TYPE Hypothesis and Theory
PUBLISHED 11 August 2025
DOI 10.3389/fhpcp.2025.1393936

OPEN ACCESS

EDITED BY
Houjun Tang,
Berkeley Lab (DOE), United States

REVIEWED BY
Dimitrios Nikolopoulos,
Virginia Tech, United States
Matthieu Dorier,
Argonne National Laboratory (DOE),
United States

*CORRESPONDENCE
Sebastian Oeste
sebastian.oeste@tu-dresden.de

RECEIVED 29 February 2024
ACCEPTED 18 July 2025
PUBLISHED 11 August 2025

CITATION
Oeste S, Höhn P, Kluge M and Kunkel J (2025) An analysis of the I/O semantic gaps of HPC storage stacks.
Front. High Perform. Comput. 3:1393936.
doi: 10.3389/fhpcp.2025.1393936

An analysis of the I/O semantic gaps of HPC storage stacks

Sebastian Oeste[1]*, Patrick Höhn[2], Michael Kluge[1] and Julian Kunkel[2]

[1]Center for Interdisciplinary Digital Sciences (CIDS), Department for Information Services and High Performance Computing (ZIH), Dresden University of Technology (TUD), Dresden, Germany, [2]Institute for Computer Science, University of Göttingen, Göttingen, Germany

Modern high-performance computing (HPC) Input/Output (I/O) systems consist of stacked hard- and software layers that provide interfaces for data access. Depending on application needs, developers usually choose higher layers with richer semantics for the ease of use or lower layers for performance. Each I/O interface on a given stack consists of a set of operations and their syntactic definition, as well as a set of semantic properties. To properly function, high-level libraries such as Hierarchical Data Format version 5 (HDF5) need to map their semantics to lower-level Application Programming Interface (API) such as Portable Operating System Interface (POSIX). Lower-level storage backends provide different I/O semantics than the layers in the stack above while sometimes implementing the same interface. However, most I/O interfaces do not transport semantic information through their APIs. Ideally, no semantics of an I/O operation should be lost while passing through the I/O stack, allowing lower layers to optimize performance. Unfortunately, there is a lack of general definition and unified taxonomy of I/O semantics. Similarly, system-level APIs offer little support for passing semantics to underlying layers. Thus, passing semantic information between layers is currently not feasible. In this article, we systematically compare I/O interfaces by examining their semantics across the HPC I/O stack. Our primary goal is to provide a taxonomy and comparative analysis, not to propose a new I/O interface or implementation. We propose a general definition of I/O semantics and present a unified classification of I/O semantics based on the categories of concurrent access, persistency, consistency, spatiality, temporality, and mutability. This allows us to compare I/O interfaces in terms of their I/O semantics. We show that semantic information is lost while traveling through the storage stack, which often prevents the underlying storage backends from making the proper performance and consistency decisions. In other words, each layer acts like a semantic filter for the lower layers. We discuss how higher-level abstractions could propagate their semantics and assumptions down through the lower-levels of the I/O stack. As a possible mitigation, we discuss the conceptual design of semantics-aware interfaces, to illustrate how such interfaces might address semantic loss—though we do not propose a concrete new implementation.

KEYWORDS
HPC, I/O, I/O stack, I/O semantics, POSIX, MPIIO

1 Introduction

The current I/O stack in high-performance computing (HPC) consists of several stacked hardware and software layers that can be highly heterogenous. Data read or written by the application passes through different interfaces on their journey through the I/O stack. Each layer attempts to efficiently map the given workload onto lower layers.

However, information about I/O semantics can be lost during translation to lower-level interfaces. Therefore, lower layers must rely on assumptions about application behavior made by higher layers. The interfaces between the layers can also offer the same API with different semantics. For example, a file system with a Portable Operating System Interface (POSIX) interface may have relaxed semantics regarding consistency for concurrent I/O operations on the same file region, potentially leading to unintended outcomes or reduced performance. In general, an interface consists of a set of operations that can be performed and their syntactic definition, as well as a set of semantic properties. While tools such as compilers can guarantee adherence to correct syntax automatically, the semantics of an operation are an implicit property of its implementation within the actual environment. The semantic implications inherent in an operation, or a series of operations, encapsulate the user's intentions. However, due to the absence of explicit semantic expressions in the APIs, this intent often gets lost. Additionally, some interfaces were historically not designed for distributed and parallel I/O, thus requiring special handling to function effectively in HPC workflows. Although the challenges posed by differing I/O semantics are well-known within the HPC community (Hildebrand et al., 2009; Devarajan and Mohror, 2023; Kuhn, 2013; Lockwood, 2017), no existing classification comprehensively describes the semantics across all layers of the I/O stack. Some existing work already gives a classification for the consistency semantics of parallel file systems (Wang et al., 2021). Others look for efficient translation to achieve semantics of an interface on another layer (Hildebrand et al., 2009). Still others have explored optimizations in lower layers that relax semantics to enhance performance (Vef et al., 2020). In this article, we present a comparative analysis of commonly used I/O interfaces in HPC, framed around a unified taxonomy of their I/O semantics.

Therefore, we make the following contributions: (1) A general baseline definition of I/O semantics and categories to classify these semantics in terms of *concurrent access*, *persistency*, *consistency*, *spatiality*, *temporality*, and *mutability* is presented that enables us to compare different interfaces regarding their I/O semantics. (2) We compare commonly used interfaces in HPC I/O stacks, analyzing their ability to transmit semantic information and identifying where semantics are lost due to layering. (3) We propose a design for a semantics-aware I/O stack. We emphasize that this work focuses on comparative classification and identifying gaps in the semantics of current HPC I/O interfaces.

The remainder of the paper is organized as follows: In Section 2, we give an overview of the layers of the I/O stack in order to assign the individual interfaces. Then, we introduce a checkpoint use case to describe the I/O APIs that function at the various layers of the I/O stack in Section 3. In Section 4 we provide a baseline definition of what an I/O semantic is. Furthermore, we propose six categories to describe the I/O semantics of an interface. Section 5 discusses commonly used I/O APIs for the different layers in the I/O stack and what semantics- they can communicate through their interfaces. Section 6 then discusses where the I/O semantics are lost in the checkpoint use case due to the layering of different interfaces. Section 7 discusses what other researchers have done regarding I/O semantics. Section 8 proposes a semantic-aware I/O stack that would be able to transport the semantics from the upper layers down to the underlying storage system. Finally, in Section 9 we conclude our study with a summary.

2 The I/O stack and the semantical gap

HPC systems offer a multilayer I/O stack on which the application is placed at the top layer and the physical storage medium at the bottom layer. Figure 1 illustrates the different layers in the I/O stack, between application and physical medium are the middleware, possibly with parallel I/O middleware, system-level, and storage layers. Higher layers may bypass some intermediate layers in the I/O stack. For example, an application might directly use POSIX I/O without involving middleware libraries, or the middleware could bypass the system layer by directly interacting with storage interfaces. As operations traverse the different layers of the I/O stack, the shape of the operations may change. This means a pattern specified at the application level might differ by the time it reaches the file system or physical medium. Intermediate layers may apply optimizations such as data aggregation, reordering, buffering, and caching (Thakur et al., 1999a; Liao et al., 2007). However, contextual information may be lost when traversing the I/O stack, causing lower layers, such as the file system, to become unaware of the application's original intent. While higher-layer interfaces allow to express the users intent more explicitly in their API for example, collective I/O operations in message passing interface (MPI)-IO (Corbett et al., 1996) or data layouts and file formats in Hierarchical Data Format version 5 (HDF5) (Folk et al., 2011), the deeper the layer, the closer to the hardware, and the flexibility of the expressible semantics decreases. For example, POSIX I/O provides an API based to work on an unstructured stream of bytes through integer handles (*file descriptors*) with no notion for parallelism or layout. On the physical medium, requests would be processed independently through multiple queues, for example, on modern Non-Volatile Memory Express (NVME) Solid State Disks (SSDs). The responsibility to ensure a correct ordering of the request is at higher layers in the operating system. That is what we call the *semantical gap* in the I/O stack. Each layer tries to translate an operation to a more primitive set of operations from the layer below. The different layers act like a semantic filter for the lower layers, which can prevent performance optimizations. We start explaining the stack from the bottom to top, starting with the physical medium up to the actual application.

2.1 Physical medium

The physical medium is the lowest level in the I/O stack and is a piece of hardware that physically stores the data. This includes volatile media, such as RAM, and non-volatile media such as Non-Volatile Random-access-memory (NVRAM), disks, or tapes. The I/O semantics on this layer are largely fixed and defined by the hardware design or firmware used. The higher software layers in

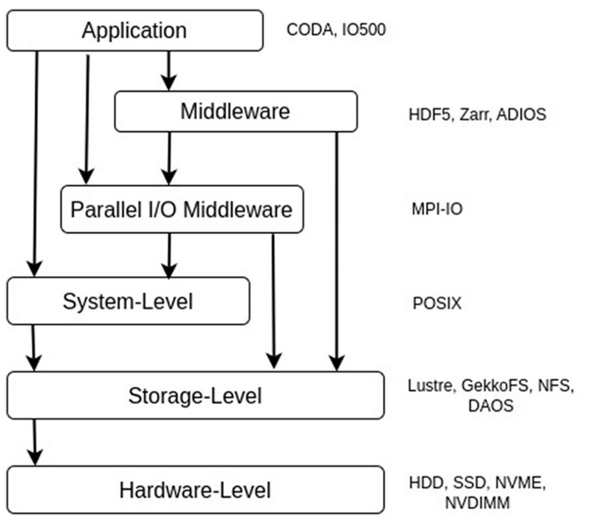

FIGURE 1
Illustration of the high-performance computing I/O stack with possible interactions between individual layers.

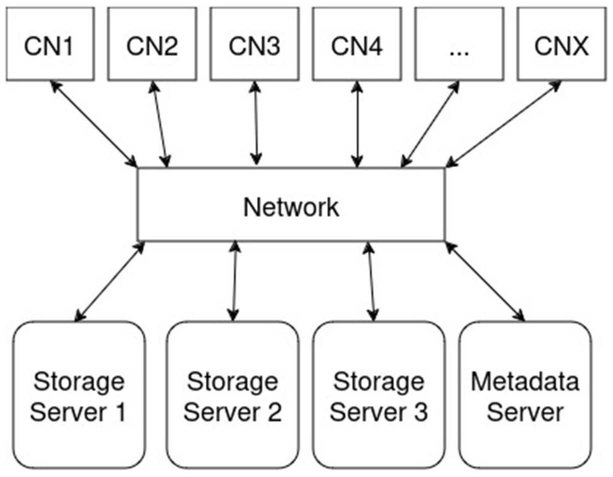

FIGURE 2
Abstract architecture of a parallel file system, illustrating metadata and storage servers connected to cluster compute nodes via a network interconnect.

the I/O stack are not able to modify the semantic properties of that layer.

