H5Prov: I/O Performance Analysis of Science Applications Using HDF5 File-level Provenance

Tonglin Li, Quincey Koziol, Houjun Tang, Jialin Liu, and Suren Byna

Lawrence Berkeley National Laboratory, Berkeley, CA, USA

***Abstract*—Systematic capture of extensive, useful science metadata and provenance requires an easy-to-use strategy to automatically record information throughout the data life cycle, without posing significant performance impact. Toward that goal, we have developed a Virtual Object Layer (VOL) connector for HDF5, the most popular I/O middleware on HPC systems. The VOL connector, called H5Prov, transparently intercepts HDF5 calls and records operations at multiple levels, namely file, group, dataset, and data element levels. The provenance data produced can also be analyzed to reveal I/O patterns and correlations between application behaviors/semantics and I/O performance issues, which enables optimization opportunities. In this effort, we analyze captured provenance information from two application benchmarks to understand HDF5 file usage and to detect I/O patterns, with preliminary results showing good promise.**

I. Introduction

The increasingly demanding data-driven sciences pose challenges to HPC systems and application data management strategies. While computing power of mainstream HPC systems have increased exponentially with new technologies such as GPUs/FPGAs, the performance of storage, especially I/O, has lagged far behind.

Due to the complex nature of parallel file systems, many scientific applications cannot achieve acceptable performance when accessing storage with standard POSIX I/O or MPI-I/O operations. However, the time and effort required to understand the I/O stack involved and extract optimal performance from it can be daunting for application science teams. In response, I/O middleware packages have been created that encapsulate this knowledge and provide it to applications with easy-to-use API interfaces.

Adoption of these I/O middleware packages is still limited, either because application users (domain scientists) lack knowledge and experience with them or because of the difficulty in deploying tuned versions of these middleware packages on all systems. Thus the efforts made for optimizing I/O performance in the I/O middleware only benefit the applications that are carefully prepared for the middleware, on systems where the middleware has been performance tuned.

Hierarchical Data Format (HDF) was originally designed as a library and data format for scientific applications. The current version, HDF5 [1], has evolved into a parallel data management system that bridges the gap between applications and the complicated, constantly-changing low-level details of storage systems. HDF5 now is the most widely used data format and management system for scientific applications [2].

HDF5 is designed to be highly extensible, and provides numerous possibilities to add enhancements and features with third-party plugins. In this work, we describe our lightweight configurable provenance logging system that records application operations on an HDF5 file.

Provenance information is generally regarded as a method of recording the "who, what, where, when, and how" of producing science results. However, we've determined that provenance data can also be enhanced and analyzed to reveal I/O patterns and correlations between application behaviors/semantics and I/O performance issues, which opens up further optimization opportunities, such as data prefetching [3], layout reorganization [4, 5], etc.

In contrast to many I/O profiling and provenance tracking systems, our system does not need a dedicated deployment, always-on server instances, or even modifications to application code. Users only need to set an environment variable to enable the provenance feature, and the provenance plugin will be automatically loaded by the HDF5 library when an application operates on HDF5 files, with provenance data recorded in a user-specified location.

Existing I/O profiling packages track low-level I/O events, such as which process writes to a file. This data is useful for tuning file system parameters, such as file striping and I/O scheduling strategies. However, application logic and data structures are invisible at that level. HDF5, on the contrary, has semantic knowledge of science data structure organization within a file. Recording the provenance of data production and later access gives us a chance to look at the resulting I/O behaviors from the application's perspective, and measure its performance impact upon the underlying file system.

The primary contributions of this paper are as follows:

- We have designed and implemented a new virtual object layer (VOL) connector for HDF5 to collect provenance on multiple levels with fine granularity.
- We have conducted large scale experiments on Cori, the Cray XC40 system at NERSC, with various storage system configurations to collect sample provenance traces and to evaluate the overhead for doing so.
- We analyzed the sample provenance data and have found some application misconfigurations and abnormal storage layer behaviors. This demonstrates the many possibilities of using HDF5 VOL connectors to optimize application logic, storage setup and middleware (HDF5) implementation.

II. HDF5 Virtual Object Layer (VOL) and Provenance Connector

A. Virtual Object Layer

Support for Virtual Object Layer (VOL) connectors was recently added to the HDF5 library. The VOL is an abstraction layer within the HDF5 library that intercepts object-level API operations on HDF5 files (such as 'file open', 'dataset write', 'group create', etc) and can forward those operations to plugins, called "VOL connectors"[6]. These connectors are dynamically loadable at runtime and enable third-party developers to build customized storage solutions for HDF5 users without having to change application code (figure 1). Therefore, HDF5 applications can benefit from new optimizations and capabilities with ease.

Existing works have demonstrated the flexibility and the benefits of using HDF5 VOL. For example, in order to exercise various object store technologies for scientific applications, researchers from NERSC, HDF and Intel have developed VOL connectors for Openstack Swift, Ceph Rados, and Intel DAOS [7]. These connectors redirect the applications' HDF5 function calls to different underlying storage, with less than 5% code change to the application. Other connector examples include an ADIOS connector, which maps HDF5 API calls to the ADIOS file format [8]; Data Elevator [9] and ARCHIE [10] which implement an efficient automatic data mover across hierarchical storage tiers for HDF5 files.

B. Design and implementation

Our provenance connector is written as a special-purpose *pass-through* VOL connector, which can be interposed above any other VOL connector to track I/O behavior to any *terminal* VOL connector the application has chosen. As the HDF5 application performs I/O operations, the VOL framework invokes our provenance connector, which logs information about the operations and then re-invokes the VOL framework to call the underlying VOL connector. In this way, the provenance connector can be inserted at run-time into any HDF5 application without even recompiling the code.

During H5Prov VOL connector design and implementation, we follow a homomorphic approach such that not only does every native HDF5 object have a counterpart in the virtual object layer, the native relationships have also been preserved in H5Prov connector. In other words, the virtual objects in the H5Prov connector have hierarchies just like their HDF5-native cousins.

