Towards an Understanding of the Performance of MPI-IO in Lustre File Systems

Jeremy Logan[1], Phillip Dickens[2]

Department of Computer Science, University of Maine
Orono, Maine, USA

[1] jeremy.logan@maine.edu
[2] dickens@umcs.maine.edu

***Abstract*—Lustre is becoming an increasingly important file system for large-scale computing clusters. The problem, however, is that many data-intensive applications use MPI-IO for their I/O requirements, and MPI-IO performs poorly in a Lustre file system environment. While this poor performance has been well documented, the reasons for such performance are currently not well understood. Our research suggests that the primary performance issues have to do with the assumptions underpinning most of the parallel I/O optimizations implemented in MPI-IO, which do not appear to hold in a Lustre environment. Perhaps the most important assumption is that optimal performance is obtained by performing large, contiguous I/O operations. However, the research results presented in this poster show that this is often the worst approach to take in a Lustre file system. In fact, we found that the best performance is often achieved when each process performs a series of smaller, non-contiguous I/O requests. In this poster, we provide experimental results supporting these non-intuitive ideas, and provide alternative approaches that significantly enhance the performance of MPI-IO in a Lustre file system.**

I. INTRODUCTION

Large-scale computing clusters with hundreds to tens of thousands of processors are being increasingly used to execute large, data-intensive applications in several scientific domains. Such domains include, for example, high-resolution simulation of natural phenomenon, large-scale image analysis, climate modelling, and complex financial modelling. The I/O requirements of such applications can be staggering, ranging from terabytes to petabytes and beyond, and managing such massive data sets has become a significant bottleneck in application performance. Thus solving this I/O scalability problem has become a critical challenge in high-performance computing.

This issue has led to the development of powerful parallel file systems that can provide tremendous aggregate storage capacity and highly concurrent access to the underlying data (e.g., Lustre [1], GPFS [15], Panasas [7]). Another important research path has been the development of parallel I/O interfaces with highperformance implementations that can work with the file system API to optimise access to the underlying storage. An important combination of file system/parallel I/O interface is Lustre, an object-based, parallel file system developed for extreme-scale computing clusters, and MPI-IO [5], which is generally considered to be the most widely-used parallel I/O API. The problem, however, is that there is currently no implementation of the MPI-IO standard that is optimised for the Lustre file system, and the performance of current implementations is, by and large, quite poor [3, 12, 21]. Given the wide spread use of MPI-IO, and the expanding utilization of the Lustre file system, it is important to provide an MPI-IO implementation that can provide high-performance, scalable I/O to MPI applications executing in the Lustre file system environment.

There are two key challenges associated with achieving high performance in a Lustre environment. First, Lustre exports only the POSIX file system API, which was not designed for a parallel I/O environment and provides little support for parallel I/O optimizations. This has led to the development of approaches (or "workarounds") that can circumvent (most of) the performance problems inherent in POSIX-based file systems and provide significantly enhanced performance (e.g., two-phase I/O[17, 18], data-sieving[20], DataType I/O [10]). The second problem is that the assumptions upon which most of these optimizations are based do not hold in a Lustre environment.

The most important and widely held assumption, and the primary focus of this poster, is that performing large, contiguous I/O operations maximizes I/O performance. The research presented here provides evidence that this may, in fact, be the worst approach in a Lustre file system environment. In fact, the best performance may be achieved when each process performs a series of smaller, non-contiguous I/O requests.

These are clearly non-intuitive results, and one focus of this poster is to provide empirical results that support this contention. The other goal is to explore alternative implementations that can provide significantly enhanced performance. The longer-term goal of this research is to provide a high-performance implementation of MPI-IO that is optimized for the Lustre file system. Toward this end, we are integrating the results of this research into ROMIO[20], a high-performance implementation of the MPI-IO standard developed and maintained at Argonne National Laboratory. We chose to work with ROMIO for three reasons: it is the most widely used implementation of MPI-IO, it is highly portable, and it provides a powerful parallel I/O infrastructure that can be leveraged in this research.

In this poster, we investigate the performance of collective write operations in two implementations of the MPI-IO

978-1-4244-2640-9/08/$25.00 © 2008 IEEE

Accepted as a poster presentation
2008 IEEE International Conference on Cluster Computing

standard on two Lustre file systems. We focus on the collective write operations because they represent one of the most important parallel I/O optimizations defined in the MPI-IO standard, and because they have been identified as exhibiting particularly poor performance in a Lustre file system.

