

# Toward Universal Analysis and Visualization of HPC Application Input/Output Behavior

Huong Luu, Amirhossein Aleyasen, Marianne Winslett, Yasha Mostofi, Kaidong Peng  
University of Illinois at Urbana-Champaign  
{luu1, aleyase2, winslett, mostofi2, kpeng7}@illinois.edu

## ABSTRACT

Analyzing the I/O performance of high-performance computing applications can provide valuable insights for application developers, users, and platform administrators. However, the analysis is difficult and requires parallel I/O expertise few users possess. Analyzing an entire platform’s I/O workload is even harder, as it requires large-scale collection, cleaning and exploration of data. To address this problem, we created a web-based dashboard for interactive analysis and visualization of application I/O behavior, based on data collected by a lightweight I/O profiler that can observe all jobs on a platform at low cost. The dashboard’s target audience includes application users and developers who are starting to analyze their application’s I/O performance; system administrators who want to look into the usage of their storage system and find potential candidate applications for improvement; and parallel I/O experts who want to understand the behavior of an application or set of applications. The dashboard leverages relational database technology, a portable graphing library, and lightweight I/O profiling to provide I/O behavior insights previously only available with great effort.

## 1. INTRODUCTION

I/O is an important factor in determining the overall performance of many high-performance computing (HPC) applications. Analyzing the I/O performance of these applications can provide valuable insights for application developers, users, and platform administrators.

For example, an analysis of the runtime behavior of an individual job is often sufficient to pinpoint its I/O bottlenecks. However, most scientific codes run not just once, but hundreds or thousands of times. Additional insights can be gleaned by analyzing not just an individual run, but the set of all runs of an application over time; we call this *self-benchmarking*. The broader view obtained through self-benchmarking can show an application’s developers and users how changes in scale, such as the number of processes, amount of data read and written, and number and types of files, affect the application’s I/O throughput and run time. The effects of other changes to the application, such as experiments with different I/O paradigms or tuning parameters (e.g., stripe size), can also be clearly seen through self-benchmarking. At an even broader scale, if all applications on a particular supercomputer instance are self-benchmarking, then we can examine the patterns in I/O behavior across that entire platform. The resulting analysis may suggest ways for the platform’s administrators to free up I/O resources, e.g., by suggesting potential improvements to the applications that use the most I/O time on the platform.

However, analyzing HPC application I/O behavior is a difficult task that requires parallel I/O expertise, a relatively rare commodity. It is even harder to analyze the I/O behavior of a platform’s entire workload – a massive amount of data must be collected, tidied up, explored, and visualized. For this reason, usually only the I/O behavior of a relatively small set of

“important” applications on a platform receives careful scrutiny. A rising tide lifts all boats, and it is perhaps reasonable to assume that improvements in the performance of “important” applications are likely to be relevant and transferable to other applications as well. But we can do better today: low-cost technology is available to make applications and HPC platforms self-benchmarking.

Figure 1 shows the workflow we use for a self-benchmarking application or platform. As shown in the figure, the first step is to use an I/O profiler to collect summary information about applications’ I/O activities. (In this paper, we consider two jobs to belong to the same application if they have the same executable name.) Second, we load the data collected by the profiler into a relational database, so that analysis will be easy in spite of the many jobs being profiled. Our scripts for parsing and uploading data into the database are available to all at [12].

Next, dashboard users specify which application(s) are of interest to them, triggering a standard set of SQL queries that access the database. The queries compute summaries of the profiles collected for all the jobs of the specified application. The dashboard converts the query results into graphs that the user can interact with and manipulate in various ways. For example, in a graph that shows the I/O throughput for each run of an application, the user can choose to sort the application’s runs by data size, number of processes, submission date and time, or other factors. The user can choose a particular date range to focus on, or look at the time spent in particular I/O activities, such as accessing globally-shared files. Finally, the insights obtained at the dashboard can be used to tune the application, closing the loop shown in Figure 1.

