DRP: Towards Illuminating the I/O Optimization Path

Hammad Ather
University of Oregon

hather@uoregon.edu

***Abstract*—The existing parallel I/O stack is complex and difficult to tune performance due to the interplay of 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. However, there is a gap between the currently available metrics, the issues they represent, and the application of solutions and optimizations that would mitigate 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 each case. In this paper, we propose an interactive, user-oriented visualization and analysis framework to pinpoint various root causes of I/O performance problems and provide a set of actionable recommendations to improve performance based on the observed characteristics of an application. We evaluate its applicability and correctness in 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.**

***Index Terms*—I/O, insights, visualization, I/O optimization**

I. 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 [1], [2]. Despite that, applications still face poor performance when accessing data. 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.

When applications suffer 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, but 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 [3] and Recorder [4] profilers can also trace I/O operations, providing a fine-grain view of the transformations the requests undergo as they traverse the parallel I/O stack. Nonetheless, despite the availability of such fine-grain traces, there is a gap between the trace collection, analysis, and tuning.

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 [3], [5] 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 [6] and Zoom-in I/O analysis [7]. 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 I/O performance of applications or to increase utilization of I/O system capabilities [8].

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 an interactive visualization based on each file, allowing to focus on a subset of ranks or zoom in to specific regions of the execution;
② Display contextual information about I/O calls (e.g., rank, size, duration, start and end times);
③ Understand how the application issues its I/O requests over time considering operation, request sizes, and spatiality of accesses;
④ Observe transformations as the requests traverse the I/O stack (e.g., MPI-IO to POSIX);
⑤ Detect and characterize the distinct I/O phases of an application throughout its execution;
⑥ Understand how the file system is accessed by the ranks involved in I/O operations;
⑦ Provide an extensible framework so new visualizations and analysis could be easily integrated;
⑧ Identify and highlight common root causes of I/O performance problems;
⑨ Provide a set of actionable items or recommendations based on the detected I/O bottlenecks.

In this paper, we propose an interactive web-based analysis framework named "*AnonIOVis*"[1] to visualize I/O traces, highlight bottlenecks, and help understand the I/O behavior of scientific applications. Achieving 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

[1]*Name of the tool is kept anonymous to facilitate double-blind review.*

solution that contains an interactive component to I/O trace analysis 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. We demonstrate the usage of our framework with multiple case studies and visualize performance bottlenecks and their solutions.

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

II. Related Work

There are a variety of tools that have been developed for performance analysis and visualization, as well as I/O bottleneck detection of HPC applications. We discuss here a few of those tools and explain the novelty of our framework in terms of performance visualization and tuning of I/O bottlenecks.

NVIDA Nsight [9] has been used extensively for the performance analysis and visualization of HPC applications. It is helpful in optimizing the overall performance of the application by providing insights regarding different issues in the application, such as CPU and GPU usage, parallelism and vectorization, and GPU synchronization. TAU [10] is an integrated toolkit for performance instrumentation, measurement, and analysis. TAU captures serial and parallel file I/O, communication, memory, and CPU metrics. Concerning I/O, TAU uses library wrapping to characterize I/O performance, which automates the instrumentation of external I/O packages and libraries. Thus, TAU can intercept POSIX and MPI-IO calls and instrument libraries such as HDF5. Similarly, tools such as Recorder [4], and IOMiner [6] are being used extensively to analyze the I/O performance of HPC applications.

The Total Knowledge of I/O (TOKIO) [11] 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. By acting as an abstraction layer between these component-level monitoring tools and high-level I/O analysis tools, TOKIO collects data from different monitoring tools and normalizes and indexes this data to present a single coherent view to be used by different analysis tools and user interfaces. Unified Monitoring and Metrics Interface (UMAMI) [12] introduces a holistic I/O analysis approach by integrating data across components, such as file systems, application-level profilers, and system components, into a single view. This single view or interface, called UMAMI, is used to provide a complete and coherent view of the issues affecting the I/O subsystem. Metrics for UMAMI are gathered by looking at tools that capture application I/O behavior and storage system traffic and collecting data related to health monitoring, job scheduling, and topology. Both approaches focus on the global view of the I/O system of large-scale machines rather than on the particular I/O issues of each application.

Tools have been developed which use Artificial Intelligence (AI) to predict and mitigate I/O contention in HPC systems. One of these tools is Analytics for I/O (AI4IO) [13], which uses AI to develop IO awareness in the HPC system. AI4IO comes with two tools: PRIONN and CanarIO. PRIONN works on predicting the resource usage of the jobs, whereas CanarIO identifies which jobs are affected by I/O contention. Both of these tools work together to predict I/O contention before it happens and take the necessary steps to mitigate it. INAM [14] presents a novel cross-stack technique to profile and analyze communication across HPC middleware and applications. By analyzing communication across the HPC stack, INAM determines the bottlenecks and provides significant speedup by resolving those bottlenecks. [15] presents an approach to update HDF5 with new parameters for superior I/O performance. It finds the best parameters for HDF5 by conducting controlled experiments on different benchmark settings with multiple repetitions. Behzad et al. [2] present an auto-tuning solution to optimize HPC applications for I/O usage. The I/O kernel of the application is first extracted by the auto-tuning framework. All the possible I/O configurations are passed to the kernel, and the best k configurations are selected. H5tuner takes these configurations as an Extensible Markup Language (XML) file and links it with the application. In order to deal with the large size of the I/O parameter space, the auto-tuning framework uses Genetic Algorithms and I/O performance modeling to reduce the search space.