For example, when creating a dataset, in addition to the native dataset in the file, a virtual dataset that contains provenance related information is also created in the H5Prov connector's memory. In a native file, a dataset is located in either a file or a group, similarly, a virtual dataset will be added to a linked list (held by the virtual file) when it's created or opened. The linked lists on different levels are used to accommodate concurrent opens on the same entity, either it's a file, group or dataset, and to do reference counting accurately.

This homomorphic design allow us to track HDF5 behavior statefully, thus being able to track more detailed information, and to easily extend features. For example, if we need to know how many times a certain dataset is accessed, we can simply add a field *cnt_accessed* to the virtual dataset structure, keep updating it, and output when it's closed. Most provenance systems (which are stateless) simply record every event they see as an entry, but are not aware of the relationship between the events. To capture the number of access to a dataset, they have to record every access to the dataset as an entry, and then count them after the upper layer application stops.

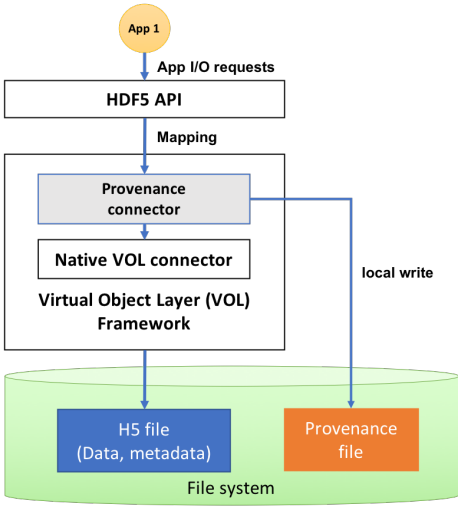


Figure 1: HDF5 VOL architecture

At the time of this paper submission, we have built a functioning and robust connector, and it has passed the extensive HDF5 regression test suite, so we are confident of correctly supporting any application usage of HDF5. Captured data includes user ID, process ID, thread ID, HDF5 operation name, duration of the operation, data object operated on (such as file/group/dataset object), the read/write size in bytes and so forth.

We have run large scale I/O benchmarks on NERSC's Cori system, with provenance tracking enabled. Because the high level I/O patterns of these benchmarks are well defined and thoroughly studied, we are creating an extensive HDF5 I/O pattern reference in the form of provenance traces for IOR. This will demonstrate the value in using provenance tracking for system optimization and make it possible to tune HDF5 more easily for a wider range of HPC systems.

C. Ease of use

To utilize H5Prov, end users don't have to change their HDF5 application code. Instead, they just need the latest HDF5 installed, and to set an environment variable with the provenance trace file path and format. They don't even need to recompile application code – HDF5 will load the VOL connector automatically. This non-invasive setup minimizes users' effort to adopt H5Prov and allows a wider range of users and developers to trace and tune their applications and lower level storage systems such as Lustre and NVRAM burst buffers.

III. Performance Evaluation

A. Experiment setup

1) Testbed: We conducted all the experiments on the Cori system at the National Energy Research Scientific Computing Center (NERSC) [11]. Cori is a Cray XC40 supercomputer with 2,388 Intel Xeon "Haswell" processor nodes and 9,688 Intel Xeon Phi "Knight's Landing" nodes. Our experiments run benchmarks on 2-128 Haswell nodes, with 64-4096 MPI ranks respectively.

2) Storage system configuration: We used Cori's Lustre file system and burst buffer as storage backends for our experiments. Lustre is configured with stripe counts of 64 and 128, and a stripe size of 16MB. Cori's burst buffer (BB) system is based on Cray Datawarp and has 20GB of SSD capacity per node. Because BB resource allocation is based on requested capacity, we requested an oversized 10TB capacity for experiments of all scales, such that the most possible BB nodes can be allocated, for maximum performance.

B. Benchmark setup

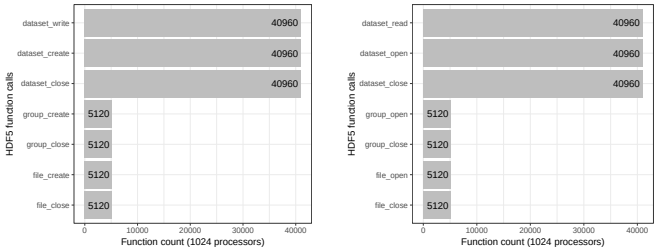
We used two benchmark kernels, namely VPIC and BDCATS (which focus on write and read-only access respectively), to test HDF5 IO performance. The VPIC-IO kernel is from a plasma physics simulation code called VPIC [12], which simulates magnetic reconnection phenomenon in space weather. In this kernel, each MPI process writes 8M ($8 * 2^{20}$) particles with each particle having 8 variables. VPIC data structures use 1-D arrays to represent each variable. The total length of each property array is equal to $n * 8 * 2^{20}$, where n is the number of MPI processes. The BDCATS-IO kernel is from a parallel clustering algorithm, used for analyzing the data produced by particle simulations, such as VPIC. In this kernel, data related to the particles is read among all the MPI processes in a load-balanced distribution. While these kernels use random data values, the I/O patterns exactly match that of the simulation and analysis. These two kernels use HDF5 and are highly tuned using MPI-IO and Lustre optimizations.

C. Data placement

We use the same data placement settings for VPIC and BDCATS. For Lustre experiments, the working directory is striped with 64 and 128 count, then the VPIC kernel run 5 times in a row, with both data and provenance trace written to the same working directory. The BDCATS kernel then follows the same pattern, reading the HDF5 files generated by VPIC kernel.

For Burst-Buffer experiments, VPIC kernel writes data and provenance traces to the same BB working directory. For BDCATS runs, we stage in pre-generated data to the BB before starting the kernel.

Since each process generates its own provenance file, large-scale operation will create thousands of files. To reduce the metadata stress on the backend storage systems, we write the provenance files to 64 pre-created sub-directories. After the benchmark is finished, provenance files are compressed and the tar ball is staged out by the BB. In this way, we ensure that application data and provenance data are using the same storage setup.



(a) VPIC (b) BDCATS

Figure 2: Involved HDF5 functions (with 1024 processors). The function occurrence of VPIC/BDCATS is highly concentrated, mostly focused on dataset read and write.

