A Multiplatform Study of I/O Behavior on Petascale Supercomputers

H. Luu, M. Winslett, W. Gropp
Univ. of Illinois at Urbana-Champaign

R. Ross, P. Carns, K. Harms
Argonne National Laboratory

Prabhat, S. Byna, Y. Yao
Lawrence Berkeley Nat'l Laboratory

ABSTRACT

We examine the I/O behavior of thousands of supercomputing applications "in the wild," by analyzing the Darshan logs of over a million jobs representing a *combined* total of six years of I/O behavior across three leading high-performance computing platforms. We mined these logs to analyze the I/O behavior of applications across all their runs on a platform; the evolution of an application's I/O behavior across time, and across platforms; and the I/O behavior of a platform's entire workload. Our analysis techniques can help developers and platform owners improve I/O performance and I/O system utilization, by quickly identifying underperforming applications and offering early intervention to save system resources. We summarize our observations regarding how jobs perform I/O and the throughput they attain in practice.

Categories and Subject Descriptors

C.4 [**Performance of systems**]: *Performance attributes, Measurement techniques*

Keywords

Input/Output; Performance Analysis; HPC; Parallel I/O

1. INTRODUCTION

The 2014 TOP500 list includes over 40 deployed petascale systems, and the high-performance computing (HPC) community is working toward developing the first exaflop system by 2023. Scientific applications on such large-scale computers often read and write a lot of data. For example, an earth science code on an IBM Blue Gene/P system at Argonne National Laboratory read ~3.5 PB during two months in 2010 [1]. With such rapid growth in computing power and data intensity, I/O remains a challenging factor in determining the overall performance of HPC codes.

Analyzing I/O behavior of applications (apps) can help improve their performance and increase the utilization of supercomputing systems. By analyzing the runtime behavior of an individual job, we can identify its I/O bottlenecks and potential implementation inefficiencies and suggest improvements to its owner and users. By analyzing the I/O behavior of an app (i.e., the set of all its jobs), we can identify patterns in its behavior. By analyzing the I/O behavior of the workload of a *platform* (i.e., a supercomputer instance), we can give the platform owners insights into the usage of their storage systems and identify apps that consume I/O resources inefficiently, so that improvements to these apps may free up resources for other apps. By analyzing the changes in I/O behavior when apps migrate to similar or radically different platforms, we can help scientists avoid unexpected performance degradation. I/O behavior analysis can even show us how the behavior of individual apps evolves over time. To accomplish all these purposes, we need a systematic approach to app-specific, platform-wide, **and** cross-platform analysis of I/O behavior.

In this paper, we show how automated collection and analysis of I/O logs across multiple platforms can help accomplish these purposes. We used Darshan [1], a lightweight instrumentation tool, to capture application-level I/O behavior at production scale. Because Darshan's overhead is low, a number of platform owners[1] have deployed it as the default option for all apps, thus enabling workload-wide and cross-platform analysis.

This paper presents insights we gleaned by analyzing Darshan logs from three large-scale supercomputers: Intrepid and Mira at the Argonne Leadership Computing Facility (ALCF) and Edison at the National Energy Research Scientific Computing Center (NERSC). The logs span a substantial period of time—4 years on Intrepid, 18 months on Mira and 9 months on Edison—and capture the I/O behavior "in the wild" of about 1M jobs, representing thousands of apps and roughly a third of the workload on these platforms. This is the first study that has been able to compare and contrast the I/O behavior and evolution of many different apps at production scale across platforms.

Our contributions fall into two categories:

- The logs provide a broad portrait of the state of HPC I/O usage on three modern platforms. For example, ***among Darshan-instrumented jobs***:
  - Every widely used I/O paradigm (file per process, global shared file, or subsetting I/O) is represented in the set of best-performing **and** worst-performing apps, in terms of aggregate I/O throughput. Thus, use of a particular paradigm does not in itself guarantee good or bad performance.
  - Roughly a third of jobs have aggregate average I/O throughput no more than that of a single contemporary USB flash memory thumb drive (~256 MB/s [2]). Three-quarters of apps never exceed the throughput of four thumb drives in *any* of their jobs. Over a third of jobs spend more time in I/O metadata functions than in transfer of actual data.

HPDC'15, June 15 - 19, 2015, Portland, OR, USA Copyright 2015 ACM
978-1-4503-3550-8/15/06...$15.00
http://dx.doi.org/10.1145/2749246.2749269

[1] In this paper, a *job* or *run* is a particular execution of an app. Unless otherwise noted, we consider two jobs to belong to the same app if and only if their executables have the same name. Someone who submits a job is a *user*; users may have to configure an app before they submit a job. Someone responsible for developing the source code of an app is its *owner*, or rather one of its owners. A widely used app may have a small set of owners and a much bigger set of users. A *platform* is a particular installation of a supercomputer. Someone responsible for configuring or administering a platform or for helping its users is an *owner* of that platform.

- Roughly half of apps have low throughput because none of their jobs access more than 1 GB of data, so that file startup costs cannot be amortized across much data transfer; or because they rely on text files instead of binary files. Even on the most data-intensive platform we studied, half of apps wrote less than 10 GB of data in 99% or more of their jobs. On one platform we studied, roughly one-fifth of apps relied exclusively on text files, which almost certainly guarantee poor performance at scale.
- Three-quarters of jobs use only POSIX to perform I/O. This does not condemn a job to poor I/O throughput, but it does suggest a need to investigate why higher-level parallel I/O libraries are not more widely used.

- We discuss ways to address these problems, including a simple and effective analysis and visualization procedure for quickly identifying apps' I/O bottlenecks; criteria for system owners to identify potentially underperforming apps; and an I/O boot camp for users/owners of underperforming apps. The resulting performance improvements could raise the level of satisfaction of users, app owners, and platform owners. Two subtle points:
  - The I/O performance of an app may satisfy its owners but not necessarily the platform owners, and vice versa. Thus analysis of I/O logs must address the needs of both populations.
  - 90% of a platform's I/O usage comes from less than 10% of its apps, but some of these apps do not have many large jobs. The greatest potential resource savings for platform owners lies in identifying and correcting an app's I/O issues *before* it becomes a top consumer of I/O time. Automated analysis can be particularly helpful here, as smaller jobs are less likely to attract expert human scrutiny.

The remainder of this paper is organized as follows. Section 2 provides background on Darshan and summarizes related work. Section 3 describes the target platforms and collected data. Section 4 presents a three-dimensional analysis of the collected logs: app-specific, platform-wide, and cross-platform. Section 5 summarizes our findings and outlines future work.

2. BACKGROUND AND RELATED WORK

For over 20 years, researchers have sought to understand HPC I/O workloads. As the size, composition, and complexity of platforms and their workloads grow continuously, the topic must be revisited in each generation of platforms (see, e.g., [3][4] from the 1990s, [5][6] from the 2000s, and [7][8] from the 2010s). Many workload studies (e.g., [5][6][8], among more recent works) use popular scientific codes such as FLASH [11], GCRM [12], Nek5000 [13], CESM [14], and their associated benchmarks as representative of the entire I/O workload. Such benchmarks are widely used to tune and refine I/O libraries and storage systems. Since these apps are widely used in their fields, any improvements made to them can benefit many users. As important as they are, however, these well-studied apps and benchmarks are not necessarily representative of the long tail of apps that constitute the majority of submitted jobs. By considering a platform's entire workload, we can gain additional insights into its I/O system usage. By considering multiple platforms and many apps, we can gain general insights into I/O performance portability.

I/O tracing is very helpful in capturing details of individual I/O functions and allowing in-depth analysis of application performance. Researchers have created many tools for generating app I/O traces, such as RIOT I/O [15], ScalaIOTrace [16], //TRACE [17], IPM [18], LANL-Trace [19], TraceFS [20], and Recorder [21]. After the traces have been generated, they can be used for app debugging, performance tuning, creating benchmarks, system analysis, or cross-platform studies. For example, the RIOT I/O tracing toolkit has been used to assess the performance of three I/O benchmarks on three platforms with GPFS and Lustre file systems. ScalaIOTrace, //TRACE and Recorder traces can be replayed to create app-specific benchmarks. I/O tracing provides very detailed information about app executions, which can be extremely useful in improving I/O performance. Such I/O tracing tools are ideal for investigating individual runs in full detail, but are too expensive to be used to find broader patterns at the scale of thousands of jobs and apps.

Kim et al. [7] characterized platform workloads by instrumenting the storage system. This approach does not provide app-specific information for analysis. In this paper, we rely on data captured for a general production workload, which can be used to characterize I/O behavior at both the app and workload levels.

Darshan [1] instruments I/O functions at multiple levels, primarily MPI-IO and POSIX I/O. Darshan collects about 30 pieces of summary data for each job, as well as 162 additional parameters for each file opened by a process of the job. Example job-level data include the numbers of processes, files accessed, and bytes read/written; aggregate I/O throughput; and total run time and I/O time [9]. After a run, users can employ Darshan's tools to parse their job's logs and summarize its I/O behavior.

Darshan's minimal collection of data (1-2% overhead, depending on the app [10][23]) allows it to be enabled for all jobs by default. This allows us to observe a platform at workload scale and to identify its jobs and apps that can most benefit from follow-up analyses with I/O tracing and other performance analysis tools. As Darshan captures all runs of the apps it observes, we can see the patterns of I/O behavior at scale and across platforms, rather than only for selected jobs.

Darshan logs have already been used for system-wide analysis. Carns et al. used two months of Darshan logs on Intrepid [1] and four months on Hopper [10] to explore how such logs can be used to improve storage system utilization and identify candidate apps for additional I/O tuning. We extend this approach to cover three platforms over a much longer period of time. To the best of our knowledge, this is the first study that compares and contrasts application I/O behavior across platforms at full scale.