All the tools mentioned above are very effective in performance visualization and detection of I/O bottlenecks in the HPC systems, but none of these 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 not only providing interactive visualizations showing the I/O performance of the application but also by providing a set of actionable items or recommendations based on the detected I/O bottlenecks. Furthermore, auto-tuning approaches are complementary to this work, as they could harness the provided insights and bottleneck detection to reduce the search space of tunable parameters.

III. "*AnonIOVis*" for Visualization, Diagnosis, and Recommendations

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 phases of our proposed solution.

A. Extracting I/O Behavior from Metrics

Darshan [3] is a tool deployed in several large-scale computing facilities to collect I/O profiling metrics. Darshan

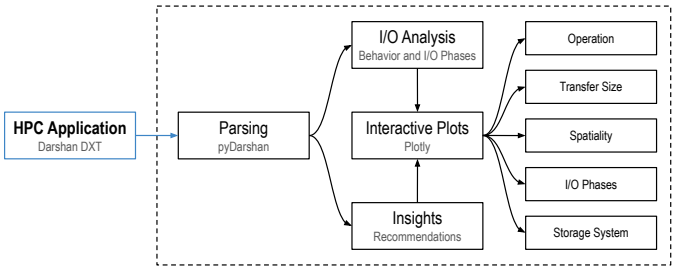


Fig. 1. Workflow used by "*AnonIOVis*" to generate meaningful interactive visualizations and a set of recommendations based on the detected I/O bottlenecks using Darshan DXT I/O traces.

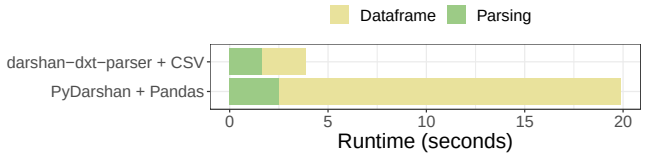


Fig. 2. Comparison of methods to extract and combine the I/O behavior data from Darshan DXT traces required to pinpoint I/O issues and generate interactive visualizations.

collects aggregated statistics with minimal overhead providing a coarse-grain view of application I/O behavior. An extended tracing module (DXT) [16] can also be enabled to capture fine-grain POSIX and MPI-IO traces. Because of its widespread use, we use Darshan log files as input to our solution.

In order to characterize an application's I/O behavior, we require an efficient way to analyze possibly large traces collected by Darshan DXT 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. Fig. 2 summarizes the time taken to obtain the required data in such approach.

Because of the multiple conversions, these additional steps add to the user-perceived time. As an alternative, we have also explored the novel PyDarshan [5], a 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* [17] dataframe. It uses a DarshanReport object, which provides a convenient wrapper to access Darshan logs. Fig. 2 illustrates the performance of both approaches. However, PyDarshan also has its shortcomings when the analysis requires an overall view of application behavior. The package currently returns a data frame containing all trace operations issued by rank, which in the case of "*AnonIOVis*" requires an additional step to iterate over all ranks and group data into a single dataframe for both analysis and interactive visualization. For the trace in Fig. 2, that represents 87.3% of the time. Thus, PyDarshan could be improved to provide direct access to all ranks at once, avoiding costly data preparation when looking at the full picture.

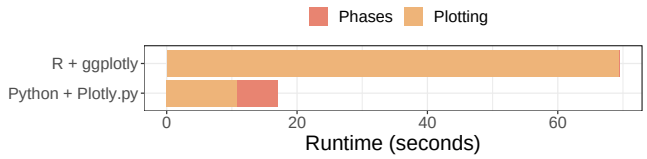


Fig. 3. Comparison of solutions to generate the interactive plots and detect I/O phases from Darshan DXT traces. Both approaches use the Plotly.js library under the hood to generate web-based interactive plots.

