Parallel I/O characterization and evaluation on large scale HPC systems

Hammad Ather

Area Exam
Department of Computer Science
University of Oregon
Eugene, OR, USA
hather@uoregon.edu

***Abstract*—Recent technological advances have led to advanced parallel computing hardware and complex I/O workloads, comprising Machine Learning, Deep Learning, and other artificial intelligence techniques. These advances have made the existing parallel I/O stack more complex and challenging to tune which if not optimized properly, can lead to massive overheads and performance degradation. With these ever-increasing complexities of the I/O stack deployed on large-scale systems, one needs to have an in-depth understanding of the I/O behavior of these systems and be aware of the performance modeling and prediction tools required to evaluate and optimize I/O. Therefore, it is critical to have a comprehensive study that end users can use as a guide to evaluate and optimize parallel I/O in their applications. This paper presents such a study by surveying the current landscape of parallel I/O characterization and evaluation on large-scale HPC systems. By taking a deep dive into the different layers of the I/O stack, this paper shows how the different access patterns are shaped as an I/O request traverses down the I/O stack and what optimizations can be made to these access patterns. The paper also looks at different workload generation methodologies and the different profiling and tracing tools that can collect performance statistics for these workloads. It also discusses different parallel I/O evaluation techniques such as statistical analysis, machine learning, and replay-based modeling. Lastly, it ties this whole discussion with the current active area of research in parallel I/O: automatically evaluating, analyzing, and optimizing parallel I/O in applications without involving an I/O expert in the loop.**

***Index Terms*—Parallel I/O, Performance Evaluation, HPC Systems, I/O Characterization**

I. Introduction

In recent years, there have been significant advances in parallel processing hardware, which has propelled HPC systems to perform computations at a much more massive scale [1]. These computations also bring a vast amount of data, which, if not appropriately managed, can add significant performance overhead and impact the application's scalability. Many times, the overhead comes from unoptimized parallel I/O. Therefore, ensuring fast and efficient parallel I/O in large-scale HPC systems is critical.

The parallel I/O stack deployed on large-scale computing systems is complex and difficult to tune. Usually, there is an interplay of multiple factors that impact the performance of data movement between storage and computing systems. Also, each stack layer has many tuning parameters and optimization techniques that can improve application performance. I/O performance optimization is a complex problem due to the multiple factors that can affect performance, such as the interdependencies among the stack layers.

When applications suffer slowdowns, pinpointing the root causes of inefficiencies requires detailed metrics and an understanding of the stack and the different I/O access patterns. These access patterns, which have been identified through numerous research studies, can have a direct impact on the overall performance of the application, therefore it is crucial to optimize these accesses. Many different optimization techniques such as collective I/O and data sieving can optimize these access patterns and lead to better performance.

To understand the I/O performance, knowing different workload generation techniques, such as benchmarks and simulations, is essential. Benchmarks play an important role in understanding the I/O performance of large-scale applications without actually incurring the overhead of running these large-scale applications. Understanding how to collect performance measurements using profiling and trace analysis tools of different I/O workloads is crucial in understanding the I/O performance. There are also many statistical analysis, predictive analysis, and replay-based analysis techniques that further analyze and evaluate the data collected by these tools and help in further understanding the I/O performance of the application.

As can be inferred from the discussion above, understanding and evaluating parallel I/O on large-scale systems can be difficult, as multiple steps involve. With the vast amount of tools and techniques available, the user might also face a dilemma in deciding which tool better serves their I/O optimization needs. To make this process easier, the end user needs to have a guide that consolidates all the information related to parallel I/O evaluation in a step-by-step manner.

This paper presents a holistic survey of the current state of the art on large-scale parallel I/O analysis and evaluation techniques in applications running on HPC systems. It surveys numerous papers on parallel I/O and provides a step-by-step process for evaluation of the I/O performance of an application. The paper first provides an in-depth detail of the

different layers of the HPC I/O stack and how these layers interact with each other. Then, it defines different I/O access patterns identified through research studies and studies how these I/O access patterns are shaped as an I/O request goes down the stack. It looks at different optimization techniques for access patterns such as data sieving, collective I/O, etc. which have been widely studied in different research studies. After looking at the HPC I/O stack and the different access patterns, the paper provides a taxonomy of the different I/O performance evaluation strategies, which is the crux of this paper. It looks at different stages of parallel I/O evaluation, such as workload generation, data monitoring and collection, and different analysis techniques and tools. To sum it up, this paper, by studying various research studies on parallel I/O evaluation, answers the following questions:

- What are the different layers of the HPC I/O stack, and how do I/O access patterns affect the I/O behavior?
- What are the different profiling and tracing tools available to characterize the I/O behavior of the application?
- What analysis techniques are HPC researchers applying for characterizing and analyzing the I/O behavior?

The rest of the paper is organized as follows: Section II looks at the different layers of the HPC I/O stack, section III talks about different I/O access patterns and some optimizations for these patterns, section IV discusses the different stages of parallel I/O evaluation and optimization and is the main focus of this paper. Lastly, section V presents the conclusion of this survey.

II. HPC I/O Stack

The complex I/O workloads of serial and parallel applications running on large-scale HPC systems are supported by the HPC I/O stack [2]. The I/O stack is complex and has multiple layers, each with many tuning parameters that can be used to improve the application performance [1], [3]. Fig. 1 shows the multi-layered I/O stack deployed on HPC systems. As can be seen from the figure, the I/O stack has multiple layers, such as the high-level I/O libraries, parallel I/O middleware, low-level I/O libraries, I/O forwarding layer, and the parallel file system between the applications and storage hardware.

A. High-level I/O libraries

High-level I/O libraries provide abstractions for data modeling and management, allowing portability and high application performance. Some of the commonly used high-level I/O libraries include HDF5 [4], ADIOS [5], NetCDF [6], and PnetCDF [7]. The HDF5 technology suite comprises a data model, a library, and a file format to store and manage data [8]. HDF5 is portable and easily extensible, allowing it to support a variety of data types and store and manage complex I/O data. NetCDF (Network Common Data Form) supports creating, accessing, and sharing array-oriented scientific data using software libraries and machine-independent data formats [9]. The NetCDF-4 package can use HDF5 for storing the data while using the original netCDF library. PnetCDF uses parallel I/O techniques to provide a high-performance and

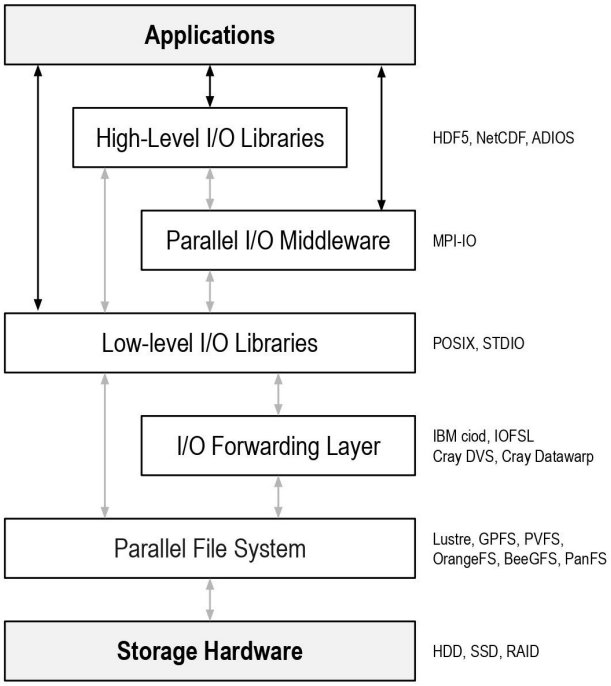


Fig. 1: Traditional parallel I/O stack deployed in production scale supercomputers [2]

efficient interface to access netCDF files. The Adaptable IO System (ADIOS) provides an efficient and flexible solution for researchers and scientists to describe the data in their code as an external to-the-code XML file for processing outside the running simulation. All of these libraries mentioned above map the abstractions of applications' data and encode the applications' data into portable file formats. They also allow the user to describe the data and the different structures in the file in the form of metadata. Apart from these high-level I/O libraries, there are also some domain-specific I/O libraries as well such as FITS [10], [11] and ROOT [12], which are used for High-Energy Physics (HEP) and astronomy.

B. Parallel and low-level I/O libraries

The parallel I/O middleware and low-level I/O libraries include MPI-IO (Message Passing Interface I/O) [13], POSIX I/O (Portable Operating System Interface I/O) [14], and STDIO (Standard Input and Output) interfaces. Users can also use these interfaces directly to perform I/O to the file systems. MPI-IO provides a low-level interface for performing parallel I/O. In MPI-IO, a file is defined as an ordered collection of typed data items. Using these typed data items, MPI-IO allows the user to define data models for their applications. It also provides two types of I/O calls which are independent and collective I/O. Independent MPI I/O calls are defined as those types of calls made by any subset of the processes participating in I/O, with each process handling its I/O independently. Some of the basic independent MPI I/O calls are `MPI_File_read()` and `MPI_File_write()`. Collective MPI I/O calls are those I/O calls that are made by all processes participating in a particular I/O sequence. The

basic collective calls are `MPI_File_write_all()` and `MPI_File_read_all()`. POSIX I/O, on the other hand, views a file as a sequence of bytes. Through this interface, contiguous regions of bytes can be transferred between the file and memory, and non-contiguous regions of bytes can be transferred from memory to a file by giving complete low-level control of the I/O operations. The major drawback of POSIX I/O in the context of HPC is that it supports very little parallel I/O. For example, ensuring collective access to files is not easy in POSIX and the user has to manually coordinate access and ensure consistency. It is also not flexible in terms of file metadata as it prescribes a specific set of metadata that a file must possess [15]. STDIO in contrast, provides abstractions to deal with the stream of input and output bytes. It is comprised of the C stdio.h family of functions [16], such as `fopen()`, `fprintf()`, and `fscanf()`. These I/O functions are commonly used in many parallel I/O applications; however, STDIO functions do not directly support random access to data files. It relies on the user to create an input/output stream, seek the position in the file from where to read or write, and then read/write bytes in sequence from/to the stream.

C. I/O forwarding layer

The I/O forwarding layer aims to reduce the number of clients concurrently accessing the Parallel File System (PFS) by providing an additional transparent layer between the compute nodes and the data servers in the form of I/O nodes. Instead of the applications directly accessing the PFS, requests are first received by these I/O nodes which forwards these requests to the PFS in a manageable way. This technique allows a smooth flow of I/O requests, providing a layer between the application and file system, and makes possible certain optimization techniques such as aggregation, request scheduling, and compression. The I/O forwarding layer technique was initially introduced for the Blue Gene/L system [17], but now has been extended to many of the top 500 supercomputers. One such example is the Tianhe_2 supercomputer which has 256 intermediate I/O nodes. These nodes, powered with high-speed SSDs, have configurations for each file which determines when data is transferred from the I/O nodes to the PFS.

D. Parallel File System

Large-scale HPC systems rely on Parallel File Systems (PFS) to provide an infrastructure for persistent shared storage and a global namespace across distributed storage servers to read and write to files. A PFS mainly has two types of servers: the data server and the metadata server. Metadata servers are responsible for handling the file metadata, which is information related to the size, permissions, and location of the file on the data servers. Before any client can access the data in a file, they must obtain the layout information of the file from the metadata. Accessing the metadata can also add some overhead, so some systems cache the metadata on the client side for faster access. However, that can lead to issues such as cache coherence, especially when many clients are concurrently accessing the PFS. On the data server, the files are distributed using data striping [18]. In the technique, the file is divided into fixed-size chunks called stripes, and the stripes are distributed to the servers in a round-robin approach. A PFS can retrieve stripes from different servers in parallel, hence increasing throughput. Different machines have different stripe sizes for example the Google File System has a stripe size of 64MB. PVFS [19], Lustre [20], [21], and GPFS [22] have default stripe sizes between 64KB and 1MB. To keep consistency when having concurrent accesses, some PFS systems use locking on the server. Lustre is one such system that uses stripe granularity, i.e., multiple clients can not access the same stripe concurrently. However, other systems, like PVFS, allow the users to deal with consistency for simplicity and better performance. PFS systems also provide mechanisms for fault tolerance which include measures such as replication of data and metadata. This is achieved by keeping mirrored servers. There can be a performance overhead for copies of the same data updated across the mirrored servers however having the same data on more than one server can improve performance by allowing parallel access. Some popular file systems include Lustre, PVFS, IMB GPFS, and the Panasas file system [23].

E. Storage Hardware

There are a variety of storage hardware used in a supercomputer. HDDs [24] are a popular storage medium that has been in use for many years now. They are made up of magnetic-surfaced rotating platters. To access any data in an HDD, a seek operation has to be performed in which the head has to be moved to the proper location. HDDs give the best performance when the underlying data needs to be accessed sequentially instead of randomly, as that reduces the overhead of the seek operation [25]. A recent flash-based alternative to hard disks is SSDs. SSDs have higher bandwidth and less overhead for sequential access than HDDs.

Summary #1

The I/O stack deployed on large scale HPC systems is complex and comprises of multiple layers with numerous tuning parameters aimed at improving application performance and supporting the diverse requirements of HPC systems. There are high-level I/O Libraries like HDF5, NetCDF, PnetCDF, and ADIOS which provide abstractions for data modeling and management, enhancing portability and performance. Then there are Parallel and Low-level I/O Libraries such as MPI-IO, POSIX I/O, and STDIO which provide parallel I/O support. The I/O Forwarding Layer is introduced to optimize access to Parallel File Systems (PFS) and acts as an intermediary between compute nodes and data servers. Parallel File System (PFS) provides persistent shared storage across distributed servers, divided into data and metadata servers. Storage Hardware provides mediums such as HDDs and SSDs.

III. Tuning the HPC I/O Stack

The parallel I/O stack deployed on large-scale computing systems has many tuning parameters and optimization techniques to improve application I/O performance [3], [26]. There can be multiple factors that can affect the I/O performance of an application; therefore, harnessing I/O performance is a complex problem due to the multiple factors that can affect it and inter-dependencies among the layers of the stack. Much research has been done on optimizing the way applications perform their I/O and tuning the different I/O layers. In the next section, we will look at the application's access pattern and different optimizations that can applied to it. We also look at different methods for I/O tuning of the HPC stack.