2.2 Storage system

The storage system or file system is the software layer above the physical medium and is responsible to map logical I/O from the application to physical I/O on the hardware. But file systems do a lot more than just present a file based interface to persist on a physical medium. Common tasks performed by file systems include *read caching*, *write buffering*, and generating additional I/O to maintain on-disk layout metadata (Gregg, 2014). Because HPC systems are distributed systems, comprising multiple compute nodes executing applications, the parallel file system is also a distributed and consists of several components. Figure 2 shows an example for a typical architecture of a parallel file system. Most parallel file systems distinguish between metadata and storage servers connected to the compute nodes over a high-speed interconnect. Clients request file and layout information from the metadata server upon opening a file, then directly perform read or write operations to storage servers. While most parallel file systems used in HPC came with a POSIX interface for portability reasons, there was a number of relaxations to its strict semantics. The most common relaxation is reducing consistency guarantees for overlapping writes to the same file region. For example, PVFS (Carns et al., 2000) and its successor OrangeFS (Bonnie et al., 2011) define the result of such access pattern as undefined. Because HPC applications typically organize their access pattern at a higher layer they leave the responsibility at these higher layers to avoid expensive locking. Furthermore, file systems like Lustre (Braams, 2002) support features like exclusive strides to give a process exclusive access to a part of a file. GekkoFS (Vef et al., 2020), designed as a job-temporal file system, relaxes the semantics of metadata operations, such as `readdir`, to eventual consistency while maintaining strong consistency for data operations. Nevertheless, it is always a trade-off for file system developers to assume certain access characteristics that the application is doing or not doing to optimize for performance. The interface between the file system and higher layers provides no mechanism to convey the application's context beyond basic instructions like reading or writing *n* bytes at a specific file offset.

2.3 System-level-API

We define the system-level storage API as the software layer interfacing with the storage system. System-level APIs provide a set of primitives for managing system resources. For storage systems, these resources are the available storage space and the data stored within it. In HPC, most storage systems use the POSIX interface for this purpose. POSIX provides a hierarchical structure of files and directories and defines functions like `open`, `read`, `write`, and `close` or `opendir`, `readdir`, `closedir`, and `mkdir` to work with them. Resources are addressed using paths that map the directory structure up to the file.

2.4 Middleware

A middleware library is a software layer that translates abstractions from high-level, domain-specific APIs to lower level APIs. At the middleware layer, libraries that combine data layout abstraction with I/O abstractions are used for example, HDF5 (Folk et al., 2011), netCDF (Rew and Davis, 1990), or ADIOS (Lofstead et al., 2009). In addition to providing data layout and formats, a common task is to improve the use of the low-level API for a given platform. For example, the MPI-IO *ADIO*, (Thakur et al., 1996) interface allows specific implementations depending on which file system is used. Middleware APIs can perform such optimizations because they possess richer contextual information about execution than lower-level APIs.

2.5 Application

The application layer is the highest layer in the I/O stack that has the most semantic information. An application, is the data producer or data consumer that initiates the I/O. For example, these are interactive tools, scientific simulations, AI applications or benchmarks. The application level has the highest degree of abstraction and most flexibility in expressing the user's intent. At this level, it is known whether it is I/O for a snapshot, a log file, a result file or reading of input data. Here, it is also known if a file would be accessed by only one process or a group of processes, just once or repeatedly. Typically, applications select libraries aligned with their scientific domain to represent data layout and access patterns. Examples include the Flexible Image Transport System (FITS) (Pence et al., 2010), commonly used in astronomy, or General Regulary-distributed Information in Binary form (GRIB) for meteorological data. These libraries provide different sections for images, ASCII tables or binary tables and a sequence of header data units to store metadata of these sections. The goal at this layer is to express operations as close as possible to the scientific domain. However, predicting application performance is challenging without considering the underlying layers within the actual execution environment. Significant complexity resides within the lower software and hardware layers, substantially influencing I/O performance.

3 Example of a checkpoint use case

In this section, we illustrate our motivation using a checkpointing use case. Checkpointing is a common technique for storing intermediate results of an application. The goal is to be able to recover from that checkpoint in the event of an error. Many traditional HPC simulations, as well as newer artificial intelligence applications use checkpointing. Typically triggered by application, state changes or specific time intervals, an application writes its current internal state to persistent storage. Because there are various types of potential failures, we assume that the checkpoint will be created to recover from a node failure. The primary focus of this section is to describe the I/O APIs that function at various levels of the I/O stack.

Figure 3 illustrates three possible storage stacks suitable for a checkpointing application. The first I/O stack in Figure 3a includes HDF5, MPI-IO, POSIX, and Lustre, utilizing high-level I/O libraries. The second stack (Figure 3b) uses only POSIX and Lustre. Finally, the third stack (Figure 3c) depicts an application using POSIX atop the GekkoFS file system.

A primary objective is to write checkpoints as quickly as possible, minimizing the interruption of application computation. In the following, how these layers work together for the different

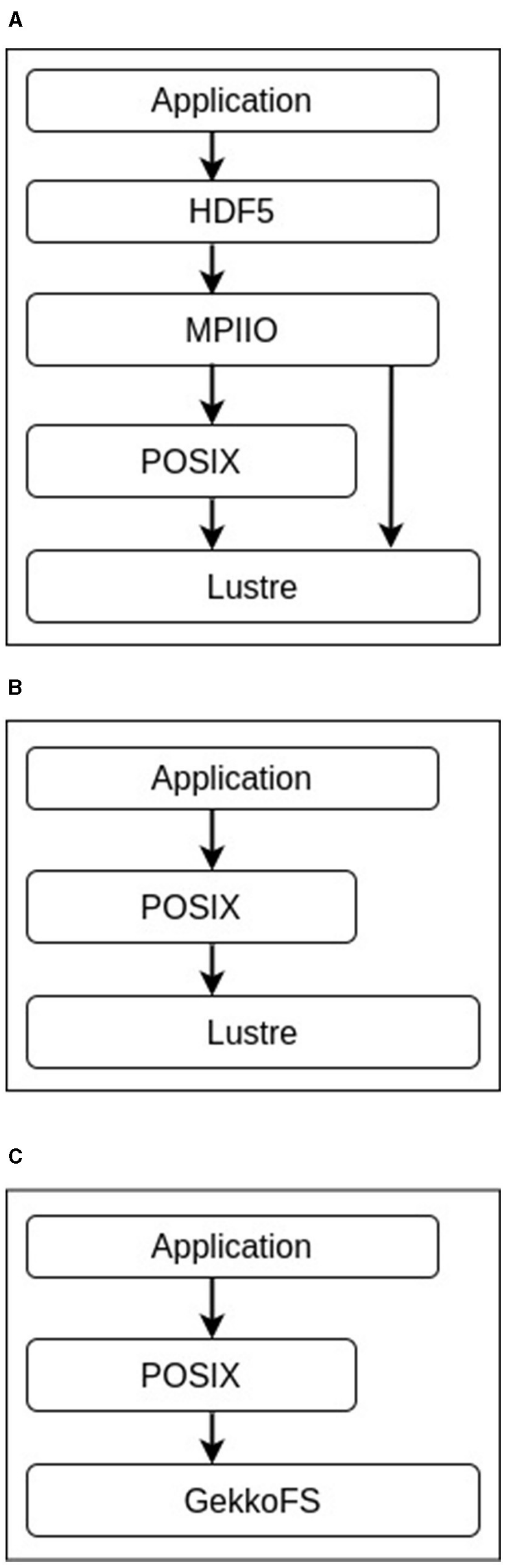

FIGURE 3
Examples I/O stacks to discuss the transport of I/O semantics. **(a)** Example I/O stack (Stack 1) using HDF5 and MPIIO as middleware on top of POSIX and Lustre. **(b)** Example I/O stack (Stack 2), where the application directly uses POSIX on Lustre. **(c)** Example I/O stack (Stack 3), where the application uses POSIX on GekkoFS.

stacks, what their tasks are, and what kind of information they need are explained.

In all three stacks (Figure 3), the user selects the storage location at program start-up, typically by specifying a file system path. The persistency of the storage defines, from which types of failures the checkpoint can be recovered from. In the first stack (Figure 3a), the application uses HDF5 for data management and storage. HDF5 is responsible for the data format and layout. HDF5 manages the data format and layout, organizing in-memory data structures and their storage layout within files. This involves defining the data structures, metadata, and layout of the checkpoint file. HDF5 employs MPI-IO as middleware for parallel I/O operations. MPI-IO uses MPI to communicate across processes during I/O operations, this enables for collective I/O by multiple processes.MPI-IO can optimize data access patterns by aggregating, deferring, or re-ordering I/O operations. Further, MPI-IO is responsible for synchronizing the parallel processes to ensure that they operate on valid data. At this layer, MPI-IO is also capable of configuring a proper data placement strategy, for example, stripe settings and pools, via a Lustre API within its abstract device interface (ADIO) (Thakur et al., 1996). To transport this information, MPI-IO skips the system-level and targets the storage system directly. Middleware layers will transport information about data sizes and layout and organize data dependencies between different processes. Depending on the checkpoint data volume and the number of processes, MPI-IO gathers data into dedicated I/O processes.

In the I/O stacks shown in Figures 3b, c, no middleware libraries are used; the application directly utilizes a POSIX I/O. Therefore, the application is responsible for managing the translation of its data structures to system-level I/O operations. The application is also responsible for synchronizing multiple processes and ensuring an appropriate file structure and layout.

Because MPI-IO also uses a POSIX I/O for file operations on Lustre, all three stacks use POSIX at the system-level. At this level, data are represented as an unstructured stream of bytes; both the volume of data and how they are divided in into individual requests are important. Grouping contiguous I/O operations into larger requests can increase I/O bandwidth and reduce the overhead of processing numerous small requests. If the data are written in fixed-size requests, optimizations such as *direct I/O*, which bypasses the page cache, might be appropriate. Information regarding overlapping accesses or read–modify–write patterns is also important. The absence of such accesses can improve performance, as it allows for less synchronization between different processes.

When an application performs I/O operations, requests are ultimately handled by the file system. The I/O stacks in Figures 3a, b use the Lustre parallel file system. Lustre provides a POSIX interface along with file system-specific options for controlling data distribution across storage servers. The parallel file system will distribute data across different storage devices. Lustre ensures that the data written or read by the checkpoint application are distributed across multiple servers to maximize the available bandwidth for parallel applications. By comparison, for small I/O requests, using a single disk reduces the overhead of maintaining multiple network connections.

The third stack in Figure 3c uses GekkoFS as a node-local, job-temporal file system. GekkoFS provides a shared namespace over node-local storage resources, such as local disk. Similar to Lustre, it provides a POSIX interface but relaxes the consistency guarantees for directory operations. The file data will be distributed over all nodes within the job. To ensure persistency beyond the job runtime, an explicit stage-out of data to more persistent storage is necessary. GekkoFS itself just provides persistency for the runtime of the job.