3. TARGET PLATFORMS AND LOGS

Darshan is deployed and enabled by default for all users of ALCF and NERSC platforms, and Edison users automatically see Darshan's I/O summary report on a web page for their completed job. But Darshan does not see every job running on a platform. Apps are not logged if they do not call `MPI_Init()` and `MPI_Finalize()`, use nondefault build scripts, or run legacy executables that are not already linked to Darshan. Further, an issue in the F90 MPI wrapper on Mira prevents Darshan from observing F90 codes (a fix has been requested from IBM). Users can also choose to disable Darshan but do not normally do so.

On average, Darshan logs on Intrepid, Mira and Edison cover roughly *a third of jobs*. In the remainder of this paper we consider only those jobs and apps observed by Darshan, and we use the term *workload* to refer to the platform workload as observed by Darshan. We do not know whether Darshan's observations are typical of the I/O behavior of the unobserved part of the

workload; but the observed fraction of the workload is large enough to interest platform owners in its own right.

Table 1 describes Intrepid, Mira, Edison, and their Darshan logs. Intrepid is an IBM Blue Gene/P computer at ALCF with 40,960 quad-core nodes, 557 TFlops peak performance, and 88 GB/s peak I/O throughput to its GPFS file system. Each set of 64 compute nodes has one of 640 dedicated I/O forwarding nodes (IONs). From Jan. 2010 to Dec. 2013, Darshan captured 239K jobs representing over 1K apps, 1405M core-hours, and up to 163K processes and moving as much as 218TB of data in one job.

Table 1. Target platforms and their Darshan logs

Platform	**Intrepid**	**Mira**	**Edison**
Architecture	BG/P	BG/Q	Cray XC30
Peak Flops	0.557 PF	10 PF	2.57 PF
Memory	80 TB	768 TB	357 TB
Cores per node	4	16	24
# of cores	160 K	768 K	130 K
Storage	6 PB	24 PB	7.6 PB
Peak I/O	88 GB/s	240 GB/s	168 GB/s
File system	GPFS	GPFS	Lustre
Period logged	Jan'10 – Dec'13	Apr'13 – Oct'14	Jan'14 – Sep'14
Jobs logged	239,304	137,311	703,647

Intrepid's successor at ALCF is Mira, an IBM Blue Gene/Q running GPFS. Mira has 48K 16-core nodes, a peak computing performance 20x faster than Intrepid, and peak I/O throughput 3x faster than Intrepid. Mira has 384 IONs, each serving 128 compute nodes. Mira entered production mode in April 2013, with Darshan enabled. The 137K jobs Darshan observed there used 1456M core-hours and up to 1.04M processes and 570 TB per job.

Edison is the newest supercomputer at NERSC, a Cray XC-30 of size and performance roughly halfway between Intrepid and Mira. Edison has 5,576 24-core nodes and a peak I/O bandwidth of 168 GB/s to its Lustre file system. Edison's cores are several times more powerful than Mira's, making up for their smaller number. Darshan observed 703K jobs consuming 75M core-hours, using up to 131K processes and moving up to 426 TB of data in one job.

Figure 1 shows that average Darshan coverage is 20% to 40% (Edison graph omitted here). For these three platforms, Figure 2 compares the number of processes per job and the bytes each job read or wrote, showing quartiles and outliers in log scale. On all platforms, some jobs run at full system scale and/or transfer over 100 TB. However, most jobs transfer relatively little data and use few processes compared with the available number of cores. On Edison, 75% of jobs use under 100 processes and/or transfer no more than 3 GB of data. On Intrepid and Mira, 50% of jobs transfer less than 4 GB and/or use no more than 2K processes. Figure 3 shows that few apps ever use more than 4K processes or transfer more than a few gigabytes of data.

We imported Darshan's log files into a MySQL database and used SQL scripts to analyze the data. Script details are important for ensuring meaningful and reproducible results on other platforms, but due to space constraints, we only discuss the critical issue of computing the aggregate I/O throughput of a job. For each process of the job, we consider the total time spent in Darshan-tracked POSIX IO or MPI-IO data and metadata function calls for all the files the process opened. We set the job's I/O time to be the largest I/O time among all its processes. We computed the job's (aggregate) I/O throughput as its total bytes moved in Darshan-tracked POSIX IO or MPI-IO calls, divided by its I/O time.

App-level I/O throughput could be computed in other ways, e.g., sum/median/average across processes, but we find the slowest process's viewpoint best for comparing throughput across many jobs/apps. Usually computation does not resume until the slowest process has finished its I/O, due to an explicit barrier or the need to exchange data with neighbors. Thus *from the app's point of view,* our formula approximates its I/O throughput, and avoids misleading statistics when I/O loads are skewed across processes.

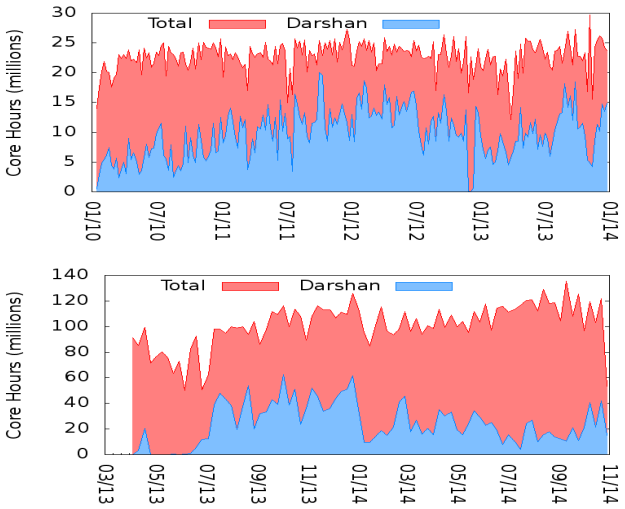

Figure 1: Darshan coverage in core hours on Intrepid (top) and Mira (bottom).

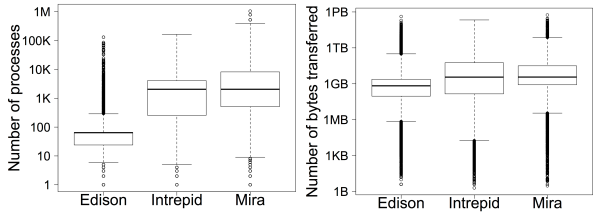

Figure 2: Cross-platform comparison of each *job's* number of processes (left) and number of bytes read/written (right).

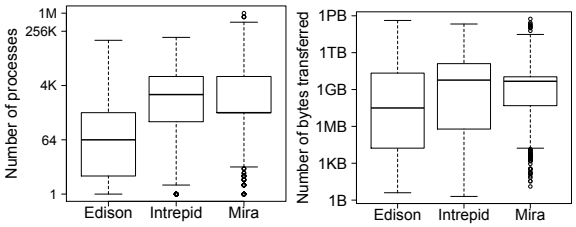

Figure 3: Cross-platform comparison of each *app's* maximum number of processes and maximum bytes read/written.

Darshan records the number of processes that a job runs on, but not the number of cores or nodes. The default Darshan configuration tracks most of the key POSIX IO or MPI-IO functions; condensing the wrapped functions' names, they are [l][f]*seek*[64], [ncmpi_][H5f]*creat*[e][64], [aio_][p][f]*read*[v][64], *lio_listio*[64], [aio_][p][f]*write*[v][64], [ncmpi_][[H5]f]*open*[64], *mmap*[64], *aio_return*[64], __[l][f]*xstat*[64], f[data]*sync*, and [ncmpi_][[H5]f]*close*. Darshan's default configuration does not track character-oriented functions such as *getc* and *putc* and their higher-level analogs *scanf* and *printf*, all intended for text data transfer. (Such functions may choose to call *read* or *write* for data access, but POSIX IO does not require them to do so, and we have not observed them doing so on our platforms.) The Darshan

developers did this to reduce overhead, assuming users would not spend much time performing character-oriented I/O.

4. ANALYSIS OF DARSHAN LOGS

4.1 Application-specific analysis

In this section, we present an analysis and visualization procedure that app users and owners can use to identify I/O bottlenecks and inefficiencies across all runs of their apps. Platform owners can use the same techniques to examine the apps that are their top users of I/O time (as identified by another set of scripts we wrote). The analysis consists of the following steps.

STEP 1. Identify where the job/app spends most of its I/O time, out of four possibilities:

a) **Global metadata**. All metadata functions for *global* files (i.e., files accessed by all processes), such as file open, close, stat, and seek functions.

b) **Nonglobal metadata.** Metadata functions for files that are not global (i.e., files accessed by a *proper subset* of the job's processes). These files may be *local*, that is, accessed by a single process; or *subset*, i.e., accessed by multiple processes, such as under a subsetting I/O paradigm.

c) **Global data I/O.** Data transfer functions for global files. These include the read, write, and sync functions.

d) **Non-global data I/O.** Data transfer for nonglobal files.

STEP 2. Identify which file(s) consume most of that time. We categorize the files along three dimensions: global, local or subset; MPI or POSIX; read-only, write-only, or read/write.

STEP 3. Examine Darshan's performance data for those files.

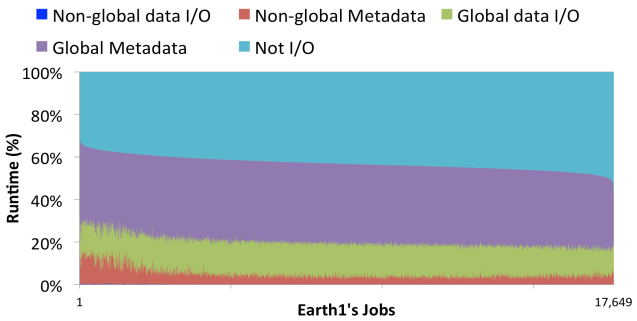

Figure 4: Breakdown of total run time for each Earth1 job.

