Illuminating the I/O Optimization Path of Scientific Applications

Hammad Ather[1,2], Jean Luca Bez[1(✉)], Boyana Norris[2], and Suren Byna[1,3]

[1] Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
{hather,jlbez,sbyna}@lbl.gov
[2] University of Oregon, Eugene, OR 97403, USA
hather@uoregon.edu, norris@cs.uoregon.edu
[3] The Ohio State University, Columbus, OH 43210, USA
byna.1@osu.edu

Abstract. The existing parallel I/O stack is complex and difficult to tune due to the interdependencies among multiple factors that impact the performance of data movement between storage and compute systems. When performance is slower than expected, end-users, developers, and system administrators rely on I/O profiling and tracing information to pinpoint the root causes of inefficiencies. Despite having numerous tools that collect I/O metrics on production systems, it is not obvious where the I/O bottlenecks are (unless one is an I/O expert), their root causes, and what to do to solve them. Hence, there is a gap between the currently available metrics, the issues they represent, and the application of optimizations that would mitigate performance slowdowns. An I/O specialist often checks for common problems before diving into the specifics of each application and workload. Streamlining such analysis, investigation, and recommendations could close this gap without requiring a specialist to intervene in every case. In this paper, we propose a novel interactive, user-oriented visualization, and analysis framework, called *Drishti*. This framework helps users to pinpoint various root causes of I/O performance problems and to provide a set of actionable recommendations for improving performance based on the observed characteristics of an application. We evaluate the applicability and correctness of *Drishti* using four use cases from distinct science domains and demonstrate its value to end-users, developers, and system administrators when seeking to improve an application's I/O performance.

Keywords: I/O · insights · visualization · I/O optimization

1 Introduction

The parallel I/O stack deployed on large-scale computing systems has a plethora of tuning parameters and optimization techniques that can improve application I/O performance [4,8]. Despite that, applications still face poor performance

A. Bhatele et al. (Eds.): ISC High Performance 2023, LNCS 13948, pp. 22–41, 2023.
https://doi.org/10.1007/978-3-031-32041-5_2

when accessing data. Harnessing I/O performance is a complex problem due to the multiple factors that can affect it and the inter-dependencies among the layers of the software and hardware stack.

When applications suffer from I/O performance slowdowns, pinpointing the root causes of inefficiencies requires detailed metrics and an understanding of the stack. There is a variety of I/O performance profiling and characterization tools, which are very helpful in diagnosing the I/O bottlenecks in an application. However, none of these tools provide a set of actionable items to guide users in solving the bottlenecks in the application. For instance, I/O profiling tools collect metrics to provide a coarse-grain view of the application's behavior when accessing data. Darshan [11] and Recorder [40] profilers can also trace I/O requests and provide a fine-grain view of the transformations the requests undergo as they traverse the parallel I/O software stack. Nonetheless, despite the availability of such fine-grained traces, there is a gap between the trace collection, analysis, and tuning steps.

A solution to close this gap requires analyzing the collected metrics and traces, automatically diagnosing the root causes of poor performance, and then providing user recommendations. Towards analyzing the collected metrics, Darshan [11,13] provides various utilities to summarize statistics. However, their interpretation is left to the user to identify root causes and find solutions. There have been many studies to understand the root causes of performance problems, including IOMiner [42] and Zoom-in I/O analysis [41]. However, these studies and tools are either application-specific or target general statistics of I/O logs. Existing technologies lack the provision of feedback and recommendation to improve the I/O performance or to increase utilization of I/O system capabilities [10].

To address these three components, i.e., analysis of profiles, diagnosis of root causes, and recommendation of actions, we envision a solution that meets the following criteria based on a visualization approach.

① Provide interactive visualization based on I/O trace files, allowing users to focus on a subset of MPI processes or zoom in to specific regions of the execution;
② Display contextual information about I/O calls (e.g., operation type, rank, size, duration, start and end times);
③ Understand how the application issues its I/O requests over time under different facets: operation, request sizes, and spatiality of accesses;
④ Observe transformations as the requests traverse the I/O software stack;
⑤ Detect and characterize the distinct I/O phases of an application;
⑥ Understand how the ranks access the file system in I/O operations;
⑦ Provide an extensible community-driven framework so new visualizations and analysis can be easily integrated;
⑧ Identify and highlight common root causes of I/O performance problems;
⑨ Provide a set of actionable items based on the detected I/O bottlenecks.

In this paper, we propose a novel interactive web-based analysis framework named "*Drishti*" to visualize I/O traces, highlight bottlenecks, and help understand the I/O behavior of scientific applications. Using *Drishti*, which is based

on the nine requirements mentioned above, we aim to fill the gap between the trace collection, analysis, and tuning phases. However, designing this framework has several challenges in analyzing I/O metrics for extracting I/O behavior and illustrating it for users to explore, automatically detecting the I/O performance bottlenecks, and presenting actionable items to users. To tackle these challenges, we devised a solution that contains an interactive I/O trace analysis component for end-users to visually inspect their applications' I/O behavior, focusing on areas of interest and getting a clear picture of common root causes of I/O performance bottlenecks. Based on the automatic detection of I/O performance bottlenecks, our framework maps numerous common and well-known bottlenecks and their solution recommendations that can be implemented by users. This paper builds upon initial feedback on some of the components of *Drishti* [5,8]. Nonetheless, it describes a broader picture, encompassing novel work and features such as the I/O bottleneck detection from extended tracing logs, I/O phase analysis, and file system usage. Our proof-of-concept uses components and issues that are often investigated by I/O experts when end-users complain about I/O performance. Though some might be obvious to an I/O expert, end-users often face a barrier. *Drishti* seeks to streamline the process and empower the community to solve common I/O performance bottlenecks.

We designed *Drishti* to be scalable through different approaches, and not limited by the number of processes/cores. Our goal in using interactive visualizations is to overcome the limitations of static plots where information and bottlenecks cannot be displayed completely due to pixel limitations. In combination with the I/O analysis and recommendations of triggered issues, one can pinpoint the causes of those issues as they are highlighted in the visualization and zoom into areas of interest. A limiting issue might arise when an application issues millions of small requests using all the ranks for a longer period of time. To tackle this challenge, we provide options to generate multiple time-sliced plots. We also laid out the foundations for a community-based effort so that additional metrics could be added to *Drishti*, combined into more complex bottleneck detection, and integrated into the interactive visualization component. We demonstrate our framework with multiple case studies and visualize performance bottlenecks and their solutions.

The remainder of the paper is organized as follows. In Sect. 2, we discuss related work. Our approach to interactively explore I/O behaviors is detailed in Sect. 3, covering design choices, techniques to detect I/O phases and bottlenecks, and available features. We demonstrate its applicability with case studies in Sect. 4. We conclude the paper in Sect. 5 and discuss future efforts.

2 Related Work

We discuss a few tools that target I/O performance analysis, visualization, or bottleneck detection in HPC applications and highlight the novelty of our work.