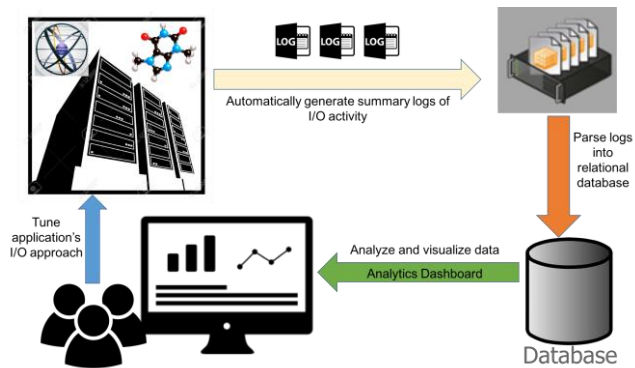


Figure 1: Self-benchmarking workflow

In theory, any I/O profiler that observes the time spent in application-level I/O function calls can do the data collection and summarization needed for self-benchmarking. In practice, for the platform to be self-benchmarking, the profiler needs to be deployed by default for all jobs on the platform, so it must add negligible overhead to each job’s execution time. In this paper, we use data from the lightweight profiler Darshan [1], which is enabled in the default environment on supercomputers at Argonne

National Laboratory (ANL), National Energy Research Scientific Computing Center (NERSC), and the National Center for Supercomputing Applications (NCSA). I/O experts have used Darshan data for application-specific, system-wide and cross-platform analysis [2][3], crafting queries and generating visualizations by hand. Replicating these activities for additional applications or platforms is time-consuming and requires I/O expertise. Our goal in this paper is to show that the analytics and visualization found in work such as [3] can be performed automatically by the dashboard, removing the I/O expert from that loop and doing away with the need for manual analysis for high-level summary statistics. We anticipate that the dashboard will also be useful for I/O experts, as it automates many tasks that they previously may have had to perform by hand.

In the remainder of the paper, we describe related work in Section 2, explain the analytics and visualization provided by the dashboard in Section 3, and offer conclusions in Section 4.

## 2. BACKGROUND AND RELATED WORK

Two main types of tools are widely used in I/O performance diagnosis and optimization: profiling and tracing tools.

I/O tracing provides detailed information about application I/O executions and allows in-depth analysis of application performance. After traces have been generated, they can be used for debugging, performance tuning, or creating benchmarks. Researchers have created a variety of I/O tracing tools; for example, RIOT I/O [4], ScalaIOTrace [5], //TRACE [6], IPM [7], LANL-Trace [8], TraceFS [9], and Recorder [10] fall into this category. I/O tracing tools are ideal for investigating individual runs in full detail, but are too heavyweight to be used for every run of every application on a platform.

I/O profiling omits many of the details recorded in I/O tracing, thus offering a lighter-weight way to analyze application I/O behavior. Popular I/O profiling tools include IOPro [11] and Darshan. IOPro is a profiling and analysis framework that traces the flow of I/O behavior in the entire parallel I/O software stack, providing a multi-level view of the interaction between different layers. IOPro has not yet been used to offer analysis across multiple runs of an application or platform workload analysis.

The Darshan I/O profiling library characterizes a job by providing statistics and cumulative timing information, including the number of processes, number of files accessed, and bytes read/written; aggregate I/O throughput; and the total run time and I/O time, among other statistics. In previous work [3], we parsed Darshan data and loaded it into a relational database, then performed application-specific, system-wide and cross-platform analysis. In this project, we continue our work with I/O profiler data to create an interactive analysis dashboard that automates the activities we carried out manually in our previous work.

Visualization-assisted data exploration becomes necessary to find useful information when working with vast amount of data. There has been much work on visualization to assist performance analysis for HPC applications, especially for communication, such as Jumpshot [13], Vampir [14] and TAU [15]. For I/O analysis, IOVis [16] is an I/O tracing and visualization tool that captures the I/O requests at multiple layers of the I/O stack and allows end-to-end analysis. IOVis provides a timeline view of activities of the entire system, from computation nodes to I/O servers. Sigovan et al [17] provide a visualization of the activities of an I/O network. Our dashboard is intended for visualizing application I/O performance, rather than I/O network performance. Memaxes [18] is an interactive visual analysis tool for memory accesses.