There are two primary contributions of this poster presentation. First, it increases our understanding of the interactions between current MPI-IO implementations, the underlying assumptions upon which they are built, and the Lustre architecture. Second, it shows how the implementation of collective I/O operations can be more closely aligned with Lustre's object-based storage architecture, resulting in significant increases in performance. We believe this poster will be of interest to a large segment of the high-performance computing community given the importance of both MPI-IO and Lustre to large-scale, scientific computing.

II. Background

The I/O requirements of parallel, data-intensive applications have become the major bottleneck in many areas of scientific computing. The reason for such poor performance has been largely attributed to the I/O access patterns exhibited by scientific applications. In particular, it has been well established that each process tends to make a large number of small I/O requests, incurring on each such request the high latency of performing I/O across a network [9, 11, 19]. However, it is often the case that in the *aggregate,* the processes are performing large, contiguous I/O operations, which historically have made much better use of the parallel I/O hardware.

MPI-IO [5], the I/O component of the MPI2 standard, was developed (in part at least) to take advantage of such global information to enhance parallel I/O performance. One of the most important mechanisms through which such global information can be obtained and leveraged is a set of *collective I/O operations*, where each process provides to the implementation information about its individual I/O request. The rich and flexible parallel I/O API defined in MPI-IO facilitates collective operations by enabling the individual processes to express complex parallel I/O access patterns in a single request (e.g., non-contiguous access patterns). Once the implementation has a picture of the global I/O request, it can combine the individual requests and submit them in a way that optimizes the particular parallel I/O subsystem.

It is generally agreed that the most widely used implementation of the MPI-IO standard is ROMIO [20], which is integrated into the MPICH2 MPI library developed and maintained at Argonne National Laboratory. ROMIO provides key optimizations for enhanced performance, and is implemented on a wide range of architectures and file systems.

The portability of ROMIO stems from an internal layer called ADIO [16] upon which ROMIO implements the MPI-IO interface. ADIO implements the file system dependent features, and is thus implemented separately for each file system (see Figure 1).

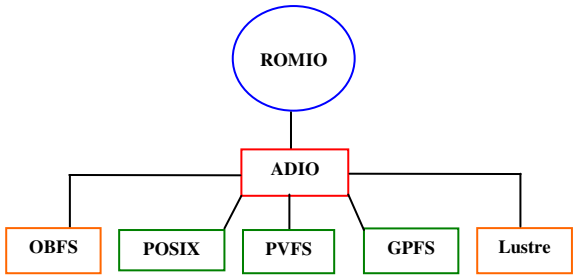

Fig. 1. ROMIO is implemented on top of ADIO, which is implemented separately for each file system.

ROMIO implements the collective I/O operations using a technique termed two-phase I/O [18, 20]. Consider a collective write operation. In the first phase, the processes exchange their individual I/O requests to determine the global request. The processes then use inter-process communication to re-distribute the data to a set of aggregator processes. The data is redistributed such that each aggregator process has a large, contiguous chunk of data that can be written to the file system in a single operation. The parallelism comes from the aggregator processes performing their writes concurrently. This is successful because it is significantly more expensive to write to the file system than it is to perform inter-process communication.

III. Lustre Architecture

Lustre consists of three primary components: file system clients (that request I/O services), object storage servers (OSSs) (that provide I/O services), and meta-data servers that manage the name space of the file system. Each OSS can support multiple Object Storage Targets (OSTs) that handle the duties of object storage and management. The scalability of Lustre is derived from two primary sources. First, file meta-data operations are de-coupled from file I/O operations. The metadata is stored separately from the file data, and once a client has obtained the meta-data it communicates directly with the OSSs in subsequent I/O operations. This provides significant parallelism because multiple clients can interact with multiple storage servers in parallel. The second driver for scalable performance is the striping of files across multiple OSTs, which provides parallel access to shared files by multiple clients.

Lustre provides APIs allowing the application to set the stripe size, the number of OSTs across which the file will be striped (the stripe width), the index of the OST in which the first stripe will be stored, and to retrieve the striping information for a given file. The stripe size is set when the file is opened and cannot be modified once set. Lustre assigns stripes to OSTs in a round-robin fashion, beginning with the designated OST index.