NVIDIA Nsight [27] and TAU [29] are used for the performance analysis and visualization of HPC applications. They provide insights into issues from the

perspective of CPU and GPU usage, parallelism and vectorization, and GPU synchronization, which can help optimize the overall performance of applications. In addition to these tools, Recorder [40] and IOMiner [42] are also used extensively to analyze the I/O performance of HPC applications. Some tools (e.g., TAU, Score-P [15], HPC Toolkit [1]) provide profiling/traces of I/O operations, with preliminary reports on observed performance. Furthermore, most of the performance visualization tools draw a line in displaying traces and metrics. The interpretation and translation of actions impose a steep learning curve for non-I/O experts. We push the state of the art by providing interactive visualizations with root cause analysis, bottleneck identification, and feedback to end-users.

The Total Knowledge of I/O (TOKIO) [21] framework provides a view of the performance of the I/O workloads deployed on HPC systems by connecting data and insights from various component-level monitoring tools available on HPC systems. It seeks to present a single coherent view of analysis tools and user interfaces. The Unified Monitoring and Metrics Interface (UMAMI) [22] introduces a holistic view of the I/O system of large-scale machines by integrating data from file systems, application-level profilers, and system components into a single interface. Both focus on the global view of the I/O system at a large scale rather than on the particular I/O issues of each application.

Tools like AI4IO [35] rely on artificial intelligence to predict and mitigate I/O contention in HPC systems. AI4IO includes two tools, PRIONN and CanarIO, which work together to predict I/O contention and take steps to prevent it. INAM [17] is a technique for profiling and analyzing communication across HPC middleware and applications, which can help identify bottlenecks and provide significant speedup by resolving those bottlenecks. H5tuner [4] is an auto-tuning solution for optimizing HPC applications for I/O usage.

All the aforementioned tools target I/O performance visualization and detecting I/O bottlenecks in HPC systems. Despite several efforts, none of the existing tools fill the translation gap which exists between determining the I/O bottlenecks and coming up with suggestions and recommendations to get rid of those bottlenecks. Our work fills this translation gap by providing interactive visualizations showing the I/O performance of the application and providing a set of actionable items or recommendations based on the detected I/O bottlenecks. Furthermore, auto-tuning approaches complement our work, as they could harness the provided insights and bottleneck detection to reduce their search space.

3 Visualization, Diagnosis, and Recommendations

We have designed and developed *Drishti* based on feedback gathered at two supercomputer facilities, the I/O-research community, and targeted end-users. In the following subsections, we discuss the design choices to support interactive visualizations, I/O behavior analysis, I/O phase detection, and how we efficiently map bottlenecks to a set of actionable items in a user-friendly way. In Fig. 1, we show the various components of *Drishti*.

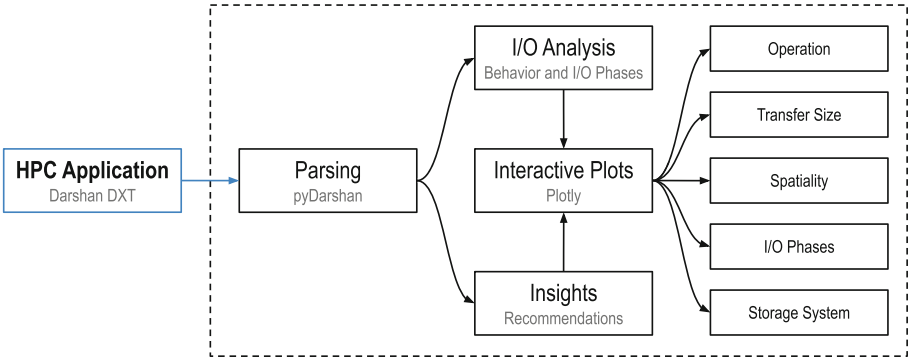


Fig. 1. *Drishti* generates meaningful interactive visualizations and a set of recommendations based on the detected I/O bottlenecks using Darshan DXT I/O traces.

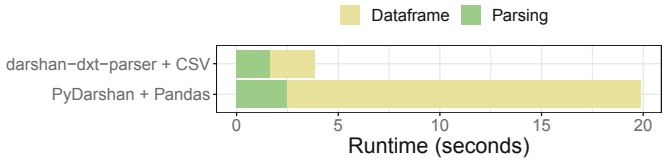


Fig. 2. Comparison of methods to extract and combine the I/O behavior metrics from Darshan DXT traces required to pinpoint I/O issues and generate visualizations.

3.1 Extracting I/O Behavior from Metrics

Darshan [11] is a tool deployed on several large-scale computing systems to collect I/O profiling metrics. Darshan collects aggregated statistics with minimal overhead providing a coarse-grain view of application I/O behavior. An extended tracing module of Darshan, DXT [44], can capture fine-grain POSIX and MPI-IO traces. Due to its widespread use, we use Darshan logs as input.

To characterize an application's I/O behavior, we require an efficient way to analyze possibly large traces collected by Darshan DXT logs that are in binary format. Darshan provides a command line solution named `darshan-dxt-parser` as part of the *darshan-util* library to parse DXT traces out of the binary Darshan log files. The parsed data is stored in a pre-defined textual format which could then be transformed into a CSV file to be analyzed. Figure 2 summarizes the time taken to obtain the required data in such approach. The trace file used in Fig. 2 is an OpenPMD use case with 1024 ranks over 64 nodes. The original trace file was of size 1.9 MB and after our transformations, the size was 23.6 MB.

Because of multiple conversions, these additional steps add to the user-perceived time. As an alternative, we have also explored PyDarshan [13], a novel Python package that provides interfaces to binary Darshan log files. With PyDarshan, we get direct access to the parsed DXT trace data in the form of a *pandas* [39] DataFrames. Figure 2 compares the performance of both approaches. However, PyDarshan also has shortcomings when the analysis requires an overall

view of application behavior. It currently returns a dictionary of DataFrames containing all trace operations issued by each rank. This data structure is not optimal if the visualization requires an overall view of the application behavior, which is the case in *Drishti*. Therefore, an additional step has to be taken to iterate through the dictionary of DataFrames and merge them into a single DataFrame for both analysis and interactive visualization. For the trace in Fig. 2, this additional merging operation represents 87.3% of the time. If PyDarshan can provide direct access to all ranks in form of a single DataFrame, costly data transformations such as the one shown in Fig. 2 can be avoided.

3.2 Exploring I/O Behavior Interactively

I/O traces can be large for applications with longer runtimes or even for relatively short applications with a large number of small I/O requests, making analysis and visualization of the behavior difficult. Static plots have space constraints and pixel resolution issues. Thus they often hide the root causes of I/O bottlenecks in plain sight. For instance, when thousands of ranks issue I/O operations concurrently, but some of them suffer interference at the server level, those lines are not visible in a static plot at a regular scale.

