File System Semantics Requirements of HPC Applications

Chen Wang
University of Illinois at Urbana-Champaign
Champaign, US
chenw5@illinois.edu

Kathryn Mohror
Lawrence Livermore National Laboratory
Livermore, US
kathryn@llnl.gov

Marc Snir
University of Illinois at Urbana-Champaign
Champaign, US
snir@illinois.edu

ABSTRACT

Most widely-deployed parallel file systems (PFSs) implement POSIX semantics, which implies sequential consistency for reads and writes. Strict adherence to POSIX semantics is known to impede performance and thus several new PFSs with relaxed consistency semantics and better performance have been introduced. Such PFSs are useful provided that applications can run correctly on a PFS with weaker semantics. While it is widely assumed that HPC applications do not require strict POSIX semantics, to our knowledge there has not been systematic work to support this assumption. In this paper, we address this gap with a categorization of the consistency semantics guarantees of PFSs and develop an algorithm to determine the consistency semantics requirements of a variety of HPC applications. We captured the I/O activity of 17 representative HPC applications and benchmarks as they performed I/O through POSIX or I/O libraries and examined the metadata operations used and their file access patterns. From this analysis, we find that 16 of the 17 applications can utilize PFSs with weaker semantics.

CCS CONCEPTS

• **Software and its engineering** → *File systems management*; *Input / output*; *Consistency*.

KEYWORDS

consistency semantics, parallel file system, scientific applications

ACM Reference Format:
Chen Wang, Kathryn Mohror, and Marc Snir. 2021. File System Semantics Requirements of HPC Applications. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '21), June 21–25, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3431379.3460637

1 INTRODUCTION

High performance computing (HPC) systems host parallel applications composed of hundreds to tens of thousands of tightly-coupled processes that typically run for hours or days. The I/O needs of these applications are supported by parallel file systems (PFSs), such as Lustre [15], BeeGFS [28] and GPFS [58]. These PFSs aggregate parallel data and metadata servers to provide high capacity and high bandwidth, even for concurrent access to a single file by the processes of a highly-parallel application, where the file data can be striped across the data servers of the PFS.

HPC applications can access the PFS directly via the POSIX file system API, however they often utilize higher-level I/O libraries specially designed for scientific I/O. For example, MPI-IO [19] supports collective I/O operations, where groups of processes use the API to concurrently execute a read or write operation. As an optimization, MPI-IO servers can perform global write buffering and aggregation to match the I/O pattern of clients to the layout of data on the data servers. Libraries such as HDF5 [23] and ADIOS [24, 44] provide higher-level storage management capabilities. For example, HDF5 provides its own directory structure, with files being replaced with typed, multidimensional numerical arrays called datasets. I/O libraries may be layered, e.g., HDF5 can be layered on top of MPI-IO to enable collective access to datasets; and, in turn, MPI-IO can be layered on top of POSIX. This layering generates complex I/O access patterns that may differ greatly from the I/O access patterns one would deduce from examining the scientific application code.

While PFSs can support high read and write bandwidths under ideal conditions, their effective performance can vary significantly depending on the I/O access patterns of applications, the PFS configuration, and on interference from other concurrently running applications [13, 27, 43]. A major impediment to PFS performance is the strict adherence to POSIX semantics, which requires sequential consistency in general and atomicity for many operations [62, 68]. The strict enforcement of these requirements impedes caching, generates significant additional traffic, and results in congestion in situations of high sharing, especially for small block reads and writes [42]. In order to avoid these performance issues, HPC I/O researchers have developed PFSs with relaxed semantics, such as UnifyFS [37], PLFS [12], Gfarm/BB [63], and GekkoFS [66], and have demonstrated significant performance improvements.

Despite the greatly improved I/O performance demonstrated by these relaxed-semantics PFSs, there remain several unsolved issues regarding their ability to correctly and efficiently support HPC applications: (a) It is not generally known a priori whether an application will run correctly on a PFS with weaker semantics. (b) It is challenging to determine the semantics needed by an application since I/O patterns depend on execution flow and on the behavior of high-level I/O libraries. (c) PFSs relax POSIX semantics in different ways which reduces the portability of applications across PFSs. (d) There are no accepted categorizations or definitions of the relaxed semantics implemented by PFSs for applications or I/O libraries to target. All in all, the lack of information leads to conservative PFS choices by HPC system designers, possibly leading to unnecessarily reduced I/O performance by many applications.

HPDC '21, June 21–25, 2021, Virtual Event, Sweden

ACM ISBN 978-1-4503-8217-5/21/06…$15.00
https://doi.org/10.1145/3431379.3460637

These critical, open issues show that there is a clear gap in our knowledge of application consistency semantics requirements and the relaxed consistency models of PFSs. In this paper, we close this gap by providing a method for testing consistency requirements of applications based on the possibility of access conflicts under weaker semantics. In addition, we develop a categorization of the consistency models of PFSs to serve as a basis for describing the semantics offered by PFSs with relaxed semantics. The contributions of this paper are summarized as follows:

- We present the I/O characteristics of 17 representative HPC applications and benchmarks using POSIX or I/O libraries. While our study focuses on I/O patterns that are relevant to the PFS semantic model, we also study access patterns that are important for the understanding of I/O performance and the metadata operations used by these applications.
- We develop a method for detecting I/O accesses that can cause conflicts under weaker consistency models.
- We provide terminology for the categorization of the consistency semantics of PFSs.

To our knowledge, this paper is the first work on determining the I/O consistency semantics requirements of parallel applications. Our results can provide critical insights for file system developers on which optimizations can be utilized by HPC applications and for HPC users who need to choose the appropriate PFS for their application's I/O requirements for correctness and performance.

The rest of this paper is organized as follows. Section 2 provides background and related work. In Section 3, we describe our categorization of PFS consistency models. Section 4 introduces definitions used for detecting conflicting I/O accesses. In Section 5, we present our algorithm for detecting conflicting accesses. Section 6 reports our results and analysis. Finally, we discuss consequences of our work and future work in Section 7.

2 BACKGROUND AND RELATED WORK

Here, we provide information on HPC applications, I/O behavior studies, the POSIX I/O interface and its consistency semantics, and POSIX and near-POSIX PFS efforts.

2.1 HPC Applications and I/O Behavior

HPC applications are highly-parallel, often with tens of thousands of processes working concurrently to simulate physical phenomena. Scientific applications tend to have regular I/O patterns due to their typical 3-phase structure: initialization, time step computation, and finalization. During initialization, parallel processes read in input files, consisting of initial data and simulation configuration information. In the computation phase, the processes loop through a series of "time steps", where in each time step, the phenomenon is simulated for some time delta, after which all parallel processes synchronize using a communication library and optionally write data to a file, either a checkpoint that can be used for recovery, or a snapshot of the current simulation state for further analysis. During the finalization phase, processes will write final data to files. The number of files accessed in parallel varies across applications, but it is common for processes in an HPC application to concurrently access a single shared file or a set of shared files in an I/O phase.

Many researchers have studied the I/O behavior of HPC applications and have noted the regularity of I/O requests [47, 53, 54, 70, 72]. These researchers collected I/O trace and profile information from application runs and analyzed them to discover patterns. In general, the researchers concluded that scientific applications share common I/O properties such as sequential file access, initial and final phases of compulsory I/O, and bursts of high-volume I/O activity at regular intervals during computation. Other researchers have focused on I/O measurement for the purpose of improving performance [18, 60]. For example, Carns et al. characterized the I/O behaviour of several scientific applications and found potential I/O performance issues of those applications, such as a large number of small writes. In contrast to these application-level studies, several efforts have examined the I/O behavior of applications at the system level [46, 55]. For example, Luu et al. [46] conducted a study of thousands of supercomputing applications that revealed that POSIX I/O is much more widely used than other high-level I/O libraries, and most applications only achieved a small fraction of available I/O performance.

In contrast to these studies, our work focuses on collecting detailed traces of I/O operations with the explicit purpose of analyzing their behavior to understand the file system semantics required by HPC applications. File system researchers looking to relax POSIX semantics often make assumptions that I/O operations are conflict-free: e.g., if concurrent processes write to the same file, each process will modify an independent segment of the file and there will be no write-after-write hazards that would affect file data integrity. However, these lower-level behaviors have not been studied. A primary contribution of this work is to fill in that knowledge gap in HPC I/O behavior understanding.

