Analyzing Parallel Applications for Unnecessary I/O Semantics that Inhibit File System Performance

Sebastian Oeste[1(✉)], Michael Kluge[1], Ronny Tschüter[2], and Wolfgang E. Nagel[1]

[1] Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, Dresden, Germany
{sebastian.oeste,michael.kluge,wolfgang.nagel}@tu-dresden.de
[2] German Aerospace Center (DLR), Dresden, Germany
ronny.tschueter@dlr.de

Abstract. Scalability and performance of I/O intensive parallel applications are major concerns in modern High Performance Computing (HPC) environments. Almost all applications use POSIX I/O explicitly or implicitly through third party libraries like MPI-IO to perform I/O operations on the file system. POSIX I/O is known to be one of the lead causes of poor I/O performance due to its restrictive access semantics and consistency requirements.

Some file systems therefore relax specific POSIX semantics to alleviate I/O performance penalties. In order to make the most effective use of the offered file systems features it is required to know what kind of POSIX semantics an application requires. Existing tools can analyze parallel I/O performance to report type and duration of executed I/O operations. There are even tools that analyse the consistency requirements of data operations, but none that also consider perfromance critical patterns of metadata operations.

In this paper, we present a novel, systematic approach that groups parallel I/O operations and analyzes their I/O semantics with respect to POSIX I/O. We provide the tool *rabbitxx* that identifies concurrent overlapping accesses to the same file but also identifies metadata accesses such as concurrent create operations in the same directory. Our work indicates that POSIX defined I/O access semantics, in its current form, are often not necessary for parallel applications.

Keywords: Performance Analysis · I/O · POSIX · Semantics · File system · HPC

A. Bienz et al. (Eds.): ISC High Performance 2023 Workshops, LNCS 13999, pp. 161–176, 2023.
https://doi.org/10.1007/978-3-031-40843-4_13

1 Introduction

High Performance Computing (HPC) systems use parallel file systems to manage accesses from thousands of compute nodes to the back-end storage system. The file system interface defines available I/O operations along with their semantics. Many common parallel file systems, such as GPFS [21] or Lustre [2], provide a POSIX [19] compliant interface. POSIX was being developed with a strong focus on local file systems. Applied in a parallel distributed file system, its restrictive access semantics can cause considerable performance degradation [10,13,28]. In some cases, like file creations in a directory, POSIX I/O semantics enforce serial processing of operations issued in parallel. The lack of appropriate support of parallel I/O renders POSIX I/O (part of the POSIX standard) a major scalability and performance bottleneck for parallel applications.

Consequently, parallel file system developers have started to discard individual POSIX features or relax their consistency requirements to enable scalability for specific workloads. The NFS [22] file system relaxes the POSIX consistency for parallel accesses to the same file to a *close-to-open* consistency model. The PVFS [5] parallel file system supports non-conflicting write consistency semantics, leaving the results of overlapping writes undefined. Many tools [12,24,32] investigate and characterize parallel I/O performance of HPC applications. A lot of tools [12,24,26,32] provide insight into the usage of programming interfaces, I/O operation types, bandwidth, and files accesses. It has also been shown that strict write consistency is not requried by many HPC applications [31].

In this paper we boarden the view and also examine performance critical pattern of metadata operations. Therefore, we introduce a method that identifies application phases and builds sets of concurrent I/O operations within these steps. We check I/O operation sets for critical I/O access patterns that potentially limit performance and thereby reveal potential for relaxing POSIX I/O semantics of parallel file systems. We provide an exemplary implementation in form of *rabbitxx*[1], an open source, post-mortem analysis tool that identifies parallel I/O operations and investigates their access semantics with regard to POSIX. With this information, developers can optimize their applications, fine-tune I/O consistency requirements of their programs, and choose the most suitable storage infrastructure or file system configuration. Furthermore, parallel file system and I/O middleware engineers can investigate which consistency guarantees today's HPC applications require.

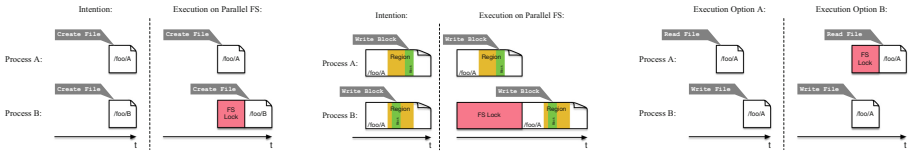
The remainder of this paper is organized as follows. Section 2 highlights performance critical I/O patterns to underline the importance of this work. Section 3 lists related work. Sections 5.1, 5.2, 5.3 present our methodology. First, we introduce an algorithm that creates sets of concurrent I/O operations from an event log of a parallel application. Then, we describe analysis modules that check these sets for different performance critical semantics. Section 6 demonstrates *rabbitxx* by analyzing two real world applications. We conclude and list future work in Sect. 7.

[1] https://github.com/blastmaster/rabbitxx.