Towards developing a modular and extensible framework (criterion ⑦ in §1), we consider two solutions. Our initial prototype to move from a static to interactive and dynamic visualization relied on plots generated in R using *ggplot2*. R is a programming language for statistical computing used in diverse fields such as data mining, bioinformatics, and data analysis. *ggplot2* is an open-source data visualization package for R to declaratively create graphics, based on The Grammar of Graphics [43] schema. A plot generated using this library could be converted into an interactive visualization by using the open-source *ggplotly* graphing library powered by *Plotly*. *Plotly* is a data visualization library capable of generating dynamic and interactive web-based charts.

However, integrating with the data extraction discussed in Sect. 3.1 would require the framework to combine features in different languages, compromising modularity, maintainability, and increasing software dependencies, possibly constraining its wide adoption in large-scale facilities. We have opted to rely on PyDarshan to extract the data. Using the open-source Plotly.py Python wrappers would simplify the code without compromising features or usability. Furthermore, it would easily allow I/O data experts to convert their custom visualizations into interactive ones and integrate them into *Drishti*. It also brought the advantage of reducing the total user-perceived time by 84.5% (from avg. of 69.45s to 10.74s), allowing such time to be better spent on detailed analysis of I/O behavior. Figure 3 summarizes this difference. This is the same OpenPMD use case, with write/read operations, of Fig. 2. Note that Fig. 3 only accounts for I/O phase analysis and plotting. The results on both Fig. 2 and Fig. 3 should be combined for the total runtime, i.e., they depict complementary information. Section 3.3 covers the I/O behavior analysis to pinpoint the root causes of bottlenecks.

As scientific applications often handle multiple files during their execution, which overlap in time (e.g., file-per-process or multiple processes to multiple

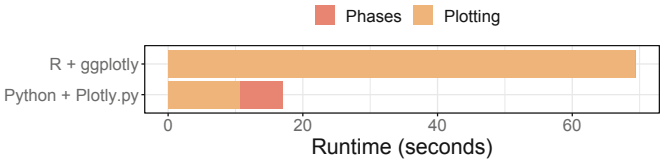
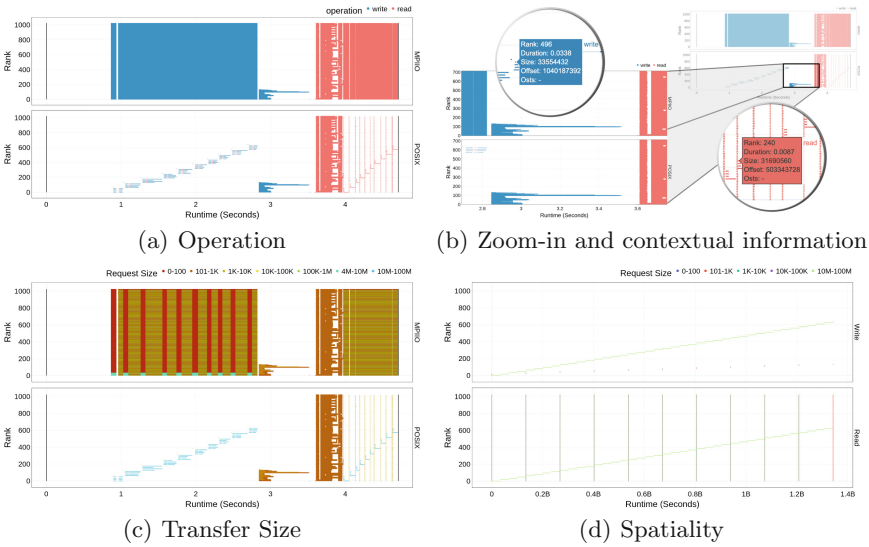


Fig. 3. Comparison of solutions to generate the interactive plots and detect I/O phases from Darshan DXT traces. Both approaches use the Plotly.js library under the hood to generate web-based interactive plots.



(a) Operation (b) Zoom-in and contextual information

(c) Transfer Size (d) Spatiality

Fig. 4. *Drishti* 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.

files approaches), *Drishti* should provide a separate visualization for each. Furthermore, those visualizations should shine some light on the application's I/O behavior from multiple perspectives, i.e., criterion ③: operation, data transfer, and spatiality. Figure 4 shows the reports of particle and mesh-based data from a scientific simulation. Plotly also meets our criteria by allowing a user to dynamically narrow down the plot to cover a time interval of interest or zoom into a subset of ranks to understand the I/O behavior (criterion ①).

Because of the complexity of the parallel I/O stack, the requests issued by an application are transformed before reaching the file system. Those transformations originate from different mappings between the data model used by an application and its file representation or by the application of I/O optimization techniques such as collective buffering and data-sieving [37] or request scheduling [6,9,12]. To shed light on these transformations, *Drishti* depicts every plot using

two synchronized facets: the first representing the MPI-IO level, and the second, its translation to POSIX level (criterion ④). For each request, by hovering over the depicted interval, it is possible to inspect additional details such as the operation type, execution time, rank, and transfer size, meeting criterion ②. Interactive examples are available in our companion repository jeanbez.gitlab.io/isc23.

When visualizing an application's I/O behavior, we are one step closer to understanding the root causes of any performance bottlenecks, demystifying data transformations, and guiding users to apply the most suitable set of optimization techniques to improve performance. We highlight that there is a lack of a straightforward translation of the I/O bottlenecks into potential tuning options. In this paper, we seek to close this gap by providing a framework to bring those issues to light, automatically detecting bottlenecks and meaningfully conveying actionable solutions to users.

3.3 Automatic Detection of I/O Bottlenecks

Several tools seek to analyze the performance of HPC applications, as discussed in Sect. 2. However, few of them focus on I/O and neither provide support for auto-detection of I/O bottlenecks in the application nor provide suggestions on how to fix those. We summarize common root causes of I/O performance bottlenecks in Table 1. Some issues require additional data or a combination of metrics collected from profilers, tracers, and system logs. For instance, Darshan's profiler only keeps track of the timestamp of the first and last operations to a given file. In contrast, its Extended Tracing module (DXT) tracks what happens in between, such as different behaviors or I/O phases.

Table 1. Root causes of I/O performance bottlenecks

Root Causes	Darshan	DXT	System	Drishti
Too many I/O phases [41]	✓	✓	✗	✓
Stragglers in each I/O phase [36]	✓	✓	✗	✓
Bandwidth limited by a single OST I/O bandwidth [23,41]	✗	✗	✓	✗
Limited by the small data size [41]	✓	✓	✗	✓
Rank 0 heavy-workload [46]	✓	✓	✗	✓
Unbalanced I/O workload among MPI ranks [41]	✓	✓	✗	✓
Large number of small I/O requests [41]	✓	✓	✗	✓
Unbalanced I/O workload on OSTs [41,46]	✓	✓	✓	✓
Bad file system weather [22,41]	✗	✗	✓	✗
Redundant/overlapping I/O accesses [12,30]	✓	✓	✗	✓
I/O resource contention at OSTs [32,45]	✗	✗	✓	✗
Heavy metadata load [23]	✓	✗	✗	✓

Drishti seeks to provide interactive web-based visualizations of the tracing data collected by Darshan, but it also provides a framework to detect I/O bottlenecks in the data (from both profiling and tracing metrics) and highlights criterion ⑧ those on the interactive visualizations along with providing a set of recommendations (criterion ⑨) to solve the issue. *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 and the certainty of the provided recommendation: HIGH (high probability of harming I/O performance), WARN (detected issues could negatively impact the I/O performance, but metrics might not be sufficient to detect application design, configuration, or execution choices), OK (the recommended best practices have been followed), and INFO (details relevant information regarding application configuration that could guide tuning solutions). The *insights* module is fully integrated with the parsing and visualization modules of the framework, so the identified issues and actionable items can enrich the reports.