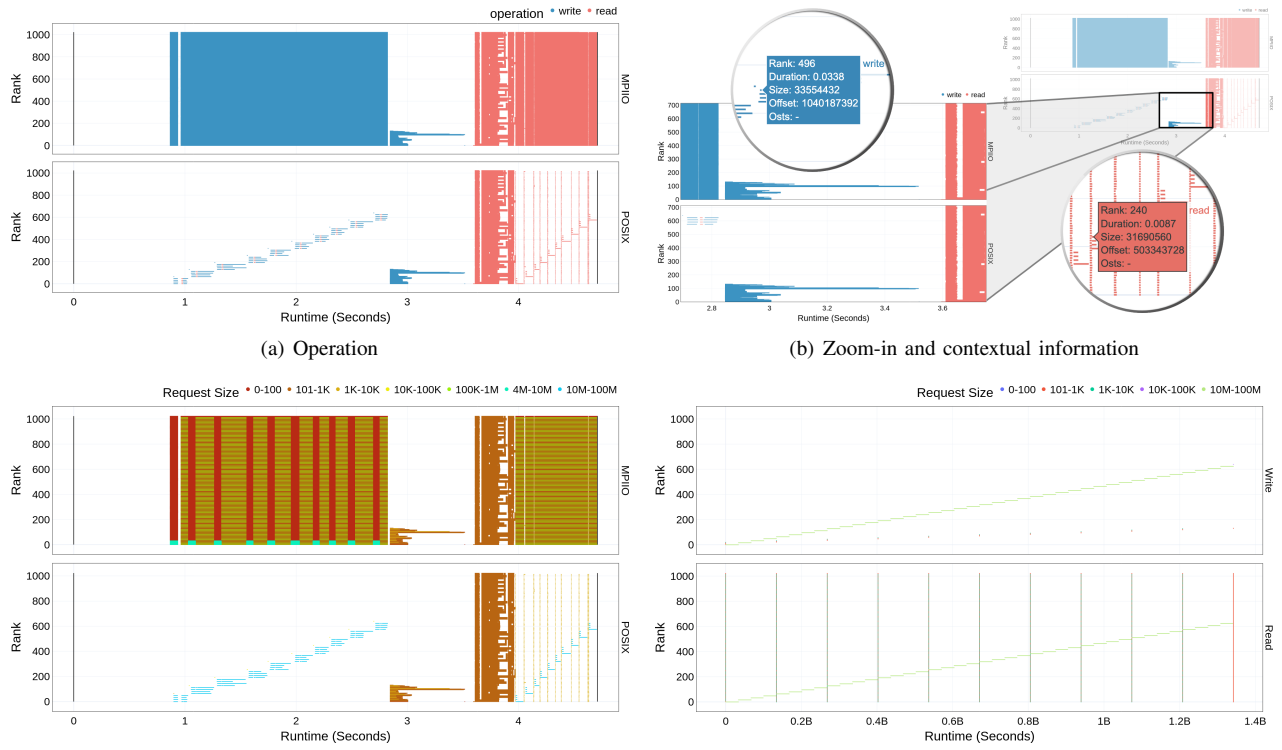
B. Interactively Exploring I/O Behavior

I/O traces can be large for applications with longer runtime or even for relatively short applications with a large number of small I/O requests, making it difficult to analyze and visualize the behavior. Static plots are limited in the information they can represent due to space constraints and pixel resolution. Such an approach often hides the root causes of I/O bottlenecks in plain sight (e.g., when thousands of rank 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 §I), 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 [18] 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 Section III-A 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 "*AnonIOVis*". It also brought the advantage of reducing the total user-perceived time by 84.5% (from avg. of 69.45 to 10.74 seconds), allowing such time to be better spent on detailed analysis of I/O behavior. Fig. 3 summarizes this difference. Section III-C 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 files approaches), "*AnonIOVis*" 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



(a) Operation

(b) Zoom-in and contextual information

(c) Transfer Size

(d) Spatiality

Fig. 4. Sample reports generated by "*AnonIOVis*" 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.

③: operation, data transfer, and spatiality. Fig. 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 [19] or request scheduling [20]–[22]. To shed light on these transformations, "*AnonIOVis*" 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. We will provide interactive examples online, which are now hidden for anonymity.

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.

C. Automatic Detection of I/O Bottlenecks

There are a variety of tools that seek to analyze the performance of HPC applications, as discussed in Section II. 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 I. 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, whereas its Extended Tracing module (DXT) tracks what happens in between, such as different behaviors or I/O phases.

"*AnonIOVis*" 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. "*AnonIOVis*" 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 information presented in the report.

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). 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 II. We briefly discuss some triggers that are directly or indirectly embedded in the visual reports.

1) Small Requests: Scientific applications that issue a large number of small requests often experience poor performance [6], [7]. We consider small requests as smaller than 1MB. Aggregating or buffering operations could significantly improve performance by leveraging a small number of larger requests. MPI-IO's collective buffering implements such solutions that are often not used by applications. In "*AnonIOVis*", we use a tunable threshold (defaults to 10%) to trigger this issue (considering accesses to all files and, especially to shared-filed, where collective I/O could be easily implemented).

2) Rank 0 Heavy-Workload: A simpler approach to accessing data in MPI-based applications is concentrating all I/O requests into a single rank, often rank 0. Furthermore, when using high-level I/O libraries, such as HDF5, rank 0 is also by default responsible for managing metadata operations unless explicitly made collective. Such scenarios force rank 0 to issue a lot of I/O requests when compared to the rest of the workload, often causing slow runtime. A case study of the huge impact of this unbalanced workload is further detailed in Section IV. To trigger this issue, "*AnonIOVis*" checks the number of read and write operations and the total transferred size of each rank. Three scenarios can arise and are considered:

1) The read count, write count, and the total request size of rank 0 are greater than the rest of the ranks, meaning rank 0 is issuing more I/O requests and transferring more data as compared to the rest of the workload;
2) The read count and the total request size of rank 0 are greater than the rest of the ranks;
3) The write count and the total request size of rank 0 are greater than the rest of the ranks.

If detected, a message is issued to the *visualization* module, and rank 0 is highlighted on the interactive graphs. A recommendation is also provided to the user, which can be helpful in mitigating the bottleneck.

3) Workload Imbalance: We apply a similar technique in detecting unbalanced workloads between the ranks, which is often related to how an application was designed to handle its data [30]–[32]. Apart from the number of read and write operations and the total request size, we also collect the total time spent in I/O operations by each rank. We use each metric's mean and standard deviation to define a threshold that filters out any outliers in the data. For this issue to be triggered, each one of the four metrics in a given rank must be higher than its corresponding threshold (mean plus one standard deviation), meaning this rank is handling an unbalanced load. All identified ranks are collected and forwarded to the *visualization* more to be highlighted on the interactive graph.

