Evaluating Progressive File Layouts For Lustre

Richard Mohr
University of Tennessee - Knoxville
Knoxville, Tennessee, U.S.A
rmohr@utk.edu

Michael J. Brim, Sarp Oral
Oak Ridge National Laboratory
Oak Ridge, Tennessee, U.S.A
{brimmj, oralhs}@ornl.gov

Andreas Dilger
Intel Corporation
Calgary, AB, Canada
andreas.dilger@intel.com

***Abstract*—Progressive File Layout (PFL) is a Lustre feature currently being developed to support multiple striping patterns within the same file. A file can be created with several non-overlapping extents, and each extent can have different Lustre striping parameters. In this paper, we discuss evaluation results for an early PFL prototype implementation. Results for streaming I/O tests showed performance for PFL files was comparable to or better than files with standard Lustre striping. When compared to synthetic dynamic striping, PFL files had better streaming I/O performance. Additionally, object placement tests showed that a single PFL layout could be used for files of widely varying sizes while still producing an object distribution equivalent to customized stripe layouts. Overall, initial test results indicate that PFL has the ability to provide a level of performance across a variety of file sizes that is not achievable with standard Lustre striping today.**

***Index Terms*—Lustre, parallel file system, file striping.**

1. Introduction

Lustre is a popular choice for data storage in High Performance Computing because of its scalable parallel nature. By striping file contents across many disks (and often across many servers), high I/O bandwiths can be achieved. This implementation is particularly well-suited for application checkpointing and large shared datasets where sequential I/O patterns are the predominant use case. For such use cases, some current Lustre deployments have demonstrated streaming bandwidths of over 1 TiB/second [1].

The striping pattern for a Lustre file is primarily defined by two parameters: stripe count and stripe size. The stripe count controls how many Object Storage Targets (OSTs) the file will be striped across, while the stripe size controls how much data is written to each OST object in the file's layout before proceeding to the next OST object in RAID-0 fashion. Both parameters must be specified when the file is created and cannot be changed afterwards. If an application's file access pattern is well-known, and the number of clients accessing the file is fixed, the stripe count and stripe size can often be chosen to maximize performance. However, as diverse HPC workloads such as data analytics become more prevalent, choosing an optimal striping pattern can become difficult. For example, a file's contents may consist of a small header followed by larger contiguous data chunks. The header might be frequently accessed by many clients using small random reads while the larger data chunks are read by only a few clients in a sequential fashion. A single choice of stripe count and stripe size for the entire file may not be adequate to serve two very distinct access patterns. Difficulties can also arise if an estimate of the file's total size is not known prior to file creation. Very large files will need to have large stripe counts to avoid hitting space limitations on individual OSTs and to increase file I/O bandwidth. However, choosing a large stripe count for a small file is unnecessary and could incur performance penalties. Since the stripe count cannot be changed after the file is created, how then does one choose an appropriate value if the size of the file is not known a priori?

To help address some of these issues, Lustre files need a more flexible way of specifying layouts. Under OpenSFS contract SFS-DEV-003, the Intel High Performance Data Division produced a new design [2] as part of the Layout Enhancement Design project. This design introduced the concept of a "composite layout" consisting of a series of components, each of which covers a specified extent of the file with its own striping configuration. The composite layout design forms the basis for a new Lustre feature called Progressive File Layout (PFL).

2. Progressive File Layout Prototype

The Progressive File Layout feature simplifies the use of Lustre so that users can expect reasonable performance for a variety of normal file I/O patterns without the need to explicitly understand their I/O model or Lustre usage details in advance. In particular, users do not necessarily need to know the total size or concurrency of output files in advance of their creation in order to achieve good performance for

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

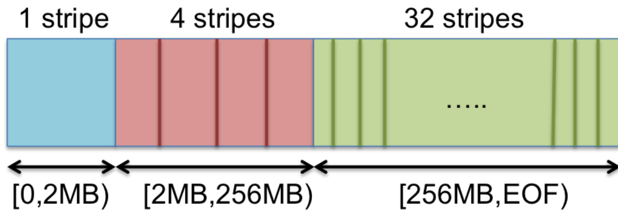


Figure 1. Example PFL Layout

both highly concurrent shared-single-file I/O or for parallel I/O to many smaller per-process files.

