Towards a High Performance Implementation of MPI-IO on the Lustre File System

Phillip Dickens, Jeremy Logan

Department of Computer Science, University of Maine
Orono, Maine, USA
dickens@umcs.maine.edu
jeremy.logan@maine.edu

Abstract—Lustre is becoming an increasingly important file system for large-scale computing clusters. The problem is that many data-intensive applications use MPI-IO for their I/O requirements, and it has been well documented that MPI-IO performs poorly in a Lustre file system environment. However, the reasons for such poor performance are not currently well understood. We believe that the primary reason for poor performance is that the assumptions underpinning most of the parallel I/O optimizations implemented in MPI-IO do not hold in a Lustre environment. Perhaps the most important assumption that appears to be incorrect is that optimal performance is obtained by performing large, contiguous I/O operations. Our research suggests that this is often the worst approach to take in a Lustre file system. In fact, we found that the best performance is sometimes achieved when each process performs a series of smaller, non-contiguous I/O requests. In this paper, we provide experimental results showing that such assumptions do not apply in Lustre, and explore new approaches that appear to provide significantly better performance.

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 [20], Panasas [8]). Another important research path has been the development of parallel I/O interfaces with high-performance 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 [6], the most widely-used 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, 15, 26]. 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 with MPI-IO 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, such that the performance of MPI-IO in such environments can be significantly improved (e.g., two-phase I/O[22, 23], data-sieving[25], DataType I/O [11]). 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 paper, is that performing large, contiguous I/O operations provides the optimal 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 paper is to document that this widely held assumption is one of the primary reasons for the poor performance of MPI-IO in Lustre. The other goal of this paper 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[25], 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 paper, we investigate the performance of collective write operations in two implementations of the MPI-IO standard in 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. We are also interested in the collective write operations because they have been identified as exhibiting particularly poor performance in a Lustre file system.

There are two primary contributions of this paper. 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. This paper should 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.

The rest of this paper is organized as follows. In Section 2, we provide background information on MPI-IO and ROMIO. In Section 3, we discuss the Lustre architecture. In Section 4, we discuss different aggregation patterns in collective I/O implementations. In Section 5, we describe the experimental design, and provide our results in Section 6. In Section 7, we discuss related work, and we provide our conclusions in Section 8.

II. Background

The I/O requirements of parallel, data-intensive applications have become the major bottleneck in many areas of scientific computing. Historically, the reason for such poor performance has been 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 overhead of performing I/O across a network [10, 13, 24]. 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 [6], 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.

The most widely used implementation of the MPI-IO standard is ROMIO [25], 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 [21] 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).

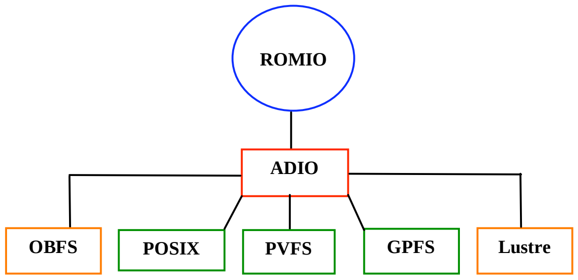

Figure 1: ROMIO is implemented on top of ADIO, which is implemented separately for each file system.

ROMIO implements the collective I/O operations using a technique termed *two-phase I/O* [23, 25]. 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 re-distributed 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.

To help clarify these ideas, consider the following example. Assume an SPMD computation where each process computes over a different region of a two-dimensional file (16 x 16 array of integers). Further, assume there are four compute nodes, four I/O nodes, and that each process has a 4 x 4 sub-array. The array is stored on disk in row-major order with a stripe unit equal to one row of the array. Also, the array is distributed among the processes in a block-block distribution as shown in Figure 2.

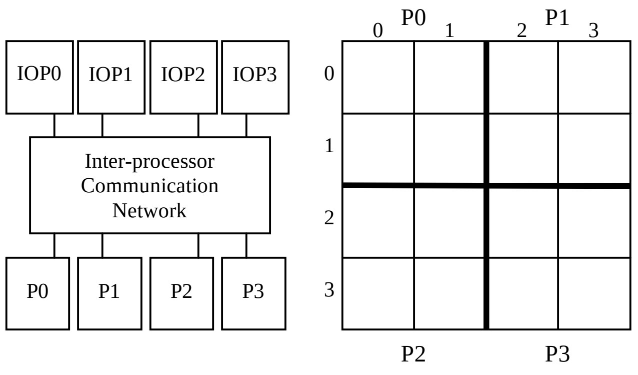

Figure 2. Example system with (a) four compute processors and four I/O processors and (b) a 4x4 array partitioned in block-block order.

