Lustre and PLFS Parallel I/O Performance on a Cray XE6

Comparison of File Systems

Brett M. Kettering
HPC-5
Los Alamos National Laboratory
Los Alamos, USA
brettk@lanl.gov

Alfred Torrez
HPC-5
Los Alamos National Laboratory
Los Alamos, USA
atorrez@lanl.gov

David Bonnie
HPC-3
Los Alamos National Laboratory
Los Alamos, USA
dbonnie@lanl.gov

David Shrader
HPC-3
Los Alamos National Laboratory
Los Alamos, USA
dshrader@lanl.gov

Abstract—**Today's computational science demands have resulted in larger, more complex parallel computers. Their PFSs (Parallel File Systems) generally perform well for N-N I/O (Input/Output), but often perform poorly for N-1 I/O. PLFS (Parallel Log-Structured File System) is a PFS layer under development that addresses the N-1 I/O shortcoming without requiring the application to rewrite its I/O. The PLFS concept has been covered in prior papers. In this paper, we will focus on an evaluation of PLFS with Lustre underlying it versus Lustre alone on a Cray XE6 system. We observed significant performance increases when using PLFS over these applications' normal N-1 I/O implementations without significant degradation in the N-N I/O implementations. While some work remains to make PLFS production-ready, it shows great promise to provide an application and underlying file system agnostic means of allowing programmers to use the N-1 I/O model and obtain near N-N I/O model performance without maintaining custom I/O implementations.**

Keywords-File System, N-1, Parallel, I/O, Performance

I. INTRODUCTION [1]

Because today's large, complex parallel computers contain many components; applications must protect themselves from component failures that interrupt the run. This is done through application checkpointing, where the state of the calculation is saved to persistent storage so that the application can be restarted from its last saved state and eventually complete.

A natural, flexible, and less metadata-intensive I/O pattern is N-1. Figure 1 depicts this I/O pattern.

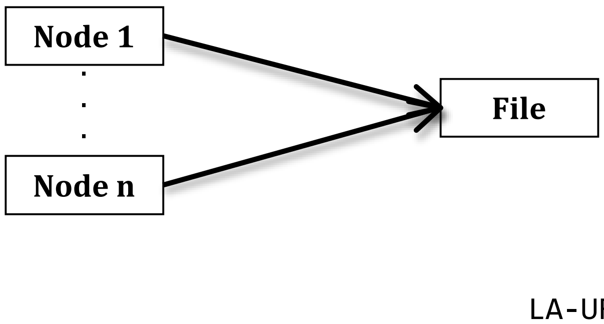

Figure 1: N-1 I/O Pattern

In this paper when we say, "N-1", we specifically mean N-1 strided. In this pattern the processes write multiple small regions at many different offsets within the file. The challenge to the PFS is that these offsets are typically not aligned with file system block boundaries. This results in many concurrent accesses to the same file at different locations. Consequently, the HDDs (hard disk drives) must perform many seek operations and multiple writes to the same file system block will be serialized. Hence, N-1 is poorly served by current PFSs, especially at very large scale. To a lesser extent PFSs must implement a file region locking scheme to ensure POSIX compliance.

Because of performance issues with N-1, some applications have changed to N-N, and can obtain good performance on today's PFSs. In N-N, each process does I/O to a handful of reasonably sized files. Figure 2 depicts this I/O pattern (where "File" can be more than one file, but usually just a few).

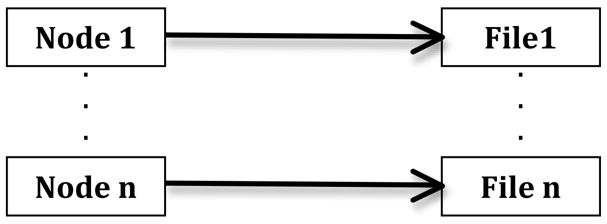

Figure 2: N-N I/O Pattern

Nevertheless, even this model is showing signs of scalability issues when applications put all these files in a single directory. Some applications have implemented custom directory hierarchies to address the issue of overloading a directory with too many files. In the N-N model, PLFS distributes the files to multiple directories, alleviating the application of this responsibility.

The rest of the paper is organized as follows. We present more detailed background and motivation in Section II describe our method for evaluation in Section III and our evaluation results in Section IV. We present the lessons we learned in Section V, and summarize in Section VI.

II. Background [1]

PLFS is publicly available at https://github.com/plfs/plfs-core. Anyone may download and use it. While LANL and its PLFS development partners have been the primary users to date, others have downloaded and used it. We have had some communications with a handful of other users experimenting with PLFS for their parallel I/O workloads.

There are three interfaces to use PLFS. They are: using MPI/IO (Message Passing Interface I/O) through MPI's ADIO (Abstract Device Interface for I/O); the normal POSIX (Portable Operating System Interface) file system interface using PLFS FUSE (File System in User Space) mount points; and the PLFS-specific API (Application Programmer's Interface).

The MPI/IO interface is simple to use, but it does require work on the part of those who maintain the development environment. There is a well-documented patching procedure for MPI/IO that requires access to the MPI source code. Once this is completed the application developer needs to link against that patched MPI and the corresponding PLFS library. The application developer needs to make one small change to the filename parameter of MPI_File_open. That change is to prepend the string, "plfs:" to the filename that is opened. Every subsequent call to a MPI/IO function is done as before. This is the recommended interface as it yields the highest performance for the least change to the application.

There is no application change to use the PLFS FUSE interface. It is as simple as using a standard POSIX filename in a POSIX open statement. Be warned that FUSE serializes all I/O requests. Running N-1 through a PLFS FUSE mount on a single node will result in very poor performance. To use this interface the System Administrators must install the PLFS FUSE daemon application and configure it to provide POSIX file system mount points that are usable by user applications. We have found that the default FUSE read and write buffer size is only 128 KiB. This small size creates a lot of overhead for applications, such as archive utilities, that would otherwise move data in much larger chunk sizes, which is much more efficient. Using read and write buffers closer to the size of the transfers these applications use or that is commonly used to write the file is best. At LANL, this size is 4 MiB to 12 MiB. There is a known patch to the Linux kernel and the FUSE kernel module that increases the buffer sizes. Even after implementing this change to provide larger FUSE buffers, there is still execution overhead using FUSE. In order to allow users to use non-MPI applications to, for example, copy or archive files these PLFS FUSE mount points are required and are the recommended interface to use for these purposes.

