2012 IEEE International Conference on Cluster Computing

The Power and Challenges of Transformative I/O

Adam Manzanares*, John Bent†, Meghan Wingate*, and Garth Gibson‡
*Los Alamos National Laboratory
Email: nmtadam@gmail.com,meghan@lanl.gov
†EMC Corporation
Email: John.Bent@emc.com
‡Carnegie Mellon University
Email: garth@cs.cmu.edu

Abstract

Extracting high data bandwidth and metadata rates from parallel file systems is notoriously difficult. User workloads almost never achieve the performance of synthetic benchmarks. The reason for this is that real-world applications are not as well-aligned, well-tuned, or consistent as are synthetic benchmarks. There are at least three possible ways to address this challenge: modification of the real-world workloads, modification of the underlying parallel file systems, or reorganization of the real-world workloads using transformative middleware. In this paper, we demonstrate that transformative middleware is applicable across a large set of high performance computing workloads and is portable across the three major parallel file systems in use today. We also demonstrate that our transformative middleware layer is capable of improving the write, read, and metadata performance of I/O workloads by up to 150x, 10x, and 17x respectively, on workloads with processor counts of up to 65,536.

I. Introduction

Long running applications on large-scale compute clusters typically protect themselves from inevitable node failures by periodically writing out checkpoints to parallel file systems [1]. Extracting high performance from the current generation of parallel file systems with petascale class supercomputers is a formidable challenge and exascale class compute platforms will only make this problem more challenging [2]. Petascale class machines currently have node counts in the tens of thousands with hundreds of thousands of processors. Exascale class compute platforms are anticipated to have node counts in the hundreds of thousands with millions of processors. Larger node counts produce larger checkpoint data sets, while at the same time requiring more frequent checkpoints due to the increased frequency of node failures. These requirements drive application checkpoint bandwidth demands ever higher.

Performance degradation in checkpoint-restart workloads is common due to the high concurrency imposed by these massively parallel, tightly coupled workloads generated from applications which write checkpoints in a bulk-synchronous manner. As we will see, this high degree of concurrency introduces bottlenecks for write bandwidth, read bandwidth, and metadata rates. These bottlenecks prevent efficient utilization of the storage hardware and thereby cause significant performance degradation. In this paper we show that a reorganization of the I/O patterns can remove these bottlenecks.

We recognize three ways to reorganize I/O patterns: (1) modify existing parallel file systems (2) modify applications (3) interpose a layer of *transformative middleware*. We define transformative middleware as a layer of indirection preserving the user's view of a file, while transforming the I/O into a manageable workload for the parallel file system.

Existing parallel file systems are very large software projects that must support general use cases and ensure data integrity. In many cases, they support full POSIX compliance, which is not necessary for HPC checkpoint workloads. Relaxation of POSIX semantics or reorganization of I/O workloads within these parallel file systems can negatively impact the entire range of their supported workloads. Transformative middleware, on the other hand, can be selectively applied to only those workloads which need it. Additionally, a layered approach is easier to implement and to debug since it serves a narrow focus and is fewer lines of code (LOC). For example, our implementation of transformative middleware, the Parallel Log-structured File System (PLFS) [3], is around ten thousand LOC, whereas all five major parallel file systems Ceph [4], GPFS [5], Lustre [6], PanFS [7], and PVFS [8], are at least 200,000 LOC. Lastly, our cluster compute resources, which includes cluster compute nodes and their high-speed network interconnect, are largely idle during I/O phases of computation. Parallel file systems are typically separated from the cluster compute resources and the high-speed interconnect network. Cluster compute resources move data to the parallel storage system using a dedicated storage network, which is typically much slower than the cluster's high-speed interconnect network. A middleware layer, residing on cluster compute nodes, has the ability to leverage resources of the cluster during I/O phases, an ability the storage sub-system lacks.

Application level changes are prohibitive because every application would need to be tuned for a particular parallel file system. Reorganizing I/O patterns requires intimate knowledge of the complex internal architectures of parallel file systems; computational scientists do not wish to become parallel file system experts nor should they. A transformative, transparent (to application developers), middleware layer can be written once for a class of applications and compute platforms in-

978-0-7695-4807-4/12 $26.00 © 2012 IEEE
DOI 10.1109/CLUSTER.2012.86

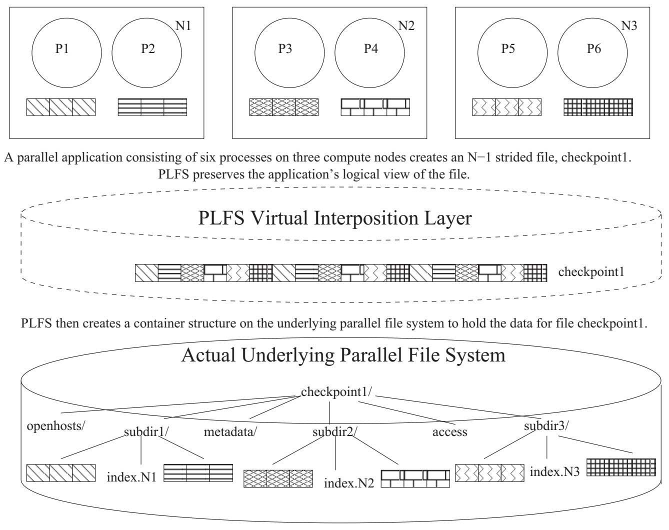

N1
P1
P2
N2
P3
P4
N3
P5
P6
A parallel application consisting of six processes on three compute nodes creates an N−1 strided file, checkpoint1.
PLFS preserves the application's logical view of the file.
PLFS Virtual Interposition Layer
checkpoint1
PLFS then creates a container structure on the underlying parallel file system to hold the data for file checkpoint1.
Actual Underlying Parallel File System
checkpoint1/
openhosts/
subdir1/
metadata/
subdir2/
access
subdir3/
index.N1
index.N2
index.N3

Fig. 1: **PLFS Architecture** *PLFS maps the users logical file into a container structure on the underlying parallel file system.*

creasing the productivity of computational scientists. Many applications also use data formatting libraries, such as HDF5 and pNetCDF, which dictate the I/O access patterns of the application [9], [10] and we are able to intercept the I/O calls of these data formatting libraries. Finally, changes to mission critical applications must be carefully vetted, a process that can be extremely time consuming. Since a transformative middleware approach can be transparent to the application the limitations of application level changes are eliminated when transformative middleware is used.