The POSIX file consistency semantics are enforced through a distributed locking system, where each OST acts as a lock server for the objects it controls. The locking protocol requires that a lock be obtained before any file data can be modified or written into the client-side cache. While the Lustre documentation states that the locking mechanism can be

disabled for higher performance [4], we have never observed such improvement by doing so.

A. *Known issues with Parallel I/O on Lustre*

Previous research efforts with parallel I/O on the Lustre file system have shed some light on factors contributing to the poor performance of MPI-IO, including the problems caused by I/O accesses that are not aligned on stripe boundaries [13, 14]. Figure 2 helps to illustrate the problem that arises when I/O accesses cross stripe boundaries. Assume the two processes are writing to non-overlapping sections of the file; however because the requests are not aligned on stripe boundaries, both processes are accessing different regions of stripe 1. Because of Lustre's locking protocol, each process must acquire the lock associated with the stripe, which results in unnecessary lock contention. Thus the writes to stripe 1 must be serialized, resulting in suboptimal performance.

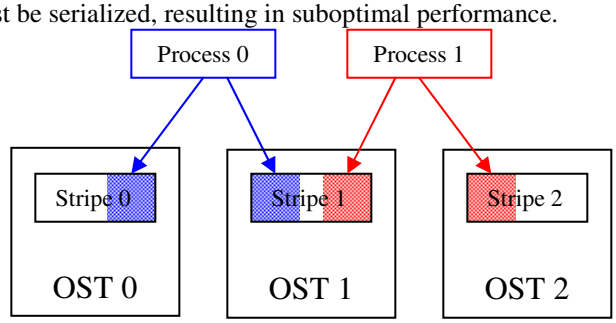

Fig. 2. Crossing Stripe Boundaries with Lustre

An ADIO driver for Lustre has recently been added to ROMIO, appearing in the 1.0.7 release of MPICH2 [6]. This new Lustre driver adds support via hints for user settable features such as Lustre striping and direct I/O. In addition, the driver insures that disk accesses are aligned on Lustre stripe boundaries.

However, our research suggests that these modifications are not sufficient to significantly improve the performance of MPIIO. This is because we believe the primary issue is the way the individual I/O requests are aggregated in a collective write operation, where the combined request is presented as large, contiguous data accesses.

The problem with performing large, contiguous writes is that it can cause significant contention at the network layer, the OSS level, and the OST level. The point may be best explained with a simple example.

Consider a two-phase collective write operation with the following parameters: four processes, a 32 MB file, a stripe size of 1 MB (within the recommended range of 1 to 4 MBs), eight OSTs, and a stripe width of eight. Assume the four processes have completed the first phase of the collective write operation, and that each process is ready to write a contiguous eight MB block to disk. Thus process P0 will write stripes 0 – 7, process P1 will write stripes 8 – 15, and so forth. This communication pattern is shown in Figure 3 below.

Two problems become apparent immediately. First, every process is communicating with every OSS. Second, every process must obtain eight locks. Thus there is significant communication overhead (each process and each OSS must multiplex four separate, concurrent communication channels), and there is contention at each lock manager for locking services (but not for the locks themselves). While this is a trivial example, one can imagine significant degradation in performance as the file size, number of processes, and number of OSTs becomes large. Thus one flaw in the assumption that performing large, contiguous I/O operations provides the best parallel I/O performance is that it does not account for the contention of file system resources (including the network).

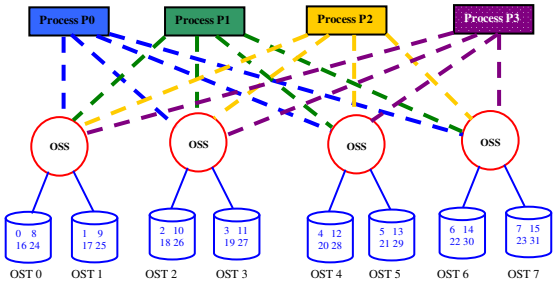

Fig. 3. Communication pattern for two-phase I/O with Lustre.

IV. Aggregation Patterns

The key question then is whether the poor performance exhibited by MPI-IO collective write operations is a result of the contention created by (in the worst case) each aggregator process communicating with each OST, and, if so, can the data aggregation patterns be modified in a way that will result in better performance. In this section, we investigate such alternative approaches.