2 Performance Critical Access Patterns

In this section we highlight three performance critical access patterns. The first two show limitations due to POSIX semantics. POSIX semantics enforce a strict serial processing for these patterns to prevent errors and guarantee data integrity. However, in a parallel use case scenario these semantics are 1) often completely unnecessary as applications can take care of data integrity more efficiently themselves, and 2) induce serious performance penalties. The last example demonstrates so-called anti-patterns that should be avoided in parallel applications.

Concurrent Create Operations within the Same Directory. POSIX requires strict consistency for file metadata and directory structures. Since parallel file creations may access the same directory structure (i.e., directory blocks), file systems have to lock it, see Fig. 1a. This is particularly relevant in parallel file systems where directory structures are globally accessible and all metadata operations have to be immediately visible on all nodes. Among others, this requires a large number of expensive global locking communication and can significantly impact the file system's metadata throughput [28].



(a) Concurrent create operations in one directory.

(b) Concurrent write operations to one file.

(c) Non-synchronized read/write operations to one file.

Fig. 1. Illustration of examples of performance critical access patterns.

Concurrent Overlapping File Accesses. The strict write consistency of POSIX requires that writes to the same region of a file are mutually exclusive. Thus, parallel writes appear as a sequential stream of operations, see Fig. 1b. The POSIX I/O semantics further dictate that write operations block until the data has been written to durable storage. Concurrently occurring read operations have to see the new data immediately. Local file systems can utilize the page cache to ensure this semantic restriction. However, parallel file systems need to hold locks and issue additional communication for this purpose.

In practice, the strict consistency semantics of write operations are unnecessary in case of non-overlapping accesses. Moreover, we assume that most HPC applications ensure non-overlapping accesses at a higher application level for scalability reasons.

Concurrent Read-Modify-Write Access to the Same Region of a File. The concurrent Read-Modify-Write access pattern describes the situation where process A reads a region of a file while process B writes to the same region of the file. This pattern indicates a possible data consistency violation. Depending on the runtime scheduling of both operations (process A reads before process B writes, or vice versa) it is undefined which "version" of the file process A reads, see Fig. 1c. In such a case the POSIX semantics only ensure that the read operation is not executed while the write operation is active. Therefore, distributed file systems need to employ global locking communication. We treat this as an anti-pattern for HPC-IO which requires stronger semantics from the underlying file system and poses performance penalties.

3 Related Work

The first paragraph of this section focuses on POSIX I/O semantics, whereas the second paragraph gives an overview on research related to I/O analysis.

POSIX I/O Semantics. The Portable Operating System Interface (POSIX) standard [19] defines portable interfaces between applications and the operating system. Its first formal specification dates back to 1988. The I/O section of the POSIX standard (henceforth referred to as POSIX I/O) comprises two essential parts. First, the POSIX I/O API specifies I/O operations and their syntactic requirements. Second, the POSIX I/O semantics define the behavior of I/O operations, e.g., guarantees with respect to consistency and correctness when using specific POSIX I/O operations. Some of these guarantees are very restrictive and impede efficient parallelization of I/O tasks [14]. The HPC community made efforts to extend POSIX I/O semantics for parallel I/O [30]. In 2006 there was a High End Computing Extensions Working Group [25].

Some parallel file systems discard certain POSIX I/O semantics to circumvent scalability issues. For instance, PVFS [5] supports non-conflicting consistency semantics and leave overlapping access to the same region of a file undefined. GekkoFS [27] is a job-temporal file system that relaxes POSIX I/O semantics for eventual consistency and improves scalability of metadata handling. Current research investigates object stores to explore trade-offs between scalability and flexible semantics [6,15].

I/O Analysis Tools. There is a number of tools to characterize I/O in the past decades [9,16,20]. For instance, Darshan [4] is widely known to characterize application I/O accurately. The Darshan tool is implemented as a set of user-space libraries and records counters for POSIX and MPI-IO operations. It yields statistics, such as MPI-IO adoption rate, number of files opened per job, or the ratio of read and write operations. ScalaIOTrace [29] records event logs including calls to MPI-IO and POSIX I/O. Data compression techniques utilize repetitive event sequences to keep event logs small, but preserve application's communication and I/O structure. Therefore, event logs recorded by ScalaIOTrace are

well-suited to replay I/O characteristics of applications. Méndez et al. [17] model I/O phases to estimate the runtime of an application for a certain I/O subsystem. They weight an I/O phase based on their repetitions and request size. Reed published together with different collaborators several papers related to I/O characterization [16,18,23]. The presented characterization is defined by three dimensions: type of operation, spatial and temporal access pattern. Byna et al. [3] present a five-dimensional characterization approach taking even the request size and number of repetition into account. They generate a signature of the I/O behavior to provide online prefetching strategies. The approaches mentioned above record statistics and metrics to describe the I/O behavior of applications or jobs on a cluster. These insights are useful to get an overview of the workload, identify performance critical operations, as well as detecting repetitive patterns.

