IOMiner: Large-scale Analytics Framework for Gaining Knowledge from I/O Logs

Teng Wang*, Shane Snyder†, Glenn K. Lockwood*, Philip Carns†, Nicholas J. Wright*, Suren Byna*
*Lawrence Berkeley National Laboratory**
Argonne National Laboratory†
Email: {tengwang, glock, njwright, sbyna}@lbl.gov, {ssnyder, carns}@mcs.anl.gov

Modern HPC systems are collecting large amounts of I/O performance data. The massive volume and heterogeneity of this data, however, have made timely performance of in-depth integrated analysis difficult. To overcome this difficulty and to allow users to identify the root causes of poor application I/O performance, we present IOMiner, an I/O log analytics framework. IOMiner provides an easy-to-use interface for analyzing instrumentation data, a unified storage schema that hides the heterogeneity of the raw instrumentation data, and a sweep-line-based algorithm for root cause analysis of poor application I/O performance. IOMiner is implemented atop Spark to facilitate efficient, interactive, parallel analysis. We demonstrate the capabilities of IOMiner by using it to analyze logs collected on a large-scale production HPC system. Our analysis techniques not only uncover the root cause of poor I/O performance in key application case studies but also provide new insight into HPC I/O workload characterization.

I. Introduction

Parallel I/O is a crucial component for handling massive data movement on high-performance computing (HPC) systems. In an effort to characterize, understand, and eventually optimize parallel I/O performance of applications, I/O logs are captured at several stages of the I/O path. At the application level, instrumentation tools such as Darshan [1], [2] collect detailed I/O statistics. Darshan stores the data in a write-optimized format; for example, each MPI job produces a compressed binary log file. The number of Darshan logs produced on production HPC systems each month depends on the number of executed jobs[1], and the data volume ranges from hundreds of gigabytes to multiple terabytes. File-system-monitoring tools, such as the Lustre Monitoring Tool (LMT) [3], periodically gather I/O load on file system servers and store the traces either in files or in databases. The frequency of file system instrumentation is a few seconds, resulting in tens of thousands of records per day. In addition, job schedulers such as Slurm [4] can log millions of jobs' resource utilization each month, such as the compute nodes used and the jobs' execution times.

An integrated analysis of all I/O instrumentation data faces several challenges. First, instrumentation tools provide ad hoc interfaces for data extraction, requiring tedious manual effort in resolving their incompatibilities. Second, instrumentation tools often store data in write-optimized format, making analytics on such formats inefficient.

In general, analysis has relied on manually written scripts, a time-consuming process with limited portability. Another option is to load data into a database and apply SQL queries. We have previously analyzed multilevel and multiplatform I/O logs by loading data into a MySQL database [5]. However, the performance of data analytics can be limited by the single database server and parallel databases on HPC system are rarely installed.

As a step toward solving the challenges of analyzing multilevel I/O instrumentation data that is massive in size and heterogeneous, and allowing users to easily identify the root causes of poor I/O performance from the complex instrumentation data, we introduce the *IOMiner* framework. The main components of IOMiner are an extensible set of unified interfaces that can be used to compose common I/O analysis operations on multilevel instrumentation data, a query-friendly and unified storage schema that hides the heterogeneity of different schema of instrumentation data and is optimized for parallel analytics on HPC, and a sweep-line-based [6] analysis function that helps users easily identify the root causes for an application's poor I/O performance. The IOMiner framework is built on Apache Spark [7], enabling interactive and ad hoc querying and statistical analysis.

The main contributions of this paper are as follows.

- A set of unified interfaces that can be used to compose the first-order and in-depth data analysis of different types of instrumentation data
- A query-friendly and unified storage schema that fuses together different instrumentation data and loads logs on demand based on the analysis to be performed
- A layout-aware data placement and task-dispatching framework for parallel data analytics using Spark. Both data placement and task dispatching are specialized for HPC storage architecture
- A sweep-line analysis function that helps identify the root causes for an application's poor I/O performance

We have evaluated IOMiner's capabilities to support both first-order and more in-depth analysis. Our analysis ranges from the distribution of the sequential I/O, per job read/write

[1]Without otherwise noted, "job" in our context refers to a program execution.

2168-9253/18/$31.00 ©2018 IEEE
DOI 10.1109/CLUSTER.2018.00062

ratio, and usage of the customized striping configuration, to the usage of different I/O middleware and the composition of various file-sharing patterns. Our analysis provides valuable insights for both system administrators and I/O specialists. Our analyses with IOMiner also include the root cause identification using the sweep-line technique for jobs experiencing poor I/O bandwidth.

The remainder of the paper is organized as follows. We present the IOMiner framework including the interfaces, storage schema, Spark-based implementation, and approach for analyzing the root causes of poor I/O performance in Section II. In Section III, we evaluate IOMiner capabilities, and in Section IV, we discuss related work. We conclude in Section V with a brief discussion of future work.

II. Design of IOMiner

In this section, we first present the design of high-level interfaces of IOMiner. We then outline IOMiner's unified storage schema for associating together various instrumentation data. We also elaborate on the sweep-line-based evaluation of applications' low I/O performance.

A. Unified Query Interfaces for I/O Log Analytics

To simplify users' effort and to make IOMiner flexible and efficient for handling queries, we have designed IOMiner with a set of interfaces applicable to different log types (e.g., Darshan, Slurm, and LMT). These interfaces include *sort*, *group*, *filter*, *project*, and *join*, which are all common to SQL. IOMiner also provides additional operators (e.g., *percentile* and *bin*) that are often used for I/O log analytics. IOMiner further complements the functions of the existing interfaces with support for user-defined functions, thus allowing users to flexibly express complex queries. All of these interfaces are layered on top of Spark, allowing the output of one operator to be supplied as the input of another operator using Spark RDD (resilient distributed dataset).

Algorithm 1 gives an overview of how to use IOMiner. Line 14 returns a list of tuples containing the jobs whose write sizes are smaller than 1 GB. This analysis is satisfied by a pipeline of operators stitched together. In line 15, the user-defined function *classify_workload* bins each tuple of *job_tuples* based on *bytes_written* (e.g., $[0-1)$ GB is workload 0, $[1, 10)$ GB is workload 1, $[10, \infty)$ GB is workload 2). These bin values are tagged to each element in *job_tuples* as a new field "workload." In line 17, the user-defined function *calculate_ost_count* accepts file tuples grouped by *job_id* and calculates the number of distinct OSTs for each job. In line 19, the percentile operator sorts *selected_tuples* by *bytes_written* and returns a percentile list for *bytes_written*. It answers queries like "What is the distribution of write sizes across all the jobs?"