2.2 POSIX I/O Interface and Semantics

The POSIX I/O interface [5] and its semantics were designed decades ago for use by a single machine with a single storage device, i.e., not for the highly-concurrent operations to PFSs typical on HPC systems. POSIX I/O operations are commonplace and include the familiar `open`, `close`, `read`, and `write` operations used by applications in many domains.

The primary challenges for parallel I/O arise from the strict semantics requirements the POSIX specification imposes on `write` and `read` operations. These requirements necessitate the use of a cache coherence protocol that is often implemented using read-/write locks. The POSIX standard [5] states:

- Any successful `read` from each byte position in the file that was modified by the last `write` shall return the data specified by the `write` for that position until such byte positions are again modified.
- Any subsequent successful `write` to the same byte position in the file shall overwrite that file data.

A previous effort [9, 67] proposed a set of extensions to the POSIX I/O API for HPC. The extensions include options to introduce "laziness" into the API to improve PFS performance. For example, the effort introduced new `stat` calls where some fields are optional and the information in other fields is not required to be current to reduce query time, and API calls to flush caches and synchronize across compute nodes when operating on files where the `O_LAZY` flag was

supplied to open. Unfortunately, this proposal was not accepted into the POSIX I/O standard. However, similar functionality is now being adopted by relaxed-semantics PFSs.

2.3 PFSs and POSIX

PFSs have been designed and implemented to support parallel workloads on HPC systems. Most support POSIX semantics; this includes widely used PFSs such as Lustre [15], GPFS [58], and BeeGFS [28]. Even PFSs that support POSIX have mechanisms for relaxing the semantics for performance. For example, Lustre allows users to disable file locking that enforces POSIX consistency semantics [8] and GPFS has options for lazy metadata updates [7]. Additionally, NFS [59], which is widely used for home directories on HPC systems, relaxes POSIX semantics in favor of performance.

Recently, HPC systems have become equipped with burst buffers (BBs), and a new crop of POSIX and near-POSIX PFSs have been developed to support them. BBs are in-system solid storage devices designed to buffer the "bursty" I/O requests from HPC applications between the compute nodes and main storage. BBs are attractive because they can smooth the bursty I/O traffic and promise better scalability and performance advantages, e.g., latencies on the order of a few μs [52]. Despite their performance advantages, BBs present challenges to users. In particular, BBs that are local to individual compute nodes, e.g. as compute-node local SSDs on the supercomputer Summit [51], present challenges for applications that perform shared file I/O because these BBs do not present a shared file namespace across compute nodes.

As a result of these challenges, a set of new PFSs have been developed to facilitate the use of BBs, including BurstFS [71], GekkoFS [66], UnifyFS [37], SymphonyFS [51], Gfarm/BB [63], and echofs [49]. Each of these PFSs was designed specifically for BBs, with the common goal of being fast and easy-to-use. Some of the PFSs focus on the problem of transferring file data to and from the BBs, e.g., SymphonyFS and echofs, while others focus on supporting shared file I/O across compute-node local BBs, e.g., BurstFS and Gfarm/BB. Because of their specialized functionality and the goal of supporting the performance advantages of BBs, many of these BB PFSs relax their adherence to strict POSIX semantics; e.g., BurstFS does not guarantee that a `read` operation will always return the result of the most recent `write` in order to optimize `write` performance.

In addition to general-purpose BB PFSs, there are also file systems designed to optimize the I/O of specific workloads as described in surveys by Lüttgau and et al. [45] and Dubeyko [22]. A notable example for HPC is PLFS [12], designed specifically for large parallel shared checkpoint files over compute-node local BBs. Another promising direction for optimizing specific I/O workloads is the idea of tunable consistency semantics [33–35, 68], where users can choose different consistency semantics at run time to improve performance, e.g., with "hints" passed to the file system implementation depending on the workload requirements.

3 PFS CONSISTENCY SEMANTICS

In this section, we discuss the consistency semantics of PFSs and present the categorization of relaxed consistency models that we use in this work. In general, the difference between the models in our categorization is based upon when updates to a shared file are visible to subsequent reads.

In this work, our algorithm to determine consistency semantics needs of an application uses only data operations and leaves consideration of metadata operations to future work. Because of this, PFSs like GekkoFS [66] and BatchFS [73], that provide relaxed metadata consistency semantics but strict POSIX data consistency semantics will be categorized as having "strong consistency semantics", as described in Section 3.1. However, we do provide analysis on the metadata operations used by our set of applications in Section 6.4 and find that the applications use only a small subset of the available POSIX metadata operations.

3.1 Strong Consistency Semantics

POSIX requires sequential consistency for reads and writes: upon successful return, modifications made by a `write` call must be visible to subsequent `read` calls until those file regions are updated. Because HPC systems do not have global clocks, we employ the partial happens-before or causality order defined by the execution order within each process and the communications across processes [38]. We use $\rightarrow$ to denote this order. We define *strong consistency semantics* with the following condition: A read r from a byte returns the value written by a write w to the byte if $w \rightarrow r$, and for any other write w' to the same byte if $w' \rightarrow w$ or $r \rightarrow w'$. Otherwise, the value returned is undefined.

Most general-purpose PFSs (e.g., Lustre, GPFS, GFS, BeeGFS and PVFS2 [1] [40, 57]) support strong consistency semantics. The disadvantages of strong consistency semantics are not readily apparent in a single node/single storage device system, in which I/O operations are serialized. However, these semantics are expensive to maintain in PFSs, where there are a potentially large number of concurrent I/O requests being handled by distributed servers. Distributed locking is a common approach to guaranteeing strong consistency semantics and is used by popular PFSs like GPFS and Lustre. Locks may be applied to blocks, file segments, full files, or other granularities of file accesses. The number of locks depends on the lock granularity and the number of sharing processes. Thus, the metadata server, where the locks are normally maintained, may become a bottleneck for large-scale applications.

3.2 Commit Consistency Semantics

The fundamental problem behind the performance issues stemming from strong consistency semantics is that the PFS is ignorant of application synchronization logic and the happens-before order of concurrent I/O operations; the PFS must make worse-case assumptions and serialize all potentially conflicting I/O operations. Alternatively, an application can provide ordering information for conflicting operations so that the PFS can implement a weaker consistency semantics. We define *commit consistency semantics* as a less strict consistency model, where "commit" operations are explicitly executed by processes, and I/O updates performed by a process to

[1] PVFS and PVFS2 (now OrangeFS [10]) provide *non-conflicting write semantics* where non-overlapping writes (potentially concurrent) are immediately visible to all processes once completed. The behaviour of conflicting writes are undefined. The semantics of these file systems fit best in our strong semantics category even though they do not meet full POSIX requirements on atomicity.

a file before a commit become globally visible upon return of the commit operation.

Many user-level and BB PFSs (e.g., BSCFS [30], UnifyFS [37], SymphonyFS [51], and BurstFS [71]) provide commit consistency semantics. Note that the "commit" operation is system-specific. For example, in UnifyFS, a commit can by performed with an `fsync` operation which makes writes performed by an individual process globally visible. Alternatively, UnifyFS also provides a *lamination* operation, which renders a file permanently read-only and makes all file data globally visible. Similarly, SymphonyFS does not support read-after-write and overlapping writes between different nodes unless `fsync()` is called. The `fsync()` operation, which flushes the cache of the caller and ensures that data is persisted, acts as the commit. A `close()` call usually also has the effect of a commit.

3.3 Session Consistency Semantics

We define *session consistency semantics* as semantics that guarantee writes by a process are visible to another process when the modified file is closed by the writing process and subsequently opened by the reading process, with the `close` happening before the `open`. Commonly known as close-to-open semantics, several PFSs implement this model including NFS [59], DDN IME [20], Gfarm/BB [63] and AFS [29].

The major difference between session semantics and commit semantics is when the writes become visible to other processes. In commit semantics, updates become globally visible after a commit operation by the writer. In session semantics one needs a pair of operations, one executed by the writer and the other by the reader.

3.4 Eventual Consistency Semantics

The most relaxed semantics model we define is *eventual consistency semantics*; we are not aware of more relaxed semantics being provided by any PFS. In this model, even with no explicit commit operation, updates from a `write` are eventually visible to all readers if no subsequent write to the same location occurs. PFSs that implement this model have more freedom to perform optimizations such as write aggregation, data reorganization, and delayed propagation.