As a case study, consider the app that consumed the most I/O time on Mira, an Earth science code we'll call "Earth1". Earth1 ran ~18K times in 4400 wall-clock hours and 36M core-hours. With Earth1's jobs ordered by their percentage of run time that is not I/O time (light blue), Figure 4 divides each job's remaining run time into the four categories in Step 1. Earth1 spent over half its time in I/O, most of which was for global file metadata.

To begin Step 2, we examined a randomly-selected Earth1 run. This job had 35 global shared files, including 24 using MPI for write-only files, 5 using POSIX for read-only files, and 6 using POSIX for write-only files. The total I/O time of the job was ~700 seconds, of which 567 seconds were spent on 6 POSIX write-only global files. Returning to the set of all Earth1 jobs, Figure 5 shows how Earth1's I/O time relates to the number of POSIX write-only global files its jobs use, as computed by our scripts. Global data I/O time increases gracefully with the number of files, while global metadata time increases much faster – even though graphs not included here show that the amount of global data transferred differs *by a factor of 3* across runs with the same number of POSIX write-only global files. In other words, I/O throughput tracks the changes in file count. This result indicates that the app owner should take a closer look at those files.

In Step 3, an I/O expert would quickly notice that according to the per-file Darshan data, each process writes the POSIX global files in relatively small pieces (<256 KB) that do not align with file block boundaries, making I/O costs high. Common issues of this nature could be included in a checklist for users or automatically recognized.

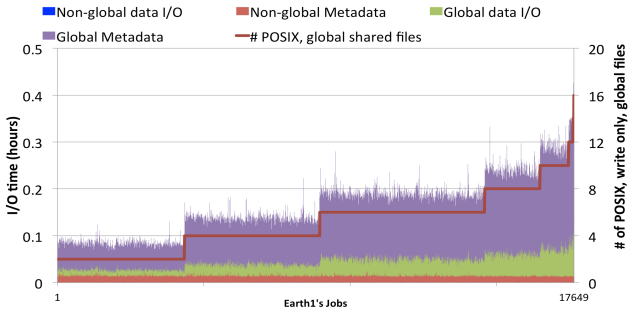

Figure 5: Earth1's I/O time and number of POSIX write-only global files (red line).

Job- and app-specific analysis can be done immediately after a run or a series of runs to help the app owner or user quickly locate an I/O bottleneck, avoiding a long-lasting inefficient implementation. Darshan's data is relatively high level, so it can give owners/users an idea about where their I/O problems may lie; owners/users may want to follow up with a tracing or debugging tool.

4.2 Platform-wide analysis

An app's inefficient use of shared system resources may impact other apps' ability to perform useful work. Platform owners can use platform-wide analyses to assess job performance, identify large underperforming apps, and offer early intervention to save system resources. In this section, we assess the performance of I/O workloads on Edison, Intrepid, and Mira and propose criteria for platform owners to quickly identify underperforming apps that consume lots of system resources.

4.2.1 Very low I/O performance is the norm for most apps on these platforms.

Even though these platforms' file systems have a peak throughput of hundreds of GB/s, few apps experience high I/O throughput.

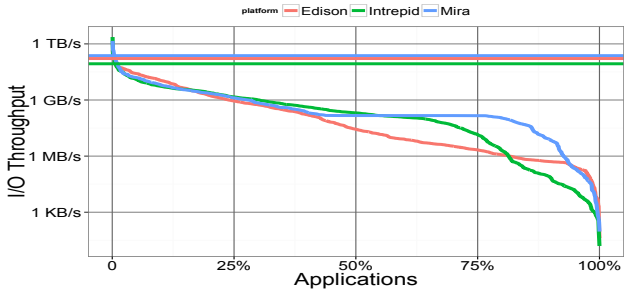

Figure 6: Maximum I/O throughput of each app across all its jobs on a platform, and platform peak I/O throughput.

For each app and platform, Figure 6 shows the *maximum* aggregate I/O throughput observed by Darshan, among all of the app's jobs on that platform. Horizontal lines show the platform's peak I/O bandwidth. (Apps exceed the platform peak when their data fits in the file system cache and reads/writes do not have to access the disk before I/O functions return.) Aggregate throughput

for three-quarters of apps never exceeds 1 GB/s, roughly 1% of average peak platform bandwidth. As noted earlier, most apps are relatively small; and no one should expect a job running on a few nodes to approach peak platform I/O bandwidth. For example, the Mira owners told us that a 1K-node job cannot expect more than ~20 GB/s I/O throughput, less than 10% of the platform peak. Looking at the situation another way, however, three-quarters of apps never exceed the aggregate throughput of four modern USB thumb drives (writes average 239 MB/s and reads average 265 MB/s on the 64 GB Lexar P10 USB3.0 [2]).

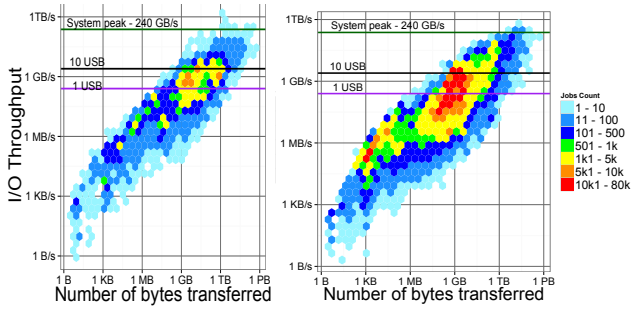

Figure 7: Number of jobs with a given I/O throughput and total number of bytes, on Mira (l) and Edison (r).

In Figure 7, each tile represents one or more *jobs*, with the tile color indicating the number of jobs. The figure shows that on Mira and Edison, a job's I/O throughput increases roughly linearly with its data size. Jobs that write very little data will not have high I/O throughput, because the fixed costs for accessing a file cannot be amortized across significant data transfer. Still, a third of jobs never reach the I/O throughput of a single modern thumb drive, and the vast majority of jobs never exceed the I/O bandwidth of 10 modern thumb drives. Intrepid (not shown) is similar.

Each vertical bar in Figure 8 represents all the jobs of one *app* on a platform. A bar's color indicates the total bytes accessed by its jobs. For example, a half-red, half-orange bar means that half the app's jobs accessed over 100 GB, and the other half accessed 10-100 GB (with perhaps a few smaller jobs not visible without magnification). Maximum and average I/O throughput for each app are indicated by squares and crosses, respectively, using the log-scale right-hand axis. The apps are sorted in decreasing order of importance for the storage system, as measured by the *total bytes transferred* across all the jobs of the app. Note that roughly half of apps do not transfer more than 1 GB of data in their jobs.

Darshan does not track text-oriented I/O functions, so apps that rely entirely on text files will register as having made metadata calls but transferring zero bytes, even if they access a lot of data and therefore are important to the storage system. Along with the apps that perform no I/O (e.g., a hello-world test), these text-only apps can be found at the far right-hand side of each graph, where there is a visible knee in the cloud of throughput dots. As the results indicate, 105 out of 1507, 201 out of 1032, and 42 out of 1183 apps open files but perform no binary I/O in *any* of their jobs on Intrepid, Mira, and Edison, respectively. Some of these apps are small by any measure, but others are not. For example, a third of the Mira text-only apps had an average job size of at least 1K processes, and a quarter of them averaged 16K or more processes per run. Some apps that heavily rely on text files also access binary files, so the counts listed above understate the extent of the usage of text files. Since we do not know how many bytes of text an app accesses, Figure 8 also understates the importance and impact on the storage system of text-based I/O. Since text-based I/O generally does not scale up well, we conclude that **text-based I/O is a more widespread practice than previously observed** and deserves further investigation.

I/O throughput for small jobs does not matter, in the sense that users and owners will be happy when a job's I/O time is only a second or so. But small jobs may be test runs for large jobs, such as the many Mira jobs in Figure 7 that transfer a terabyte of data and spend 10–20 minutes in I/O. Thus, small jobs may allow us to identify poor I/O practices before significant amounts of platform and user time have been wasted. Further, an app consisting entirely of relatively small jobs can still be a top user of I/O time on a platform. We consider these two points in the following discussions, which focus on apps that are heavy users of I/O time.