In this paper, we explore the transformative middleware approach to improving real-world application's I/O performance with an implementation of one such middleware layer, PLFS. In our evaluation, we show that this layer of transformative middleware achieves speedups in the write, read, and metadata performance of I/O workloads by up to 150x, 10x, and 17x respectively, on process counts of up 65,536. The rest of the paper is organized as follows: Section II gives an overview of the architecture of PLFS and how it relates to current I/O challenges. Section III demonstrates the write performance of PLFS across a set of benchmarks. Section IV details read performance challenges unique to PLFS, solutions to these challenges, and an evaluation of the read performance of PLFS with these enhancements. Section V details enhancements of PLFS designed to improve the metadata performance of I/O workloads along with an evaluation of their performance. An evaluation of large-scale read and metadata performance is presented in Section VI. The related work is presented in Section VII and the conclusions and future work are presented in Section VIII.

II. Background

Current applications running on large-scale supercomputers, currently in the petascale era, present challenging patterns of I/O to parallel file systems. Due to the high rate of failures on such large machines, applications use checkpointing to ensure forward progress. Because these applications are very tightly coupled, the checkpointing workload is necessarily synchronous: each of the processes in the cooperative job concurrently saves its state into a logically related object in the file system. There are two typical I/O patterns that applications use to checkpoint data: *N-1* in which the shared object in the file system is a single file into which all the processes concurrently write, and *N-N* in which the shared object is typically a single directory into which each of the processes writes a unique file.

Our previous work has established that three major parallel file systems, GPFS, Lustre, and PanFS, all achieve up to two orders of magnitude higher bandwidth for *N-N* workloads than for *N-1* workloads [3]. PLFS is a layer of transformative middleware that we developed at the Los Alamos National Laboratory (LANL), to convert logical *N-1* workloads into physical *N-N* workloads in order to bridge the performance gap between these two I/O patterns.

PLFS is middleware that sits between the user application and a file system that is parallel in our case and is subsequently referred to as the *underlying parallel file system* . As illustrated in Figure 1, PLFS creates a *container* on the underlying parallel file system that is a physical directory sharing the same name as the logical file stored into PLFS. The logical view of the file matches the I/O access pattern of the application. Within the container, there are multiple files and directories: an *access* file holds ownership and access control information, a *metadir* caches information about the logical file size, an *openhost* directory contains information about any process that currently has a PLFS file open for write access, and finally there are *subdirs* that contain the process specific *data* and *index* log files.

The log files are the key contribution of PLFS which allow it to transform *N-1* into *N-N*. Each process writing to the shared logical file is redirected *transparently* to write to a unique physical file within one of the container's subdirs. In addition to decoupling the concurrency by having each process write to a unique physical file, PLFS also transforms random I/O into sequential by always appending to the data log file regardless of the physical offset to which the data is intended. In order to maintain the users logical view of the file, PLFS also appends a record of each write to a unique index file for each process.

The major difference between PLFS and current parallel file systems is that current parallel file systems do a large amount of work during the write phase of file I/O to place logical bytes deterministically. Primarily they do so for two main reasons: one, to serialize writes to the same offsets, and two, to quickly lookup data offsets for reads. In contrast, PLFS uses timestamps to resolve writes to the same offsets and defers the work of creating offset maps until the file is opened for read access[1]. In effect, PLFS has deferred the resolution of the file from the write phase to the read phase. This deferral introduces several complications for large-scale I/O workloads and we have investigated and developed solutions to these complications in this work.

Currently there are two methods for a user to access a PLFS virtual file system, a mount point on the system that relies on FUSE [11] to intercept I/O calls, or by directly linking the PLFS library into the user application. The PLFS mount accessed through FUSE is the most transparent method for a user to gain the benefits of PLFS because they need only to place their files in the PLFS mount point.

To support our analysis of read workloads for this paper, we have added a third interface to PLFS by implementing a PLFS driver within the MPI-IO library [12]. MPI provides an abstract device interface, *ADIO* [13], that we leverage to reroute I/O calls to the PLFS library. The MPI-IO layer also allows us to inherit communicators and job info which enables us to augment PLFS with several key collective optimizations as demonstrated in Section IV.

III. Write Performance

PLFS was originally designed to speed up the write performance of checkpointing workloads performing file I/O using the *N-1* pattern. PLFS has proved to be very successful and a summary of the write performance speedups achieved by PLFS across a set of *N-1* workloads generated by several applications is shown in Figure 2. By decoupling a file into non-shared component pieces PLFS eliminated serializations on the underlying parallel file system that were responsible for the large performance discrepancy between *N-1* and *N-N* I/O workloads. PLFS achieved these speedups while maintaining the logical view of the file.

By transparently transforming the I/O workload to boost performance, PLFS remains applicable to a wide set of checkpoint applications. This performance improvement was also demonstrated on three major parallel file systems, which all demonstrated serializations when data was written to these respective file systems using the *N-1* I/O pattern. PLFS was designed for high write performance on checkpointing workloads

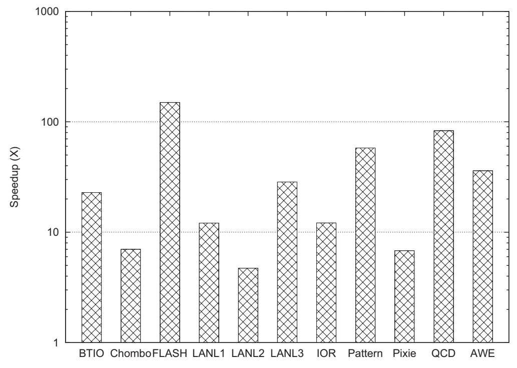


Fig. 2: **Summary of write performance results.** *This graph demonstrates the write performance improvement PLFS achieves for N-1 workloads on a set of applications.*

and showed high read performance for smaller scale reads, but an initial push to place PLFS into production revealed some limitations when high I/O concurrency is used during a read of a checkpoint file written with a high level of I/O concurrency. In this work we focus on two important aspects of end to end I/O performance, read and meta data performance, and demonstrate our solutions at high levels of I/O concurrency.

