LA-UR-11-10426
Approved for public release; distribution is unlimited.

Title: Reducing Concurrency Bottlenecks in Parallel I/O Workloads

Author(s): Manzanares, Adam C.
Bent, John M.
Wingate, Meghan

Intended for: SC 2011, 2011-11-12/2011-11-18 (Seattle, Washington, United States)

Reducing Concurrency in Parallel File Systems

Adam Manzanares
Los Alamos National Laboratory
adamm@lanl.gov

John Bent
Los Alamos National Laboratory
johnbent@lanl.gov

Meghan Wingate
Los Alamos National Laboratory
meghan@lanl.gov

Milo Polte
Carnegie Mellon University
milop@cs.cmu.edu

Garth Gibson
Carnegie Mellon University
garth@cs.cmu.edu

ABSTRACT

To enable high performance parallel checkpointing we introduced the Parallel Log Structured File System (PLFS). PLFS is middleware interposed on the file system stack to transform concurrent writing of one application file into many non-concurrently written component files. The promising effectiveness of PLFS makes it important to examine its performance for workloads other than checkpoint capture, notably the different ways that state snapshots may be later read, to make the case for using PLFS in the Exascale I/O stack.

Reading a PLFS file involved reading each of its component files. In this paper we identify performance limitations on broader workloads in an early version of PLFS, specifically the need to build and distribute an index for the overall file, and the pressure on the underlying parallel file system's metadata server, and show how PLFS's decomposed components architecture can be exploited to alleviate bottlenecks in the underlying parallel file system.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management-File organizations

General Terms

Performance, Design

Keywords

High performance computing, parallel computing, parallel file systems and IO

1. INTRODUCTION

Parallel I/O for high performance computing platforms is currently a challenge and the complexity of Exascale compute platforms will push I/O requirements higher. Currently, application developers must carefully manage their I/O to achieve high performance from parallel file sytems. There exists a large body of middleware specifically designed for high performance parallel I/O but certain I/O workloads remain a challenge [10] [4]. One particular workload, shared file checkpointing, is particularly difficult for current generation parallel file systems and we introduced the Parallel Log Structured File System (PLFS) at the Los Alamos National Laboratory (LANL) in order to decouple a shared file, which led to large write performance gains [2]. PLFS was designed for checkpointing workloads, but we envision that PLFS will be a part of our Exascale I/O stack because it is capable of transforming logical I/O workloads into physical I/O workloads tuned for an underlying parallel file system. The orginal PLFS paper focused on the write performance of PLFS, and as we began to investigate the read performance of PLFS it was quickly realized that the original design of PLFS was causing poor scalability in terms of read performance.

The low read performance of PLFS at scale is largely attributed to the architecture of PLFS which requires a logical write to write a non-shared index and data file. To read back a file the non-shared index files must be aggregated and the orginal design of PLFS attempted to aggregate this information non-collectively. This placed a heavy buden on the underlying parallel file system which must deal with an increasing amount of file open operations that is a product of the number processes that wrote and the number of processes that are attempting to read the file. To improve the read performance of PLFS we designed several collective I/O strategies that limit the number of open operations, lowering the concurrent access to files.

Metadata management is another challenge for many current parallel file systems and will only become increasingly difficult with the sheer amount of cores and nodes required for next generation high performance computers. Current parallel file systems are challenged by particular metadata workloads and massive scale file creation in a single directory is one particular workload that has been identified and studied [6].Since PLFS allows us to rearrange workloads we also demonstrate the performance gains possible when detailed knowledge of the underlying parallel file system is leveraged to improve metadata performance. Using PLFS we can turn independent metadata servers into a federated system capable of improving metadata performance by reducing the amount of concurrent access to a metadata server.