A. Application's Access Patterns

I/O requests by an HPC application can be issued in diverse ways depending on how the application is modeled and coded. Although there is no set definition to describe the access patterns of an application, a lot of research has been done on HPC I/O applications to understand what can constitute an access pattern, which includes the number of issued requests, the requests' sizes, and their spatial location in the file [27]. These patterns have a direct impact on the overall performance of the application therefore it is really important to optimize these data accesses [28], [29].

There are three classifications of the access patterns: local, global, and system-wide. The local access patterns of an application are in the context of an individual process or a task. They are employed to identify and optimize the client side of the application. The global access pattern looks at all the processes and tasks. It is more applicable in optimizing the forwarding layer or the file system as they have a complete picture of all the accesses made by each process. System-wide access patterns look at the access patterns revolving around shared storage infrastructure and the I/O nodes and help optimize the system-wide performance. Fig. 2 presents a taxonomy of the different access patterns discussed in this paper.

1) Access Patterns Features: In this section, we briefly look at some of the features that can be used to describe an I/O access pattern and their usage in the I/O stack.

a) Operation: The first access pattern is the basic I/O operations, such as read and write. It also includes the append operation, which performs a write operation to append the data to a file but first positions the file offset at the end of the file using a seek operation. The write operation and the moving of file offset are considered a single atomic operation.

b) File Approach: The second access pattern is how processes perform parallel I/O and access the files to read or write data. There can be multiple scenarios in which MPI ranks access the files. The first scenario, as shown in Fig. 3(a), is the file-per-process approach in which each MPI rank writes to an individual file. If the number of processes is too large, a subfiling approach [30] is also used in which data is aggregated into a small subset of processes that access a small number of files. The problem with the file-per-process approach and the subfiling approach is that the post-processing of the data in the files is complex as the data is scattered across multiple files.

The second scenario has a shared file for all the processes. Typically, in this scenario, there is a single rank (commonly rank 0), which receives the data from all the other processes or ranks, aggregates it, and writes it to the shared file. This approach is shown in Fig. 3(b). The limitation of this approach is that it can have a significant memory overhead on the aggregator node, which receives and aggregates the data from all the ranks. To tackle this issue and use the memory more efficiently, we can have multiple ranks that aggregate the data and write to a file. This approach is shown in Fig. 3(c). Another approach is to have all the ranks write to pre-defined and non-overlapping locations in a file, as shown in Fig. 3(d) using implicit coordination. The drawback of this approach is that the I/O requests are not aggregated between the ranks on the same compute node.

c) Spatial Locality: The spatial locality or spatiality directly impacts the I/O performance of the application. Spatiality refers to the file offsets between consecutive I/O accesses. Generally, three types of spatial accesses are sequential (contiguous), strided, and random. The type of spatial access directly correlates with the I/O performance of the application, as the storage hardware and software are directly affected by the data accessed in it. A file system can perform better if the data is accessed in a specific pattern [31].

In sequential or contiguous access, each process accesses the file in contiguous chunks, and that property holds for all the I/O requests. These types of accesses are common in the file-per-process approach. In strided access, chunks of the file are accessed by the process with a fixed-size gap called a stride. After each request, the file pointer is increased by the same size. Strided accesses are more common in shared files. In random file accesses, the file is accessed randomly without any set pattern. Because of the performance overhead added by random accesses to file, such accesses are uncommon in traditional HPC workloads [32].

d) Interfaces: Interfaces are essential in parallel I/O as they define the APIs and semantics for accessing the data in files. Some of the most common I/O interfaces are POSIX I/O, MPI-IO, and STDIO. We will discuss each interface in detail, also looking at their usability in terms of performing parallel I/O.

POSIX (Portable Operating System Interface) [14] composes a set of standards defined by IEEE to maintain compatibility between different operating systems, allowing an application to run the basic features of an operating system. POSIX also provides an I/O API, called POSIX I/O, to deal with the parallel file system. Introduced in 1988, POSIX I/O introduced asynchronous and synchronous behaviors for local file systems. One of the significant advantages of POSIX I/O was that it was highly portable. However, this portability comes with a price, as there are consistency requirements for POSIX write operations in which the application execution has to be stopped for a write call until a read call reads the data

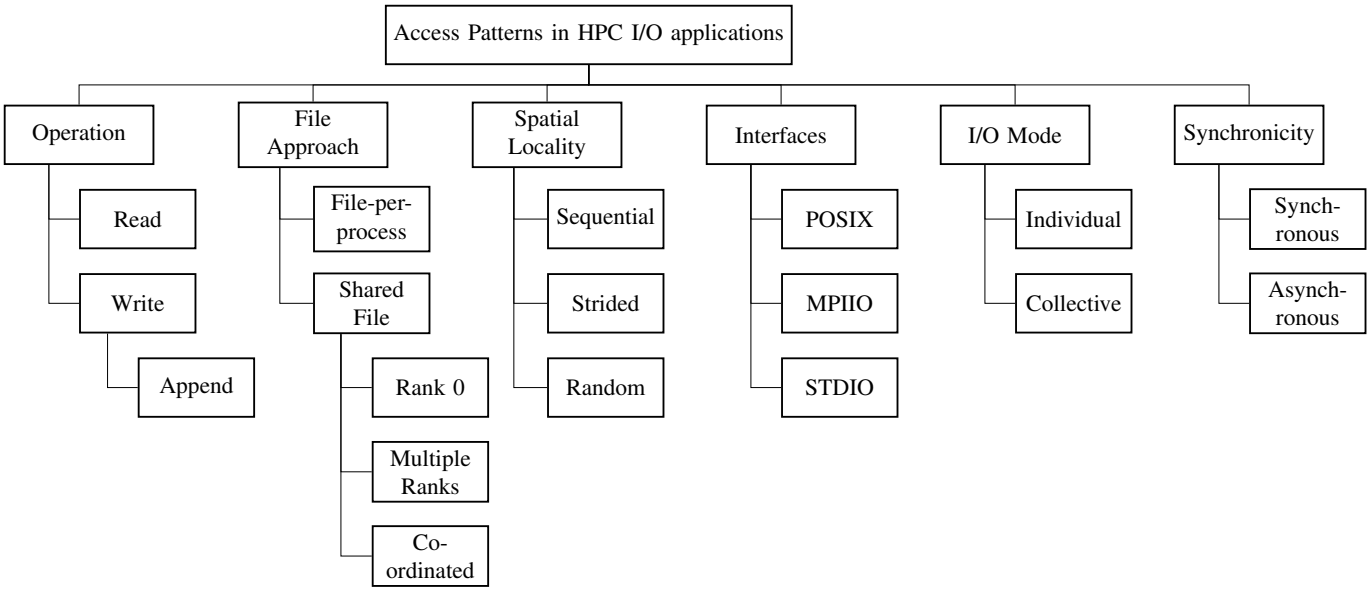


Fig. 2: Access patterns of HPC I/O applications identified using various research studies

just written. This introduces complexity in using POSIX I/O for large-scale distributed and parallel file systems, as these systems have to adhere to strict consistency requirements using some locking mechanism. This can limit the parallelism of the file system at a large scale. POSIX also leaves the burden of coordinating parallel accesses, buffering, and flushing to the end user, which can be problematic for the user if, for example, they are working in a shared-file parallel I/O approach.

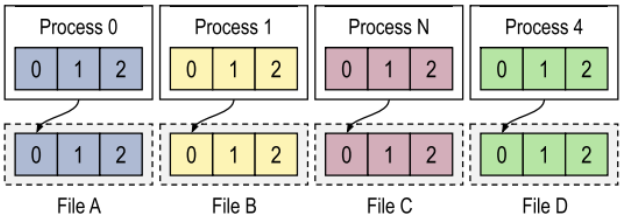
MPI-IO [13] is an extension to the MPI standard and provides an I/O API that comprises the message-passing capabilities of MPI. MPI-IO provides a high-level interface to define the data partitioning among the processes and supports asynchronous I/O. This allows I/O operations to run in parallel with computation. MPI-IO also has relaxed consistency requirements as compared to POSIX. MPI-IO is built on top of ADIO [33], which is an abstract device interface for parallel I/O. ADIO makes the implementation of parallel I/O interfaces portable and efficient by allowing any parallel API to be implemented on top of ADIO. It is not intended to be used directly by programmers and is only an abstraction for building parallel I/O interfaces on top of it. ROMIO [34] is one such example, which is a portable implementation of MPI-IO and uses ADIO to develop this implementation. MPI-IO also provides features related to data access, such as blocking or non-blocking and independent or collective accesses. The I/O access patterns generated by MPI-IO are complex, and these patterns can be modified as they traverse the stack because of optimizations and transformations made by MPI-IO.

STDIO (Standard I/O library) abstracts all file operations to provide an interface to perform operations on a stream of bytes. It consists of the C stdio.h family of functions such as `fprintf, fscanf, fopen` etc [16]. STDIO does not support random access to data and the application must open a stream, seek to the desired offset, and then read/write the data to the file. STDIO has been increasingly used for various HPC workloads such as in genomics and biology to store sequencing information in textual format [32], [35]. Even though studies [36] have shown that the use of STDIO has increased in supercomputers, it still performs poorly on different transfer sizes on Cori and Summit and hence shows poor I/O performance overall.

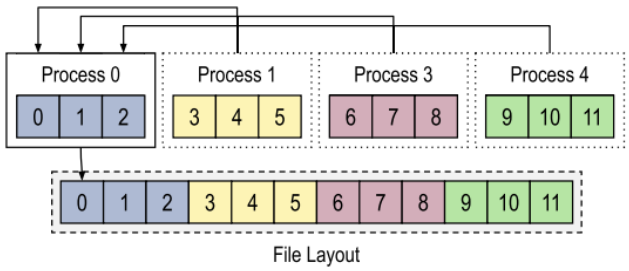
e) I/O Mode: I/O mode defines how a parallel process accesses a file. There are two I/O modes: individual and collective. In individual mode, each process (MPI rank) accesses the file individually, whereas, in collective I/O mode, all the ranks accessing the same file issue a single collective call to access that file. Collective I/O is considered a critical optimization in parallel I/O and is readily available in MPI-IO. It is particularly effective when the file accesses of different processes are non-contiguous and interleaved. This mode allows for optimizations such as collective buffering and data sieving [37]. A basic diagram of how collective I/O works is shown in Fig. 4

Collective I/O mode works in a two-phased strategy. In the first communication phase, all the processes participating in the collective I/O call send their data to the "aggregator" nodes. In the I/O phase, the "aggregator" issues the read or write request to the system. This allows larger and contiguous access to the file system even though the individual requests are non-contiguous. One example of a collective I/O function is `MPI_File_write_at_all`. Here `_all` indicates that all the processes in the MPI communicator will call the write function and `_at` indicates that the position in the file is specified as part of the call.

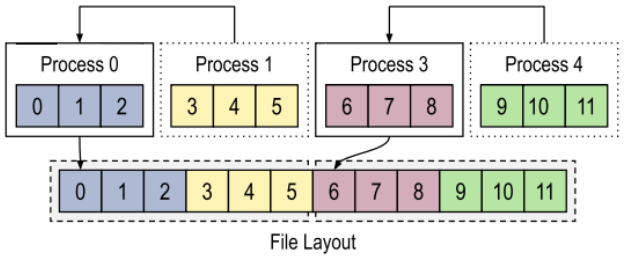
f) Synchronicity: Synchronicity deals with how an I/O operation affects the flow of the application. Synchronous or blocking I/O routines halt the execution of the application until the I/O operation is completed. Whereas asynchronous



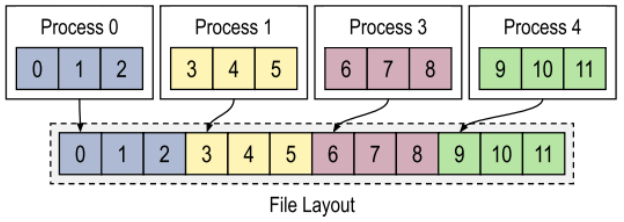
((a)) File-per-process



((b)) Shared file - Rank 0



((c)) Shared file - Multiple Rank



((d)) Shared file - Coordinated

Fig. 3: File approaches for parallel I/O [2]

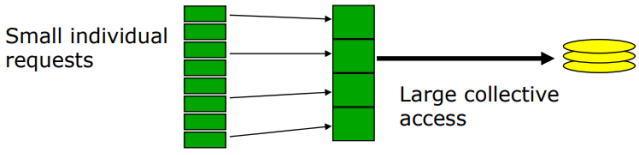


Fig. 4: Basic diagram depicting how collective I/O works

or non-blocking I/O overlaps I/O operations with computation or communication steps, allowing the application execution to progress. Asynchronous I/O provides significant performance speedup when we access large amounts of data. Both POSIX and MPI-IO have support and synchronous and asynchronous I/O.

B. Optimizing Access Patterns across the HPC I/O stack

This section will discuss some optimizations for access patterns, such as request aggregation, request reordering, and request scheduling.

a) Request Aggregation and Reordering: Request aggregation and reordering are two optimization techniques applied at the I/O middleware layer to transform the access patterns to be more suitable for the layers underneath. These optimization techniques include collective buffering and data sieving, part of ROMIO [34], a portable implementation of MPI-IO.

Research [38] has shown that small and random I/O requests harm the I/O performance of the system. Merging these small requests into larger, fewer requests spanning a large portion of the file is better for performance. ROMIO uses data sieving [37] to make fewer requests to the file system. When a process makes small independent, non-contiguous data requests, ROMIO does not access each data section separately. Let's assume that an MPI rank issues four non-contiguous data accesses. Instead of making four requests to the file server, using data sieving, ROMIO will read a single contiguous chunk of data in a temporary buffer using a single call, starting from the first requested byte till the last request byte. Once we have the data in the buffer, ROMIO will extract the data needed by the process in the process's buffer. Fig. 5 shows how data sieving works.

One potential problem with data sieving is that there can be memory issues if the user requests a large amount of data to be read. This problem can further worsen if there are large holes in the data. To deal with this, ROMIO provides multiple user-controlled parameters to define the maximum amount of contiguous data that can be read into the buffer. This includes parameters like `ind_rd_buffer_size` and `ind_wr_buffer_size`. If the data to be read is larger than the value defined by these parameters, data sieving is broken down into chunks, reading only as much data at a time defined by the parameter.