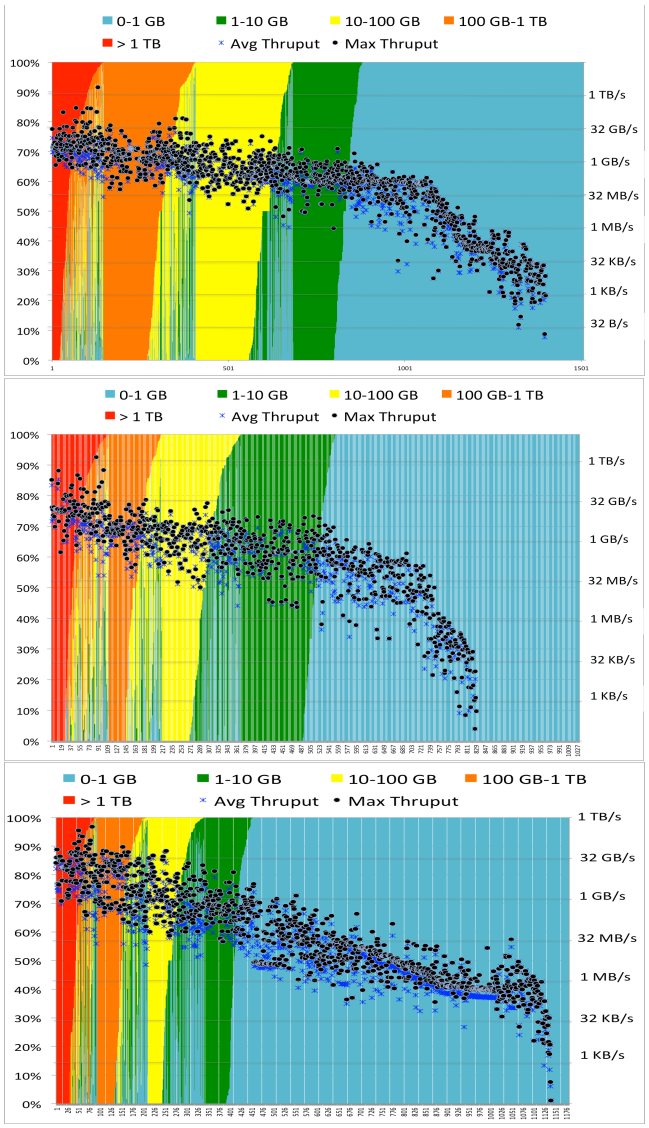

Figure 8: Breakdown of each app's jobs, by bytes written in each job, and average and maximum I/O throughput of each app's jobs. Intrepid is at the top, Mira in the middle, Edison at the bottom.

4.2.2 Platform I/O resource usage is dominated by a small number of jobs and apps.

On Edison, Intrepid, and Mira, the total I/O time consumed by all jobs observed by Darshan is 5,920 hours, 13,052 hours, and 5,335

hours, respectively. With jobs sorted by their total I/O time, Figure 9 shows the cumulative portion of platform I/O time that they use. On Edison, the top 10% of jobs consume 90% of the I/O time. On Intrepid and Mira, the top 25% of jobs consume 90% of the I/O time. The curve is even steeper for apps (not shown): 90% of I/O time goes to under 4% of apps on Intrepid, 3% on Mira, and 6% on Edison; each platform has approximately 1K–1.5K apps. These results echo the findings of [1], in which a single app dominated I/O time usage in a two-month study.

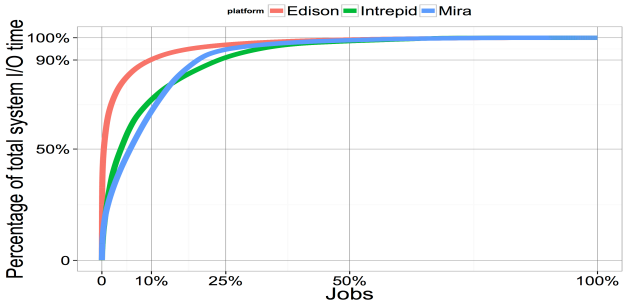

Figure 9: Cumulative percentage of platform I/O time consumed by jobs.

Let us look at these apps more closely. Table 2 and Table 3 show the 15 biggest apps on Mira and Edison, in terms of total I/O time across all their jobs. In what follows, we refer to these as the **big-time apps.** (Materials1, Turbulent1, and Molecular1 each merge two apps with near-identical executable names. We consider apps to be the same across platforms if their executables' names differ at most in version numbers). Apps whose names are in bold are on the big-time list for multiple platforms. Since Darshan is not configured to observe data accesses using character-oriented I/O, the I/O time for text-file-based apps is undercounted when picking out the big-time apps. To save space, we omit Intrepid's table, which includes Mira's Earth1, Physics2, Turbulence2, and Molecular2 at ranks 5, 7, 13, and 15, respectively, and Edison's Weather1 at rank 3. The top 15 big-time apps account for 83% of I/O time on Mira, 70% on Edison, and 73% on Intrepid. The total data read/written across all their jobs varies from a high of 10 PB for Earth1 and Materials3 to a low of 1 TB for PDE1.

Table 2: Mira's 15 Apps with Biggest Total I/O Time

#	App	Total I/O time (h)	Total run time (wall h)	# of jobs	Total bytes (TB)	Median job GB/s	Run time I/O %
1	**Earth1**	2,480	4,406	17,649	10,037	1.205	56%
2	**Materials1**	577	22,912	4,579	196	.103	3%
3	Turbulence1	428	4,121	972	153	.123	10%
4	Physics1	150	3,387	762	1,051	.475	4%
5	**Physics2**	133	6,262	1,966	1,115	.467	2%
6	Climate1	95	2,039	1,520	112	.291	5%
7	Molecular1	89	27,826	19,622	156	.571	0%
8	**Turbulence2**	83	671	335	251	.212	12%
9	Turbulence3	74	96	323	1,961	1.700	77%
10	Physics3	67	202	66	51	3.274	33%
11	**Molecular2**	67	1,686	2,480	34	.167	4%
12	PDE1	62	120	298	1	.098	52%
13	Plasma1	48	934	58	3,052	18.32	5%
14	Physics4	42	202	309	90	.186	21%
15	Aero1	41	61	151	359	2.505	67%

Improvements in big-time apps' throughput may free up resources for others to use and improve the satisfaction of all users. This principle drives the attention given to important apps and their I/O benchmarks; and indeed, the I/O behavior of at least five of the apps in Table 2 and three in Table 3 is well studied and carefully tuned. However, apps with I/O bugs and with I/O paradigms that are suboptimal for their situation also appear in the tables. For example, as we discuss elsewhere, PDE1 used global text files with many processes, and Earth1 used relatively small POSIX writes to global files. Indeed, the apps in these tables are top in *usage* of I/O time, not top in terms of I/O throughput. Apps that are extremely successful in extracting I/O performance will not be listed in the tables unless their total data size is incredibly high.

Table 3: Edison's 15 Apps with Biggest Total I/O Time

#	App	Total I/O time (h)	Total run time (wall h)	# of jobs	Total bytes (TB)	Median job GB/s	Run time I/O %
1	Materials2	1,109	3,397	847	60	.016	33%
2	Materials3	505	7,329	78,302	10,351	.475	7%
3	Physics5	395	2,698	2,171	6	.005	15%
4	Physics6	322	3,353	6,687	15	.010	10%
5	Materials4	263	8,252	1,231	17	.038	3%
6	Molecular3	249	7,392	2,194	51	.036	3%
7	**Materials1**	219	11,671	16,221	44	.109	2%
8	Materials5	215	21,439	34,213	27	.061	1%
9	Materials6	213	983	926	16	.070	22%
10	Chem1	145	18,909	5,412	4	.013	1%
11	Materials7	129	453	5,769	18	.039	29%
12	**Weather1**	110	686	299	1,189	.660	16%
13	Materials8	103	1,011	1,383	2,477	7.993	10%
14	Materials9	93	175	12,344	266	.860	53%
15	Plasma2	89	102	41	246	2.265	87%

In the tables, the percentage of run time that big-time apps devote to I/O rises from ~0% for Molecular1 on Mira to 87% for Plasma2 on Edison. Owners and users of apps at the low end of this range are likely to be happy with their I/O throughput, even if platform owners are not. **Boosting the minimum aggregate throughput for all big-time apps to 1 GB/s would save platform owners 42% of total I/O time on Intrepid** (3758 hours out of 8920)**, 41% on Mira** (1803 hours out of 4435), **and 85% on Edison** (3542 hours out of 4158). Jobs running concurrently with big-time apps might also benefit from increased I/O resource availability.

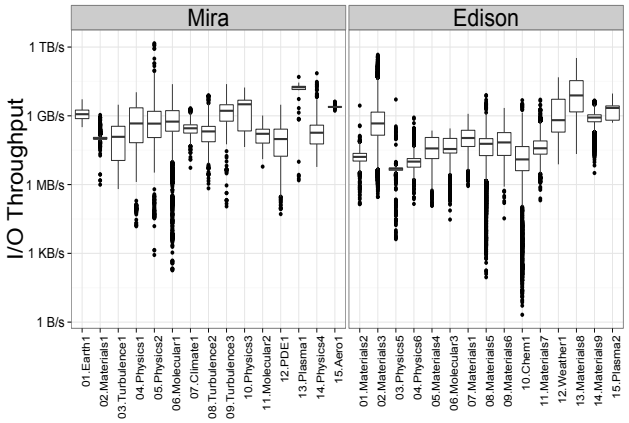

Figure 10: Big-time apps' throughput on Mira and Edison.

According to the tables, less than a quarter of Edison's and Mira's big-time apps get over 1 GB/s I/O throughput in their median job; only one gets over 10 GB/s in its median job (Plasma1, 18 GB/s on Mira). Figure 10 shows the quartiles and outliers for the I/O throughput of the big-time apps' jobs on Mira and Edison. As was true for the set of all jobs, big-time apps' jobs get better I/O throughput when they have more data. Figure 11 shows this with a four-category breakdown of the big-time apps' performance, based on whether they have *small data* (read/write under 10 GB) and/or *few processes* (under 2K). Figure 11 shows that most big-time apps' jobs with big data and processes get 1-16 GB/s of throughput on Mira. As we will see, each platform has apps with much higher median throughput than the big-time apps.

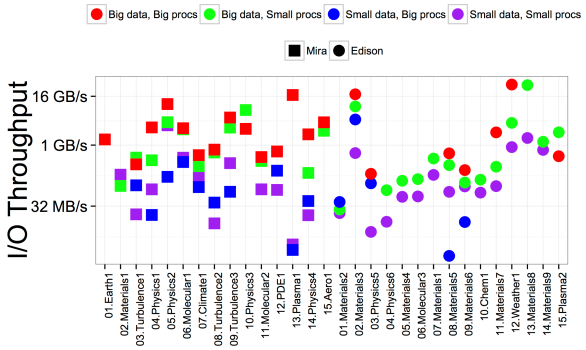

Figure 11: Average I/O throughput of Mira's (squares) and Edison's (circles) big-time apps' jobs, by job size.