Table 2. Triggers evaluated by *Drishti* for each Darshan log.

Level	Interface	Detected Behavior
HIGH	STDIO	High STDIO usage* (> 10% of total transfer size uses STDIO)
OK	POSIX	High number* of sequential read operations (≥ 80%)
OK	POSIX	High number* of sequential write operations (≥ 80%)
INFO	POSIX	Write operation count intensive* (> 10% more writes than reads)
INFO	POSIX	Read operation count intensive* (> 10% more reads than writes)
INFO	POSIX	Write size intensive* (> 10% more bytes written then read)
INFO	POSIX	Read size intensive* (> 10% more bytes read then written)
WARN	POSIX	Redundant reads
WARN	POSIX	Redundant writes
HIGH	POSIX	High number* of small† reads (> 10% of total reads)
HIGH	POSIX	High number* of small† writes (> 10% of total writes)
HIGH	POSIX	High number* of misaligned memory requests (> 10%)
HIGH	POSIX	High number* of misaligned file requests (> 10%)
HIGH	POSIX	High number* of random read requests (> 20%)
HIGH	POSIX	High number* of random write requests (> 20%)
HIGH	POSIX	High number* of small† reads to shared-files (> 10% of reads)
HIGH	POSIX	High number* of small† writes to shared-files (> 10% of writes)
HIGH	POSIX	High metadata time* (one or more ranks spend > 30 seconds)
HIGH	POSIX	Rank o heavy workload
HIGH	POSIX	Data transfer imbalance between ranks (> 15% difference)
HIGH	POSIX	Stragglers detected among the MPI ranks
HIGH	POSIX	Time imbalance* between ranks (> 15% difference)
WARN	MPI-IO	No MPI-IO calls detected from Darshan logs
HIGH	MPI-IO	Detected MPI-IO but no collective read operation
HIGH	MPI-IO	Detected MPI-IO but no collective write operation
WARN	MPI-IO	Detected MPI-IO but no non-blocking read operations
WARN	MPI-IO	Detected MPI-IO but no non-blocking write operations
OK	MPI-IO	Detected MPI-IO and collective read operations
OK	MPI-IO	Detected MPI-IO and collective write operations
HIGH	MPI-IO	Detected MPI-IO and inter-node aggregators
WARN	MPI-IO	Detected MPI-IO and intra-node aggregators
OK	MPI-IO	Detected MPI-IO and one aggregator per node

* Trigger has a threshold that could be further tunned. Default value in parameters.
† Small requests are consider to be < 1 MB.

The interactive visualizations are enhanced using multi-layered plots, with each layer activated according to the detected bottleneck keeping the original behavior in the background (criterion ⑧). The idea behind highlighting the bottlenecks on the interactive visualizations, apart from classifying the bottlenecks in different categories, is to allow the user to actually visualize where the bottlenecks are in the application. This will allow them to get more detailed information about the bottlenecks and give them more clarity about the application behavior which the textual information alone cannot provide. Furthermore, we complement the interactive visualization with a report based on 32 checks covering common I/O performance pitfalls and good practices, as summarized in Table 2. We provide the multi-layered plot functionality for the *operation* plot for now. Each layer of the plot shows a different variant of the base graph, for example, one layer can show one of the bottlenecks in the graph, and the other can show the base chart.

3.4 Exploring I/O Phases and Bottlenecks

HPC applications tend to present a fairly consistent I/O behavior over time, with a few access patterns repeated multiple times over their execution [20]. Request scheduling [6,9], auto-tuning [2–4] and reinforcement-learning [7,19] techniques to improve I/O performance also rely on this principle to use or find out the best configuration parameters for each workload, allowing the application to fully benefit from it in future iterations or executions. We can define an I/O phase as a continuous amount of time where an application is accessing its data following in a specific way or following one or a combination of access patterns. Nonetheless, factors outside the application's scope could cause an I/O phase to take longer, such as network interference, storage system congestion, or contention, significantly modifying its behavior. Seeking to detect I/O phases, *Drishti* adds an interactive visualization based on DXT trace data. This visualization gives a detailed picture of I/O phases and I/O patterns in the data and is very helpful in extracting information related to bottlenecks such as stragglers, meeting our criterion ⑤.

Finding the I/O phases from trace data is not trivial due to the sheer amount of data, often representing millions of operations in the order of milliseconds. We use PyRanges [31] to find similar and overlapping behavior between an application's MPI ranks and a threshold value to merge I/O phases closer to each other. PyRanges is a genomics library used for handling genomics intervals. It uses a 2D table to represent the data where each row is an interval (in our case, an operation), and columns represent chromosomes (i.e., interface and operation), the start and end of an interval (i.e., operation).

While identifying the I/O phases, we keep track of the duration between each I/O phase that represents computation or communication. Once we have the duration of all the intervals between the I/O phases, we calculate the mean and standard deviation of such intervals. A threshold is calculated by summing up the mean and the standard deviation, and it is used to merge I/O phases close to each other into a single I/O phase. We do that because due to the small

Algorithm 1. Merging I/O phases by a threshold

```
end ← df[end][0]
prev_end ← 0
while i < len(df) do
    if df[start][i] − end <= threshold then
        prev_end ← df[end][i]
    end if
    if df[start][i] − end > threshold OR i = len(df) − 1 then
        chunk_end ← df[prev_index : i].copy()
        end ← df[end][i]
        prev_end ← i
    end if
end while
```

time scale of the operations, we might end up with a lot of tiny I/O phases that, from the application's perspective, represent a single phase. As of now, the merging threshold cannot be changed dynamically from the visualization interface. Algorithm 1 describes the merging process. We take an I/O phase and check if the difference between the end of the last I/O phase and the start of this I/O phase is less than equal to the threshold value. We keep on merging the I/O phases till they satisfy this condition.

Figure 5 shows a sample I/O phases visualization, that is fully interactive supporting zoom-in/zoom-out. The phases are generated for MPIIO and POSIX separately. Hovering over an I/O phase displays the fastest and slowest rank in that phase and their durations.