The significant advantage of data sieving is that it always reads the data in large chunks, avoiding the cost of small file accesses but at the cost of reading more data. A potential caveat can be that the holes between the data are too large and can outweigh the cost of reading extra data to avoid small accesses. However, that is not the case most of the time.

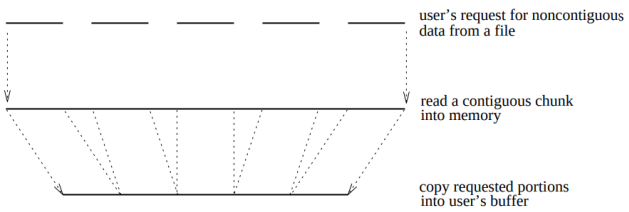


Fig. 5: Basic diagram depicting how data sieving works [37]

Two-phase I/O [39] is another access pattern optimization technique originally proposed for distributed systems. In two-

phase I/O, aggregator processes perform collective read and write. Aggregators are processes that issue the I/O requests to the file system. In collective reads, the aggregator is responsible for the part of the file requested by the collective reads and then distributes the chunks of the file to the processes participating in collective reads. Similarly, in collective write, the aggregator gathers data from a subset of processes participating in collective write into contiguous chunks in memory and writes the aggregated data to the file system. ROMIO provides two user-defined tuning parameters to control collective I/O. These parameters are `cb_nodes` and `cb_buffer_size`. `cb_nodes` refers to the number of aggregators and `cb_buffer_size` refers to the maximum buffer size on each aggregator. Fig. 6 shows how ROMIO performs collective reads.

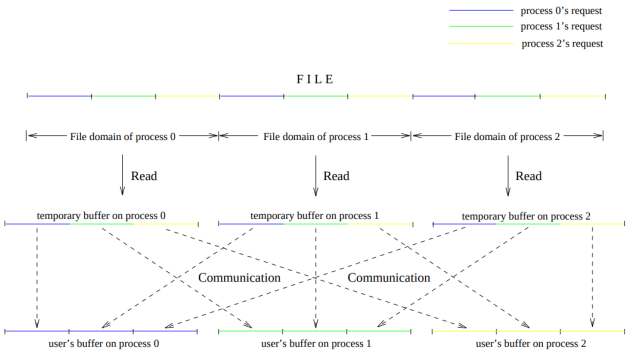


Fig. 6: ROMIO collective read [37]

b) Request Scheduling: On a large-scale supercomputer, multiple HPC applications can communicate concurrently with the Parallel File System (PFS) to perform I/O operations. This concurrency can lead to performance issues, and appropriate I/O scheduling techniques must be applied across various levels of the I/O stack to mitigate this issue.

There are a lot of I/O scheduling techniques applied at different levels of the stack (clients, I/O nodes, or data servers) that optimize concurrent access to the file system by organizing or reordering the requests. Let's assume two applications are issuing I/O requests to the file system. It is a possibility that when these requests arrive at the I/O forwarding layer or the file system, they are interleaved, hindering the performance. Even if multiple processes in one application are accessing a shared file contiguously, their requests to the file system might be perceived as non-contiguous because of interleaving. This phenomenon is called interference and is shown in Fig. 7.

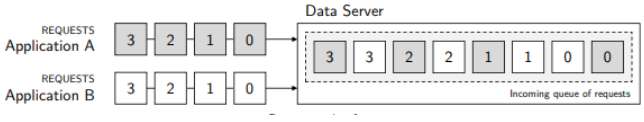


Fig. 7: Interference in I/O requests [27]

A variety of I/O schedulers are available, and each has a different level of complexity. Some use simple algorithms such as First Come, First Serve (FIFO) [40] while others like aIOLi [41], Object-Based Round Robin [42], and Network Request Scheduler [43] are more complex. We will look at some of these I/O schedulers now.

[44] presents a two-choice I/O scheduler that tracks the real-time performance of different storage servers to detect stragglers and uses that information to avoid placing I/O requests on stragglers. It builds upon the native two-choice algorithm and proposes collaborative probe and pre-assigns algorithms to improve performance. InterferenceRemoval [45] is another I/O scheduler that identifies those parts of the file involved in interfering accesses and replicates those portions to other data servers for better concurrency.

IOrchestrator [46] is another I/O scheduler that exploits the spatiality of the application for request scheduling. It introduces the concept of *reuse distance*, which is the time between two requests of the same program that have been sent to the data server. Assuming that these requests from the same program have a strong spatiality, IOrchestrator only dedicates resources to this one program. [47] uses a similar approach and introduces the concept of *time windows*. All the I/O requests forwarded to the storage server are considered time series and divided into fixed-size *time windows*. The algorithm sorts the I/O requests in a *time window* by application ID, and assuming that each application has strong spatiality, the execution will be optimized as the requests belonging to the same application are executed simultaneously.

[48] presents hierarchical I/O scheduling for collective I/O. Collective I/O has been used widely to address the issues of a large number of I/O requests. As systems grow, the performance of collective I/O can also be highly impacted by the increasing shuffle cost caused by highly concurrent data accesses. Hierarchical I/O scheduling (HIO) algorithm addresses this problem by using shuffle cost analysis to schedule applications' I/O requests for optimal performance.

In [49], a cross-application coordination technique to mitigate I/O interference is presented. Cross-application interference can greatly impact the performance of applications as there can be a lot of I/O requests with different sizes and requirements. To mitigate this issue, [49] presents four strategies: serializing, interrupting, interference, and dynamically adapting the best strategy based on the application's access pattern. [50] proposes an adaptive algorithm to vary the I/O workload based on the file system's performance.

Summary #2

There are different techniques for optimizing I/O access patterns. This includes techniques like collective I/O, data sieving, and two-phase I/O. Data sieving merges these small requests into larger, fewer requests spanning a large portion of the file, which is better for overall performance. Two phase I/O uses aggregators two perform collective reads and writes. There are also a variety of I/O schedulers that optimize concurrent access to the file system by organizing or reordering the requests.

IV. Evaluating Parallel I/O: A step by step guide

In this section, we first provide a taxonomy, as depicted in Fig. 8, for the iterative process of evaluating and optimizing parallel I/O. The I/O evaluation and optimization mainly consists of three major phases: (1) Measurements and Statistics Collection, (2) Modeling and Prediction, and (3) Simulation [67]. For the scope of this survey, we will only be looking at (1) Measurements and Statistics Collection and (2) Modeling and Prediction. We survey different research works and tools for each phase and present our results in the sections below.

A. Measurement and Statistics Collection

In this phase, we look at different methods for generating I/O workloads through real-world data or benchmarks. Then, we look at some tools that can be used to collect statistics and information about the I/O behavior of these workloads.

1) I/O Workloads: The first step in parallel I/O evaluation is generating and characterizing I/O workloads. Different workload generation techniques are available, some discussed in the sections below.

a) Application Code: The application code is the most accurate workload that can be used for characterization purposes. However, many times, the application code is not accessible or is too huge or long to be profiled for its I/O behavior. If the application code is unavailable, benchmarks can be used to generate synthetic workloads for characterization.

b) Benchmarks: Benchmarks simulate the workflow and access patterns of the original application, allowing performance evaluation and tuning of the original application without actually incurring the overhead of running it. In this section, we study different system and application benchmarks. System benchmarks use different interfaces and access patterns to benchmark parallel file and storage system performance, whereas application benchmarks simulate different access patterns to benchmark application performance.

We study different benchmarks [68], [69] available out there in this section and categorize them based on whether they are parallel file and storage system benchmarks or application benchmarks. For parallel file and storage system benchmarks, we study different characteristics of the benchmarks such as data, metadata, GPU support, etc. For application benchmarks, we study different characteristics such as the layers of the I/O stack these benchmarks cover and the I/O access patterns they generate. These findings are summarized in Table I and II.

c) Workload Replication and Simulation Frameworks: Proxy applications are small and simplified codes of large and complex production applications that encapsulate the important features of these applications without forcing the user to assimilate large and complex code bases. [70] presents proxy applications called MiniApps, which are based on large-scale application code at the Oak Ridge National Lab (ONRL). Encapsulating all the important features and details of these large-scale applications, these MiniApps run on production systems at ORNL. [71] is another work that replicates five different I/O workloads using MACSio [72] proxy application and Darshan [73].

[74] provides a novel framework, Durango, to generate scalable workloads from real applications using the performance modeling language Aspen and the HPC CODES framework. [75] presents IOWA, a novel I/O workload abstraction for generating diverse I/O workloads based on the inputs sources. [76] extracts the I/O pattern of the application and generates a suitable proxy application.

2) I/O Monitoring and Collection: Once we have the I/O workload available, the next step in I/O performance evaluation is I/O data monitoring and collection. This includes collecting I/O performance data of the application, such as time spent in I/O operations, read and write throughput, and total execution time. This data is then used to characterize the I/O performance of the application and further use it for analysis, modeling, and prediction.

There are a variety of I/O monitoring and collection tools available out there that help understand the I/O behavior of the application by looking at the interaction between the application workload and the underlying I/O stack. These tools can be divided into two categories based on how they collect the performance data. This performance data can be of two types: `traces` and `profiles`. `Profiles` store critical I/O information and statistics, such as file access patterns, number of floating point operations performed, number of function invocations, average execution time of a function, etc. `Traces`, on the other hand, provide a more fine-grained view of the I/O performance of the application, reporting data like timestamps of function calls along with their chronology. Because of their detailed metric collection, `traces` increase the overhead for I/O monitoring, degrading overall system performance.

This section will discuss the HPC I/O profiling and tracing tools available to the HPC community. More specifically, we will examine how these tools work and which layer of the HPC I/O stack they characterize. We will also look at some storage-system-level monitoring tools available to characterize the I/O behavior of the storage system. Lastly, we will also look at some recent monitoring systems that provide an all-encompassing and cohesive view of the application's I/O behavior.

a) I/O Profiling and Tracing Tools: Darshan [73], [77] is a commonly used application I/O characterization tool deployed on many large-scale HPC systems. It collects fine-grain trace data for POSIX and MPI-IO layers (parallel and low-

Benchmark	Data	Metadata	GPU Support	File System	Storage System
IOR [51]	✓	✗	✗	✓	✓
Mdtest [52]	✗	✓	✗	✓	✗
MD-Workbench [53]	✗	✓	✗	✓	✗
Fio [54]	✓	✗	✗	✗	✓
Elbencho [55]	✓	✗	✓	✓	✓
IOzone [56]	✓	✗	✗	✓	✗
Bonnie++	✓	✗	✗	✓	✓
OBDFilter-Survey [57]	✓	✗	✗	✓	✗
IOBench [58]	✓	✗	✗	✗	✓

TABLE I: I/O benchmarks for parallel file and storage systems

Benchmark	Low Level	Middleware	High Level	Synthetic	Application	I/O Mode	Operation	Spatiality	File Approach
H5bench [59]	✓	✓	✓	✓	✓	✓	✓	✓	✗
DLIO [60]	✗	✗	✓	✓	✗	✓	✗	✓	✓
HACC-IO [61]	✓	✓	✓	✓	✗	✓	✗	✗	✓
FLASH-IO [62]	✗	✗	✓	✓	✗	✗	✓	✗	✗
b-eff-io [63]	✓	✓	✗	✓	✗	✓	✓	✓	✓
MPI Tile I/O [64]	✗	✓	✗	✓	✗	✓	✓	✗	✗
NETCDF-Bench [65]	✗	✗	✓	✓	✗	✓	✓	✗	✗
Parabench [66]	✓	✓	✗	✓	✗	-	-	-	-

TABLE II: I/O benchmarks for application performance

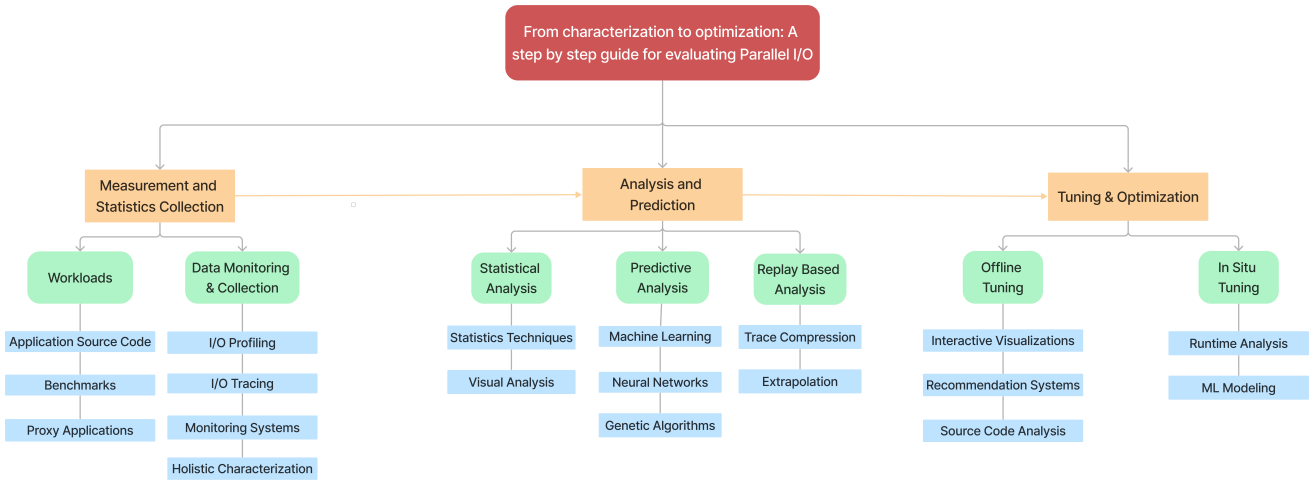


Fig. 8: Taxonomy of different phases in parallel I/O evaluation [67]

level I/O layer) and basic data for high-level I/O libraries such as HDF5 and PNetCDF. There have been recent advances in *Darshan* [77] to add support for tracing Parallel File Systems as well, such as Lustre.