D. Investigated HDF5 functions

In the current version, HDF5 has 68 function calls for serving wide range of user requirement. In VPIC and BDCATS, only 10 major function calls are used:

- `file_create`
- `file_open`
- `file_close`
- `group_create`
- `group_open`
- `group_close`
- `dataset_create`
- `dataset_open`
- `dataset_close`
- `dataset_read`
- `dataset_write`

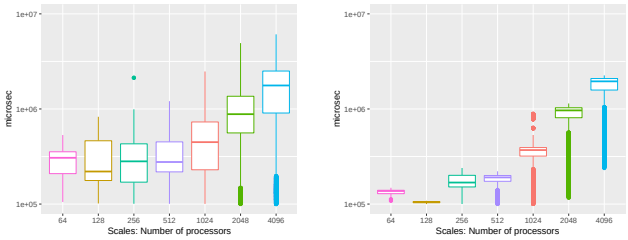
The occurrence frequencies can be found in Figure 2.

E. Provenance capture overhead

We evaluate the provenance overhead from two aspects, time cost and trace file footprint. To accurately record the time used just by the provenance operations, we put additional timers in each of the VOL operations. Beside the timers that measure the whole operation's running time, another timer is put around the underlying native HDF5 operations that we can use to calculate the pure VOL overhead. All these times sum up to be the measurement time overhead. For the space overhead, we use the size of merged trace files.

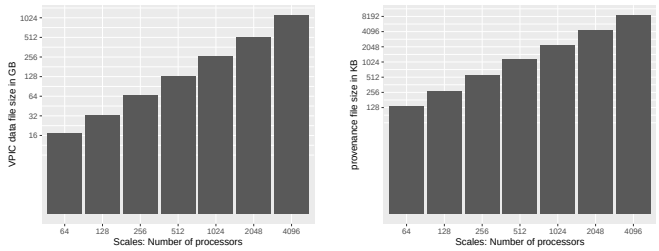
For most of the scaling runs, H5Prov took less than one second of total time to do it's job, comparing to around 20 seconds of application total running time, this includes wrapping native HDF5 objects, managing its metadata and writing to trace files.

With regards to scalability, H5Prov shows good performance. The time overhead increases gracefully, and the overhead only start to be meaningful when it's scaling up to 1024 processes. This is primarily due to the independent nature of H5Prov. H5Prov benefits from the burst buffer on both time overhead and variance 3. From a space overhead perspective,



(a) Lustre: stripe count = 128 (b) Burst-buffer

Figure 3: H5Prov temporal overhead: accurately recorded provenance function running time. The time overhead is proportional to scale and benefits from faster storage systems such as Burst-buffer.



(a) Accessed data size (b) H5Prov trace file sizes

Figure 4: H5Prov spacial overhead is negligible comparing to the accessed data volume. The only performance related issue is the number of trace files, which equals the number of processes.

H5Prov trace file size is directly proportional to the total number of operations (figure 4).

IV. Analysis of HDF5 provenance data

In this section, we demo the analysis on H5Prov trace data. Using H5Prov, we captured very fine granular trace for HDF5 applications. It allows us for the first time to have such a close look at every function.

A. Finding Hot and straggler functions

H5Prov provides statistical information about every function, such as the number of times it's called and the the duration of the function call. This enable us to find the "hot functions" on both the frequency and total time cost. Since both VPIC and BDCATS are relatively simple, and only used limited HDF5 functions, the data may not look interesting (figure 2). But it will give us insights when the application is more complex and uses more HDF5 functions.

B. Data operations

Dataset read and write take the most time during BDCATS/VPIC operation. However, the provenance trace shows that back-end storage systems have very different influence on their performance. The stripe count settings for Lustre has little impact to dataset read (figure 5a and 5b), even the burst buffer has only marginal advantage over Lustre, although BB helps to reduce the straggler operations slightly(figure 5c).

On the other hand, dataset write performance highly depends on the storage system capability. Counter-intuitively, with stripe count being 128, dataset write actually is moderately slower than that 64 stripe count on small to moderate scales (64 to 512 processors). We find this pattern in many other operations, such as dataset_read/open/close (figure 5, 8 and 9). But higher stripe count allow applications to scale more smoothly, and with less variance. This is because Lustre needs more time to coordinate between more stripes, thus a raised baseline latency. This effort is rewarded with better utilized parallelism on larger scales where the same workload is spread to more servers.

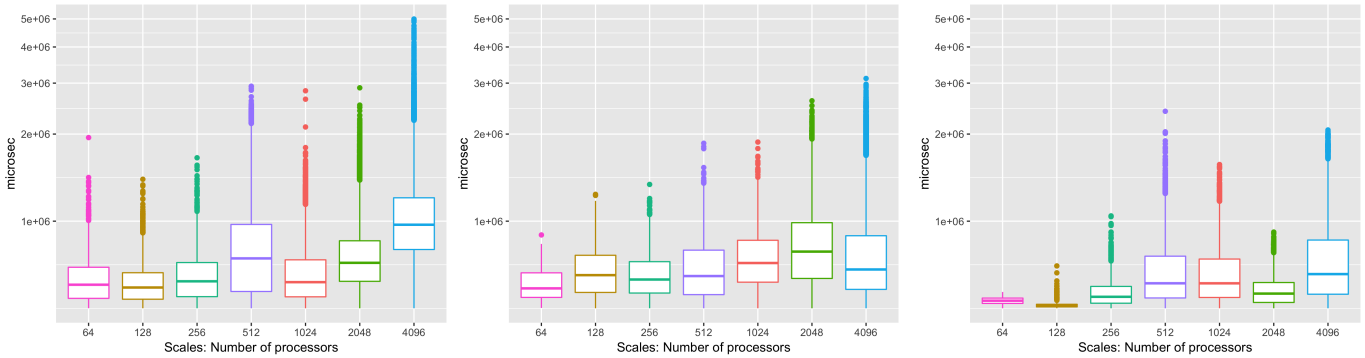
Meanwhile the burst buffer brought 10-50x performance gain and better scalability to VPIC (figure 6c), which spends most of the time in dataset_write. It is worth noting that the application benefits from the burst buffer the most on dataset_write, especially with large chunks of data, but only gets limited benefits on small I/Os, such as most of HDF5 metadata operations. This was not expected, since the burst buffer is expected to perform better on small IOPs, and bears further investigation.