Wang et al. [31] developed a method for detecting I/O accesses that can cause conflicts under weaker consistency models. Further, they provide a terminology for the categorization of the consistency semantics of parallel file systems. They found out that 16 of 17 applications can utilize file systems with weaker semantics. However, the algorithm to determine consistency semantics needs of an application uses only data operations. In contrast, our work also provides analysis modules for I/O patterns of performance critical metadata operations. Further, we analyze groups of parallel I/O operations, considers all combinations of their concurrent execution. Instead of being focused on the I/O events as they appear in an individual application run.

4 Methodology

This section gives an overview of our approach. The basis of our analysis method is information about I/O operations executed by an application.

Application Instrumentation: To acquire the necessary information from an application, we use the performance measurement infrastructure Score-P[2] [11,26]. Therefore, we intercept application calls to I/O and synchronization routines.

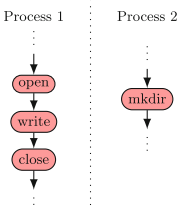
Application Execution and Trace Data Generation: When the instrumented application executes, it generates an OTF2 [7] trace file containing I/O and synchronization operations. A trace file represents an event log. Figure 2a shows information stored within a trace. Each event contains the time when the event occurred, the process that executed the event, and the type of that event. This information enables reconstruction of event sequences and their transformation into a graph structure (Fig. 2b).

Construction of the I/O Graph: We read the event sequences from the trace file and convert them into a *directed acyclic graph* (DAG) data structure.

[2] http://www.score-p.org/.

Time	Process	Operation
...		
21	1	open(f1)
42	2	mkdir(d2)
55	1	write(f1)
66	1	close(f1)
...		

(a) Application trace



(b) Graph representation

Fig. 2. Example of an application trace along with the corresponding graph representation of the event sequence.

Section 5.1 presents detailed information about the graph construction. Subsequent processing and analysis tasks work on the graph data structure.

Creation of Concurrent I/O Sets: First, we analyze each process individually and identify phases in its event sequence. A *Phase* contains all I/O operations between two synchronization points. Section 5.2 explains the phase identification step in more detail. Then, we expand our analysis over multiple processes. We identify corresponding phases across all processes and define sets of concurrent I/O operations (*CIO-Set*). A *CIO-Set* contains all I/O operations that can be executed concurrently on the participating processes. Consequently, the *CIO-Set* algorithm must not only handle the I/O event sequence as stored in the trace file. Multiple executions of the instrumented application might generate slightly different event sequence orders. Therefore, our creation algorithm also considers event permutations in order to find all possible *CIO-Sets*. We introduce the *CIO-Set* algorithm in Sect. 5.2.

Analysis of I/O Access Patterns and Their Semantics: Our analysis uses the *CIO-Sets* to evaluate whether the I/O access patterns of an application require specific POSIX I/O semantics. Analysis results reveal potential performance bottlenecks as well as options to relax POSIX I/O semantics for HPC applications. Section 5.3 presents currently implemented analysis modules.

5 Implementation

This section describes the implementation of our methodology presented in Sect. 4. We use the example introduced in Fig. 3 throughout this section to explain the construction of the I/O graph and the algorithm identifying the *CIO-Sets*.

5.1 Construction of the I/O Graph

An application trace builds the foundation for the I/O graph construction. We distinguish three kinds of vertices in the I/O graph: *I/O*, *synchronization* and

synthetic vertices. An I/O vertex represents an individual I/O event, e.g., a write request, or the creation of a new file. The synchronization vertex represents communication or synchronization between processes. These are either collective operations, e.g., `MPI_Barrier`, or blocking point-to-point operations, e.g., an `MPI_Send`, `MPI_Recv` pair. The synthetic vertices model a dedicated *Program-Start* and *Program-End* vertex. Trace event information (e.g., number of transferred bytes) is stored in the vertex properties.

Each I/O graph starts with a common synthetic *Program-Start* vertex. The *Program-Start* vertex connects to the first non-synthetic vertex of each process. From this point, each branch represents the execution of the respective process including I/O and synchronization events. A common synthetic *Program-End* vertex finalizes the I/O graph. An edge from the last non-synthetic vertex of each process connects to the *Program-End* vertex.