Algorithm 1: Example analysis code using IOMiner

```
1  # Instantiate an IOMiner instance
2  miner = IOMiner("setup.json")
3  # construct IOMiner's storage format
4  miner.init_stores(start_date, end_date)
5  # load Darshan's job-level statistics as tuples
6  job_tuples = miner.load("darshan_job")
7  # load Darshan's file-level statistics
8  file_tuples = miner.load("darshan_file")
9  # load Slurm job scheduler statistics
10 slurm_tuples = miner.load("slurm")
11 # load Lustre Monitoring Tools (LMT) traces
12 lmt_tuples = miner.load("lmt")
13 # filter jobs that write less than 1GB data and bin them
14 selected_tuples = job_tuples.project("job_id",
     "bytes_written").filter("bytes_written < 1GB")
15 binned_tuples = job_tuples.bin("workload",
     classify_workload, "bytes_written")
16 # select a list of tuples containing each job's ID and
     the number of Lustre storage targets (OSTs) used
17 grouped_tuples = file_tuples.group("job_id",
     calculate_ost_count)
18 # generate a percentile list for Bytes_Written
19 percents = job_tuples.percentile("bytes_written")
20 # return each job's node count (in Slurm log) and and
     write/read size (in Darshan log)
21 joined_tuples = job_tuples.join(slurm_tuples,
     "job_id").project("job_id", "node_count",
     "bytes_read", "bytes_written")
22 # return the jobs that write more than 10GB data
23 low_bw_tuples = job_tuples.filter("bytes_written >
     10GB").join(file_tuples, "job_id")
24 # extract contributing factors for poor write
     performance
25 perf_factors = low_bw_tuples.extract_perf_factors("0",
     "1GB", "write")
```

B. Query-Friendly Storage Schema

A unified set of query interfaces eases users' effort in defining analytic functions. Various challenges arise in providing efficient execution of these functions because of distinct characteristics of instrumentation data. For example, existing instrumentation tools generally provide vendor-specific interfaces, and it is nontrivial to implement the same function multiple times based on the interfaces of each instrumentation tool. Moreover, the storage schema of existing instrumentation tools are not designed to efficiently support analytics and query patterns. Mining results from their individual formats can incur heavy overheads. For instance, Darshan produces one log file for each job, which can be parsed by Darshan utility[2] into a human-readable log file containing both coarse-grained job-level statistics, such as the total number of reads issued by all processes in the

[2]More information about "Darshan utility" is available at http://www.mcs.anl.gov/research/projects/darshan/docs/darshan3-util.html

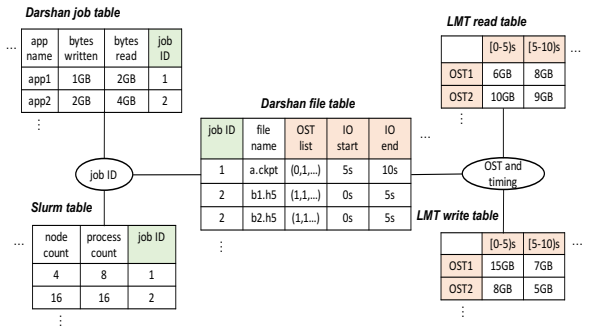

Figure 1. Overview of IOMiner's unified storage schema. Different tables are connected to each other based on the fields in the same colors. Slurm tables, LMT tables, and Darshan tables are representative of job scheduler instrumentation, storage-server-side instrumentation, and application-level instrumentation, respectively.

job, as well as fine-grained file-level statistics, such as the total number of reads issued on each file. These two types of statistics are stored in two different regions of the log file. For a query that requires a scan of all job-level and file-level statistics, an analysis framework has to open/close all the jobs' log files and to perform repeated seek operations inside each log to extract all the requested information. Answering this query incurs substantial metadata access (with file open/close) and file seek overhead, since Darshan can produce millions of logs in a month.

To address these challenges, we have designed a unified storage schema that abstracts away the format difference between diverse instrumentation tools and is query-friendly. In Figure 1, the statistics of different instrumentation tools are formatted as tables and stored as separate files. Formatting happens when the users instantiate an IOMiner instance and invoke *init_stores* to convert logs produced during a given time period (Line 4 of Algorithm 1); only statistics in logs of that period are extracted and reshaped into IOMiner's storage format. This formatting operation happens only once. Later operations for the same time period will not repeat this formatting process if these tables already exist. In Figure 1, the formatted tables can be connected with each other based on specific fields. For instance, the Slurm table contains job statistics (e.g., number of cores/nodes used in a job) supplementary to Darshan's per job statistics, so tuples in the Slurm table can be associated with Darshan tables using job_id. On the other hand, LMT tables contain the system-wide read/write size for each OST during each 5-second interval (see Figure 1); the Darshan file table also contains the OST list for each file and the corresponding I/O timing. Associating these two types of logs based on OSTs and I/O time allows users to derive valuable insights, such as how I/O operations on one file are interfered with by other I/O operations being serviced on the same set of OSTs.

In order to avoid the excessive metadata access and file seek overheads of using Darshan's format, IOMiner combines the job-level statistics of all Darshan logs into a single table (Darshan job table in Figure 1). IOMiner does the same for the file-level statistics and stores them into a separate table (Darshan file table in Figure 1). All tuples in these tables are ordered by job_id. By extracting the statistics from Darshan's individual job logs and storing them in two tables, a query based on a given condition (e.g., $bytes\_written < 1GB$) does not have to scan through all the job logs and thus avoids the repeated open/close overhead. On the other hand, separating the job-level and file-level statistics into two tables more efficiently answers the queries performed exclusively on job-level and file-level statistics. For instance, answering a query that could be satisfied by the job-level statistics does not have to scan the file-level statistics if they are separated. In order to accelerate analytics based on the file system (Lustre in this case) information, IOMiner stores the list of storage targets (Lustre OSTs) used for each file access as a bitmap (shown in the Darshan file table in Figure 1). With this design choice, users can easily calculate the OST count used by each job by performing a union operation on all bitmaps of its files. They can also find the OSTs used by a selected set of jobs by performing intersection operations on the bitmaps of each job.

C. Spark-based Implementation for HPC Environments

The storage schema mentioned in Section II-B is best suited to running analytics operations using a single process, since only one or two tables are constructed for each log type (i.e., one for Slurm, two for Darshan, as shown in Figure 1). However, supercomputers can run as many as millions of jobs each month. Since Slurm and Darshan create one or multiple long tuples for each job, mining useful information from their tables using a single process is time-consuming. To accelerate data analysis, IOMiner uses parallel processing implemented with PySpark, a Python API for the Spark framework. In PySpark, the driver process splits data into multiple partitions and parallelizes data analytics by dispatching tasks to multiple executors. Since HPC adopts a different storage architecture from the cloud computing environment [8], where Spark is typically used, we have designed a specialized data layout and task-dispatching mechanism for Spark on HPC, which parallelizes the data-loading operations from the storage servers (e.g., OST for Lustre) in a layout-aware manner.