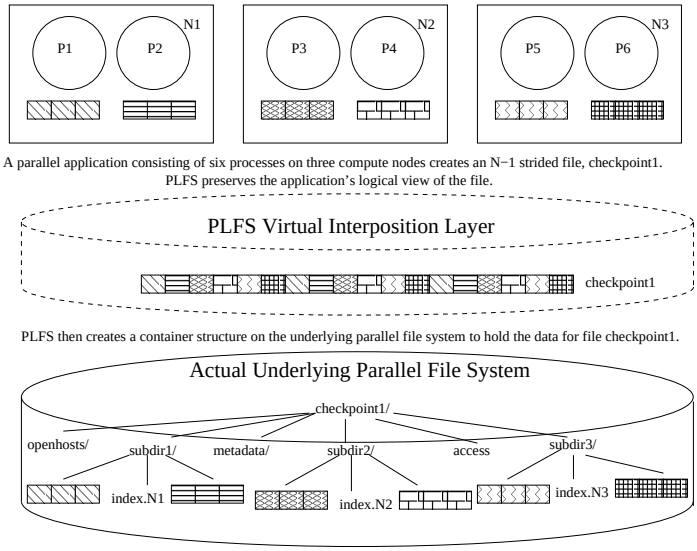


Figure 1: **PLFS Architecture** *This figure demonstrates how a shared logical file, checkpoint1 is decoupled into component pieces on the underlying parallel file system. PLFS maps the logical file into a container structure that holds one openhost and metadata directory and multiple subdirs. The openhost and metadata directories hold information on the processes that have a PLFS file open and the last logically written offset, respectively. The subdir directories contain the non-shared data files that each process writes and a non-shared index that maps non-shared data to logical offsets in checkpoint1.The container also holds an access file that holds the user and access information for the file.*

This paper details the importance of PLFS for current and future generation file systems and we offer the following novel contributions.

- We investigate the relationship between the PLFS architecture and the read performance of PLFS
- We develop three collective I/O strategies that improve the read performance of PLFS.
- We provide a detailed analysis of the read performance of PLFS under several scientific workloads including several I/O kernels representing LANL workloads.
- We illustrate the improvement of metadata performance by leveraging the PLFS architecture to federate a group of independent metadata servers.

The rest of the paper is organized as follows: Section 2 provides a summary of the architecture and design of PLFS. Section 3 details the solutions developed to achieve high read performance from PLFS. Section 4 examines the read performance of PLFS using I/O kernels. Section 5 explores the capability of PLFS to ingest detaild file system knowledge to improve metadata and I/O performance. Section 6 provides an overview of related work. The conclusion is presented in Section 7

2. PLFS DESIGN AND ARCHITECTURE

At LANL critical applications performing N-1 I/O were only able to achieve a small fraction of the available bandwidth of the parallel file system whereas applications performing N-N I/O were able to utilize nearly all of the available bandwidth. In an N-1 parallel I/O workload N processes read or write to the same file as contrasted to a N-N workload where N processes write to N separate files. PLFS was developed as a virtual file system that is capable of transforming an N-1 I/O workload into an N-N workload vastly improving the write bandwidth for applications that perform N-1 I/O. PLFS was successful at speeding up a set of test applications by up to 300x and the performance benefits of PLFS were demonstrated on three underlying parallel file systems available at LANL: Lustre,GPFS, and Panasas [5][9][12]. A key design principle of PLFS is that we attempt to keep the I/O transformation hidden from users. The architecture of PLFS is highlighted in Figure 1 and a detailed description of the important design principles and structures that support the transformation of I/O workloads follows.

PLFS uses a mount point to designate a location for files that will be transformed on the underlying parallel file system. Currently there are three methods for a user to access a PLFS mount, a mount point on the system that relies on FUSE to intercept I/O calls, the PLFS ADIO interface of MPI-IO, and by directly linking the PLFS library to the user applications. 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 data files in the PLFS mount point. FUSE is a framework that allows the development of userspace file systems [1] and is capabe of intercepting I/O calls and redirecting them to the PLFS library. ADIO is an abstract device interface for MPI-IO that is also able to reroute I/O calls to the PLFS library and also allows us to inherit communicators and job info from MPI [11] . PLFS gives each process access to a non-shared file effectively transforming the N-1 I/O workload into N-N. PLFS achieves this by creating a container structure on the underlying parallel file system. The container is a directory

Openers	Opens	Files	Description	Shorthand Notation	Figure Reference
N	N^2	N	Original PLFS Design	$N - N^2 - N$	5(a)
1	N	N	Index Broadcast	1-N-N	5(b)
1	1	1	Index Flatten	$1 - 1 - 1$	5(b)
N	N	N	Parallel Index Read	$N - N - 1$	5(c)

Table 1: Summary of Workload Taxonomy Used

with the same name of the PLFS logical file and contains an access file and owner file that are used for checking access control and ownership respectively. The openhost directory within the container has a record of any processes that currently have the file open for write access. In PLFS there is no assumption that every process has the ability to communicate with a library such as MPI, so PLFS achieves some communication through the underlying file system. The metadir directory within the container serves a similar purpose and serves as a repository of the last offsets and file sizes seen by each process which can be used to quickly detemine the size of a PLFS file.