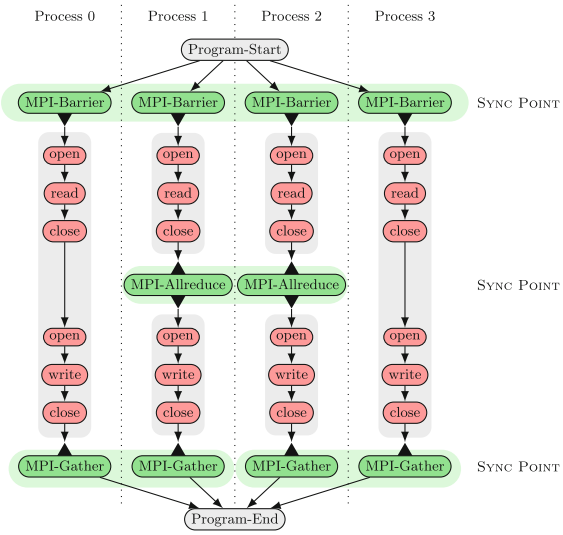


Fig. 3. Example of an I/O graph. The graph consists of synthetic (gray), I/O (red), and synchronization (green) vertices, synchronization points across processes (horizontal green bars), and *Phases* (vertical gray bars). A *Phase* of an individual process is enclosed by a *CIO-Start* (▼) and *CIO-End* (▲) event. (Color figure online)

Figure 3 illustrates the I/O graph constructed from the trace of a program running on four processes. The event sequence starts with an `MPI_Barrier` across all processes. Then, each process reads data, before an `MPI_Allreduce` operation synchronizes process 1 and 2. Subsequently, a second stage of I/O operations starts. The event sequence ends with two `MPI_Gather` operations. Processes 0 and 1 participate in the first operation, processes 2 and 3 in the second operation.

The first part of the I/O graph construction reads the trace data, analyzes the event, and constructs the corresponding vertex. After adding the vertex, the

next part creates an edge between the current vertex and the previous one of the process. This reflects the event flow for each process.

Table 1. Summary of notation

Notation	Description
Process-Group	The set of process IDs that participate on a synchronization event
Sync-List	The unique list of *CIO-End* events of the *Phases* in the current *View*
Sync-Pairs	List of all unique combinations of pairs of elements in *Sync-List*

5.2 Identification of Concurrent I/O Sets

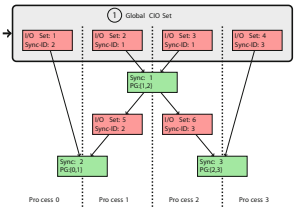
In order to identify sets of concurrent I/O operations, we process the I/O graph. While traversing the graph, we define sets of I/O events. Each I/O event is represented by its *Vertex ID*. Additionally, we collect the *Vertex ID*s of the *CIO-Start*/ *CIO-End* synchronization events which start/stop a *Phase*. Since programs do not have to start or end with synchronization, it is also valid to set a synthetic event as *CIO-Start* or *CIO-End* event.

Per-Process Phase Identification. We perform a depth-first-traversal on the graph to identify its *Phases*. A *Phase* can be in one of two states: *open* or *close*. A *Phase* is *open* as soon as the object is initialized. In other words, it has at least a *CIO-Start* event. A closed *Phase* means that the *CIO-End* event of the *Phase* was set and no further I/O events can be added to the *Phase*.

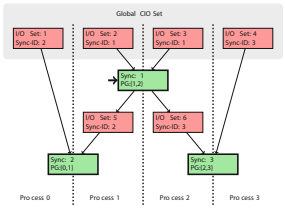
The phase identification finishes when all vertices are processed. The resulting *Phases* provide a local view of the I/O behavior of a single process. Consequently, we store the *Phases* in a per-process queue in the order of their occurrence. While defining global *CIO-Sets*, we process these queues and keep track of the currently active *Phase* in each queue. A *View* is a collection of the currently active *Phases* across all processes.

Global CIO-Sets. The algorithm to identify global *CIO-Sets* consists of four parts. The next paragraphs provide a detailed explanation of each part based on the I/O graph of our example (Fig. 3). Table 1 shows the notation used.

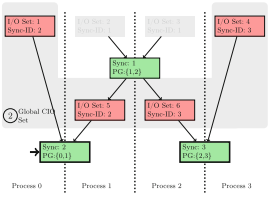
Initialize View. The algorithm starts with a *View* containing the first *Phases* of all processes. In our example, Fig. 4a highlights a *View* containing the first *Phases* of four processes. All I/O events in these *Phases* will be merged into a global *CIO-Set*. To decide how the *View* can be updated, the *CIO-End* event of the *CIO-Set* must be found.



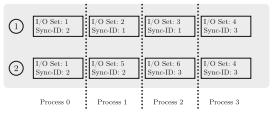
(a) The highlighted fields depict the *Phases* of the current *View*, which will be merged into a new *CIO-Set*.



(b) Identifying the boundaries of the *CIO-Set*. Synchronization 1 becomes the *CIO-End* of the *CIO-Set* of the current *View*.



(c) Updated the *View*. The new *View* contains the *Phases* for the next *CIO-Set*, the *CIO-End* is synchronization 2.



(d) Two overlapping *CIO-Sets* resulting from the merge part depicted in Figures 4a to 4c.

Fig. 4. Steps of the CIO-Set algorithm.