The PLFS API yields good performance, and it does not require additional System Administration work to provide PLFS FUSE mount points. Nevertheless, using this interface requires a substantial change to the application. Any MPI/IO or POSIX I/O call must be changed to the corresponding call from the PLFS_API. For example, instead of calling MPI_File_open or POSIX's fopen the application must call plfs_open. If a special-purpose application to perform some operation on PLFS files is needed, this is a good choice because it will yield excellent performance, not require PLFS FUSE mount points, and it is no more work to use PLFS API calls than it is to use MPI/IO or POSIX I/O calls.

PLFS currently supports three workload types. These are N-1, N-N, and 1-N. They are also known as "shared_file", "file_per_proc",and "small_file", respectively. The first two were briefly covered in the Introduction, section I.

The primary driver for developing PLFS was the N-1 workload because it is a popular way for large, parallel applications to checkpoint their state and PFSs have traditionally not handled this workload well. In a nutshell, PLFS converts N-1 into N-N in a directory hierarchy. When parallel applications write a file using N-1, they typically barrier so that all processes are ready to write before any writing begins. Next, the processes are released from the barrier and begin to write the data for which they are responsible. Finally, when the processes are done writing they reach another barrier. Once all processes have completed writing and are at the barrier the file is closed, the data is synced to the PFS, and the processes resume computing. A consequence of this algorithm is that there is no guarantee that the data arrives to be written to storage in order. For example, it is very possible for 1 MiB of data intended to start at offset 0 (zero) in the file to arrive after 1 MiB of data intended to start at offset 3 MiB in the file. If we were to attempt to reorder the data and write it so that it physically lands at its logical location, write performance would suffer due to buffering, reordering and storage device seek latencies. As indicated in its name, PLFS is a log-structured file system. That is, it physically writes the data to storage in the order the data arrives, irrespective of its logical position in the file. PLFS maintains an index mechanism that maps a write's logical position in the file to the physical position to which it was actually written when the data arrived. Thus, when a PLFS file is read, the index is consulted and the data is accessed directly. There can be many write cycles, and, generally, just one read – when the application restarts, it reads its last known state one time. Experience has shown that reading a PLFS file in this manner performs well. The one drawback is a scalable and well-performing index management mechanism is needed for very large files. We are currently working on this problem, and it will be discussed at the end of this paper. By converting the N-1 workload to an N-N and writing data as it arrives, PLFS overcomes the two major issues that slow N-1 performance on PFSs, namely HDD seeks and file region locking.

In order to transform how an N-1 application does I/O to something that is more efficient for the underlying PFSs, N-N, there is some overhead. All the interfaces create a small number of directories and files used to manage the files and information PLFS uses to access the files as if they were a single N-1 file. The number of overhead directories and files depends on the interface used. We won't go into detail on the small variations, but we are happy to provide those details to interested parties. Generally speaking the overhead is: for the

MPI's ADIO interface, 2 * number of writers files – one data file and one index file per writer; for the PLFS FUSE interface, number of writers files + number of nodes files – one data file per writer and one index file per node; and for the PLFS API interface, 3 * number of writers files – one data file per writer, one index file per writer, and one meta file per writer.

For example, there are 16 cores per node on ACES's (Advanced Computing at Extreme Scale, ACES is a partnership of LANL (Los Alamos National Laboratory) and SNL (Sandia National Laboratory)) Cray XE6. A typical MPI application will have 16 MPI processes per node. An application doing N-1 through MPI's ADIO, where every process is a writer, would create 32 files per node: 16 data files and 16 index files. Additionally, a small number of files and directories will be created that are used by PLFS itself for tracking, debug, and other management purposes.

At this time most applications that use N-N will achieve slightly better performance by not using PLFS. This is because currently, most applications do not create so many files that the PFS's metadata server and directory locking function are overwhelmed because of file count. Some PFSes claim they can handle up to 10 million files in a single directory without a reduction in performance. As HPC systems move towards Exascale, even 10 million files may become a small number for HPC applications. Some applications have already implemented a hierarchical directory structure to address the performance issues. Using PLFS for N-N addresses the coming Exascale file count issue by physically distributing the files in a hierarchical directory structure for the application. However, it logically presents to the users as if all the files were in a single directory. Although most N-N applications will not achieve a performance improvement using PLFS, N-N applications with access to multiple PFSs may benefit from PLFS because PLFS enables the aggregation of multiple PFSs into a single virtual PFS with more hardware and MDSs to use concurrently [2]. There is no file count overhead for N-N. When one creates a directory on a N-N mount point a directory is created on each PLFS backend storage device over which the N files are distributed.

The final PLFS workload is in an experimental stage. We became aware of a couple of users' need to write many small files per process, a few KiB or less each. One application estimated that if it converted to this workload it would write up to 400,000 files per process, each file containing 2 KiB – 10 KiB. Writing so little data to each file does not allow a PFS to amortize the file creation/sync/close overhead over much data and consequently the PFS performs very poorly with such a workload. PLFS overcomes this issue by actually reducing the number of files created. Instead of N processes * M files/process, it creates N files (one per process) and stacks the other M files within the one file. Each process also creates two small files in which it keeps metadata. It logically presents to the user as if there were N * M files. We call this workload 1-N. We have some very preliminary numbers for the file creation process and creating many files per process and writing a very small amount of data to each file. These numbers give us hope that we can help applications creating many thousands of small files per process. In order to achieve good performance the workload is not completely POSIX-compliant. Explicit sync calls are required to ensure the latest data is on storage if a thread on another node needs it before the application doing the writing terminates. There is a file count overhead for 1-N. One file is the data log for all the files. One file is almost a traditional PLFS index dropping but each index entry needs another field to specify which file it refers to in the data log. Normal N-1 PLFS index entries don't need this since the data in the data log only belongs to a single file. Since we intersperse data for many files into one data log in 1-N, the index entry needs this new field to identify to which file the data chunk belongs. This field could hold a string for the file name but that would make the index log too big and would introduce variable-length fields since some filenames are short and some are long. So, we just put an integer into that file_id field. Finally, we need a third file that maps that integer to a string name.

III. Method for Evaluation

The primary means of evaluation was using a benchmark application and full-scale simulation applications from two key project teams at LANL. These are detailed in the subsequent subsections.