## 3. THE DASHBOARD

To ensure portability and scalability, the dashboard’s visualization facilities run on the client side and are implemented using the freely available HighCharts library, which is written in JavaScript. In the implementation described in this paper, we use MySQL database instances and communicate with them over the web using SQL queries, PHP, and JSON. The current dashboard implementation comprises roughly 12K lines of code.

Figure 2 shows the layout of the dashboard screen: a navigation panel down the left-hand side (area 1), basic options across the top (area 2), and a chart area (area 3). The navigation panel includes two types of functionality: the *Analyze Platform I/O Workload* section provides platform-wide analyses, which are geared toward the needs of platform administrators; and the *Analyze Application I/O Behavior* section is used for application-specific analyses. Clicking on a specific entry in the navigation menu highlights that entry and takes the user to that particular kind of analysis, as described in the sections that follow.

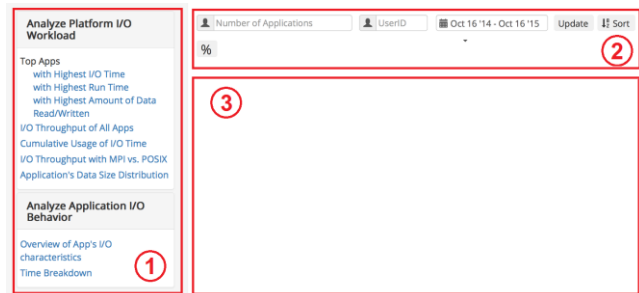


Figure 2: Layout of the dashboard

Across the top of the screen are options to narrow down the scope of an analysis. Once the user has entered values for the options across the top of the screen, those values persist when the user chooses a new type of analysis in the navigation menu, unless those options are not relevant for that type of analysis. The options are the same for most, though not all, of the entries in the navigation menu, so we discuss them here.

First, filling in the field *Application Name* narrows down the scope of the analysis to a particular application, i.e., executable name. Clicking on that field brings up a list of the applications with jobs in the database and visible to the user; an administrator can see all applications, while an ordinary user can only see applications run by members of his or her project. The current implementation is specifically for users with administrative privileges that allow them to examine all applications.

Second, the user may fill in the *User* field, to narrow the scope of the analysis to the jobs run by a particular user. Both the *User* and *Application Name* fields may be filled in if desired; this is particularly useful for widely used applications that offer several different I/O paradigms and may be used in many different ways.

Third, the user may fill in a date range to indicate the jobs of interest. The default is to show jobs submitted within the past year; other options include the past month, week, day, or any custom range of days within the scope of the data in the database.

Fourth, the *Update* button should be clicked when the user has changed other options along the top row, to generate the graphs and tables corresponding to the newly chosen values for those options.

Fifth, once the user has clicked the *Update* button to produce a graph, the *Sort* button provides many different ways to order the

data shown in that graph. For example, the user could choose to sort the jobs shown along the X axis of a graph by submission date, I/O throughput, and so forth. This feature is extremely important for making hidden patterns of behavior visible.

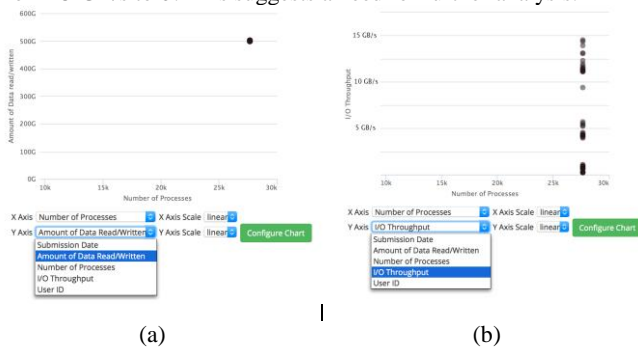
Finally, a little lower down on the right-hand side is a printer icon. Clicking that icon brings up several options for printing out the current analysis results.