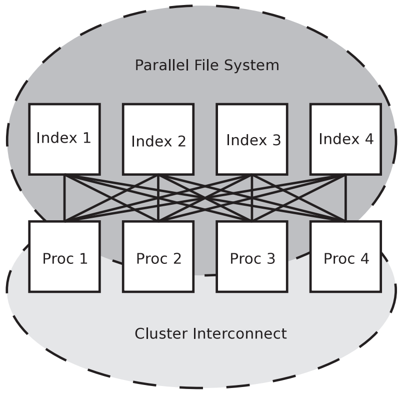
IV. Read Performance

PLFS, a transformative middleware layer, is write optimized so one should anticipate that the read performance would suffer. For PLFS files written and read with lower process counts, the read performance is surprisingly strong, but for restarting jobs with large process counts using PLFS, we discovered that the *effective read bandwidth*[2] of PLFS was diminishing with job size. As described in Section II, each process requires index information from every single process that wrote a checkpoint file. The original design of PLFS kept processes uncoordinated when a file was written or read.

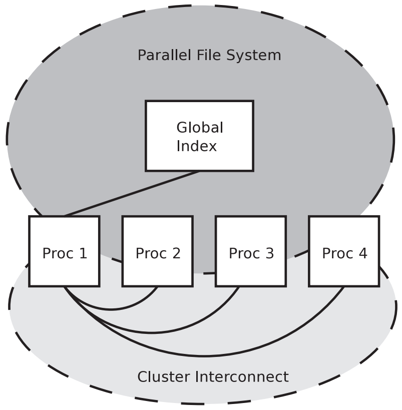
If a PLFS file is written and then read by N processes, this workload will require N^2 opens from the underlying parallel file system. N reader processes are all attempting to open and read N index files concurrently. With N in the tens of thousands, this is an extreme demand on the metadata servers of the underlying parallel file system and uncoordinated read access of these files on the underlying parallel file system were the cause of diminishing effective read bandwidth for large scale restart jobs.

The two solutions[3] we developed to mitigate the read issues presented by the write optimized design of PLFS are to:

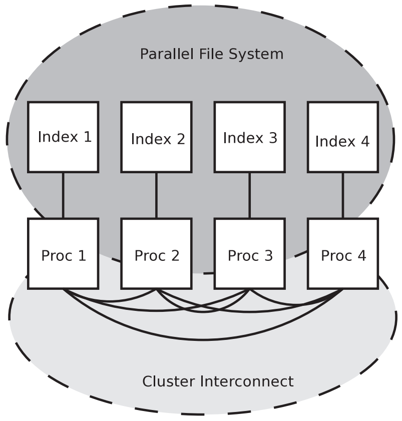
- Aggregate the global index on the write close and on read open broadcast the results of the aggregation from one process to all processes (Index Flatten)
- Aggregate the global index on read open leveraging all processes (Parallel Index Read).



(a) Original PLFS Design



(b) Index Flatten



(c) Parallel Index Read

Fig. 3: **Index Aggregation Techniques** *The figures above represent the I/O workloads generated on the parallel file system and cluster interconnect by PLFS during a file open using the original design and our two solutions to improve the read bandwidth of PLFS.*

It is important to note that our HPC cluster computer platforms have high-speed cluster interconnect networks and compute nodes that are largely idle during checkpoint I/O. The techniques we developed to improve the read performance of PLFS share the design goal of leveraging these idle resources in order to minimize the amount of I/O requests that must be served by the underlying parallel file system. We transform the read I/O workload of PLFS, such that redundant I/O requests to the underlying parallel file system are eliminated.

A. Index Flatten

The Index Flatten solution begins once a file is being written and has every single write process buffer index information up to a threshold. If the per-process index ends up being less than or equal to the threshold on all write processes, then we aggregate this information when the file is closed. Each process sends their buffered index to a root process, which writes the aggregation, *global index*, of the index information from all processes to the underlying parallel file system. When a read job is started a root process reads the global index and broadcasts the contents of the global index to every process. This solution is represented by Figure 3b, and it is important to note that index aggregation takes place during the close of a newly written file.

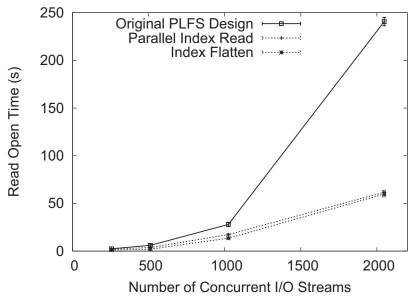
During the read of a PLFS file, this solution results in one file open on the underlying parallel file system the global index, dramatically reducing concurrent access to the index files. This approach has the limitation that it may degrade write performance. Index aggregation is performed when a newly written file is closed, increasing the close time, which lowers the effective write bandwidth of PLFS. If a user can tolerate an impact to the write performance of a file to improve read performance than they should select this approach. For example, a user could use the Index Flatten technique if they plan on writing a file once, but reading it back many times.

B. Parallel Index Read

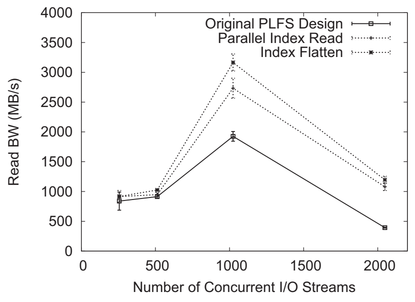
Although the Index Flatten technique is successful at reducing concurrent access to index files, write performance may suffer. Since PLFS is a write optimized file system, we developed a technique that would attempt to match the performance of the Index Flatten approach without impacting write performance. The resulting Parallel Index Read technique is represented by Figure 3c. This technique has one process assign work to groups of processes. Each group of processes has a group leader who assigns work to members of the group. Each process within a group reads its assigned subindices and returns information with the subindices to its respective group leader. The group leaders aggregate indices within their group and then exchange this information with the other group leaders. After the group leaders receive all of the indexing information from all other group leaders they merge the group leader results into a global index and then broadcast the global index to every process in their group. This technique lowers the numbers of opens on the underlying parallel file system to N and performs index aggregation when a PLFS file is opened for read access. An *MPI_ALL_TO_ALL* could deliver the data within the index files to all processes, but our approach transforms the data at various levels of the collective hierarchy, which is not possible with the *MPI_ALL_TO_ALL* call.