Figure 2 shows this data layout. The Darshan job table, Darshan file table, and Slurm table are split into the same number of subtables. Each subtable is placed on one OST as a separate file, and IOMiner intentionally balances the number of subtables on each OST. In doing so, it also places Darshan files and Slurm files that contain the same set of jobs on the same OSTs. For instance, in Figure 2, three Slurm subtables, Darshan job subtables, and Darshan file subtables belonging to Job [1–30] and [31–60] are placed on OST1 and OST2, respectively. In this way, each OST contains 9 subtables belonging to the same set of jobs, and

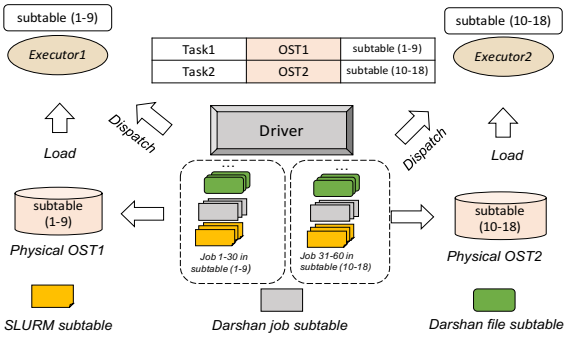

Figure 2. Data distribution and task dispatching of Spark-based implementation for HPC. The Slurm table, Darshan job table, and Darshan file table are partitioned into 6 subtables each; each subtable contains information of 10 jobs. The Slurm table, Darshan job table, and Darshan file subtables containing the same set of jobs are placed on the same OST.

a joining operator based on job_id does not incur data traffic on additional OSTs, isolating each executor's traffic on one OST without competing for other OST's bandwidth.

When IOMiner is instantiated, the driver constructs a list of data-loading tasks for all the executors and distributes the tasks. These executors load data into their memory in parallel (see Figure 2). Each task in the driver's task list contains the subtables to be dispatched to one executor. The task list is created in a way that isolates each executor's load operations on one OST. This approach minimizes the contention when multiple executors are competing for the same set of OSTs. In Figure 2, Executors 1 and 2 need to contact with only one OST, since all their assigned subtables are placed on one OST. When the executor count is not same as the OST count, IOMiner assigns subtables in a way that either each OST is shared by a balanced and distinct set of executors (Executor count $>$ OST count) or each executor isolates its traffic on a distinct set of OSTs (OST count $>$ Executor count).

D. Sweep-line-based Root Cause Analysis of Poor I/O Performance

Analyzing an application's poor I/O performance has been a common effort of both the application developers and the system administrators. However, existing approaches are generally based on manual profiling; no easy approach has been devised that can filter applications with poor-I/O performance from a massive set of logs and provide insightful feedback to the users.

To fill this void, we designed IOMiner to mine the instances of poorly performing I/O from logs and to identify how different contributing factors to performance are playing a role: in Line 23-25 of Algorithm 1, the *extract_perf_factors* function returns a list of tuples, where each tuple includes the values of a set of key performance-contributing factors to each job whose write bandwidth is between 0 and 1GB/s. We currently consider the following five common contributing factors in IOMiner.

- **Small I/O requests (Small) –** The percentage of small I/O requests among all the I/O requests: A large number incurs longer time for writing/reading data to/from disks because of slow file seek performance.
- **Nonconsecutive I/O requests (Non-consec) –** The percentage of noncontiguous I/O requests: I/O requests that are not requesting consecutive byte streams result in small and random I/O requests to the file system.
- **Utilization of collective I/O (Coll):** A value of 0 or 1 that indicates whether collective I/O has been used in a job. A job using collective I/O typically optimizes its read/write operations by transforming the small, nonconsecutive I/O into fewer larger ones.
- **Number of OSTs used by each job (OST#):** The total number of distinct OSTs used by a job. A large value means that a job uses more storage resource.
- **Contention level (Proc/OST):** The ratio of the process count accessing a file to the file's stripe count. A high value implies that a large number of processes are competing for the same OSTs' bandwidth.

While these factors are considered as the common reasons for an application's low I/O performance, we believe there can be other contributing factors outside this scope, such as bad OST performance caused by occasional device failure. IOMiner is extensible to consider additional factors.

A key question is how to filter the low-bandwidth jobs accurately and to compute the values of the contributing factors. As one straightforward approach, we derive these values based on Darshan's job-level statistics (e.g., tuples in the Darshan job table in Figure 1). For instance, we calculate each job's write bandwidth B_{write} using Equation 1, where S_{write} denotes the total bytes written by this job and T_{write_end} and T_{write_start} are the completion time of the last write and the start time of the first write, respectively. All these values are available in the Darshan job table.

$$B_{write} = S_{write}/(T_{write\_end} - T_{write\_start}) \quad (1)$$

These write time measurements are too coarse-grained, however, and we cannot accurately reflect the true write bandwidth of the application. For instance, in Figure 3,

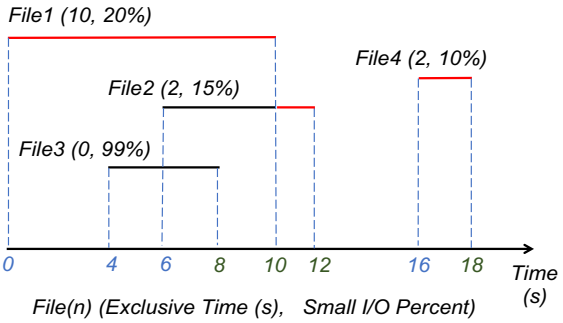

Figure 3. Schematic representation of the I/O timing of all the files in a job. The critical path is marked in the red line, which is the period that has I/O activity. Exclusive time is the non-overlapping I/O time for each file that falls on the critical path.

processes in one job produce four files during different time

slots. There is also an I/O idle period (between 12 s and 16 s on the x-axis). In this example, the write time reported by Darshan is 18 s; however, the actual write time without the idle time is 14 s. Consequently, this approach downplays the true write bandwidth and can falsely mark a job as a low-performance instance. Instead, we measure the write time by looking at the timing on the *critical path*, defined as the period that has I/O activity. In Figure 3, the critical path includes $0 - -12$ seconds and $16 - -18$ seconds. To find out the critical path, we introduce the concept of *critical files*, defined as the minimum set of files whose I/O time cover the critical path. In Figure 3, File1, File2, and File4 are the critical files. We further define *exclusive time* as the non-overlapping portions of I/O time for the critical files. In Figure 3, the exclusive times for File1, File2, and File4 are marked in red, and the lengths are 10 s, 2 s, and 2 s, respectively. The total I/O time is derived by summing up these exclusive times. Following this more accurate timing, we calculate a small I/O percentage of these critical files (P_{small}) by Equation 2.

$$P_{small} = \frac{\sum_{i=1}^{n} p_i t_i}{\sum_{i=1}^{n} t_i} \quad (2)$$

In this equation, p_i refers to the small I/O percentage of $File_i$, t_i refers to the exclusive time of $File_i$, and n is the number of critical files. The resultant P_{small} is an average of all critical files' p_i weighted by their ratio of exclusive time (t_i) to the total I/O time ($\sum_{i=1}^{n} t_i$). The value of nonconsecutive I/O percentage, and OST contention levels can be derived by using the same approach. The benefits of this approach are threefold. First, it helps users precisely filter the low-bandwidth jobs from all the analyzed jobs. Second, it allows users to identify the potential causes for a job's low performance from the above contributing factor values. Third, for those jobs whose performance does not exhibit strong correlation with the calculated contributing factor values, it provides a list of critical files, so that users can identify the causes from these files' I/O statistics.