The PFL feature is implemented using composite layouts for regular files. A composite layout allows a file to have multiple components, each of which allows specifying a different layout (stripe count, stripe size, etc.) for non-overlapping extents of the file. This allows the user or application to progressively increase the number of OST objects across which a file is striped as the size of the file increases beyond specific thresholds. By using a PFL layout, small files can have only one (or a few) stripes allocated for the start of the file, while large files can grow to have stripes allocated across many or all OSTs in the filesystem. Since the definition of "small" and "large" files can vary by site, user, or application, it is possible to specify intermediate stages between a small file and a large file so that the number of stripes in the file can increase in a step-wise manner as the size grows. The number of sub-layouts within a PFL file, as well as the number of stripes in each, is tunable by the user though still subject to implementation and environmental limitations such as maximum layout size, number of available OSTs, and alignment of component extents to stripe boundaries. An example of a 3-component PFL layout for a file is shown in Figure 1.

A PFL prototype implementation was developed in the first half of 2015 to allow evaluation of the implementation complexity and potential performance improvements that could be achieved using this approach. This prototype allows exploration of the IO performance characteristics of composite file layouts and provides a better understanding of the changes required to the client and server layout handling code. During the prototype phase, a full design of the PFL architecture was developed so that implementation could proceed in multiple phases while ensuring that the code for each phase was usable by the subsequent phase. The ability to create, read, write, and unlink files with one or more components as specified by the user is implemented. This allows the client to read and write PFL files with different striping parameters for each component, and the MDS server to create and destroy files with multiple components. No OST changes were necessary for this phase of implementation.

To avoid complexity in the prototype, there is no ability to automatically allocate objects for uninitialized components while the file is being written. For testing purposes, the layout needs to be extended incrementally in userspace before each component of the file is written. Also, the prototype does not have the ability to specify a filesystem-wide or parent-directory composite layout template that is inherited by newly created files. Instead, each PFL file needs to be explicitly created with the desired composite layout. The prototype also does not include support for The Lustre file system checker (LFSCK) to detect and correct inconsistencies in composite layouts.

While the evaluation tests described in this paper used layouts with progressively increasing stripe counts, it should be noted that this is not a requirement for PFL. The PFL design allows the striping parameters of each component to be specified individually, so a layout with progressively decreasing stripe counts or a layout with variable stripe counts is perfectly valid. For real world scenarios, it is expected that the PFL layout will be dictated by the requirements of the user's application.

3. Evaluation

3.1. Experimental Platforms

3.1.1. LLNL Hyperion. The LLNL Hyperion testbed [3] contains 32 clients consisting of Dell 6220 servers with 16 cores or Dell 6100 servers with 8 cores. The nineteen Lustre servers have 2.60 GHz Intel Xeon E5-2670 CPUs and 64 GiB of RAM. All clients and servers contain a Mellanox DDR IB HCA. For storage, ten NetApp E-5460 controllers are used with sixty 3 TB drives each. The MDT is configured as a 10.9 TB RAID-10 4+4 device. Each OST is configured as a 21.8 TB RAID-6 8+2 device, and there are 2-3 OSTs per OSS with 52 OSTs in total. The MDT and OSTs are formatted with ldiskfs, and all clients and servers run Lustre 2.7.52 with PFL patches.

3.1.2. ORNL Lustre Testbed. The ORNL Lustre testbed consists of 35 Dell R720 servers, two Mellanox SX6036 FDR Infiniband switches, and a SAN. All nodes withn the testbed run the latest version of CentOS 6. Each server has two Intel Xeon 2630v2 processors running at 2.6 GHz, a Mellanox ConnectX-3 single port FDR Infiniband HCA, and two 500 GB 7200RPM SATA hard drives.

Nineteen nodes are used as clients and have 128 GiB 1600 MHz DDR3 RAM. Clients run a PFL-patched version of Lustre 2.7.52 built with Mellanox OFED 3.1. One of the client nodes is used as a login node, two are large-memory nodes (384 GiB RAM each), and the remaining sixteen client nodes are used as compute nodes for the tait compute cluster. Four nodes are used as LNET routers and include an additional ConnectX-3 FDR HCA. Two nodes are used as MDS/MGS nodes in a fail-over pair with six 15,000 RPM enterprise SAS drives used for MDT storage. The remaining eight nodes are used as OSS nodes, each having a quad-port 8 Gb Qlogic fibre channel HBA connected to the SAN.