4.2.3 Early intervention by platform owners can identify apps with I/O problems, save I/O resources, and improve user satisfaction.

Table 2 and Table 3 show that most of the big-time apps on Mira and Edison ran over a thousand times, and all but three ran over a hundred times. Clearly, early intervention where needed could have saved a huge amount of system resources. As Figure 12 shows, almost all big-time apps have small jobs, especially on Edison, which is the newest platform; early smaller jobs are the ideal point for recognizing and addressing problems.

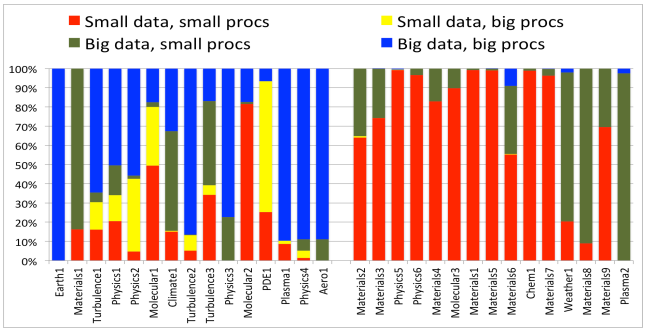

Figure 12: Job sizes for Mira (l) and Edison (r) big-time apps.

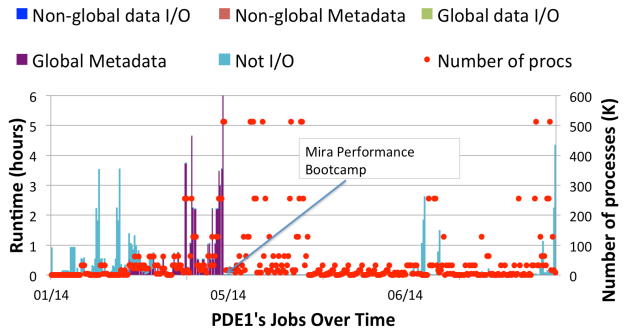

Figure 13: Evolution of PDE1's I/O paradigms. Red dots show the number of processes of each job (right-hand y-axis).

For example, PDE1 in Table 2 used ~13M core-hours on Mira and spent 52% of its run time in I/O. When PDE1 ran at scale (64K–128K processes) in its first implementation, I/O consumed almost all of its run time. For example, one job with 512K processes took 7 hours and over 3.5 million core-hours. Figure 13 includes a stacked bar for each successive PDE1 job, breaking down its total run time; the 7-hour run is excluded because it is off the chart. The clump of blue bars in Figure 13 shows that in its early runs at scale, PDE1's I/O time was devoted to metadata functions; in fact, the data transfer time for most files was zero. This tells us that the files are being read/written with functions not tracked by Darshan, namely, character-oriented functions for text files.

Conversations with PDE1's owner confirmed that the initial implementation used *fprintf* to write the output file accessed by all processes. After PDE1's owner attended Mira performance boot camp, the owner created an MPI-IO–based implementation that runs in 11 seconds with 512K processes. PDE1's owner would have benefited from automated analysis of the Darshan logs from its early jobs in Figure 13. Without extending Darshan to track character-oriented I/O functions, a script can still find apps that make heavy use of text files, by searching the logs for instances of files with high metadata time and zero data read/write time.

The logs also show how app I/O behavior evolves over time. PDE1's earliest runs used few processes, so its I/O paradigm was inexpensive relative to computation. As the number of processes went up, I/O dominated (purple bars). The purple bars disappear with the change to MPI-IO.

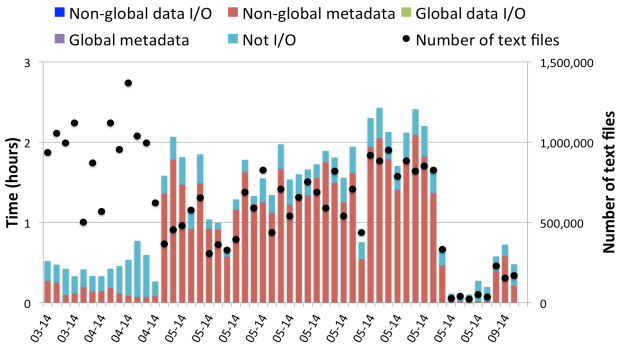

Figure 14: Earth2 read hundreds of thousands of text files.

As another example, consider "Earth2", an Earth science code that ran for 60 hours wall time on Mira and consumed about 100K core hours. It read from hundreds of thousands to over a million files and spent the vast majority of its time in I/O, as shown in Figure 14. Its I/O time breakdown reveals the tell-tale pattern of text files: high metadata time and zero data access time. Later, its owners identified a bug that put their read operations inside an unrelated nested loop, rather than outside. This costly bug persisted for a long time before it was noticed. The situation is another argument for automated early intervention.

We suggest the following four criteria to help platform owners identify apps whose I/O behavior makes them candidates for further investigation. The criteria are not absolute indicators of I/O problems, but rather help to narrow down the number of applications to consider.

- Apps using a text file I/O approach, such as PDE1 and Earth2. A query for jobs that use only text files finds 2121 jobs from 59 apps on Edison, 5561 jobs from 237 apps on Mira and 4725 jobs from 171 apps on Intrepid.
- Apps with many files and high metadata costs. For example, a query for Mira jobs with over 100k files and metadata time that is more than one third of run time finds 111 jobs from 11 apps, including Physics4 (discussed in Section 4.2.6).
- Apps with little data but large I/O time. For example, on Edison, a query for jobs with under 4 GB of data that spend over 5 minutes in I/O finds 4020 jobs from 79 apps. One of the apps has more than 500 jobs that match this criterion.

- Big time apps, such as the Top 15 discussed earlier.

The filtering capability further emphasizes the importance of having a central database about system workload that will enable early intervention from platform owners to save system resources and improve system utilization.

4.2.4 POSIX I/O is far more widely used than parallel I/O libraries.

The HPC community has worked hard to create a stack of parallel I/O libraries, including MPI-IO, HDF5, and NetCDF. But Figure 15 shows that users tend to stick with the POSIX I/O library (open, read, write). Nearly 95% of jobs visible to Darshan on Edison use POSIX exclusively. On Intrepid and Mira, the percentages are 80% and 50%, respectively. The remaining jobs use MPI-IO directly or use the libraries built atop MPI-IO (e.g., HDF5), for at least one of their files. MPI-IO is used most often among mid-sized jobs, in terms of their number of processes.

The POSIX-only approach does not necessarily mean low I/O performance; with care, POSIX apps can have high throughput. However, using MPI-IO offers more chances for decent I/O performance. As shown in Figure 16, on Mira and Intrepid, about 45% of jobs that used the MPI-IO library achieve more than 1 GB/s of aggregate I/O throughput, while less than 20% of POSIX-only jobs reach 1 GB/s. On Edison, most apps that used MPI-IO do not do so efficiently, although some have excellent throughput. We return to this point in our cross-platform analysis.

Carns et al. [1] analyzed the usage of different I/O interfaces and found that most jobs with few processors used POSIX I/O, while jobs with many processors used POSIX primarily for reads, if at all. MPI-IO prevailed among jobs with many processors and apps that wrote more data than they read. We found that, in addition, POSIX is popular among many-processor jobs. This result agrees with another study in [26].

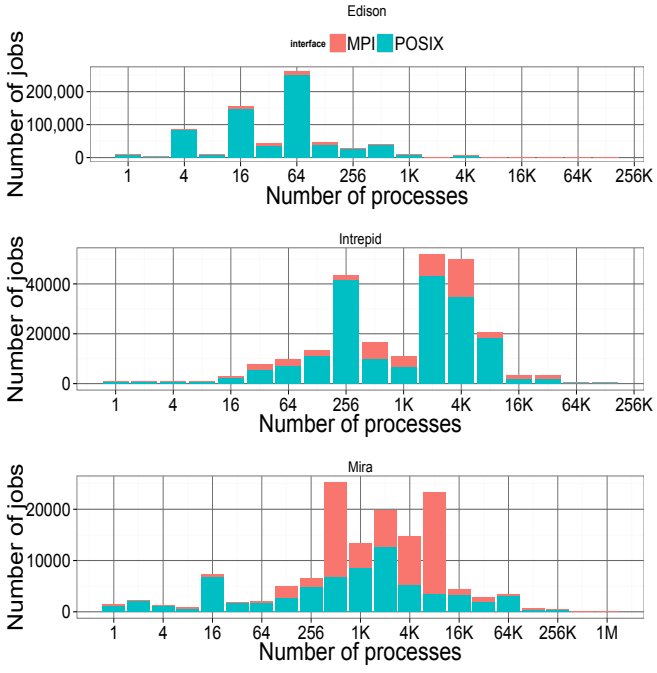

Figure 15: Number of jobs using POSIX IO only (teal) and using MPI-IO directly or indirectly for at least one file (red).

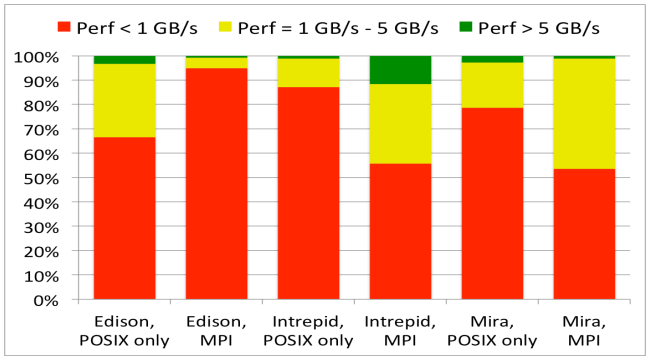

Figure 16: I/O throughput for apps that use only POSIX-IO and those that use MPI-IO for at least one file.