There are multiple sub directories (subdirs) that contain the two data files that correspond to a particular process, the data and the index file. Each process appends incoming data to its non-shared data file and also records a map between the incoming logical offset and the physical offset in the non-shared data file to the process specific index file [8]. To reduce the amount of concurrent access to a particular directory we create multiple subdirs within each file container and each process uses its hostname as a hash into a particular subdir. Writing to non-shared data files and keeping indexing information for the shared file effectively transforms the N-1 write workload into N-N, but the distributed indices pose a challenge during a attempt to read back a PLFS file. Since each process could potentially read from any logical offset in the PLFS file, each reader must aggregate the information from every single index file. In the FUSE case the index is aggregated on a node basis, but when MPI-IO is used there are N index files, where N is the number of processes that wrote the file. In the MPI-IO case of writing and then reading a PLFS file with N processes, N readers will attempt to read N indices at the same time causing concurrent access to N files by N process and this I/O workload negatively impact the performance of the underlying parallel file system.

2.1 Read Open Workload Taxonomy

Table 2 summarizes the workload taxonomy that will be used throughout the Boosting PLFS read performance section to describe workloads that are generated on the read open phase of a PLFS file. For each workload we provide the openers, opens, files, description, , a shorthand notation, and reference to figure 2. The openers field describes the number of processes that are going to generate requests to the parallel file system. The opens field describes the amount of open operations that the workload generates on the underlying parallel file system. The files field is the the number of files that will be opened during the workload. It is important to note that each file can be opened multiple times, therefore we have included the opens field. These workloads and their relationship to the solutions we developed to improve the read bandwidth of PLFS are described in the following sections.

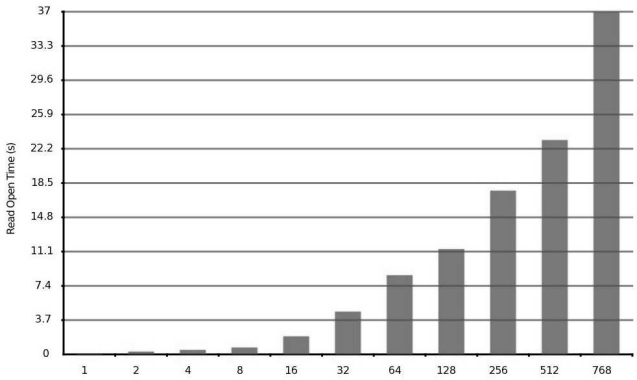


Figure 3: **Read open times as a function of process count** *This graph illustrates the problem with the original design of PLFS. As process count is increased the time to open the file is increasing exponentially.*

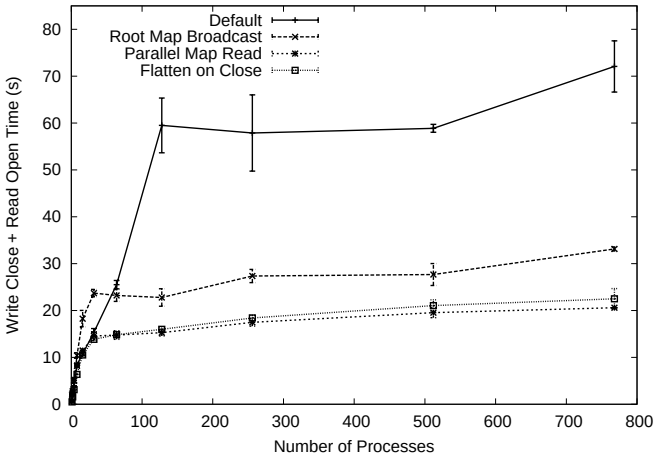
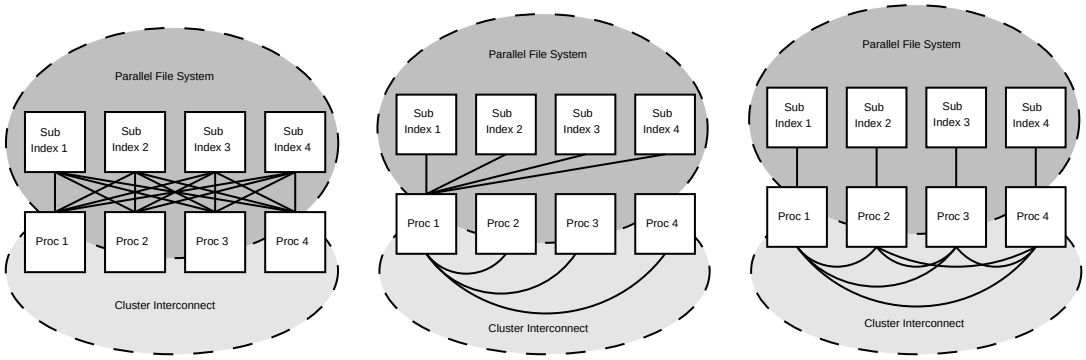