The SAN is composed of 10 pairs of LSI 2680 controllers with each OSS server interfacing with one pair of controllers. Each controller pair frontends two shelves of twelve drives each, and all drives are presented as individual block devices to the OSS servers. Multiple fibre channel

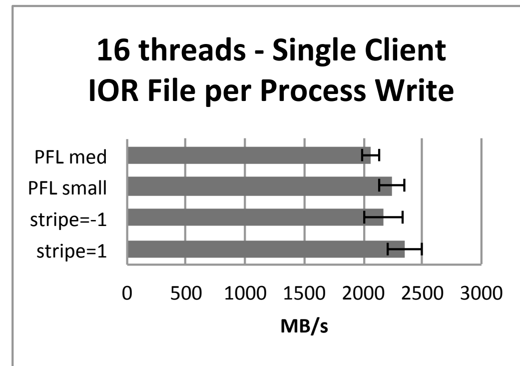
paths are used for performance and redundancy. Each OSS has 8 OSTs, and each OST is composed of a 3-drive ZFS zpool. The LNET routers have one connection to each SX6036. One SX6036 has connections to tait, and the other has connections to the Lustre server nodes.

3.2. PFL Performance on Hyperion

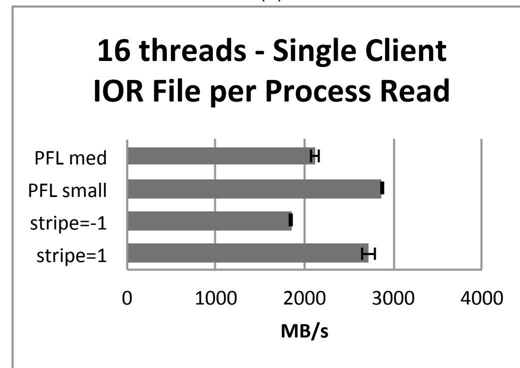
Initial performance tests for the PFL prototype were conducted on the LLNL's Hyperion test cluster. For assessing the file I/O performance, the IOR benchmark [4] was used. The IOR tests used the POSIX interface, and each test was carried out 5 times. Sixteen test threads were launched on each test client. Single client and 32 client test cases were evaluated for the file-per-process mode for read and write access. Each thread wrote or read 4 GB of data to ensure client-side caching effects were negated. For each read or write operation, a 2 MB I/O transfer size was used. Figure 2 shows the file per process results and Figure 3 shows shared single file test results. Error bars are included for each measurement at plus and minus one standard deviation of the sample population of five test runs.

For these IOR tests, three different PFL layouts were used along with two traditional layouts. The traditional Lustre file layouts used either a single OST (stripe=1) or all OSTs in the file system (stripe=-1). The "PFL small" layout has a single component that uses one OST for the entire file. This is the PFL equivalent of the "stripe=1" traditional layout. For the "PFL med" layout, two components are used. The first component uses a single OST for the first 16 MB of the file while the second component uses four OSTs for the remainder of the file. The "PFL large" layout is similar to the "PFL med" layout except that the second component stops at 128 MB, and a third component is added to use 47 OSTs for the remainder of the file. In total, the three components of the "PFL large" layout span 52 OSTs, making this layout the PFL equivalent of the "stripe=-1" traditional layout.