Figure 5 shows the unbalanced workload detected in the sample Darshan file. Unbalanced ranks are highlighted in

TABLE I
ROOT CAUSES OF I/O PERFORMANCE BOTTLENECKS

Root Causes	Darshan	DXT	System	"*AnonIOVis*"
Too many I/O phases [7]	✓	✓	✗	✓
Stragglers in each I/O phase [23]	✓	✓	✗	✓
Bandwidth limited by a single OST I/O bandwidth [7], [24]	✗	✗	✓	✗
Limited by the small data size [7]	✓	✓	✗	✓
Rank 0 heavy-workload [25]	✓	✓	✗	✓
Unbalanced I/O workload among MPI ranks [7]	✓	✓	✗	✓
Large number of small I/O requests [7]	✓	✓	✗	✓
Unbalanced I/O workload on OSTs [7], [26]	✓	✓	✓	✓
Bad file system weather [7], [12]	✗	✗	✓	✗
Redundant/overlapping I/O accesses [22], [27]	✓	✓	✗	✓
I/O resource contention at OSTs [28], [29]	✗	✗	✓	✗
Heavy metadata load [24]	✓	✗	✗	✓

TABLE II
TRIGGERS EVALUATED BY "*AnonIOVis*" 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 < 1MB.

contrast to the base I/O behavior of the application (faded). "*AnonIOVis*" also issues a set of recommendations such as better balancing the data transfer between the application ranks, tuning the stripe size and count to better distribute the data, and double checking the need to set NO_FILL values if the application uses netCDF and HDF5 to solve the detected issue. "*AnonIOVis*" also identifies read and write imbalance and a high number of random read and small write operations in the application, suggesting the use of MPI-IO collective and considering non-blocking/asynchronous I/O operations.

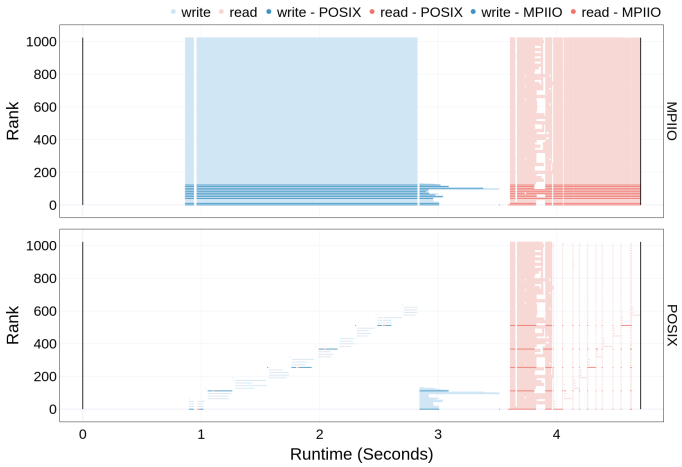


Fig. 5. Unbalanced I/O workload among MPI ranks detected by "*AnonIOVis*" and highlighted on the interactive visualization.

4) Individual Operations: Collective buffering and data sieving [19] are two optimization techniques that are used when applications issue collective operations through the MPI-I/O interface, creating larger and contiguous access to the underlying storage system. "*AnonIOVis*" detects when this interface is used with individual operations, triggering a recommendation to improve performance. The interactive visualization exposes such behavior, as depicted in Fig. 10. In that particular case, despite collective operations being used by the application, a bug in HDF5 was causing the library to issue independent calls instead, harming performance. That explains why both MPI-I/O and POSIX facets look quite similar, highlighting that no aggregations happened.

D. 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 [33]. Request scheduling [20], [21], auto-tuning [2], [34], [35] and reinforcement-learning [36], [37] 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, "*AnonIOVis*" 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 [38] 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 computing 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 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. 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.

Algorithm 1 Merging I/O phases by a threshold

$end \leftarrow df[end][0]$
$prev\_end \leftarrow 0$
while $i < len(df)$ **do**
  if $df[start][i] - end <= threshold$ **then**
    $prev\_end \leftarrow df[end][i]$
  end if
  if $df[start][i] - end > threshold$ OR $i = len(df) - 1$ **then**
    $chunk\_end \leftarrow df[prev\_index : i].copy()$
    $end \leftarrow df[end][i]$
    $prev\_end \leftarrow i$
  end if
end while

Fig. 6 shows a sample I/O phases visualization, that is fully interactive supporting zoom-in/zoom-out. The I/O 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.

Understanding an application's I/O phases allow the detection of additional performance bottlenecks, as detailed by Table I. To showcase how "*AnonIOVis*" could be used in

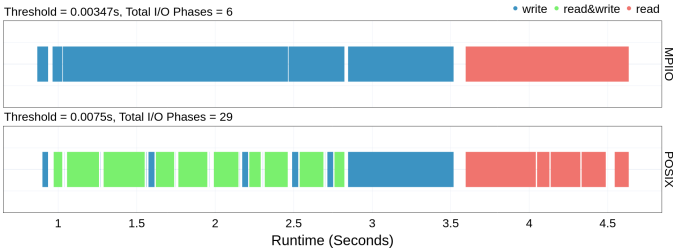


Fig. 6. Interactive I/O phases visualization in MPI-I/O and POSIX layers.

this context, we briefly cover synchronous and asynchronous requests, stragglers, and multiple I/O phases.

1) 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 [39], [40]. 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. In the particular case of HDF5, the Asynchronous I/O VOL Connector [41] 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 Section IV.

2) I/O Stragglers: I/O stragglers in each phase could define the critical path impairing performance. "*AnonIOVis*" has an exclusive plot to highlight the I/O phases and the fast and straggler in each phase. We handle each interface separately due to the transformations that happen as requests go down the stack. Fig. 7 depicts the stragglers in each I/O phase. The dotted lines represent the start and the end of an I/O phase. In each phase, the fastest and the slowest rank is shown. Upon hovering over the rank, we can see the rank number and its duration. Combined with this contextual information, it is possible to detect slow ranks across the entire execution or storage servers consistently delivering slow performance.