Figure 4: **Read Open + Write Close Times** *This graph represents the time to aggregate the global index with the orginal design and the solutions we developed. Note: the write close time is included because the Index Flatten solution conducts work on a write close.*

3. BOOSTING PLFS READ PERFORMANCE



(a) Original Design (b) Index Broadcast/Flatten (c) Parallel Index Read

Figure 2: **Index Aggregation Techniques** *The figures above represent the workloads generated on the parallel file system and cluster interconnect by PLFS using the original design and our three solutions to improve the read bandwidth of PLFS. In the original design of PLFS, all processes are required to open all of the index files. In the Index Broadcast/Flatten technique, one process aggregates the indices and then passes this result over the cluster interconnect to other processes involved in the read operation. This aggregation of indices can take place on the open for read access(Index Broadcast) or on the close of a newly written PLFS file(Index Flatten). The parallel index read assigns the reading of subindices processes and then all processes communicate the subindex results over the cluster interconnect to aggregate the index.*

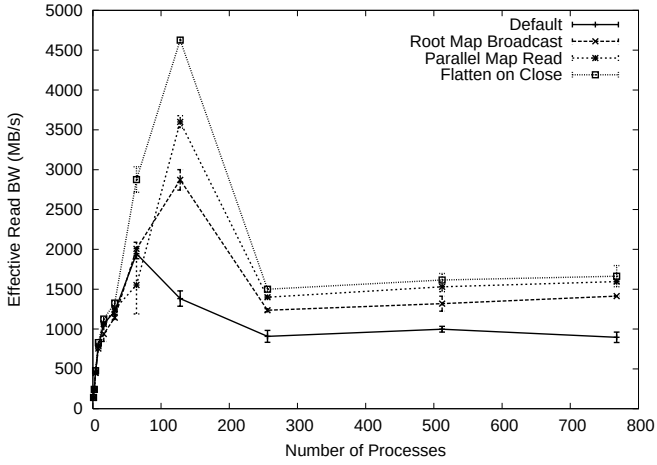


Figure 5: **Read Bandwidth** *This graph shows the read bandwidth of the orginal design of PLFS as compared to the three solutions we developed to improve the read performance of PLFS.*

When PLFS was originally placed and tested on Roadrunner [3][**?**] during a scheduled maintenance period, we started to test the read performance of PLFS at large scales. We quickly noticed that our effective read bandwidths were diminishing with increasing process counts and our I/O test harness was able to provide us with enough information to determine that the read open times for PLFS files were quickly growing with the scale of the job and began surpassing the pure read times. We repeated these tests on a smaller scale cluster that we generally have greater access to and produced figure 3. This confirmed our suspicion that the read open times were dominating the overall time of a read operation and we would have to develop a solution for the high read open times produced by PLFS. The three solutions we developed are to:

- Aggregate the global index on read open with one process and broadcast this result to every other process (Index Broadcast)
- 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).

These three techniques are outlined in the following sections and they all share the common property of reducing the amount of I/O requests that are directed to the underlying parallel file system. They all also leverage the high speed cluter interconnect to some degree and reduce the amount of concurrent access to files as compared to the original design of PLFS. The current solutions to reduce read open times of a PLFS file that we have implemented are collective and assume that the user is accessing PLFS through the ADIO interface of MPI-IO. All of the techniques to reduce the read open time need to know all processes that are attempting to access a PLFS file and require a method of communication amongst this group of processes and MPI-IO provides this functionality. We are currently investigating a method to get job information and communication methods amongst PLFS FUSE mount points across different nodes and reserve this for future work.