While there are several PFSs that provide eventual consistency semantics, they may impose additional constraints to provide better performance. For example, PLFS [12] implements eventual consistency semantics and is designed specifically for large parallel checkpoint files, where it converts an N-1 (N clients, one file) write access pattern into an N-N (N clients, N files) pattern. In PLFS, the outcome of two overlapping writes is not guaranteed to be correct with respect to the happens-before relationship even with explicit synchronization. Another example is echofs [49], which is designed for node-local BBs. Although echofs provides the POSIX interface, it manages data by the use of memory mapped files, and POSIX semantics is only enforced locally to each compute node. Globally, data becomes visible when it is eventually transferred out to the system-level PFS.

3.5 Discussion

A summary of the PFSs we discussed and their consistency semantics is shown in Table 1. Our categorization does not cover all the semantic differences between the file systems, but is sufficient for our purposes. Most PFSs we discussed provide strong consistency semantics for I/O operations performed by a *single process*, where a read of a file location returns the value last written to that location by the same process, if no other process modifies that location. BurstFS is an exception, where a read following two writes from the same process could return the value of either write. PLFS and PVFS2 also do not provide such a guarantee because the behaviour of overlapping writes is simply undefined.

Table 1: HPC file systems and their consistency semantics.

Consistency Semantics	File Systems
Strong Consistency	GPFS, Lustre, GekkoFS, BeeGFS, BatchFS, OrangeFS
Commit Consistency	BSCFS, UnifyFS, SymphonyFS, BurstFS
Session Consistency	NFS, AFS, DDN IME, Gfarm/BB
Eventual Consistency	PLFS, echofs, MarFS [31]

Adherence to stronger consistency semantics normally imposes higher overhead to guarantee the given consistency model, with metadata servers a likely bottleneck. PFSs that implement weaker consistency semantics can alleviate such bottlenecks. The underlying assumption behind using weaker semantics is that HPC applications do not normally access files via interleaved reads and writes to random offsets, so stronger consistency is not required.

In the following sections, we address the central questions of this paper: Do applications really need strong consistency semantics from a PFS? If not, what is the weakest model that suffices for a given application? In this work, we focus on the strongest three consistency models, excluding eventual consistency, because traditional scientific applications rely on a deterministic relationship between writes and reads. Eventual consistency may be applicable for non-traditional, emerging scientific workloads, e.g., workflows in which simulation data is pipelined to analysis modules, but we reserve analysis of these workloads for future work.

4 I/O PATTERNS

The I/O pattern of an application describes how the application accesses the PFS. A key concern for us in this work is whether multiple processes in the application concurrently access the same file, and, if so, whether the accesses are conflicting, and whether and how the accesses are synchronized. Another concern is whether accesses are "random" or "sequential", as this has a significant impact on performance. I/O patterns can be studied at different granularities. At a very high granularity, the POSIX API and most I/O libraries require users to set flags when opening a file. Common flags indicate the file will be accessed for reads only, writes only, both reads and writes, and for appending to the file. This very high granularity information for I/O patterns does not provide sufficient information for our study.

To examine the consistency semantics needs of applications, we focus on byte-level, fine-grained I/O patterns. The main focus of our study is about identifying potential conflicting I/O operations where delayed writes may cause errors. But we also harvest information on the I/O access pattern, whether random or sequential,

and about executed meta-operations. As expected, HPC applications do not access files randomly, and sequential appends are very common, e.g., for log files or snapshots of an ongoing simulation. However, when using I/O libraries like HDF5, the metadata operations of those libraries may introduce more complicated patterns. The I/O patterns can be studied at two levels: (1) the local pattern of accesses performed by one process, and (2) the global pattern of accesses generated collectively by the I/O calls across processes. Both levels of patterns affect performance, but in different ways. As we will show in Section 6.2, the global pattern is likely to appear more random than the local pattern since the I/O requests from concurrent processes are interleaved in time. However, because of the nature of scientific applications, the interleaved accesses from multiple processes are not truly random, especially when collective I/O and libraries such as MPI-IO are used that may perform data aggregation before accessing the PFS.

4.1 Overlaps and Conflicts

Conflicting accesses can occur when two I/O operations access the same location of a file. We call this situation an *overlap*. Overlaps can cause conflicts if one of the two operations is a write. If two overlapping operations by distinct processes are concurrent, then the outcome of the operations to the file is non-deterministic even under POSIX semantics: Writes are not atomic, and accesses can be interleaved in arbitrary manner. We assume now (tested later in Section 5.2) that the programs we test are "race-free": If the parallel application performs conflicting I/O operations, then these accesses are synchronized and are not concurrent. Thus, if a process writes data to a file and another process reads that data, a synchronization will ensure that the read does not start before the write completed. But, if the PFS provides weaker semantics, a conflict may still happen, as the write may not be visible to the reader when it completes. This can occur in four cases:

- RAW-[S|D]: read-after-write by the same process (S) or by different processes (D).
- WAW-[S|D]: write-after-write by the same process (S) or by different processes (D).

We define these four cases as *potential conflicts*. Whether they are actual conflicts depends on the PFS semantics. In the majority of PFSs, conflicting accesses by the same process will take effect in the right order so that only RAW-D and WAW-D are potentially problematic. Note that a write-after-read pair cannot cause a conflict, as we assume conflicting operations are properly synchronized and the read will complete before the write starts.

The information about potential conflicts is important at different levels: A programmer running the application on a PFS with weak consistency can prevent the conflicts by inserting `commit` operations at suitable points, or the designer of a parallel I/O library can insert `commit` operations automatically. On the other hand, if the application can tolerate relaxed consistency, then the PFS or I/O libraries can leverage the tolerance for improved performance.

5 DETECTING OVERLAPS AND CONFLICTS

To analyze the I/O behaviors of an application, we need to extract its dynamic I/O operations. The operations depend on the application logic, but also on parameters such as the PFS and I/O library settings, and on the underlying hardware configuration such as the number of data servers.

We utilize the multi-level I/O tracing tool Recorder [69] to generate traces from applications. Recorder captures I/O operations at multiple layers of the I/O stack, currently supporting HDF5, MPI-I/O, and POSIX, which gives us the ability to identify the I/O layer responsible for introducing conflicts. Additionally, Recorder generates detailed trace records for I/O operations in each I/O layer used either explicitly or implicitly by the application. The trace records include entry/exit time stamps, function name, and all function parameters, except the data buffer content. This level of detail allows us to identify operations that introduce overlaps and conflicts. While our focus is on identifying potential conflicts in file accesses, the detailed traces obtained also enable us to identify to what extent file accesses are sequential or random, which is important for performance optimizations.

We analyze the traces to detect overlaps and conflicts by building on the algorithm for detecting I/O operation overlaps developed in our prior work [69] and modify it to additionally detect conflicts.

5.1 Detecting Overlaps

To detect overlaps, we employ the algorithm from our prior work [69], where each record is a tuple $(t, r, os, oe, type)$, where t is the entry timestamp, r is the rank of the process who made the call, os and oe are the starting and ending offsets of this I/O operation, and $type$ indicates a read or write operation.

Calculating the offset of an I/O operation is not always straightforward. For functions like `pwrite`, the offset and length are included in the arguments of the call, but for functions like `write`, the offset is not specified, but depends on previous accesses to the file. Therefore, the algorithm tracks the most up-to-date offset for each file. For metadata operations like `open` and `seek`, we update the offset according to the open flag (e.g., O_CREAT, O_TRUNC, or O_APPEND) and the seek flag (e.g., SEEK_CUR, SEEK_END, or SEEK_SET) respectively. For operations such as `write` and `fwrite`, we increment the current offset by the number of bytes accessed by that function.

Once we have the correct offset for each function, we use Algorithm 1 to construct an overlapping pair table P. This algorithm is quadratic in the worse case, since each I/O operation could overlap with all others. In practice, the running time (sorting excepted) is linear in the number of records. Although we have not done so, sorting can be replaced by merging as records for each rank are already sorted.