A small effort was made to perform some basic file operations that are done by the application users in their day-to-day management of their files.

A. fs_test Benchmark

fs_test is an open source I/O pattern emulation benchmark application developed at LANL. It is freely available at https://github.com/orgs/fs-test [3].

This benchmark application can be used to emulate a real applications I/O pattern. It supports the MPI/IO, POSIX, and PLFS API I/O interfaces. For the purposes of this evaluation we used this benchmark application to generate a maximum write and read bandwidth scenario where a lot of data is written to amortize away the overhead of open/sync/close operations. We ran tests using MPI/O to compare PLFS and Lustre PFSs.

B. Silverton Applications

Silverton is a project with applications that implement a computational fluid dynamics capability developed at LANL for the study of high-speed compressible flow and high-rate material deformation. Silverton's applications implement a three-dimensional Eulerian finite difference code, solving problems with a wide variety of EOSs (equations of state), material strength, and explosive modeling options [4].

Scientists simulate large-scale physical phenomena using this suite of applications, and have done so for many years. Consequently, there are many problems that exist for the applications that can provide intense bursts of checkpoint I/O. The applications have a handful of I/O types from which to choose. One of them is MPI/IO N-1 with node aggregation. That is, all MPI ranks on a node send their data to one rank, which aggregates the data and performs the I/O on behalf of the node's MPI ranks. On the ACES Cray XE6 this meant one of every 16 MPI ranks doing checkpoint I/O. This I/O type improves the N-1 performance by amortizing fewer open/sync/close operations over more data.

C. EAP (Eulerian Applications Project) Applications

EAP is a project with applications such as xRAGE (Radiation Adaptive Grid Eulerian), which is a radiation-hydrodynamics code using a Godunov solver on an Eulerian mesh with an AMR (Adaptive Mesh Refinement) algorithm, and a radiation diffusion algorithm. It solves the Euler equations for compressible gas dynamics using finite-volume methods. It has been used to study fluid flow in highly distorted systems [5].

This suite of applications has been evolving at LANL for many decades. They are the true workhorse applications at LANL for this type of computation. Here too, there are many, many well-understood problems that generate appropriately sized checkpoint I/O for our comparison purposes. Furthermore, there was a recent problem of interest that generated one of the largest checkpoint files we'd been able to use, at 23 TiB. This file was a good test of the Lustre PFS, PLFS, and our archive. These applications also have a handful of I/O types from which to choose. One of them is a custom I/O model that was modified for the Cray XE6's Lustre PFS to stripe a file over 160 OSTs (Object Storage Targets, which are logical storage units, for example, RAIDx LUN or a Zpool) and set the stripe size (the number of bytes written to an OST before moving to the next OST) to a large value. This I/O method uses one writer per OST over which the file is striped. Each writer aggregates data until it has one stripe width of data buffered and then writes it to an OST. By this manner, each writer writes the maximum data allowed in a stripe to its own OST with each write, thus maximizing its I/O bandwidth. It was a great challenge for PLFS, a general-purpose solution, to compare against this I/O method, a custom solution that is tuned to maximize bandwidth for each PFS to which it is ported. PLFS was used for these applications' MPI/IO method where each process does its own I/O.

Also, using the 23 TiB file was a great opportunity for us to measure how PLFS's performance scales when we combine multiple PFSs into one virtual PFS. The ACES Cray XE6 has divided its PFS capability into three Lustre PFSs: lscratch2 is a ¼-sized PFS; lscratch3 is a ½-sized PFS; and lscratch4 is another ¼-sized PFS. PLFS, unlike any other I/O mechanism of which we are aware, allows us to combine these three PFSs into one very large virtual PFS.

D. Basic File Operations

Users commonly need to adjust their files. The most common operations, other than simple listings (e.g. ls) are renaming (mv), copying (cp), removing (rm or unlink) and archiving (hsi). Simple experiments were conducted on files up to 2.2 TiB to compare PLFS and Lustre PFSs' performance for these operations.

IV. Evaluation Results

The Lustre file system has mitigated the effects of N-1 through OST striping. By telling Lustre to lay the file out over many OSTs and how much to write to each OST before moving to the next an application can concurrently engage many of the Lustre file system components. For example, suppose an application chooses to aggregate I/O to 160 writing processes and sets up so that each process writes 12 MiB each time it writes. The application can tell Lustre that it wants the file striped over 160 OSTs and to write 12 MiB stripe sizes. This can be done through ioctl (I/O Control) calls or interactively. Interactively it is done with this lfs (Lustre file system) command:

```
% lfs setstripe -c 160 -s 12M <filename>
```

The main issue with this is that it is file system-specific. When the application runs on a system with a different PFS the means for achieving the highest possible I/O rate changes. PLFS removes this issue by being configured to get the best performance from the underlying PFS while providing the same I/O interface to the application from system to system.

A. fs_test Benchmark

We used fs_test to generate a maximum write and read bandwidth scenario where a lot of data is written to amortize away the overhead of open/sync/close operations. Each process wrote/read 48 MiB per write/read for 5 minutes. We ran tests using MPI/O to compare PLFS and Lustre PFSs. fs_test provides performance results for "Effective" and "Raw" bandwidths. The former includes the overhead time for the open/sync/close operations as well as the write and read operations. This is indicative of the user experience. The latter excludes the open/sync/close overhead time. It is an indication of the write and read bandwidths that the PFSs can achieve.

Figures 3 - 6 show the fs_test results. There are a couple of missing measurements where jobs were not run because of system resource demands for the real applications. Specifically, these were the 32768 and 65536 processors runs for N-1.

These experiments were run while other jobs ran. Consequently, they were competing for PFS resources. The measurements were taken using the ACES Cray XE6 lscratch3 PFS.

It appears that the 32768 processors run for PLFS N-1 saw reduced performance due to another application probably doing I/O at the same time.