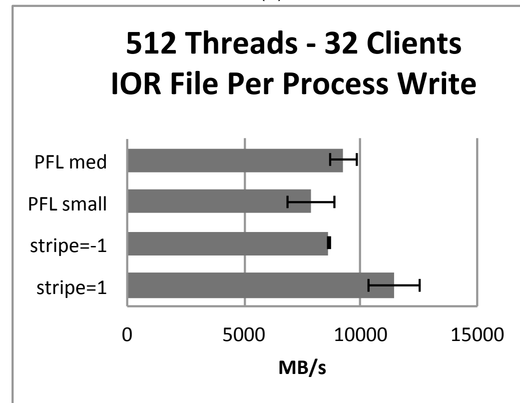
The single client file per process test shown in Figure 2a is intended to highlight that maximum performance is achieved for multiple application threads/processes writing to separate files when using only a single stripe per file. This places all of the I/O for each thread onto a single OST, and minimizes the amount of contention at each OST. The "stripe=1" results show the optimum performance possible for applications writing in this mode. The "PFL small" files also have one stripe per file, and the results show that the performance of these files is near the "stripe=1" optimal performance (within testing variance). Write performance for "PFL med" and "stripe=-1" are lower because these layouts use more OSTs, resulting in more contention. However, the impact is relatively low because there are only 16 threads writing to these files, but there are 52 OSTs to handle this load, and the writes can complete asynchronously. This is in contrast to the read performance results shown in Figure 2b where OST contention results in much lower performance for "PFL med" and "stripe=-1". As with write performance, the read performance for "PFL small" and "stripe=1" are comparable.



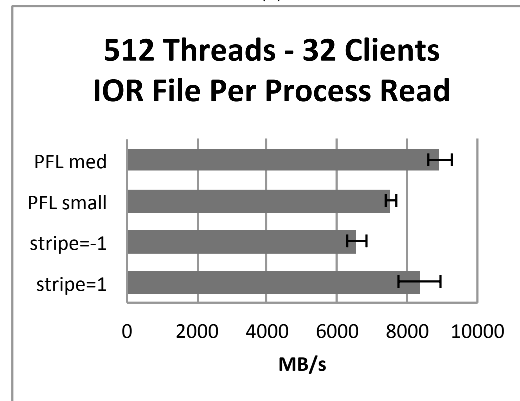
(a)



(b)

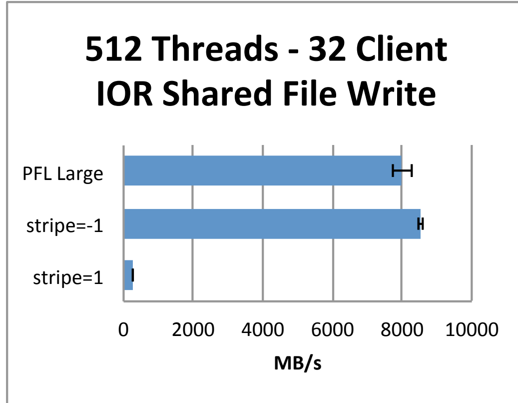


(c)

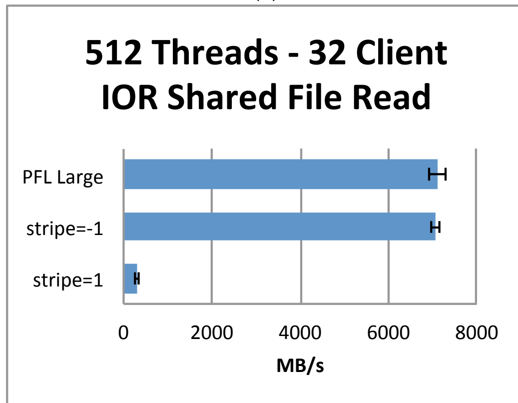


(d)

Figure 2. IOR file per process test results on LLNL's Hyperion test cluster



(a)



(b)

Figure 3. IOR single shared file test results on LLNL's Hyperion test cluster

By increasing IOR to 512 threads over 32 clients, we begin to see larger differences in performance. The results for write performance are shown in Figure 2c, and here we see a bigger discrepancy between the "stripe=1" and "stripe=-1" performance due to OST contention. Write performance for "PFL med" is better than "stripe=-1" because fewer OSTs are used for each file. The "PFL small" write performance is unusual in that it shows reduced performance compared to "stripe=1". Both show significant variance and it is believed that this variability is an artifact of the test system being used concurrently for other testing, causing the poor "PFL small" write performance. Results for read performance, seen in Figure 2d, show "PFL small" in line with "stripe=1". Once again, performance for "stripe=-1" is much lower than "stripe=1" due to OST contention.