Algorithm 1 Detecting overlaps

```
1: Sort tuples by os
2: for each tuple T_i do
3:     for each tuple T_j, j > i do
4:         if os_j > oe_i then
5:             break ▷ subsequent tuples will not overlap with T_i
6:         else
7:             P[r_i, r_j] ← 1                    ▷ T_i and T_j overlap
```

5.2 Detecting Conflicts

We use timestamps in the traces to determine the order of I/O operations from different nodes. Since the timestamps come from the local system clocks, large clock skews could result in incorrect ordering. To reduce skew, we perform a barrier operation when starting the run and adjust timestamps in the trace records using the exit time from the barrier as $time = 0$. We found that clock drift on the system we used can be ignored, because clock skews in the traces we collected are less than 20 microseconds, while potentially conflicting I/O operations are 10's of milliseconds apart.

In order to further validate our methodology, we analyzed traces of the FLASH application (Section 6.3), which was the one application that exhibited conflicting I/O accesses. We matched sends to receives and collective functions invocations, so as to determine the execution order imposed by the communications between processes: e.g., a send starts before the receive completes, and a barrier starts at all nodes before it completes at any node. We found that conflicting I/O operations were properly synchronized by the MPI calls: If call A and B performed conflicting I/O accesses, and call A had a lower timestamp than call B, then A necessarily executed before B, due to the program synchronization logic. Thus, we can assume that timestamp order of conflicting I/O operations matches their execution order, and that this execution order is enforced by the program logic. (The order of non-conflicting I/O operations does not affect the computation.)

Now we can describe the algorithm for detecting conflicts. Two tuples $(t_1, r_1, os_1, oe_1, type_1)$ and $(t_2, r_2, os_2, oe_2, type_2)$, where $t_1 < t_2$, are a conflict pair if the following conditions are satisfied:

(1) The pair overlaps: either $os_1 \leq os_2 \leq oe_1$ or $os_2 \leq os_1 \leq oe_2$.
(2) The first operation is a write: $type_1 = write$.
(3) For commit semantics: process r_1 does not execute any commit operation after t_1 and before t_2.
(4) For session semantics: there is no close operation on process r_1 with t_c and open operation on process r_2 with t_o so that $t_1 < t_c < t_o < t_2$

We expand the overlap detection algorithm presented above (Section 5.1) to identify those overlaps that correspond to a read-after-write conflict or a write-after-write conflict, and whether the two conflicting accesses are on the same process or on distinct processes. In order to test the third condition, we need to find, for each write, what is the earliest succeeding commit executed by the same process. In order to test the fourth condition, for each I/O operation we need to find the earliest time an ensuing close is executed and the latest time a preceding open is executed by the same process. We expand each record $(t, r, os, oe, type)$ with two additional fields: to, the time of the last preceding open and tc, the time of the first succeeding close or commit by process r. Then $(t_1, r_1, os_1, oe_1, type_1, to_1, tc_1)$ and $(t_2, r_2, os_2, oe_2, type_2, to_2, tc_2)$, with $t_1 < t_2$, conflict in commit semantics if they overlap, $type_1 = write$, and $tc_1 > t_2$; they conflict in session semantics if they overlap, $type_1 = write$, and it is not the case that $t_1 < tc_1 < to_2 < t_2$.

We can mark records with the time of the last preceding open and next following commit or close by traversing the records of each process in timestamp order. Alternatively, we can create a table of successive commit and close operations and a table of successive open operations for each process. Conditions three and four can be checked by performing one or two binary searches in the table. Since the number of open, close, and commit operations usually is very small the overhead for the binary searches will be negligible.

6 RESULTS

Here, we present the results of our investigation of the I/O patterns of HPC applications and of our algorithm for detecting I/O access conflicts. First, we explore our findings of the access patterns from both the application's level and a PFS's perspective. Next, we present the conflicts detected under different consistency semantics. Finally, we show the metadata operations observed from each application and I/O layer.

6.1 System and Application Configurations

We performed our experiments on the Quartz system at Lawrence Livermore National Laboratory (LLNL). Each Quartz node consists of an Intel Xeon E5-2695 with two sockets and 36 cores in total, with 128GB memory; the nodes are connected via Omni-Path. The operating system is TOSS 3. Slurm is used to manage user jobs. The PFS is an LLNL customized version of Lustre, 2.10.6_2.chaos.

We selected 17 HPC applications: 11 real-world scientific applications, 4 I/O benchmarks (MACSio, pF3D-IO, VPIC-IO and HACC-IO), and one machine learning application (LBANN). The full list of these applications and their configurations is given in Table 5. The applications are representative of the typical workloads at a supercomputing center and span a variety of domains. They perform I/O using the POSIX API and a variety of I/O libraries: MPI-IO [19], HDF5 [23], Silo [48], NetCDF [41] or ADIOS2 [24].

An application's I/O patterns depend on its intrinsic I/O operations but also on configuration parameters of the I/O libraries and the underlying file systems. Because of this, for applications that can employ multiple I/O libraries, we run the application using each of the I/O libraries supported by the application. We expect the I/O patterns, especially with respect to I/O conflicts, should not depend on the scale of runs. To confirm this, we ran all applications at two different scales: (1) 8 nodes with 8 processes per node, for 64 MPI ranks in total; and (2) 32 nodes with 32 processes per node, for 1024 MPI ranks in total. Our results confirmed our expectation, as we found no differences due to scale in the I/O patterns for any application we studied. Thus, for ease of presentation, we focus on the results collected from 64-process runs in the rest of this paper.

As much as possible, we used the same compiler and library versions for our runs, but needed to make exceptions in some cases for dependency and compatibility issues. Overall, we used three different compiler and I/O library combinations to build 15 applications from source. We only had access to the binaries of the remaining two (pF3D-IO and VASP). We summarize the build and link information in Table 2.

6.2 Access Patterns Overview

We first categorize the applications we study according to the high-level I/O access patterns they exhibit in Table 3 to show our coverage of the possible behaviors of applications. In our categorization, we use an $X - Y$ notation where X represents the number of processes performing I/O, and Y represents the number of files accessed. $X = N$ indicates that all processes perform I/O operations, while

Table 2: Build and link configurations for the applications in our experiments. We do not have access to the source code of pF3D-IO and VASP, the information reported here is retrieved from the `ldd` command. The versions for other I/O libraries whenever used are: ADIOS 2.5.0, NetCDF 4.3.3.1 and Silo 4.10.2.

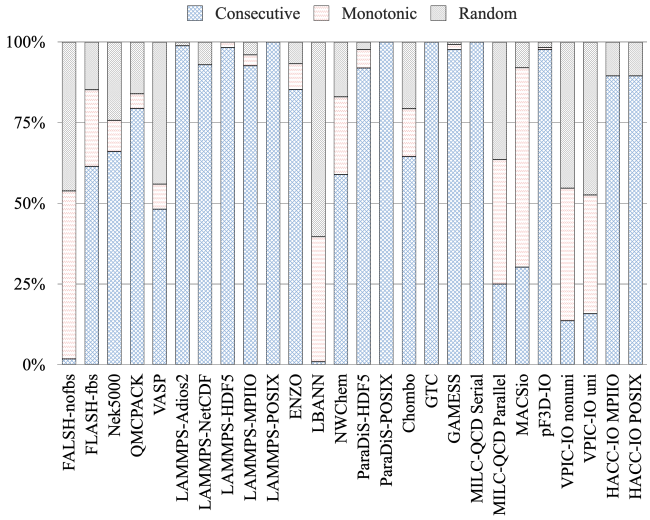
Applications	Compiler	MPI	HDF5
ENZO, NWChem, GAMESS, LAMMPS, QMCPACK, Nek5000, GTC, MILC-QCD, HACC-IO, VPIC-IO	Intel 19.1.0	Intel MPI 2018	HDF5 1.12.0
pF3D-IO, VASP	Intel 18.0.1	MVAPICH 2.2	
LBANN	GCC 7.3.0	MVAPICH 2.3	HDF5 1.10.5
ParaDiS, Chombo, FLASH, MACSio	Intel 19.1.0	Intel MPI 2018	HDF5 1.8.20

$X = M$ indicates that I/O is executed by a subset of processes. An $N - N$ pattern typically indicates that each process accesses a distinct file; an $M - M$ pattern typically indicates that each of the M processes aggregates the I/O requests of a subset of N/M processes, and each aggregator accesses a distinct file.