C. I/O Benchmark Performance Analysis

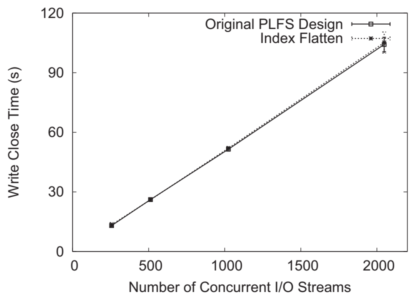
All of the results in Figure 4 were collected using *MPI-IO Test*, a tunable synthetic I/O workload generator developed at Los Alamos [14]. These results were collected on a production cluster at Los Alamos. The cluster has 64 nodes each with 16 AMD Opteron cores for a total of 1024 processors. Each node has 32GB of memory and nodes are interconnected with an Infiniband network. The cluster is also connected to a 551 TB Panasas file system through a 10GigE storage network. Each data point is an average of 10 runs and we have included error



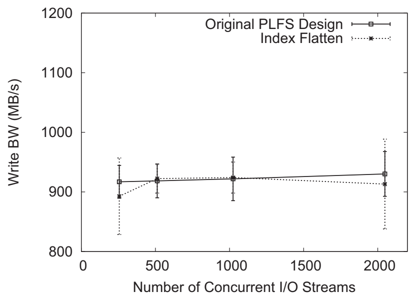
(a) Read Open Time



(b) Read Bandwidth



(c) Write Close Time



(d) Write Bandwidth

Fig. 4: **Read Scaling Issues** *These set of graphs compare the original design of PLFS against the collective techniques that were developed to improve the read performance of PLFS.*

bars representing the standard deviation. Each concurrent I/O stream writes/reads 50 MB in 50Kb increments.

Figure 4a compares the *Read Open Time* of the Original PLFS Design versus the two techniques we developed for this paper with increasing numbers of concurrent I/O streams. The Read Open Time represents the time to aggregate the indices within the PLFS container into a global index. Note: the global index needs is already aggregated with the Index Flatten approach, it does need to be broadcast to all processes though. From this figure it is clear that the Parallel Index Read technique and the Index Flatten technique scale much better than the Original PLFS Design. We attribute this speed up to the coordinated access to the parallel file system that both provide. At process counts of 2048, both of the techniques are roughly 4x faster than the Original Design in terms of index aggregation. There are two phases of index aggregation where this coordination proves to be beneficial. First, we reduce the number of file opens, which reduces the metadata workload on the underlying parallel file system. Second, we also coordinate read access to the indices, which reduces the amount of seek operations on the underlying parallel file system.

In Figure 4b we compare the read bandwidth of the index optimization techniques we have developed against the Original PLFS Design. Our definition of read bandwidth includes the time to open and close the file. Since we demonstrated that the time to open a file for read access varies with these approaches in Figure 4a, we have chosen to compare the read bandwidth of all of these approaches. For smaller scale workloads the read bandwidth is nearly identical for all of the approaches. At 1024 concurrent I/O streams we see a divergence in read bandwidth performance with the Index Flatten technique providing the highest bandwidth. The Index Flatten technique needs to only distribute the global index that was generated when the file was closed after it was written. In contrast, the Parallel Index technique must collect all of the information that is spread throughout the indices within the container when a file is opened for read access. Another observation that we have noticed is that caching is a factor when there are 1024 concurrent I/O streams. Our theoretical peak bandwidth is 1.25 GB/s which is exceeded with all of the approaches. We still see that the Parallel Index technique and the Index Flatten technique provide much higher read bandwidth as compared to the Original PLFS Design. At 2048 concurrent I/O streams the performance of our index aggregation optimizations becomes closer, but they share a large performance improvement over the Original PLFS Design, around 3x faster in terms of Read Bandwidth.

Figures 4c and 4d illustrate the performance implications of the Index Flatten technique. We have omitted results from the Parallel Index Read approach because it shares the same behavior of the Original PLFS Design when a file is closed. The time to close the file after it has been written is shown in Figure 4c. The performance is roughly the same until we have 2048 concurrent writers where the Index Flatten technique has a slightly higher close time and more variance in the results. Figure 4d compares the write bandwidth of the two approaches and we see that the Index Flatten technique has lower bandwidth with much greater variance at 256 and 2048 concurrent writers. At 256 concurrent writers the Write Close Time of Index Flatten is slightly lower, but the time to write the file is also low leading to a noticeably lower writer bandwidth. At 2048 concurrent I/O streams the write bandwidth is lower and we also see that the variance of the results is much higher. The Index Flatten technique must aggregate index information from multiple processes, whereas the Original PLFS Design and the Parallel Index Read technique simply close the file.

D. I/O Kernel Performance Analysis

In addition to testing with a synthetic I/O benchmark (IOR), we also investigate the read performance of PLFS across a diverse set of I/O kernels derived from five applications that perform *N-1* I/O. LANL 1 and LANL 3 are I/O kernels derived from applications developed at LANL that are run regularly on our largest supercomputers, consuming between them a large portion of the computational cycles at LANL. All of these results are an average of 10 runs and were collected on the same cluster that was used in Section IV. These results were collected using a parallel file system that we did not have exclusive access to producing some variance across our results.

PLFS can boost read performance in two key ways; spreading the I/O workload over many storage resources and by improving pre-fetch opportunities for the underlying parallel file system. Since PLFS decouples writers we know that a shared file written by PLFS is able to engage more spindles in the underlying parallel file system as compared to direct access to the underlying parallel file system. If the read pattern matches the write pattern it follows that the read of the PLFS file will also engage more storage resources. The second benefit that PLFS presents on a read is the fact that the decoupled read streams make prefetching on the underlying parallel file system easier. In the case in which the read pattern matches the write pattern we know that each process will read data sequentially from one *log-structured* PLFS data file within the container, which is a pattern that is conducive to prefetching on the underlying parallel file system. If the same read pattern was read directly from a file on the underlying parallel file system then multiple readers would be requesting data from multiple offsets in the same file reducing prefetching opportunities.

PLFS can also negatively impact read performance because direct access to the underlying parallel file system does not need to complete index aggregation. Our solutions have driven the index aggregation time lower, but it is not a negligible amount of time. If any application reads a small amount of data from a file, the index aggregation time could dwarf the time spent reading the data. This will result in a lower effective read bandwidth, which is an important metric from the user standpoint. Although PLFS has this disadvantage during reads, in our presented benchmarking results PLFS was able to deliver read bandwidth that matched or exceeded direct access to the underlying parallel file system. All of the following results were collected with the Parallel Index Read aggregation technique, which we have chosen as the default behavior of PLFS, due to its relatively high read performance and the fact that index aggregation is conducted when the file is opened for read access.