Implemented as a set of user-space libraries, *Darshan* does not require any source code modification to instrument the application. Instead, it can be added to the application either statically or dynamically. Dynamic executables use the `LD_PRELOAD` environment variable to inject *Darshan* instrumentation in the application, whereas static executables link *Darshan* to the application during the linking phase. Instead of working as a typical tracing tool that logs every I/O operation, a bounded amount of data for each file is collected by *Darshan*, allowing compact storage of data. This data contains metrics such as I/O operations count, I/O access sizes, timers, and other statistical data relevant to the I/O performance of the application. Here are some of the metrics collected for each layer [73]:

- POSIX: read, write, open, seek, stat, mmap, fopen
- MPIIO: collective, independent, split, or nonblocking read and write
- MPIIO datatypes and hints
- Access type: unaligned, sequential, consecutive, and strided access

- Timestamps for open, close, first and last I/O
- Total bytes read and written
- Total time spent within POSIX and MPI-IO I/O operations
- Histograms of access, stride, datatype, and extent sizes

Fig. 9 shows how *Darshan* instruments an application. During the application execution, *Darshan* instruments different layers of the I/O stack for each process, generating data records characterizing the application's I/O workload and writing that data to the process's file while deferring all communication and I/O operations until the application terminates. When the application terminates, as part of the `darshan_shutdown` routine, it performs data aggregation and compression, which comprises aggregating and compressing the data of all the file-per-process into a single file. *Darshan* does this process in parallel and uses MPI-IO collective writes to make this step faster. This ability to defer communication and I/O until application shutdown, coupled with the bounded data collection, makes *Darshan* lightweight and portable, allowing seamless deployed on large-scale HPC systems such as the Argonne Leadership Computing Facility (ALCF), the National Energy Research Scientific Computing Center (NERSC), and the National Center for Supercomputing Applications (NCSA).

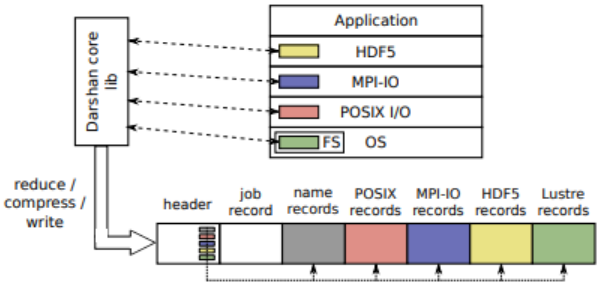


Fig. 9: High level overview of *Darshan's* architecture [77]

The overhead introduced by *Darshan* [73] is negligible. Fig. 10 shows the overhead of different versions of *Darshan* on the IOR [51] benchmark. The box plots indicate the distribution of I/O times in each case (with or without *Darshan*). It can be seen from the figure that the overhead when *Darshan* is enabled is very minimal and has very little impact on application performance.

Darshan eXtended Tracing (DXT) [78] extends *Darshan* to provide fine-grain traces of the application I/O to study behaviors of a wide range of workloads in greater depth. The DXT module is disabled by default and can be switched on using an environment variable. The overhead introduced by this module is also minimal (around 1%). DXT logs can be parsed, analyzed, and visualized offline, allowing this analysis on any system. This analysis can detect issues like workload imbalance, lock contention, and stripe misalignment.

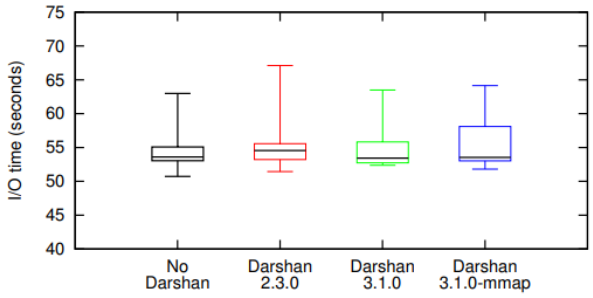


Fig. 10: Impact on application performance with Darshan enabled [77]

Recorder [79] is a multi-level I/O tracing framework that captures I/O information from multiple levels of the I/O stack, including HDF5, MPI-IO, and POSIX. It requires no source code changes, and the user controls which stack layers can be traced. It uses function interpositioning to prioritize itself over the standard functions using library preloading in Linux. Once *Recorder* is specified as a preloaded library, it intercepts HDF5, POSIX, and MPI-IO calls and reroutes these requests to the tracing routine. The tracing routine saves the function name, file name, entry timestamp, and the function parameters and then calls the original function and records its exit timestamp. All of this information is converted into a trace record and then compressed. Fig. 11 shows some of the features of the *Recorder* framework. [80] is a newer version of *Recorder* and adds new features such as trace format optimization, visualizations, and improved tracing for POSIX.

Feature	Recorder
Parallel file system compatibility	Yes
Ease of installation and use	1 (V. Easy)
Anonymization	3 (Medium)
Event Types	System calls and Library calls
Control of trace granularity	Moderate
Replayable trace generation	Planned
Trace replay fidelity	N/A
Reveals dependencies	No
Intrusive vs. Passive	1 (V. Passive)
Analysis tools	Planned
Trace data format	Binary and Human Readable
Accounts for time skew and drift	No
Elapsed time overhead	Under investigation

Fig. 11: Features of the *Recorder* framework [79]

IOPin [81] is a dynamic instrumentation tool developed to analyze the performance of parallel I/O. It uses Pin [82], which is a lightweight binary instrumentation tool to intercept the MPIIO and PVFS layers. Because of this, *IOPin* does not require recompilation of the source code or the I/O software stack. The basic architecture of *IOPin* is shown in Fig. 12. The figure shows how *IOPin* performs dynamic instrumentation when a collective MPIIO write call is issued. As can be seen

from the figure, two Pin profiling processes on the client and server sides generate trace information at the border of the MPI and PVFS layers. The client-side Pin process contains information such as the MPI call ID, rank, PVFS call ID, I/O type (read/write), and latency. The server-side Pin process stores information such as PVFS server ID, latency in the server, bytes to be read/written, the number of disk accesses, and disk throughput for the MPI I/O call at runtime

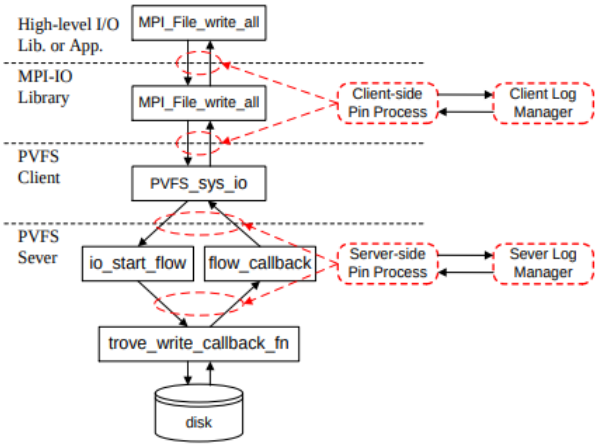


Fig. 12: *IOPin* framework [81]

TAU [83], [84] is a portable and flexible integrated toolkit of performance instrumentation and analysis of HPC applications. Although *TAU* is not I/O specific profiling tool, it does provide different instrumentation techniques to analyze the I/O performance of the application. One of these approaches is pre-processor-based instrumentation, in which the header files are redefined to instrument POSIX I/O calls. It also performs MPI instrumentation using the PMPI interface

Score-P [85] is a performance measurement infrastructure for profiling, event trace recording, and online analysis of High-Performance Computing (HPC) applications. *Score-P* provides different ways to instrument the application code, either through automatic compiler instrumentation or manual instrumentation. In parallel I/O, *Score-P* can instrument POSIX I/O and MPI I/O routines.

Mistral [86] is a commercially available HPC I/O monitoring tool from Altair. *Mistral*, due to its lightweight, can be easily deployed on production systems and is flexible enough to monitor I/O patterns on large-scale HPC and cloud systems. It uses files called contracts to store the rules for monitoring and throttling I/O. These contracts can be modified at runtime by privileged users. With its system and storage agnostic features, *Mistral* can monitor each job and file system's read, write, metadata use, and storage performance. Below is a list of data that Mistral collects.

- Collect I/O data per job, user group, mount point, and host
- Track read(), write(), and metadata operations
- Track time spent in I/O for each application to get file system performance
- Track statistics like the total, mean, and max time spent doing I/O per job
- Tracks CPU and memory utilization

Ellexus Breeze is another flexible and user-friendly offline analysis tool to help optimize and tune complex Linux applications. It can also capture MPIIO and POSIX calls and store the information related to them in binary trace files. *Ellexus Breeze* classifies I/O calls into three different categories:

- Bad I/O: Small read and write, Zero-byte I/O, Backward seek etc.
- Medium I/O: Large read and write, Forward seek, Delete operations, etc.
- Good I/O: Medium-sized read and write, etc.

IOPro [87] is a performance analysis and visualization framework for parallel I/O HPC applications. *IOPro* is different from the tools mentioned above in this way that it can instrument the complete parallel I/O stack. Instead of manually instrumenting each layer of the stack, *IOPro* takes the I/O software stack as input and generates an instrumented I/O stack to trace specific I/O functions and individual components across each layer. A high-level framework of *IOPro* is shown in Fig. 13

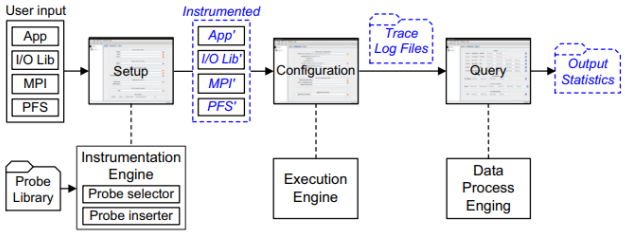


Fig. 13: Overview of the *IOPro* framework [87]

As can be seen from the framework in Fig. 13, *IOPro* has three main components: instrumentation engine, execution engine, and data processing engine. The instrumentation engine consists of a probe selector and a probe inserter. It inserts these probes into the I/O software stack automatically and generates an instrumented version of the software stack. The execution engine is responsible for building and compiling the instrumented software stack. The data process engine collects all trace log files from each layer of the instrumented I/O stack.

IOscope [90] is a flexible I/O tracer for characterizing the I/O patterns of the storage systems' workloads. It relies on the *extended Berkeley Packet Filter (eBPF)* technology to instrument the kernel and communicate the target kernel data towards a userspace process. The I/O pattern which the tool detects is the order in which the I/O requests access the file offsets during accessing on-disk data. *IOscope* consists of two tools: *IOscope classic* and *IOscope mmap*. The main idea is to trace and filter the workloads' I/O requests to construct the workloads' I/O patterns.

RIOT I/O [89] is another I/O tracer that uses function interpositioning to intercept the POSIX and MPI-IO libraries storing timing and other information about each function call.

Tool	Profiling	Tracing	Monitoring	High Level	Parallel and Low Level	Parallel File System	Storage
Darshan [73]	✓	✗	✗	✓	✓	✗	✗
IOPin [81]	✓	✗	✗	✗	✓	✓	✗
IOPro [87]	✓	✗	✗	✓	✓	✓	✗
IPM [88]	✓	✗	✗	✗	✓	✗	✗
TAU [83]	✓	✓	✗	✓	✓	✗	✗
Score-P [85]	✓	✓	✗	✗	✓	✗	✗
Darshan DXT [78]	✗	✓	✗	✗	✓	✗	✗
Recorder [79]	✗	✓	✗	✓	✓	✗	✗
RIOT I/O [89]	✗	✓	✗	✗	✓	✗	✗
IOscope [90]	✗	✓	✗	✗	✗	✗	✓
ScalaIOTrace [91]	✗	✓	✗	✓	✗	✗	✗
Mistral/Breeze [86]	✗	✗	✓	✗	✓	✗	✗
DKRZ Monitoring [92]	✗	✗	✓	✗	✗	✓	✓
LLview [93]	✗	✗	✓	✗	✗	✓	✓
GUIDE [94]	✗	✗	✓	✗	✗	✓	✓
LMT [95]	✗	✗	✓	✗	✗	✓	✓

TABLE III: I/O profiling, tracing, and monitoring tools and their characteristics

Similarly, *ScalaIOTrace* [91] is another tracing and analysis tool that collects trace data across multiple layers in the I/O stack. It also provides novel capabilities to analyze and collect statistical information from the collected traces automatically. *IPM* [88], developed at Lawrence Berkeley National Lab, is a portable profiling tool that profiles parallel programs' performance aspects and resource utilization. It uses an interposition layer to catch all the calls between the application and the file system layer.

The German Climate Computation Centre (DKRZ) developed a monitoring system [92] to develop the workload of the Mistral Supercomputer. This flexible and extensible monitoring system gathers statistics from 3340 client nodes, 24 login nodes, and 2 Lustre file systems. This system comprises open-source components such as Grafana and a self-developed data collector. It provides statistics about the login nodes, user jobs, and the workload manager queue. Initially, the monitoring system gives the users an overview of the system's current state, providing information such as the current load of login nodes and the number of used nodes on Slurm partitions. For each node in the system, the monitoring system collects metrics such as CPU usage, memory usage, luster file system usage, and energy consumption.

The Julich Supercomputing Centre (JSC) developed *LLview* [93] in 2004 to provide an interactive graphical tool to monitor jobs running on different workload managers such as SLURM. It has a job reporting module, which provides detailed information about all the individual jobs running on the system. The way *LLview* works is that it interacts with the different sources in the system to collect performance data and aggregate that data to present to the user in a web portal. Apart from providing a live view of the system, this web portal provides a list of the jobs running on the system in the form of a table, with each row showing the aggregated information for that job, such as the owner, project, start time, end time, etc. Upon clicking on a job, the user can see timeline graphs for the key performance metrics. All metrics in job reporting are gathered minutely, and the detailed reports can be made available offline as well in the form of a PDF.

GUIDE (Grand Unified Information Directory Environment) [94] is another framework that has been developed to collect, federate, and analyze log data from the Oak Ridge Leadership Computing Facility (OLCF). First, the data is extracted, and logs are collected. The framework collects data at each level of OLCF subsystems. For example, at the storage subsystem level, data is collected for the disk layer (the 2,016 OSTs, encompassing the 20,160 disks), the redundant RAID, controllers (72), the OSSes (288), and the Lustre PFS level. In the next step, the data is preprocessed using data cleansing techniques. In the federation step, the data is federated in a scalable repository to make post-processing and visualization of the data easier. Lastly, in the post-processing step, a suite of analytics and post-processing techniques are applied to get insights from the data.