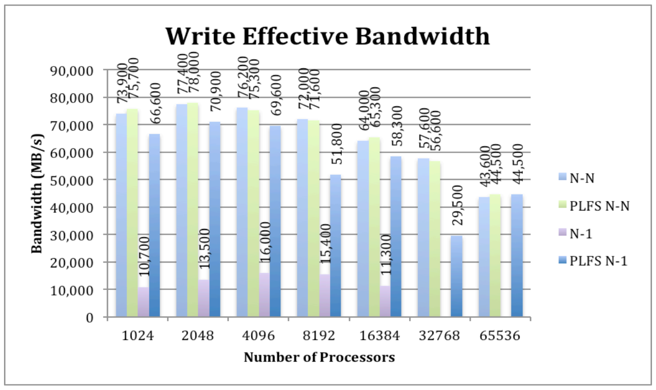

Figure 3: fs_test Maximum Effective Write Bandwidth Results

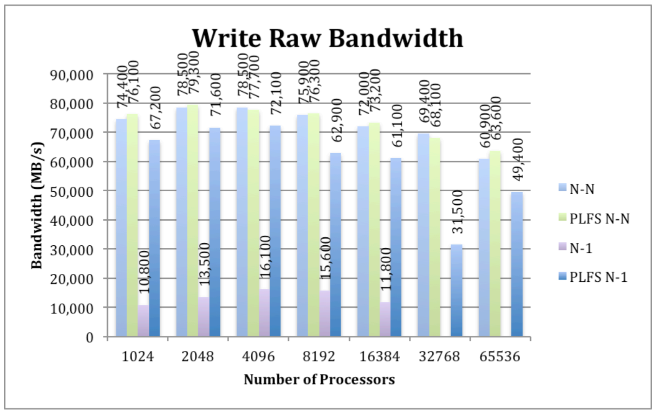

Figure 4: fs_test Maximum Raw Write Bandwidth Results

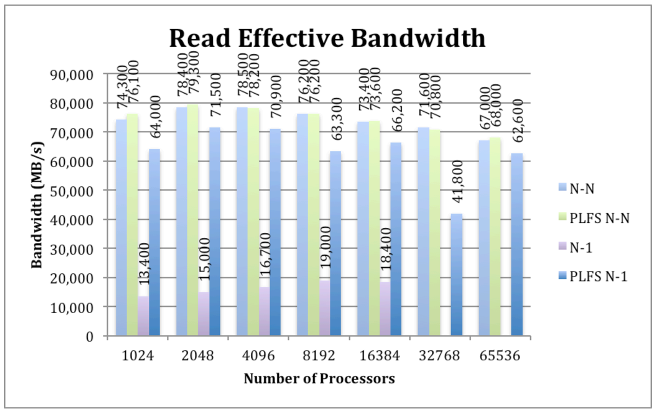

Figure 5: fs_test Maximum Effective Read Bandwidth Results

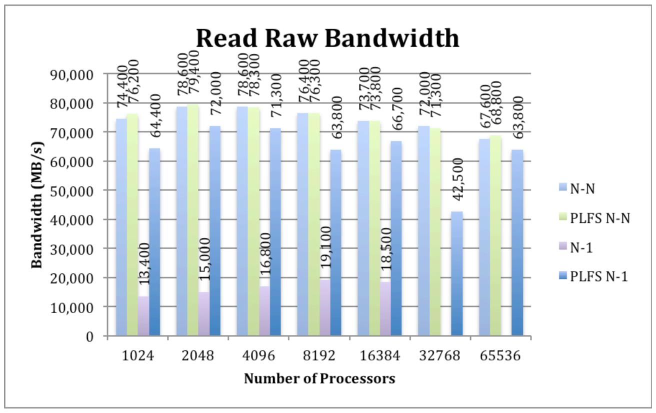

Figure 6: fs_test Maximum Raw Read Bandwidth Results

In this scenario we observe that PLFS N-N and Lustre N-N performance is nearly identical. More importantly we observe that PLFS N-1 significantly outperforms Lustre N-1, and what is more, it is generally within 10% of the PLFS and Lustre N-N performance.

These results indicated to us that PLFS might do very well at improving the N-1 performance of real applications. And so we moved on to experiment with the Silverton and EAP applications.

B. *Silverton Applicatons*

The first order of business was to modify the Silverton applications' shared code that opens a file via MPI/IO. This involved a very small change to declare some temporary variables and prepend the required PLFS prefix for MPI/IO, "plfs:" to the filename to be opened. Next we enhanced Silverton's setup script to define the environment to point to the patched MPI and PLFS installations that support PLFS use. Finally we had to add a reference to ${PLFS_LDFLAGS} to Silverton's Makefiles so that the applications linked in the proper libraries.

Once we had applications that could use PLFS, it was time to make some measurements. We started with a smaller problem to test the waters.

Silverton's I/O mode, "mpiio", is N-1 where data is aggregated to a single writer on each node and that writer writes 1 MiB each write. To maximize its performance we set the stripe count to 137 because this was the number we recalled being used by EAP's BulkIO (it was actually 136), and the stripe size to 1 MiB. We later changed to the stripe count to the maximum allowed value of 160, but that did not significantly change the results.

These experiments were run while other jobs ran. Consequently, they were competing for PFS resources. The measurements were taken using the ACES Cray XE6 lscratch3 PFS. The results were promising.

Figures 7 - 8 show the write bandwidth results for the two largest output file types in a Silverton application simulation. The restart file was 1.15 TiB and the graphics file was 215.33 GiB.