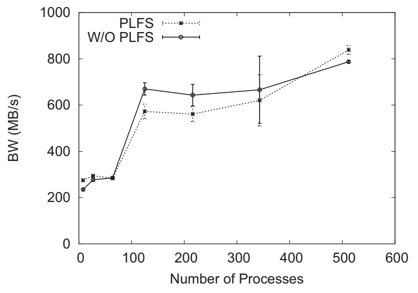
1) Pixie 3D: The Pixie 3D benchmark is an I/O kernel derived from the Pixie 3D MHD (Magneto Hydro-Dynamic) code [15], is widely used for parallel I/O benchmarking, and does I/O through the Parallel-NetCDF [10] library. We compared the performance of PLFS to a direct read from the underlying parallel file system, (PanFS), using a large data size (1GB per process) in which all of the Pixie processes read from a shared file. In Figure 5a we observe that the direct access to the underlying parallel file system outperforms PLFS for smaller process counts, but the read performance of PLFS scales better and outperforms direct access to the underlying parallel file system for larger process counts. In all cases the performance of PLFS relative to direct access to the underlying parallel file system is extremely close.

2) ARAMCO: The Saudi ARAMCO I/O kernel represents a seismic processing application and uses MPI-IO and HDF5 [12], [9]. For process counts lower than 300, PLFS is able to improve the read performance of the Saudi ARAMCO kernel by up to 8X. However, as the job size grows, direct access to the underlying parallel file system begins to outperform PLFS. This is because the ARAMCO code is a strong scaling application which writes the same total amount of data regardless of processor count. In other words, index aggregation time will rise as the time spent reading the data decreases.

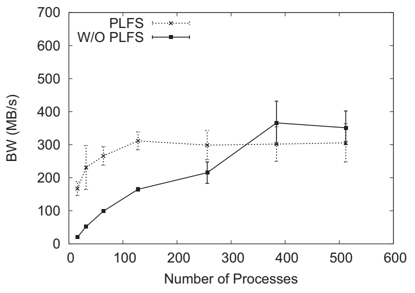
3) IOR: The IOR benchmark [16] is a synthetic parallel I/O benchmark developed at Lawrence Livermore National Laboratory. We configured it to write and read to a shared file (*i*.e. N-1). Each process accessed 50MB in 1MB increments. In this benchmark PLFS outperforms direct access to the underlying parallel file system for all process counts by up to 4.5X. Note that we did have to modify IOR to remove read-write mode on it's open. PLFS does not support read-write access to files accessed by multiple processes at the same time.

4) MADBENCH: The MadBench parallel I/O benchmark [17], a cosmic microwave background radion simulation, was developed at Lawrence Berkeley National Laboratory and is based on the MADspec code. The benchmark by default runs the entire simulation, but we ran only the I/O phase that writes out a file and the phase that reads it back in its entirety. As we did for IOR, we modified it to convert all opens in read-write mode to opens in read-only mode. In Figure 5d, we again observe improved read performance when using PLFS.

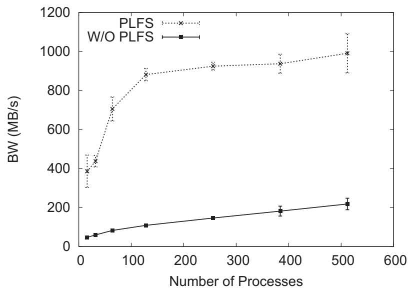
5) LANL 1: LANL 1 is an I/O kernel representing a mission critical scientific code developed at LANL and is a weak scaling application. We ran the benchmark such that each individual write and read were in five hundred thousand byte increments (approximately 500K). Since this application also writes data in an N-1 strided fashion the performance improvements are due to better distributions of data across file



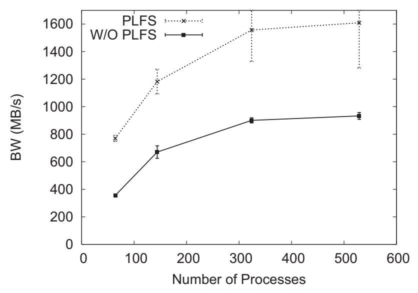
(a) Pixie 3d



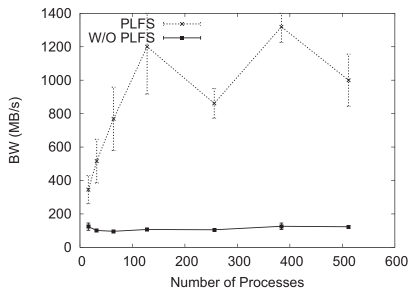
(b) Aramco Kernel



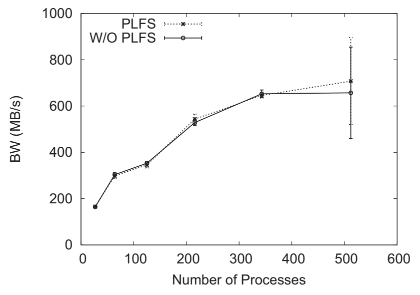
(c) IOR



(d) Madbench



(e) LANL 1



(f) LANL 3

Fig. 5: **Experimental Results.** *This collection of graphs represents the read performance of PLFS across a diverse set of I/O kernels derived from five applications and one synthetic benchmark (IOR).*

system resources and increased opportunities for prefetching. This leads to increased read performance for all process counts with a maximum speedup of 10X when 384 processes are used.

6) LANL 3: LANL 3 is an I/O kernel representing another important application in use at LANL. Using MPI-IO hints, we enabled collective buffering [18] for this application since by default it performs writes and reads in 1024 byte increments. This application performs strong scaling I/O to a shared file and the total amount of data written was 32 GB for this test. As seen in Figure 5f, the performance of PLFS is similar to the underlying parallel file system. The interesting observation in this graph is the fact that PLFS slightly outperforms the underlying parallel file system for the largest scale. With collective buffering there are a fixed amount of nodes that write data with a fixed buffer size, so the index size stays the same with a strong scaling application using collective buffering. Since the index size has not grown any larger but we have improved opportunities for prefetching by decoupling the writes and reads we observe that PLFS outperforms the underlying parallel file system for the largest process counts.