In the remainder of this section, we first present the dashboard’s functionality for analyzing the I/O behavior of a single application, then its facilities for examining the I/O behavior across an entire platform. All screenshots in this paper are taken from the dashboard implementation as of October 2015, using real data from a major supercomputer installation. We have obfuscated the names of the applications to preserve the privacy of their developers and users.

### 3.1 Analyze an Application’s I/O Behavior

The dashboard provides two ways to analyze an application’s I/O behavior, via a scatter plot to provide an overview of the application’s I/O characteristics, or via a stacked bar graph that supports a more detailed analysis of how the application spends its time. These two functions, *Overview of App’s I/O Characteristics* and *Time Breakdown*, are located at the bottom of the left-hand side navigation pane of the dashboard, as shown in Figure 2. If the user has not already filled in the *Application Name* or *User* fields, s/he must do so after clicking on *Overview of App’s I/O Characteristics* or *Time Breakdown*. After filling in any other desired options such as a date range, the user clicks *Update* to perform the analysis and generate the corresponding graphs.

As shown in Figure 3, *Overview of App’s I/O Characteristics* gives a high-level view of an application’s I/O needs and job configurations. (In this and subsequent figures, so that the graphs can be seen more easily, we have clipped off the screenshot’s navigation pane and the options buttons, which remain the same across almost all analyses.) The user chooses which aspects of I/O behavior to show in each scatter plot, by choosing between the options given at the bottom of the screen: submission date, bytes read/written, number of processes, I/O throughput, or user ID. These options determine the features of the scatter plot, including what to show on the X and Y axes, and what scale to use for each axis (linear or log10). The real application shown in the figure uses the same number of processes and reads and writes the same amount of data in all of its many runs (a). Figure 3(b) shows that despite that consistency, its I/O throughput in different jobs varies from 15 GB/s to 0! This suggests a need for further analysis.

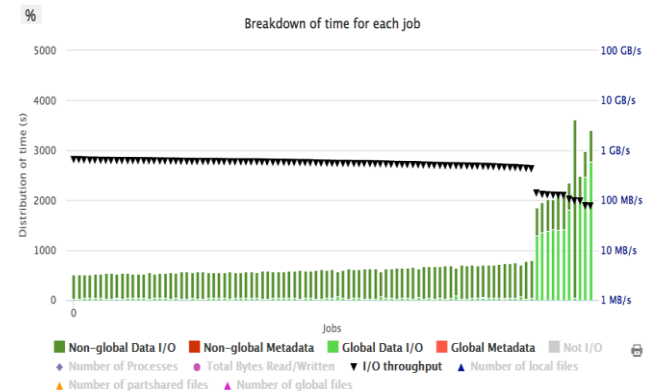


**Figure 3: Two example *Overview of App’s Characteristics* graphs, showing (a) number of processes vs. bytes, and (b) number of processes vs. I/O throughput**

*Time Breakdown* is the most important analysis for identifying how an application is spending its time. As shown in Figure 4, we break down the total runtime into *Not I/O* plus four different kinds of I/O time; the former includes computation and communication time. I/O time is divided into the four categories we found most helpful in the manual analyses in our previous work [3]:

- **Global Metadata.** All metadata functions for global files (i.e., files accessed by all processes), such as file open, close, stat, and seek functions.
- **Non-global 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 part-shared files, which are accessed by multiple (but not all) processes, for example under a subsetting I/O paradigm.
- **Global Data I/O.** Data transfer functions for global files. These include the read, write, and sync functions.
- **Non-global Data I/O.** Data transfer for non-global files.

We expect that this way of breaking down the I/O time will be easy to understand even for users without I/O expertise, because it aligns with the major decisions they had to make on data management, such as whether they are willing to cope with thousands of result files or prefer to just have a few.