Determine Boundaries. The next part analyzes the current *View*, identifies participating *Phases* and inserts all *CIO-End* events contained therein into the *Sync-List*. In our example, this results in *Sync-List* $= \{2, 1, 3\}$ (Fig. 4a). The *CIO-End* events are denoted as *Sync-IDs* in Fig. 4a. Then, all unique combinations of pairs of the elements in *Sync-List* are built. There are $\frac{n(n-1)}{2}$ pairs, where n is the number of *CIO-End* events. In our example, this gives *Sync-Pairs* $= \{\{1, 2\}, \{1, 3\}, \{2, 3\}\}$. Next, for each pair in *Sync-Pairs*, the algorithm checks if the synchronization operations referred in the pair are dependent or independent. Two synchronization operations are called independent if the intersection of their *Process-Group* is empty. Otherwise, synchronization operations with processes participating in both synchronizations of the pair are called dependent. In our example, the synchronizations 2 and 3 depend on synchronization 1. In other words, some processes participating in synchronization 2 or 3 have a *Phase* in their current *View* where the *CIO-End* event refers to another synchronization. In this case, it refers to synchronization 1.

Define Global. CIO-Set Fig. 4b illustrates synchronization 1 as the *CIO-End* event of the first global *CIO-Set*. Therefore, the creation of the global *CIO-Set* is finished and the *CIO-Set* is saved for further analysis.

Update View. Next, all processes in the *Process-Group* of the *CIO-End* event update the *View* to their next *Phase*. In our example, processes 1 and 2 can

update to their next *Phase* (Fig. 4b). The *View* remains unchanged for process 0 and 3. Figure 4c shows the next part of the algorithm with an updated *View*. In this part, the *Sync-List* contains the synchronization events 2 and 3 ($Sync\text{-}List = \{2, 3\}$). Both synchronization events are independent, since $Process\text{-}Group\ (2) \cap Process\text{-}Group\ (3) = \emptyset$. When independent synchronizations are detected, the algorithm updates the *View* for each of them recursively. This is necessary because it is not guaranteed which of the independent synchronization events will occur first. Figure 4d shows both resulting *CIO-Sets* of the parts before. The resulting *CIO-Sets* are overlapping. *Phases* 1 and 4 of the processes 0 and 3 occur in both sets because they are not synchronized with other processes until synchronization 2 and 3. Therefore, they actually may occur in parallel with the *Phases* 2 and 3 but also with 5 and 6.

The algorithm continues until all *Phases* are processed. After the algorithm finished, we start to analyze the I/O behavior within *CIO-Sets*. In the next section, we discuss our analyses.

5.3 Analysis of I/O Access Patterns and Their Semantics

In this section, we describe the design and implementation of our analysis features. The *rabbitxx* design follows a modular approach. Each specific analysis task is implemented in its own analysis module. In a first step, the *rabbitxx* tool reads the event trace and calculates the *CIO-Sets*. In the scope of this work we present three analysis modules covering the three access patterns introduced in Sect. 2. All three modules produce a text-based report as output. The following paragraphs introduce each analysis module. The *rabbitxx* analysis capabilities are easily extendable by adding new analysis modules.

Module A: Concurrent Create Operations within the Same Directory. This analysis module filters `create` operations within a *CIO-Set* and checks if they target the same directory. If it detects such operations, it reports the *CIO-Set* number, the number and the function name of the `create` operations, and the affected directory.

Module B: Concurrent Overlapping File Accesses. This analysis module checks each *CIO-Set* for overlapping accesses. Therefore, it calculates the access intervals of `read` and `write` operations for each process. Then, for each file, it compares the respective intervals between all processes. If the module detects overlaps, it reports the *CIO-Set* number, the file name, the range that overlaps, and all affected processes.

Module C: Concurrent Read-Modify-Write Access to the Same Region of a File. This analysis module checks for all *CIO-Set*s if any read access will be overwritten by another process with a write operation to the same file offset. If the module detects this case, it reports the *CIO-Set* number, the file name, and all affected processes.

6 Experiments and Evaluation

This section presents the results of our analysis modules for two HPC benchmark applications. We choose two benchmarks which utilize I/O kernels from real-world HPC workloads. To verify our approach, we choose the MADbench2 [1] and HACC-IO [8] benchmark applications.

All tests were executed on *Taurus*[3], a generic BULL Linux Cluster at Technische Universität Dresden. Taurus consists of 2189 compute nodes with 43000 cores attached to a 5.1PB disk-based, and a 43TB SSD-based Lustre parallel file system.

6.1 MADbench2

MADbench2 is a benchmark based on the MADspec code, which calculates the maximum likelihood angular power spectrum of the Cosmic Microwave Background radiation. It operates on large floating-point matrices which are too large to hold them completely in main memory. Therefore, data will be written to disk and read back later.