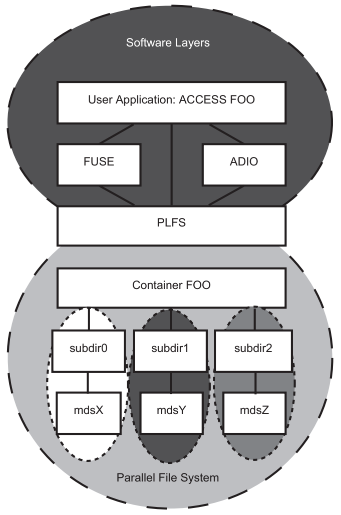
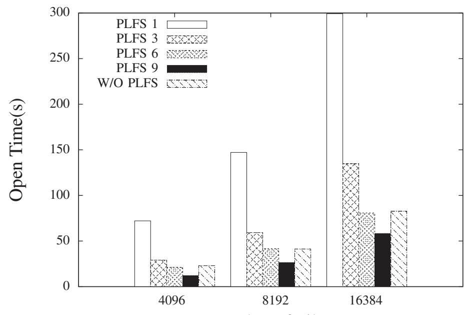


Fig. 6: **PLFS Federated Metadata Servers** *This figure highlights how the architecture of PLFS allows us to spread the subdirs within PLFS containers across different namespaces managed by multiple metadata servers.*

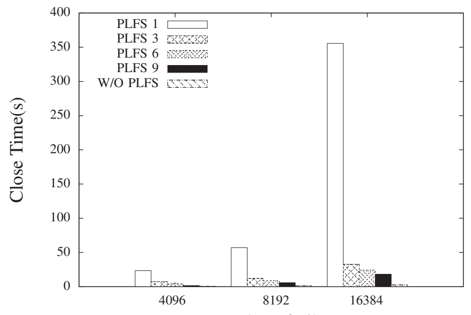
V. Meta Data Performance

As we have discussed, large-scale *N-1* workloads suffer poor performance from parallel file systems due to bottlenecks incurred while N processes concurrently write to a single shared object. On the other hand, *N-N* workloads achieve much higher performance because they write to unique objects. This in fact is how PLFS achieves its speedups: by transforming *N-1* workloads into *N-N*.

However, *N-N* workloads are themselves not entirely free from concurrency bottlenecks. Although their write phase is, their create phase is not. Interestingly, the create phase of an



(a) N-N Open Time



(b) N-N Close Time

Fig. 7: **Metadata Performance Graphs** *For these set of graphs we compare the metadata performance of PLFS with varying numbers of metadata servers.*

N-N workload is very similar to the write phase on an *N-1* workload: massive concurrent writes to a shared object, which is a file in the case of *N-1* writes and a directory in the case of *N-N* creates. The obvious solution to this problem is to use multiple metadata servers and to decouple the concurrency similar to the approach in GIGA+ [19]. GIGA+ demonstrated that large-scale concurrent access to a single directory is not scalable. We differ from GIGA+ in the fact that we use a static technique to spread the I/O workload over a set of directories.

Although distributed metadata servers have long been discussed [20], the reality is that current production parallel file systems do not transparently spread workloads across multiple metadata domains. PanFS does have multiple metadata servers but the division between them is rigid; the realm of each must be mounted as a separate file system so there is no ability to spread the workload within any one directory across multiple metadata servers.

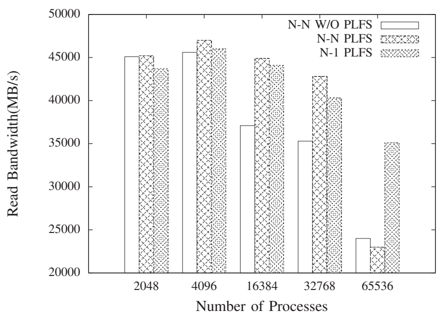
To address this limitation, we augmented PLFS with *federated metadata management* in two ways. The first technique distributes entire PLFS containers across a set of metadata servers whereas the second distributes the subdirs within the containers across metadata servers. The second technique is represented by Figure 6. Both techniques aim to address the metadata bottleneck in large-scale *N-N* workloads: the first for application generated *N-N* workloads and the second for the physical *N-N* workloads created by PLFS's transformation of logical *N-1* workloads. Federated metadata management can be used to glue multiple file systems together and reduce the concurrent access to a single directory, which is applicable to many file systems.

Our first set of results were collected with the previous cluster and we had each process open and close multiple files within separate threads. This approximates a large-scale *N-N* job and our results for these tests are presented in Figure 7. We have chosen to only examine *N-N* workloads because they represent the heaviest metadata load that PLFS can generate, since each file create requires container creation. These two graphs represent the open and close time, respectively, for a simulated large-scale *N-N* job. The open and close phases of the file operations are where the metadata workload is heaviest and all of our reported open times include file creation times.

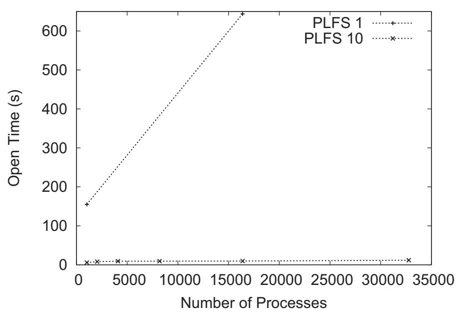
In Figure 7a we see that the open time for files is improved with increasing metadata servers (MDS). We expect to see lower open times as the number of meta data servers is increased because we are spreading the workload across more hardware. When we compare the *N-N* performance of PLFS against the underlying parallel file system we see that PLFS with six and nine MDS outperforms direct access to the underlying parallel file system. Note that in our results PLFS-*X* represents PLFS configured with *X* MDS. Although PLFS has the burden of container creation for each file, this is offset by the performance benefits of federated metadata servers. Figure 7b presents the results of increasing MDS and its impact on the close time of files. As we increase the amount of MDS our close time goes down as expected, but we are no longer able to outperform direct access to the underlying parallel file system. Closing a file is a lightweight task as compared to file creation and this is reflected in the fact that the underlying parallel file system outperforms PLFS in all cases.