We illustrate possible alternative approaches using a set of simple examples, and assume the following system characteristics: The file to be written is 16 MB, the stripe size is 1 MB, and there are four OSTs. We assume a stripe width of four, meaning that the stripes are allocated among the four OSTs in a round robin fashion. This is shown in Figure 4.

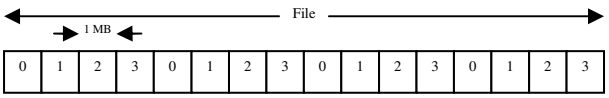

Fig 4. Allocation of Data Stripes to OSTs

In this figure, the blocks represent the individual data stripes and are labelled with the OST on which that stripe is stored. Thus, for example, data stripe 0 is stored on OST 0, stripe 5 is stored on OST 1, stripe 6 is stored on OST 2, and so forth.

Assume there are four aggregator processes, and that we are in the first phase of a two-phase collective write operation. As noted above, the current approach is to divide the file into four, contiguous (and non-overlapping) file regions, and to assign to each aggregator one such region. This pattern is shown in Figure 5.

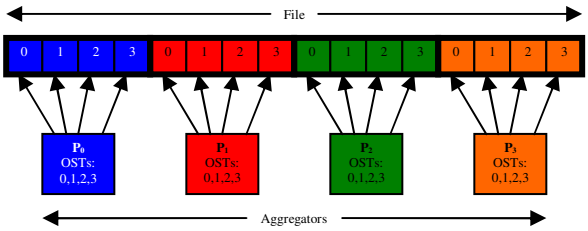

Fig. 5. Two-phase I/O file access pattern. Each processor must interact with each OST.

We can reduce the number of OSTs with which each aggregator process communicates by modifying the data aggregation pattern. Assume the size of the blocks to be written is increased to 2 MB, and that the blocks are allocated to the aggregator processes in a round robin pattern, as shown in Figure 6. In this case, each aggregator writes one block of data, skips over the next three blocks in the file (i.e., the next six data stripes), then writes its second block of data, and so forth. Thus each aggregator process is still responsible for 4 MB of data, but because of the altered write pattern, now only communicates with two OSTs rather than four. The trade-off is that each process must make two separate I/O requests to write its data to disk. Because the number of aggregators is a multiple of the number of OSTs and the block size is a multiple of the stripe size, the aggregator processes will communicate with the same two OSTs throughout the collective write operation.

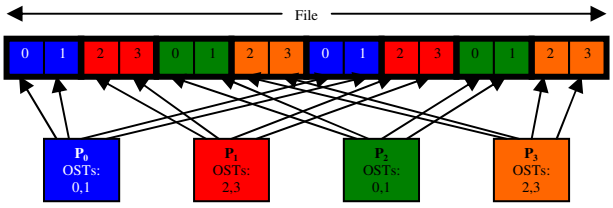

Fig. 6. Reducing the communication burden using smaller accesses. In this case, each processor communicates with two of the OSTs.

We can further reduce the number of OSTs with which an aggregator process must communicate by reducing the block size to 1 MB (the stripe size). Again, the blocks to be written are allocated round robin among the aggregators. As in the example above, each aggregator is still responsible for 4 MB of data, but only communicates with a single OST during the collective operation. The trade-off is that each aggregator must now make four individual I/O requests, each of which writes 1 MB of data to the file.

These alternative data aggregation strategies are widely known in the high-performance parallel I/O community, and are simply non-contiguous I/O operations. In fact, much of the research in parallel I/O has focused on developing alternative techniques, such as two-phase I/O, that aggregate such small, non-contiguous I/O requests to make larger, contiguous requests that are presented to the file system. In the next section, we provide experimental data comparing the performance of these approaches to parallel I/O.

It is worth noting that a seemingly simple approach to alleviating such communication overhead would be to increase the stripe size such that it matches the large contiguous accesses performed by aggregators. While this might be effective in some cases, there are two difficulties with this solution in general. First, it is only possible to adjust the striping parameters for a Lustre file at creation time, so such a strategy cannot be applied to existing files. Second, the creation of exceedingly large stripes may cause performance issues if the file is to be read by another application with a different aggregation method.

V. Experimental Design

We were interested in the impact of the data aggregation patterns on the throughput obtained when performing a collective write operation in a Lustre file system. To investigate this issue, we performed a set of experiments on two large-scale Lustre file systems at two different research facilities on the TeraGrid[8]. The Lustre file systems used in this research were located at two of the facilities on the TeraGrid: Indiana University and the Texas Advanced Computing Center at the University of Texas.