Lustre Monitoring Tool (LMT) [95] is an open-source tool that monitors Lustre activity on HPC systems and presents a MySQL database that contains aggregated Lustre-specific counters on each object storage server (OSS) and metadata server (MDS). LMT provides information such as bytes read-/written, CPU load averages, and metadata operation rates. Similarly, *ggiostat* is another tool to collect similar data from IBM Spectrum Scale file systems. Developed at the Argonne Leadership Computing Facility (ALCF), it provides information such as bytes read/written and reads, writes, and metadata operations count retrieved from the server and client clusters.

b) Holistic Characterization Tools: Until now, we have looked at different I/O characterization tools and storage-server-level monitoring tools that instrument and monitor different layers in the I/O stack. These tools collect performance data for individual components in the I/O stack, but looking at this individually does not convey the whole picture of the application's I/O performance. In many cases, the data from different components and layers of the stack needs to be combined to get a more holistic view of the application's I/O behavior. This often requires involving an I/O expert in

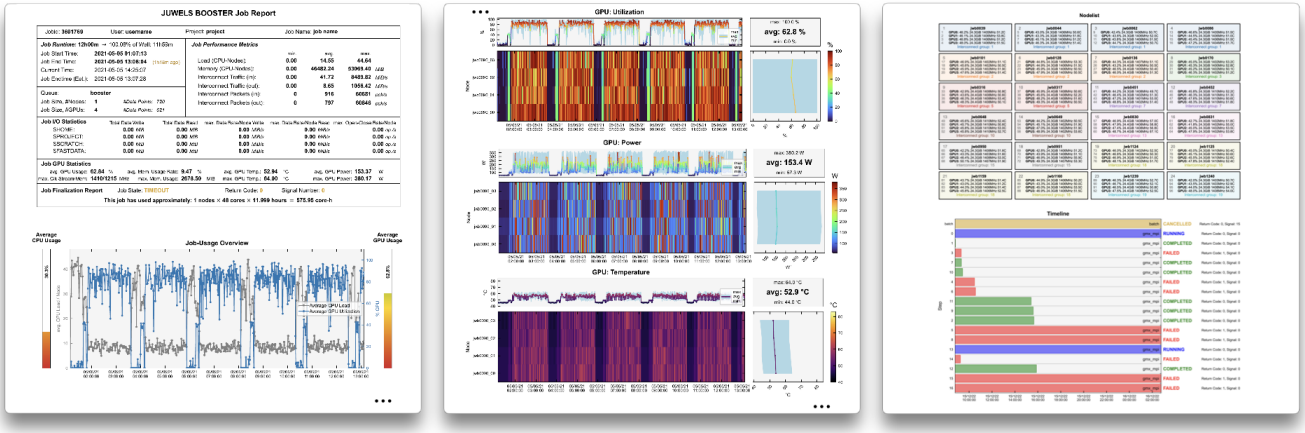


Fig. 14: A detailed PDF report of a node generated by *LLview* [93]

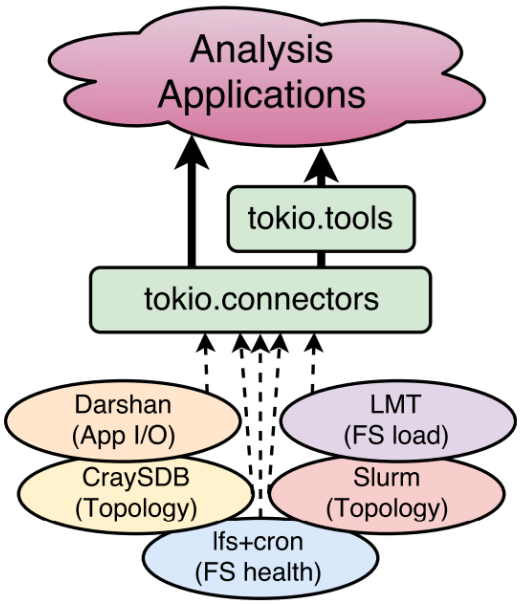


Fig. 15: TOKIO Architecture [96]

the loop to translate this disparate data, but not only is this practice labor-intensive, but it is also not sustainable as the system grows and scales. To tackle this issue, frameworks and monitoring systems have been developed that combine the insights from the performance data collected by various component-level monitoring tools and provide a holistic view of performance across the entire I/O stack. We will look at some of these frameworks in this section.

The Total Knowledge of I/O (TOKIO) [96] is a framework that connects different component-level monitoring and characterization tools, combines their insights, and presents a single, holistic view of the I/O performance across the entire I/O stack, which static analysis tools and user interfaces can further use. The distinguishing factor in the TOKIO framework is that it provides a modular implementation that connects to whatever monitoring and profiling tools are available on the HPC system. It mainly collects data from the tools that profile application behavior such as Darshan and tools that look at storage system and network traffic and health.

Connectors are the foundational layer of TOKIO. These modular and independent components provide an interface to connect with individual component-level tools, providing data from these tools. On top of connectors are *TOKIO tools*, which make this data collected by the connectors semantically closer to how analysis applications would like that data to be. These tools provide abstractions to hide the complexities of the underlying data source. High-level applications like command line tools, statistical and data analysis tools can easily connect with TOKIO interfaces to analyze the holistic data collected.

Similar to TOKIO, the Unified Monitoring and Metrics interface (UMAMI) [97] also provides a normalized, holistic view of the I/O behavior of the application. With the component-level data already being collected by different tools at the application and storage-system levels, this data is analyzed over one month, and the changes in the metrics are shown in UMAMI. [35] presents a multiplatform study in which Darshan logs representing a combined total of six years of I/O behavior of a million jobs across three leading HPC systems are mined and analyzed. It studies the evolution of the I/O behavior of the application over time, and based on the findings, the study provides techniques to improve the I/O performance of an application.

Beacon [98] is an end-to-end I/O monitoring and diagnosis tool developed for the TaihuLight supercomputer. Beacon monitors I/O for different nodes such as the compute, I/O forwarding, metadata, and storage nodes. At each node, Beacon deploys a daemon to monitor I/O and collect performance data for later aggregation. All this aggregated data is stored in JSON objects, which are then used to build profiling and analysis services such as detecting anomalies, per job I/O performance, etc.

Summary #3

Different holistic characterization tools are available, such as TOKIO [96], UMAMI [97], and Beacon [98]. Each tool provides different methodology to combine performance data, for example, TOKIO provides a modular implementation to connect with different profiling tools and aggregate their data, UMAMI provides a unified monitoring system for the data collected by different tools at the application and storage-system levels, and Beacon monitors different nodes in the I/O stack.

B. Analysis, Modeling, and Prediction

Once the profile and trace data are collected for the HPC I/O application using the different I/O profiling, tracing, and monitoring tools that we have discussed in the previous section, the next step in parallel I/O evaluation is to analyze this data to identify I/O issues, model I/O performance, and predict future I/O performance of the HPC system. In this section, we look at different tools and research work that has been done to analyze the I/O performance of an HPC system. This work can be divided into three main categories: Statistics and Analysis, Predictive Analysis, and Replay-Based Modeling.

1) Statistics and Analysis: The traditional way of analyzing I/O data is through applying statistics and analysis techniques to extract meaningful patterns from the I/O traces, I/O characterization profiles, and other logs. Different statistical techniques such as Arithmetic Mean, Standard Deviation, Linear Regression, Probabilities, etc. are applied to the data to classify, correlate, and extract meaningful patterns. Applying statistical analysis on HPC workloads and data requires in-depth system knowledge and extensive human effort.

[99] presents an extensive experimental study exploring the root causes of I/O interference in HPC systems. It first identifies the points of contention in an HPC storage system and evaluates how the application's access patterns, file system and network configuration, and storage devices affect interference. Based on the experiments on the Grid'5000 testbed and the OrangeFS file system, the study highlights seven root causes of I/O interference. [100] investigates various I/O performance issues by studying and analyzing performance data collected over a year at two leadership high-performance computing centers. The study looks at transient and long-term trends in I/O performance variability using different analysis techniques, such as correlative analysis and financial market technical analysis techniques, on time series I/O performance data to identify regions of interest. The insights provided in this paper can help broaden the scope of instrumentation and analysis tools.

IOMiner [101] is a large-scale analytics framework to analyze instrumentation data using a centralized storage schema that combines log data collected using different instrumentation tools and a sweep-line analysis function that identifies root causes of poor I/O performance of an application. The centralized storage schema is designed to take away any format difference that the different log data might have, making it query-friendly and easy to use for the sweep-line analysis function. One of the challenges that *IOMiner* addresses is how to mine useful information and insights from the performance data collected on supercomputers, which can run as many as millions of jobs quickly. To address this challenge, *IOMiner* uses a Python API for the Spark framework called PySpark, which speeds up data analysis on large-scale systems using parallel processing. In the analysis phase, *IOMiner* provides an analysis function that looks at five contributing factors to the application's poor I/O performance. These factors are:

- Small I/O requests
- Nonconsecutive I/O requests
- Utilization of collective I/O
- Number of OSTs used by each job
- Contention level

[35] presents a comprehensive study of the I/O behavior of applications on three large-scale supercomputers. It analyzes the Darshan logs on the three different supercomputers, spanning years and months. Through in-depth analysis of these logs, the paper looks at different aspects of the I/O performance of the applications on each supercomputer. For example, one of the paper's analyses is a platform-wide analysis in which the performance of I/O workloads on the three supercomputers is studied, and techniques are presented to identify underperforming apps. The analyses show that most of the apps on each supercomputer never exceed the platform peak I/O throughput, and low I/O throughput is the norm. Fig. 16 shows the maximum I/O throughput of each app across all its jobs on the three supercomputers under consideration, with the horizontal line showing the platform peak I/O throughput. The paper also presents other insights, such as discovering that a few jobs and apps mainly dominate each platform's I/O usage.

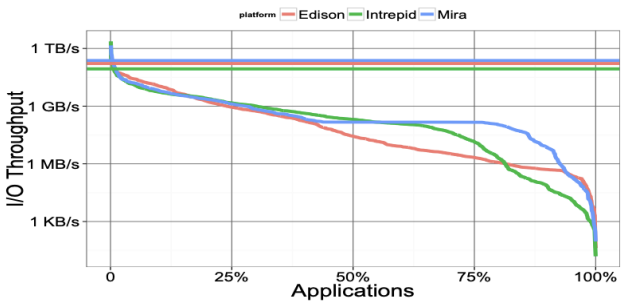


Fig. 16: Maximum I/O throughput of each application and the platform peak I/O throughput [35]

[102] presents a comprehensive study and analysis of the I/O performance of a production leadership-class storage system by collecting large amounts of trace data using the *Recorder* [79] tracing tool. After collecting the data, the study thoroughly analyzes this trace data to get insights into the performance and variability of the storage system. It provides some feedback on resolving the issues encountered in the analyses. The paper looks at two main scenarios to understand

Research/Framework	Analysis Technique	Analysis Goal
Yildiz et al [99]	Evaluates point of contentions in HPC storage system	Identify root causes of I/O interference
Lockwood et al [100]	Correlative and financial market technical analysis on year-long performance data	Identify various I/O performance issues
IOMiner [101]	Centralized storage schema that combines log data of different instrumentation tools	Detect root causes of poor I/O performance such as small I/O, non-collective I/O etc.
Luu et al [35]	Statistical analysis of Darshan logs spanning years	Study the I/O behavior of applications on three large-scale supercomputers
Wan et al [102]	Analyzes Recorder trace data	Study performance and variability of the storage system and provide feedback
pytokio [96]	Analysis routines and connectors to extract insights from monitoring tools	Holistic analysis of the I/O performance of parallel systems
GUIDE [94]	Analyzes log data from the Oak Ridge Leadership Computing Facility (OLCF)	Understand the performance of the storage system

TABLE IV: Statistical analysis tools/research and a brief description of what kind of analysis they perform

I/O performance do the storage system: 1) *Performance of I/O issued to a single OST*, and 2) *Impact of concurrent access to single OST*. It also looks at other scenarios, such as the effect of system caching on user-perceived performance and the effect of large-scale parallel I/O. Some of the key insights of the paper are:

- Concurrent access to a single OST might not lead to performance degradation for specific write sizes
- To maintain high throughput, system caching is critical
- Interference can be generated along the I/O path because of large scale parallel I/O
- Imbalanced I/O traffic distribution among OSTs can lead to performance degradation

PyTokio [96] is a Python implementation of the TOKIO framework discussed in previous sections. *PyTokio* makes holistic analysis of the I/O performance of parallel systems easier by providing connectors to interface with many commonly used monitoring tools. It also provides analysis routines to extract insights from the data collected from the different monitoring tools. One of the components of *PyTokio* is tokio tools which provides a high-level abstraction for accessing the combined data collected from multiple monitoring tools, allowing different analysis tools to build upon *PyTokio*. GUIDE (Grand Unified Information Directory Environment) [89] is another framework that we discussed earlier, which collects, federates, and analyzes log data from the Oak Ridge Leadership Computing Facility (OLCF). It also provides some analytics and visualizations of the storage system, like looking at the Lustre OST usage over time, I/O block sizes and space efficiency, and I/O request size distribution.

2) Predictive Analysis: Predictive Analysis, as the name suggests, is the kind of analysis that predicts future events based on the current information provided. Such analysis deploys techniques like data mining, predictive modeling, and machine learning on the trace/log data to predict future performance. By building a predictive model using the trace data available, accurate predictions on the future performance of the HPC system can be made. Such an analysis also includes auto-tuning techniques to predict the best parameters for each layer of the HPC I/O stack for optimized performance.

To predict future I/O operations, Omnisc'IO [103] presents a novel approach using a grammar-based model of the I/O behavior of the application. Not only does it predict when future I/O operations will occur, but it also tells where these operations will occur and how data will be accessed. The grammar-based model is built at runtime using an algorithm derived from Sequitur [112]. The way Omnisc'IO operates is that first, it gets the program's call stack, associating each call with an integer called *context symbols*. Then, it builds a grammar-based model from these *context symbols*. Once the grammar is built, it predicts future events by choosing a predictor from the grammar and associating each predictor with a weight. A brief architecture of Omnisc'IO is shown in Fig. 17.