3.1 Read Open Workload Taxonomy

Table 2 summarizes the workload taxonomy that will be used throughout the Boosting PLFS read performance section to describe workloads that are generated on the read open phase of a PLFS file. For each workload we provide the openers, opens, files, description, , a shorthand notation, and reference to figure 2. The openers field describes the number of processes that are going to generate requests to the parallel file system. The opens field describes the

amount of open operations that the workload generates on the underlying parallel file system. The files field is the the number of files that will be opened during the workload. It is important to note that each file can be opened multiple times, therefore we have included the opens field. These workloads and their relationship to the solutions we developed to improve the read bandwidth of PLFS are described in the following sections.

3.2 Index Broadcast

PLFS improves write bandwidths by transparently mapping an N-1 workload into N-N by giving each process access to a non shared file. Each process must also write an index that maps logical writes to physical locations, which is required to read the information in a PLFS file. This architectural design improves write bandwidths for challenging I/O patterns, but presents a challenge when a read occurs. Any process could potentially read from any logical position in the file which could map to any of the non-shared data files that are created by PLFS on a write. PLFS was originally designed with a shared-nothing approach and this allowed PLFS to achieve the high write performance, but sticking to the shared-nothing approach on a read meant that each process attempts to read every single index file that is created on the write. If a PLFS file is written and read by N processes, this will produce the $N - N^2 - N$ workload on the parallel file system, which is represented by figure 2a. This workload requires the parallel file system to handle N^2 opens because N processes are all attempting to open N files. This in turn, places an extreme demand on the metadata servers of the parallel file system when we attempt to read a file with a large amount of proceses that was previously written by a large amount of processes. This burden that was placed on the parallel file system was the culprit for the exponentially increasing read open times that are a function of job size.

To remedy this problem we decided to assign one process to read all of the scattered indices, combine them, and broadcast this result to every other process. This collective solution is illustrated in figure 2b and it is important to note that the index aggregation takes place when a file is opened for read access. This appoach is possible with the ADIO interface of PLFS because we are using MPI and we can have the root process of the MPI job read all of the scattered sub-indices. Every other rank will wait for a broadcast of the global index while the root process merges the scattered sub-indices in-memory and then broadcasts this result to every other process involved in the current MPI job. This effectively morphs the orginal workload into the $1 - N - N$ workload which has one process reading the N indices so that the parallel file system now only has to deal with N file opens. This approach has a clear advantage over the $N - N^2 - N$ workload because there is not concurrent access to the N index files from multiple nodes.

3.3 Index Flatten

When a PLFS file is written, each process must write the index information for each write issued through PLFS and ideally the data size of each write operation should be large enough to amortize the cost of writing the index information. For the index flatten solution for the low read bandwidth of PLFS, as a file is written, we hold all of the index information in a buffer of a fixed size. If the index ends up being less than or equal to the fixed buffer size on all processes then we aggregate all of the subindices from all processes involved in the write operation to the root process when the user attempts to close the file. The root process then merges all of the subindices and writes out one global index to the underlying parallel file system containing information from all of the merged subindices. This solution is also represented by figure 2b, but it is important to note that the index aggregation takes place during the close of a newly written file.

This solution produces the $1 - 1 - 1$ workload on a read and requires one process to read one file, which results in one open on the parallel file system dramatically reducing concurrent access to files. This approach is an improvement over the Index Broadcast technique, but it has the disadvantage that it degrades the write performance of a file because the index aggregation is performed on the write close. This approach has to be used with caution because it reduces the effective write bandwidth because merging scattered indices and writing is conducted during the close of a newly written PLFS file. Since PLFS was originally designed as a write optimized checkpointing file system we decided to allow users the choice in selecting this technique. If they 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 Index Flatten if they plan on archiving a PLFS file because the servers responsible for moving data into archival storage are limited in number and processing power and would only be required to open one global index. Index Flatten reduced the amount of concurrent access to files on the underlying parallel file system required to read a PLFS file, but we also realized that we were not utilizing the parallelism and the high speed interconnect available on our cluster computer systems. This realization led to the final collective index aggregation technique, the Parallel Index Read.

3.4 Parallel Index Read

Although the Index Flatten technique was successful at reducing concurrent access to files, the cost to achieve this reduction was incurred when the file was written. PLFS is a write optimized file system so we decided to develop a technique that would match the performance of the Index Flatten approach but not impact the write performance of PLFS. The Parallel Index Read technique was our approach that met these particular design goals. This approach has one processor assign work to groups of processes. Each group of processes has a group leader who assigns work to members of the group. Once each process is done reading its assigned subindices it passes its result to its respective group leader. The group leaders aggregate subindices 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 allowed us to produce a $N - N - N$ workload, but unlike the Index Flatten operation every process opens one subindex and the results are communicated among the processes through the cluster interconnect. Figure 2c illustrates this technique and it should be noted that most of the index aggregation work

takes place on the cluster nodes and interconnect as opposed to the parallel file system. Since our cluster interconnect is typically much faster than the parallel file system and the cluster nodes are underutilized during I/O phases it seemed natural to leverage these resources.