4.2.5 Metadata costs often exceed data I/O costs.

Metadata costs are a major factor in the I/O throughput of apps [10]. Averaging across the platforms in Figure 17, roughly 40% of jobs spend more time in metadata functions than in reading and writing data. We have already touched on a variety of reasons for this problem: the prevalence of small-data jobs and apps, which Figure 17 highlights; the hidden problem of overreliance on text files; and small data request sizes.

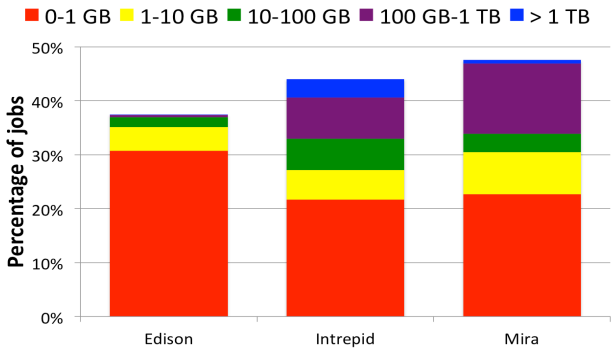

Figure 17: Breakdown of jobs by total data size, for jobs that spend more time in metadata functions than in data transfer.

4.2.6 No major I/O paradigm is always good or bad.

Text files almost guarantee poor throughput at scale; we do not consider apps using this minor I/O paradigm in this section.

As mentioned earlier, nonglobal files can be broken down into *local* files (i.e., accessed by one process) and *subset* files (i.e., accessed by more than one process but not all processes). An app uses the subset paradigm because it makes sense for the scientific problem and computational method—for example, adaptive mesh refinement—or because the owners want to put a subset of the processes (e.g., one process per node) in charge of all I/O. We call the latter *subsetting I/O*. Subsetting I/O can reduce contention and the number of files, but requires care for a good implementation. Taken to the extreme, subsetting turns into serial I/O, where one process does all the I/O, which never scales. In interpreting logs, we must distinguish between these three kinds of subset files.

Local files, often called file-per-process, are easy for users to implement, with no coordination between processes. But as the number of processes goes up, metadata costs can be high, and post-run data analysis and file management become painful. The use of global files, each accessed by all processes, can keep the job's input/result data tidy. But global files can have high metadata costs at scale, and contention can be an issue. Good

implementations of this paradigm tend to require expertise, and the resulting parallel I/O libraries have a learning curve for users.

Some of these categories can be broken down further. For example, a sophisticated app might use subsetting I/O with files that are accessed by (and thus "global" to) all the processes allowed to perform I/O. And an app can use multiple paradigms in different jobs or inside one job. But the coarser breakdown suffices for our purposes. Each paradigm—global, local, and subset—has its pros and cons, and each is found among jobs with the worst *and* best I/O throughputs.

Local-file paradigm. If a job has enough data, it may be able to avoid the pitfall of excessive metadata costs at scale. An excellent example is the set of all jobs that access at least 1M files, grouped by app in Figure 18. Each job is represented by one vertical bar, subdivided into colors based on how it spends its I/O time. Each job also has a yellow dot indicating its throughput and a black X indicating its data size. Two apps are very tightly packed: Physics4 (116 jobs on Mira) and Plasma1 (1 subset-paradigm job on Intrepid and 41 local-file paradigm jobs on Mira). Figure 18 shows how fast the local file paradigm can be: Mira1 attains over 10 GB/s with ~25 TB of data, and Plasma1 attains about 20 GB/s for its many jobs with 60–80 TB of data. Plasma1 on Mira shows that even with millions of local files and metadata costs (red) exceeding transfer costs (blue), I/O throughput can reliably reach a level that would be the envy of most apps. But the I/O time of the vast majority of local-file jobs in the figure is almost totally dominated by metadata costs, resulting in extremely low throughput for example Physics4. The throughput closely tracks the data size, both of which use the same right-hand y-axis. Two of Plasma1's data points are off the chart: 174 TB and 100 TB of data.

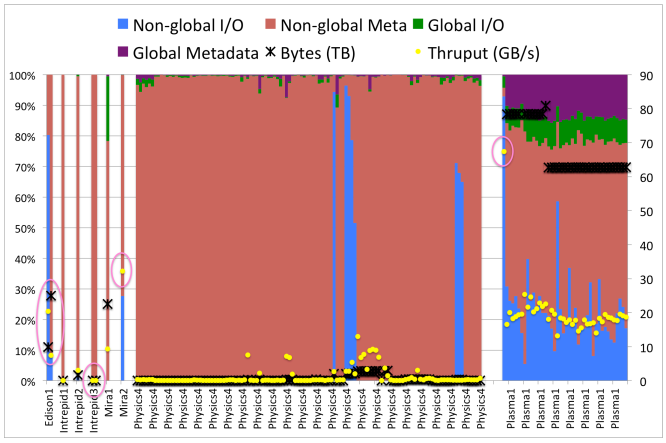

Figure 18: I/O throughput and I/O time breakdown for jobs that access over a million files.

Subset paradigm. In Figure 18, jobs that used the subset paradigm have pink circles around their yellow throughput dots. Two apps used the subset paradigm exclusively, and the figure shows that it can be very effective: Mira2 attained ~30 GB/s and Edison1 had 10–20 GB/s—far better than most apps. (Mira2's job has 165 TB of data, putting that data point off the chart.) But the third app, Plasma1, is the star, with over 60 GB/s in its lone subsetting job on Intrepid. The logs show that 1/8 of Plasma1's processes performed I/O in that job, and approximately 1/225 of Mira2's. Edison1 is using subset files, but not *I/O* subsetting; recall that subsetting serves other purposes too, such as AMR I/O.

Subsetting is not a panacea: Intrepid3 has poor I/O throughput, totally dominated by metadata costs. However, Intrepid3 was not doing *I/O* subsetting, as three-quarters of its processes wrote to the same file. For a better example of ineffective I/O subsetting, consider Turbulence1, which ran on Intrepid and Mira and is among Mira's big-time apps; its I/O time there averages 10% of run time. Figure 19 shows Turbulence1's Intrepid jobs, sorted by non-I/O time; the dark blue blocks are jobs using POSIX IO with subsetting (ratio 1000:1), and the light green blocks use MPI-IO with global files. No matter what paradigm is used, the I/O time has little impact on total run time, so the owners would have little motivation to try other I/O approaches. (One way to achieve this insensitivity is to dedicate processes to I/O, so computation can resume once the output data has been sent to those processes.) Examining a randomly selected job, however, we see that 90% of Turbulence1's I/O requests are of size 8 B, which could be inefficient for the storage system and could impact other users.

Climate1 also offers I/O subsetting, along with interfaces to a variety of storage options. Figure 20 shows that users took advantage of these different options in its many jobs on Intrepid. Through other channels, we know that Climate1's owner worked very hard to tame metadata costs and reach its median job throughput on Mira, which Table 2 pegs at a low 0.3 GB/s. But Climate1's throughput may still be hurt by very small I/O request sizes. For example, in three randomly selected Mira and Intrepid jobs including both primarily POSIX and primarily MPI-IO runs, over half its I/O requests have size ~100 B. A randomly selected Intrepid job shows subsetting ratios ranging from 4:1 to 1000:1 during different parts of the job; each job subsets differently, with little visible impact on I/O throughput. With median job I/O time at just 5% of total run time on Mira, Climate1's owner has little incentive to refine its I/O approach further.

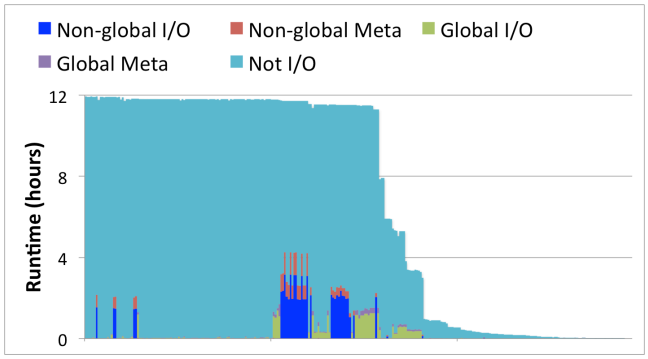

Figure 19: Turbulence1's 290 jobs on Intrepid.

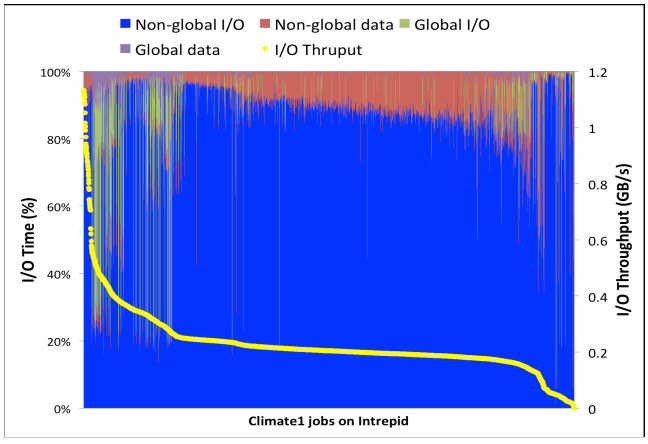

Figure 20: Climate1's 3578 Intrepid jobs, sorted by thruput.