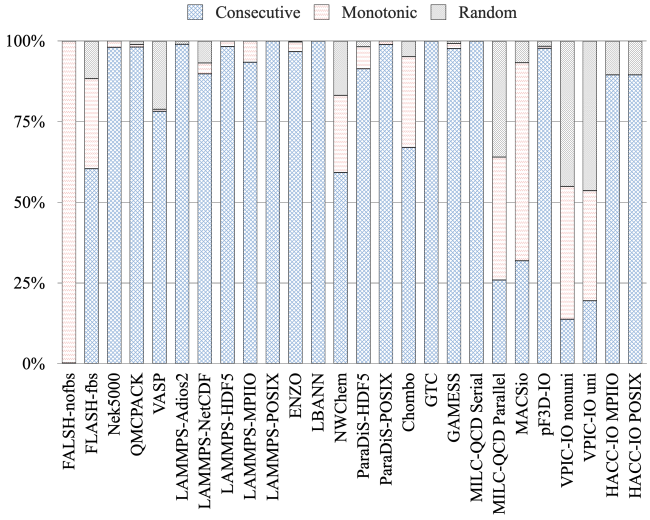
We categorize the access patterns in the files as: *consecutive*, *monotonic* or *random*: Let o_i and n_i be the offset and the number of consecutive bytes accessed by the i-th I/O operation. The *consecutive* pattern requires $o_{i+1} = o_i + n_i$. The *monotonic* pattern only requires that $o_{i+1} > o_i + n_i$. All other accesses are considered *random*. Consecutive and monotonic accesses are often *strided* or *strided cyclic*: At each I/O phase, process i accesses the file at offset $ai + b$ and all processes access the same number of bytes (except for a small amount of extra metadata that could be introduced by the I/O library). We see that the applications we have chosen for our study provide good coverage of the possible space of I/O patterns exhibited by HPC applications. Surprisingly, many of the applications exhibit a 1-1 pattern for accessing files. We anticipated that nearly all applications would perform parallel I/O of some sort, but we see that is not the case. We note that most applications show a 1-1 pattern when reading input files, but for space reasons we do not include that aspect in our table. Also note that Table 3 only shows the patterns we observed. Our runs are not exhaustive across all possible configurations for these applications, which may show different patterns. For example MILC-QCD, with the *save_parallel* parameter (MILC-QCD Parallel), uses an N-1 pattern for checkpointing, whereas with the *save_serial* parameter (MILC-QCD Serial), it uses only one rank for I/O. FLASH is another example, which will be discussed in more details in Section 6.2.2.

Now we discuss the low-level access patterns of our applications. Figure 1(a) shows the global access patterns from the perspective of the PFS, and Figure 1(b) shows the aggregated local access patterns from the perspective of individual processes. We compute the percentage of each access type by dividing the number of accesses for that type by the number of total accesses, across all files accessed by the application. Each bar in the charts represents a single execution of an application configuration.

From the perspective of a single process, random accesses to a file are rare. From the global perspective of the PFS, accesses are sometimes much more random (e.g., FLASH-nofbs and LBANN), but global random accesses are still rare. The global access pattern is same as the local pattern when each process accesses a distinct file. It is also regular when all processes access the same file and the accesses are closely coordinated, as is the case for collective I/O. These results clearly indicate that PFS performance can be improved by read-ahead or by aggregating delayed writes, both at the client and at the server side.



(a) Global pattern from the perspective of the PFS.



(b) Local pattern from the perspective of individual processes.

Figure 1: Overview of low-level access patterns

In the interest of space, we are not able to inspect the detailed I/O patterns of each application we studied. Instead, in the remainder of

Table 3: High-level access patterns of applications studied.

	Consecutive	Strided	Strided Cyclic
N-N	ENZO, pF3D-IO, HACC-IO, NWChem		
N-M		MACSio	
N-1	LBANN, VASP	Chombo, FLASH-nofbs, ParaDiS-HDF5, ParaDiS-POSIX, MILC-QCD Parallel	
M-M	GAMESS, LAMMPS-Adios2		
M-1		LAMMPS-MPIIO	FLASH-fbs, VPIC-IO
1-1	GTC, Nek5000, NWChem, QMCPACK, VASP, MILC-QCD Serial, LAMMPS-HDF5, LAMMPS-NetCDF, LAMMPS-POSIX		

this section we focus on four representative applications: LAMMPS, ParaDiS, FLASH, and LBANN.

6.2.1 LAMMPS and ParaDiS. We highlight LAMMPS and ParaDiS because both of these applications can employ multiple I/O libraries and show different I/O patterns for each library. For both LAMMPS and ParaDiS, we note that when using the POSIX API, all I/O accesses are consecutive from both the local and global perspectives. However, when the applications use higher-level I/O libraries, random accesses are introduced. This is primarily due to bookkeeping and optimizations performed by the I/O libraries, e.g., HDF5 stores and accesses metadata that is interspersed within the user file, leading to random accesses.

6.2.2 FLASH. We selected the FLASH application because it can be configured to employ independent or collective I/O. If the parameter "block size" is set to be dynamic, then independent I/O is used (FLASH-nofbs), and if the parameter is fixed, collective I/O is used (FLASH-fbs). As expected, with collective I/O, the global access pattern is much less random.

Figure 2 shows detailed file access patterns (write-only) for the two configurations of FLASH (64 ranks run) for accessing checkpoint and plot files. The charts in (a) and (d) show the access patterns generated by writing a checkpoint file in the two I/O modes (collective vs. independent). When independent I/O is used, every process participates in the I/O activity, whereas when collective I/O is enabled, the MPI-IO library (via calls from HDF5) aggregates I/O accesses and only six aggregator processes access the PFS. The small I/O accesses at the beginning of the file are HDF5 metadata operations. Because ~ 30 processes are involved in metadata writes, it suggests that the MPI-IO aggregators are not employed for metadata. Figure 2(c) shows the access patterns of a plot file with collective I/O, where only rank 0 writes data to the plot file, but around 30 ranks participate in HDF5 metadata operations.

The independent I/O behavior of FLASH-nofbs shown in Figure 1(a) exhibits ~50% random accesses. We plot the accesses to a checkpoint file over time in Figure 2(b) and (e). As Figure 2(e) shows, there is a large amount of parallelism in those accesses, which is expected. However, we see a different pattern when we focus on a single rank as show in Figure 2(f), where for rank 0, the accesses are mostly monotonic.

6.2.3 LBANN. We chose to highlight LBANN because it is an example of a read-intensive application, which differs from the majority of scientific simulations that are write-intensive. All processes in LBANN concurrently execute the POSIX API `read()` call to load the entire dataset into memory. Similar to FLASH-nofbs, from the global view of the PFS, there are a large portion of random accesses because all reads are issued in parallel. However, from the local view of a single process, all reads are consecutive because every rank reads all bytes of the file from the beginning to the end.

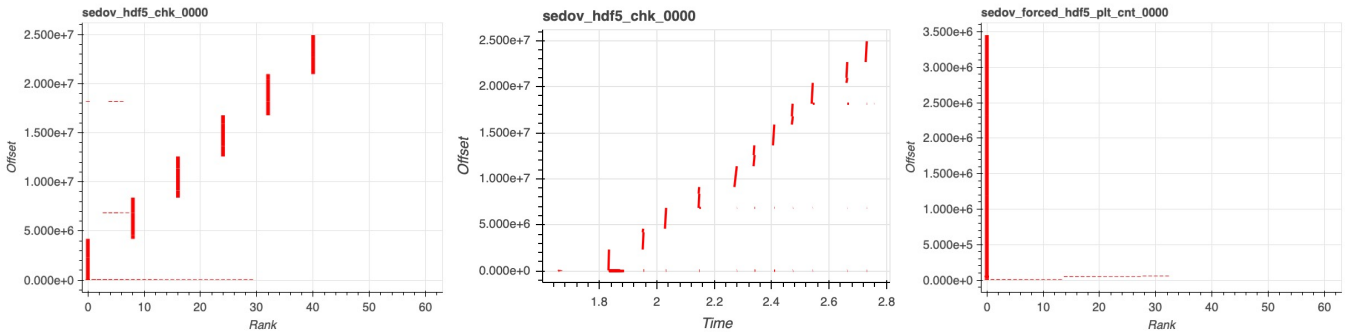
6.3 Access Conflicts with Different Semantics

Here, we report our findings on the semantics needs of scientific applications and show support for the assumption that strong consistency semantics are rarely required. We use the algorithm from Section 5.2 to detect conflicts for the 17 applications under session semantics and commit semantics [2], and show the results for session semantics in Table 4. Seven of our applications exhibit conflicting I/O accesses under session semantics, but in only one application (FLASH) the conflict involves two distinct processes. Since all but one of the PFSs we studied can correctly handle RAW and WAW conflicts on the same process (BurstFS being the exception), all the applications but FLASH will run correctly with session semantics.