This section explained the role of individual layers for three different I/O stacks in a checkpointing application. In the next section, we introduce a taxonomy for I/O semantics that groups the necessary information for different layers into separate categories. After that, Section 5 explains what the different interfaces can express in terms of our semantic taxonomy. Finally, in Section 6, we revisit our example and explain what semantics are lost through layering because interfaces cannot transport them.

4 A definition and taxonomy of I/O-semantics

This section provides a definition of I/O semantics. We distinguish between a general definition and the specific features provided by an interface. The semantics are categorized to describe common I/O APIs.

4.1 I/O semantic

In literature, I/O semantics is mostly abstracted to consistency semantics of I/O operations (Wang et al., 2021, 2024; Thakur et al., 1999b; Hildebrand et al., 2009). We define I/O semantics more universally.

I/O Semantics Definition:

I/O semantics describe the meaning of I/O operations in the current execution context and their consequences for subsequent I/O operations.

An I/O interface consists of a set of operations, defines their syntax (API), specifies how a valid operation are expressed, and includes a set of semantic properties. I/O semantics describe the meaning and behavior of an I/O operation within its execution context. This includes how data is accessed, what transformations are applied, and the guarantees provided by successful I/O operations. In contrast, an API specifies how an interface must be used in an implementation. One could say that an API represents the operational view, while I/O semantics represent the behavioral view. A well-written API definition includes semantics to ensure the user correctly understands the API. Execution context refers to the surrounding system in which an operation executes. This includes both hardware and software components, such as storage servers, networks, disks, file system software, and operating systems. It also includes I/O operations interacting with the same resources. Different components within the execution context might implement the same API for portability but provide differing semantics.

TABLE 1 Overview of the semantic categories defined for parallel I/O and their characteristics.

Semantic category	Characteristic	Meaning
Concurrency	Exclusive/Shared	Are resources exclusively used by one process or thread or shared by many?
Persistency	Application/Job/System	Where should data be placed, and how persistent should it be against failures?
Consistency	Sequential/Session/ /Commit/Eventual	When should updates be visible to subsequent operations?
Spatiality	Contiguous/Strided/ Random	How data are expected to be accessed?
Temporality	Once/Periodic/Random	When will data be reused?
Mutability	Read-only/Read-write/ Overwrite/Append	Will the data change, and if so, how?

Additionally, distinguishing between the semantics and features of an interface is important. Semantics describe what an interface allows users to accomplish. Thus, semantics help clarify the purpose and behavior of the interface and its operations. Interface features describe the technical implementation details.

4.2 Semantic categories

To compare interfaces in terms of their I/O semantics, a taxonomy is needed. We define six semantic categories to describe storage system requirements and parallel I/O access behavior. Furthermore, each category has a number of relevant characteristics in the HPC context. In the following, we describe the semantic categories into which we subdivide I/O semantics, as well as their characteristics, and we describe how storage systems can benefit from this information. Table 1 summarizes the categories and their characteristics.

Concurrent access: Concurrent access semantic define whether I/O accesses on a resource are executed *exclusive* or *shared*. Exclusive means that only one process or thread executes I/O operations. By comparison, shared access means that multiple processes or threads execute I/O operations concurrently. We subdivide the characteristics of concurrent access into

- *Exclusive*—the I/O access is executed by exactly one process or thread, and
- *Shared*—the I/O access is executed by multiple processes or threads concurrently.

If an underlying storage system knows that a resource is accessed exclusively, it might avoid expensive locking to protect the resource from concurrent access or cache the resource in memory or storage locations closer to the requesting thread or process.

Persistency: Persistency describes the durability of stored data across different failure scenarios and time scales. It characterizes how long data remain accessible and under what condition they may be lost. Persistence storages guarantee modifications are retained and are power-fail safe, whereas volatile storage keeps data only during the application's lifetime (Scargall, 2020). In terms of I/O semantics within a distributed environment, persistency defines *how* resistant data must be against failures. Therefor, we define persistency characteristics based on the data lifetime and failure conditions:

- *Application Level*—Data are stored volatilely (e.g., in memory) and persist only during the application's execution. They are lost if the application crashes.
- *Job Level*—Data persist throughout an HPC job's execution (e.g., on node-level resources) but may be lost when the jobs ends or if a node fails.
- *System Level*—Data are stored on dedicated storage servers (e.g., a parallel file system). Once written, data are persistent, surviving job termination and system reboots.

Describing persistency semantics at different layers of the I/O stack is important for understanding data durability, optimizing performance, and ensuring fault tolerance. Applications need to know how long data will persist and under what conditions it may be lost. For example, a checkpointing system may store data on node-local storage (job-level persistency) but must stage it out to a parallel file system (system-level persistency) to enable recovery after node failures. Applications and middleware can select storage on persistency needs. For example, MPI-IO may optimize data placement by buffering writes in volatile memory before committing data to persistent storage. Consequently, persistency semantics aid applications, middleware, and storage systems in optimizing data placement, caching strategies, and fault tolerance.

Consistency: Consistency defines *when* updates to shared data become visible to subsequent operations.

For the consistency characteristics, we follow the categorization of Wang et al. (2021).

- *Sequential Consistency*—We define strong consistency with the condition that a read from a byte returns the value written by a write to the byte if the write happens before the read.
- *Commit Consistency*—Commit consistency is defined as updates to a byte or a region of bytes are visible to other processes after an explicit commit operation, for example, a `flush`.
- *Session Consistency*—Session consistency defines updates are consistent outside of session boundaries, for example the time when a file is closed until it is opened by an process.
- *Eventual Consistency*—Eventual consistency does not define an explicit point in time when data become consistency; instead, it is expected that after a long enough period all subsequent reads to shared data will return the same value.

Knowing the consistency requirements of operations on shared data likely holds the greatest potential for optimizing parallel file systems. Most parallel file systems provide a POSIX interface for portability, which requires *sequential consistency*. However, strict consistency is not required for most HPC applications (Wang et al., 2021). By relaxing the consistency semantics, parallel file systems could eliminate costly consistency protocols and distributed locking. A number of commonly used parallel file systems in HPC employ relaxed consistency semantics. For example, the Network File System (NFS) uses a *close-to-open*, *session consistency*. While file systems like PVFS and OrangeFS relax sequential consistency for updates to the same file offset (Carns et al., 2000; Bonnie et al., 2011).

Spatiality: Spatial access patterns describe how an application intends to structure and access data. However, the actual storage layout may be reorganized by lower layers. Therefore, spatiality consists of two aspects: the expressed layout, how applications and middleware structure I/O, and the realized layout, how file systems and storage hardware store the data. Because actual data placement is ultimately determined by lower layers in the I/O stack and cannot be directly controlled by the application, we define spatiality characteristics based on the application's intended data access pattern:

- *Contiguous*—Data are accessed contiguously, and offsets increase monotonically, where $offset_{start} < offset_{end}$ and $IOP1_{offset_{end}} == IOP2_{offset_{start}} - 1$, and $IOP1$ and $IOP2$ refer to two consecutive I/O operations of one process.
- *Strided*—Data are accessed non-contiguously, with offsets increasing by a fixed *stride* between the end offset of an operation and the start offset of the next operation, where $IOP2_{offset_{start}} == IOP1_{offset_{end}} + stride$.
- *Random*—Data are accessed at random offsets that do not follow an obvious pattern.

Layers in the I/O stack can leverage spatiality semantics to optimize data placement, buffering, caching, and retrieval. The application layer can express expected spatial patterns for optimized data structures (e.g., column-major vs. row-major layouts) to improve memory locality. Middleware layers can aggregate and reorganize I/O requests to better align with underlying storage structures. For example, MPI-IO can aggregate small, scattered writes into larger, more efficient requests before passing them to the storage system. Data access with a non-contiguous, strided pattern could enable optimization techniques like *two phase-IO* (del Rosario et al., 1993; Kang et al., 2020), where the data distribution on compute resources is decoupled from the storage distribution. On the system-level, spatiality hints can optimize buffering, prefetching, and caching strategies. The POSIX `posix_fadvise` call can inform the operating system about anticipated access patterns, improving read-ahead efficiency. This can be used to improve sequential access performance. Underlying storage systems are responsible for the actual data placement. If they support explicit data placement, system and middleware libraries can use spatiality hints to tune stripe patterns according to expected access pattern (e.g., aligning stripe sizes with strided access). Consequently, we differentiate storage systems based on their support for explicit data placement, allowing higher layers to optimize data layout based on access pattern hints.

Temporality: In the context of HPC I/O, temporality semantics describe how frequently and in which temporal patterns data resources are accessed during a programs execution. It essentially measures temporal locality, quantified by reuse distance. A reuse distance describes the number of distinct data accesses that occur between two consecutive accesses to the same data item (Ahmadian et al., 2021; Lee et al., 2011). Unlike spatial access patterns, which concern *where* data is accessed in storage, temporality focuses on *when* or how often data are reused. Therefore, temporality can be described as the number of distinct data accesses that have happened between two accesses to the same file. No particular ordering of operations is assumed; instead, temporality captures how often a resource is utilized by a process over time. Because HPC applications often exhibit bursty I/O patterns (Yu et al., 2020; Tang et al., 2017), expressing such information could be beneficial. We group the number of accesses into three categories:

- *Once*—The data are accessed only once during the program's runtime and not reused thereafter,
- *Periodic*—The accessed data will be used multiple times, following a periodic pattern, and
- *Random*—Access occurs multiple times at irregular intervals without an obvious pattern during the program's runtime.

By integrating these categories into the semantics of the I/O stack, the system can make informed decisions regarding caching layers, buffer management, and pre-fetching strategies. If a dataset is known to be accessed only once, the storage system can avoid unnecessary caching and optimize for data streaming. For a periodic reuse pattern, if the reuse period is short compared to the cache retention time, the data can be retained in the cache. Even if the period is longer, knowing the pattern allows caches or burst buffers to be sized or tuned to hold the data until the next use. Furthermore, a known periodic pattern can be used for pre-fetching data before it's needed by the application, thereby reducing I/O latency (Byna et al., 2008). With this information available within the layers of the I/O stack, middleware libraries can avoid unnecessary buffering for one-time accesses. For periodic patterns, datasets could be retained in memory between phases, or periodic writes could be batched and flushed in the background. At the system-level, this information could enable write-through policies such as `O_DIRECT` or `posix_fadvise(..., POSIX_FADV_NOREUSE)` to reduce cache pollution from one-time accesses and facilitate sequential read-ahead for periodic access. Multi-tiered storage systems can discard data used only once from faster storage tiers, keeping space available for data with higher reuse frequency. For periodic accesses, parallel file systems can pin files or cache stripes accordingly.

Mutability: Mutability semantics describe if and how data are expected to change after being created. Within the HPC I/O stack, mutability can be expressed in multiple ways. A common expression of mutability are open modes such as the POSIX or MPI-IO flags (`O_RDONLY`, `MPI_MODE_RDONLY`) to open a file in read-only mode. Another form includes file access permissions in the file system's metadata, which restrict read and write access for specific users or processes. Finally, the file system's implementation itself affects mutability; for instance, copy-on-write file systems never overwrite existing blocks but instead store references to new blocks containing updated data.

