Torquati, M., Ciaramella, A.: Extending a scientific workflow engine with streaming I/O capabilities: DAGonStar and CAPIO. In: 1st workshop on high-performance eScience (HiPES), Co-located with the Euro-Par 2024 conference, Madrid, Spain (2025)
22. Using simple PID-inspired controllers for online resilient resource management of distributed scientific workflows Future Generation Computer Systems 95, 615–628 (2019). https://doi.org/10.1016/j.future.2019.01.015

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.