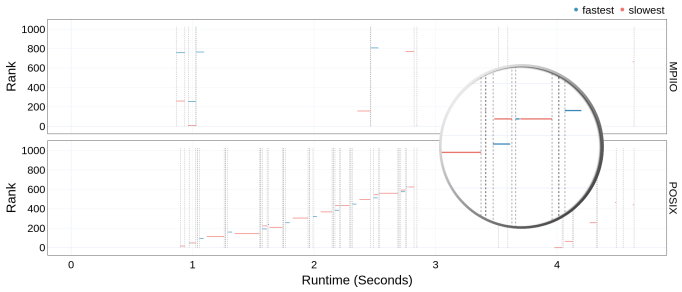


Fig. 7. Stragglers are identified in red for each I/O phase.

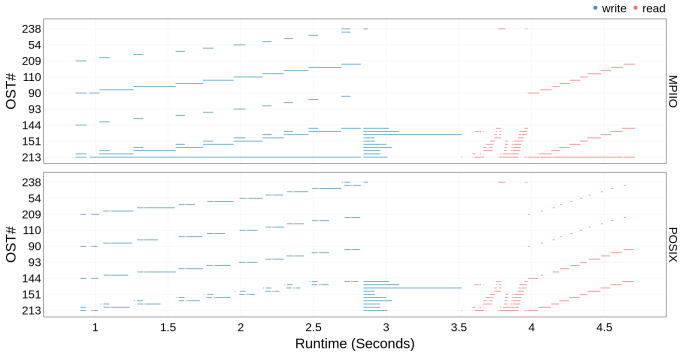


Fig. 8. Lustre data storage (OST) access over time.

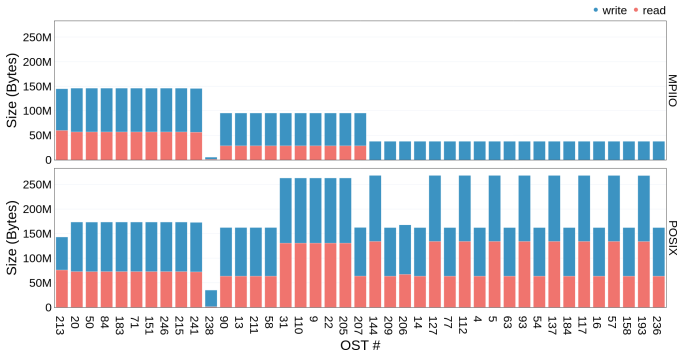


Fig. 9. OST data transfer visualization generated by "*AnonIOVis*".

E. Towards Exploring File System Usage

Additional logs are required to correctly detect bottlenecks related to unoptimized file system accesses, as detailed in Table I. 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. "*AnonIOVis*" provides an exclusive visualization to explore the OST usage of the I/O requests, as depicted in Fig. 8. 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. "*AnonIOVis*" also depicts the data transfer sizes (writes and reads) for each OST at both the MPI-I/O and POSIX levels (Fig. 9).

IV. Results

We demonstrate "*AnonIOVis*" to identify I/O performance bottlenecks. 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).

A. 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 (PFS) capable of achieving a peak I/O bandwidth of 744 GB/s. Cori's PFS is comprised of 244 Object Storage Servers.

On the other hand, 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 both a storage and metadata server.

B. I/O Bottlenecks in OpenPMD

Open Standard for Particle-Mesh Data Files (OpenPMD) [42] is an open meta-data schema targeting particle and mesh data in scientific simulations and experiments. Its library [43] provides back-end support for multiple file formats such as HDF5 [44], ADIOS [45], and JSON [46]. 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 10 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 seconds (avg. of 5 runs).

Based on the initial visualization and the provided report (Fig. 10), 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 [7]. Moreover, "*AnonIOVis*" has detected an imbalance when accessing the data. This is further highlighted in Fig. 11 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. "*AnonIOVis*" aided in the discovery of 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, "*AnonIOVis*" 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. Fig. 12 illustrates the optimized behavior of OpenPMD, dropping 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.