A key question is how to extract the critical files and their exclusive time. As a naive approach, one can determine the critical files in n rounds of iterations. Each round adds one critical file to the solution by comparing the timing of all the files not in the solution. This approach gives $O(n^2)$ complexity. Instead, we solve this problem more efficiently ($O(nlog(n))$) using a sweep-line algorithm [6]. It is an algorithmic paradigm in computational geometry. The idea behind sweep line is to image a line swept across the whole plane (e.g., the plane containing all the lines in Figure 3), stopping at the points that have I/O activities (e.g., 0, 4, 6, 16 as I/O starts for a file, and 8, 10, 12, 18 as I/O ends), and updating the critical paths and exclusive times upon each activity. The complete solution is available once this line reaches the final activity (e.g., 18 s in Figure 3). The solution is shown in Algorithm 2.

Algorithm 2: Sweep-line-based poor I/O analytics

Input: *file_lst*: the list of all the files in a job

Output: *share_lst*: the list of time shares for each file

```
1  begin
2      for f in file_lst do
3          points.add((f.start, f.name, type_start))
4          points.add((f.end, f.name, type_end))
5      sort points by time
6      for p in points do
7          if p.type == type_start then
8              if heap.size() == 0 then
9                  first = p
10             heap.push(p)
11         if p.type == type_end then
12             heap.del(p)
13             if p.fname == first.fname then
14                 share_lst.add((first.fname, first.start,
                    p.end))
15                 if heap.size() ! = 0 then
16                     first = heap.peek()
17                     first.start = p.end
```

Lines 2-4 show an iteration over all the files. Two tuples are created for each file based on its start_time and end_time, respectively. These points are sorted in an ascending order based on their times (Line 5). Then the algorithm sweeps through all these points (Line 6-17) to add each encountered point of type "start" (referred to as *start point*) to a heap (Line 10) and to remove it from a heap when its counterpart *end point* is encountered (Line 12). In doing so, the time range of all stashed points in the heap dynamically changes. For the scenario shown in Fig. 3, when File1's start point is added first, the current time range is 0 to 10. When File3's start point is added next, the range remains as 0 to 10 because File3's range is 4 to 8, which is a subset of the current range. When File2 is added, the range is adjusted to 0 to 12 based on File2's range (6 to 12). We can see that the end of the current range varies as new start points are encountered, just like an expanding sweep line. When an end point is encountered (Line 11), its file name is matched with the first start point (first in Line 9) that joins the sweep line (Line 13), if they belong to the same file, this file's exclusive time is added to the output list (Line 14), and the sweep line is reset by selecting a top point from the heap as the first point of the new sweep line (Line 16), and setting its start_time as the current time suggested by $p.end$ (Line 17). In Fig. 3, File1 ends at 10; it is added to the output list as (File1, 0, 10). At this point, File2 is the only file in the

heap (File3 already ends at 8). File2's entry is selected from the heap and becomes the first of the sweep line. With this algorithm, (File2, 10, 12) and (File4, 16, 18) are added to the output list subsequently.

III. Evaluation

In this section, we evaluate IOMiner's support for various types of analysis. We ran a large number of analysis tasks, but because of the page limit we present only a few interesting results. We use IOMiner to perform simple analysis tasks that can be accomplished by grouping, sorting, filtering, and projection operations, and in-depth analysis that can be answered by grouping/binning with the user-defined functions. We also discuss our use of IOMiner to find root causes of an application's poor I/O performance.

A. Experimental Setup

We ran IOMiner on the Cray XC40 system Cori at the National Energy Research Scientific Computing Center (NERSC) to analyze the I/O performance logs collected on the same system. Cori consists of two partitions: one has ≈2,400 nodes with Intel Xeon "Haswell" processors, and another has ≈9,700 Intel Xeon Phi "Knights Landing" (KNL) processors. Cori has several different user-accessible file systems, including a disk-based Lustre system, an SSD-based burst buffer, and a disk-based GPFS file system.

Our analyses were performed on the logs produced by the application-level Darshan profiling tool, Slurm job scheduler, and storage server-side Lustre Monitoring Tool LMT. We analyzed the logs generated across the entire month of Jan. 2018. Overall, analysis on LMT shows an aggregate read/write traffic of 42 PB on the storage servers. The analysis on Slurm logs indicates an aggregate of 98 million core-hours from 3.15 million successfully completed jobs. Our analyzed Darshan logs cover 9.4% of the I/O traffic from LMT logs and 20.2% of CPU-hours from Slurm logs, accounting for 0.45 million of the total jobs considered.

B. Performance of IOMiner

In our first experiment, we demonstrate the execution time of IOMiner in retrieving results to answer "How many jobs use customized stripe configuration?" To answer this question, IOMiner selects all the jobs using Lustre and scans across all the files in each job to determine whether their *stripe size*, *stripe count*, and *stripe offset* are the same as the Lustre's default setting (1048576, 1, and -1, respectively, for the three parameters).

Figure 4 shows IOMiner's execution times with a varying executor count. We place two executors on each node to leverage its NUMA architecture. We observe that IOMiner performs the best with 32 executors, with 18× speedup over the single-executor case. We also note that increasing executors beyond 32 does not scale, mainly because the data size per executor becomes smaller and communication overhead between the driver and the executors becomes a dominant factor. We have also measured the performance of extracting data directly from Darshan's own log format using one process, answering this query involves opening, closing and scanning each log for the stripe information, which could not finish in our requested wall time limit (6 hours).

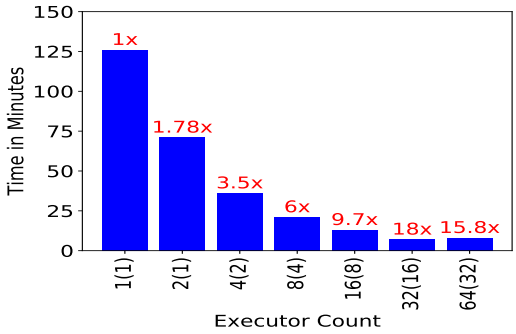

Figure 4. Performance of Spark-based IOMiner. m(n): m executors on n nodes.