VI. Large Scale Results

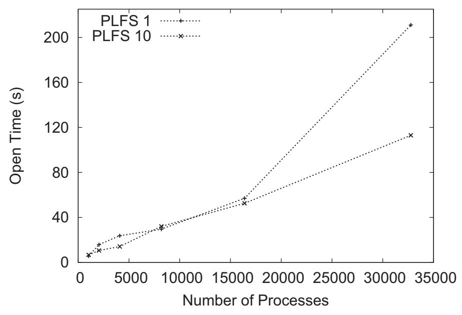
Although all of our previous results have demonstrated the performance improvements of our index aggregation optimizations and strategies to boost metadata performance, we have decided to included large-scale test results to further validate our techniques. The results of large-scale testing are demonstrated in Figure 8 and we have included tests that focus on the read and metadata performance of the approaches we have introduced in this work. We do not include any write performance numbers because PLFS has proven to have high write performance with large process counts. These results were collected on our latest large-scale supercomputer, Cielo, which is a Cray XE6 machine with 8894 nodes and 142,304 compute cores interconnected with a Cray Gemini network. Each node has 32 GB of memory and the cluster is connected to a 10PB Panasas parallel file system. Cielo is currently #6 on the top 500 list of supercomputer sites, achieving just over a



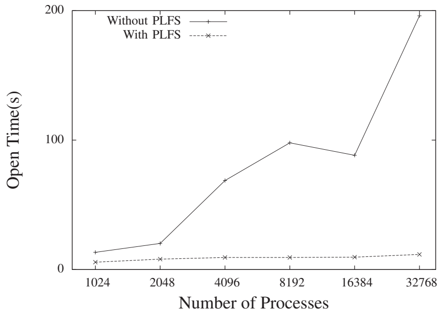
(a) Large Scale Read Performance



(b) Large N-N Open Time



(c) Large N-1 Open Time



(d) PLFS vs W/O PLFS Large-Scale

Fig. 8: **Large-scale Results** *For these set of graphs we ran large-scale tests on a production cluster at LANL.*

petaflop in sustained computing performance. Note that in all of these results each process resides on a non-shared compute core of our cluster.

Figure 8a demonstrates the read performance of PLFS and our underlying parallel file system using our synthetic I/O workload generator at large-scales of up to 65536 processors. In practice we see the best bandwidth from our underlying parallel file system with *N-N* workloads and have omitted *N-1* workloads that read directly from the underlying parallel file system. With all of these results we see that for *N-1* workloads PLFS achieves read performance that is close to or exceeds the underlying parallel file system. *N-1*workloads read with PLFS outperform the underlying parallel file system for all process counts except for 2048. *N-N* read workloads using PLFS are close to or exceed the underlying parallel file system across all process counts. These results were collected using the Parallel Index Read technique and 10 federated MDS.

To validate our metadata performance results for large-scale concurrent jobs we ran a set of experiments on Cielo using up to 32,768 processes. We evaluated PLFS federated metadata management with both *N-N* and *N-1* workloads. In each case, we compare the performance of PLFS using one, ten, and twenty MDS, denoted by PLFS-1, and PLFS-10, respectively. In Figure 8b we present the *N-N* write open time. As explained previously container creation for *N-N* workloads is metadata intensive and we see that PLFS using one MDS performs poorly. Increasing the MDS count to 10 improves open times significantly. In an *N-N* workload PLFS creates a container for each file, so it is important that we distributed these containers across multiple MDS.

The *N-1* write open results are presented in Figure 8c. For smaller scales there is not much of an improvement in the metadata performance when using multiple MDS. In this case we are only creating one PLFS container that all processes share, so a single MDS is not overwhelmed with file creation requests at small scale. As process count is increased we again observe that PLFS with 10 MDS outperforms PLFS with 1 metadata server.

A comparison of the performance of PLFS-10 to direct access to the underlying parallel file system is presented in Figure 8d. In this graph it is clear that federated meta data management within PLFS is capable of improving the metadata performance of *N-N* workloads. Direct access to the underlying parallel file system is only capable of using one metadata server and higher process counts require more file creations. PLFS with *federated metadata management* spreads the metadata workload across multiple directories boosting metadata performance. This leads to reduced open times when PLFS is used for all process counts with a maximum speed up of 17X when the workload is ran with 32768 processors.

VII. Related Work

Transformative Approaches There are several transformative I/O middleware layers currently developed for HPC environments. Reaching Exascale I/O performance is likely to rely on these middleware layers capable of managing parallel I/O workloads [21]. Work has been conducted on matching the user view of parallel I/O to optimized workloads on a parallel file system [22], but PLFS takes this further and attempts to mask I/O workload and system configuration parameters from users. The Scalable Checkpoint restart library speeds the checkpointing bandwidth of applications by writing data to memory on neighboring nodes, but is limited to N-N workloads [23]. I/O forwarding is a promising technique capable of aggregating I/O in order to present efficient access patterns to parallel file systems [24], but may require additional I/O infrastructure to be viable. GIGA+ is a project that efficiently manages large-scale file ingestion to a single directory, which represents a heavy meta data workload [19], but is a dynamic approach. Since we are focused with large-scale checkpointing workloads we can spread the I/O workload across multiple directories with static hashing techniques. DataStager is an asynchronous I/O service layer that is designed to handle checkpoint workloads, we differ with our focus on achieving high I/O performance during explicit I/O phases in order to avoid the introduction of jitter into computation phases [25].

Collective I/O Collective I/O was introduced as a method to improve the performance of parallel storage systems, by implementing buffering and prefetching for distributed I/O requests [18]. These techniques proved beneficial, but the logical partitioning of the file can cause a large amount of data movement. This work was further improved to adjust the logical partitioning of a file, so that locking on the underlying parallel file system would be minimized [26]. PLFS is an improvement in that it eliminates locking without data movement and also eliminates the requirement of user defined tuning parameters. The collective I/O project most similar to PLFS is the Adaptable I/O System (ADIOS) from Oak Ridge National Laboratory [27]. ADIOS is an I/O library and API for scientific codes that efficiently groups scientific array data and is capable of writing the data in a log-structured format. Unlike PLFS, ADIOS provides an API similar to the HDF-5 and parallel netCDF data formatting libraries, and not a POSIX interface [10], [9].

Large Scale Storage Systems The previous related work is all attempting to manage the problems that large-scale concurrent access to a storage system present. Another community, large-scale distributed systems, have also faced storage concurrency issues but their solutions are vastly different. Large-scale distributed storage systems have been developed by Google and they have chosen to abandon POSIX interfaces and relax POSIX semantics in order to achieve large-scale concurrency [28]. PLFS hasn't abandoned a POSIX interface, but rather chosen to relax POSIX write semantics to achieve independence on the write phase. Google has also developed a distributed key/value store that is implemented on top of their file system and there now exist several distributed key-value stores [29], [30], [31]. The trend shows that to achieve large scale concurrency storage systems should be application aware. In our community it may not be economically feasible to build and design custom storage solutions, so transformative I/O libraries are an attractive alternative.