We use MADbench2 in I/O-only mode, which skips calculations. We instrument and run the MADbench2 code on 16 and 64 processes. For the configuration, we choose POSIX as IOMETHOD in synchronous IOMODE. Because we want to investigate overlapping file accesses, we select "shared file access" as FILETYPE. Afterwards, we analyze the resulting trace with our tool *rabbitxx*. Independent of the number of application processes, our analysis identifies six different *CIO-Sets*. The resulting sets are non-overlapping because the MADbench2 code synchronizes with collective MPI synchronization routines encompassing all processes. Except from the number of processes and files accesses, the semantic analysis results are equal for all runs. Thus, we present only analysis results for the runs with 64 processes in the following.

Our first analysis module tests whether concurrent create operations appear in the same directory. The analysis report the application issues 64 create operations in the same directory within *CIO-Set* 2. The affected directory is located on the parallel file system. Consequently, all participating processes open the same file.

The second analysis module tests for overlapping accesses to the same address range within a file. Due to the strong consistency of write operations in POSIX I/O, overlapping writes can significantly impair the scalability of a parallel file system. Overlapping accesses to the same file region could also result in undefined behavior in case of file systems that relax POSIX I/O consistency, such as PVFS [5] or NFS [22]. Our analysis module reports no overlapping accesses for MADbench2. Nevertheless, Table 2 shows that all participating processes access the same file in *CIO-Sets* 2 to 5. Such a non-overlapping access pattern on a shared file is common for HPC applications. Our analysis of MADbench2 provides one example for such applications, that usually coordinate parallel accesses by themselves at a higher level.

Table 2. MADbench2: Concurrent accesses per file.

CIO-Set	File	#processes
2	/lustre/scratch2/.../MADbench2/files/data	64
3	/lustre/scratch2/.../MADbench2/files/data	64
4	/lustre/scratch2/.../MADbench2/files/data	64
5	/lustre/scratch2/.../MADbench2/files/data	64

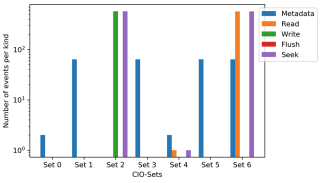


Fig. 5. HACC-IO: The number of operation types per *CIO-Set*.

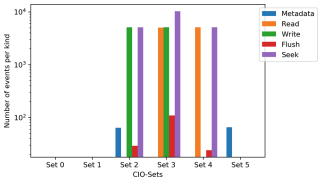


Fig. 6. MADbench2: Number of operation types per *CIO-Set*.

The third analysis module investigates whether concurrent read-modify-write accesses appear to the same region of a file. Our analysis module reports no such accesses for MADbench2. The application coordinates its accesses in a non-contiguous, non-overlapping manner. Figure 6 shows the number of operation types per *CIO-Set* for MADbench2. Concurrent read and write accesses occur only in *CIO-Set* 3. All other *CIO-Sets* perform either read or write or none of both. The results support the assumption that HPC applications typically issue their I/O operations in distinct phases.

6.2 HACC-IO

HACC-IO is an I/O-performance benchmark that computes an n-body simulation of collision-less fluids in space. HACC-IO uses random I/O write operations as well as all-to-all communication patterns. For our analysis we instrument HACC-IO with Score-P and run it with 16, 32, 64, and 128 processes. The *rabbitxx* analysis reports seven non-overlapping *CIO-Sets* for each of these configurations. Again, as all configurations show similar behavior, we present only results for the run with 64 processes here.

The analysis of concurrent creates report *CIO-Set* 1 and 5 issue 64 `open64` operations on the same file. The identified behavior is similar to MADbench2. The POSIX I/O semantics for `open` operations dictate that the file system needs to hold file descriptors for each accessing processes and a corresponding file description entry. If the accessed file does not exist, the file system has to create it and update the metadata entries. This metadata update requires additional locking and communication in distributed environments.

[3] https://doc.zih.tu-dresden.de/jobs_and_resources/hardware_overview/.

HACC-IO follows a "shared file access" pattern. Our analysis reports no overlapping accesses to the same address range of a file. However, Table 3 shows that *CIO-Sets* 1, 2, 3, 5, and 6 access the same file concurrently. This result strengthens our assumption that many HPC applications are designed for parallel file systems and avoid accesses to the same file with overlapping offsets.

Table 3. HACC-IO: Concurrent accesses per file.

CIO-Set	**File**	**#processes**
0	/lustre/scratch2/.../haccio/haccio-out.data	1
1	/lustre/scratch2/.../haccio/haccio-out.data	64
2	/lustre/scratch2/.../haccio/haccio-out.data	64
3	/lustre/scratch2/.../haccio/haccio-out.data	64
4	/lustre/scratch2/.../haccio/haccio-out.data	1
5	/lustre/scratch2/.../haccio/haccio-out.data	64
6	/lustre/scratch2/.../haccio/haccio-out.data	64

In analogy to MADbench2, our analysis does not report any read-modify-write file access patterns. No *CIO-Set* of HACC-IO contains simultaneous read and write operations. Figure 5 shows that all concurrent write operations are executed in *CIO-Set* 2, while read operations occur in *CIO-Set* 4 and 6.