C. First-Order statistics

The results suggest that only 0.08% of jobs adopt a customized stripe setting. This observation raises a concern for those common I/O workloads (i.e., N-1 and N-M in Section III-D) that involve multiple processes concurrently writing/reading the shared files. With a default *stripe count* of 1, all processes sharing the same file are bottlenecked by the bandwidth of a single OST. Despite this, other HPC centers have reported low rates of custom stripe settings [9].

We performed a set of first-order statistical analyses to provide examples of IOMiner's capabilities. We discuss new observations in I/O analysis using these analytics.

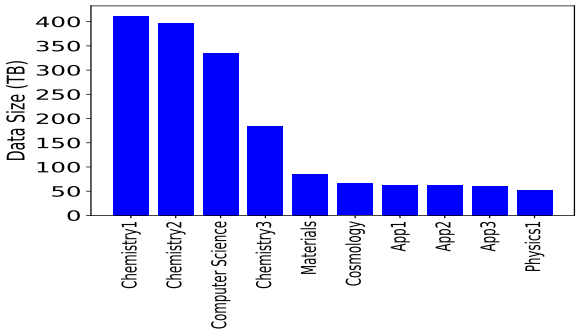

Figure 5. Top-ten read-intensive applications.

1) Top I/O-intensive applications: The top I/O-intensive applications are of particular interest to the I/O specialists in optimizing applications' I/O performance and to file system designers in dealing with peak application I/O workloads. We leverage IOMiner to select the top most read- and write-intensive applications by grouping these jobs based on executable names of applications and then calculating the

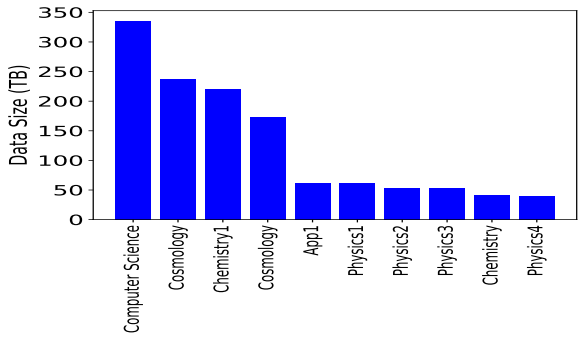

Figure 6. Top-ten write-intensive applications.

aggregate read/write sizes for all jobs in each application and ranking them by their aggregate bytes read/written. Figures 5 and 6 show the applications and their aggregate read/write sizes. We have anonymized the applications using their science area. Overall, these top ten I/O-intensive applications consume 72% and 76% of the entire read and write traffic captured by Darshan, respectively.

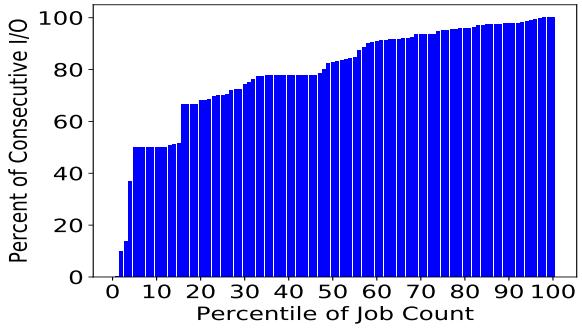

Figure 7. Distribution of sequential I/O percent among jobs.

2) Distribution of sequential I/O: Sequential I/O is a friendly I/O pattern for the disk-based file systems and the SSD-based burst buffers. In this pattern, an I/O request issued by a process immediately accesses the end byte of the previous I/O operation. I/O bandwidth can be enhanced by aggregating multiple consecutive writes/reads, which may be smaller than a file system page size, into fewer accesses. To analyze how many HPC applications can benefit from such optimization, we define the ratio of sequential I/O P_{seq} as Equation 3, where N_{seq} and N_{tot} refer to the number of sequential I/O requests and the total reads/writes count in each job. We then calculate the percentile of P_{seq} and plot its cumulative distribution, as shown in Figure 7. In this figure, each point on the x-axis represents the percentile of jobs whose P_{seq} is below its value on the y-axis. We can see that only close to 10% of jobs' P_{seq} are below 50%; in other words, 90% of jobs' P_{seq} are above 50%. In addition, we see that almost half of the jobs' P_{seq} are above 80%. This observation implies that sequential I/O is highly prevalent in the HPC I/O workloads and that further I/O optimizations such as I/O aggregation can improve I/O performance.

$$P_{seq} = N_{seq}/N_{tot} \tag{3}$$

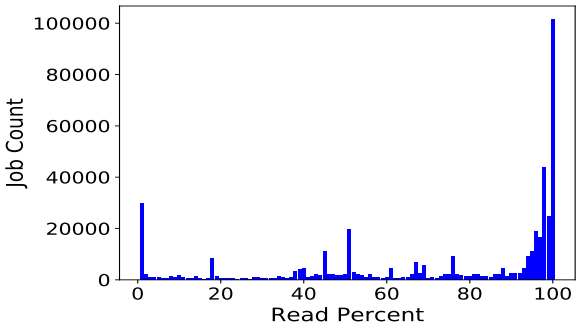

Figure 8. Distribution of read ratio among jobs.

3) Distribution of the read ratio: Checkpointing has been considered as a dominating I/O workload on HPC systems [10], [11], [12], [13]. For this reason, the majority of I/O optimization efforts focus on accelerating checkpointing, a workload featured by bursty writes. In contrast, read optimization has received relatively less attention. However, it remains unknown whether read optimization deserves more effort. To answer this question, we define the read ratio as Equation 4, where B_{read} and B_{write} refer to the total bytes read and written by each job, respectively. We then bin the jobs based on their P_{read} into 100 percentiles, and we calculate the job counts in each bin. As can be seen from Figure 8, although a substantial fraction of jobs is either write-only (7% with $P_{read} = 0$) or read-only (23% with $P_{read} = 100$), the majority of jobs are featured with a mixed workload. In addition, there is a burst of jobs with $P_{read} > 90\%$, accounting for 53% of the total job count. This analysis informs us that further studies and optimizations on read are worth the investment.

$$P_{read} = \frac{B_{read}}{B_{write} + B_{read}} \tag{4}$$

D. I/O pattern analysis

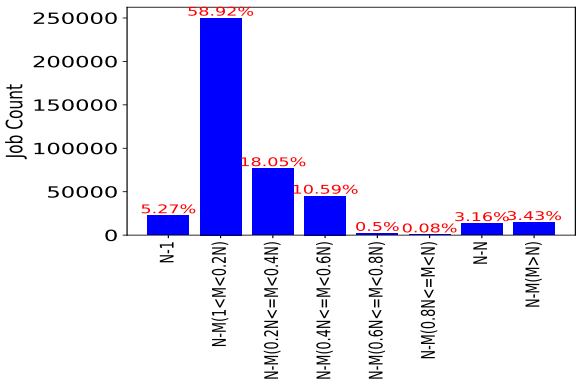

Figure 9. File-sharing patterns among different jobs.