For HPC I/O semantics, we separate access permissions from semantic intent, defining mutability based on whether and how data changes during the file's access period:

- *Read-Only*—Data do not change during access.
- *Read-Write*—Data will be read and written during the access.
- *Overwrite*—Data are written, potentially overwriting existing data.
- *Append*—Writes are appended only at the end of the file.

A given mutability semantics over the I/O stack offer optimization potential in the following cases. Middleware libraries, such as MPI-IO, can pre-compute offsets to reduce coordination overhead for append semantics. For overwrite semantics, optimizations such as data sieving or two-phase I/O could be applied. Read-only accesses can enable caching or pre-fetching (Dinh et al., 2017) optimizations. Storage systems might employ sequential striping in append mode to reduce seek times, and journaling or copy-on-write in overwrite mode to coalesce writes.

5 Semantics of I/O-interfaces

This section supports our goal of comparing interfaces, using the semantic taxonomy introduced in Section 4. Our intent is not to develop or propose new interfaces, but to map existing ones to the defined semantic categories. We choose interfaces that are widely adopted within HPC applications. At the end of the section, we compare the semantics of the APIs and discuss where information gets lost.

5.1 Middleware I/O interfaces

MPI-IO: The MPI is a standardized communication protocol used for managing and coordinating highly parallel scientific workloads. MPI-IO, as the standard for I/O within MPI, was first defined in the second version of the MPI standard (Message Passing Interface Forum, 1996). In addition to standard data types, derived data types can be employed in data partitioning to accommodate custom data structures. There are different MPI-IO implementations; the standard only defines the required API and semantics.

Concurrent access in MPI-IO is possible by using explicit offsets, individual file pointers, and shared file pointers. It is synchronized through blocking, non-blocking, and split collective mechanisms. Coordination between different execution units can be either non-collective or collective. Therefore, accessing a file collectively is possible by defining the group of processes that open the file through a communicator passed to `MPI_File_open`.

MPI-IO cannot provide any guarantees in terms of persistency. Persistency in the scope of MPI-IO is solely governed by the execution context, for example the file system underlying the currently accessed data.

For parallel I/O with conflicting concurrent access between processes, having a deterministic sequence, particularly in the case of write operations, is essential. Otherwise, the final result cannot be predicted beforehand. By default, MPI-IO does not guarantee any ordering of individual calls (Message Passing Interface Forum, 2023).

Two types of consistency can be distinguished. First, sequential consistency can be guaranteed (Padua et al., 2011) if the concurrent processes open the file collectively and enable atomic mode. However, this setting significantly degrades performance when used with HDF5. Atomic mode ensures that changes to a file are immediately visible to other processes in the group that opened the file collectively (Padua et al., 2011). Additionally, sequential consistency can be guaranteed for concurrent processes when accessing the file using a single file handle (Message Passing Interface Forum, 2023).

Second, a *sync/barrier/sync* session can be used to achieve consistency. In this approach, user-managed synchronization of the processes is required (Message Passing Interface Forum, 2023). With the *sync/barrier/sync* construct, data become visible only after both the writer and the reader have executed a *sync* operation to flush the data. A *sync* operation may be `MPI_File_sync`, `MPI_File_open`, or `MPI_File_close`, `thakur_implementing_1999`. To enforce ordering between the two *sync* operations, a *barrier* operation is necessary. The *sync/barrier/sync* semantics resemble session consistency, with synchronization boundaries akin to session start and end.

Additionally, hints regarding file access can be provided through the `MPI_Info` object and passed to functions such as `MPI_File_open`, `MPI_File_set_view`, or `MPI_File_set_info`. Temporality can be expressed by using the access styles "read_once" and "write_once," although no standard hint exists specifically for temporal reuse. Spatiality can be influenced by enabling collective buffering and defining the "cb_nodes" and, more generally, the "io_node_list" variables, as well as file layout parameters such as "striping_unit" and "striping_factor" via `MPI_Info`. Non-contiguous data accesses can be described using MPI-derived data types, such as `MPI_Type_create_subarray` or `MPI_Type_vector`. Mutability can be defined while opening the file using the `MPI_MODE_RDONLY` mode. The semantics of this mode are identical to the POSIX counterpart, as discussed in a subsequent section.

Characteristics for semantic categories:

- **Concurrent access:** MPI-IO has a notation for both *shared* or *exclusive* access.
- **Persistency:** No notion, depends on lower layers.
- **Consistency:** Sequential or *sync/barrier/sync* session consistency semantics.
- **Spatiality:** Via hints to `MPI_Info` and MPI-derived data types.
- **Temporality:** Hints can be given via `MPI_Info`.
- **Mutability:** During file open via *flags*.

HDF5: The HDF5 is a widely used file format in scientific codes. As a hierarchical data format, HDF5 internally consists of *groups* and *datasets*. Compared to POSIX terminology, these can be seen as directories and files, respectively. The access of files on disk is realized through specific virtual object layers (VOLs) and virtual file drivers (VFDs). In terms of backends, HDF5 uses MPI-IO for parallel I/O and POSIX I/O for sequential I/O. An asynchronous volume connector is also available (Tang et al., 2019, 2022).

Standard HDF5 was designed for single-core use, where no concurrent access occurs, and thus resources remain truly exclusive. HDF5 supports parallel concurrent access either through parallel data access using parallel HDF5 (pHDF5) or the single writer multiple reader (SWMR) pattern. We focus on parallel HDF5 with MPI-IO, as it is more widely adopted within HPC. For pHDF5, exclusive access is permitted only for operations that do not modify structural metadata. All other operations must be performed using shared access. MPI communicators are used to define groups of processes.

HDF5 cannot provide any guarantee in terms of persistency. Similar to MPI-IO, it depends on the lower layers of the I/O stack. However, the choice of VOL can have limited influence in this context, for example, when the HDF5 Cache VOL storage type is set to memory (Zheng et al., 2022).

Consistency semantics are governed by the selected VFD, as HDF5 files are usually located on an underlying file system. pHDF5 uses MPI-IO as its standard VFD, and therefore, its consistency model is generally identical to that of MPI-IO (The HDF Group et al., 2012). To achieve consistency among parallel processes, many operations (e.g., `H5Fcreate`, `H5Fopen`) must be called collectively.[1] Collective calls require all participating parallel processes to follow the same execution order of function calls. These collective API calls have specific requirements regarding data type, data space, access properties, and creation properties for participating processes in the MPI communicator (cf. see footnote 1).

Similar to MPI-IO, pHDF5 supports atomicity using the sync–barrier–sync mechanism. It can be enabled through the function `H5Fset_mpio_atomicity` (Koziol and Breitenfeld, 2015). The major drawback of atomic mode is a significant drop in read and write performance (The HDF Group et al., 2012).

HDF5 allows spatial access patterns to be described using hyperslabs to define non-contiguous selections within a dataset. Additionally, HDF5 datasets support chunking, which enables efficient caching and compression for non-contiguous access. Further access hints can be conveyed through MPI-IO when used in combination with MPI.

For temporality, HDF5 offers indirect support via APIs to control cache sizes and flushing—for example, `H5Dflush` or the chunk cache configuration, which can be aligned with temporal reuse.

Mutability can be defined when opening the file by using the `H5F_ACC_RDONLY` or `H5F_ACC_SWMR_READ` modes. The semantics of the `H5F_ACC_RDONLY` mode are identical to those of the POSIX counterpart, as discussed in a subsequent section.

1 Collective Calling Requirements in Parallel HDF5 Applications. https://docs.hdfgroup.org/archive/support/HDF5/doc/RM/CollectiveCalls.html (Accessed September 18, 2023).

Characteristics for semantic categories:

- **Concurrent access:** Via MPI-IO for parallel access.
- **Persistency:** No notion, depends on lower layers.
- **Consistency:** Sequential consistency.
- **Spatiality:** Via hyperslab, chunking, and MPI-IO.
- **Temporality:** Only indirect support.
- **Mutability:** During file open via flags.

5.2 System-level I/O interfaces

POSIX I/O: POSIX is the Portable Operating System Interface standard (posix, 2018), originally developed by the IEEE Computer Society and the Open Group in the late 1980s. The main objective was to maintain compatibility between operating systems. POSIX I/O, as the I/O component, defines both synchronous and asynchronous interfaces. It is widely available and the most commonly used system-level API for I/O. However, POSIX was designed for local file systems, not with parallelism in mind, and it has not changed significantly in many years. Its widespread use in HPC environments is primarily due to its portability. There was an effort to extend POSIX I/O for HPC and parallel file systems with features such as shared file descriptors, group open, lazy metadata, non-contiguous read/write interfaces, and bulk metadata operations (Vilayannur et al., 2008). Unfortunately, these extensions were not included in future revisions of the POSIX standard. Consequently, POSIX lacks support for the execution of collective operations. It is not possible to express that an operation is executed by a group of multiple processes or threads. Nevertheless, ideas such as lazy metadata queries made it into the Linux system call interface (e.g., with functions like `statx`).

POSIX requires sequential consistency for data and metadata. This means that a `write` operation issued by a process blocks until the system can guarantee that any subsequent `read` will retrieve the data that were just written. In a distributed system, where remote processes are unaware of what local processes are modifying, supporting sequential consistency comes at a cost. Parallel file systems require synchronization and often distributed locking to enforce sequential consistency, which impacts performance. For this reason, some parallel file systems already relax these consistency requirements.

In POSIX, data are stored in regular files and directories. Files and directories are organized in a hierarchical, treelike structure. Each file can be addressed by its *path name*, which may be absolute or relative to the current working directory. Whether the data are persistent or volatile depends on the physical medium of the file system that underlies the path. The actual underlying physical medium is defined by the execution context, for example, by the block device behind the mount point where the file path ends. An update is persistent when the data are successfully transferred, where the term "successfully transferred" is defined as follows:

> *For a write operation to a regular file, when the system ensures that all data written is readable on any subsequent open of the file (even those that follow a system or power failure), in the*

absence of a failure of the physical medium. For a read operation, when an image of the data on the physical storage medium is available to the requesting process.

This means that, when a POSIX I/O function has "successfully transferred" its data, it is guaranteed that the data are written consistently to the underlying file system. According to our definition from Section 4, persistency is defined by lower layers in the I/O stack, such as the underlying file system. POSIX I/O itself has no notion of a required level of persistency.

Because POSIX exposes I/O as an unstructured stream of bytes, the API has little support for expressing spatiality or temporality of data accesses. The `posix_fadvise` function can provide hints to the kernel about the access pattern for a region of a file, supporting the choice of suitable caching or pre-fetching behavior. These hints can convey information such as whether the data will be accessed sequentially or randomly, but neither strided access patterns nor explicit data placement is supported. Regarding temporality, the `posix_fadvise` function supports flags such as `POSIX_FADV_NOREUSE` and `POSIX_FADV_WILLNEED`, which correspond to one-time and periodic semantics.