When many clients are trying to read and write from a single large file, the performance bottleneck is based on the number of OSTs over which the file is striped. Having only a single stripe for such files (the default for Lustre) severely limits the bandwidth available to the application as can be seen in Figure 3. In order to get the maximum performance, the user or application would need to explicitly specify a higher stripe count. If the user is unaware of the need to explicitly specify a higher stripe count for the shared output file, the application I/O performance will be significantly below the possible peak value. In this case, the "stripe=1" performance is only 3.4% of the "stripe=-1" performance, which is in line with the expected 1/52 = 1.9% theoretical performance. The "PFL large" and "stripe=-1" files are striped over 52 OSTs in the filesystem. However, as this test is intended to simulate a file that has grown through the "PFL small" and "PFL med" regions, there are only 47 stripes at the end of the "PFL large" file where most of the 512 threads are writing. The expected aggregate "PFL large" performance is about 47/52 = 90% of the "stripe=-1" file. The IOR write test results show the actual performance of "PFL large" at approximately 94% of "stripe=-1". The IOR shared file read test results show a similar trend.

TABLE 1. STRIPING PATTERNS FOR WATERMARK STRIPING TESTS

Pattern Name	Stripe Layout
IOR.1	0-4 TiB: 4 stripes
IOR.2	0-4 TiB: 8 stripes
IOR.3	0-4 TiB: 16 stripes
IOR.4	0-1 TiB: 4 stripes; 1-4 TiB: 8 stripes
IOR.5	0-1 TiB: 4 stripes; 1-4 TiB: 16 stripes
IOR.6	0-1 TiB: 4 stripes; 1-2 TiB: 8 stripes; 2-4 TiB: 16 stripes

3.3. Comparing PFL to Synthetic Dynamic Striping

PFL provides the ability to assign different Lustre striping characteristics to contiguous segments of a file as it grows. In previous work [5], the ORNL Lustre testbed was used to evaluate the effects of a synthesized form of such dynamic striping using a watermark-based strategy where the stripe count is increased once a file's size exceeds one of the chosen watermarks. This evaluation did not require modifications to the Lustre client or server software, as we simulated dynamic striping by splitting files into segments at the predetermined watermarks. Segments were stored as files in directories with different striping configurations. For example, if there are two watermarks at 1 GiB and 10 GiB and dynamic striping is to be applied to a 14 GiB file, then three file segments are created from the original file representing: (1) the first 1 GiB of the file, (2) the next 9 GiB of file data between offsets 1 GiB and 10 GiB, and (3) the remaining 4 GiB of the file beyond the 10 GiB offset. Each file segment is then written to a directory having a specific stripe count and stripe width. Benchmark applications were modified to use a simple I/O interposition layer that makes the collection of file segments appear as a single file.

One of the benchmark applications used was IOR. For the IOR tests, six different striping patterns were used: three with standard striping and three with dynamic watermark striping. These striping patterns are listed in Table 1. Each IOR test used 64 processes spread evenly across 16 nodes to read and write a shared 4 TiB file with a 64 GiB block size. This same test was run five times for each striping pattern.

With the recent PFL prototype implementation, it is possible to run these same tests, but instead of splitting a single file across several directories with different stripe