N-N and N-1 are considered as two primary I/O patterns of HPC parallel applications [14]. In the N-N pattern, each process (typically, an MPI process) writes/reads a private file. In the N-1 pattern, all the processes concurrently access the same shared file. Many widely adopted I/O benchmarks have been designed to emulate these I/O patterns [15], [16]. However, the distribution of I/O pattern usage in parallel applications running on the production HPC systems is an open question. To answer this, we define the file-sharing ratio (R) in Equation 5, where P is the process count in a job and F is file count in the job. We use IOMiner to provide a histogram of the job counts based on R, as shown in Figure 9. Note that in the Figure, N-1 case includes the scenario when a list of shared files is accessed by a job, each read/written by all the processes within a distinct time step. We have also excluded the jobs using only one process, which belong to both N-1 and N-N patterns.

$$R = P/F \tag{5}$$

We can see that N-N and N-1 jobs constitute only a small percentage of all the jobs. In contrast, most jobs exhibit the N-M pattern, where either each file is shared by a subset of processes ($1 < M < N$) or one process works on multiple files ($M > N$). This observation suggests that existing benchmarks can be adapted to a wider spectrum of file-sharing patterns to more accurately capture the real I/O behavior of the application.

E. I/O middleware usage analysis

POSIX-IO and MPI-IO have been predominant I/O middleware used for performing parallel I/O. MPI-IO [17] has been developed for roughly two decades to optimize I/O in MPI applications. Its collective I/O optimization, where a small set of MPI processes act as aggregators for performing larger I/O requests to improve performance, is a key optimization method. High-level I/O libraries, such as HDF5 [18] and PnetCDF [19], [20], are built on top of MPI-IO to take advantage of various optimizations. While these libraries are efficient for enhancing applications' I/O performance, it remains unknown how many applications actually use them in production. In this study, we use IOMiner to bin the jobs based on process count, and we calculate the job count using POSIX I/O, MPI-IO in independent mode, and MPI-IO with collective optimizations.

We group four types of jobs based on the process count: MPI jobs using one process (denoted as One); using 2 to 1,024 processes (Small); using 1,025 to 8192 processes (Medium); and using more than 8,192 processes (Large). As shown in Figure 10, among the Darshan logs we evaluated, POSIX I/O is used by 98.8% and 99.6% by single-process (One) and Small jobs, respectively. Although there is a higher use of MPI-IO usage in Medium (22.2%) and Large (45.9%) jobs compared with Small jobs, POSIX-IO remains as the most-used I/O interface. Collective I/O optimizations are enabled in most of the MPI-IO jobs. This situation is probably because of the use of HDF5, the most common parallel I/O library used by the applications on Cori, where collective I/O is enabled. Overall, we can see that although MPI-IO has been a long-standing interface, most HPC users are still committed to POSIX I/O. As I/O specialists spend more effort on the future I/O stacks for exascale computing, one of the challenges is to persuade POSIX-IO users to use the new I/O techniques.

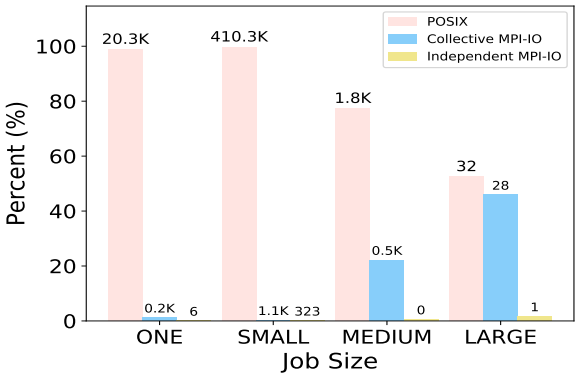

Figure 10. Usage of POSIX and MPI-IO among jobs with different sizes. P: process count; One: $P = 1$; Small:$1 < P \leq 1024$; Medium:$1024 < P \leq 8192$; Large: $P > 8192$.

F. Root Cause Analysis of the Low I/O bandwidth Jobs

While data-intensive applications running at large scale often can obtain good I/O performance, factors such as I/O pattern, number of I/O requests, and number of storage targets used can affect the sustained performance. To identify the root causes of applications' low I/O performance, we have used IOMiner in analyzing the logs of jobs using more than 1,000 processes, where each process was writing/reading at least 10 MB of data, and the aggregate sustained I/O bandwidth was lower than 1 GB/s. We then analyzed the root causes of these jobs' poor I/O bandwidth using the techniques described in Section II-D. The filtering condition returns records pertaining to 251 low-read-bandwidth jobs from 17 applications and to 724 low-write-bandwidth jobs from 16 applications. We show these applications and values of various factors contributing to the I/O performance as parallel coordinate plots [21] in Figures 11 and 12. In these figures, different colors represent different applications. For instance, in Figure 11, we can see that 140 out of 251 jobs are in blue, and they belong to application 10. Since the bandwidth of a job may also be limited by the use of a single node, and the node count information for each job is recorded in a Slurm scheduler log, we also extract this information by joining the Slurm table with the Darshan table (shown as Node#). We found that many jobs experience a contention level (Proc/OST) larger than 3 (e.g. 56 jobs for read, and 395 jobs for write). Further investigation of the I/O logs of these applications revealed their use of

default Lustre stripe setting of one OST, causing many processes to concurrently write/read a shared file using the OST, with a resulting bottleneck of the bandwidth on this OST. On the other hand, the impact of collective, small, nonconsecutive I/O and OST count is less perceivable since their values are randomly distributed across their x-axis. We have also analyzed the well-performing jobs whose bandwidth is beyond 20 GB/s (not shown due to the space limit) and were not able to discover a regular trend among those contributing factors either. These observations inform us that the root causes for jobs' low I/O performance can be a synergistic effect of multiple contributing factors and thus warrant further analysis of application I/O logs.

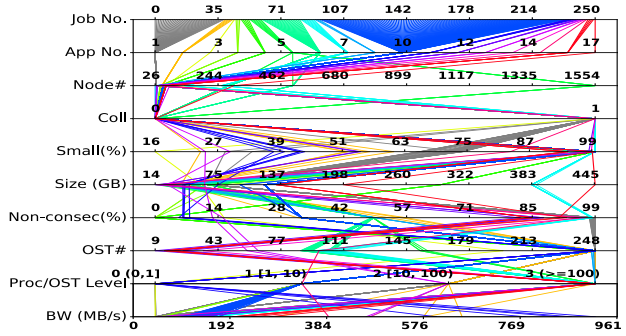

Figure 11. Impact of contributing factors to low-read-bandwidth jobs. Small, Nonconsec, Coll, OST#, and Proc/OST Level refer to the contributing factors defined in Section II-D, respectively. Node# refers to the number of nodes used by the job. Size refers to the aggregate read size for each job. BW refers to the aggregate bandwidth for each job. App No. refers to the numbering of applications. Job No. refers to the numbering of jobs. Multiple jobs can belong to the same application.