Global files. Global files did not perform well for Earth1, which made small POSIX IO requests, or Climate1, which made small requests with both MPI-IO and POSIX. But Figure 21 shows that global MPI-IO files work well for the jobs of the "Physics7" app on Edison, shown sorted by throughput. The I/O throughput of Physics7's median job is 7 GB/s, helped along by its tendency to access data in 1 MB requests, well aligned with storage block boundaries. Also, Physics7 might not be using the default Lustre settings, which are slow for MPI-IO [24]. Physics7's users experimented twice with nonglobal files: once when they first arrived on Edison and then again after about a hundred jobs, always using a dozen or more processes. Both trials were quickly abandoned.

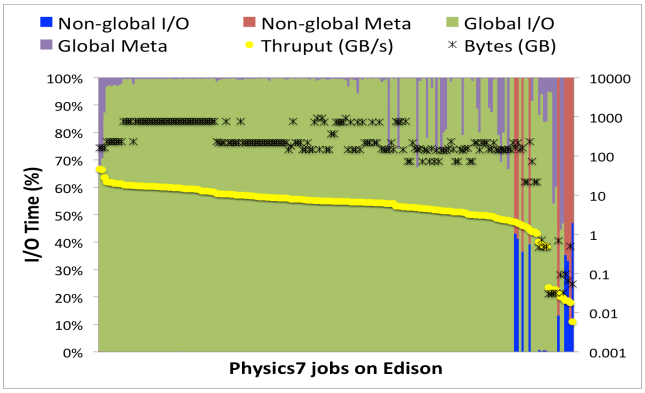

Figure 21: Physics7's 199 Edison jobs, sorted by throughput.

4.3 Cross-platform analysis

Supercomputer lifetimes are short; a new and faster platform is always on the way. But improved performance does not always come easily for users, as noted by Anantharaj et al.: "The high development and maintenance effort required to tune [applications] to multiple platforms is considered a large burden, taking time and resources that might otherwise be spent on other aspects of the projects" [25].

Migration to a new platform normally requires retuning of code for good performance, and I/O is no exception. Seemingly small details of the storage system can have a huge impact on a particular app's throughput [22]. Further, the general trend toward packing more cores into each node tends to increase file access contention for processes in the same node. Thus an app running with the same number of processes on a new platform might see throughput fall even if the new storage system is similar to the old and has higher peak throughput. Therefore, to maintain current throughput, app I/O may need retuning even when moving to a similar but faster platform. Case studies and I/O benchmarks have provided such insights in the past; Darshan can potentially help us examine the impact of migration at a larger scale.

Using the same naming methodology as in Table 2 and Table 3, we found the apps that ran on two or more of our platforms: 82 apps on both Intrepid and Mira, 39 on Mira and Edison, 27 on Intrepid and Edison, and 10 on all three platforms. For each such app, we compared the median aggregate I/O throughput of its jobs across platforms. However, most of these median jobs have small total data, as do most apps; Figure 22 illustrates this with a box plot of job data size for the ten apps that ran on all three platforms, with apps separated by vertical black bars. We have already observed that a small-data job will have well under 1 GB/s aggregate I/O throughput. Thus, the difference in median aggregate I/O throughput of jobs on different platforms is due primarily to differences in a job's total data size. For a fair cross-platform comparison of these apps, we need to match job sizes across platforms. With over a hundred apps to match up, we present just three case studies here.

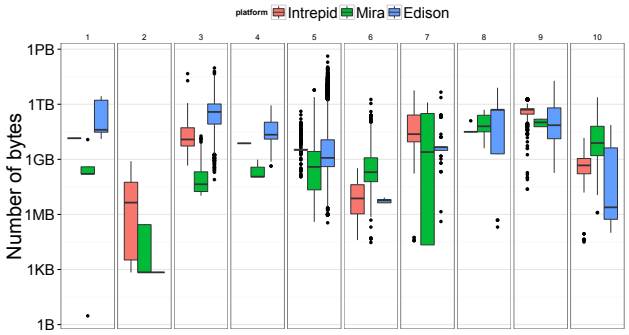

Figure 22: Quartiles and outliers of total bytes accessed by each job, for the ten apps that ran on all three platforms.

Case study 1: Earth1 is the number 1 big time app on Mira and number 4 on Intrepid. Figure 23 shows I/O throughput and data size of all Earth1 jobs on Mira and Intrepid. Median job throughput drops from 4.5 GB/s on Intrepid to 1.2 GB/s on Mira. Data size also declines but remains too big to explain the drop.

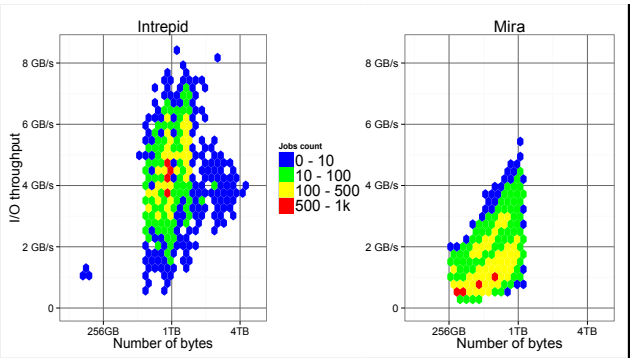

Figure 23: Earth1's jobs, broken down by data size and I/O throughput, on Intrepid (left) and Mira (right).

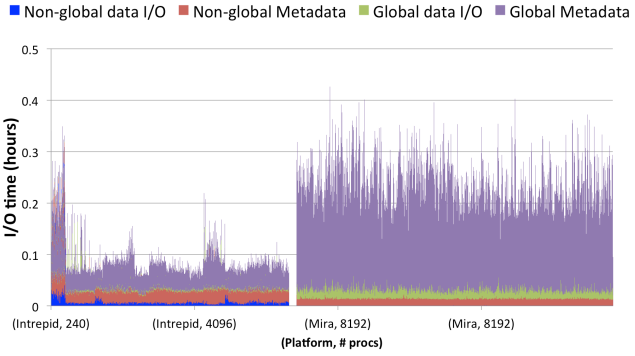

Figure 24: Earth1 jobs' I/O time on Intrepid (l) and Mira (r).

As shown in Figure 24, Earth1's main Mira bottleneck is metadata activity for global files. As discussed earlier, Earth1 uses POSIX to write to global shared files. Earth1's jobs on Mira use more processes, which are packed more tightly into nodes than on Intrepid. With more processes and less total data, request sizes drop. Tighter packing, more processes issuing requests, and smaller requests all increase contention, and throughput drops.

Case study 2: The "Crossplat1" physics code is the rightmost app in Figure 22. Figure 25 shows that in general, Crossplat1 scales

well with increasing data on Mira and Edison, and with more processes on Edison. On Intrepid, Crossplat1 rarely exceeded 1 GB/s throughput.

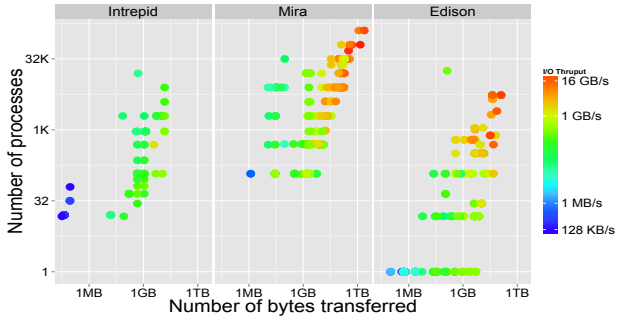

Figure 25: I/O throughput, data size, and number of processes for each of Crossplat1's jobs on three platforms.

We applied the three-step app-specific analysis procedure to Crossplat1 on Mira and Edison, and found that most I/O time was spent in non-global I/O of a number of local POSIX read/write files (*#lPrwf* for short). Figure 26 depicts this for Mira, with jobs sorted by #lPrwf. The log-scale right-hand y-axis is for the overlay variables: I/O throughput, total bytes and #lPrwf. For a fixed #lPrwf, I/O throughput increases nicely with data size. But when #lPrwf is 512 or more, metadata costs shoot up (tall red bars). This suggests that limiting #lPrwf may improve throughput for Crossplat1 on Mira. Crossplat1's behavior on Edison was similar (graph omitted) except that #lPrwf did not exceed 256, so metadata costs remained modest in almost all jobs on Edison.

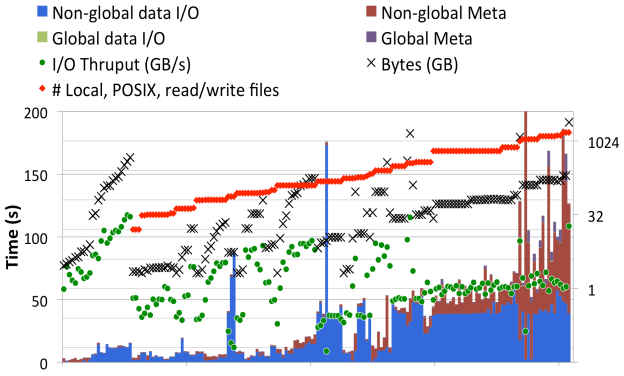

Figure 26: I/O time and throughput (green dot), bytes accessed (black X) and number of local POSIX read/write files (red diamond) for each of Crossplat1's jobs on Mira.

Case study 3: Weather1 is the ninth app in Figure 22 and a big-time app on Edison and Intrepid. Weather1 has few Mira runs, and we do not consider them here. Figure 27 shows that Weather1's I/O throughput was consistently low on Intrepid, but as high as 48 GB/s on Edison. The scaling pattern is unclear.