Mutability can be expressed using *flags* passed to the `open` call. For example, the flags `O_RDONLY`, `O_RDWR`, and `O_WRONLY` define whether a file should be opened for read-only access, read-write access, or write-only access, respectively. Append semantics can be expressed with `O_WRONLY | O_APPEND`, which guarantees that every `write()` appends to the end of the file. The `O_APPEND` flag guarantees atomicity for individual writes but not for coordination among multiple processes. If a file is opened in read-only mode, any write operation to that file descriptor will fail.

Characteristics for semantic categories:

- **Concurrent access:** POSIX has not notation for *shared* or *exclusive* access.
- **Persistency:** No notion; depends on lower layers.
- **Consistency:** Sequential consistency.
- **Spatiality:** Via `posix_fadvise` for contiguous and random accesses.
- **Temporality:** Via `posix_fadvise`.
- **Mutability:** During file open via *flags*.

5.3 Storage-level I/O interfaces

NFS: NFS is a common protocol for sharing files between Unix systems over a network. Nevertheless, it is not considered a parallel file system, as it does not support concurrent write access to the same file (Pawlowski et al., 2000). NFS, in versions prior to 4.1, only allowed exclusive access to files. Version 4.1 introduced byte-range locks, which allowed concurrent access to the same file.

By default, NFS relies on a client-side caching mechanism. Delayed writes may be lost if the client crashes before syncing. In such cases, NFS can only guarantee job-level persistency. After synchronization (e.g., a successful call to `close` or `fsync`), NFS can also provide system-level persistency. With the `sync` mount option, NFS is forced to commit writes immediately to the server, thereby providing system-level persistency.

Using default settings, NFS guarantees close-to-open consistency, that is, changes are written back to the server only when the client closes the file. Therefore, changes on the client are not immediately reflected on the server, and different clients might see inconsistent states at any point in time (Kuhn, 2013). This allows for higher performance compared to strict POSIX I/O semantics. NFS Version 4.1 removed the previously required exclusive file access in favor of byte-range access within a file. The NFS extension, parallel NFS (pNFS), also resolved the single-server bottleneck by decoupling state and data servers (Fridella et al., 2010). The physical location of the data is retrieved as a "layout" from the state servers and used by the layout client to access the data (Hildebrand et al., 2009).

NFS does not support explicit data placement or distribution of data in terms of spatiality. Similarly, there is no explicit support for temporality in the NFS protocol.

Characteristics for semantic categories:

- **Concurrent access:** No notion.
- **Persistency:** Default Job level; after successful system-level synchronization.
- **Consistency:** Session; *close-to-open* consistency.
- **Spatiality:** No notion.
- **Temporality:** No notion.
- **Mutability:** Via POSIX open *flags*.

DAOS: Distributed Asynchronous Object Storage (DAOS) is a modern storage system designed for flash-based architectures (Liang et al., 2020). DAOS provides its own I/O API and does not rely on POSIX. The DAOS API includes functions for manipulating pools, containers, and objects. User content is stored exclusively in objects. In this section, we focus on functions that handle user data, not administrative functions.

Pools group storage devices together. Within each pool, containers—similar to S3 buckets—are created to store objects. These objects hold user data in various layouts. Unlike user interfaces such as POSIX, DAOS provides a broader range of data accessors, e.g., arrays and key-value stores.

DAOS includes an abstraction layer called DFS, which serves as a POSIX replacement for applications that cannot directly use the DAOS API. One of DAOS's core design principles is to process all types of write requests as quickly as possible. Consistency is enforced when data are read (Barton, 2015).

DAOS offers a transactional interface for container versions. This is feasible because DAOS is primarily designed for non-volatile memory DIMMs (Dual in-line memory modules) such as storage class memory or NVRAM. NVRAM can store metadata for ongoing I/O operations very efficiently and provide fast read access for fine-grained retrieval. Newer versions of DAOS will utilize standard dynamic random-access-memory (DRAM) and implement a flash-based, log-structured persistence layer.

To provide persistence across nodes, DAOS can replicate data and write them to NVMe SSDs. For users and library developers, the main storage interfaces are a key-value store and an array interface (SODA Foundation, 2023).

Locality for all operations is defined via $O(1)$ algorithms that determine metadata and data placement across storage devices participating in a pool. Metadata is always stored on NVRAM devices. Small writes are stored in NVRAM, while larger writes are directed to NVMe.

To ensure atomicity, all I/O requests can be wrapped in transactions that may be committed or canceled in batches. To support this and provide commit consistency, DAOS internally relies on a versioning object store (VOS). The VOS ensures that read requests return consistent data. For this purpose—and to support snapshots—objects are versioned using epoch numbers. Objects transition from one epoch to the next by combining all open transactions into a new consistent state.

The primary motivation for this approach is to avoid the worst-case assumptions of POSIX. This shift enables an optimistic model, where the I/O layer assumes that applications manage concurrent and overlapping I/O requests effectively.

Distributed Asynchronous Object Storage (DAOS) does not support immutable data directly; instead, a container or object can be opened in an immutable view of a previous epoch to represent a specific state. Data overwriting is supported through its transactional object store interface.

Node-local persistency is effectively achieved the moment data is stored. The use of NVRAM, combined with an appropriate redundancy scheme, makes it highly unlikely that transactions will be lost.

Characteristics for semantic categories:

- **Concurrent access:** DAOS has no notation for *shared* or *exclusive* access.
- **Persistency:** System level; dependent on the usage of storage class memory (SCM) or DRAM.
- **Consistency:** Commit; uses transactions for consistency.
- **Spatiality:** Location are defined by pool settings.
- **Temporality:** No notion of temporality.
- **Mutability:** Via transactions and epoch-based views.

Lustre: The Lustre parallel file system is widely used in HPC centers around the world. It has a long development history and provides a classical POSIX interface, and therefore, POSIX interface semantics. The challenges of POSIX semantics and scalability are well-known in the HPC community and among Lustre developers. Consequently, many features aimed at optimizing limitations imposed by POSIX semantics are present in current releases of Lustre.

Consistency is semantically sequential, and as with POSIX, there is no mechanism to explicitly express concurrent access. Lustre distributes data across multiple physical devices; files stored on these object storage targets (OSTs) are typically *striped* across multiple OSTs. This allows for higher bandwidth when reading or writing in parallel. Striping in Lustre also enables allocation of a file on specific storage *pools*. Pools group OSTs using the same storage technology and therefore exhibit similar performance characteristics. This provides a notion of spatiality, as the application or middleware can choose, for example, to store data on an NVMe- or disk-based pool.

With persistent client caches, Lustre can also cache data on local storage resources of a compute client, such as local SSDs (Qian et al., 2019). Additionally, features like DOM (Data on Metadata) help accelerate small file access by storing the contents of small files directly on the MDT (Metadata Target) instead of using OSTs (Fragalla et al., 2020).

The size and number of OSTs used to store stripes are configurable via a separate tool (*lfs*) or through the *llapi* interface. Besides stripe configuration, the Lustre llapi provides functions to create or open a file with a specific stripe configuration. Using `llapi_ladvise`, applications can provide I/O hints on a Lustre file to the server. In this way, applications can express *temporality* to the server and specify whether data should be pre-fetched into the server cache. This is the server-side equivalent of the `posix_fadvise` function, which operates on the client side.

Characteristics for semantic categories:

- **Concurrent access:** Lustre uses POSIX interface (no notation). But, through striping the distribution of the file can be adjusted to support parallel access.
- **Persistency:** System level.
- **Consistency:** Sequential consistency.
- **Spatiality:** Supports explicit data placement, striping data accross multiple OSTs, different storages are addressable via pools.
- **Temporality:** Via `posix_fadvise` on the client and `llapi_ladvise` on the server.
- **Mutability:** Via POSIX open *flags*.

GekkoFS: GekkoFS is a job-temporal *ad-hoc* file system that utilizes node-local storage resources and runs in user space (Vef et al., 2020). It is designed to boost HPC applications by leveraging local storage resources such as NVMe or SSDs and benefiting from lower latencies. GekkoFS provides a POSIX interface with sequential consistency semantics for any file system operation that accesses a specific file or data region within a file. This includes `read` and `write` operations, as well as metadata operations that target a single file—e.g., file creation—which are sequentially consistent. However, consistency for directory operations or those involving an unknown number of files beforehand (e.g., `readdir`) is relaxed to eventual consistency.

Moreover, studies on HPC application behavior have shown that certain file system operations, such as `move` and `rename`, are rarely used (Wang et al., 2021; Lensing et al., 2016). As GekkoFS is designed to implement only the essential POSIX operations commonly used in HPC, it either does not support these operations or makes them optionally supported, such as `rename`.

Data are stored in stripes across the node-local storage devices, and metadata is distributed across all file system clients in a key-value store. This design enables fast metadata operations and good scalability (Vef et al., 2020). In contrast to Lustre, GekkoFS does not support explicit data placement, and stripe settings cannot be adjusted at runtime. Consequently, system and middleware libraries are unable to utilize spatiality hints for optimizing data placement.

GekkoFS provides job-level persistency but does not guarantee data survival beyond the job's execution unless explicitly staged out. It does not expose runtime APIs for declaring reuse intent in any form.

GekkoFS, with its job-temporal lifetime and relaxed semantics, is often used as an I/O accelerator before accessing other storage systems. Because its configuration is done explicitly per job, GekkoFS does not support many customization options tailored to specific workloads.

Characteristics for semantic categories:

- **Concurrent access:** GekkoFS uses a POSIX interface; no explicit notion.
- **Persistency:** Job level.
- **Consistency:** sequential consistency for direct operations and eventual consistency for directory operations.
- **Spatiality:** Support for striping data but no explicit data placement.
- **Temporality:** No notion.
- **Mutability:** Via POSIX open *flags*.

5.4 Physical medium I/O interfaces

Storage hardware interface protocols include Small Computer System Interface (SCSI), Serial Advanced Technology Attachement (SATA), NVMe (NVM Express), and the DRAM interface for non-volatile DRAM. These can be grouped into two categories: the DRAM interface and all other storage interfaces.

DRAM is directly accessed by the central processing unit (CPU) via load/store commands and is byte-addressable. The memory controller of the CPU, if implemented, is capable of ordering and batching requests. In HPC scenarios, it will always load and store complete cache lines. Conflicting commands to the same memory address issued from different threads are managed by the cache coherence protocol.

The operating system kernel manages the remaining interfaces by issuing commands that typically target a specific device and storage block, usually with a fixed granularity. These commands are primarily Advanced Technology Attachement (ATA) commands or their derivatives. Modern storage devices allow the transmission of many commands through parallel queues. Historically, rotating disks could only reorder commands to optimize physical access patterns. Modern NVMe devices, however, may execute commands in any order. According to Section 2.1.2 of the NVM Express command set specifications (NVM Express, Inc., 2021), a write to position X and a read from position X issued in parallel may be executed in any order.