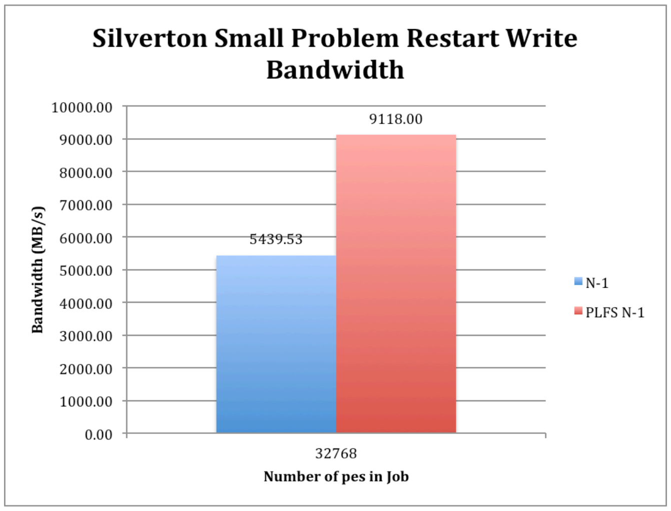

Figure 7: Silverton Small Restart Write Bandwidth Results

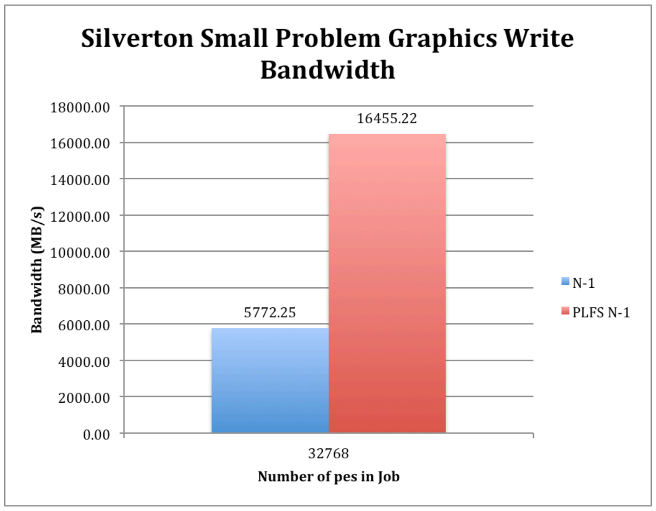

Figure 8: Silverton Small Graphics Write Bandwidth Results

PLFS outperformed Lustre by 1.68x for the restart file and by 2.85x for the graphics file.

The graphics file is processed by another application. We briefly experimented with that process for the large problem and those results will be discussed where large file results are presented.

We did restart the simulation from the restart file. PLFS read performance was better than Lustre by 1.28x. See Figure 9.

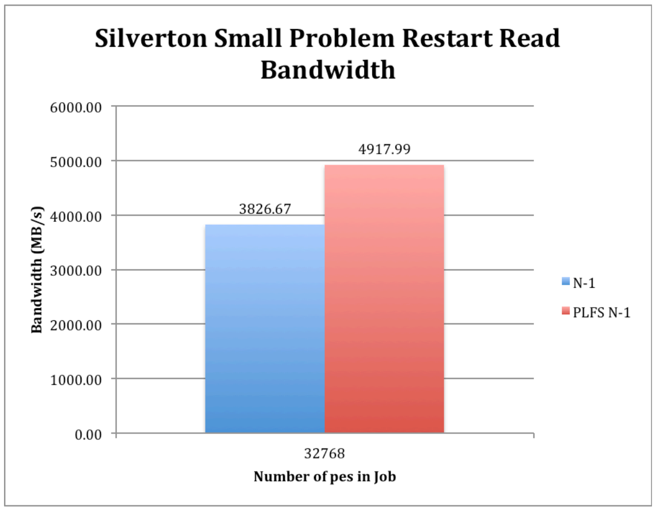

Figure 9: Silverton Small Restart Read Bandwidth Results

Encouraged by the results with the small test problem, we wanted to try a larger problem. Unable to acquire access to a large active calculation, we worked with a Silverton developer to create a problem that would generate large restart dumps, ~17 TiB at 32K pes.

Silverton has a N-1 mode called, "mpiio_ctg". Each process has a segment of the N-1 file where it writes its results contiguously. Generically, we call this N-1 segmented. We widely striped this file to a stripe count of 160 and used the same 1 MiB stripe size.

We ran an apples-to-apples comparison for this problem on lscratch3. We compared Silverton's N-1 ("mpiio"), N-1 segmented ("mpiio_ctg"), and N-N ("mpiio_fms"), all for PLFS and Lustre. We discovered a bug in Silverton's N-1 segmented mode for the graphics file, but we show the results for the restart file in figures 10 and 11. Figure 10 shows the average performance over a handful of runs (6 for 512 processors with a file size of 274.63 GiB; 6 for 4096 processors with a file size of 2.15 TiB; and 3 for 32768 processors with a file size of 17.16 TiB). Because performance variance can occur in a non-dedicated system, in Figure 11 we show the maximum write bandwidth as an indicator of potential I/O rates.

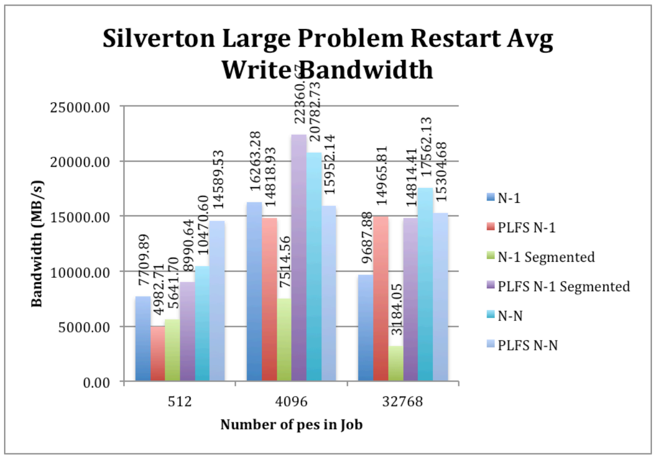

Figure 10: Silverton Large Average Restart Write Bandwidth Results

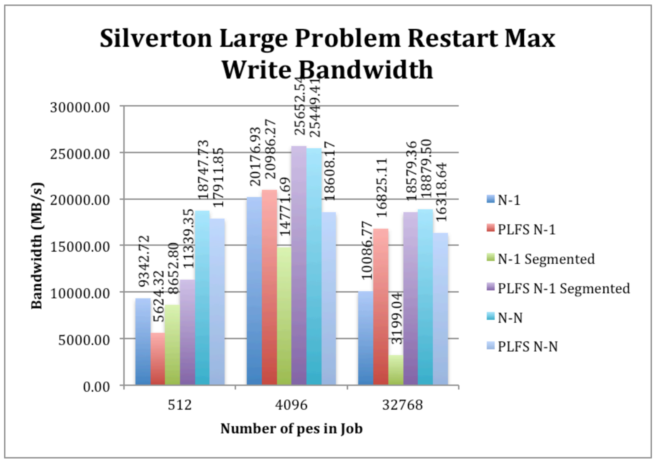

Figure 11: Silverton Large Restart Maximum Write Bandwidth Results

At 512 processors, N-1 outperformed PLFS N-1. By 32768 processors PLFS N-1's advantage was 1.5x. This is likely because at 512 processors there were only 32 writers, which is well under the OST count for the file, so it was not suffering the concurrent access problem. By 4096 processors (256 writers) there were slightly more writers than OSTs, and at 32768 processors (2048 writers) there were many more writers than OSTs. PLFS N-1 segmented always outperformed N-1 segmented. At 32768 processors its performance advantage was 4.7x. PLFS N-N is generally in the same performance regime as N-N. This is likely because the concurrence problem for traditional N-1 was more present since the number of writers was equal to the number of processors, so 16x the number of writers as with N-1.