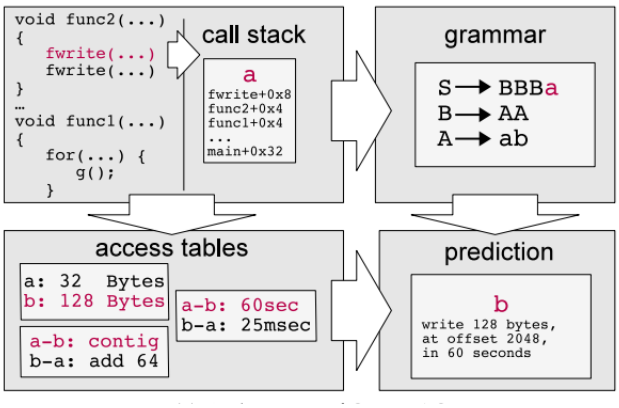


Fig. 17: Omnisc'IO Architecture [103]

[104] presents a novel framework for modeling and predicting the execution times of MPI programs. The framework captures the syntax tree of the parallel program and automatically instruments it, collecting important features. The instrumentor, developed in clang, inserts detective code around loops, branches, assignments, and MPI communications, generating an instrumented version of the code. Once the features are collected, the goal is to find a correlation between the collected features and the execution time. This multivariate nonlinear regression problem is solved using a random forest approach.

Research/Framework	Prediction Technique	Prediction Output
Omnisc'IO [103]	Grammer-based model	Next I/O operation
Sun et al [104]	Random forest approach	Execution Time
Schmidt et al [105]	Artificial Neural Networks	File Access Times
Nemirovsky et al [106]	Six Different Prediction Models[1]	Storage System Performance State
IONET [107]	Deep Neural Networks	Latency of every I/O of a full workload
Isaila et al [108]	Gaussian Process Model	I/O performance sensitivity to application and file characteristics
Behzad et al [109]	Genetic Algorithm	Best combination of parameters for different layers of the I/O stack
Bagbaba et al [110]	Random Forest	Collective I/O performance
Kim et al [111]	Six Different Prediction Models[2]	I/O performance such as read, write throughput

TABLE V: Predictive analysis tools along with their prediction technique and output

[105] is another research that uses machine learning with artificial neural networks to analyze and predict file access times of a Lustre file system.

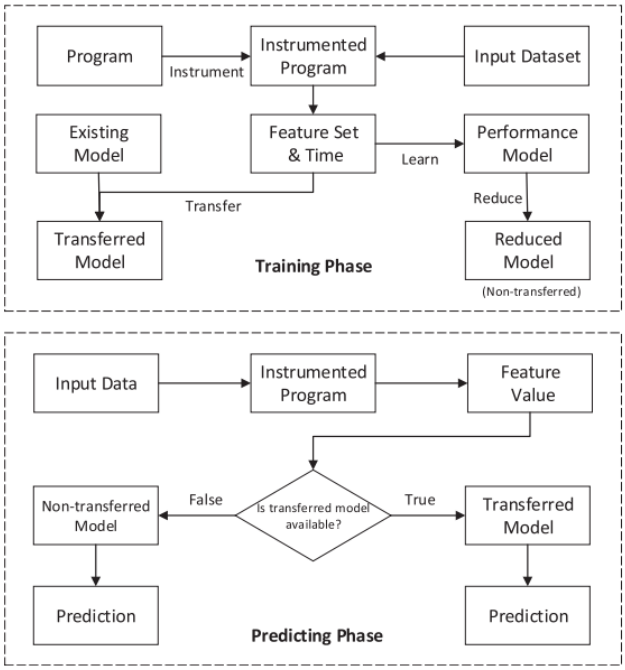


Fig. 18: Overview of automated performance modeling of HPC applications using machine learning [104]

[106] presents a lightweight parallel test harness to collect I/O data on HPC systems and monitor performance states on different storage subsystems. Then, it formulates a machine learning model to predict the transitions between those performance states during runtime. Treating this as a classification problem, this work uses a classifier to predict which I/O state a future I/O operation will likely encounter for an I/O path to an OSS. This work applies six commonly used machine learning classifiers to the dataset: classification and regression trees (CART), naive bayes (NB), gradient boosting (GBT), support vector machines (SVM), random forests (RF), and neural networks (NN). According to the results, SVM provides the best prediction accuracy.

IONET [107] is another ML-based I/O latency predictor that builds models of storage devices. It then uses this model to predict the latency of every I/O of a full workload running on a target storage cluster without running it on the cluster. It collects traces from various sources and industry partners and then designs an ML model to learn from them. The ML model uses machine learning techniques such as Deep Neural Networks, Random Forest, Logistic Regression, etc., to predict I/O latency from these traces. [108] presents a sensitivity-based modeling framework that predicts the I/O performance of the system and its variability as a function of application and file system characteristics using a Gaussian Process Model.

[109] optimizes the I/O performance of applications using an autotuning framework. This autotuning framework can efficiently optimize the different layers of the I/O stack, such as HDF5, MPI-IO, and Lustre, without requiring any source code changes. The framework has two main components: *H5evolve* and *H5Tuner*. *H5evolve* uses a genetic algorithm to search the I/O parameter space to find the best parameter combinations and then uses *H5Tuner* to inject the new I/O parameters in the application with minimal user involvement. [110] presents another machine learning-supported I/O autotuning framework to tune I/O parameters for different layers of the stack.

[111] predicts I/O performance using a regression-based approach by integrating system logs from various sources. First, the framework builds a joint database to store logs from different sources and then selects the important features from these logs using different scoring and feature selection algorithms. Once the features are selected, six different regression algorithms are developed, which use these features to predict I/O performance. Fig. 20 shows how the proposed framework works.

3) Replay-based Modeling: Replay-based modeling is another form of modeling and analysis that relies on historical I/O traces or characterization data. These traces, which contain detailed information about an HPC application's computation and I/O behavior, can be analyzed to replay the I/O behavior of the original application through I/O workload replication and benchmarks. These benchmarks and workloads can then be further used to predict the I/O performance of the original application. They can also test how the original application

[1]Classification and Regression Trees (CART), Naive Bayes (NB), Gradient Boosting (GBT), Support Vector Machines (SVM), Random Forests (RF), and Neural Networks (NN)

[2]Linear, Polynomial, K-nearest neighbors, Gradient boosting random forest (GBDT), Random Forests (RF), Multilayers perceptron (MLP), and Convolutional neural network (CNN)

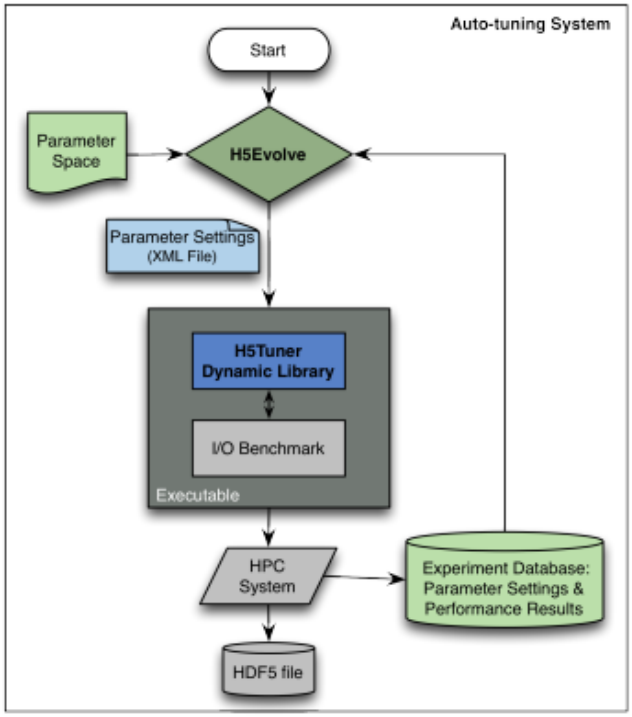


Fig. 19: Architecture of the autotuning framework presented in [109]

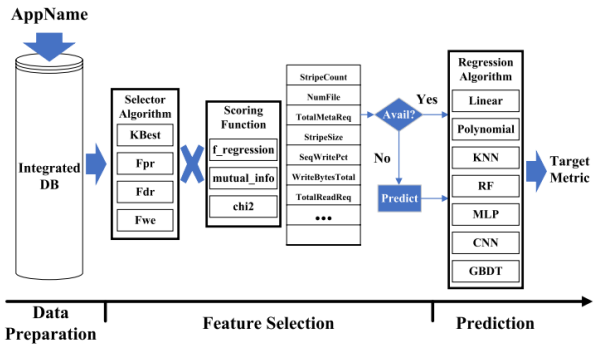


Fig. 20: Overall workflow of the proposed framework in [111]

will behave in different real-world deployments and scenarios. Apart from this, these replay-based models can also generate workloads for storage systems. There are a variety of tools and research works that study replay-based modeling. We will discuss some of this work in the following section.

[113] presents a replay-based modeling framework to generate portable benchmarks for I/O intensive parallel applications automatically. It introduces a trace merging and trace compression algorithm, which it uses to generate benchmarks of the original I/O application, mimicking its computation, communication, and I/O access patterns. These portable benchmarks can also be used to predict the I/O performance of the original application without the extra overhead, as the benchmarks are a scaled-down version of the original I/O application. Fig. 21 shows the overall workflow of this framework. The framework uses a suffix tree-based algorithm to compress the traces to reduce the redundant data introduced because of the loops. It extracts and compresses these loops to reduce the size of the trace by several orders of magnitude. After compressing the traces, they are merged and are used to generate the C code of the benchmark for the original I/O application.

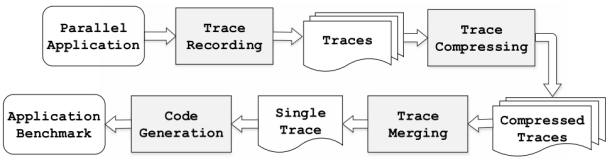


Fig. 21: Overall workflow of the proposed framework in [111]

[114] presents an extrapolation and replay-based tool called ScalaIOExtrap and ScalaIOReplay. It builds upon the ScalaIOTrace [91] tool and provides extended functionality for extrapolation and replay-based I/O analysis. It uses ScalaIOTrace to collect and analyze a small number of traces, determining the relationship between the different parameters and the number of ranks. It then uses ScalaIOExtrap to generate a single trace file for an arbitrary number of ranks. The experiments conducted with this tool show that the new I/O trace generated by ScalaIOExtrap retains the trace structure, I/O size, and the number of operations. It also preserves the event ordering and time accuracy in the new trace. This tool enables large-scale parallel I/O evaluation without actually executing the original application.

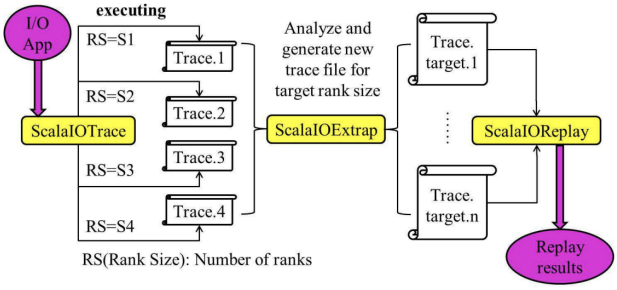


Fig. 22: Framework of ScalaIOTrace, ScalaIOExtrap, and ScalaIOReplay [114]

IOscope [90] is another tracing tool that solves the problem of high overhead incurred by collecting large traces across multiple layers of I/O stack by generating specific and ready-to-visualize traces of the applications' I/O patterns. SynchroTrace [115] is another trace-based simulation tool for multithreaded applications that generates dependency-aware architecture agnostic traces. It also provides a replay mechanism aware of these dependencies and can simulate synchronization actions to estimate the performance and power of chip multiprocessor systems (CMP).

Research/Framework	Analysis Technique	Analysis Goal
Hao et al [113]	Suffix tree-based compression algorithm	Portable benchmark of the original application to mimic computation, communication, and I/O
ScalaIOExtrap [114]	Trace Extrapolation to generate a single trace	C code of the benchmark for the original I/O application
IOscope [90]	Specific and ready-to-visualize traces of the applications' I/O patterns	Reduce overhead of collecting a large amount of traces
SynchroTrace [115]	Replay mechanism aware of system dependencies	Dependency-aware architecture agnostic traces for multithreaded applications

TABLE VI: Replay-based modeling tools/research and a brief description of what kind of modeling they do

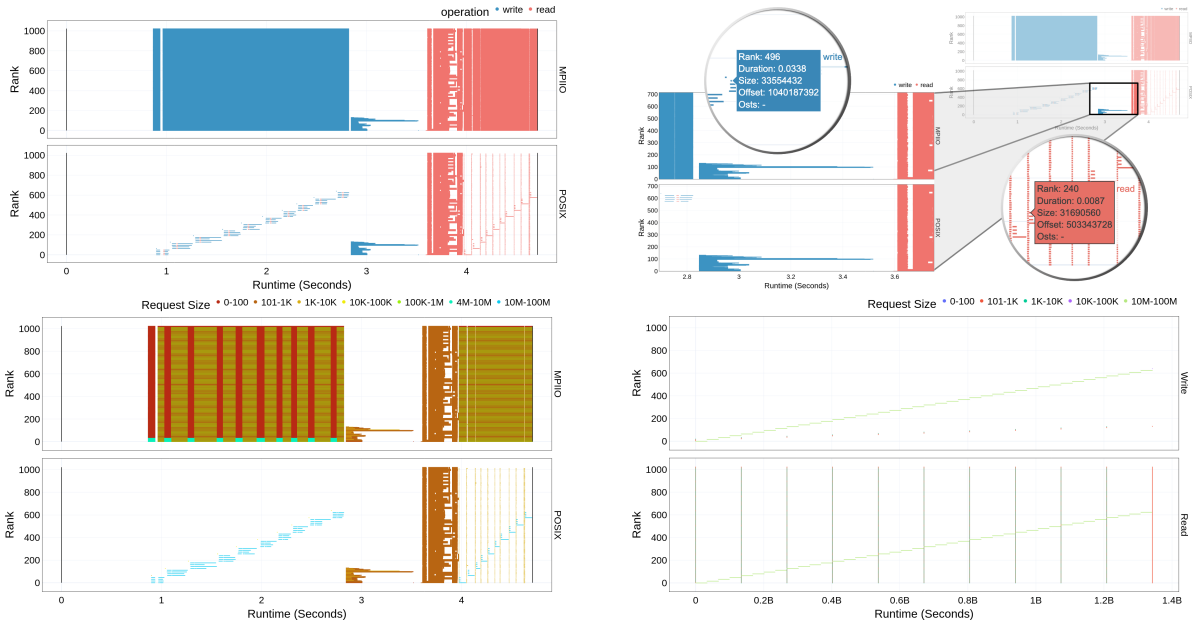


Fig. 23: reports focusing on different facets of the I/O behavior: (a) operations; (b) contextual information regarding the operations; (c) transfer sizes; and (d) spatial locality of the requests into the file. Combined, they provide a clear picture of the I/O access pattern and help identify the root causes of performance problems.