In Figure 28, each Weather1 job on Intrepid is represented by a vertical bar whose colors give a breakdown of the job's total I/O time (left-hand y-axis). The figure also shows each job's I/O throughput (black X), number of processes (yellow dot) and data size (blue plus) on the log-scale right-hand y-axis. The jobs are sorted by data size. Different I/O paradigms were used by different users, visible in the figure as four distinct blocks of colors. Weather1 spent most of its I/O time in MPI global shared files and never reached 1 GB/s of throughput under any paradigm, even when accessing over 1 TB of data.

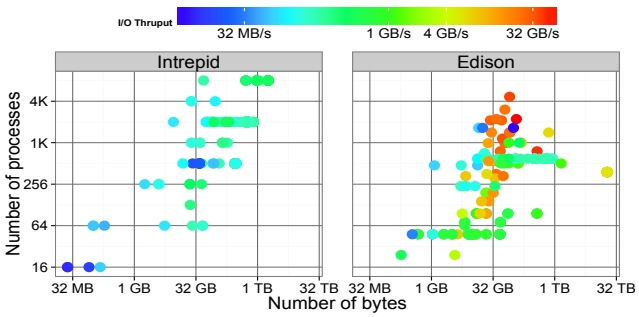

Figure 27: Breakdown of Weather1's jobs by I/O throughput, number of processes, total data size, and platform.

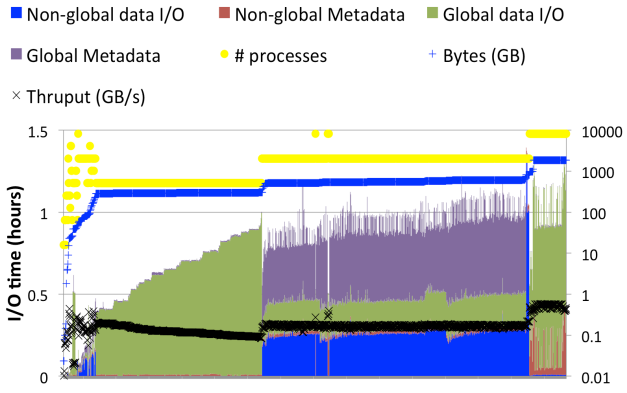

Figure 28: I/O time breakdown of Weather1 jobs on Intrepid.

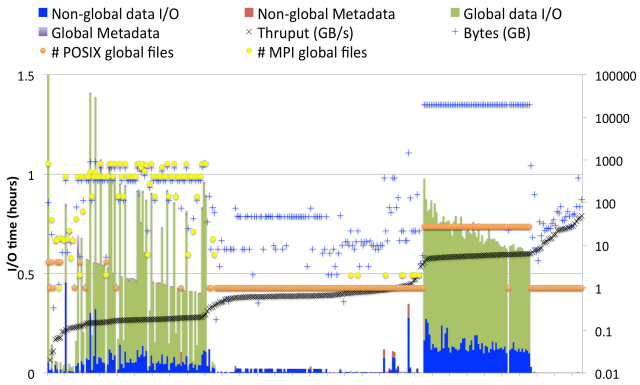

Figure 29: I/O time breakdown of Weather1 jobs on Edison.

Weather1 fares better on Edison, where a third of the jobs exceed 1 GB/s throughput, as shown in Figure 29 with jobs sorted by I/O throughput (black X). The figure also shows each job's data size (blue plus), number of POSIX global files (orange dot) and number of MPI global files (yellow dot) on the log-scale right-hand y-axis. Here, Weather1 jobs fall into three groups. The first group uses MPI-IO global shared files and has consistently low throughput (<0.2 GB/s). The second group uses local files and more modest data sizes (always under 1 TB) and throughput closely tracks data size, reaching as high as ~48 GB/s. The third group of jobs has extremely large data (over 10 TB), and uses POSIX global files; these jobs attain 3-6 GB/s. Darshan does not observe whether jobs tune Lustre parameter settings, but it is worth noting that these results are in line with others' observations that the default settings on Lustre lead to low MPI-IO performance [24], and that the local file I/O paradigm tends to perform relatively well on Edison.

5. CONCLUSIONS AND FUTURE WORK

Efficient I/O performance is a critical part of modern supercomputing. Lightweight tools such as Darshan can augment traditional benchmarking and tracing tools, and provide an overall understanding of the I/O behavior of apps, workloads, and platforms. This paper used Darshan I/O logs to provide a broad view of I/O behavior on three leading HPC platforms. Our results lead us to believe that while tremendous progress has been made in hardware and software research for HPC I/O, gaps remain in the adoption of best practices by scientific application developers. For instance, strategies such as usage of text files and raw, low-level POSIX I/O calls will be untenable on future platforms; adoption of higher-level I/O libraries can help increase the longevity of codes on future generations of supercomputers. HPC I/O specialists need to ensure that app developers understand the tradeoffs in different ways of performing I/O, perhaps through I/O boot camps and tutorials offered in cooperation with platform owners. Our results also lead us to believe that while much research effort is invested in extreme- scale testing and optimization, a large fraction of the HPC community has modest-scale metadata and data challenges; designers of HPC facilities must take these needs into account when designing and provisioning I/O resources. We believe that tools such as Darshan can give platform owners critical insights into system utilization; early and proactive intervention into suboptimal I/O behavior can greatly enhance the utilization of a platform's HPC resources.

Acknowledgments. We thank our application and platform owners and users for helpful discussions. This work was supported by NSF 0938064 and the U.S. Department of Energy, Office of Science, Advanced Scientific Computer Research, under contracts DE-AC02-06CH11357 and DE-AC02-05CH11231; the work used resources from ALCF and NERSC.

REFERENCES

[1] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, R. Ross ,Understanding and improving computational science storage access through continuous characterization, ACM Trans. on Storage, 7(3):8, 2011.

[2] Fastest USB Thumb Drive, 14 February 2014. http://www.maximumpc.com/fast_usb_thumb_drive_2014.

[3] B. K. Pasquale, G. C. Polyzos, A static analysis of I/O characteristics of scientific applications in a production workload, Supercomputing, 1993.

[4] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, M. L. Best, File-access characteristics of parallel scientific workloads, IEEE Transactions on Parallel and Distributed Systems 7(10):1075–1089, Oct. 1996.

[5] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E.L. Miller, D.D.E. Long, T. McLarty, File system workload analysis for large scale scientific computing applications, IEEE Conference on Mass Storage Systems and Technologies, 2005.

[6] S. Saini, D. Talcott, R. Thakur, P. Adamidis, R. Rabenseifner, R. Ciotti, Parallel I/O performance characterization of Columbia and NEC SX-8 superclusters, IPDPS, 2007.

[7] Y. Kim, R. Gunasekaran, G. M. Shipman, D.A. Dillow, Z. Zhang, B.W. Settlemyer, Workload characterization of a leadership class storage cluster, 5th Petascale Data Storage Workshop, 2010.

[8] S. Saini, J. Rappleye, J. Chang, D. Barker, P. Mehrotra, R. Biswas, I/O performance characterization of Lustre and NASA applications on Pleiades, HiPC, 2010.

[9] Darshan-util installation and usage, http://www.mcs.anl.gov/research/projects/darshan/docs/darshan-util.html.

[10] P. Carns, Y. Yao, K. Harms, R. Latham, R. Ross, K. Antypas, Production I/O characterization on the Cray XE6, Cray User Group Meeting, 2013.

[11] FLASH: http://www.flash.uchicago.edu/site/

[12] GCRM: https://svn.pnl.gov/gcrm

[13] Nek: http://nek5000.mcs.anl.gov/index.php/Main_Page

[14] CESM: http://www2.cesm.ucar.edu/

[15] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. A. Herdman, I Miller, A. Vadgama, A. H. Bhalerao, S. A. Jarvis, Parallel File System Analysis Through Application I/O Tracing, http://eprints.dcs.warwick.ac.uk/1582/The Computer Journal, 56 (2), 2013.

[16] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, Scalable I/O tracing and analysis, Parallel Data Storage Workshop, 2009.

[17] M. P. Mesnier, M. Wachs, R.R. Sambasivan, J. Lopez, J. Hendricks, G. R. Ganger, D. O'Hallaron, //TRACE: Parallel trace replay with approximate causal events, File and Storage Technologies, 2007.

[18] N. J. Wright, W. Pfeiffer, A. Snavely, Characterizing parallel scaling of scientific applications using IPM, LCI International Conference on High-Performance Clustered Computing, 2009.

[19] LANL-Trace: http://institutes.lanl.gov/data/software/index.php#lanl-trace

[20] A. Aranya, C. P. Wright, E. Zadok, TraceFS: A file system to trace them all, File and Storage Technologies, 2004.

[21] H.V.T. Luu, B. Behzad, R. Aydt, M. Winslett, A multi-level approach for understanding I/O activity in HPC applications, Workshop on Interfaces and Abstractions for Scientific Data Storage, 2013.

[22] S. Langer, B. Still, D. Hinkel, B. Langdon, E. Williams, A pF3D case study of obtaining good I/O performance while running on over 100,000 processors, http://visitbugs.ornl.gov/attachments/43/Langer-pf3d-IO-study-Mar2011-final1.pdf

[23] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, K. Riley, 24/7 characterization of petascale I/O workloads. IEEE CLUSTER, 2009.

[24] B. Behzad, S. Byna, S. M. Wild, Prabhat, M. Snir. Improving parallel I/O autotuning with performance modeling. High-performance Parallel and Distributed Computing, 2014.

[25] V. Anantharaj, F. Foertter, W. Joubert, J. Wells, Approaching exascale: Application requirements for OLCF leadership computing, Oak Ridge National Laboratory, July 2013. https://www.olcf.ornl.gov/wp-content/uploads/2013/01/OLCF_Requirements_TM_2013_Final1.pdf

[26] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC), 2008.