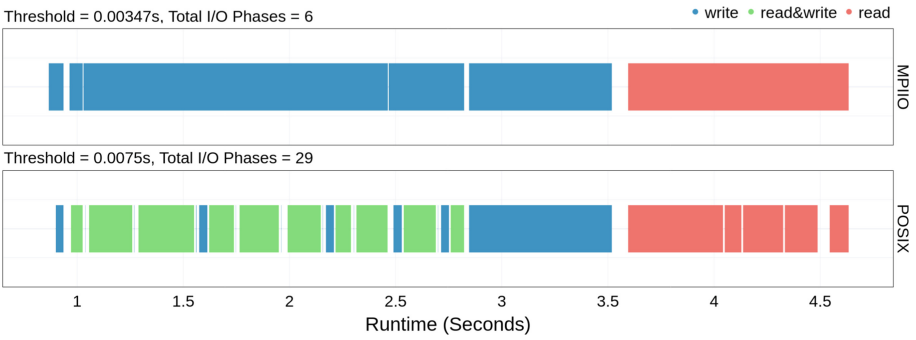


Fig. 5. Interactive I/O phases visualization in MPI-I/O and POSIX layers.

Understanding an application's I/O phases allow the detection of additional performance bottlenecks, as detailed by Table 1. To showcase how *Drishti* could be used in this context, we briefly cover synchronous and asynchronous requests, stragglers, and multiple I/O phases.

Blocking I/O Accesses. From a scientific application's perspective, I/O operations can be synchronous or asynchronous. Asynchronous I/O is becoming increasingly popular to hide the cost associated with I/O operation and improve overall performance by overlapping computation or communication with I/O operations [26,33]. Multiple interfaces (e.g., POSIX and MPI-IO) and high-level I/O libraries (e.g., HDF5) provide both blocking and non-blocking I/O calls. For HDF5, the Asynchronous I/O VOL Connector [34] can explore this feature.

If we consider only the profiling data available in Darshan, it only captures the number of non-blocking calls at the MPI-IO level and not when they happened. To provide a detailed and precise suggestion of when asynchronous could benefit the application, we rely on the I/O phases and the intervals between those to provide such recommendations. We demonstrate a use case with a block-structured adaptive mesh refinement application in Sect. 4.

I/O Stragglers. I/O stragglers in each phase define the critical path impairing performance. *Drishti* has an exclusive visualization to highlight the I/O phases and their stragglers (Fig. 6). We handle each interface separately due to the transformations that happen as requests go down the stack. The dotted lines represent the boundaries of an I/O phase. In each, the fastest and the slowest rank is shown. Combined with contextual information, it is possible to detect slow ranks across the entire execution or storage servers consistently delivering slow performance.

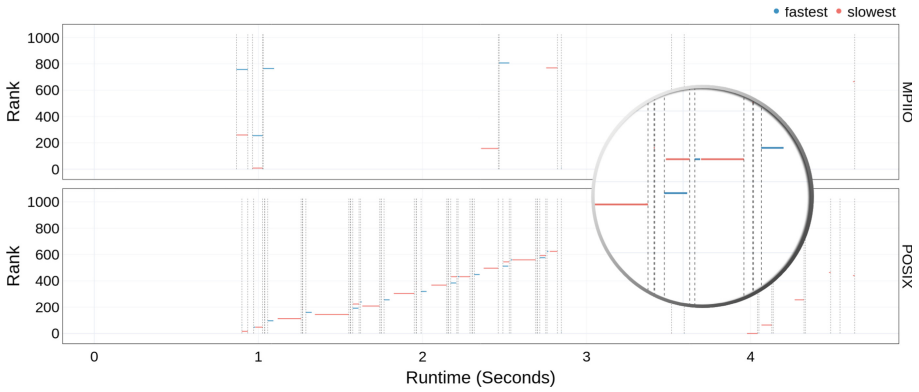


Fig. 6. Stragglers are identified in red for each I/O phase. (Color figure online)

3.5 Towards Exploring File System Usage

Additional logs are required to correctly detect bottlenecks related to unoptimized file system accesses, as detailed in Table 1. Nonetheless, Darshan DXT

captures some information that could provide an initial overview of the storage servers' use if the underlying file system is Lustre and that integration is enabled. *Drishti* provides an exclusive visualization to explore the OST usage of the I/O requests, as depicted in Fig. 7. Furthermore, because of file stripping, a request at the MPI-I/O level might be broken down and require access to multiple storage devices to be completed, which explains why the information at both levels is not the same. *Drishti* can also depict the data transfer sizes (writes and reads) for each OST at both the MPI-I/O and POSIX levels.

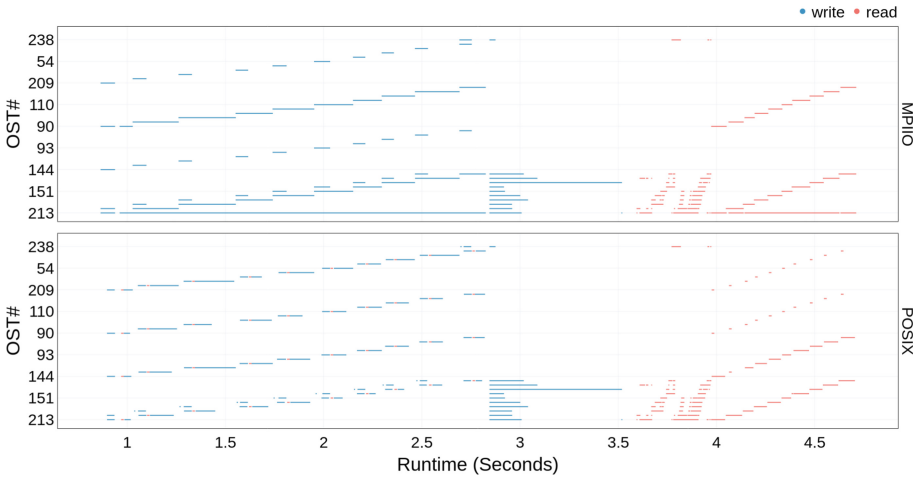


Fig. 7. Lustre data storage (OST) access over time.

4 Results

We selected the OpenPMD (Sect. 4.2) and AMReX (Sect. 4.3) use cases from distinct science domains to demonstrate *Drishti*'s value to end-users, developers, and system administrators. Both came from interactions with their core developers about concerns related to poor I/O performance. Existing solutions previously tried did not uncover all the root causes of performance inefficiencies. Experiments were conducted in two production supercomputing systems: Cori at the National Energy Research Scientific Computing Center (NERSC) and Summit at the Oak Ridge Leadership Computing Facility (OLCF). We have also probed the I/O research community and targeted end-users to gather feedback on the tool's features and helpfulness. For instance, highlighting bottlenecks uncovered by the heuristic analysis, indexing, and filtering the generated visualizations based on the file are some enhancements added from community-driven feedback. User-interface presentation of the contextual data was also shaped based on such evaluation.