In summary, our analysis of MADbench2 and HACC-IO shows that both applications concurrently create files in the same directory. Furthermore, they perform non-conflicting, non-overlapping accesses to shared files. Both applications calculate offsets and use `seek` operations to manage their file accesses. We conclude that these kinds of applications do not require the atomicity and strong consistency for read and write operations defined by POSIX I/O. In addition, the POSIX I/O semantics for metadata consistency and file creation represent a potential performance bottleneck that parallel application cannot circumvent with the current API. HPC applications can benefit from the concept of file descriptors shared across multiple processes or a collective file open operation for multiple processes. Moreover, both applications were only accessing files in a single directory. Consequently, strict consistency of a global namespace as dictated by POSIX I/O seems unnecessary for many HPC applications. Instead, we propose application-private namespaces where consistency requirements hold for participating nodes, not the whole cluster. The analysis of required I/O semantics of HPC applications indeed requires further research.

7 Conclusion and Future Work

This paper presents a systematic approach to analyze parallel applications for understanding the need of specific I/O semantics. Based on the event log of a

parallel application, we generate a graph that preserves the happens-before relation of events on individual processes and reflects synchronization points across multiple processes. We present a two-stage algorithm using this information to identify all sets of concurrent I/O events. As a result, we get all sets of I/O events that may occur in parallel within an application – either as observed in the given event log or in any other parallel execution order with the given synchronization operations. Then, we analyze the I/O behavior and determine which POSIX I/O semantics are unnecessary for the application.

Our evaluations indicate that the fundamental semantic restrictions implied by the POSIX I/O standard might not be necessary for many HPC applications. Dropping of unnecessary semantics would remove significant performance bottlenecks from the parallel file system. One example of this is the strict consistency of write operations. The results of our analyses reveal potential for replacing or relaxing POSIX I/O semantics in future storage systems.

In the future, we plan to add more analysis capabilities to *rabbitxx*.

References

1. Borrill, J., Carter, J., Oliker, L., Skinner, D., Biswas, R.: Integrated performance monitoring of a cosmology application on leading HEC platforms. In: 2005 International Conference on Parallel Processing (ICPP 2005), pp. 119–128. IEEE (2005). https://doi.org/10.1109/ICPP.2005.47, https://ieeexplore.ieee.org/document/1488607/
2. Braam, P.J., Zahir, R.: Lustre: a scalable, high performance file system. Clust. File Syst. **8**(11), 3429–3441 (2002). https://cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf
3. Byna, S., Chen, Y., Sun, X.H., Thakur, R., Gropp, W.: Parallel I/O prefetching using MPI file caching and I/O signatures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 1–12. IEEE (2008). https://doi.org/10.1109/SC.2008.5213604
4. Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., Riley, K.: 24/7 characterization of petascale I/O workloads. In: 2009 IEEE International Conference on Cluster Computing and Workshops, pp. 1–10 (2009)
5. Carns, P., Ligon, W., Ross, R., Thakur, R.: PVFS: a parallel file system for Linux clusters. In: 4th Annual Linux Showcase and Conference, vol. 4, pp. 1–11 (2000)
6. Danilov, N., Rutman, N., Narasimhamurthy, S., Bent, J.: Mero: co-designing an object store for extreme scale (2016). https://www.pdsw.org/pdsw-discs16/wips/danilov-wip-pdsw-discs16.pdf
7. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W.E., Wolf, F.: Open trace format 2: the next generation of scalable trace formats and support libraries. In: De Bosschere, K., D'Hollander, E.H., Joubert, G.R., Padua, D., Peters, F., Sawyer, M. (eds.) Applications, Tools and Techniques on the Road to Exascale Computing. Advances in Parallel Computing, vol. 22, pp. 481–490. IOS Press (2012). https://doi.org/10.3233/978-1-61499-041-3-481
8. Habib, S., et al.: The universe at extreme scale: multi-petaflop sky simulation on the BG/Q. In: SC 2012: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 4:1–4:11. IEEE (2012). https://dl.acm.org/citation.cfm?id=2388996.2389002