3.5 Performance Analysis

Figure 4 compares the index aggregation times of the orginal design of PLFS to all three solutions that we developed to reduce the amount of concurrent access to files on the underlying parallel file system. The results were generated with a synthetic I/O workload tool developed at LANL. The Index Broadcast technique was able to improve the performance of PLFS by 2.2X and this improvement is largely attributed to the reduction in concurrent access. The root process responsible for aggregating indices is doing the same amount of work that each process had to complete in the original design. The difference is that there is no longer conncurrent access to all of the subindex files by all of the processes attempting to read the file. The Index Flatten solution was able to improve the performance of PLFS by 3.2X and is an improvement over the Index Broadcast technique because processes have in-memory index information that is passed to the root process. This is a contrast to the Index Broadcast technique which is required to access index information that is in files on the parallel file system. The Index Flatten solution needs to open one file on the parallel file system to access all of the index information and then broadcasts this result to all processes minimizing concurrent access to the global index file on the parallel file system. The Parallel Index Read technique aggregates the index on the read open phase of file access and is an improvement over the Index Broadcast technique because the work of aggregating sub-indices is spread amongst processes attempting to read the PLFS file. This approach was able to improve the performance of PLFS by 3.5X and is the best solution when looking at the index aggregation time. The important thing to note is that all of the solutions developed to improve the read bandwidth scale linearly with the number of processes as opposed to the orginal design which scaled exponentially in terms of the amount of open operations on the parallel file system.

Figure 5 plots the read performance of the orginal design of PLFS as compared to the solutions we devised to reduce concurrent access to files on the parallel file system. These results were also collected using LANL's synthetic I/O workload generator. Again looking at larger scales it is evident that the orginal design degrades read performance as the scale is increased. The highest read bandwidth achieved is the Index Flatten operation, which improve read bandwidth by 1.8X for 768 processes. The Index Flatten operation shows the largest gain in terms of read bandwidth because the read bandwidth number does not include the index aggregation penalty incurred on the write close of the file. The Parallel Index Read approach has the second highest read bandwidth and improves the peformance of the original design of PLFS slightly less than 1.8X. The Index Broadcast technique has a performance gain of 1.6X and is not expected to scale as well as the Parallel Index Read approach. The improvement in terms of read bandwidth of all our solutions grows with the number of processes and illustrates the importance of reducing the amount of concurrent access to files on the underlying parallel file system. Since the Parallel Index Read approach was capable of achieving read bandwidth close to the performance of Index Flatten without incurring a penalty on the write phase of a PLFS file we have chosen this as the default behavior of PLFS accessed through ADIO and all of our results in the read analysis use this default.

4. PLFS READ ANALYSIS

As LANL high performance computing users started to test PLFS it was noticed that they were using PLFS, a write optimized parallel file system, as a general purpose file system. PLFS is heavily write optimized and we realized that an analysis of the read performance across several I/O kernels was missing from our original work. The initial investigation into the read performance of PLFS lead to the index aggregation solutions presented in Section 3. Once we were satisfied with our results generated from our synthetic I/O tool it was decided that a test of the read performance of PLFS would not be complete without results from a set of parallel I/O benchmarks that represent a variety of parallel I/O workloads. We have also included results from the read performance of two I/O kernels that represent applications in heavy use at LANL.