The shapes of the curves are similar, though at 512 processors N-N has more of an advantage over PLFS. However, as the processor count grows, especially at 32768 processors, PLFS is even closer to being the same as N-N.

In order to visualize Silverton graphics files with Ensight they must be converted to the Ensight format. The Silverton gd_es application does this. The test problem started with one 76 GiB graphics file and produced one 2.4 GiB file, sixteen 2.2 GiB files, one 400 byte file, and one 11 KiB file. The time for this conversion on Lustre was 00:05:37, and on PLFS it took 00:06:49, a 1.2x advantage for Lustre. There were issues getting multiple Ensight servers working and the data set was too large for a single server, so we were unable to time the Ensight ingestion times.

C. *EAP Applications*

EAP had previously modified its code and input deck parameters to allow for PLFS use. The extent of the changes was to introduce a hint key parameter that sets the prefix, "plfs:" that is prepended to the file specification when the target PFS is PLFS. EAP has a custom cshrc file that sets up the build environment. The only changes necessary here are to load a MPI module that has been patched for PLFS and load the PLFS module.

Initial measurements were made with the EAP Asteroid test problem, which generates a 627.93 GiB restart file. We compared BulkIO, MPIIND, and PLFS through MPIIND. a BulkIO is a custom N-1 that works as previously described. MPIIND is N-1 where each process does its own I/O. This problem does one run that writes a restart dump 20 times. The measurements are all directly from the EAP code's output. The measurements were taken using the ACES Cray XE6 lscratch4 PFS The results were promising.

The experiments were run while other jobs ran. Consequently, they were competing for PFS resources.

In Figure 12 we see that PLFS MPIIND outperformed BulkIO by 3.5x and MPIIND by 3.8x for writes, on average over 20 restart dumps.

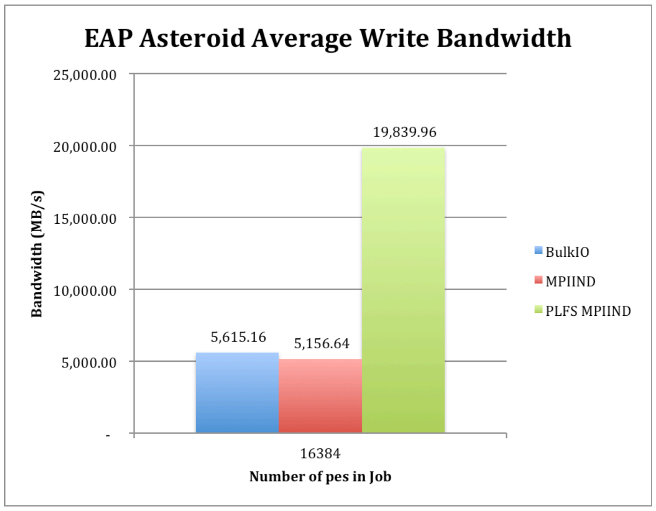

Figure 12: EAP Asteroid Restart Average Write Bandwidth

Because of the aforementioned performance variance that occurs in a non-dedicated system, we looked at the maximum write bandwidth as an indicator of potential I/O rates. PLFS still had a slight advantage. Figure 13 shows that the maximum write bandwidth for PLFS MPIIND outperformed BulkIO by 1.1x and MPIIND by 1.94x.

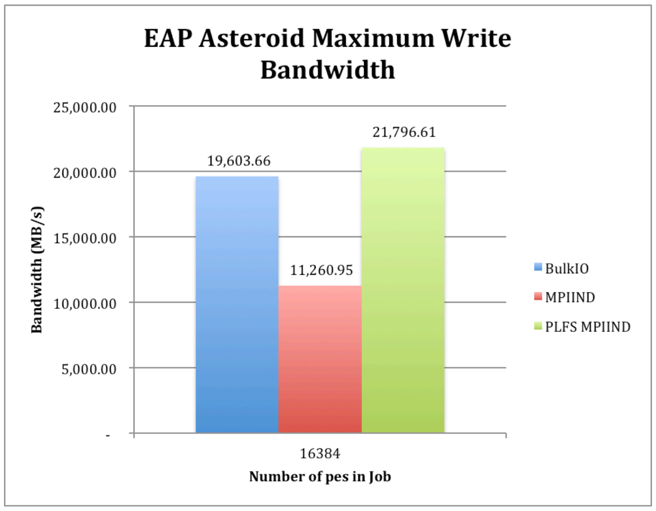

Figure 13: EAP Asteroid Restart Maximum Write Bandwidth

We did restart the Asteroid problem from the restart file. PLFS read performance was better than BulkIO by 1.58x, and relatively equal to MPIIND. See Figure 14.

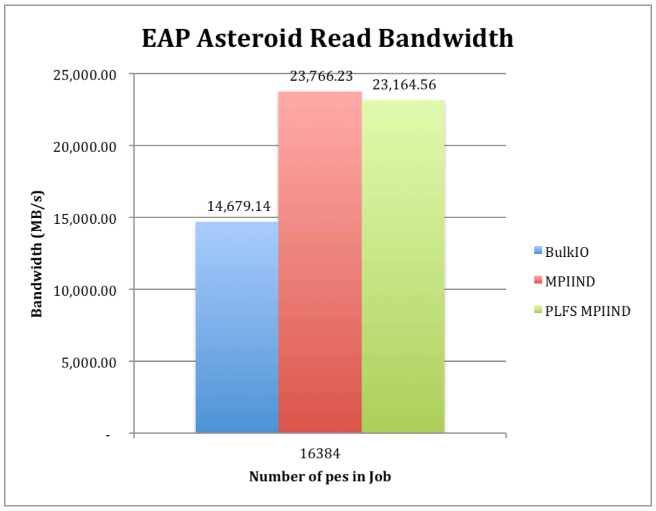

Figure 14: EAP Asteroid Restart Read Bandwidth

Encouraged by the results with the Asteroid problem, we wanted to try a large-scale problem from an active calculation. The EAP personnel provided an input deck and restart dump of 23 TiB that was well into its calculation so that it wasn't sparse. We call this problem "MD" for Marcus Daniels, the EAP developer who helped us to run this problem.