C. Metadata operations

HDF5 metadata operations behave very differently from data operations. They are generally very light-weight requests and only access very small pieces of data. There are some collective metadata operations though, and in these cases we expected to see overhead increases proportional to the scale. However, most of metadata operations have comparable performance on different scales and backend storage systems only have marginal impact on this.

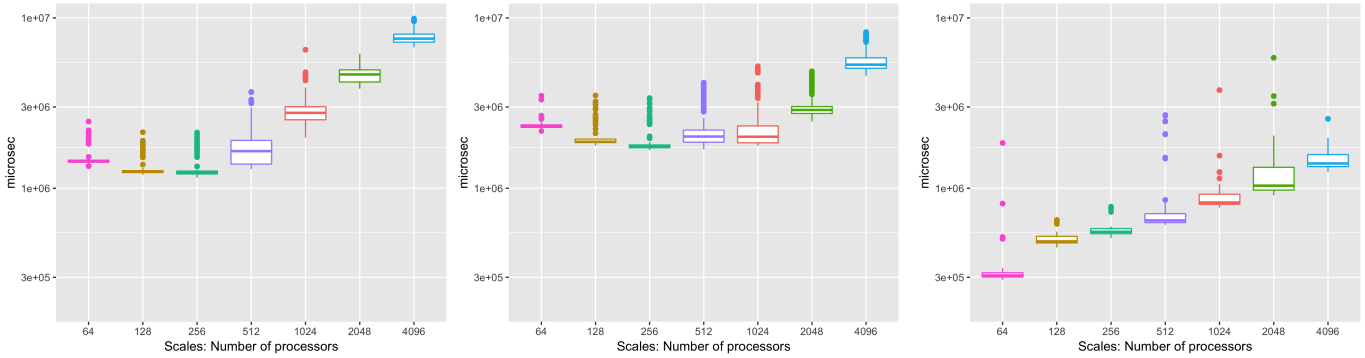
1) Dataset create/open/close: Dataset create and open operations show similar patterns when scaling. Both scale well, but the latter shows less variance (figure 8 and 7). This is because dataset_create is a collective operation and needs a synchronization (coordinated updates to the same HDF5 file) between all processes while dataset_open is an independent and read-only operation. On dataset_create, the burst buffer helps to eliminate stragglers dramatically, thus showing a narrower latency distribution.

On dataset_open, BB haves strangely, significantly slower than Lustre on all scales. At this time we're still investigating this behavior. Another unexpected result is that BDCATS' dataset_close is about 10x slower than VPIC's (figure 9 and 10)s. Note that files in BDCATS are opened for read-only access while they are writable in VPIC (and thus may require a "flush" action upon closing). This phenomenon is consistent across all scales and backend file systems and implies there could be different optimization solutions adopted in HDF5 core for dataset_close when it handles different open mode (read-only or append/write). Similar phenomenon can be found in group_open and group_create (figure 11 and 12), but NOT in group_close (figure 13 and 14).



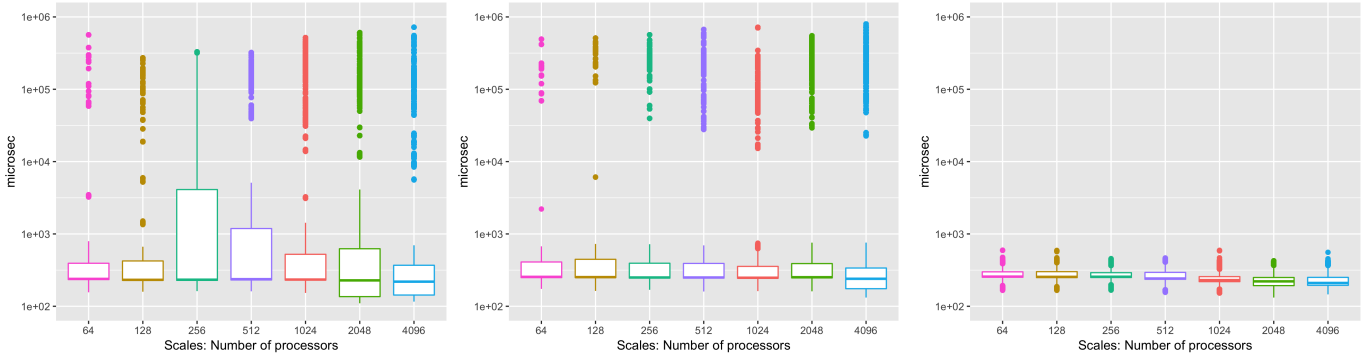
(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 5: Dataset read scaling: BDCATS. HDF5 dataset read is very efficient, and performs consistently across different storage systems. Burst-buffer didn't help much on reading.



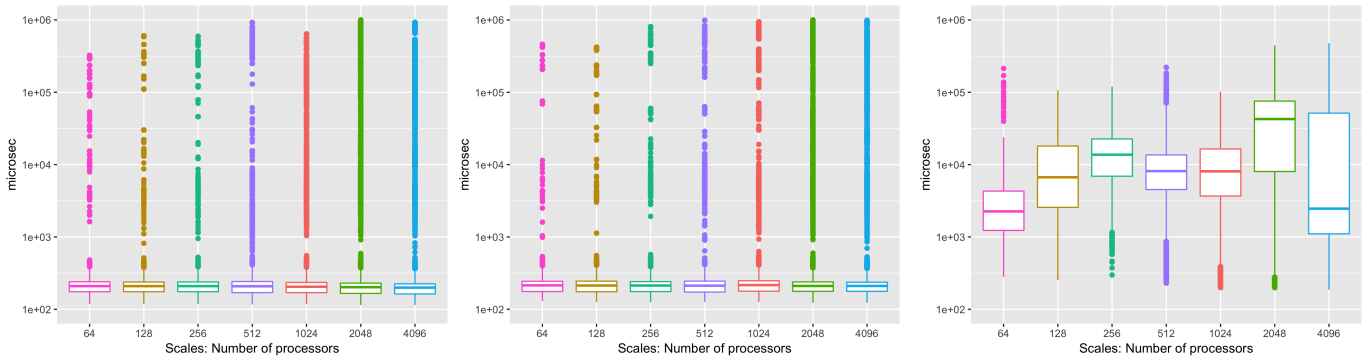
(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 6: Dataset write scaling: VPIC. Involving locking mechanisms on file system access, Burst-buffer provided more than 10x performance gain comparing to Lustre. On Lustre, higher stripe count allows applications to scale more smoothly, and with less variance, but may bring some coordination overhead between Lustre OTSs. This is significant on small scales, but is well paid off on larger scales when the overhead is amortized.