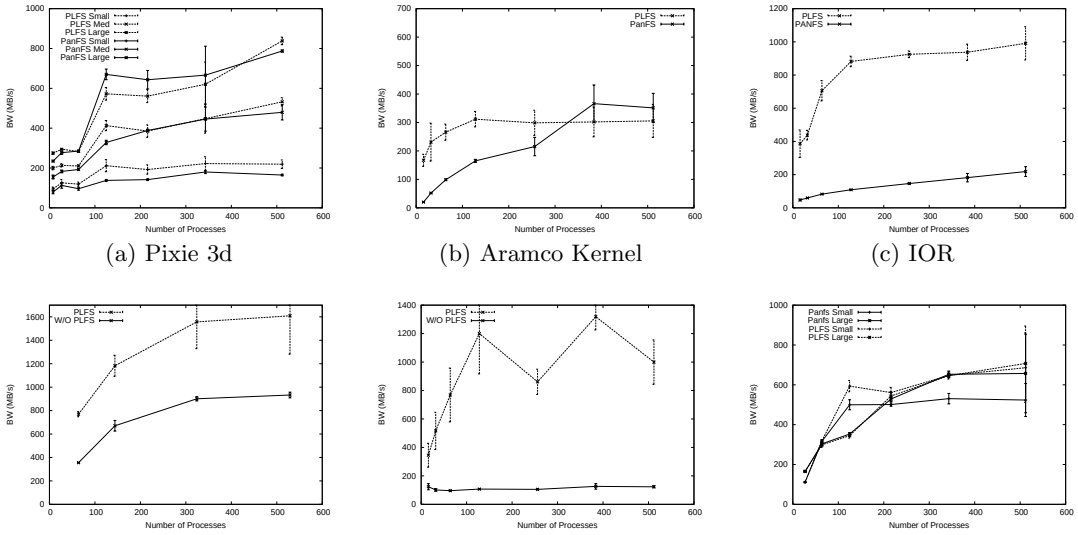
4.1 Pixie 3D

The Pixie 3D benchmark is an I/O kernel derived from the Pixie 3D MHD (Magneto Hydro-Dynamic) code [7] and is widely used for parallel I/O benchmarking. We compared the performance of PLFS to the underlying parallel file system (PanFS) using three data sizes: small (16MB per process), medium (128MB per process), and large (1GB per process). It is important to note that this benchmark is reading from a shared file. From figure 6a we can see that PLFS outperforms the underlying parallel file system by a modest amount for all process counts when the data size is small. For the medium sized file we see that PanFS outperforms PLFS for smaller process counts, but the read performance of PLFS surpasses PanFS for the large process counts. The large data size follows a pattern that is similar to the medium sized file. (PNETCDF4 mention)

When PLFS writes a shared file each write is placed in a log file on a per process basis. When reading back a shared file these log structured files allow the underlying parallel file system to perform read ahead and caching more efficiently. The problem with PLFS is that there is a penalty to aggrete the scattered indices and this penalty is reduced when either the raw read time is longer (larger files) or the size of the index is small (ratio of data written to index size is low). Although PanFS and PLFS differ slightly depending on the amount or processes used and the size of data the read performance of PLFS is close enough to PanFS that we consider this a strong result for our write optimized file system.

4.2 ARAMCO

The Saudi ARAMCO is an I/O kernel for the seismic processing map using MPI-IO and HDF5. For process counts lower than 300 PLFS is able to improve the read performance of the Saudi ARAMCO kernel by up to 8X. The ARAMCO I/O benchmark writes a fixed size of data re-



(a) Pixie 3d (b) Aramco Kernel (c) IOR

(d) Madbench (e) LANL Anonymous 2 (f) LANL Anonymous 3

Figure 6: **Experimental Results.** *This collection of graphs represents the read performance of PLFS across a diverse set of parallel I/O benchmarks.*

gardless of the amount of processes (8GB). For larger process counts PanFS is 1.2X faster than PLFS. The ARAMCO I/O benchmark writes a fixed size of data regardless of the amount of processes. The large speedup in I/O at small scales when using PLFS is due to several factors. The first factor is that with smaller process counts there are less index files. Since each process writes one large set of data every subindex will be the same size, increasing the number of processes increases the number of subindexes. The second reason is that the workload presented to the underlying parallel file system is conducive to read ahead. Withouth PLFS each process will attempt to read at offsets scattered throughout one file. Since PLFS decouples the file on the write phase when the file is subsequently read each process reads from one file in logically increasing offsets.

4.3 LANL Anonymous 3

LANL anonymous 3 is an I/O kernel representing a heavily used application at Los Alamos. Collective buffering is turned on for this application because the application tries to write/read 1024 bytes per operation resulting in low performance from PLFS and the underlying parallel file system. This benchmark also writes and reads from a shared file and writes 4GB for the small tests and 32GB for the large tests. PanFS and PLFS have similar performance numbers with a couple of interesting observations. PLFS has a lower read performance for both data sizes when 128 processes are used. PLFS has a certain amount of overhead incurred when the index has to be aggregated and if the amount of data actually read is not enough to amortize the cost of reading the index our performace will be lower than the underlying parallel file system. The read performance of PLFS and PanFS is roughly the same for all other process counts. The one exception being that the smaller file written to PanFS levels off after 256 process while every other test has slighlty increasing bandwidth.