**Figure 4: Example *Time Breakdown* graph**

In the legend below the graph, the user can choose which of these four types of I/O time to show in the graph, as well as the Non-I/O Time, by clicking on them in the legend. In Figure 4, the grayed-out parts of the legend are what the user has chosen to omit from the graph. Clicking the % button toggles between using absolute time units on the Y axis on the left-hand side of a graph, or showing a percentage breakdown of time. The user can also choose different job sort orders on the X axis to more easily see relationships between factors that may influence I/O throughput. The user has the option of superimposing overlay variables on the graph, including the number of processes, amount of data read/written, I/O throughput, number of local files, part-shared files and global files. These overlay variables use the right-hand Y axis, and they can be used to determine the sort order for jobs on the X axis, whether or not the user has omitted them from the graph. As shown in Figure 5, the user can sort the jobs on multiple characteristics at the same time, which we have found particularly useful in our previous manual analyses.

Adding overlay variables to a *Time Breakdown* graph helps users to literally see correlations, and to narrow down the number of factors they need to consider more closely. So does the sort order. For example, in Figure 4, the user has asked to see the jobs sorted in decreasing order of an overlay variable, the I/O throughput. As

shown, when global-file data I/O time (light green bars) becomes large, I/O throughput dives. This suggests the user should take a closer look at how the application accesses globally shared files.

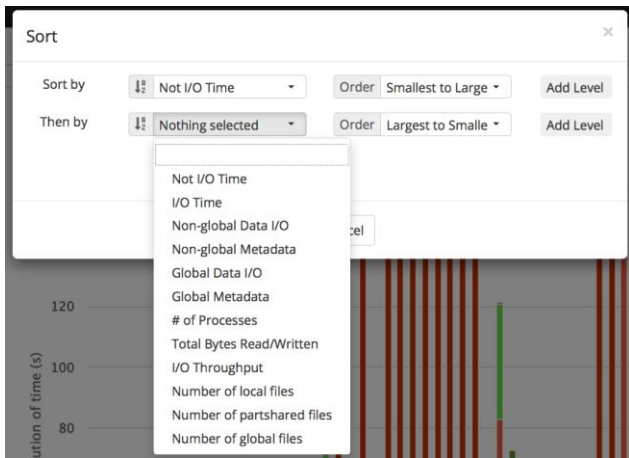


Figure 5: Multiple levels of sorting options

### 3.2 Analyze a Platform’s I/O Workload

The dashboard currently supports several types of analysis of platform I/O workload, to help administrators understand the usage of their storage systems and identify potential candidates for further tuning. Due to space limitations, we cannot describe all the platform-wide analyses currently supported by the dashboard, so we focus on those we expect to be most useful.

One big concern for administrators is whether applications can fully utilize the platform’s high performance I/O system, and what fraction of the peak throughput applications actually achieve. This analysis is performed by *I/O Throughput of All Apps* on the navigation menu. Figure 6 shows an example result, giving the maximum and median throughput of all runs of all applications on the platform during the user-specified time period. The user has sorted the applications in decreasing order of their maximum throughput. As depicted, among about 400 applications, nearly half of them never exceeded 1 GB/s of I/O throughput in any run, though some exceeded 100 GB/s. Even for applications with at least one job that attained high throughput, their median job throughput can be several orders of magnitude lower.

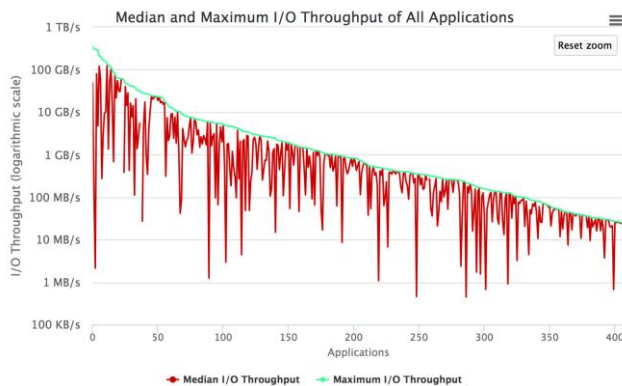


Figure 6: Max vs. median I/O throughput of all applications

Platform administrators may save many system resources by focusing on improving the performance of the applications that consume the most resources. There are several ways to define these *top applications*, such as those that consume the most I/O