Any enforcement of request ordering in storage must be handled by the operating system or higher software layers, typically by flushing the relevant queues. In terms of consistency semantics, modern hardware essentially operates under eventual consistency. There is no inherent atomicity in storage hardware—each command is treated independently by the device, without regard to other commands in flight. While some devices may support WORM (write once, read many) functionality, this is generally not the case in HPC environments.

Characteristics for semantic categories:

- **Concurrent access:** No notion of concurrent access.
- **Persistency:** Provides persistency.
- **Consistency:** Eventual consistency; requests are processed simultaneously in several queues, the operating system/file system has to ensure consistency and ordering requirements of the requests.
- **Spatiality:** Data are stored on the device.
- **Temporality:** No notion of temporality.
- **Mutability:** No notion of mutability; this is usually a feature of the firmware.

5.5 Comparison of interface I/O semantics

Table 2 compares the different interfaces at each layer for their ability to express the characteristics of our semantic categories from Section 4.

We can conclude that middleware libraries provide the most flexibility in expressing I/O semantics. A common design pattern is to provide an interface within the middleware to implement I/O operations for different I/O backends, e.g., POSIX, object stores, databases, or in-memory storage systems. This way, the middleware can map its semantics to different underlying layers in the storage stack and possibly skip the system layer.

MPI-IO is the only interface that can explicitly express parallel access to a resource and provide coordination of I/O operations that are collectively applied to a file. HDF5 can use MPI-IO for its I/O operations if it is layered on top of MPI-IO. At lower layers, information about parallel access is lost, because neither the system-level nor the storage system interfaces are able to convey this information.

Persistency semantics are solely defined by the storage layer; higher-layer interfaces provide no notion for specifying their persistency requirements. However, especially for distributed storage systems, it would be beneficial to propagate persistency requirements to guide data placement and ensure data safety.

Consistency semantics are mostly sequential at the upper layers. MPI-IO provides a mode in which sequential consistency can be relaxed to *sync/barrier/sync* semantics, which resemble session consistency. In the storage layer, consistency appears to be the main target for optimization regarding I/O semantics. Different file systems implement various consistency models and rely on applications or middleware to be aware of them. It has been shown that HPC applications can run under relaxed consistency semantics (Wang et al., 2021; Oeste et al., 2023). Moreover, storage systems with relaxed consistency models can deliver better performance for I/O (Vef et al., 2020; Wang et al., 2024).

For spatiality semantics, we distinguish between the intended access pattern and the ability to express explicit

TABLE 2 Comparison of the semantics of I/O interfaces.

Interface	Concurrent Access	Persistency	Consistency	Spatiality	Temporality	Mutability
MPIIO	✓	×	Sequential/Session	✓	✓	✓
HDF5	✓	×	Sequential	✓	✓	✓
POSIX I/O	×	×	Sequential	×	✓	✓
NFS	×	Job level	Session	×	×	✓
DAOS	×	System level	Commit	✓	×	✓
Lustre	×	System level	Sequential	✓	✓	✓
GekkoFS	×	Job level	Sequential/Eventual	×	×	✓
NVME SSD	×	✓	Eventual	×	×	×

✓ indicates the transport of this semantic category is supported by the interface. × indicates that there is no support for this semantic category in the interface.

data placement at the storage layer. Our study shows that middleware libraries such as HDF5 and MPI-IO can provide spatiality-related hints to the lower layers. However, if they are layered on top of POSIX, this information might get lost. Storage systems, by comparison, can support spatiality by grouping storage devices into pools and defining stripe patterns to guide data distribution.

Temporality, expressed as hints to guide caching, is supported by all middleware libraries and by POSIX I/O, but not all storage systems are capable of utilizing this information.

All interfaces, except NVMe SSDs, can express mutability semantics to specify whether they merely read or also modify a resource.

We conclude with the following summary:

1. Middleware libraries are able to express most semantics through their APIs and can be extended to support additional semantics.
2. Persistency is defined solely by the storage layer; higher layers do not support expressing persistency requirements.
3. MPI-IO is the only interface that allows the expression of parallel access to a file.
4. Consistency semantics are defined individually by each interface. There is no standardized way to express required consistency guarantees.
5. System-level interfaces offer limited support for expressing I/O semantics; they are primarily designed for single-node use.
6. Storage systems focus primarily on persistency and consistency semantics but offer a wide range of features in these areas.

In this section, each interface was discussed independently. In the next section, we will explore how I/O semantics are lost when interfaces are layered together, as they are in real-world I/O stacks.

6 Semantic losses through the I/O stack

When data move through the different layers of the I/O stack, relevant I/O semantics attached to them can become less apparent or be lost. This section takes the checkpoint example from Section 3 and discusses which I/O semantics might be lost through the layers of the different interfaces in our three stacks from Figure 3.

At the application level, information about I/O semantics can be assumed to be complete. The user's intent should be directly expressed in the source code. For all three stacks in Figure 3, the application must define the storage path for the checkpoint data. The persistency semantics are defined by the storage system associated with that path. As shown in Section 5, none of the system or middleware interfaces can express this information. Consequently, the underlying file system cannot report whether it guarantees the required persistency semantics. The persistency semantics of that data are lost at the application level, and the user must know which file system meets their persistency requirements. In Section 3, we described that the checkpoint should be used to recover from potential node failures. The I/O stack from Figure 3c cannot satisfy this requirement without additional data transfers because GekkoFS is a job-temporal file system that uses node-local storage. Consequently, data cannot be recovered after a node failure unless they are transferred to a storage location with stronger persistency semantics. If persistency semantics had been transported through the stack, at least a warning could have been issued.

For the I/O stack in Figure 3a, the middleware layer translates the HDF5 memory layout into a layout expressible by lower-layer primitives. HDF5 on top of MPI-IO can coordinate multiple processes accessing shared files. MPI-IO can transport the concurrent access semantics of the application by defining the number of processes participating in I/O operations within a corresponding MPI communicator for collective calls. HDF5 provides additional functions to configure the collective behavior of MPI-IO. Spatiality semantics can be expressed through information about data access structure using MPI-derived data types and MPI file views. Features like HDF5 *dataset* and *dataspace* allow defining a view of the data to be written. Further, MPI-IO can use `posix_fadvise` to inform lower layers about intended temporal and spatial access patterns. Additionally, MPI-IO can invoke Lustre-specific API calls (e.g., to create a file with an optimized stripe pattern) to guide explicit data placement.

For the I/O stack in Figure 3c, spatiality semantics are lost at the application level. POSIX I/O has no mechanism to transport data placement information to the underlying file system. Furthermore, GekkoFS neither implements `posix_fadvise` nor provides a public interface for controlling data distribution. In contrast, the I/O stack in Figure 3b uses Lustre, which supports explicit data placement and `posix_fadvise`, but the application must use them correctly.

The translation to the system layer happens when MPI-IO converts I/O operations into POSIX I/O calls. Because POSIX I/O has no concept of parallel file access, the upper layer must coordinate I/O operations across processes. At this point, concurrent access semantics are lost. Processes perform I/O independently as unstructured byte streams. For the first I/O stack in Figure 3a, MPI-IO handles this coordination. In the stacks from Figures 3b, c, the application is responsible for coordinating concurrent access, and the semantics are already lost at the application level. Information such as temporality and spatiality becomes less apparent. Although POSIX I/O includes the `fadvise` call to provide access hints to the underlying storage system, not all file systems support it.

For the two I/O stacks using Lustre (Figures 3a, b), temporality hints can be conveyed through `MPI_Info`, `posix_fadvise`, and `llapi_ladvise`. Thus, client caches can be disabled, as checkpoint data are not reused after being written. In contrast, the I/O stack in Figure 3c uses GekkoFS, which does not implement `posix_fadvise`, causing temporality semantics to be lost at this layer.

The sequential consistency semantics of POSIX I/O prohibit unsynchronized updates to the same file region. Because synchronization is expensive in lower I/O layers, most parallel file systems relax POSIX consistency semantics (see Section 5). MPI-IO can also relax consistency using a *sync/barrier/sync* session consistency model. Thus, multiple interfaces in an I/O stack may define different consistency models, but none can propagate requirements to lower layers. This can result in inconsistent data if a higher layer assumes a stronger model than the lower layer can provide, or in degraded performance if unnecessary synchronizations are enforced for weaker models. For example, writing to the same file with multiple processes over NFS (which offers close-to-open session consistency) may lead to undefined behavior. However, with MPI-IO as middleware, it can handle the necessary synchronizations and flushes, albeit at a performance cost. Relaxed consistency at the file system layer is based on the assumption that applications synchronize their I/O correctly. Without this, distributed file systems would require locking on every update, which would significantly degrade performance.

When I/O reaches the hardware layer, most semantics are no longer present. Requests wait in one of possibly many queues to be processed by the device firmware, with no indication of the number of processes they originated from. There is no concept of a file anymore, and the processing order of requests is not guaranteed. The hardware layer assumes all semantic requirements have been satisfied by higher layers. Semantics such as spatiality are irrelevant at this point, as the I/O has already reached its destination. The hardware layer is optimized for throughput and persistence, offering high-performance predictability but the least semantic awareness.

To summarize, persistency semantics are solely defined by the storage system and cannot be expressed by higher layers, making them effectively lost at the application level. In the first I/O stack (Figure 3a), concurrent access can be expressed by MPI-IO but is lost at the middleware layer. Spatiality, temporality, and mutability semantics can be propagated through the I/O stack using `MPI_Info`, `posix_fadvise`, and `llapi_ladvise`. In stacks that do not use MPI-IO (Figures 3b, c), concurrent access semantics are lost at the application level. In the third stack (Figure 3c), spatiality and temporality semantics are lost at the system-level. Mutability semantics are preserved across all three stacks down to the hardware. Consistency semantics are individually defined by each interface but are not propagated through the I/O stack. As a result, neither signaling unmet consistency requirements nor optimizing based on relaxed assumptions is possible. The user must know what guarantees can be assumed for a given I/O stack.

In the next section, Section 7, we discuss existing approaches the I/O community has taken to address the semantic gap. Following that, in Section 8, we present our proposal for a semantics-aware I/O stack.

7 Related work

In this section, we discuss some of the research on I/O semantics. We examine how I/O semantics can be analyzed and what has been done to address the challenges posed by the *semantic gap*.

Primarily, two papers have analyzed the I/O semantics of HPC applications. Wang et al. (2021) examined the consistency semantics requirements of 17 HPC applications on parallel file systems. They found that 16 out of 17 applications can utilize parallel file systems (PFSs) with weaker semantics than the strong consistency guarantees of POSIX. Moreover, they provide a categorization of the consistency guarantees offered by PFSs. In further work (Wang et al., 2024), they proposed a formal framework for storage consistency models and showed that weaker consistency models can improve I/O performance. We follow their categorization in our discussion of consistency semantics.

We extended Wang's investigation by developing a tool that groups I/O operations that could be executed in parallel and analyzes the semantics of both data and metadata operations (Oeste et al., 2023). Both studies concluded that most of the strong consistency semantics required by POSIX are not necessary for HPC applications. In particular, for data operations such as strict `write` consistency, enforcement at this layer appears unnecessary. However, we showed that parallel metadata operations, such as concurrent file creations in the same directory, are a common pattern in HPC applications that affects performance and scalability.