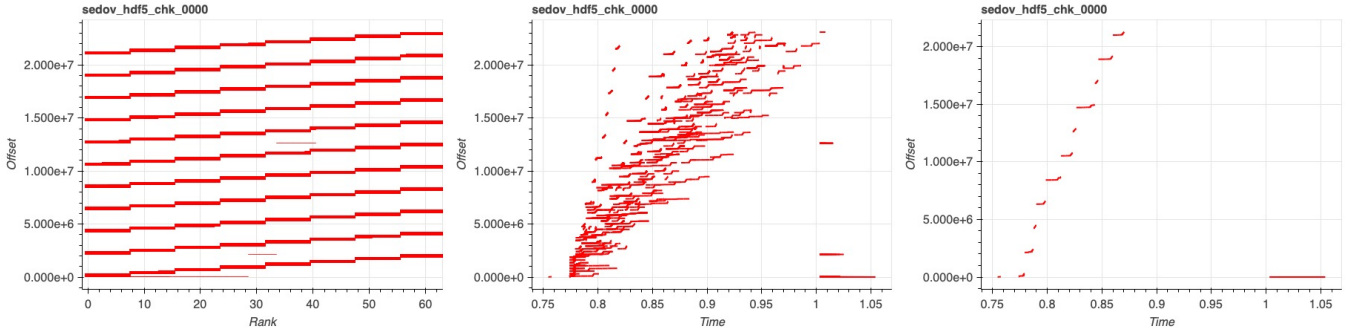
We employed our conflict detection algorithm for commit semantics, and the conflicts in FLASH disappeared, but the conflict pattern of the other applications was unchanged. The conflicts in FLASH are caused by the flushes of HDF5 metadata. During the checkpoint step, FLASH calls `H5Fflush()` (which flushes both data and metadata) after having written one dataset. The file is closed once all datasets have been written. Before the file close, session semantics do not guarantee the latest updates are seen by other processes so the conflict is inevitable. In comparison, the commit operation (`fsync()` called by `H5Fflush()`) in commit semantics makes the updates visible to all processes and avoids the conflict. No conflicting accesses are generated by LAMMPS when using POSIX, MPI-IO, and HDF5 for I/O. The conflicts appeared only when NetCDF or ADIOS are used, where the conflicts are caused by library metadata operations. For example, in LAMMPS-ADIOS the conflict is due to the overwriting of a single byte of the ADIOS metadata file (*/md.idx).

Some conflicts can be avoided with little effort, especially when they are introduced by I/O libraries. For example, in FLASH the conflicts are caused by flushes of metadata, and to avoid the conflicts we can either enable the HDF5 collective metadata mode (which would have only rank 0 perform all metadata I/O) or simply remove

[2] Our test for a commit operation is positive if `fsync, fdatasync, fflush, fclose` or `close` are called by the application or I/O library.



(a) Collective I/O per rank, checkpoint. (b) Collective I/O over time, checkpoint. (c) Collective I/O per rank, plot file.



(d) Independent I/O per rank, checkpoint. (e) Independent I/O over time, checkpoint. (f) Rank 0 independent I/O over time, checkpoint.

Figure 2: Collective I/O write patterns of FLASH-fbs (a, b, c) and independent I/O write patterns of FLASH-nobfs (d, e, f).

the call to `H5Fflush()`. In the latter case, correctness is still guaranteed in the absence of failures since the `H5Fclose()` in the end implies an `H5Fflush()`. With a single line code change, FLASH can run correctly on all file systems that support session semantics or commit semantics.

In summary, all but one of the applications we studied can execute correctly with session semantics, provided that conflicts on the same process are properly handled. The one exception can be handled with a single line change to an I/O library. Under commit semantics, the results are similar since applications do not make much use of `fsync` or other commit operations.

6.4 Metadata Operations

Because metadata operations can introduce performance bottlenecks, PFS developers may choose to relax POSIX metadata requirements. For example, it is rare for a scientific application to access the `atime` attribute of its data files. A PFS developer may choose to update `atime` only once at the end of the execution in order to reduce the number of update messages sent to the metadata server (or to avoid invalidation messages if client-side caches are used). Figure 3 shows POSIX I/O metadata and utility I/O operations [3] used in the applications we studied. We indicate where the invocations occur, in the MPI library, in HDF5, or in the application or another library. (We cannot further refine the last category Since Recorder does not trace other libraries.)

Figure 3: Metadata operations used by applications.

[3] The operations we monitored were: mmap, mmap64, msync, stat, stat64, lstat, lstat64, fstat, fstat64, getcwd, mkdir, rmdir, chdir, link, linkat, unlink, symlink, symlinkat, readlink, readlinkat, rename, chmod, chown, lchown, utime, opendir, readdir, closedir, rewinddir, mknod, mknodat, fcntl, dup, dup2, pipe, mkfifo, umask, fileno, access, faccessat,tmpfile, remove, truncate, ftruncate

Table 4: Conflicts with session semantics. 'S' indicates the conflicting operations are called by the same process; 'D' indicates that the conflict involves multiple processes. Under commit semantics, the conflicts from FLASH disappeared.

Application	I/O Library	WAW		RAW	
		S	D	S	D
FLASH	HDF5	✓	✓		
ENZO	HDF5			✓	
NWChem	POSIX	✓		✓	
pF3D-IO	POSIX			✓	
MACSio	Silo	✓			
GAMESS	POSIX	✓			
LAMMPS	ADIOS	✓			
LAMMPS	NetCDF	✓			
LAMMPS	HDF5				
LAMMPS	MPI-IO				
LAMMPS	POSIX				
MILC-QCD	POSIX				
ParaDiS	HDF5				
ParaDiS	POSIX				
VASP	POSIX				
LBANN	POSIX				
QMCPACK	HDF5				
Nek5000	POSIX				
GTC	POSIX				
Chombo	HDF5				
HACC-IO	MPI-IO				
HACC-IO	POSIX				
VPIC-IO	HDF5				

We see that each application configuration uses only a small set of metadata operations, and many operations like `rename()`, `chown()` and `utime()` are not used by any application. I/O libraries introduce more metadata operations than direct use of the POSIX API, and each library introduces a different set of operations. For example, compared to ParaDiS-POSIX, ParaDiS-HDF5 uses three more metadata operations, `lstat()`, `fstat()`, and `ftruncate()`. Similarly in LAMMPS, only two operations are observed for LAMMPS-POSIX, but LAMMPS using I/O libraries introduces additional operations such as `getcwd()` and `unlink()`. In some cases, we observed a POSIX call in the source code but did not found it in our traces. This is the case for `unlink()` in ENZO. This could be due to the chosen run configurations, or to dead code in the application.

7 DISCUSSION

The results of our work provide HPC users a methodology for examining the I/O patterns of their applications to determine whether using a relaxed-consistency PFS is appropriate. We have made all the data and code used in this paper public (https://github.com/uiuc-hpc/Recorder) so that the community can use and build upon it. The data includes traces files, input/output files, and a detailed report for each application run, including information such as I/O sizes, function counters, conflicts detected for each file, etc. The code implements the algorithms we used for analyzing the I/O traces.

Moving forward, our hope is that our approach can impact the community by providing a basis for determining the semantics and operations needed by applications and provided by PFSs. Unfortunately today, it is difficult for HPC users to know what operations are supported by non-POSIX PFSs as the support is often poorly documented. Better documentation of the supported operations, deviations from POSIX semantics, and more uniformity in terminology across PFSs will greatly impact the HPC community.

For future work, we plan to expand our conflicts detection algorithm to support metadata operations and complex HPC workflows consisting of multiple applications. In addition, we plan to investigate other semantics properties, such as safety and order semantics, in order to define a more precise semantics model for PFSs.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-814852. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the DOE Early Career Research Program and by NSF CCF grant 17-63540.