C. 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

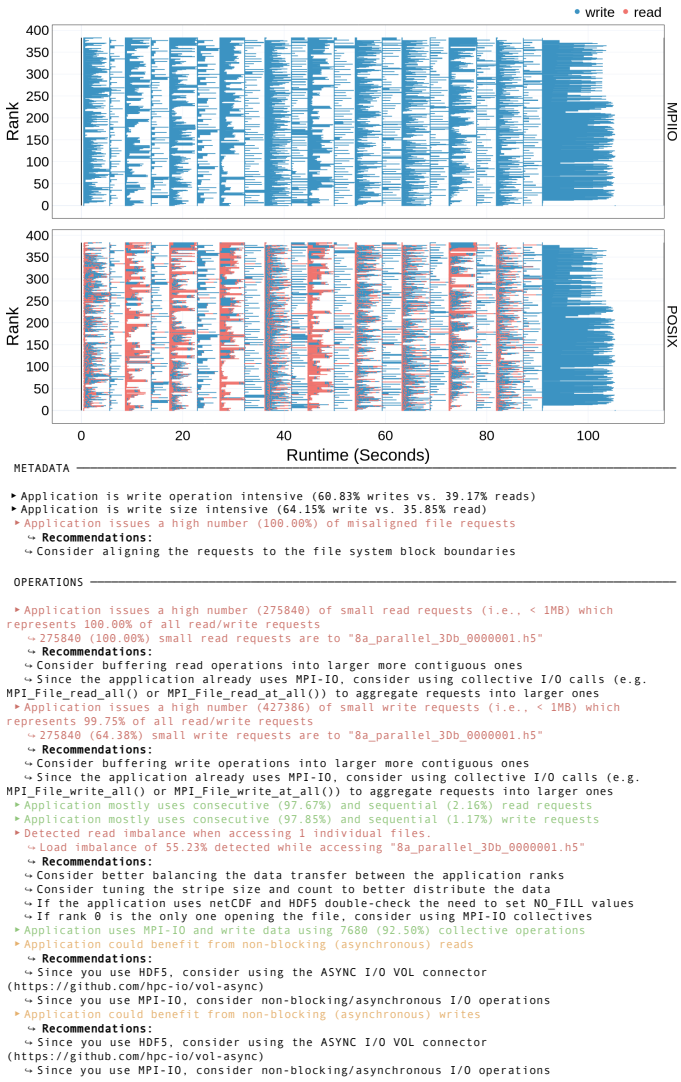


Fig. 10. Interactive visualization and recommendations report generated by "*AnonIOVis*" for the OpenPMD baseline execution in Summit.

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. Fig. 13 shows the interactive baseline execution and the report generated by "*AnonIOVis*".

From the provided recommendations, since AMReX uses the high-level HDF5 library, we have added the asynchronous I/O VOL Connector [41] 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 "*AnonIOVis*" 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 instead. Fig. 14 shows the optimized version with a total speedup of $2.1\times$ (from 211 to 100 seconds).

As demonstrated by design choices and these two use cases, "*AnonIOVis*" meets all the initial criteria (defined in §I) we set to close the gap between analyzing the collected I/O metrics and traces, automatically diagnosing the root causes of poor

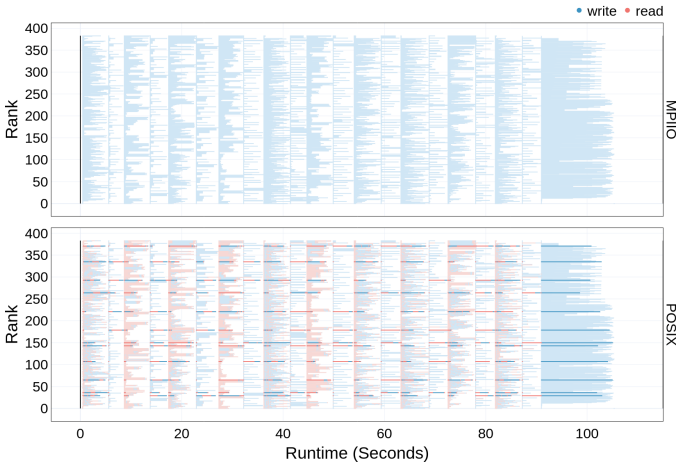


Fig. 11. "*AnonIOVis*" highlights unbalanced workload on the OpenPMD execution. It also illustrates that the application does not use collective I/O.

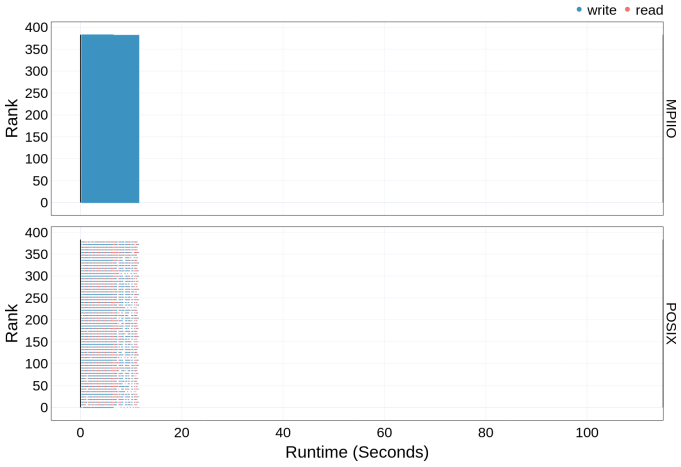


Fig. 12. Optimized OpenPMD execution after following the recommendations provided by "*AnonIOVis*" to avoid common I/O performance bottlenecks.

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 [48] and the end-to-end (E2E) [49] domain decomposition I/O kernel. The interactive visualizations and reports will be available in a companion repository.

V. 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 sought to design a framework that could face the challenges in analyzing I/O metrics for 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.