Regarding the challenges posed by the *semantic gap* between applications and storage systems, several approaches have been explored. We group them into three basic categories: (1) optimization of middleware libraries, (2) tuning of storage systems, and (3) development of automatic characterization and optimization systems.

The development and optimization of middleware libraries is an ongoing research topic—libraries such as HDF5 are continuously being extended. For example, to utilize the

DAOS storage system directly via the DAOS API (bypassing POSIX I/O), an HDF5 VOL connector was developed. This enables more features such as asynchronous I/O and better performance (Soumagne et al., 2022).

A second major area of effort is the optimization of storage systems, particularly file systems. A number of file systems relax POSIX semantics for performance or introduce specialized features to better support certain workloads. Content Addressable Parallel File System (CAPFS) is a file system with a tunable consistency framework that can be adapted to specific applications. The authors argue that a single consistency policy across the entire file system is suboptimal, as it cannot meet all application requirements simultaneously. CAPFS uses an optimistic concurrency control mechanism, assuming that conflicts are rare. However, the paper only considers semantics for file content and not metadata. Notably, CAPFS allows manipulation of semantics at runtime at various granularities, e.g., sub file, file, or partition-wide (Vilayannur et al., 2005).

The widely used Lustre file system is continually integrating new features to improve performance and mitigate the *semantic gap* in the I/O stack. Examples include writeback metadata caching (Qian et al., 2022), where a lightweight in-memory file system is used as a metadata write-back cache instead of the traditional write-through caching model. Another example is the enhancement of directory tree walks in Lustre (Qian et al., 2023) by prefetching metadata and using eventually consistent (lazy) updates to avoid additional RPCs and limitations from serialized system-level interfaces. Lustre's persistent client caching (LPCC) (Qian et al., 2019) is another feature that speeds up workloads by leveraging local SSDs. However, such features require interaction with the parallel file system through separate interfaces such as configuration options or additional tools like *lctl* and *lfs*. While meaningful defaults are possible with experienced administrators, dynamically optimizing I/O parameters and configurations based on the workload remains an open research problem.

The third area of active research involves autotuning strategies for storage systems based on the requirements and characteristics of different applications. The motivation comes from the complexity of selecting optimal parameters for heterogeneous storage systems.

Mimir is an approach for configuring layers in the I/O stack to align with user intent and enhance performance. In this model, user intent is interpreted and handled by Mimir as a proxy, rather than modifying each individual layer. The authors demonstrate use cases spanning high-level I/O libraries (e.g., MPI-IO, HDF5), POSIX-level interfaces, and middleware I/O libraries (Devarajan and Mohror, 2023).

LabStor is a modular platform for developing and deploying customizable high-performance I/O stacks in user space. As Logan et al. argue, current I/O stacks are too rigid to allow easy modification. Their solution involves a modular system, mostly running in user space, with a minimal kernel module for OS integration. This module can replace the internal kernel VFS (Logan et al., 2022).

LABIOS (Kougkas et al., 2019, 2020) is another approach aimed at supporting multiple I/O workloads with conflicting requirements under a single storage system. It provides a label-based architecture to promote flexibility, versatility, agility, and malleability in storage. While the authors demonstrate improved I/O performance and overall execution time, the architecture introduces a parallel system to the current I/O stack to overcome existing interface limitations.

8 Toward a semantic aware I/O stack

Our analysis shows that semantic information—such as consistency, temporality, and concurrency intent—is often lost as I/O operations traverse the layers of an HPC storage stack. This limits the ability of lower layers to make informed decisions that could optimize performance, enforce correctness, or provide user-level guarantees.

It becomes apparent that higher layers such as the application and middleware layers are able to express more I/O semantics through their APIs than the system layer. Storage systems like parallel file systems could perform several optimizations based on I/O semantics if they had information about the application's needs. For example, some locking could be disabled when weaker consistency is sufficient for a workload, or more extensive caching could be enabled when a process accesses a file exclusively.

Transporting such information through the I/O stack is challenging, especially when middleware libraries need to utilize system-level interfaces to interact with the storage system. This forces storage systems to make assumptions about access patterns and application behavior. Because storage systems are primarily designed to reliably store data in a consistent manner, they often have to make more pessimistic assumptions than necessary, which negatively impacts performance. We call this the *semantic gap* in the I/O stack.

To address this, we conceptually outline what a semantics-aware I/O stack could look like. Figure 4 illustrates where we would place such an semantics-aware interface in the I/O stack.

This is not intended as a concrete implementation proposal, but rather as a design-oriented thought experiment that builds on our comparative analysis and semantic taxonomy. Such a stack would allow I/O interfaces at higher levels (e.g., application or middleware) to annotate operations with explicit semantic descriptors—using the six categories defined in Section 4. These descriptors would then be propagated through the stack, allowing lower layers to adapt behavior based on declared intent. For example:

- A parallel file system might skip costly locking if it knows an application guarantees exclusive access (concurrency).
- A caching layer could adjust pre-fetching and eviction strategies based on reuse expectations (temporality).
- A job-temporal file system might trigger a warning or stage-out policy if system-level persistency is required but not supported.

Semantics-aware interfaces could adopt a declarative model, where intent is conveyed alongside operations via metadata,

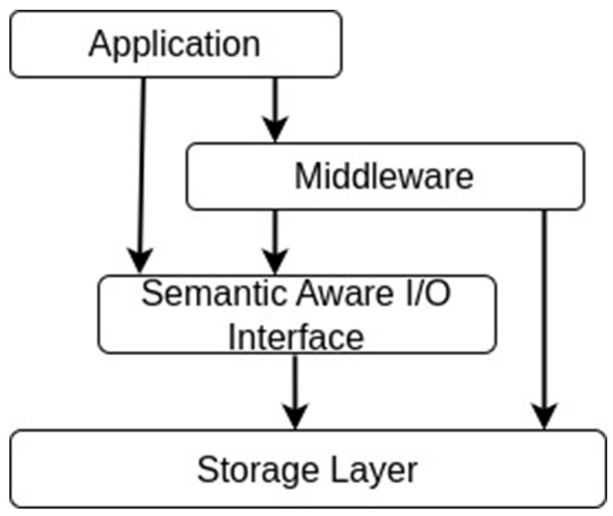

FIGURE 4
Illustration of a semantic-aware I/O stack where the middleware directly interfaces with the storage system.

configuration hints, or API extensions. Crucially, we emphasize that our goal is not to propose a new interface, but to motivate the need for semantic transparency based on the gaps identified in existing interfaces. The taxonomy we introduced can serve as a foundation for evaluating and guiding enhancements to current APIs, middleware, or file systems.

By better surfacing and transporting semantic intent, future I/O stacks could close the gap between user behavior and system behavior—improving both performance and correctness in HPC workflows.

9 Summary and conclusions

In this article, we presented a comparison of I/O interfaces used in HPC storage stacks, focusing on how these interfaces express and handle I/O semantics. To support this analysis, we introduced a taxonomy that categorizes I/O semantics into six key dimensions: concurrency, persistency, consistency, spatiality, temporality, and mutability. By applying this taxonomy to widely used interfaces such as POSIX, MPI-IO, HDF5, and others, we demonstrated how semantic information is often lost as data flow through the layers of the I/O stack. Our comparison revealed that while middleware layers can express rich semantic intent, lower layers—particularly system and hardware interfaces—typically lack mechanisms to receive or act on this information.

We aimed to clarify and compare the semantic capabilities of existing interfaces through a structured framework. This comparative approach provides a foundation for future work on improving semantic propagation across I/O stacks and for designing tools or extensions that make semantic intent more transparent and actionable.

By exposing where and how semantic loss occurs, our work supports the broader effort to build more performant and predictable I/O systems in HPC by better understanding and classifying the interfaces we use.

Data availability statement

The original contributions presented in the study are included in the article/supplementary material, further inquiries can be directed to the corresponding author.

Author contributions

SO: Conceptualization, Investigation, Methodology, Writing – original draft, Writing – review & editing. PH: Investigation, Writing – original draft, Writing – review & editing. MK: Methodology, Writing – original draft, Writing – review & editing. JK: Conceptualization, Methodology, Writing – original draft, Writing – review & editing.

Funding

The author(s) declare that financial support was received for the research and/or publication of this article. The research in this paper was conducted as part of the MCSE project (Funding code: 16ME0663K) which was funded by BMFTR (Federal Ministry of Research, Technology and Space) as part of the SCALEXA funding guideline, which in turn was funded by the European Union - NextGenerationEU. This work was supported by the NHR Center of TU Dresden, which is jointly financed by the BMFTR and the free state of Saxony.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

References

Ahmadian, S., Salkhordeh, R., Mutlu, O., and Asadi, H. (2021). ETICA: efficient two-level I/O caching architecture for virtualized platforms. *CoRR, abs/2106.07423.* doi: 10.1109/TPDS.2021.3066308

Barton, E. (2015). *DAOS–An Architecture for Extreme Scale Storage.* SNIA (Storage Networking Industry Association). Available online at: https://www.snia.org/sites/default/files/SDC15_presentations/dist_sys/EricBarton_DAOS_Architecture_Extreme_Scale.pdf

Bonnie, M. M. D., Ligon, B., Marshall, M., Ligon, W., Mills, N., Sampson, E. Q. S., et al. (2011). "Orangefs: advancing pvfs," in *USENIX Conference on File and Storage Technologies (FAST).*

Braam, P. J., and Zahir, R. (2002). Lustre: a scalable, high performance file system. *Cluster File Syst.* 8, 3429–3441.

Byna, S., Chen, Y., Sun, X.-H., Thakur, R., and Gropp, W. (2008). "Parallel I/O prefetching using MPI file caching and I/O signatures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (IEEE Press), 44. doi: 10.1109/SC.2008.5213604

Carns, P. H., Ligon, W. B., Ross, R. B., and Thakur, R. (2000). "Pvfs: a parallel file system for linux clusters," in *Proceedings of the 4th Annual Linux Showcase Conference - Volume 4, ALS'00* (USA: USENIX Association), 28.

Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J.-P., et al. (1996). "Overview of the mpi-io parallel I/O interface," in *Input/Output in Parallel and Distributed Computer Systems*, 127–146. doi: 10.1007/978-1-4613-1401-1_5

del Rosario, J. M., Bordawekar, R., and Choudhary, A. (1993). Improved parallel I/O via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News* 21, 31–38. doi: 10.1145/165660.165667

Devarajan, H., and Mohror, K. (2023). "Mimir: extending I/O interfaces to express user intent for complex workloads in HPC," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 178–188. doi: 10.1109/IPDPS54959.2023.00027

Dinh, A., Wang, J., Wang, S., Chen, G., Chin, W.-N., Lin, Q., et al. (2017). UStore: a distributed storage with rich semantics. *arXiv preprint arXiv:1702.02799.*

Folk, M., Heber, G., Koziol, Q., Pourmal, E., and Robinson, D. (2011). "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, 36–47. doi: 10.1145/1966895.1966900

Fragalla, J., Loewe, B., and Kling Petersen, T. (2020). New lustre features to improve lustre metadata and small-file performance. *Concurr. Comput.* 32:e5649. doi: 10.1002/cpe.5649

Fridella, S., Black, D. L., and Glasgow, J. (2010). *Parallel NFS (pNFS) Block/Volume Layout.* Request for Comments RFC 5663, Internet Engineering Task Force.