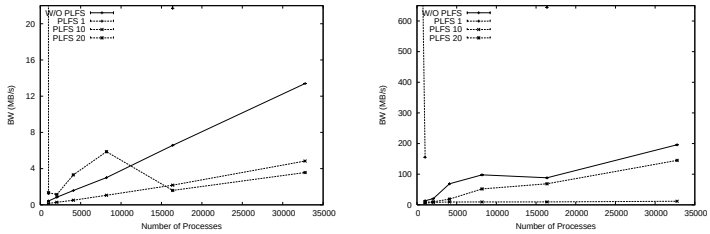
4.4 IOR

The IOR benchmark is a parallel I/O bechmarking utility developed at the Lawrence Livermore National Laboratory (LLNL). The benchmark has many configurable options and we chose to run IOR writing and reading to a shared file. We also configured each process to write 50MB and write and read in 1MB increments. IOR writes the file in a segmented manner meaning that each process has data in one region of the file. In this benchmark PLFS outperforms PanFS for all process counts. PLFS is able to improve the read performance of PanFS by up to 4.5X. This can also be explained by the fact that read ahead for the underlying parallel file system is difficult because each process accesses data in completely different regions of the file. Although we are reading one logical file, in PLFS the reading takes place in separate files meaning that readahead takes place on every single non-shared data file.

5. FEDERATED MDS

One of the downsides of PLFS is that if a user decides to run an N-N workload through PLFS there is a performance penalty associated container structure creation of a file accessed through PLFS. We realized that this penalty was largely incurred by metadata servers having to create the files and directories that make up a container. The PanFS storage system that we currently have assigns a metadata manager on a per volume basis. Typically we have users of the parallel file system spread across multiple volumes to spread the metadata workload. Realizing that the PLFS subdirs within the container could potentially be placed on any volume we decided to make plfs storage space across all volumes, which is illustrated in figure 7.

Once the storage space for PLFS on each volume was set in place a slight modification was made to the PLFS library to hash a subdir to a specific volume. This allowed us to aggregate the metadata performance of multiple metadata



(a) N-N Read Open Times (b) N-N Write Open Times

(c) N-1 Read Open Times (d) N-1 Write Open Times

Figure 8: **Federated Metadata Server Results** *This collection of graphs represents the performance gains possible when we leverage the PLFS middleware layer to federate metadata servers on our underlying parallel file system.*

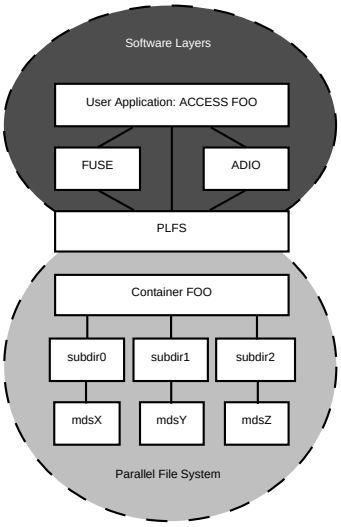


Figure 7: **PLFS Federated Metadata Servers** *This figure highlights how the architecture of PLFS allows us to map PLFS container subdirectories (subdirs) to metadata servers that manage separate namespaces.*

servers creating a federated metadata server system.

6. RELATED WORK

7. CONCLUSIONS

I will be a nice summary of our contributions.

8. ACKNOWLEDGMENTS

I will be the ack when I am done.

9. REFERENCES

[1] FUSE: Filesystem in Userspace. http://fuse.sourceforge.net/.

[2] Plfs: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.

[3] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 1:1–1:11, Piscataway, NJ, USA, 2008. IEEE Press.

[4] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich io methods for portable high performance io. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[5] S. Microsystems. Lustre file system, October 2008.

[6] 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*, FAST'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[7] B. Philip, L. Chacón, and M. Pernice. Implicit adaptive mesh refinement for 2d reduced resistive magnetohydrodynamics. *J. Comput. Phys.*, 227:8855–8874, October 2008.

[8] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10:26–52, February 1992.

[9] 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*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[10] 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.

[11] 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*, FRONTIERS '96, pages 180–, Washington, DC, USA, 1996. IEEE Computer Society.

[12] 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*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.