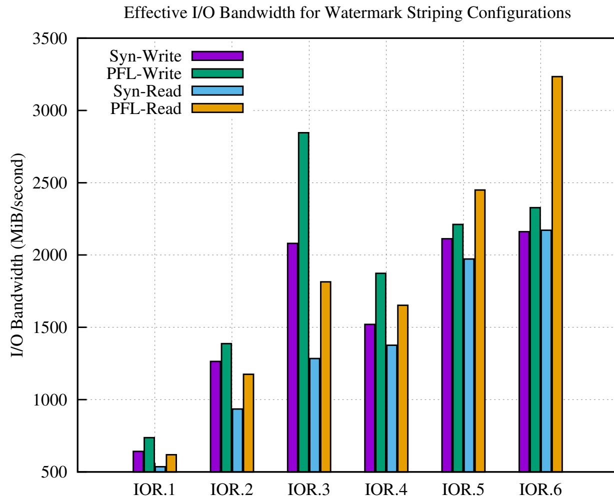


Figure 4. Performance Comparison of Synthetic Dynamic Striping vs. PFL

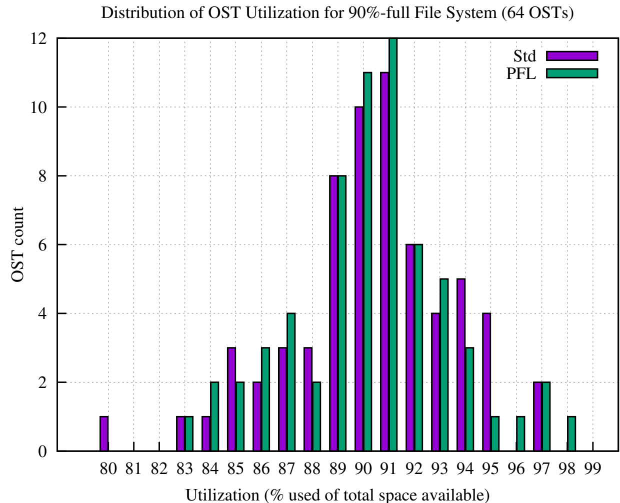


Figure 5. Comparison of Object Placement Distribution for Standard Striping vs. PFL

counts, we can now use a PFL layout to achieve varying stripe counts within the same file. The performance of PFL can then be compared to the previous results of synthetic dynamic striping. This comparison is shown in Figure 4. The values shown represent the median values of the five runs for each IOR test.

Although the data shows that PFL outperforms synthetic striping for both reads and writes, care must be taken when making such a direct comparison. First, the versions of Lustre used in these tests were different. For synthetic striping, the clients used Lustre 2.5.2 while the servers ran Lustre 2.5.3. For PFL striping, all clients and servers ran a patched version of Lustre 2.7.52. Lustre 2.7 contains some performance improvements which might affect the results. Additionally, the MDS server for the PFL test was configured to use only a round-robin allocation scheme when assigning OSTs to components of a file. This differs from the default allocation scheme used by Lustre which tends to select OSTs that have the most free space available. The main conclusion to draw from this data is that the performance of the PFL protoype follows the same overall pattern as the synthetic dynamic striping tests. As more stripes are added to the file, performance increases.

3.4. PFL Object Placement

One issue commonly seen when managing a Lustre file system is the use of poor striping patterns for files. Users do not always choose file stripe counts that are appropriate for the size of the file being created. Often the files have a much smaller stripe count than is recommended for large files which can lead to imbalanced usage of certain OSTs, or in the worst cases, can cause OSTs to completely fill up. Likewise, it is possible that a user may choose to use a very large stripe count for a small file which can be detrimental to performance. These sub-optimal file layouts

TABLE 2. Distribution of File Sizes

File Size	Percentage	Stripe Count
1 MB	70%	1
64 MB	20%	4
128 GB	9%	16
4 TB	1%	48

are not necessarily the result of conscious choices by the user, but simply result from the use of inherited file layouts.

With PFL, it is possible to define a file layout that uses an increasing stripe count for different extents of the same file making it suitable for a wide variety of file sizes. To test how well this might work, we begin by specifying a file size distribution for the test. Based on statistics from the OLCF Atlas file systems, we arrive at the approximate distribution shown in Table 2. For each file size, we also specify an ”ideal” stripe count that might be used for a traditional Lustre striping layout. These stripe counts were chosen to be suitable values for the 64-OST Lustre file system in the ORNL testbed.

A set of IOR scripts was created to produce files of the target size in relative proportion as listed in the table. These scripts were run until the file system was approximately 90% full. Information about each OST's usage (both capacity and inodes) was gathered to create a profile of this best case scenario where all files in the file system are properly striped based on file size. The test was then repeated using PFL to specify a single layout that would be used by all files regardless of size. The PFL layout components were chosen to match the file distribution. In other words, the first component of the layout used a single stripe for the first 1 MB of the file, the second component used 4 stripes for the 1-64 MB range, etc. OST usage information for this PFL test was gathered to compare with the first test case. In this way, we can begin to determine how closely a single PFL specification might match our ideal scenario.