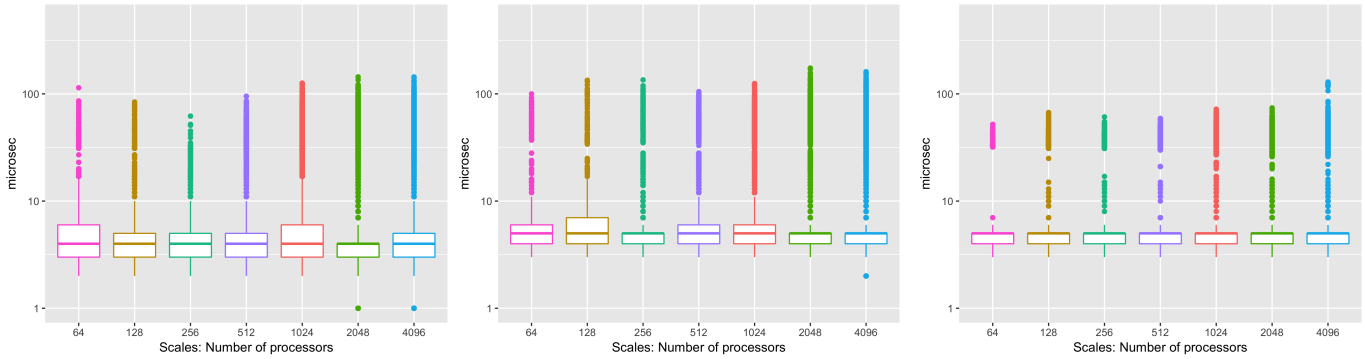
(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 7: Dataset create scaling: VPIC. Faster storage helps the most to reduce variance and straggler operations, but not for mean or median time.



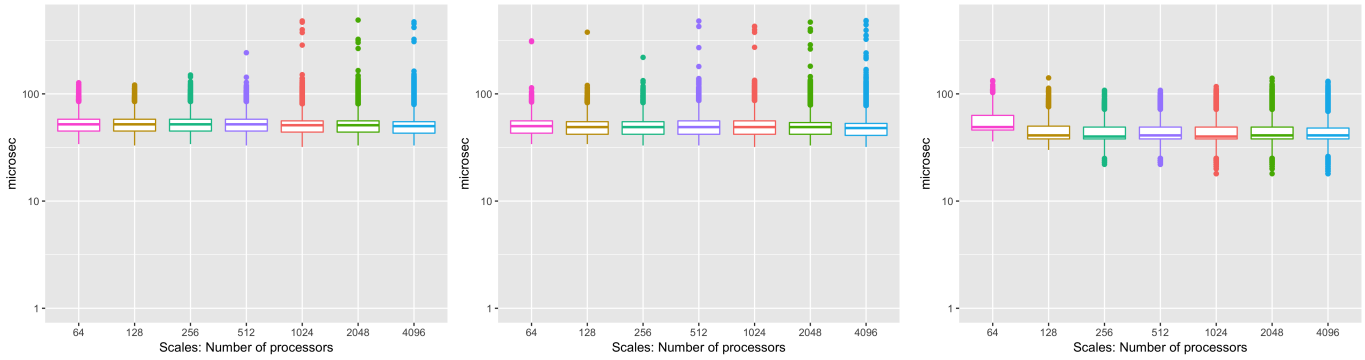
(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 8: Dataset open scaling: BDCATS. This operation performs small data reading from file system's view. Storage system performance have little impact over dataset open. Burst-buffer's abnormal behavior should be investigated further.



(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 9: Dataset close scaling: VPIC. Dataset close is a simple function, only needs to release some local resource and has no collective operations.



(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 10: Dataset close scaling: BDCATS

2) Group create/open/close: We notice that group_open takes slightly longer median time to finish and has much wider variance than does group_create (figure 11 and 12), this seems to be counter-intuitive because the collective group_create needs to sync up with all processes. We found that an internal metadata collective read caching mechanism is not activated by default. With the optimization enabled, group_open and file_open can reach the same performance on Lustre as it on Burst-buffer. (We didn't run full scale test for this due to time limitations.)

3) File create/open/close: File level metadata performance is fairly straightforward. More Lustre OSTs and burst buffer nodes both help to reduce the mean time and operation variance (figure 15 and 16).

V. Related work

Darshan [13] is a well-established application-level I/O characterization tool, that captures applications' I/O behavior at large scales on production systems. Darshan provides light-weight I/O tracing, and has been deployed on many production supercomputer systems. In contrast to the H5Prov connector, Darshan sits between storage system and application software stack, and focuses on very low-level information, such as recording read, write, data volume, timestamp and so on. It's not aware of the I/O semantics to the application and so users need to make sense from the trace in a statistical manner, and "guess" the relationship between two I/O events.

TOKIO [14]is a framework for holistic characterization and analysis of I/O workloads on HPC systems. The implementation of TOKIO is PyTOKIO [15]. TOKIO provides an abstraction layer between component-level monitoring tools already deployed on HPC platforms (such as topology information from Slurm and Cray SDB, application I/O from Darshan, file system load information from LMT) and higher-level I/O analysis tools that utilize this data. TOKIO plays the role of an aggregator that receives traces from various sources on different levels and outputs a organized and unified data for further analysis. TOKIO tracks many aspects of a production system, but it lacks of the semantics application I/O that H5Prov can gather.

IOMiner [16] is an I/O log analysis framework. IOMiner provides an interface for analyzing trace or log data, a unified storage schema that hides the heterogeneity of the raw instrumentation data, and a sweep-line-based algorithm for root cause analysis of poor application I/O performance. IOMiner can be used to analyze H5Prov provenance data.