VIII. Conclusions

In this work we augmented our write-optimized transformative middleware layer, PLFS in order to gain large increases in read bandwidth and metadata rates. We then demonstrated the power and flexibility of transformative I/O by improving the write, read, and metadata performance of I/O workloads by 150x, 10x, 17x respectively, on processor counts of up to 65,536 in an HPC I/O stack. The complexity of current and planned parallel file systems leads us to believe that transformative I/O libraries will continue to be part of future storage system stacks. These libraries, developed by storage systems researchers, keep detailed knowledge of the *underlying parallel file system* hidden from users. The flexibility of these libraries leads us to believe that similar techniques may have implications beyond HPC.

Although not the focus of this paper, we presented brief arguments for why transformative I/O is well-placed in middleware as opposed to being embedded in the applications or within parallel file systems. We are mere years away from the exascale era. Due to their complexity, these sorts of data transformations will not make their way into parallel file systems in time. We anticipate that middleware will play a key role in the I/O stack of exascale systems and hope that our contributions in this paper will inform their design[4].

Notes

[1]Note that PLFS does assume that the clocks on the cluster are synchronized; however large scientific workloads do not typically overwrite data so this assumption is not problematic in practice.

[2]The effective read bandwidth includes the open, read, and close time in the bandwidth calculation.

[3]Both of these solutions assume the use of the MPI-IO interface, which we leverage for coordination

[4]LA-UR-12-10085

References

[1] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, pp. 375–408, September 2002. [Online]. Available: *http://doi.acm.org/10.1145/568522.568525*

[2] E. Elnozahy and J. Plank, "Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 2, pp. 97 – 108, april-june 2004.

[3] J. Bent, B. Mcclelland, G. Gibson, P. Nowoczynski, G. Grider, J. Nunez, M. Polte, and M. Wingate, "Plfs: a checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 21:1–21:12. [Online]. Available: *http://doi.acm.org/10.1145/1654059.1654081*

[4] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320. [Online]. Available: *http://dl.acm.org/citation.cfm?id=1298455.1298485*

[5] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02. Berkeley, CA, USA: USENIX Association, 2002. [Online]. Available: *http://portal.acm.org/citation.cfm?id=1083323.1083349*

[6] S. Microsystems, "Lustre file system," October 2008. [Online]. Available: *http://www.sun.com/software/products/lustre/docs/lustrefilesystem_wp.pdf*

[7] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 2:1–2:17. [Online]. Available: *http://portal.acm.org/citation.cfm?id=1364813.1364815*

[8] P. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "Pvfs: A parallel file system for linux clusters," in *IN PROCEEDINGS OF THE 4TH ANNUAL LINUX SHOWCASE AND CONFERENCE*. MIT Press, 2000, pp. 391–430.

[9] "The HDF Group," http://www.hdfgroup.org/.

[10] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netcdf: A high-performance scientific i/o interface," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 39–. [Online]. Available: *http://doi.acm.org/10.1145/1048935.1050189*

[11] "FUSE: Filesystem in Userspace," http://fuse.sourceforge.net/.

[12] H. Taki and G. Utard, "Mpi-io on a parallel file system for cluster of workstations," in *Cluster Computing, 1999. Proceedings. 1st IEEE Computer Society International Workshop on*, 1999, pp. 150 –157.

[13] R. Thakur, W. Gropp, and E. Lusk, "An abstract-device interface for implementing portable parallel-i/o interfaces," in *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 180–. [Online]. Available: *http://portal.acm.org/citation.cfm?id=795667.796725*

[14] Gary Grider and James Nunez and John Bent, "LANL MPI-IO Test," http://institutes.lanl.gov/data/software/, jul 2008.

[15] B. Philip, L. Chacón, and M. Pernice, "Implicit adaptive mesh refinement for 2d reduced resistive magnetohydrodynamics," *J. Comput. Phys.*, vol. 227, pp. 8855–8874, October 2008. [Online]. Available: *http://portal.acm.org/citation.cfm?id=1410469.1410593*

[16] R. Hedges, B. Loewe, T. T. McLarty, and C. Morrone, "Parallel file system testing for the lunatic fringe: The care and feeding of restless i/o power users," in *MSST*, 2005, pp. 3–17.

[17] J. Borrill, J. Carter, L. Oliker, D. Skinner, and R. Biswas, "Integrated performance monitoring of a cosmology application on leading hec platforms," in *Parallel Processing, 2005. ICPP 2005. International Conference on*, 2005, pp. 119 – 128.

[18] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 182–. [Online]. Available: *http://dl.acm.org/citation.cfm?id=795668.796733*

[19] S. Patil and G. Gibson, "Scale and concurrency of giga+: file system directories with millions of files," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, ser. FAST'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 13–13. [Online]. Available: *http://portal.acm.org/citation.cfm?id=1960475.1960488*

[20] S. Microsystems, "Lustre scalability," 2009. [Online]. Available: *http://wiki.lustre.org/images/4/41/Barton_LUG_2009_Scalability.pdf*

[21] G. Grider, J. Nunez, J. Bent, S. Poole, R. Ross, E. Felix, L. Ward, E. Salmon, and M. Bancroft, "Coordinating government funding of file system and i/o research through the high end computing university research activity," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 2–7, 2009.

[22] Y. Chen, M. Winslett, Y. Cho, and S. Kuo, "Automatic parallel i/o performance optimization in panda," in *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '98. New York, NY, USA: ACM, 1998, pp. 108–118. [Online]. Available: *http://doi.acm.org/10.1145/277651.277677*

[23] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: *http://dx.doi.org/10.1109/SC.2010.18*

[24] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. Ross, and K. Yoshii, "Accelerating i/o forwarding in ibm blue gene/p systems," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: *http://dx.doi.org/10.1109/SC.2010.8*

[25] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "Datastager: scalable data staging services for petascale applications," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, ser. HPDC '09. New York, NY, USA: ACM, 2009, pp. 39–48. [Online]. Available: *http://doi.acm.org/10.1145/1551609.1551618*

[26] W.-k. Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective i/o based on underlying parallel file system locking protocols," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 3:1–3:12. [Online]. Available: *http://dl.acm.org/citation.cfm?id=1413370.1413374*

[27] J. Lofstead, S. Klasky, S. K., N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," June 2008. [Online]. Available: *http://www.adiosapi.org/uploads/clade110-lofstead.pdf*

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: *http://doi.acm.org/10.1145/945445.945450*

[29] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, pp. 4:1–4:26, June 2008. [Online]. Available: *http://doi.acm.org/10.1145/1365815.1365816*

[30] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, April 2010. [Online]. Available: *http://doi.acm.org/10.1145/1773912.1773922*

[31] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220. [Online]. Available: *http://doi.acm.org/10.1145/1294261.1294281*