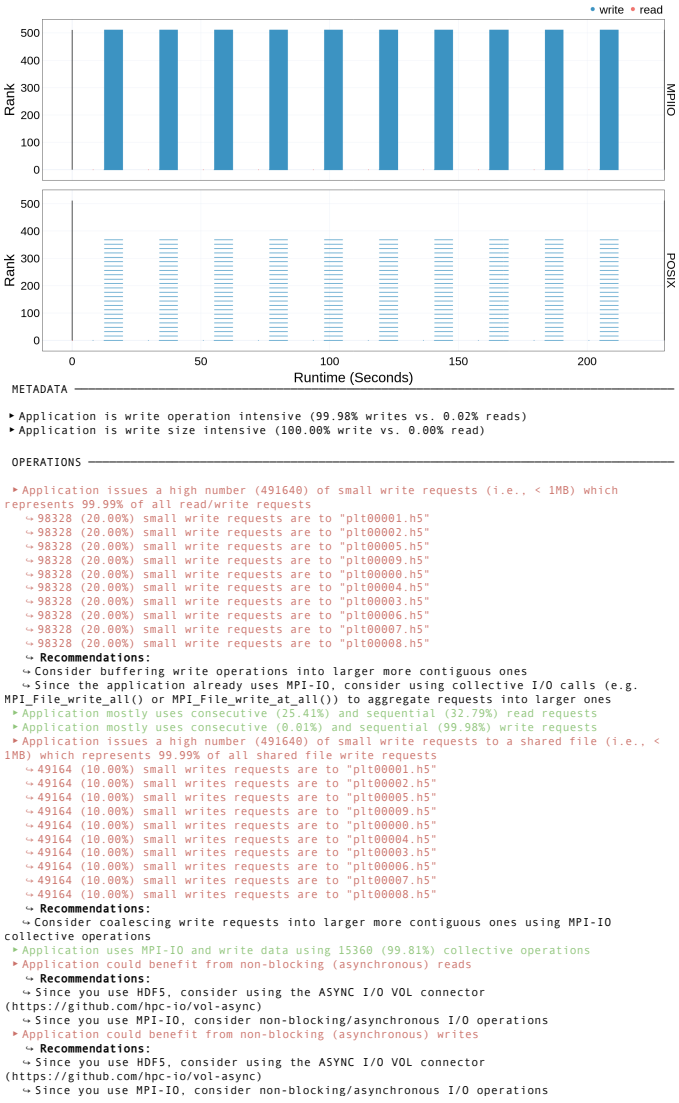


Fig. 13. "*AnonIOVis*" report generated for the AMReX baseline in Cori.

"*AnonIOVis*", 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.

"*AnonIOVis*" is available as open-source for the scientific community to expand the set of triggers and recommendations (links not provided for anonymity). Due to the interactive nature of our solution, we will also provide a companion repository with all traces, analyses, visualizations, and recommendations generated in this work.

In 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

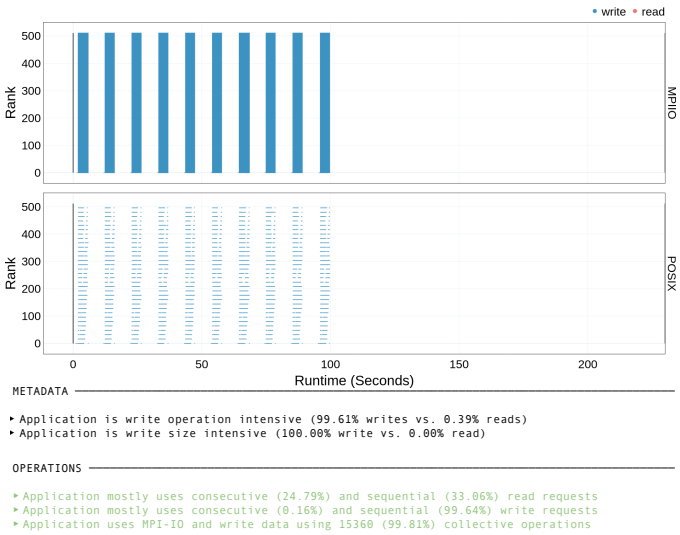


Fig. 14. AMReX execution after using asynchronous operations enabled by using HDF5 ASYNC-VOL connector as recommended by "*AnonIOVis*".

and parallel file system logs).

VI. Acknowledgement

The text for this manuscript is derived from a conference submission and there were co authors of the paper as well. The paper had the following authors: Hammad Ather, Jean Luca Bez, Boyana Norris, and Suren Byna.

References

[1] J. L. Bez *et al.*, "I/O Bottleneck Detection and Tuning: Connecting the Dots using Interactive Log Analysis," in *2021 IEEE/ACM Sixth Int. Parallel Data Systems Workshop (PDSW)*, 2021, pp. 15–22.

[2] 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.

[3] P. Carns *et al.*, "Understanding and Improving Computational Science Storage Access through Continuous Characterization," *ACM Trans. Storage*, vol. 7, no. 3, Oct. 2011.

[4] C. Wang *et al.*, "Recorder 2.0: Efficient Parallel I/O Tracing and Analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2020, pp. 1–8.

[5] Darshan Team, "pyDarshan." [Online]. Available: https://github.com/darshan-hpc/darshan/tree/main/darshan-util/pydarshan

[6] T. Wang *et al.*, "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.

[7] 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 *CCGRID*, 2019, pp. 102–111.

[8] P. Carns, J. Kunkel, K. Mohror, and M. Schulz, "Understanding I/O Behavior in Scientific and Data-Intensive Computing (Dagstuhl Seminar 21332)," *Dagstuhl Reports*, vol. 11, no. 7, pp. 16–75, 2021.

[9] "NVIDIA Nsight Systems." [Online]. Available: https://developer.nvidia.com/nsight-systems

[10] S. Shende *et al.*, "Characterizing I/O performance using the TAU performance system," in *ParCo 2011*, ser. Advances in Parallel Computing, vol. 22. IOS Press, 2011, pp. 647–655.

[11] 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," *CUG*, 1 2018.

[12] G. K. Lockwood *et al.*, "UMAMI: A Recipe for Generating Meaningful Metrics through Holistic I/O Performance Analysis," in *PDSW-DISCS*, 2017, p. 55–60.

[13] M. Taufer, "AI4IO: A Suite of Ai-Based Tools for IO-Aware HPC Resource Management," in *International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2021, pp. 1–1.

[14] P. Kousha *et al.*, "INAM: Cross-Stack Profiling and Analysis of Communication in MPI-Based Applications," in *Practice and Experience in Advanced Research Computing*, 2021.

[15] B. Xie, H. Tang, S. Byna, J. Hanley, Q. Koziol, T. Li, and S. Oral, "Battle of the Defaults: Extracting Performance Characteristics of HDF5 under Production Load," in *CCGrid 2021*, 2021.