Gregg, B. (2014). *Systems Performance: Enterprise and the Cloud.* London: Pearson Education.

Hildebrand, D., Nisar, A., and Haskin, R. (2009). "pNFS, POSIX, and MPI-IO: a tale of three semantics," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage* (Portland Oregon: ACM), 32–36. doi: 10.1145/1713072.1713082

Kang, Q., Ross, R., Latham, R., Lee, S., Agrawal, A., Choudhary, A., et al. (2020). "Improving all-to-many personalized communication in two-phase I/O," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–13. doi: 10.1109/SC41405.2020.00014

Kougkas, A., Devarajan, H., Lofstead, J., and Sun, X.-H. (2019). "Labios: a distributed label-based I/O system," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC'19* (New York, NY, USA: Association for Computing Machinery), 13–24. doi: 10.1145/3307681.3325405

Kougkas, A., Devarajan, H., and Sun, X.-H. (2020). Bridging storage semantics using data labels and asynchronous I/O. *ACM Trans. Storage* 16, 1–34. doi: 10.1145/3415579

Koziol, Q., and Breitenfeld, S. (2015). *A Brief Introduction to Parallel HDF5.* The HDF Group. Available online at: https://www.alcf.anl.gov/files/Parallel_HDF5_1.pdf

Kuhn, M. (2013). "A semantics-aware I/O interface for high performance computing," in *Supercomputing*, eds. J. M. Kunkel, T. Ludwig, and H. W. Meuer (Berlin, Heidelberg: Springer Berlin Heidelberg), 408–421. doi: 10.1007/978-3-642-38750-0_31

Lee, W., Park, S., Sung, B., and Park, C. (2011). *Improving Adaptive Replacement Cache (arc) by reuse distance.* Technical report.

Lensing, P. H., Cortes, T., Hughes, J., and Brinkmann, A. (2016). "File system scalability with highly decentralized metadata on independent storage devices," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 366–375. doi: 10.1109/CCGrid.2016.28

Liang, Z., Lombardi, J., Chaarawi, M., and Hennecke, M. (2020). "Daos: a scale-out high performance storage stack for storage class memory," in *Supercomputing Frontiers*, eds. D. K. Panda (Cham: Springer International Publishing), 40–54. doi: 10.1007/978-3-030-48842-0_3

Liao, W., k., Ching, A., Coloma, K., Choudhary, A., and Ward, L. (2007). "An implementation and evaluation of client-side file caching for MPI-IO," in *2007 IEEE International Parallel and Distributed Processing Symposium*, 1–10. doi: 10.1109/IPDPS.2007.370239

Lockwood, G. K. (2017). *What's So Bad About POSIX I/O?* The Next Platform. Available online at: https://www.nextplatform.com/2017/09/11/whats-bad-posix-io/ (Accessed July 29, 2025).

Lofstead, J., Zheng, F., Klasky, S., and Schwan, K. (2009). "Adaptable, metadata rich IO methods for portable high performance IO," in *2009 IEEE International Symposium on Parallel Distributed Processing* (Rome, Italy: IEEE), 1–10. doi: 10.1109/IPDPS.2009.5161052

Logan, L., Garcia, J. C., Lofstead, J., Sun, X., and Kougkas, A. (2022). "LabStor: a modular and extensible platform for developing high-performance, customized I/O stacks in userspace," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis* (Dallas, TX, USA: IEEE), 1–15. doi: 10.1109/SC41404.2022.00028

Message Passing Interface Forum (1996). *MPI-2: Extensions to the message-passing interface.* Technical report. University of Tennessee, Knoxville.

Message Passing Interface Forum (2023). *MPI: A message-passing interface standard - version 4.1.* MPI-Forum.

NVM Express, Inc. (2021). *NVM Command Set Specification, Revision 1.0a.* NVM Express, Inc. Available online at: https://nvmexpress.org/wp-content/uploads/NVMe-NVM-Command-Set-Specification-1.0a-2021.07.26-Ratified.pdf

Oeste, S., Kluge, M., Tschüter, R., and Nagel, W. E. (2023). "Analyzing parallel applications for unnecessary I/O semantics that inhibit file system performance," in *High Performance Computing*, eds. A. Bienz, M. Weiland, M. Baboulin, and C. Kruse (Cham: Springer Nature Switzerland), 161–176. doi: 10.1007/978-3-031-40843-4_13

Padua, D., Ghoting, A., Gunnels, J. A., Squillante, M. S., Meseguer, J., Cownie, J. H., et al. (2011). "MPI-IO," in *Encyclopedia of Parallel Computing*, eds. D. Padua (Boston, MA: Springer US), 1191–1199. doi: 10.1007/978-0-387-09766-4

Pawlowski, B., Shepler, S., Beame, C., Callaghan, B., Eisler, M., Noveck, D., et al. (2000). "The NFS Version 4 Protocol," in *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)* (Amsterdam, NL: NLUUG).

Pence, W. D., Chiappetti, L., Page, C. G., Shaw, R. A., and Stobie, E. (2010). Definition of the flexible image transport system (fits), version 3.0. *Astron. Astrophys.* 524:A42. doi: 10.1051/0004-6361/201015362

posix (2018). *IEEE Standard for Information Technology-Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7.* IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008), 1–3951.

Qian, Y., Cheng, W., Zeng, L., Li, X., Vef, M.-A., Dilger, A., et al. (2023). "Xfast: extreme file attribute stat acceleration for lustre," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '23* (New York, NY, USA: Association for Computing Machinery). doi: 10.1145/3581784.3607080

Qian, Y., Cheng, W., Zeng, L., Vef, M.-A., Drokin, O., Dilger, A., et al. (2022). "Metawbc: posix-compliant metadata write-back caching for distributed file systems," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–20. doi: 10.1109/SC41404.2022.00061

Qian, Y., Li, X., Ihara, S., Dilger, A., Thomaz, C., Wang, S., et al. (2019). "Lpcc: hierarchical persistent client caching for lustre," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19* (New York, NY, USA: Association for Computing Machinery). doi: 10.1145/3295500.3356139

Rew, R., and Davis, G. (1990). Netcdf: an interface for scientific data access. *IEEE Comput. Graph. Applic.* 10, 76–82. doi: 10.1109/38.56302

Scargall, S. (2020). *Programming Persistent Memory: A Comprehensive Guide for Developers.* Berkeley, CA: Apress. doi: 10.1007/978-1-4842-4932-1

SODA Foundation. (2023). *DAOS source code.* GitHub. Available online at: https://github.com/daos-stack/daos/tree/master/src (Accessed October 2, 2023).

Soumagne, J., Henderson, J., Chaarawi, M., Fortner, N., Breitenfeld, S., Lu, S., et al. (2022). Accelerating hdf5 I/O for exascale using daos. *IEEE Trans. Paral. Distr. Syst.* 33, 903–914. doi: 10.1109/TPDS.2021.3097884

Tang, H., Koziol, Q., Byna, S., Mainzer, J., and Li, T. (2019). "Enabling transparent asynchronous I/O using background threads," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, 11–19. doi: 10.1109/PDSW49588.2019.00006

Tang, H., Koziol, Q., Ravi, J., and Byna, S. (2022). Transparent asynchronous parallel I/O using background threads. *IEEE Trans. Paral. Distr. Syst.* 33, 891–902. doi: 10.1109/TPDS.2021.3090322

Tang, K., Huang, P., He, X., Lu, T., Vazhkudai, S., and Tiwari, D. (2017). "Toward managing HPC burst buffers effectively: draining strategy to regulate bursty I/O behavior," in *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (IEEE), 87–98. doi: 10.1109/MASCOTS.2017.35

Thakur, R., Gropp, W., and Lusk, E. (1996). "An abstract-device interface for implementing portable parallel-I/O interfaces," in *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)* (Annapolis, MA, USA: IEEE Comput. Soc. Press), 180–187. doi: 10.1109/FMPC.1996.558080

Thakur, R., Gropp, W., and Lusk, E. (1999a). "Data sieving and collective I/O in ROMIO," in *Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation* (Annapolis, MD, USA: IEEE), 182–189. doi: 10.1109/FMPC.1999.750599

Thakur, R., Gropp, W., and Lusk, E. (1999b). "On implementing MPI-IO portably and with high performance," in *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems* (Atlanta Georgia USA: ACM), 23–32. doi: 10.1145/301816.301826

The HDF Group (2012). *Enabling a Strict Consistency Semantics Model in Parallel HDF5*. The HDF Group. Available online at: https://support.hdfgroup.org/HDF5/doc/Advanced/PHDF5FileConsistencySemantics/PHDF5FileConsistencySemantics.pdf

Vef, M.-A., Moti, N., Süß, T., Tacke, M., Tocci, T., Nou, R., et al. (2020). Gekkofs—a temporary burst buffer file system for hpc applications. *J. Comput. Sci. Technol.* 35, 72–91. doi: 10.1007/s11390-020-9797-6

Vilayannur, M., Lang, S., Ross, R., Klundt, R., Ward, L., and Snl (2008). *Extending the POSIX I/O interface: A parallel file system perspective*. Technical Report ANL/MCS-TM-302,946036. doi: 10.2172/946036

Vilayannur, M., Nath, P., and Sivasubramaniam, A. (2005). "Providing tunable consistency for a parallel file store," in *FAST*. USENIX Association.

Wang, C., Mohror, K., and Snir, M. (2021). "File system semantics requirements of hpc applications," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '21* (New York, NY, USA: Association for Computing Machinery), 19–30. doi: 10.1145/3431379.3460637

Wang, C., Mohror, K., and Snir, M. (2024). Formal definitions and performance comparison of consistency models for parallel file systems. *IEEE Trans. Paral. Distr. Syst.* 35, 1092–1106. doi: 10.1109/TPDS.2024.3391058

Yu, J., Yang, W., Wang, F., Dong, D., Feng, J., and Li, Y. (2020). Spatially bursty I/O on supercomputers: causes, impacts and solutions. *IEEE Trans. Paral. Distr. Syst.* 31, 2908–2922. doi: 10.1109/TPDS.2020.3005572

Zheng, H., Vishwanath, V., Koziol, Q., Tang, H., Ravi, J., Mainzer, J., et al. (2022). "HDF5 Cache VOL: efficient and scalable parallel I/O through caching data on node-local storage," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* (Taormina, Italy: IEEE), 61–70. doi: 10.1109/CCGrid54584.2022.00015