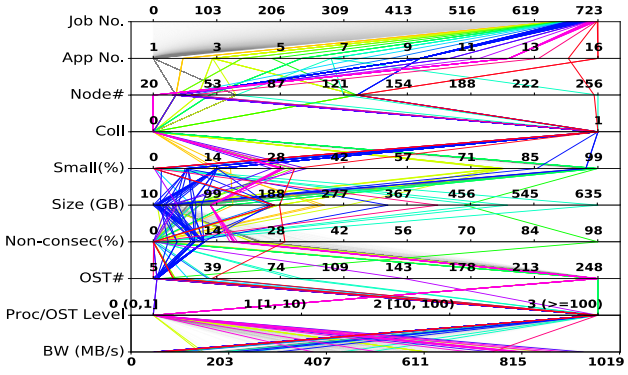

Figure 12. Impact of contributing factors to low-write-bandwidth jobs.

To further investigate the root causes for the individual applications, we select the low-bandwidth jobs that use all the system's OSTs (248), and we classify them based on their applications. Tables I and II list the contributing values for the low-read-bandwidth and low-write-bandwidth

Table I
CONTRIBUTING FACTOR VALUES FOR LOW-READ-BANDWIDTH APPLICATIONS WITH ALL OSTS UTILIZED.

App	Small(%)	Non-consec(%)	Coll(%)	Proc/OST
Meteorology1	53	53	0	25
Physics1	44	6	0	1
Climate1	71	98	0	100
Quantum1	99	0	0	1

applications, respectively. One can see that jobs belonging to the same applications (same color) generally share similar contributing factor values, so we present the contributing factors of one representative job for each application. The only exception is `Physics1` in Table II, where all these jobs share two different sets of contributing factor values (`Physics1.1` and `Physics1.2`).

In Table II, we find that the small-write-percentage of `Physics1.1`, `Physics1.2`, and `Chemistry1` all stay at high values ($> 50\%$). However, we cannot conclude that small writes are the culprit; their writes are mostly consecutive. These small and consecutive writes give the operating system ample opportunity to aggregate the small writes into larger ones. To more precisely find out the causes, we used a sweep-line algorithm (shown in Algorithm 2) to analyze the timing of writing individual files on the critical path. It turns out that the root reason for low performance of `Physics1.1` is that its I/O time is bottlenecked by all processes (1,024) writing to one large file on only one OST. The low performance of `Physics1.2` and that of `Chemistry1` is because the root process writes significantly more data than do the other MPI processes, dominating the write time on critical path. On the other hand, we have also observed that all these contributing factors stay low for `App1`. Using Algorithm 2, we find that the long I/O time of `App1` is because of files being written intermittently and their write times are not well overlapped with each other. Synchronization also may occur between writing different files, contributing to poor I/O performance.

In Table I, we see that the read bandwidths of `Meteorology`, `Climate1`, and `Quantum1` are impacted by one or multiple factors. For instance, we observe a large percentage of small and nonconsecutive I/O in `Meteorology1` and `Climate1` ($> 50\%$), which is the primary reason for the two applications' low read bandwidth. However, we also find that `Physics1` is an exception, where all the factors stay at low values. Using Algorithm 2, we observe that reading one file takes up 95% of the total read time on the critical path. These types of root causes could provide sufficient evidence to system administrators at supercomputing facilities for communicating with the users and application developers.

Table II
CONTRIBUTING FACTOR VALUES FOR LOW-WRITE-BANDWIDTH APPLICATIONS WITH ALL OSTS UTILIZED.

App	Small(%)	Non-consec(%)	Coll(%)	Proc/OST
App1	29	17	0	1
Physics1.1	98	0	0	848
Physics1.2	98	0	0	1
Chemistry1	76	2	0	1

G. Discussion

Our analysis of the I/O logs on the Cori system at NERSC using IOMiner provides multiple insights. First, read/write traffic on HPC is dominated by only a few applications (Section III-C1), and most applications predominately use sequential I/O (Section III-C2). File system and middleware developers can pay special attention to these top data-intensive applications' I/O demands and be open to the techniques that boost the sequential I/O bandwidth, such as I/O aggregation. Second, we have identified several directions that are worth further investigation. For instance, read workloads are as common as write workloads (Section III-C3); and optimizations on applications' read performance, such as data reorganization and caching, can bring significant benefits. POSIX I/O is still the most widely used I/O middleware (Section III-E); hence next-generation I/O stack designs must take into consideration whether POSIX consistency is a real requirement. Since N-M pattern is the most common file-sharing pattern on HPC systems (Section III-D), benchmarks have to be developed to represent this pattern. Furthermore, our root-cause analysis suggests that the reasons for applications' poor I/O performance can be diverse, either as a result of the synergistic effects of the contributing factors discussed in Section II-D or other factors beyond this scope. With the help of the sweep-line algorithm, HPC users can locate one or multiple bottleneck files on the critical path and find out the root causes for poor-I/O-performance by looking only at these files' statistics. IOMiner provides an extensible and flexible framework to filter a massive number of logs as well as to sift through individual application traces.

IV. RELATED WORK

Tracing I/O activity and analyzing the traces has been one of the most prominent techniques for characterizing I/O performance. A limitation of the existing I/O tracing and their performance tools is that the analysis of the traces and statistics and the identification of any performance problems have to be performed manually. Such efforts require expertise in parallel I/O systems. Luu et al. [5] have analyzed a large number of Darshan logs collected at multiple supercomputing facilities and summarized that most applications obtain significantly lower I/O performance than the peak capability, and that several applications do not even use parallel I/O libraries. Besides these work on application-level analysis [22], [23], [24], there are numerous work on file system level characterization [25], [26], [27], [28], [29], [30]. These efforts were generally performed manually, and focus on a single level I/O traces, conducting a similar analysis would require significant effort.

pytokio [31] is software facilitating holistic characterization and analysis of multi-level I/O traces. It defines abstract connectors to various monitoring tools, and allows users to extract data from these sources using these connectors. Though both pytokio and IOMiner support analytics on multi-level I/O traces, IOMiner differs in that it is designed for large-scale parallel analytics. GUIDE [32] is another framework for analyzing multi-level I/O traces, different from GUIDE, IOMiner focuses more on applications' I/O behavior and their performance impact, which is useful for both the facility operators and the application developers, while GUIDE targets to deliver system-level statistics to the facility operators, such as the file system workload, the network traffic, etc.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we present a holistic I/O analysis framework called IOMiner. IOMiner provides unified interfaces and read-friendly storage schema for users to perform an integrated analysis on different types of instrumentation data, such as application and file-system-level logs and job scheduler logs. The whole framework is built on top of Spark and is optimized for parallel queries of HPC storage. Furthermore, IOMiner allows users to conveniently identify the root causes for an application's poor I/O performance based on a sweep-line algorithm. Our analysis provides several novel insights for HPC users and demonstrates that the root causes for an application's low I/O performance can be diverse. As future work, we will extend IOMiner with the ability to intelligently learn the new contributing factors for low I/O performance, identify the list of contributing factors that synergistically account for the low performance, and perform system-wide application I/O diagnostics. We will also provide support for more types of I/O instrumentation data under our framework. For instance, ggiostat [33] is a monitoring tool for the GPFS [34] file system. It could be integrated into IOMiner framework in the same way as LMT.