At Indiana University, we used the Big Red cluster that consisted of 768 IBM JS21 Blades, each with two dual-core PowerPC 970 MP processors and 8 GB of memory. The compute nodes were connected to Lustre through 24 Myricom 10 gigabit Ethernet cards. The Lustre file system (Data Capacitor) is mounted on Big Red, and consists of 52 Dell servers running Red Hat Enterprise Linux, 12 DataDirect Networks S29550, and 30 DataDirect Networks 48 bay SATA disk chassis, for a total capacity of 535 Terabytes. The MPI implementation was MPICH2. There were 96 OSTs on the Data Capacitor.

The other Lustre installation was Ranger, located at the Texas Advanced Computing Center (TACC) at the University of Texas. There are 3,936 SunBlade x6420 blade nodes on Ranger, each of which contains four quad-core AMD Opteron processors for a total of 62,976 cores. Each blade is running a 2.6.18.8 x86_64 Linux kernel from kernel.org. The Lustre parallel file system was built on 72 Sun x4500 disk servers, each containing 48 SATA drives for an aggregate storage capacity of 1.73 Petabytes. On the Scratch file system used in these experiments, there were 50 OSSs, each of which hosted six OSTs. The bottleneck in the system was a 1-Gigabyte per second throughput from the OSSs to the network.

We varied both the number of processes participating in the collective write operation and the data aggregation patterns. For these experiments, we categorize such patterns based on the number of OSTs with which each aggregator process communicated. Thus, for example, the write pattern shown in Figure 5 would be termed a 4-OST pattern, and that shown in Figure 6 would be categorized as a 2-OST pattern.

On Big Red, we were able to obtain 104 nodes, and, for ease of experimentation, used 52 OSTs, a 13-GB file, and experimented with 13, 26, 52, and 104 aggregator processes. On Ranger, we utilized between 50 and 200 nodes with a 50-GB file, and 50, 100, and 150 OSTs. All of the tests used one

processor per node. In these experiments, we simulated two-phase I/O by performing only the writes corresponding to the various aggregation patterns under consideration. That is, the appropriate data was assigned to each process without performing data re-distribution. We believe this to be valid based on the fact that the cost of writing to disk is orders of magnitude greater than the cost of inter-processor communication. The comparison with the existing MPI implementations (MPICH2 on Big Red and MVAPICH-2 on Ranger) was done by using a collective call to MPI_File_write_at_all. However, the data was distributed on the processes in a way that conformed to the expected aggregation pattern, and thus no data re-distribution was required in that case either. In all cases, the writes were aligned on stripe and lock boundaries.

VI. Experimental Results

The results of the experiments are shown in Figures 7 and 8. First, consider the experimental data obtained from Big Red/Data Capacitor. The two most striking results are that the 2-OST pattern consistently provided the best performance, and the 16-OST pattern and MPI consistently provided the worst performance. While the MPI results may have been affected by some additional processing in the two-phase I/O operation, it did not have to perform any data re-distribution. Thus we assume that its poor performance is due primarily to its aggregation patterns. This is supported by the fact that the 16-OST pattern, which included no additional processing, also performed quite poorly, especially compared to the other OST aggregation patterns.

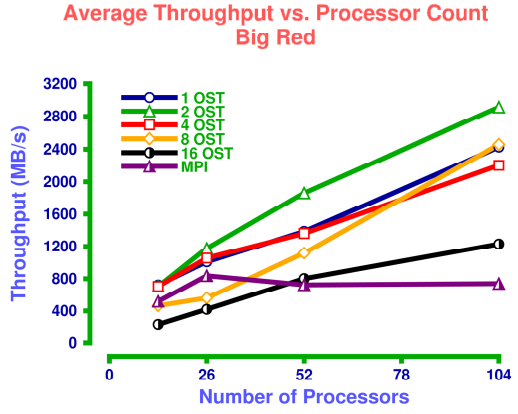

Fig. 7. Performance results from Big Red at Indiana University

The results from Ranger are even more striking. As can be seen, there was a tremendous increase in performance (compared to MPI-IO) with the 1-OST pattern when the number of aggregator processes was equal to the number of OSTs. In particular, there was a two-fold increase in performance in the 50-OST configuration, and a five-fold increase in both the 100-OST and 150-OST patterns. While MPI-IO did not necessarily provide the worst performance (generally obtained by the 16- and 32-OST patterns), its performance could be described as lacklustre in all configurations.