system resources (highest I/O time), long running applications (highest runtime), or the applications that read/write the most data. For each of these definitions, the user can get an overview of top applications by clicking the corresponding *Top App* navigation menu entry and specifying how many applications to show (default is 10). As shown in Figure 7, a *Top App* analysis presents the user with a table containing aggregated high-level information about each top application. This includes the application name, total execution time of all its jobs, total I/O time of all its jobs, its total number of jobs, the percentage of time spent in I/O on average across all its jobs, and the median I/O throughput of its jobs. Below the table, the dashboard presents one scatter plot for each application in the table, as depicted in Figure 8. Each scatter plot is similar to that in the *Overview of App’s I/O Characteristics* function. The dashboard’s grid of scatter plots facilitates the comparison and contrast of the I/O characteristics of all the top applications.

Top Applications with Highest I/O Time							
Id	Name	Total Runtime (hours)	Total I/O Time (hours)	Total Amount of Data Read/Written (TB)	Number of jobs	Average Percentage of Runtime in I/O	Median I/O Throughput (GB/s)
1		16021	705	1893	11895	6%	0.05
2		381	219	1244	3346	59%	1.58
3		1342	77	651	1649	11%	0.24
4		657	75	42	278	23%	0.15
5		878	44	137	503	6%	4.16
6		62	36	189	1772	65%	0.71
7		195	25	2	65	23%	0.2
8		37	23	623	83	62%	7.97
9		23	22	23	321	88%	0
10		263	19	18	839	21%	0.14
11		20	18	4232	226	83%	1.33
12		210	17	2003	9	8%	36.86
13		2770	14	135	3028	1%	2.95
14		31	8	6	11	27%	0.19
15		9305	8	16	38674	0%	0

Figure 7: Example *Top Apps* table

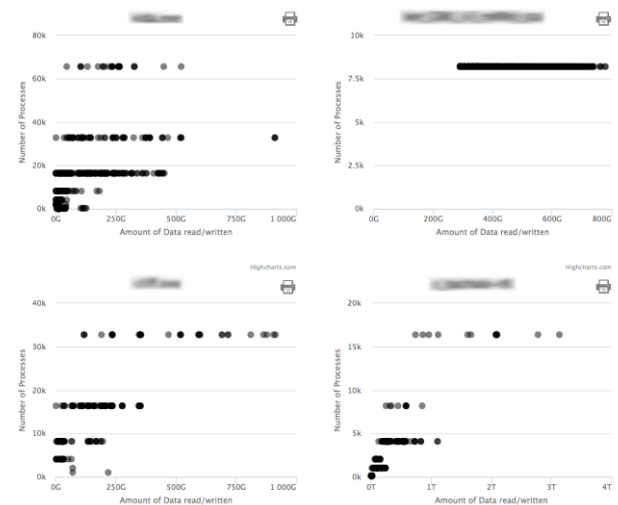


Figure 8: Example grid of scatter plots for *Top Apps*

## 4. CONCLUSIONS AND FUTURE WORK

Self-benchmarking was inspired by the fact that HPC applications already run so many times that they can serve as their own I/O benchmarks, if we can automatically profile all their runs and automate the analysis of the resulting data. We leveraged existing relational database technology, lightweight I/O profilers, and freely available, portable visualization libraries to accomplish this goal. Our choice of what to visualize was guided by our previous experience in manually analyzing and visualizing the I/O

performance of applications observed by profilers: we automated the functionality we had found most useful in understanding I/O behavior at supercomputing installations in our previous work.

Currently, we are serving as our own alpha users as we work with the developers and users of key applications on the Blue Waters platform. Our next set of users will be the administrators of other platforms where Darshan runs. We will release the dashboard to all users only after we address the security issues inherent in allowing external access to a database residing at a supercomputing site. For example, the dashboard must defend against SQL injection attacks and must support strong authentication of users. Our scripts to parse I/O profiles generated by Darshan and upload them into a relational database have already been released [12].