ACKNOWLEDGMENT

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under contract numbers DE-AC02-05CH11231 and DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under contract number DE-AC02-05CH11231.

REFERENCES

[1] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access through Continuous Characterization," *TOS*, vol. 7, no. 3, p. 8, 2011.

[2] C. Xu, S. Snyder, O. Kulkarni, V. Venkatesan, P. Carns, S. Byna, R. Sisneros, and K. Chadalavada, "DXT: Darshan eXtended Tracing," *CUG*, 2017.

[3] *Lustre Monitoring Tool*, 2016, https://github.com/LLNL/lmt.

[4] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple Linux Utility for Resource Management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[5] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A Multiplatform Study of I/O Behavior on Petascale Supercomputers," in *HPDC*. ACM, 2015, pp. 33–44.

[6] D. Souvaine, "Line Segment Intersection Using A Sweep Line Algorithm," *Tufts University*, 2005.

[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *NSDI*. USENIX Association, 2012, pp. 2–2.

[8] Y. Wang, R. Goldstone, W. Yu, and T. Wang, "Characterization and Optimization of Memory-Resident MapReduce on HPC Systems," in *IPDPS*. IEEE, 2014, pp. 799–808.

[9] F. Wang, H. Sim, C. Harr, and S. Oral, "Diving into petascale production file systems through large scale profiling and analysis," in *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems - PDSW-DISCS '17*. New York, New York, USA: ACM Press, 2017, pp. 37–42. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3149393.3149399

[10] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System," in *SC*. IEEE Computer Society, 2010, pp. 1–11.

[11] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. Panda, "A 1 PB/s File System to Checkpoint Three Million MPI Tasks," in *HPDC*. ACM, 2013, pp. 143–154.

[12] H. Jin, T. Ke, Y. Chen, and X.-H. Sun, "Checkpointing Orchestration: Toward A Scalable HPC Fault-Tolerant Environment," in *CCGRID*. IEEE, 2012, pp. 276–283.

[13] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An Ephemeral Burst-Buffer File System for Scientific Applications," in *SC*. IEEE Press, 2016, p. 69.

[14] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A Checkpoint Filesystem for Parallel Applications," in *SC*. ACM, 2009, p. 21.

[15] H. Shan and J. Shalf, "Using IOR to Analyze the I/O Performance for HPC Platforms," *CUG*, 2007.

[16] F. Shorter, "Design and Analysis of A Performance Evaluation Standard for Parallel File Systems," Ph.D. dissertation, Clemson University, 2003.

[17] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*. IEEE, 1999, pp. 182–189.

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

[19] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel NetCDF: A High-Performance Scientific I/O Interface," in *SC*. IEEE, 2003, pp. 39–39.

[20] T. Wang, K. Vasko, Z. Liu, H. Chen, and W. Yu, "Enhance Parallel Input/Output with Cross-Bundle Aggregation," *IJHPCA*, vol. 30, no. 2, pp. 241–256, 2016.

[21] A. Inselberg, *Parallel Coordinates: Visual Multidimensional Geometry and Its Applications*. Berlin, Heidelberg: Springer-Verlag, 2009.

[22] Z. Liu, B. Wang, T. Wang, Y. Tian, C. Xu, Y. Wang, W. Yu, C. A. Cruz, S. Zhou, T. Clune *et al.*, "Profiling and Improving I/O Performance of A Large-Scale Climate Scientific Application," in *ICCCN*. IEEE, 2013, pp. 1–7.

[23] D. Devendran, S. Byna, B. Dong, B. Van Straalen, H. Johansen, N. Keen, and N. F. Samatova, "Collective I/O Optimizations for Adaptive Mesh Refinement Data Writes on Lustre File System," *CUG*, 2016.

[24] J. Li, W.-k. Liao, A. Choudhary, and V. Taylor, "I/O Analysis and Optimization for an AMR Cosmology Application," in *CLUSTER*. IEEE, 2002, pp. 119–126.

[25] S.-H. Lim, H. Sim, R. Gunasekaran, and S. S. Vazhkudai, "Scientific User Behavior and Data-Sharing Trends in a Petascale File System," in *SC*. ACM, 2017, p. 46.

[26] R. Gunasekaran, S. Oral, J. Hill, R. Miller, F. Wang, and D. Leverman, "Comparative I/O Workload Characterization of Two Leadership Class Storage Clusters," in *PDSW*. ACM, 2015, pp. 31–36.

[27] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems," in *IPDPS*. IEEE, 2016, pp. 750–759.

[28] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination," in *IPDPS*. IEEE, 2014, pp. 155–164.

[29] S. Byna, A. Uselton, Prabhat, D. Knaak, and Y. He, "Trillion Particles, 120,000 Cores, and 350 TBs: Lessons Learned from a Hero I/O Run on Hopper," https://www.nersc.gov/assets/Hopper-Hero-IO-run.pdf, 2013.

[30] L. Wan, M. Wolf, F. Wang, J. Y. Choi, G. Ostrouchov, and S. Klasky, "Comprehensive Measurement and Analysis of the User-Perceived I/O Performance in a Production Leadership-Class Storage System," in *ICDCS*. IEEE, 2017, pp. 1022–1031.

[31] G. K. Lockwood, N. J. Wright, S. Snyder, and P. Carns, "TOKIO on ClusterStor: Connecting Standard Tools to Enable Holistic I/O Performance Analysis," in *CUG*, 2018.

[32] S. S. Vazhkudai, R. Miller, D. Tiwari, C. Zimmer, F. Wang, S. Oral, R. Gunasekaran, and D. Steinert, "GUIDE: A Scalable Information Directory Service to Collect, Federate, and Analyze Logs for Operational Insights into a Leadership HPC Facility," in *SC*. ACM, 2017, p. 45.

[33] G. K. Lockwood, W. Yoo, S. Byna, N. J. Wright, S. Snyder, K. Harms, Z. Nault, and P. Carns, "UMAMI: A Recipe for Generating Meaningful Metrics through Holistic I/O Performance Analysis," in *PDSW*. ACM, 2017, pp. 55–60.

[34] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, 2002, pp. 231–244.