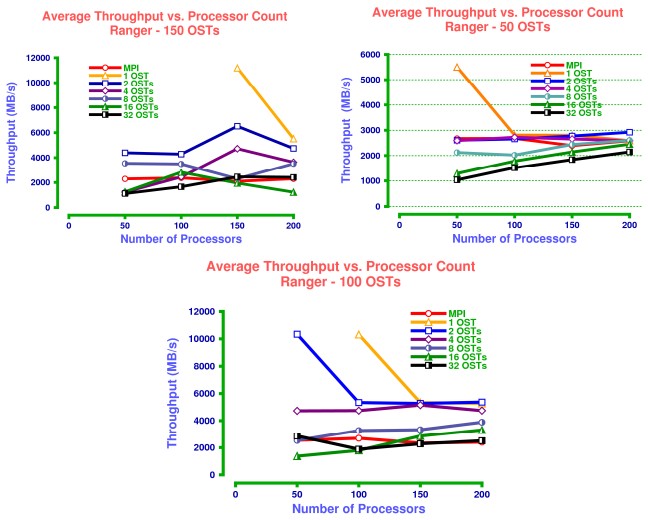

Fig. 8. Performance results from Ranger at the University of Texas

A. Discussion

The problem with performing large, contiguous I/O operations in a Lustre environment is that it leads to a many-to-many communication pattern between aggregator processes and OSTs. The fact that MPI-IO, and the 16- and 32-OST write patterns consistently provided the worst performance strongly suggests that it was, in fact, the overhead of multiple processes talking to multiple OSTs that was responsible for the poor performance. This was further supported by the fact that the 1- and 2-OST patterns provided the best performance on both architectures.

These results also lend strong support to other studies on Lustre showing that maximum I/O performance is obtained when individual processes write to independent files concurrently [4, 21]. Further, it helps explain the commonly held belief of (at least some) Lustre developers that parallel I/O is not necessary in a Lustre environment and does little to improve performance [2]. While we do not subscribe to this view, we now at least understand its origins.

These results do, however, indicate that it is worthwhile to go down a (somewhat) different path in the development of a high performance ADIO driver for Lustre. While it is certainly necessary to ensure that all I/O requests are properly aligned with the data striping and locking patterns, these results show that doing so is not sufficient to significantly improve the performance of MPI-IO. What appears to be also necessary is the awareness and management of the communication patterns between the aggregator processes and the OSTs. Our current research is focusing on the incorporation of these insights into a new ADIO driver for the Lustre file system.

VII. Related Work

The most closely related work is from Yu et al. [21], who implemented the MPI-IO collective write operations using the Lustre file-join mechanism. In this approach, the I/O processes write separate, independent files in parallel, and then merge these files using the Lustre file-join mechanism. They showed that this approach significantly improved the

performance of the collective write operation, but that the reading of a previously joined file resulted in low I/O performance. As noted by the authors, correcting this poor performance will require an optimization of the way a joined file's extent attributes are managed. The authors also provide an excellent performance study of MPI-IO on Lustre.

The approach we are pursuing does not require multiple independent writes to separate files, but does limit the number of Object Storage Targets (OST) with which a given process communicates. This maintains many of the advantages of writing to multiple independent files separately, but does not require the joining of such files.

Larkin and Fahey [12] provide an excellent analysis of Lustre's performance on the Cray XT3/XT4, and, based on such analysis, provide some guidelines to maximize I/O performance on this platform. They observed, for example, that to achieve peak performance it is necessary to use large buffer sizes, to have at least as many IO processes as OSTs, and, that at very large scale (i.e., thousands of clients), only a subset of the processes should perform I/O. While our research reaches some of the same conclusions on different architectural platforms, there are two primary distinctions. First, our research is focused on understanding of the poor performance of MPI-IO (or, more particularly, ROMIO) in a Lustre environment, and on implementing a new ADIO driver for object-based file systems such as Lustre. Second, our research is investigating both contiguous and non-contiguous access patterns while this related work focuses on contiguous access patterns only.