9. He, J., et al.: Discovering structure in unstructured I/O. In: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 1–6. IEEE (2012). https://doi.org/10.1109/SC.Companion.2012.11
10. Hildebrand, D., Nisar, A., Haskin, R.: pNFS, POSIX, and MPI-IO: a tale of three semantics. In: Proceedings of the 4th Annual Workshop on Petascale Data Storage, PDSW 2009, pp. 32–36. ACM (2009). https://doi.org/10.1145/1713072.1713082, https://portal.acm.org/citation.cfm?doid=1713072.1713082
11. Knüpfer, A., et al.: Score-P: a joint performance measurement run-time infrastructure for periscope, Scalasca, TAU, and Vampir. In: Brunst, H., Müller, M.S., Nagel, W.E., Resch, M.M. (eds.) Tools for High Performance Computing 2011, pp. 79–91. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31476-6_7
12. Kunkel, J.M., et al.: The SIOX architecture – coupling automatic monitoring and optimization of parallel I/O. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2014. LNCS, vol. 8488, pp. 245–260. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07518-1_16
13. Latham, R., Ross, R., Thakur, R.: The impact of file systems on MPI-IO scalability. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 87–96. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30218-6_18
14. Lockwood, G.: What's so bad about Posix I/O (2017). https://www.nextplatform.com/2017/09/11/whats-bad-posix-io/
15. Lofstead, J., Jimenez, I., Maltzahn, C., Koziol, Q., Bent, J., Barton, E.: DAOS and friends: a proposal for an exascale storage system. In: SC 2016: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 585–596 (2016). https://doi.org/10.1109/SC.2016.49
16. Madhyastha, T.M., Reed, D.A.: Learning to classify parallel input/output access patterns. IEEE Trans. Parallel Distrib. Syst. **13**(8), 802–813 (2002). https://doi.org/10.1109/TPDS.2002.1028437
17. Méndez, S., Rexachs, D., Luque, E.: Modeling parallel scientific applications through their input/output phases. In: 2012 IEEE International Conference on Cluster Computing Workshops, pp. 7–15. IEEE (2012). https://doi.org/10.1109/ClusterW.2012.37
18. Oly, J., Reed, D.A.: Markov model prediction of I/O requests for scientific applications. In: Proceedings of the 16th international conference on Supercomputing, ICS 2002, pp. 147–155. ACM (2002). https://doi.org/10.1145/514191.514214, https://doi.acm.org/10.1145/514191.514214
19. 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), pp. 1–3951 (2018)
20. Sayed, S.E., Bolten, M., Pleiter, D., Frings, W.: Parallel I/O characterisation based on server-side performance counters. In: Proceedings of the 1st Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems (PDSW-DISCS), pp. 7–12. IEEE Press (2016). https://doi.org/10.1109/PDSW-DISCS.2016.006
21. Schmuck, F., Haskin, R.: GPFS: a shared-disk file system for large computing clusters. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST 2002, pp. 231–244. USENIX Association (2002). https://dl.acm.org/citation.cfm?id=1083323.1083349
22. Shepler, S., et al.: Network file system (NFS) version 4 Protocol. RFC 3530, RFC Editor (2003). https://www.rfc-editor.org/pdfrfc/rfc3530.txt.pdf

23. Smirni, E., Reed, D.A.: Lessons from characterizing the input/output behavior of parallel scientific applications. Perform. Eval. **33**(1), 27–44 (1998). https://doi.org/10.1016/S0166-5316(98)00009-1, http://www.sciencedirect.com/science/article/pii/S0166531698000091
24. Snyder, S., Carns, P., Harms, K., Ross, R., Lockwood, G.K., Wright, N.J.: Modular HPC I/O characterization with darshan. In: Proceedings of the 5th Workshop on Extreme-Scale Programming Tools, ESPT 2016, pp. 9–17. IEEE (2016). https://doi.org/10.1109/ESPT.2016.9
25. The Open Group: High End Computing Extensions Working Group (2006). https://collaboration.opengroup.org/platform/hecewg
26. Tschüter, R., Herold, C., Wesarg, B., Weber, M.: A methodology for performance analysis of applications using multi-layer I/O. In: Aldinucci, M., Padovani, L., Torquati, M. (eds.) Euro-Par 2018. LNCS, vol. 11014, pp. 16–30. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96983-1_2
27. Vef, M.A., et al.: GekkoFS - a temporary distributed file system for HPC applications. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 319–324 (2018). https://doi.org/10.1109/CLUSTER.2018.00049
28. Vef, M.A., Tarasov, V., Hildebrand, D., Brinkmann, A.: Challenges and solutions for tracing storage systems: a case study with spectrum scale. ACM Trans. Storage **14**(2), 18:1–18:24 (2018). https://doi.org/10.1145/3149376, https://doi.acm.org/10.1145/3149376
29. Vijayakumar, K., Mueller, F., Ma, X., Roth, P.C.: Scalable I/O tracing and analysis. In: Proceedings of the 4th Annual Workshop on Petascale Data Storage, PDSW 2009, pp. 26–31. ACM, New York (2009). https://doi.org/10.1145/1713072.1713080, https://doi.acm.org/10.1145/1713072.1713080
30. Vilayannur, M., Lang, S., Ross, R., Klundt, R., Ward, L.: Extending the POSIX I/O interface: a parallel file system. Perspective (2008). https://doi.org/10.2172/946036, http://www.osti.gov/servlets/purl/946036-pnI90N/
31. Wang, C., Mohror, K., Snir, M.: File system semantics requirements of HPC applications. In: Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2021, pp. 19–30. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3431379.3460637
32. Wang, C., Sun, J., Snir, M., Mohror, K., Gonsiorowski, E.: Recorder 2.0: efficient parallel I/O tracing and analysis. In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1–8. IEEE (2020)