We ran an apples-to-apples comparison for this problem on lscratch2. We wanted to see how BulkIO did with the largest PFS we could use, so we ran this problem on lscratch3. Finally we wanted to assess PLFS MPIIND on the largest PFS we could use for it. Since PLFS has the ability to combine multiple PFSs into a single virtual PFS, we configured a combination of lscratch2, lscratch3, and lscratch4 for that run.

Using lscratch2 PLFS MPIIND outperformed BulkIO by 1.5x. When we targeted the larger lscratch3 with BulkIO we gained some performance. Targeting the virtual PFS with PLFS MPIIND used 2x the PFS hardware and outperformed BulkIO on lscratch3 by 3.1x. This would suggest that for this problem we should expect to see the 1.5x apples-to-apples performance gain for PLFS MPIIND over BulkIO. See Figure 15.

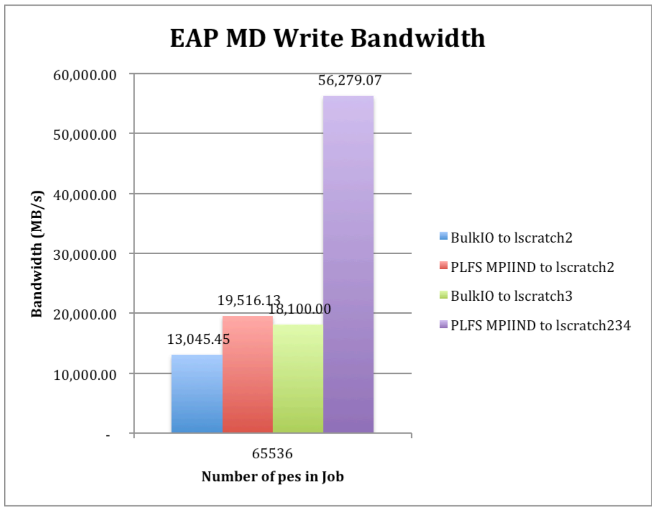

Figure 15: EAP MD Restart Write Bandwidth

Initially we could not restart from the PLFS MPIIND "MD" restart file because the index was so large that the read would not complete before memory was exhausted. Our work on this problem is discussed in the Scalable Index Management subsection of the Lessons Learned from the Evaluation section. We implemented a workaround that assumed we were restarting with the same number of processors that wrote the file and then we were able to successfully read the restart file. Nevertheless, we could not continue with "MD" as it reached a point where a memory exhaustion problem in the application was encountered. This was an application error, not an I/O issue.

While PLFS MPPIND outperformed BulkIO by 2.1x on lscratch2 for read bandwidth, and BulkIO increased its performance by doubling the PFS hardware by using lscratch3, we did not see the PLFS MPIIND increased read performance scale by quadrupling the PFS hardware when we combined lscratches 2, 3, and 4 into a single virtual PFS. Read results are shown in Figure 16.

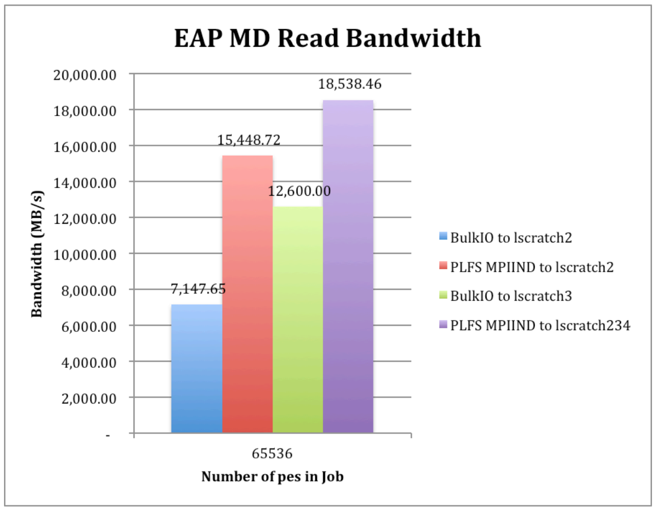

Figure 16: EAP MD Restart Read Bandwidth

We did not assess the usability of EAP's data inspection and analysis tools when the files are on a PLFS PFS. However, we do not expect to see anything different than we saw with the same experiments we did with Silverton's files on a PLFS PFS. Those results are documented in the next subsection, Basic File Operations., of this section.

D. *Basic File Operations*

Users commonly need to adjust their files. The most common operations, other than simple listings (e.g. *ls*) are renaming (*mv*), copying (*cp*), removing (*rm* or *unlink*), and archiving (*psi*, parallel storage interface, and *hsi*, hierarchical storage interface). Simple experiments were conducted selecting from 2.2 TiB, 601 GiB, and 18 GiB Silverton files to compare these operations on PLFS and Lustre.

In order to do these operations a mount point is required for the PLFS PFS. These mount points are provided using FUSE.

There was no distinguishable difference renaming a file. On both PLFS and Lustre PFSs the operation took ~1 second.

We tested all four combinations of copying a file: Lustre to Lustre (L-L), PLFS to PLFS (P-P), Lustre to PLFS (L-P), and PLFS to Lustre (P-L). The copy operations for the two larger files were taking a very long time, so we used the smallest file, 18 GiB. The L-L copy was the fastest so we normalize that to 1.0x. The P-P copy was 9.5x. The L-P copy was 8.7x. The P-L copy was 5x. After completing the archive testing we made a discovery about the size of the FUSE buffers that increased the archive performance that will be discussed in the archive operation performance results. We have not had the opportunity to repeat the copy operations, but we expect this change to help the simple copy operations in PLFS be comparable to Lustre.

The Linux *rm* command does a lot more work involving metadata server interaction for the file that is removed than the *unlink* command. Since PLFS converts a single file into many files, it was believed that it would pay an overhead price due to its many files versus the one file that would be on a Lustre file system. So, we tested *rm* and *unlink*, where *unlink* just removes the file's entry in the metadata server rather than first stat'ing, then removing the entry, then stat'ing again to ensure it is gone. The 601 GiB file was removed from PLFS and Lustre in less than 1 second. Unlink operations were even faster, 0.02 seconds.

To test archive operations we began with the LANL-developed *psi*. Initially, archiving from PLFS to HPSS (High Performance Storage System, LANL's archive system) was 14x slower than archiving from Lustre to HPSS.