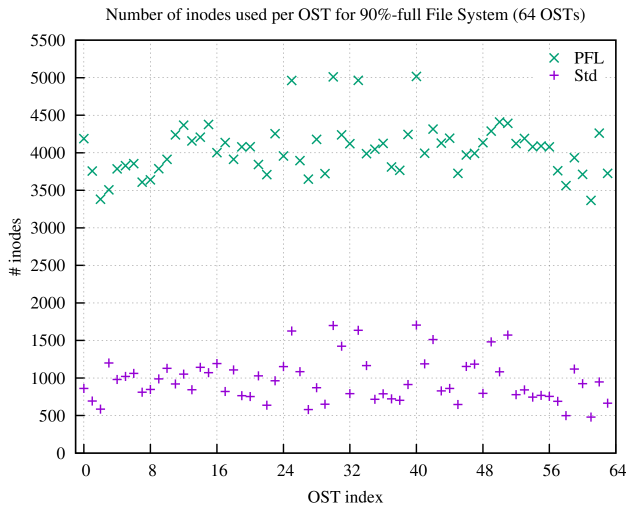


Figure 6. Comparison of Inode Distribution for Standard Striping vs. PFL

Figure 5 shows the distribution of OST usage for these two tests. While there are some variations, overall the distribtion of file objects is very similar. The traditional striping results in an average OST usage of 90.31% with a standard deviation of 3.26%. For the PFL tests, the average OST usage is 90.29% with a standard deviation of 3.10%. Based on the results of this small-scale test, it appears that using a single PFL layout for a wide variety of file sizes might result in object placement across OSTs that is near optimal.

In addition to the OST usage distribution, we can look at the number of inodes allocated on each OST. The inode distribution for the two tests is shown in Figure 6. In both cases, inode usage across the OSTs is reasonably uniform. The most obvious difference is that the inode distribution for the PFL test is significantly higher than the standard Lustre striping. This is an artifact of the current PFL prototype implementation which requires that allocation of OST objects for each layout component occur at file creation time. This means that any file, regardless of size, will have 69 OST objects allocated to it.

4. Conclusion

Progressive File Layouts provide increased flexibility in defining Lustre file striping without sacrificing performance. IOR tests show PFL performance comparable to or better than traditional Lustre striping. In particular, IOR file per process performance was similar for "PFL small" and "stripe=1" while IOR shared file performance was nearly identical for "PFL large" and "stripe=-1". This demonstrates that an appropiately chosen PFL layout could achieve nearly optimal performance for both test cases, something that is not possible with traditional Lustre layouts. Object placement test results reinforce this idea by showing that a single PFL layout used for a wide range of file sizes can optimally distribute OST objects. The current prototype implementation also exhibits the same performance characteristics that were expected based on previous synthetic striping tests.

While further work is needed to make PFL ready for production and implement missing featrures, these initial evaluation tests illustrate the possible utility of Progressive File Layouts. Advanced users will have more options for tuning Lustre file striping parameters to optimize their workloads. At the same time, naive users can expect reasonable performance over a variety of I/O workloads simply by using the same default PFL layout.

Acknowledgments

This work was supported by the United States Department of Defense (DoD) and used resources of the Computational Research and Development Programs at Oak Ridge National Laboratory. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

Portions of the benchmarking were performed on the Hyperion cluster located at Lawrence Livermore National Laboratory, supported by the Department of Energy's National Nuclear Security Administration.

References

[1] S. Oral, J. Simmons, J. Hill, D. Leverman, F. Wang, M. Ezell, R. Miller, D. Fuller, R. Gunasekaran, Y. Kim *et al.*, "Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 217–228.

[2] J. Hammond and A. Dilger, "Layout enhancement high level design," February 2014. [Online]. Available: http://wiki.lustre.org/Layout_Enhancement

[3] "Hyperion project." [Online]. Available: https://hyperionproject.llnl.gov

[4] W. Loewe, T. McLarty, and C. Morrone, "Interleaved or random (IOR) - parallel filesystem I/O benchmark." [Online]. Available: https://github.com/LLNL/ior

[5] J. Reed *et al.*, "Evaluating dynamic file striping for lustre," in *International Workshop on the Lustre Ecosystem: Challenges and Opportunities*, March 2015.