The views and opinions of the authors do not necessarily reflect those of the U.S. government or Lawrence Livermore National Security, LLC neither of whom nor any of their employees make any endorsements, express or implied warranties or representations or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of the information contained herein.

REFERENCES

[1] 2010. The Gyrokinetic Toroidal Code. http://phoenix.ps.uci.edu/GTC
[2] 2016. MILC Code Version 7. http://www.physics.utah.edu/~detar/milc/milc_qcd.html
[3] 2016. PIOK: Parallel I/O Kernels. https://code.lbl.gov/projects/piok
[4] 2018. HACC IO Kernel from the CORAL Benchmark Codes. https://asc.llnl.gov/coral-benchmarks#hacc
[5] 2018. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (2018), 1–3951.
[6] 2019. Flash Center for Computational Science. http://flash.uchicago.edu
[7] 2020. IBM Spectrum Scale Version 5.0.0 Administration Guide. https://www.ibm.com/support/knowledgecenter/STXKQY_5.0.0/com.ibm.spectrum.scale.v5r00.doc/pdf/scale_adm.pdf
[8] 2020. Lustre Software Release 2.x Operations Manual. https://lustre.org/documentation
[9] 2020. POSIX EXTENSIONS. https://www.pdl.cmu.edu/posix
[10] 2020. The OrangeFS Project. http://www.orangefs.org
[11] Mark Adams, Peter O Schwartz, Hans Johansen, Phillip Colella, Terry J Ligocki, Dan Martin, ND Keen, Dan Graves, D Modiano, Brian Van Straalen, et al. 2015. *Chombo Software Package for AMR Applications-Design Document.* Technical Report.
[12] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. 2009. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 1–12.
[13] Julian Borrill, Leonid Oliker, John Shalf, and Hongzhang Shan. 2007. Investigation of Leading HPC I/O Performance Using a Scientific-Application Derived Benchmark. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. 1–12.
[14] Kevin J Bowers, BJ Albright, L Yin, B Bergen, and TJT Kwan. 2008. Ultrahigh Performance Three-Dimensional Electromagnetic Relativistic Kinetic Plasma Simulation. *Physics of Plasmas* 15, 5 (2008), 055703.
[15] Peter Braam. 2019. The Lustre Storage Architecture. *arXiv preprint arXiv:1903.01955* (2019).
[16] Greg L Bryan, Michael L Norman, Brian W O'Shea, Tom Abel, John H Wise, Matthew J Turk, Daniel R Reynolds, David C Collins, Peng Wang, Samuel W

Table 5: Application Input and Run Configuration Information

Application	Version	I/O Library	Configuration Description
FLASH [6]	4.4	HDF5	2D 512x512 Sedov explosion problem. 100 time steps; Checkpointing at every 20 steps.
Nek5000 [36]	v19.0rc1	POSIX	Eddy solutions in doubly-periodic domain with an additional translational velocity. This case monitors the error for an exact 2D solution to the Navier-Stokes equations. 1000 timesteps; Checkpointing at ever 100 steps.
QMCPACK [32]	3.9.2	HDF5	A short diffusion Monte Carlo calculation of a water molecule. 100 warmup steps; 40 computation steps; Checkpointing at every 20 steps.
VASP [61]	5.4.4	POSIX	Simulate elastic properties and energies for zinc-blended GaAs at a given volume and pressure.
LBANN [65]	0.1000	POSIX	Train and test Autoencoder with CIFAR-10 dataset. The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes.
LAMMPS [56]	20Mar 3	ADIOS NetCDF HDF5 MPI-IO POSIX	2D LJ flow simulation. 100 steps in total and checkpointing at every 20 steps. Dump only atoms unscaled coordinates. Different I/O libraries are usd for writing the dump file.
ENZO [16]	enzo-dev 20200623	HDF5	Non-cosmological Collapse test: a sphere collapses until becoming pressure supported.
NWChem [64]	6.8.1	POSIX	3-Carboxybenzisoxazole Gas-phase Dynamics at 500K. 5 equilibration steps, 30 data gathering steps and print output every 5 steps. Write out solute coordinates to the trajectory file every step.
ParaDiS [17]	2.5.1.1	HDF5 POSIX	Use fast multipole method for far-field forces to simulate dislocations in a sample copper.
Chombo [11]	3.2.7	HDF5	A 3D variable-coefficient AMR Poisson solve in which the RHS and the coefficients are sinusoidals.
GTC [1]	0.92	POSIX	Built-in example run (gtc.64p.input) of the Gyrokinetic Toroidal code.
GAMESS [25]	June 30, 2019 R1	POSIX	Closed shell functional test on a C1 conformer of ethyl alcohol.
MILC-QCD [2]	7.8.1	POSIX	MILC collaboration code for lattice QCD calculations.
MACSio [21]	1.1	Silo	Simulate the I/O behaviours of ALE3D [50]. Silo is used for I/O.
pF3D-IO	-	POSIX	Simulates one pF3D [39] checkpoint step. The total output of one process is about 2GB.
HACC-IO [4]	1.0	MPI-IO POSIX	The HACC I/O benchmark captures the I/O patterns of the HACC [26] simulation code. This includes the checkpoint and restarts as well as the analysis outputs produced by the simulation. It also captures the various I/O interfaces used in HACC, namely, POSIX I/O, MPI collective I/O and MPI independent I/O.
VPIC-IO [3]	0.1	HDF5	VPIC [14] is a scalable particle physics simulation. The I/O pattern of VPIC-IO is a 1D particle array of a given number of particles where each particle has eight variables.

Skillman, et al. 2014. Enzo: An Adaptive Mesh Refinement Code for Astrophysics. *The Astrophysical Journal Supplement Series* 211, 2 (2014), 19.

[17] Wei Cai and Vasily V Bulatov. 2004. Mobility Laws in Dislocation Dynamics Simulations. *Materials Science and Engineering: A* 387 (2004), 277–281.

[18] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 2009. 24/7 Characterization of Petascale I/O Workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 1–10.

[19] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. 1995. Overview of the MPI-IO Parallel I/O Interface. In *IPPS'95 Workshop on Input/Output in Parallel and Distributed Systems*. 1–15.

[20] DDN. 2020. DDN INFINITE MEMORY ENGINE. https://www.ddn.com/products/ime-flash-native-data-cache

[21] James Dickson, Steven Wright, Satheesh Maheswaran, Andy Herdman, Mark C Miller, and Stephen Jarvis. 2016. Replicating HPC I/O Workloads with Proxy Applications. In *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*. IEEE, 13–18.

[22] Viacheslav Dubeyko. 2019. Comparative Analysis of Distributed and Parallel File Systems' Internal Techniques. *arXiv preprint arXiv:1904.03997* (2019).

[23] Mike Folk, Albert Cheng, and Kim Yates. 1999. HDF5: A File Format and I/O Library for High Performance Computing Applications. In *Proceedings of supercomputing*, Vol. 99. 5–33.

[24] William F Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, et al. 2020. ADIOS 2: The Adaptable Input Output System. A Framework for High-Performance Data Management. *SoftwareX* 12 (2020), 100561.

[25] Mark S Gordon and Michael W Schmidt. 2005. Advances in Electronic Structure Theory: GAMESS a Decade Later. In *Theory and applications of computational chemistry*. Elsevier, 1167–1189.

[26] Salman Habib, Adrian Pope, Hal Finkel, Nicholas Frontiere, Katrin Heitmann, David Daniel, Patricia Fasel, Vitali Morozov, George Zagaris, Tom Peterka, et al. 2016. HACC: Simulating Sky Surveys on State-of-the-Art Supercomputing Architectures. *New Astronomy* 42 (2016), 49–65.

[27] Jaehyun Han, Deoksang Kim, and Hyeonsang Eom. 2016. Improving the Performance of Lustre File System in HPC Environments. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 84–89.

[28] Frank Herold, Sven Breuner, and Jan Heichler. 2014. An Introduction to BeeGFS. (2014). https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf

[29] John H Howard, Michael L Kazar, Sherri G Menees, David A Nichols, Mahadev Satyanarayanan, Robert N Sidebotham, and Michael J West. 1988. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems (TOCS)* 6, 1 (1988), 51–81.