VI. Conclusion

In this paper, we describe the design, implementation and application of H5Prov, a new provenance virtual object layer (VOL) connector for HDF5. We have shown that H5Prov trace is a powerful tool to detect application I/O patterns and unusual storage behaviors. With large scale experiments, we have verified that H5Prov is lightweight, scalable and extensible, and is capable of revealing many I/O issues that are difficult to find with other tracing tools.

Even without the knowledge of the HDF5 library implementation, H5Prov can provide us useful insights about application I/O. It is worth noting that the sample data we analyzed only covers function names and the duration of the function calls for simplicity, but this is just a small subset of what H5Prov can record. With all the data fields enabled and adding new features such as tracking collective I/O, we will be able to reveal more hidden facts about HDF5 applications' I/O behavior and use them to improve HDF5 and application performance.

VII. Future Work

Based on this work, we will collect provenance traces from a larger number of applications and will analyze the provenance data with data mining/machine learning techniques to find correlation and/or causation between I/O patterns and performance degradation. With the knowledge about these I/O pattern relationships and performance, we plan to optimize HDF5 operations, and the optimizations can be verified by benchmarks and real applications, using the provenance connector.

Acknowledgments

This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] "The HDF5 library and file format," https://www.hdfgroup.org/solutions/hdf5/, accessed: 2019-04-26.

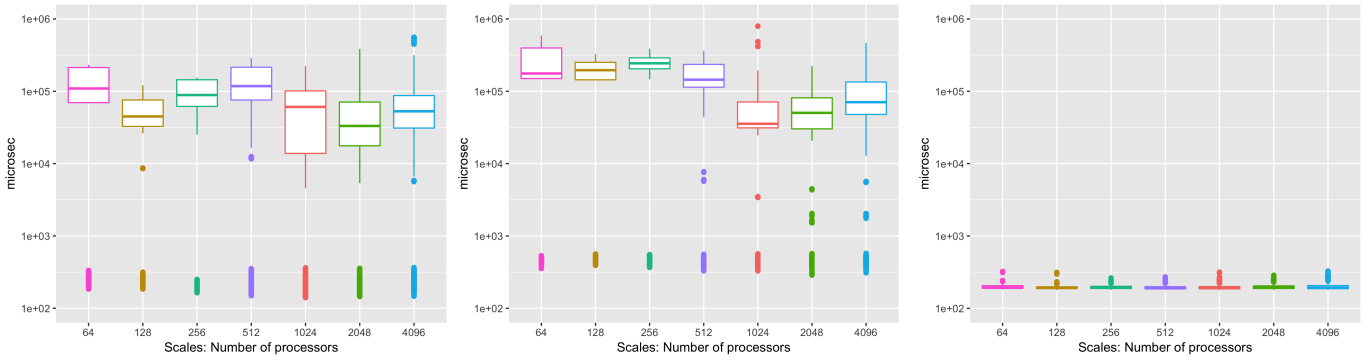
[2] "Automatic Library Tracking Database at NERSC," https://sdm.lbl.gov/exahdf5/papers/201810-HDF5-Usage.pdf, accessed: 2019-4-28.

[3] H. Tang, X. Zou, J. Jenkins, D. A. Boyuka, S. Ranshous, D. Kimpe, S. Klasky, and N. F. Samatova, "Improving read performance with online access pattern analysis and prefetching," in *European Conference on Parallel Processing*. Springer, Cham, 2014, pp. 246–257.

[4] H. Tang, S. Byna, S. Harenberg, X. Zou, W. Zhang, K. Wu, B. Dong, O. Rubel, K. Bouchard, S. Klasky *et al.*, "Usage pattern-driven dynamic data layout reorganization," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 356–365.

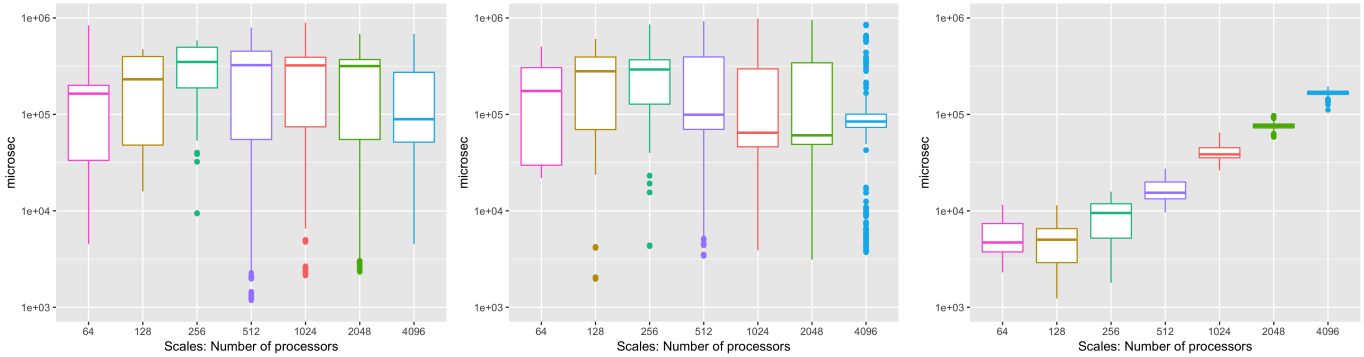
[5] H. Tang, S. Byna, S. Harenberg, W. Zhang, X. Zou, D. F. Martin, B. Dong, D. Devendran, K. Wu, D. Trebotich *et al.*, "In situ storage layout optimization for amr spatio-temporal read accesses," in *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016, pp. 406–415.

[6] "HDF5 VOL User Guide," https://support.hdfgroup.org/HDF5/doc/UG/HDF5_Users_Guide.pdf, accessed: 2019-1-18.