4.1 I/O Systems in NERSC and OLCF

Cori is a Cray XC40 supercomputer at NERSC. It has 2,388 Intel Xeon Haswell, and 9,688 Intel Xeon Phi Knight's Landing (KNL) compute nodes. All compute nodes are connected to a $\approx$ 30 PB Lustre parallel file system with a peak I/O bandwidth of 744 GB/s. Cori's PFS is comprised of 244 Object Storage Servers.

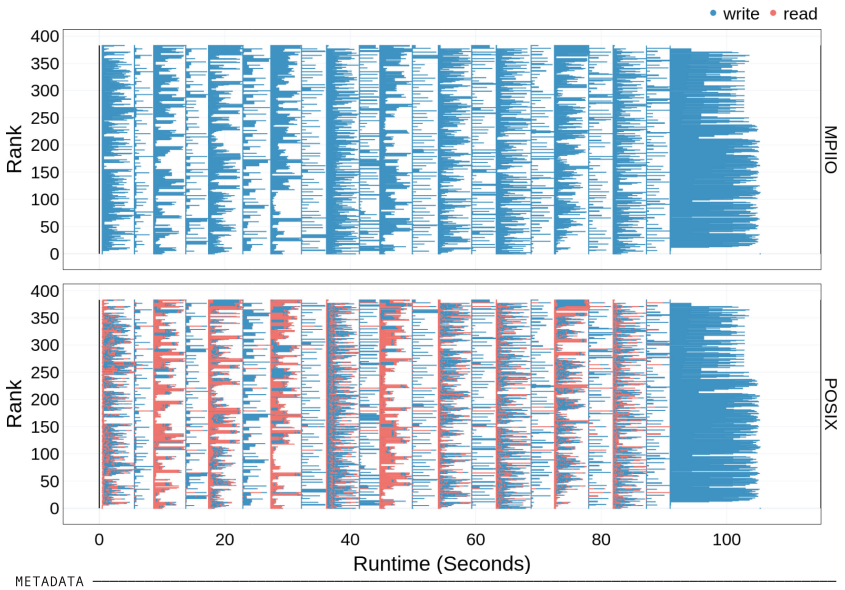
Summit is a 4,608 compute nodes IBM supercomputer at OLCF. Summit is connected to a center-wide 250 PB Spectrum Scale (GPFS) file system, with a peak bandwidth of 2.5 TB/s. It has 154 Network Shared Disk servers, each managing one GPFS Native RAID serving as data and metadata server.

4.2 I/O Bottlenecks in OpenPMD

Open Standard for Particle-Mesh Data Files (OpenPMD) [14] is an open metadata schema targeting particle and mesh data in scientific simulations and experiments. Its library [16] provides back-end support for multiple file formats such as HDF5 [38], ADIOS [25], and JSON [28]. In the context of this experiment, we focus on the HDF5 format to store the 3D mashes [65536 $\times$ 256 $\times$ 256], represented as grids of [64 $\times$ 32 $\times$ 32] composed by [64 $\times$ 32 $\times$ 32] mini blocks. The kernel runs for 10 iteration steps writing after each one. Figure 8 depicts a baseline execution of OpenPMD in the Summit supercomputer, with 64 compute nodes, 6 ranks per node, and 384 processes, prior to applying any I/O optimizations alongside the triggered issues. For this scenario, OpenPMD takes on average 110.6 s (avg. of 5 runs).

Based on the initial visualization and the provided report (Fig. 8), it becomes evident that the application I/O calls are not using MPI-IO's collective buffering tuning option. Furthermore, the majority of the write and read requests are small ($<$ 1MB), which is known to have a significant impact on I/O performance [41]. Moreover, *Drishti* has detected an imbalance when accessing the data. This is further highlighted when the user selects that issue in the interactive web-based visualization.

Nonetheless, after careful investigation, we confirmed that the application and the HDF5 library supposedly used collective I/O calls, though the visualization depicted something entirely different. *Drishti* aided in discovering an issue introduced in HDF5 1.10.5 that caused collective operations to be instead issued as independent by the library. Once that was fixed, we noticed that the application did not use collective metadata operations. Furthermore, *Drishti* reported misaligned accesses which pointed us toward tuning the MPI-I/O ROMIO collective buffering and data sieving sizes to match Alpine's 16MB striping configuration and the number of aggregators. Following the recommendations provided by *Drishti*, the runtime dropped to 16.1 seconds, a 6.8$\times$ speedup from the baseline execution. The complete interactive report for the optimized execution is available in our companion repository.



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

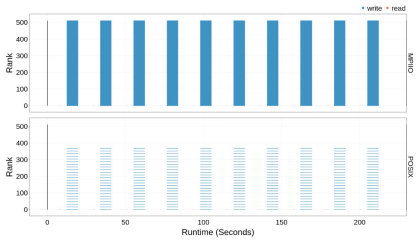
Fig. 8. Interactive visualization and recommendations report generated by *Drishti* for the OpenPMD baseline execution in Summit.

4.3 Improving AMReX with Asynchronous I/O

AMReX [47] is a C++ framework developed in the context of the DOE's Exascale Computing Project (ECP). It uses highly parallel adaptive mesh refinement (AMR) algorithms to solve partial differential equations on block-structured meshes. AMReX-based applications span different areas such as astrophysics, atmospheric modeling, combustion, cosmology, multi-phase flow, and particle accelerators. We ran AMReX with 512 ranks over 32 nodes in Cori supercomputer, with a 1024 domain size, a maximum allowable size of each subdomain used for parallel decomposal as 8, 1 level, 6 components, 2 particles per cell, 10 plot files, and a sleep time of 10 seconds between writes. Table 3 (left) shows the interactive baseline execution and the report generated by *Drishti*.

Table 3. *Drishti* report generated for the AMReX in Cori.

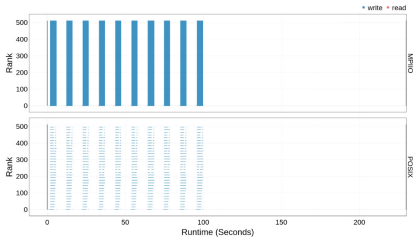
Baseline AMReX




```
METADATA
 ▸ Application is write operation intensive (99.98% writes vs. 0.02% reads)
 ▸ Application is write size intensive (100.00% write vs. 0.00% read)

OPERATIONS
 ▸ Application issues a high number (491648) of small write requests (i.e., < 1MB) which represents 99.99% of all read/write requests
   ↪ 98328 (20.00%) small write requests are to "plt00001.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00002.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00005.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00009.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00000.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00004.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00003.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00006.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00007.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00008.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 (25.41%) and sequential (42.78%) read requests
 ▸ Application mostly uses consecutive (0.01%) and sequential (99.98%) write requests
 ▸ Application issues a high number (491648) of small write requests to a shared file (i.e., < 1MB) which represents 99.99% of all shared file write requests
   ↪ 49164 (10.00%) small writes requests are to "plt00001.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00002.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00005.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00009.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00000.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00004.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00003.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00006.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00007.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00008.h5"
   ↪ Recommendations:
   ↪ Consider coalescing write requests into larger more contiguous ones using MPI-IO collective operations
 ▸ Application uses MPI-IO and write data using 15360 (99.81%) 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
```