[30] IBM. 2020. Burst Buffer Shared Checkpoint File System. https://github.com/IBM/CAST/tree/master/bscfs

[31] Jeffrey Thornton Inman, William Flynn Vining, Garrett Wilson Ransom, and Gary Alan Grider. 2017. MarFS, a Near-POSIX Interface to Cloud Objects. *; Login* 42, LA-UR-16-28720; LA-UR-16-28952 (2017).

[32] Jeongnim Kim, Andrew D Baczewski, Todd D Beaudet, Anouar Benali, M Chandler Bennett, Mark A Berrill, Nick S Blunt, Edgar Josué Landinez Borda, Michele Casula, David M Ceperley, et al. 2018. QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter* 30, 19 (2018), 195901.

[33] Michael Kuhn. 2013. A Semantics-Aware I/O Interface for High Performance Computing. In *International Supercomputing Conference*. Springer, 408–421.

[34] Michael Kuhn. 2015. Dynamically Adaptable I/O Semantics for High Performance Computing. In *International Conference on High Performance Computing*. Springer, 240–256.

[35] Michael Kuhn, Julian Martin Kunkel, and Thomas Ludwig. 2009. Dynamic File System Semantics to Enable Metadata Optimizations in PVFS. *Concurrency and Computation: Practice and Experience* 21, 14 (2009), 1775–1788.

[36] Argonne National Laboratory. 2020. NEK5000 v19.0. https://nek5000.mcs.anl.gov

[37] Lawrence Livermore National Laboratory. 2020. UnifyFS: A File System for Burst Buffers . https://github.com/LLNL/UnifyFS.

[38] Leslie Lamport. 1978. Time, Clocks and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558.

[39] Steven H Langer, Abhinav Bhatele, and Charles H Still. 2014. pF3D Simulations of Laser-Plasma Interactions in National Ignition Facility experiments. *Computing in Science & Engineering* 16, 6 (2014), 42–50.

[40] Rob Latham, Neil Miller, Robert Ross, Phil Carns, et al. 2004. A next-generation parallel file system for Linux cluster. *LinuxWorld Mag.* 2, ANL/MCS/JA-48544 (2004).

[41] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. 2003. Parallel netCDF: A High-Performance Scientific I/O Interface. In *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE, 39–39.

[42] Glenn Lockwood. 2017. What's so bad about POSIX I/O? https://www.nextplatform.com/2017/09/11/whats-bad-posix-io/. The Next Platform.

[43] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. 2010. Managing Variability in the IO Performance of Petascale Storage Systems. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[44] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. 2008. Flexible IO and Integration for Scientific Codes Through the Adaptable IO System (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. 15–24.

[45] Jakob Lüttgau, Michael Kuhn, Kira Duwe, Yevhen Alforov, Eugen Betke, Julian Kunkel, and Thomas Ludwig. 2018. Survey of Storage Systems for High-Performance Computing. *Supercomputing Frontiers and Innovations* 5, 1 (2018), 31–58.

[46] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. 2015. A Multiplatform Study of I/O Behavior on Petascale Supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. 33–44.

[47] Ethan L Miller and Randy H Katz. 1991. Input/Output Behavior of Supercomputing Applications. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. 567–576.

[48] Mark Miller. 2009. Silo–A Mesh and Field I/O Library and Scientific Database. *Lawrence Livermore National Laboratory. https:// wci.llnl.gov/ simulation/ computer-codes/silo* (2009).

[49] Alberto Miranda, Ramon Nou, and Toni Cortes. 2018. echofs: A Scheduler-Guided Temporary Filesystem to Leverage Node-local NVMs. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 225–228.

[50] Charles R Noble, Andrew T Anderson, Nathan R Barton, Jamie A Bramwell, Arlie Capps, Michael H Chang, Jin J Chou, David M Dawson, Emily R Diana, Timothy A Dunn, et al. 2017. *ALE3D: An Arbitrary Lagrangian-Eulerian Multi-Physics Code*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[51] Sarp Oral, Sudharshan S Vazhkudai, Feiyi Wang, Christopher Zimmer, Christopher Brumgard, Jesse Hanley, George Markomanolis, Ross Miller, Dustin Leverman, Scott Atchley, et al. 2019. End-to-end I/O Portfolio for the Summit Supercomputing Ecosystem. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[52] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing Memory-mapped I/O for Fast Storage Devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 813–827.

[53] Barbara K Pasquale and George C Polyzos. 1993. A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. 388–397.

[54] Barbara K Pasquale and George C Polyzos. 1994. Dynamic I/O Characterization of I/O Intensive Scientific Applications. In *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. IEEE, 660–669.

[55] Tirthak Patel, Suren Byna, Glenn K Lockwood, and Devesh Tiwari. 2019. Revisiting I/O Behavior in Large-Scale Storage Systems: the Expected and the Unexpected. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[56] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of computational physics* 117, 1 (1995), 1–19.

[57] Robert B Ross, Rajeev Thakur, et al. 2000. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th annual Linux showcase and conference*. 391–430.

[58] Frank B Schmuck and Roger L Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters.. In *FAST*, Vol. 2.

[59] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck. 2003. Rfc3530: Network File System (NFS) Version 4 Protocol.

[60] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K Lockwood, and Nicholas J Wright. 2016. Modular HPC I/O Characterization with Darshan. In *2016 5th workshop on extreme-scale programming tools (ESPT)*. IEEE, 9–17.

[61] Guangyu Sun, Jenö Kürti, Péter Rajczy, Miklos Kertesz, Jürgen Hafner, and Georg Kresse. 2003. Performance of the Vienna Ab Initio Simulation Package (VASP) in Chemical Applications. *Journal of Molecular Structure: THEOCHEM* 624, 1-3 (2003), 37–45.

[62] Houjun Tang, Suren Byna, François Tessier, Teng Wang, Bin Dong, Jingqing Mu, Quincey Koziol, Jerome Soumagne, Venkatram Vishwanath, Jialin Liu, et al. 2018. Toward Scalable and Asynchronous Object-Centric Data Management for HPC. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 113–122.

[63] Osamu Tatebe, Shukuko Moriwake, and Yoshihiro Oyama. 2020. Gfarm/BB—Gfarm File System for Node-Local Burst Buffer. *Journal of Computer Science and Technology* 35, 1 (2020), 61–71.

[64] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus JJ Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Apra, Theresa L Windus, et al. 2010. NWChem: A Comprehensive and Scalable Open-Source Solution for Large Scale Molecular Simulations. *Computer Physics Communications* 181, 9 (2010), 1477–1489.

[65] Brian Van Essen, Hyojin Kim, Roger Pearce, Kofi Boakye, and Barry Chen. 2015. LBANN: Livermore Big Artificial Neural Network HPC Toolkit. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments* (Austin, Texas) *(MLHPC '15)*. ACM, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/2834892.2834897

[66] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. 2018. GekkoFS: A Temporary Distributed File System for HPC Applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 319–324.

[67] Murali Vilayannur, Samuel Lang, Robert Ross, Ruth Klundt, Lee Ward, et al. 2008. *Extending the POSIX I/O Interface: A Parallel File System Perspective*. Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).

[68] Murali Vilayannur, Partho Nath, and Anand Sivasubramaniam. 2005. Providing Tunable Consistency for a Parallel File Store. In *FAST*, Vol. 5. 2–2.

[69] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, and Elsa Gonsiorowski. 2020. Recorder 2.0: Efficient Parallel I/O Tracing and Analysis. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1–8.

[70] Feng Wang, Qin Xin, Bo Hong, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Tyce T McLarty. 2004. File System Workload Analysis for Large Scale Scientific Computing Applications. In *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*. 139–152.

[71] Teng Wang, Kathryn Mohror, Adam Moody, Weikuan Yu, and Kento Sato. 2015. BurstFS: A Distributed Burst Buffer File System for Scientific Applications. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[72] C Eric Wu. 1995. Parallel I/O Workload Characteristics Using Vesta. In *in Proceedings of the IPPS'95 Workshop on Input/Output in Parallel and Distributed Systems, IEEE Computer Society*. Citeseer.

[73] Qing Zheng, Kai Ren, and Garth Gibson. 2014. BatchFS: Scaling the File System Control Plane with Client-Funded Metadata Servers. In *2014 9th Parallel Data Storage Workshop*. IEEE, 1–6.