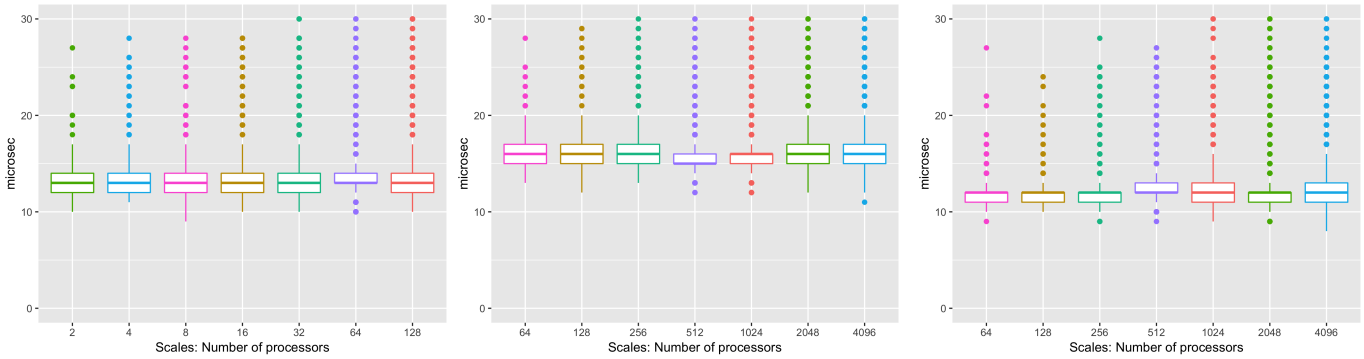
(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 11: Group create scaling: VPIC. Group create is a small but collective function, thus suffers from straggler operations. The burst buffer has much lower response time, so it eliminates the stragglers and speeds up the collective operation as a whole.



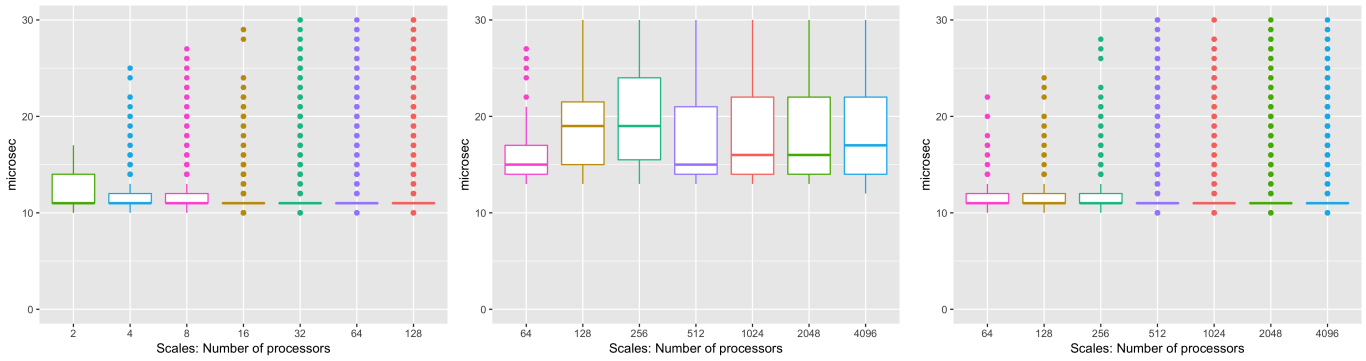
(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 12: Group open scaling: BDCATS. Similar to group create, group open gets benefits from Burst-buffer, but because the cache for collective metadata reading is disabled by default, it shows a moderate scalability issue.



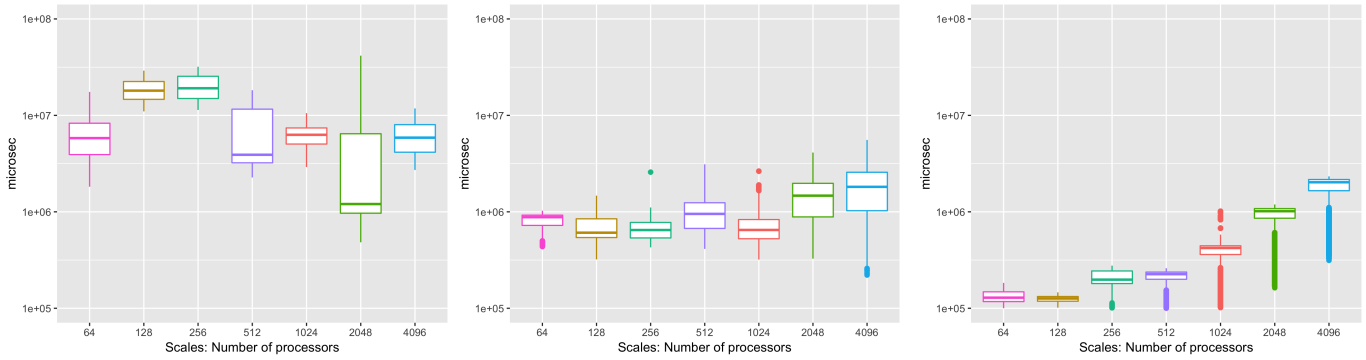
(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 13: Group close scaling: BDCATS



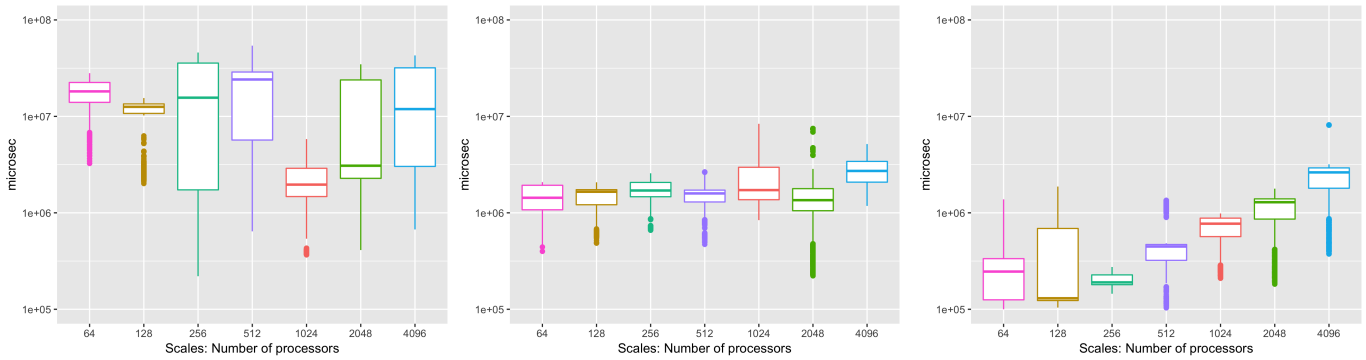
(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 14: Group close scaling: VPIC