I/O profilers are ideal for showing the big picture, i.e., providing situational awareness. For example, the dashboard can help users quickly locate a bottleneck in their I/O approach and see how their application's I/O patterns evolve over time. Similarly, the dashboard can show administrators the I/O throughput that thousands of applications have obtained on their platform. Because they omit many details, I/O profilers are inexpensive enough to be broadly deployed without significant impact on applications' run-time performance. But because they omit many details, I/O profilers never tell the entire story. They may point a finger at an application and its I/O bottleneck, e.g., by showing that almost all its I/O time is going to metadata function calls in a handful of globally shared POSIX read-only files. But to learn exactly why those calls are so slow, an I/O expert may need to run the application several times with an I/O tracing tool and examine those more detailed trace results. For example, the I/O profiler that we used does not know how the user/application/platform has tuned the underlying storage system, and these tuning decisions can have a major impact on performance. In sum, the dashboard can provide useful insights into I/O behavior, but the insights do not always go deep enough to understand why an application is having performance problems. To understand the complete picture, the HPC community needs both I/O profilers and tracing tools.

This work was supported by NSF ACI 15-35177.

## 5. REFERENCES

- [1] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, K. Riley, 24/7 characterization of petascale I/O workloads. IEEE CLUSTER, 2009.
- [2] 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.
- [3] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Prabhat, Suren Byna and Yushu Yao. A Multiplatform Study of I/O Behavior on Petascale Supercomputers. In Proceedings of The 24th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC), 2015.
- [4] 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/TheComputerJournal>, 56 (2), 2013.
- [5] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, Scalable I/O tracing and analysis, Parallel Data Storage Workshop, 2009.
- [6] 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.
- [7] N. J. Wright, W. Pfeiffer, A. Snively, Characterizing parallel scaling of scientific applications using IPM, LCI International Conference on High-Performance Clustered Computing, 2009.
- [8] LANL-Trace: <http://institutes.lanl.gov/data/software/index.php#lanl-trace>
- [9] A. Aranya, C. P. Wright, E. Zadok, TraceFS: A file system to trace them all, File and Storage Technologies, 2004.
- [10] 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.
- [11] Seong Jo Kim, Yuanrui Zhang, Seung Woo Son, Mahmut T. Kandemir, Wei-keng Liao, Rajeev Thakur, Alok N. Choudhary, IOPro: a parallel I/O profiling and visualization framework for high-performance storage systems. The Journal of Supercomputing 71(3): 840-870 (2015)
- [12] Darshan-SQL scripts: <https://github.com/huongluu/DarshanSQL>
- [13] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp, "From trace generation to visualization: A performance framework for distributed parallel systems," in Proc. of ACM/IEEE Supercomputing (SC00), November 2000.
- [14] A. Knupfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Muller, and W. E. Nagel, "The Vampir performance analysis tool- set," in Tools for High Performance Computing, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Springer, Berlin, pp. 139-155.
- [15] S. S. Shende and A. D. Malony, "The TAU parallel performance system," Int. J. High Perform. Comput. Appl., vol. 20, no. 2, pp. 287-311, 2006.
- [16] C. Muelder, C. Sigovan, K.-L. Ma, J. Cope, S. Lang, K. Iskra, P. Beckman, and R. Ross, "Visual analysis of I/O system behavior for high-end computing," in Proceedings of the 3rd International Workshop on Large-Scale System and Application Performance (LSAP '11), 2011, pp. 19-26.
- [17] Sigovan, Carmen, Chris Muelder, Kwan-Liu Ma, Jason Cope, Kamil Iskra, and Robert Ross. "A visual network analysis method for large-scale parallel i/o systems." In Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, pp. 308-319. IEEE, 2013.
- [18] Alfredo Giménez, Todd Gamblin, Barry Rountree, Abhinav Bhatele, Ilir Jusufi, Peer-Timo Bremer, and Bernd Hamann. 2014. Dissecting on-node memory access performance: a semantic approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 166-176.