Optimized AMReX




```
METADATA
 ▸ Application is write operation intensive (99.61% writes vs. 0.39% reads)
 ▸ Application is write size intensive (100.00% write vs. 0.00% read)

OPERATIONS
 ▸ Application mostly uses consecutive (24.79%) and sequential (33.86%) read requests
 ▸ Application mostly uses consecutive (8.16%) and sequential (99.64%) write requests
 ▸ Application uses MPI-IO and write data using 15360 (99.81%) collective operations
```

From the provided recommendations, since AMReX uses the high-level HDF5 library, we have added the asynchronous I/O VOL Connector [34] so operations are non-blocking and we could hide some of the time spent in I/O while the application continues its computation. Furthermore, as *Drishti* looks at the ratio of operations to trigger some insights, for this particular case, we can verify that the majority of write requests are small (< 1MB) for all 10 plot files. To

increase those requests, we have set the stripe size to 16MB. Table 3 (right) shows the optimized version with a total speedup of 2.1× (from 211 to 100 s). The interactive report is available in our companion repository.

As demonstrated by design choices and these two use cases, *Drishti* meets all the initial criteria (defined in Sect. 1) we set to close the gap between analyzing the collected I/O metrics and traces, automatically diagnosing the root causes of poor performance, and then providing users with a set of actionable suggestions. The designed solution provides a framework that can further be extended and refined by the community to encompass additional triggers, interactive visualizations, and recommendations. We have also conducted a similar analysis for h5bench [18] and the end-to-end (E2E) [24] domain decomposition I/O kernel. These are available in our companion repository.

5 Conclusion

Pinpointing the root causes of I/O inefficiencies in scientific applications requires detailed metrics and an understanding of the HPC I/O stack. The existing tools lack detecting I/O performance bottlenecks and providing a set of actionable items to guide users to solve the bottlenecks considering each application's unique characteristics and workload. In this paper, we design a framework to face the challenges in analyzing I/O metrics: extracting I/O behavior and illustrating it for users to explore interactively, detecting I/O bottlenecks automatically, and presenting a set of recommendations to avoid them.

Drishti, an interactive web-based analysis framework, seeks to close this gap between trace collection, analysis, and tuning. Our framework relies on the automatic detection of common root causes of I/O performance inefficiencies by mapping raw metrics into common problems and recommendations that can be implemented by users. We have demonstrated its applicability and benefits with the OpenPMD and AMReX scientific applications to improve runtime.

Drishti is available on GitHub at github.com/hpc-io/drishti with an open-source license. Scientific community can expand the set of triggers and recommendations. Due to the interactive nature of our solution, we have also provided a companion repository jeanbez.gitlab.io/isc23 with all traces, visualizations, and recommendations in this work.

In our future work, we will integrate additional metrics and system logs to broaden the spectrum of I/O performance issues we can detect and visualize by providing a global API to consume metrics from distinct sources (e.g., Recorder's traces and parallel file system logs). We will also make the thresholds used for I/O phase visualization more generic so that they take into account different factors such as parallel file system performance degradation etc. Apart from this, we will work on approaches to map performance optimization recommendations to the exact source code line numbers through static code analysis and enhance the sample solutions in *Drishti* reports with modified code instead of generic snippets. Lastly, we plan to prepare guidelines on how the community can contribute to this tool as this will aid in keeping up with the latest advancements in I/O libraries and systems. As novel systems come online, we will also reach out to them to provide the necessary support.

Acknowledgment. This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research was also supported by The Ohio State University under a subcontract (GR130303), which was supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR) under contract number DE-AC02-05CH11231 with LBNL. This research used resources of the National Energy Research Scientific Computing Center under Contract No. DE-AC02-05CH11231.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCTOOLKIT: tools for performance analysis of optimized parallel programs. CCPE **22**(6), 685–701 (2010). https://doi.org/10.1002/cpe.1553
2. Agarwal, M., Singhvi, D., Malakar, P., Byna, S.: Active learning-based automatic tuning and prediction of parallel I/O performance. In: 2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW), pp. 20–29 (2019). https://doi.org/10.1109/PDSW49588.2019.00007
3. Bağbaba, A.: Improving collective I/o performance with machine learning supported auto-tuning. In: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 814–821 (2020). https://doi.org/10.1109/IPDPSW50202.2020.00138
4. Behzad, B., Byna, S., Prabhat, Snir, M.: Optimizing I/O performance of HPC applications with autotuning. ACM Trans. Parallel Comput. **5**(4) (2019). https://doi.org/10.1145/3309205
5. Bez, J.L., Ather, H., Byna, S.: Drishti: guiding end-users in the I/O optimization journey. In: 2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW), pp. 1–6 (2022). https://doi.org/10.1109/PDSW56643.2022.00006
6. Bez, J.L., Boito, F.Z., Schnorr, L.M., Navaux, P.O.A., Méhaut, J.F.: TWINS: server access coordination in the I/O forwarding layer. In: 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pp. 116–123 (2017). https://doi.org/10.1109/PDP.2017.61
7. Bez, J.L., Zanon Boito, F., Nou, R., Miranda, A., Cortes, T., Navaux, P.O.: Adaptive request scheduling for the I/O forwarding layer using reinforcement learning. Futur. Gener. Comput. Syst. **112**, 1156–1169 (2020). https://doi.org/10.1016/j.future.2020.05.005
8. Bez, J.L., et al.: I/O bottleneck detection and tuning: connecting the dots using interactive log analysis. In: 2021 IEEE/ACM 6th International Parallel Data Systems Workshop (PDSW), pp. 15–22 (2021). https://doi.org/10.1109/PDSW54622.2021.00008
9. Boito, F.Z., Kassick, R.V., Navaux, P.O., Denneulin, Y.: AGIOS: application-guided I/O scheduling for parallel file systems. In: International Conference on Parallel and Distributed Systems, pp. 43–50 (2013). https://doi.org/10.1109/ICPADS.2013.19
10. Carns, P., Kunkel, J., Mohror, K., Schulz, M.: Understanding I/O behavior in scientific and data-intensive computing (Dagstuhl Seminar 21332). Dagstuhl Rep. **11**(7), 16–75 (2021). https://doi.org/10.4230/DagRep.11.7.16
11. Carns, P., et al.: Understanding and improving computational science storage access through continuous characterization. ACM Trans. Storage **7**(3) (2011). https://doi.org/10.1109/MSST.2011.5937212