In [14], it was shown that aligning the data to be written with the basic striping pattern improves performance. They also showed that it was important to align on lock boundaries. This is consistent with our analysis, although we expand the scope of the analysis significantly to study the algorithms used by MPI-IO (ROMIO) and determine (at least some of) the reasons for suboptimal performance.

VIII. Conclusions and Future Research

The research presented in this poster has attempted to develop a better understanding of the poor performance of MPI-IO in Lustre file systems. We hypothesized that the problem was related to the high overhead associated with writing large, contiguous blocks of data to the file system, which results in a many-to-many communication pattern between MPI-IO aggregation processes and OSTs. We devised a set of experiments to test our hypothesis, and, based on the results, believe that this provides a very plausible explanation for the perform issues. Our current research is focused on the development of collective I/O algorithms that take such overhead costs into account when determining the most appropriate data aggregation patterns. Our longer-term goal is to incorporate such algorithms into an ADIO driver that is optimised for Lustre file systems.

Additional performance studies of Lustre itself need to be undertaken to develop a better understanding of the factors relating to the high overhead costs of communicating with multiple OSTs. Network contention and lock protocol processing are two likely causes, but there may be other contributing factors that are not currently known. Finally, we are working to develop a methodology for determining the particular OST pattern that will provide the best performance for a given architecture.

IX. Acknowledgements

This research was funded through Grant #0702748 from the National Science Foundation.

References

[1] Cluster File Systems, Inc. http://www.clustrefs.com
[2] Frequently Asked Questions. http://www.lustre.org
[3] I/O Performance Project http://wiki.lustre.org/index.php?title=IOPerformanceProject
[4] Lustre: scalable, secure, robust, highly-available cluster file system. An offshoot of AFS, CODA, and Ext2. www.lustre.org/
[5] MPI-2: Extensions to the Message-Passing Interface. Message Passing Interface Forum http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html
[6] MPICH2 Home Page. http://www.mcs.anl.gov/mpi/mpich
[7] The Panasas Home Page. http://www.panasas.com
[8] The Teragrid Project http://www.teragrid.org
[9] Avery Ching, Choudhary, A., Coloma, K., Liao, W.-k., Ross, R. and Gropp, W., Noncontiguous I/O Accesses through MPI-IO. In the *Proceedings of the Third International Symposium on Cluster Computing and the Grid (CCGrid)*, (2002), 104-111.
[10] Avery Ching, Choudhary, A., Liao, W.-k., Ross, R. and Gropp, W., Efficient Structured Access in Parallel File Systems. In the *Proceedings of the IEEE International Conference on Cluster Computing*, (2003), 326-335.
[11] Isaila, F. and Tichy, W.F., View I/O: improving the performance of non-contiguous I/O. In the *Proceedings of the IEEE Cluster Computing Conference*, (Hong Kong).
[12] Larkin, J. and Fahey, M. Guidelines for Efficient Parallel I/O on the Cray XT3/XT4 *CUG 2007*, 2007.
[13] Liao, W.-k., Ching, A., Coloma, K., Choudhary, A. and Kandemir, M., Iproving MPI Independent Write Performance Using A Two-Stage Write-Behind Buffering Method. . In the *Proceedings of the Next Generation Software (NGS) Workshop*, (2007).
[14] Liao, W.-k., Ching, A., Coloma, K., Choudhary, A. and Ward, L., An Implementation and Evaluation of Client-Side File Caching for MPI-IO. In the *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '07)*, (2007).
[15] Schmuck, F. and Haskin, R., GPFS: A shared-disk file system for large computing clusters. . In the *Proceedings of the Conference on File and Storage Technologies*, (IBM Almaden Research Center, San Jose, California).
[16] Thakur, R., Gropp, W. and Lusk, E., An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In the *Proceedings of the Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation*.
[17] Thakur, R., Gropp, W. and Lusk, E., Data Sieving and Collective I/O in ROMIO. In the *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 182-189.
[18] Thakur, R., Gropp, W. and Lusk, E., On Implementing MPI-IO Portably and with High Performance. In the *Proceedings of the Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems*, 23-32.
[19] Thakur, R., Gropp, W. and Lusk, E. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing*, *28* (1). 83-105. January, 2002.
[20] Thakur, R., Ross, R. and Gropp, W. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation, Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, May 2004.
[21] Yu, W., Vetter, J., Canon, R.S. and Jiang, S., Exploiting Lustre File Joining for Effective Collective I/O In the *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, (2007).