C. Optimizing Parallel I/O

Until now, we have looked at various tools and research studies that profile, trace, and evaluate I/O workloads on large-scale systems. When performance is slower than expected, end-users, developers, and system administrators rely on these I/O profiling and tracing information to pinpoint the root causes of inefficiencies and bottlenecks. However, despite the availability of numerous tools that collect I/O metrics on production systems, it is not obvious where the I/O bottlenecks are, what their root causes are, and what to do to solve them without actually involving an I/O expert in the loop. Hence, there is a gap between the metrics collected by these tools, the issues they represent, and the application of solutions and optimizations to eliminate these issues [116]. Streamlining analysis, investigation, and recommendations could close gaps without specialists intervening in each case. Researchers in the I/O community are actively studying this.

DXT Explorer [3] is one such interactive web-based log analysis tool that visualizes Darshan DXT logs and helps understand the I/O behavior of applications. The tool automatically analyzes and parses Darshan DXT log data to generate different interactive visualizations focusing on different facets of the I/O performance of the application, such as operations, transfer sizes, spatial locality, OST Usage, and I/O Phases. By adding an interactive component to Darshan trace analysis, this tool aids researchers, developers, and end-users to visually inspect their applications' I/O behavior, zoom in on areas of interest, and have a clear picture of where is the I/O problem [116].

Drishti [26] is another command line analysis framework that analyzes Darshan logs to pinpoint the various root causes of the I/O problems, providing actionable recommendations to eliminate these bottlenecks and improve I/O performance. Drishti relies on counters available in Darshan profiling logs to detect common bottlenecks and classify the insights into four categories based on the impact of the triggered event [26]. The tool also can pinpoint the exact line in the source code where changes need to be made to optimize performance. Based on this analysis, the tool generates a report, as shown in Fig. 24, highlighting the application issues and providing recommendations to solve those issues.

```
METADATA ──────────────────────────────────────────────────────────────────────

▸ Application is write operation intensive (60.83% writes vs. 39.17% reads)
▸ Application is write size intensive (64.15% write vs. 35.85% read)
▸ Application issues a high number (100.00%) of misaligned file requests
  ↪ Recommendations:
  ↪ Consider aligning the requests to the file system block boundaries

OPERATIONS ────────────────────────────────────────────────────────────────────

▸ Application issues a high number (275840) of small read requests (i.e., < 1MB) which
represents 100.00% of all read/write requests
  ↪ 275840 (100.00%) small read requests are to "8a_parallel_3Db_0000001.h5"
  ↪ Recommendations:
  ↪ Consider buffering read operations into larger more contiguous ones
  ↪ Since the appplication already uses MPI-IO, consider using collective I/O calls (e.g.
MPI_File_read_all() or MPI_File_read_at_all()) to aggregate requests into larger ones
▸ Application issues a high number (427386) of small write requests (i.e., < 1MB) which
represents 99.75% of all read/write requests
  ↪ 275840 (64.38%) small write requests are to "8a_parallel_3Db_0000001.h5"
  ↪ Recommendations:
  ↪ Consider buffering write operations into larger more contiguous ones
  ↪ Since the application already uses MPI-IO, consider using collective I/O calls (e.g.
MPI_File_write_all() or MPI_File_write_at_all()) to aggregate requests into larger ones
▸ Application mostly uses consecutive (97.67%) and sequential (2.16%) read requests
▸ Application mostly uses consecutive (97.85%) and sequential (1.17%) write requests
▸ Detected read imbalance when accessing 1 individual files.
  ↪ Load imbalance of 55.23% detected while accessing "8a_parallel_3Db_0000001.h5"
  ↪ Recommendations:
  ↪ Consider better balancing the data transfer between the application ranks
  ↪ Consider tuning the stripe size and count to better distribute the data
  ↪ If the application uses netCDF and HDF5 double-check the need to set NO_FILL values
  ↪ If rank 0 is the only one opening the file, consider using MPI-IO collectives
▸ Application uses MPI-IO and write data using 7680 (92.50%) collective operations
▸ Application could benefit from non-blocking (asynchronous) reads
  ↪ Recommendations:
  ↪ Since you use HDF5, consider using the ASYNC I/O VOL connector
(https://github.com/hpc-io/vol-async)
  ↪ Since you use MPI-IO, consider non-blocking/asynchronous I/O operations
▸ Application could benefit from non-blocking (asynchronous) writes
  ↪ Recommendations:
  ↪ Since you use HDF5, consider using the ASYNC I/O VOL connector
(https://github.com/hpc-io/vol-async)
  ↪ Since you use MPI-IO, consider non-blocking/asynchronous I/O operations
```

Fig. 24: Drishti report for the OpenPMD benchmark [26]

V. Conclusion

Understanding the I/O performance of large-scale workloads can be a complex task, especially now with the ever-increasing complexities of the underlying parallel computing hardware and the advent of novel workloads comprising artificial intelligence, machine learning, and big data. In such a scenario, one needs to have a guide that they can follow to evaluate and optimize the parallel I/O of their application. This work provides that guide by providing an in-depth study comprising more than 100 research papers to evaluate parallel I/O on large-scale computing systems. It first looks at the HPC I/O stack and different access patterns in detail studying how an I/O request is affected by different access patterns as it goes from the higher level in the parallel stack down to the storage system, depicting the complexities of the I/O stack as well. The study also provides a comprehensive list of different profiling and tracing tools that can characterize I/O performance and provides a unique synthesis of this information in the form of different tables throughout the paper. The study also looks at a variety of parallel I/O evaluation techniques which involve statistical analysis, predictive analysis, and replay-based modeling. It ends this discussion by talking about some of the latest research that is being done in bridging the gap between trace collection and optimizing parallel I/O without involving an I/O expert in the loop. 1his paper serves as a complete guide for anybody who wants the analyze and evaluate the

I/O performance of their application running on any large-scale HPC system.

REFERENCES

[1] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing i/o performance of hpc applications with autotuning," *ACM Trans. Parallel Comput.*, vol. 5, no. 4, mar 2019. [Online]. Available: https://doi.org/10.1145/3309205

[2] J. L. Bez, S. Byna, and S. Ibrahim, "I/o access patterns in hpc applications: A 360-degree survey," *ACM Comput. Surv.*, vol. 56, no. 2, sep 2023. [Online]. Available: https://doi.org/10.1145/3611007

[3] J. L. Bez, H. Tang, B. Xie, D. Williams-Young, R. Latham, R. Ross, S. Oral, and S. Byna, "I/o bottleneck detection and tuning: Connecting the dots using interactive log analysis," in *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, 2021, pp. 15–22.

[4] The HDF Group. (1997-NNNN) Hierarchical Data Format, version 5. Https://www.hdfgroup.org/HDF5/.

[5] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu, "Hello adios: the challenges and lessons of developing leadership class i/o frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3125

[6] C. Lee, M. Yang, and R. A. Aydt, "Netcdf-4 performance report," 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:15588867

[7] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netcdf: A high-performance scientific i/o interface," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 39. [Online]. Available: https://doi.org/10.1145/1048935.1050189

[8] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," 03 2011, pp. 36–47.

[9] R. Rew and G. Davis, "Netcdf: an interface for scientific data access," *IEEE Computer Graphics and Applications*, vol. 10, pp. 76–82, 1990. [Online]. Available: https://api.semanticscholar.org/CorpusID:11171299

[10] Hanisch, R. J., Farris, A., Greisen, E. W., Pence, W. D., Schlesinger, B. M., Teuben, P. J., Thompson, R. W., and Warnock, A., "Definition of the flexible image transport system (fits)*," *A&A*, vol. 376, no. 1, pp. 359–380, 2001. [Online]. Available: https://doi.org/10.1051/0004-6361:20010923

[11] D. C. Wells, E. W. Greisen, and R. H. Harten, "FITS - a Flexible Image Transport System," , vol. 44, p. 363, Jun. 1981.

[12] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, P. Canal, D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. G. Maline, M. Goto, J. Iwaszkiewicz, A. Kreshuk, D. M. Segura, R. Maunder, L. Moneta, A. Naumann, E. Offermann, V. Onuchin, S. Panacek, F. Rademakers, P. Russo, and M. Tadel, "Root — a c++ framework for petabyte data storage, statistical analysis and visualization," *Computer Physics Communications*, vol. 180, no. 12, pp. 2499–2512, 2009, 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465509002550

[13] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snirt, B. Traversat, and P. Wong, *Overview of the MPI-IO Parallel I/O Interface*. Boston, MA: Springer US, 1996, pp. 127–146. [Online]. Available: https://doi.org/10.1007/978-1-4613-1401-1_5

[14] "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)*, pp. 1–3951, 2018.

[15] N. H. Prickett, "What's so bad about posix i/o? - the next platform," Apr 2018. [Online]. Available: https://www.nextplatform.com/2017/09/11/whats-bad-posix-io/

[16] Oct 2020. [Online]. Available: https://www.iso.org/standard/74528.html

[17] G. Almási, R. Bellofatto, J. Brunheroto, C. Caşcaval, J. G. Castaños, L. Ceze, P. Crumley, C. C. Erway, J. Gagliano, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and K. Strauss, "An overview of the blue gene/l system software organization," in *Euro-Par 2003 Parallel Processing*, H. Kosch, L. Böszörményi, and H. Hellwagner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 543–555.

[18] J. Stender, B. Kolbeck, F. Hupfeld, E. Cesario, E. Focht, M. Hess, J. Malo, and J. Martí, "Striping without sacrifices: Maintaining posix semantics in a parallel file system." 01 2008.

[19] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *4th Annual Linux Showcase & Conference (ALS 2000)*. Atlanta, GA: USENIX Association, Oct. 2000. [Online]. Available: https://www.usenix.org/conference/als-2000/pvfs-parallel-file-system-linux-clusters

[20] "Lustre : A scalable , high-performance file system cluster," 2003. [Online]. Available: https://api.semanticscholar.org/CorpusID:16120094

[21] A. George, R. Mohr, J. Simmons, and S. Oral, "Understanding lustre internals. second edition." [Online]. Available: https://www.osti.gov/biblio/1824954

[22] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02. USA: USENIX Association, 2002, p. 19–es.

[23] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system." 01 2008, pp. 17–33.

[24] S. Nedev and V. Kamenov, "Hdd performance research," 01 2018.

[25] D. A. Patterson, *Computer Organization and Design*. Elsevier Science amp; Technology, 2013.

[26] J. L. Bez, H. Ather, and S. Byna, "Drishti: Guiding end-users in the i/o optimization journey," in *2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW)*, 2022, pp. 1–6.

[27] J. L. Bez, "Dynamic tuning and reconfiguration of the i/o forwarding layer in hpc platforms," Ph.D. dissertation, 05 2021.

[28] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun, "I/o acceleration with pattern detection," in *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 25–36. [Online]. Available: https://doi.org/10.1145/2493123.2462909

[29] C.-S. Kuo, A. Shah, A. Nomura, S. Matsuoka, and F. Wolf, "How file access patterns influence interference among cluster applications," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, 2014, pp. 185–193.

[30] S. Byna, M. Chaarawi, Q. Koziol, J. Mainzer, and F. Willmore, "Tuning hdf5 subfiling performance on parallel file systems." [Online]. Available: https://www.osti.gov/biblio/1398484

[31] F. Z. Boito, E. C. Inacio, J. L. Bez, P. O. A. Navaux, M. A. R. Dantas, and Y. Denneulin, "A checkpoint of research on parallel i/o for high-performance computing," vol. 51, no. 2, mar 2018. [Online]. Available: https://doi.org/10.1145/3152891

[32] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12.

[33] R. Thakur, W. Gropp, and E. Lusk, "An abstract-device interface for implementing portable parallel-i/o interfaces," in *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)*, 1996, pp. 180–187.

[34] R. Thakur, E. Lusk, and W. Gropp, "Users guide for romio: A high-performance, portable mpi-io implementation." [Online]. Available: https://www.osti.gov/biblio/564273

[35] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A multiplatform study of i/o behavior on petascale supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 33–44. [Online]. Available: https://doi.org/10.1145/2749246.2749269

[36] J. L. Bez, A. M. Karimi, A. K. Paul, B. Xie, S. Byna, P. Carns, S. Oral, F. Wang, and J. Hanley, "Access patterns and performance behaviors of multi-layer supercomputer i/o subsystems under production load," in *Proceedings of the 31st International Symposium on High-*

Performance Parallel and Distributed Computing, ser. HPDC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 43–55. [Online]. Available: https://doi.org/10.1145/3502181.3531461

[37] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation*, 1999, pp. 182–189.

[38] T. Wang, S. Byna, G. K. Lockwood, S. Snyder, P. Carns, S. Kim, and N. J. Wright, "A zoom-in analysis of i/o logs to detect root causes of i/o performance bottlenecks," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 102–111.

[39] J. Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel i/o via a two-phase run-time access strategy," *ACM SIGARCH Computer Architecture News*, vol. 21, pp. 31–38, 12 1993.

[40] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. B. Ross, and Y. Ishikawa, "Optimization techniques at the i/o forwarding layer," *2010 IEEE International Conference on Cluster Computing*, pp. 312–321, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:14886509

[41] A. lèbre, G. Huard, Y. Denneulin, and P. Sowa, "I/o scheduling service for multi-application clusters," 09 2006.