12. Carretero, J., et al.: Mapping and scheduling hpc applications for optimizing I/O. In: Proceedings of the 34th ACM International Conference on Supercomputing. ICS'20 (2020). https://doi.org/10.1145/3392717.3392764
13. Darshan team: pyDarshan. https://github.com/darshan-hpc/darshan/tree/main/darshan-util/pydarshan
14. Huebl, A., et al.: openPMD: a meta data standard for particle and mesh based data (2015). https://doi.org/10.5281/zenodo.1167843
15. Knüpfer, A., et al.: Score-P: a joint performance measurement run-time infrastructure for periscope, scalasca, TAU, and vampir. In: Brunst, H., Müller, M.S., Nagel, W.E., Resch, M.M. (eds.) Tools High Perform. Comput., pp. 79–91. Springer, Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31476-6_7
16. Koller, F., et al.: openPMD-api: C++ & python API for scientific I/O with openPMD (2019). https://doi.org/10.14278/rodare.209
17. Kousha, P., et al.: INAM: cross-stack profiling and analysis of communication in MPI-based applications. In: Practice and Experience in Advanced Research Computing (2021). DOIurl10.1145/3437359.3465582
18. Li, T., Byna, S., Koziol, Q., Tang, H., Bez, J.L., Kang, Q.: h5bench: HDF5 I/O kernel suite for exercising HPC I/O patterns. In: CUG (2021)
19. Li, Y., Bel, O., Chang, K., Miller, E.L., Long, D.D.E.: CAPES: unsupervised storage performance tuning using neural network-based deep reinforcement learning. In: SC'17 (2017). DOIurl10.1145/3126908.3126951
20. Liu, Y., Gunasekaran, R., Ma, X., Vazhkudai, S.S.: Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems. In: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 819–829. IEEE (2016). https://doi.org/10.1109/SC.2016.69
21. Lockwood, G.K., Wright, N.J., Snyder, S., Carns, P., Brown, G., Harms, K.: TOKIO on ClusterStor: connecting standard tools to enable holistic I/O performance analysis. CUG (2018). https://www.osti.gov/biblio/1632125
22. Lockwood, G.K., et al.: UMAMI: a recipe for generating meaningful metrics through holistic I/O performance analysis. In: PDSW-DISCS, p. 55–60 (2017). https://doi.org/10.1145/3149393.3149395
23. Lockwood, G.K., et al.: A year in the life of a parallel file system. In: SC'18 (2018). https://doi.org/10.1109/SC.2018.00077
24. Lofstead, J., et al.: Six degrees of scientific data: reading patterns for extreme scale science IO. In: HPDC'11, pp. 49–60. ACM, New York (2011). https://doi.org/10.1145/1996130.1996139
25. Lofstead, J.F., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In: CLADE, pp. 15–24. ACM, NY (2008). https://doi.org/10.1145/1383529.1383533
26. Nicolae, B., et al.: VeloC: towards high performance adaptive asynchronous checkpointing at large scale. In: IPDPS, pp. 911–920 (2019). https://doi.org/10.1109/IPDPS.2019.00099
27. NVIDIA: Nsight systems. https://developer.nvidia.com/nsight-systems
28. Pezoa, F., et al.: Foundations of JSON schema. In: Proceedings of the 25th International Conference on World Wide Web, pp. 263–273 (2016)
29. Shende, S., et al.: Characterizing I/O performance using the TAU performance system. In: ParCo 2011, Advances in Parallel Computing, vol. 22, pp. 647–655. IOS Press (2011). https://doi.org/10.3233/978-1-61499-041-3-647

30. Snyder, S., et al.: Modular HPC I/O characterization with darshan. In: ESPT '16, pp. 9–17. IEEE Press (2016). https://doi.org/10.1109/ESPT.2016.006
31. Stovner, E.B., Sætrom, P.: PyRanges: efficient comparison of genomic intervals in Python. Bioinformatics **36**(3), 918–919 (2019). https://doi.org/10.1093/bioinformatics/btz615
32. Sung, H., et al.: Understanding parallel I/o performance trends under various HPC configurations. In: Proceedings of the ACM Workshop on Systems and Network Telemetry and Analytics, pp. 29–36 (2019). https://doi.org/10.1145/3322798.3329258
33. Tang, H., Koziol, Q., Byna, S., Mainzer, J., Li, T.: Enabling transparent asynchronous I/O using background threads. In: 2019 IEEE/ACM 4th International Parallel Data Systems Workshop (PDSW), pp. 11–19 (2019). https://doi.org/10.1109/PDSW49588.2019.00006
34. Tang, H., Koziol, Q., Ravi, J., Byna, S.: Transparent asynchronous parallel I/O using background threads. IEEE TPDS **33**(4), 891–902 (2022). https://doi.org/10.1109/TPDS.2021.3090322
35. Taufer, M.: AI4IO: a suite of Ai-based tools for IO-aware HPC resource management. In: HiPC, pp. 1–1 (2021). https://doi.org/10.1109/HiPC53243.2021.00012
36. Tavakoli, N., Dai, D., Chen, Y.: Log-assisted straggler-aware I/O scheduler for high-end computing. In: 2016 45th International Conference on Parallel Processing Workshops (ICPPW), pp. 181–189 (2016). https://doi.org/10.1109/ICPPW.2016.38
37. Thakur, R., Gropp, W., Lusk, E.: Data sieving and collective I/O in ROMIO. In: Proceedings Frontiers '99 7th Symposium on the Frontiers of Massively Parallel Computation, pp. 182–189 (1999). https://doi.org/10.1109/FMPC.1999.750599
38. The HDF Group: Hierarchical data format, version 5 (1997). http://www.hdfgroup.org/HDF5
39. The pandas Development Team: pandas-dev/pandas: Pandas (2020). https://doi.org/10.5281/zenodo.3509134
40. Wang, C., et al.: Recorder 2.0: efficient parallel I/O tracing and analysis. In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1–8 (2020). https://doi.org/10.1109/IPDPSW50202.2020.00176
41. Wang, T., et al.: A zoom-in analysis of I/O logs to detect root causes of I/O performance bottlenecks. In: CCGRID, pp. 102–111 (2019). https://doi.org/10.1109/CCGRID.2019.00021
42. Wang, T., et al.: IOMiner: large-scale analytics framework for gaining knowledge from I/O Logs. In: IEEE CLUSTER, pp. 466–476 (2018). https://doi.org/10.1109/CLUSTER.2018.00062
43. Wilkinson, L.: The Grammar of Graphics (Statistics and Computing). Springer-Verlag, Berlin (2005)
44. Xu, C., et al.: DXT: darshan eXtended tracing. CUG (2019)
45. Yildiz, O., et al.: On the root causes of cross-application I/O interference in HPC storage systems. In: IEEE IPDPS, pp. 750–759 (2016). https://doi.org/10.1109/IPDPS.2016.50
46. Yu, J., Liu, G., Dong, W., Li, X., Zhang, J., Sun, F.: On the load imbalance problem of I/O forwarding layer in HPC systems. In: International Conference on Computer and Communications (ICCC), pp. 2424–2428 (2017). https://doi.org/10.1109/CompComm.2017.8322970
47. Zhang, W., et al.: AMReX: block-structured adaptive mesh refinement for multiphysics applications. Int. J. High Perform. Comput. Appl. **35**(6), 508–526 (2021). https://doi.org/10.1177/10943420211022811