LANL was in the process of converting from using *psi* to *hsi*. Working with Michael Gleicher, developer and maintainer of *hsi*, and David Sherrill, a LANL archive developer, we were able to determine the sizes of *hsi's* buffers and I/O requests to the PFS, which were 8 MiB and 4 MiB, respectively. Using PLFS debug tools we were able to see that FUSE was receiving 128 KiB I/O requests and asking PLFS to read that much at a time. That means that hsi was issuing two 4 MiB I/O requests to fill its buffer, which were turned into sixty-four 128 KiB I/O requests to the actual PFS.

Through a LANL colleague, H. B. Chen, we learned how to patch the Linux kernel and FUSE kernel module to use larger buffers that matched the buffers being used by PLFS client applications. A few members of the LANL HPC System Administration team integrated these changes into their configuration management system and produced a FTA (File Transfer Agent, a separate system of nodes that LANL uses solely to move files between file systems or between file systems and archive) with Linux and FUSE patched to use 4 MiB buffers so that we could measure transfers from PLFS to HPSS and show that these transfers were actually faster than transfers from Lustre to HPSS by 1.36x - 2.39x, depending on the size of the file transferred.

V. Lessons Learned from the Evaluation

There are a couple known shortcomings that prevent PLFS from being used as a production application PFS capability. These are overcoming the lack of performance of some basic file operations using Linux or third-party utilities on PLFS FUSE mount points and the lack of a scalable index management capability.

A. *Basic File Operations*

We think we have solved this issue by increasing the FUSE read/write buffer sizes. We need to prove this through testing.

Unfortunately, FUSE read/write buffer sizes can not be changed using a configuration file, but rather source code patches and a Linux kernel and FUSE kernel module rebuild are required. It is impractical to rewrite every Linux or third-party utility to use the PLFS API and avoid using PLFS FUSE mount points.

We highly recommend patching the Linux kernel and FUSE kernel module to use at least 4 MiB read/write buffers. By this manner the performance of basic file operations commands is greatly improved to be on par with or better than for non-PLFS PFSs. PLFS will not do an I/O request larger than the size used in any individual write when writing the file. In order to get the maximum benefit, the PLFS threadpool_size configuration parameter in the PLFSRC file should be set so that the enough threads are spawned to concurrently fill the buffer by dividing the buffer size by the size of the individual writes.

That said, the threadpool_size is on a per-request basis. So if multiple threads issue reads, then PLFS will create a separate thread pool for each thread doing reads. Therefore, it is possible that a highly concurrent workload on top of PLFS will result in PLFS spawning a crippling number of threads. Caveat emptor.

B. Scalable Index Management

PLFS's major drawback is that in order to read a file, every processor in the parallel job must read the entire index into memory. When this was done with the 23 TiB file, the index consumed ~750 MiB per processor and the application soon exhausted the node's DRAM. In order to read larger files, which tend to have larger indexes (though if an application does many small writes the index can also be large), a scalable index management capability is required. It is not scalable for each processor to read the entire index into memory.

We are currently in the assessment phase of what appears to be a promising approach to address this problem, MDHIM (Multi-Dimensional Hashing Indexing Middleware).

MDHIM is a parallel key/value store framework written in MPI. Unlike other big data solutions, MDHIM has been designed for an HPC environment and to take advantage of high-speed networks [6]. It provides a high-performance, distributed index look-up capability that scales because it does not require an application to load the entire index into memory and because it can distribute the indexes over a non-fixed number of servers.

VI. Summary

We observed as much as 5x improvement in write performance for the benchmark application. We observed a 1.5x write performance improvement for the Silverton and EAP applications when using lscratch3, and a 3x write performance improvement when we doubled the size of the file system for PLFS, as PLFS is capable of combining multiple file systems into one larger virtual file system.

While some work remains to make PLFS production-ready, it shows great promise to provide an application and underlying PFS-agnostic means of allowing programmers to use N-1 and obtain near N-N performance without maintaining custom I/O implementations.

Acknowledgment

We wish to thank the DOE's (Department of Energy) NNSA (National Nuclear Security Administration) ASC (Advanced Simulation & Computing) Program at LANL for funding this work.

We are grateful to our CRADA (Cooperative Research and Development Agreement) partner, EMC, for its contribution to furthering PLFS capabilities.

We thank CMU's (Carnegie Mellon University) PDL (Parallel Data Laboratory), under the direction of Professor Garth Gibson, for its contributions to PLFS.

Cray has been supportive by integrating the PLFS MPI ADIO interface to Cray's proprietary MPI.

The following people were contributors to the work that was done to write this paper.

PLFS: Aaron Torres, David Bonnie, Alfred Torrez, David Shrader, Chuck Cranor (CMU), John Bent (EMC), EMC's Engineering Team, and William Tucker (Cray).

Smog/Cielito/Cielo and Archive Support: H. B. Chen, Tim Harrington, Kathleen Kelly, David Sherrill, Quellyn Snead, and Jim Williams.

Silverton: Rob Kelsey, Martin Torrey, and Wayne Weseloh.

EAP: Marcus Daniels, Dave Nystrom, Mike McKay, Bob Weaver and Chuck Wingate.

Viz: Paul Weber.

HSI: Michael Gleicher, of Gleicher Enterprises.

FUSE: H. B. Chen, LANL.

References

[1] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, M. Wingate. "PLFS: A Checkpoint Filesystem for Parallel Applications", SC 2009, Nov. 2009.

[2] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, A. Torrez. "Storage Challenges at Los Alamos National Lab", 28th IEEE Symposium on Massive Storage Systems and Technologies, MSST 2012, 2012.

[3] J. Bent, D. Bonnie, G. Grider, B. Kettering, M. McClelland, J. Nunez, D. Shrader, A. Torres, A. Torrez. fs_test with problem setup and execution environment, https://github.com/orgs/fs-test, 2014.

[4] W. Weseloh. Standard description of the Silverton Project's code, Jan. 2014.

[5] E.S. Dodd, J.H. Schmidt, J.H. Cooley. "A base-line model for direct-drive ICF implosions in the xRAGE code", 55th Annual Meeting of the APS Division of Plasma Physics, Volume 58 Number 16, Nov. 2013.

[6] G. Grider, H. Greenberg. MDHIM, https://github.com/mdhim.