[16] C. Xu, S. Snyder, O. Kulkarni, V. Venkatesan, P. Carns, S. Byna, R. Sisneros, and K. Chadalavada, "DXT: Darshan eXtended Tracing," *CUG*, 1 2019. [Online]. Available: https://www.osti.gov/biblio/1490709

[17] The pandas Development Team, "pandas-dev/pandas: Pandas," feb 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3509134

[18] L. Wilkinson, *The Grammar of Graphics (Statistics and Computing)*. Berlin, Heidelberg: Springer-Verlag, 2005.

[19] 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.

[20] F. Z. Boito, R. V. Kassick, P. O. Navaux, and Y. Denneulin, "AGIOS: Application-Guided I/O Scheduling for Parallel File Systems," in *Int. Conference on Parallel and Distributed Systems*, 2013, pp. 43–50.

[21] J. L. Bez, F. Z. Boito, L. M. Schnorr, P. O. A. Navaux, and J.-F. Méhaut, "TWINS: Server Access Coordination in the I/O Forwarding Layer," in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2017, pp. 116–123.

[22] J. Carretero *et al.*, "Mapping and Scheduling HPC Applications for Optimizing I/O," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20, 2020.

[23] N. Tavakoli, D. Dai, and Y. Chen, "Log-assisted straggler-aware i/o scheduler for high-end computing," in *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, 2016, pp. 181–189.

[24] G. K. Lockwood *et al.*, "A Year in the Life of a Parallel File System," in *SC'18*, ser. SC '18, 2018.

[25] J. Yu, G. Liu, W. Dong, X. Li, J. Zhang, and F. Sun, "On the Load Imbalance Problem of I/O Forwarding Layer in HPC Systems," in *Int. Conf. on Computer and Communications (ICCC)*, 2017, pp. 2424–2428.

[26] J. Yu *et al.*, "On the load imbalance problem of I/O forwarding layer in HPC systems," in *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, 2017, pp. 2424–2428.

[27] S. Snyder *et al.*, "Modular HPC I/O Characterization with Darshan," in *ESPT '16*. IEEE Press, 2016, p. 9–17.

[28] O. Yildiz *et al.*, "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 750–759.

[29] H. Sung *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*, 2019, p. 29–36.

[30] B. Xie *et al.*, "Characterizing Output Bottlenecks in a Supercomputer," in *SC'12*, 2012, pp. 1–11.

[31] O. Yildiz *et al.*, "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 750–759.

[32] N. Tavakoli, D. Dai, and Y. Chen, "Client-Side Straggler-Aware I/O Scheduler for Object-Based Parallel File Systems," *Parallel Comput.*, vol. 82, no. C, p. 3–18, feb 2019.

[33] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 819–829.

[34] M. Agarwal, D. Singhvi, P. Malakar, and S. Byna, "Active Learning-based Automatic Tuning and Prediction of Parallel I/O Performance," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, 2019, pp. 20–29.

[35] A. Bağbaba, "Improving Collective I/O Performance with Machine Learning Supported Auto-tuning," in *IEEE Int. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 814–821.

[36] Y. Li, O. Bel, K. Chang, E. L. Miller, and D. D. E. Long, "CAPES: Unsupervised Storage Performance Tuning Using Neural Network-Based Deep Reinforcement Learning," in *SC'17*, Nov. 2017.

[37] J. L. Bez, F. Zanon Boito, R. Nou, A. Miranda, T. Cortes, and P. O. Navaux, "Adaptive Request Scheduling for the I/O Forwarding Layer Using Reinforcement Learning," *Future Generation Computer Systems*, vol. 112, pp. 1156–1169, 2020.

[38] E. B. Stovner and P. Sætrom, "PyRanges: efficient comparison of genomic intervals in Python," *Bioinformatics*, vol. 36, no. 3, pp. 918–919, 08 2019.
[39] H. Tang, Q. Koziol, S. Byna, J. Mainzer, and T. Li, "Enabling Transparent Asynchronous I/O using Background Threads," in *2019 IEEE/ACM Fourth Int. Parallel Data Systems Workshop (PDSW)*, 2019, pp. 11–19.
[40] B. Nicolae *et al.*, "VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale," in *IPDPS*, 2019, pp. 911–920.
[41] H. Tang, Q. Koziol, J. Ravi, and S. Byna, "Transparent Asynchronous Parallel I/O Using Background Threads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 891–902, 2022.
[42] A. Huebl *et al.* (2015) openPMD: A meta data standard for particle and mesh based data. [Online]. Available: doi.org/10.5281/zenodo.1167843
[43] F. Koller *et al.* (2019) openPMD-api: C++ & Python API for Scientific I/O with openPMD. [Online]. Available: doi.org/10.14278/rodare.209
[44] The HDF Group. (1997-) Hierarchical Data Format, version 5. [Online]. Available: http://www.hdfgroup.org/HDF5
[45] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and Integration for Scientific Codes through the Adaptable IO System (ADIOS)," in *CLADE*. NY, USA: ACM, 2008, pp. 15–24.
[46] F. Pezoa *et al.*, "Foundations of JSON Schema," in *Proceedings of the 25th International Conference on World Wide Web*, 2016, pp. 263–273.
[47] W. Zhang *et al.*, "AMReX: Block-structured adaptive mesh refinement for multiphysics applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, pp. 508–526, 2021.
[48] 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 *CUG*, 2021.
[49] J. Lofstead *et al.*, "Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO," in *HPDC'11*. New York, NY, USA: ACM, 2011, p. 49–60.