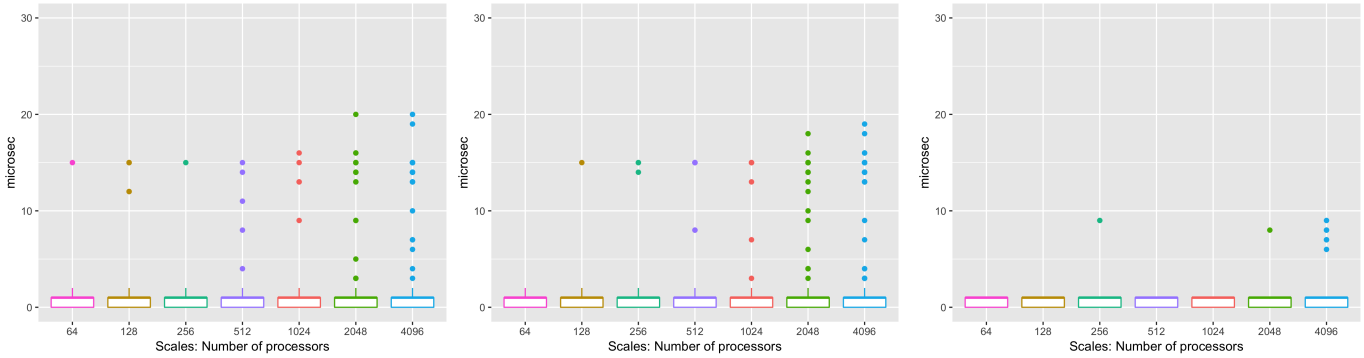
(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 15: File open scaling: BDCATS. In this function, one data file is opened in read-only mode by all processes. Note that this involves with a file system metadata request, and there are only 5 metadata servers (MDT) in Cori's Lustre setup, having more stripe count accesses more MDT servers, thus the workload is better distributed and shows lower mean time and variance.



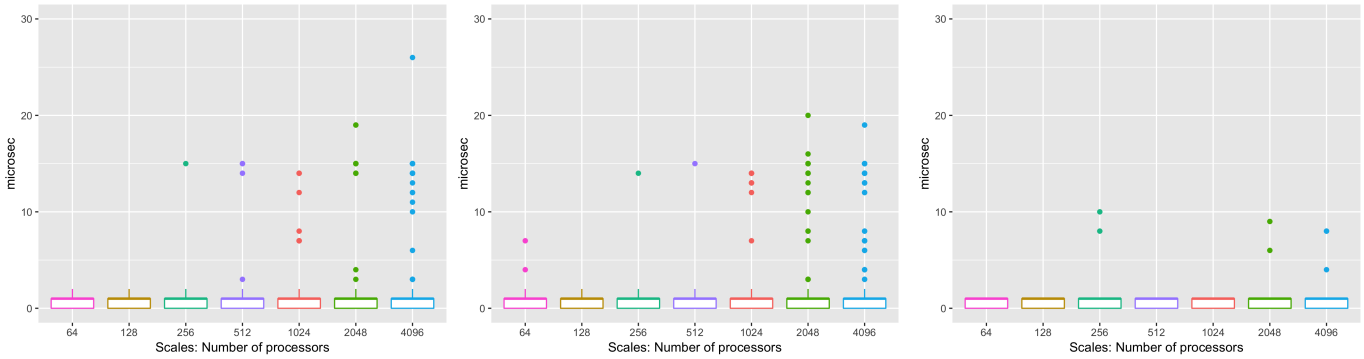
(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 16: File create scaling: VPIC. File create is similar to file open, except for a write mode, so it shows same pattern as file open does.



(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 17: File close scaling: VPIC



(a) Lustre: stripe count = 64 (b) Lustre: stripe count = 128 (c) Burst-buffer

Figure 18: File close scaling: BDCATS

[7] J. Liu, Q. Koziol, G. F. Butler, N. Fortner, M. Chaarawi, H. Tang, S. Byna, G. K. Lockwood, R. Cheema, K. A. Kallback-Rose, D. Hazen, and M. Prabhat, "Evaluation of HPC Application I/O on Object Storage Systems," in *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage Data Intensive Scalable Computing Systems (PDSW-DISCS)*, Nov 2018, pp. 24–34.

[8] "ADIOS VOL," https://bitbucket.org/berkeleylab/exahdf5/src/master/vol_plugins/swift/, accessed: 2019-4-20.

[9] B. Dong, S. Byna, K. Wu, , H. Johansen, J. N. Johnson, and N. Keen, "Data Elevator: Low-Contention Data Movement in Hierarchical Storage System," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, Dec 2016, pp. 152–161.

[10] B. Dong, T. Wang, H. Tang, Q. Koziol, K. Wu, and S. Byna, "ARCHIE: Data Analysis Acceleration with Array Caching in Hierarchical Storage," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 211–220.

[11] "Cori the supercomputer at NERSC," http://www.nersc.gov/users/computational-systems/cori, accessed: 2018-10-9.

[12] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, "Taming Parallel I/O Complexity with Auto-tuning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 68:1–68:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503278

[13] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access Through Continuous Characterization," *ACM Trans. Storage*, vol. 7, no. 3, pp. 8:1–8:26, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/2027066.2027068

[14] G. K. Lockwood, S. Snyder, T. Wang, S. Byna, P. Carns, and N. J. Wright, "A Year in the Life of a Parallel File System," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 74:1–74:13. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291656.3291755

[15] G. Lockwood, N. Wright, S. Snyder, P. Carns, G. Brown, and K. Harms, "TOKIO on ClusterStor: Connecting Standard Tools to Enable Holistic I/O Performance Analysis," https://escholarship.org/uc/item/8j14j182, accessed:

2019-4-18.

[16] T. Wang, S. Snyder, G. Lockwood, P. Carns, N. Wright, and S. Byna, "IOMiner: Large-Scale Analytics Framework for Gaining Knowledge from I/O Logs," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 466–476.