[42] Y. Qian, E. Barton, T. Wang, N. Puntambekar, and A. Dilger, "A novel network request scheduler for a large scale storage system," *Computer Science - Research and Development*, vol. 23, pp. 143–148, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:25880347

[43] F. Zanon Boito, R. Kassick, P. Navaux, and Y. Denneulin, "Automatic i/o scheduling algorithm selection for parallel file systems," *Concurrency and Computation Practice and Experience*, vol. 28, 08 2015.

[44] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'io: A grammar-based approach to spatial and temporal i/o patterns prediction," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 623–634.

[45] X. Zhang and S. Jiang, "Interferenceremoval: Removing interference of disk access for mpi programs through data replication," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 223–232. [Online]. Available: https://doi.org/10.1145/1810085.1810116

[46] X. Zhang, K. Davis, and S. Jiang, "Iorchestrator: Improving the performance of multi-node i/o systems via inter-server coordination," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[47] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-side i/o coordination for parallel file systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2063384.2063407

[48] J. Liu, Y. Chen, and Y. Zhuang, "Hierarchical i/o scheduling for collective i/o," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013, pp. 211–218.

[49] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "Calciom: Mitigating i/o interference in hpc systems through cross-application coordination," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 155–164.

[50] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "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*, 2010, pp. 1–12.

[51] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. IEEE Press, 2008.

[52] [Online]. Available: https://asc.llnl.gov/sites/asc/files/2020-09/mdtest-summary.pdf

[53] [Online]. Available: https://www.vi4io.org/tools/benchmarks/md-workbench

[54] [Online]. Available: https://fio.readthedocs.io/en/latest/fio_doc.html

[55] Staff, "Elbencho - a new storage benchmark for ai - high-performance computing news analysis," Feb 2021. [Online]. Available: https://insidehpc.com/2021/02/elbencho-a-new-storage-benchmark-for-ai/

[56] [Online]. Available: https://www.ibm.com/docs/en/linux-on-z?topic=tests-iozone

[57] [Online]. Available: https://wiki.lustre.org/OBDFilter_Survey

[58] B. Wolman and T. M. Olson, "Iobench: a system independent io benchmark," *SIGARCH Comput. Archit. News*, vol. 17, no. 5, p. 55–70, sep 1989. [Online]. Available: https://doi.org/10.1145/71302.71309

[59] T. Li, S. Byna, Q. Koziol, H. Tang, J. L. Bez, and Q. Kang, "h5bench: HDF5 I/O Kernel Suite for Exercising HPC I/O Patterns," in *Proceedings of Cray User Group Meeting, CUG 2021*.

[60] H. Devarajan, H. Zheng, A. Kougkas, X.-H. Sun, and V. Vishwanath, "Dlio: A data-centric benchmark for scientific deep learning applications," 05 2021, pp. 81–91.

[61] "Virtual institute for i/o." [Online]. Available: https://www.vi4io.org/tools/benchmarks/hacc-io

[62] [Online]. Available: https://www.ucolick.org/~zingale/flash_benchmark_io/#intro

[63] R. Rabenseifner, A. Koniges, J.-p. Prost, and R. Hedges, "The parallel effective i/o bandwidth benchmark: b-eff-io," 12 2001.

[64] S. Saini, D. Talcott, R. Thakur, P. Adamidis, R. Rabenseifner, and B. Ciotti, "Parallel i/o performance characterization of columbia and nec sx-8 superclusters," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 99, 03 2007.

[65] "Virtual institute for i/o." [Online]. Available: https://www.vi4io.org/tools/benchmarks/netcdf-bench

[66] O. Mordvinova, D. Runz, J. M. Kunkel, and T. Ludwig, "I/o performance evaluation with parabench — programmable i/o benchmark," *Procedia Computer Science*, vol. 1, no. 1, pp. 2125–2134, 2010, iCCS 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050910002395

[67] S. Neuwirth and A. K. Paul, "Parallel i/o evaluation techniques and emerging hpc workloads: A perspective," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 671–679.

[68] "Virtual institute for i/o." [Online]. Available: https://www.vi4io.org/tools/benchmarks/start

[69] "I/o benchmarks, applications, traces." [Online]. Available: https://web.cels.anl.gov/~thakur/pio-benchmarks.html

[70] O. B. Messer, E. D'Azevedo, J. Hill, W. Joubert, M. Berrill, and C. Zimmer, "Miniapps derived from production hpc applications using multiple programing models," *Int. J. High Perform. Comput. Appl.*, vol. 32, no. 4, p. 582–593, jul 2018. [Online]. Available: https://doi.org/10.1177/1094342016668241

[71] J. Dickson, S. A. Wright, S. Maheswaran, J. A. Herdman, D. Harris, M. C. Miller, and S. A. Jarvis, "Enabling portable i/o analysis of commercially sensitive hpc applications through workload replication," 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:37609466

[72] "Virtual institute for i/o." [Online]. Available: https://www.vi4io.org/tools/benchmarks/macsio

[73] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops (CLUSTER)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2009, pp. 1–10. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CLUSTR.2009.5289150

[74] C. D. Carothers, J. S. Meredith, M. P. Blanco, J. S. Vetter, M. Mubarak, J. LaPre, and S. Moore, "Durango: Scalable synthetic workload generation for extreme-scale application performance modeling and simulation," ser. SIGSIM-PADS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 97–108. [Online]. Available: https://doi.org/10.1145/3064911.3064923

[75] S. Snyder, P. Carns, R. Latham, M. Mubarak, R. Ross, C. Carothers, B. Behzad, H. V. T. Luu, S. Byna, and Prabhat, "Techniques for modeling large-scale hpc i/o workloads," ser. PMBS '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2832087.2832091

[76] J. Dickson, S. Wright, S. Maheswaran, A. Herdman, M. C. Miller, and S. Jarvis, "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)*, 2016, pp. 13–18.

[77] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular hpc i/o characterization with darshan," in *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, 2016, pp. 9–17.

[78] C. Xu, S. Snyder, O. Kulkarni, V. Venkatesan, P. Carns, S. Byna, R. Sisneros, and K. Chadalavada, "Dxt: Darshan extended tracing." [Online]. Available: https://www.osti.gov/biblio/1490709

[79] H. Luu, B. Behzad, R. Aydt, and M. Winslett, "A multi-level approach for understanding i/o activity in hpc applications," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, 2013, pp. 1–5.

[80] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient parallel i/o tracing and analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 1–8.

[81] S. J. Kim, S. W. Son, W.-k. Liao, M. Kandemir, R. Thakur, and A. Choudhary, "Iopin: Runtime profiling of parallel i/o in hpc systems," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 18–23.

[82] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi, "Simulation and analysis engine for scale-out workloads," ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926293

[83] S. S. Shende and A. D. Malony, "The tau parallel performance system," vol. 20, no. 2, p. 287–311, may 2006. [Online]. Available: https://doi.org/10.1177/1094342006064482

[84] S. S. Shende, A. D. Malony, W. Spear, and K. Schuchardt, "Characterizing i/o performance using the tau performance system," in *International Conference on Parallel Computing*, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:361834

[85] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[86] [Online]. Available: https://altair.com/mistral

[87] S. J. Kim, Y. Zhang, S. W. Son, M. Kandemir, W.-K. Liao, R. Thakur, and A. Choudhary, "Iopro: A parallel i/o profiling and visualization framework for high-performance storage systems," *J. Supercomput.*, vol. 71, no. 3, p. 840–870, mar 2015. [Online]. Available: https://doi.org/10.1007/s11227-014-1329-0

[88] N. J. Wright, W. Pfeiffer, and A. Snavely, "Characterizing parallel scaling of scientific applications using ipm," 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:16489254

[89] S. Wright, S. Hammond, J. Pennycook, R. Bird, A. Herdman, I. Miller, A. Vadgama, A. Bhalerao, and S. Jarvis, "Parallel file system analysis through application i/o tracing," *The Computer Journal*, vol. 56, pp. 141–155, 02 2013.

[90] A. Saif, L. Nussbaum, and Y.-Q. Song, "Ioscope: A flexible i/o tracer for workloads' i/o pattern characterization," in *High Performance Computing*, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds. Cham: Springer International Publishing, 2018, pp. 103–116.

[91] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Scalable i/o tracing and analysis," ser. PDSW '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 26–31. [Online]. Available: https://doi.org/10.1145/1713072.1713080

[92] J. M. Kunkel, E. Betke, M. Bryson, P. Carns, R. Francis, W. Frings, R. Laifer, and S. Mendez, "Tools for analyzing parallel i/o," in *High Performance Computing*, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds. Cham: Springer International Publishing, 2018, pp. 49–70.

[93] [Online]. Available: https://www.fz-juelich.de/en/ias/jsc/services/user-support/software-tools/llview?expand=translations%2Cfzjsettings%2Cnearest-institut

[94] S. S. Vazhkudai, R. Miller, D. Tiwari, C. Zimmer, F. Wang, S. Oral, R. Gunasekaran, and D. Steinert, "Guide: A scalable information directory service to collect, federate, and analyze logs for operational insights into a leadership hpc facility," in *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[95] [Online]. Available: https://wiki.lustre.org/Lustre_Monitoring_Tool

[96] G. K. Lockwood, N. J. Wright, S. Snyder, P. Carns, G. Brown, and K. Harms, "Tokio on clusterstor: Connecting standard tools to enable holistic i/o performance analysis." [Online]. Available: https://www.osti.gov/biblio/1632125

[97] G. K. Lockwood, W. Yoo, S. Byna, N. J. Wright, S. Snyder, K. Harms, Z. Nault, and P. Carns, "Umami: a recipe for generating meaningful metrics through holistic i/o performance analysis," in *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, ser. PDSW-DISCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 55–60. [Online]. Available: https://doi.org/10.1145/3149393.3149395

[98] B. Yang, X. Ji, X. Ma, X. Wang, T. Zhang, X. Zhu, N. El-Sayed, H. Lan, Y. Yang, J. Zhai, W. Liu, and W. Xue, "End-to-end I/O monitoring on a leading supercomputer," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 379–394. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/yang

[99] O. Yildiz, M. Dorier, S. Ibrahim, R. B. Ross, and G. Antoniu, "On the root causes of cross-application i/o interference in hpc storage systems," *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 750–759, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:17570971

[100] G. K. Lockwood, S. Snyder, T. Wang, S. Byna, P. Carns, and N. J. Wright, "A year in the life of a parallel file system," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 931–943.

[101] T. Wang, S. Snyder, G. Lockwood, P. Carns, N. Wright, and S. Byna, "Iominer: Large-scale analytics framework for gaining knowledge from i/o logs," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 466–476.

[102] L. Wan, M. Wolf, F. Wang, J. Y. Choi, G. Ostrouchov, and S. Klasky, "Comprehensive measurement and analysis of the user-perceived i/o performance in a production leadership-class storage system," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 1022–1031.

[103] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'io: A grammar-based approach to spatial and temporal i/o patterns prediction," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 623–634.

[104] J. Sun, G. Sun, S. Zhan, J. Zhang, and Y. Chen, "Automated performance modeling of hpc applications using machine learning," *IEEE Transactions on Computers*, vol. 69, no. 5, pp. 749–763, 2020.

[105] J. F. Schmid and J. M. Kunkel, "Predicting i/o performance in hpc using artificial neural networks," *Supercomputing Frontiers and Innovations*, vol. 3, no. 3, p. 19–33, Sep. 2016. [Online]. Available: https://superfri.org/index.php/superfri/article/view/105

[106] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal, and A. Cristal, "A machine learning approach for performance prediction and scheduling on heterogeneous cpus," in *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2017, pp. 121–128.

[107] J. Kunkel, M. Zimmer, and E. Betke, "Predicting performance of non-contiguous i/o with machine learning," in *High Performance Computing*, J. M. Kunkel and T. Ludwig, Eds. Cham: Springer International Publishing, 2015, pp. 257–273.

[108] F. Isaila, P. Balaprakash, S. M. Wild, D. Kimpe, R. Latham, R. Ross, and P. Hovland, "Collective i/o tuning using analytical and machine learning models," in *2015 IEEE International Conference on Cluster Computing*, 2015, pp. 128–137.

[109] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, "Taming parallel i/o complexity with auto-tuning," *Proceedings of the ACM/IEEE Supercomputing Conference*. [Online]. Available: https://www.osti.gov/biblio/1311633

[110] A. Bağbaba, "Improving collective i/o performance with machine learning supported auto-tuning," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 814–821.

[111] S. Kim, A. Sim, K. Wu, S. Byna, and Y. Son, "Design and implementation of i/o performance prediction scheme on hpc systems through large-scale log analysis," *Journal of Big Data*, vol. 10, 05 2023.

[112] C. Nevill-Manning and I. Witten, "Identifying hierarchical strcture in sequences: A linear-time algorithm." *J. Artif. Intell. Res. (JAIR)*, vol. 7, pp. 67–82, 09 1997.

[113] M. Hao, W. Zhang, Y. Zhang, M. Snir, and L. T. Yang, "Automatic generation of benchmarks for i/o-intensive parallel applications," *Journal of Parallel and Distributed Computing*, vol. 124, pp. 1–13, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S074373151830738X

[114] X. Luo, F. Mueller, P. Carns, J. Jenkins, R. Latham, R. Ross, and S. Snyder, "Hpc i/o trace extrapolation," in *Proceedings of the 4th Workshop on Extreme Scale Programming Tools*, ser. ESPT '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2832106.2832108

[115] K. Sangaiah, M. Lui, R. Jagtap, S. Diestelhorst, S. Nilakantan, A. More, B. Taskin, and M. Hempstead, "Synchrotrace: Synchronization-aware architecture-agnostic traces for lightweight multicore simulation of cmp and hpc workloads," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 1, mar 2018. [Online]. Available: https://doi.org/10.1145/3158642

[116] H. Ather, J. L. Bez, B. Norris, and S. Byna, "Illuminating the i/o optimization path of scientific applications," in *High Performance Computing*, A. Bhatele, J. Hammond, M. Baboulin, and C. Kruse, Eds. Cham: Springer Nature Switzerland, 2023, pp. 22–41.