Assume each process is ready to write its data to disk and enters into a collective write operation. In the first phase, the processes exchange information about their individual requests to determine the aggregate I/O request, and determine the best strategy for writing the data to disk. In this case, it is determined to be optimal for each process to write a single row of the array to disk in parallel. To implement this strategy, process P0 must send array elements (1, 0) and (1, 1) to process P1, and must receive elements (0, 2) and (0, 3) from process P1. The exchanges between processes P2 and P3 are similar. Once each process receives the data it needs, they write their portion of the data to disk in one I/O request in parallel (note that in this example each process is an aggregator).

We further explore collective write operations in the sections that follow.

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 meta-data 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 [12]. 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 [17, 18]. Figure 3 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.

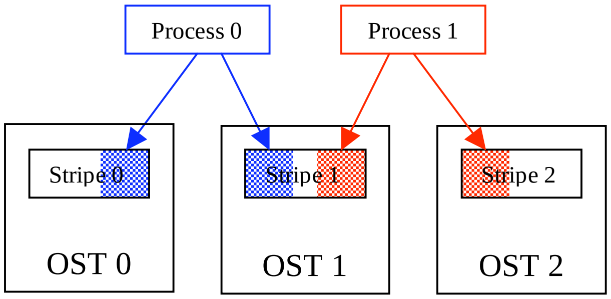

Figure 3: Crossing Stripe Boundaries with Lustre

An ADIO driver for Lustre has recently been added to ROMIO, appearing in the 1.0.7 release of MPICH2 [7]. 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 MPI-IO. 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 4 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).

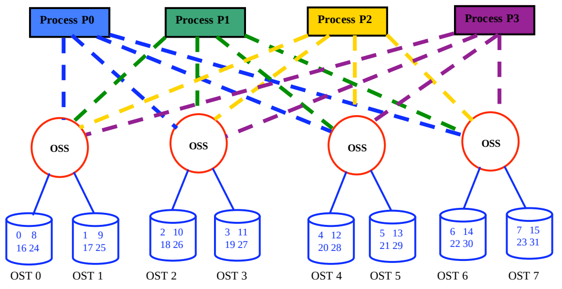

Figure 4: Communication pattern for two-phase I/O with Lustre.

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 5.

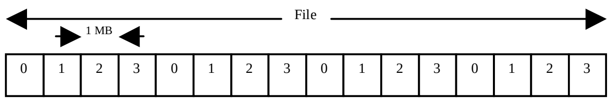

Figure 5: Allocation of Data Stripes to OSTs

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 6.

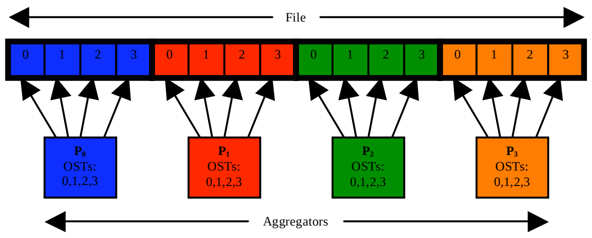

Figure 6: Two-phase I/O file access pattern. Each processor must interact with each OST.

We can reduce the number of OSTs with which each aggregator process must communicate 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. This is shown in Figure 7. 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 now each process must make two separate I/O requests to write its data to disk. It is important to emphasize that the aggregator processes communicate with the same two OSTs throughout the collective write operation. This is because the number of aggregators is a multiple of the number of OSTs, and the block size is a multiple of the stripe size.

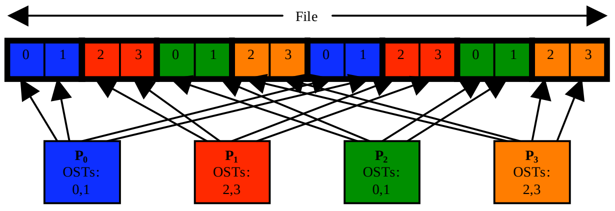

Figure 7: Reducing the communication burden using smaller accesses. In this case, each processor communicates with two of the OSTs.

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. This new write patterns is shown in Figure 8. As in the examples 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.

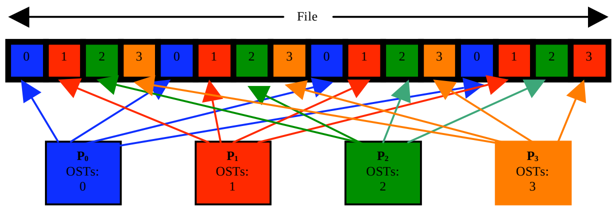

Figure 8: Minimizing communication. Each processor requires access to data on only one OST.

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 I/O 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[9]. The TeraGrid was developed with funding from the National Science Foundation, and is the world's largest open facility for scientific research. It consists of eleven research centers connected via a 40-gigabit per second backbone network. The Lustre file systems used in this research were located at two of the facilities on the TeraGrid: the National Center for Supercomputing Applications (NCSA, located at the University of Illinois at Urbana-Champaign) and Indiana University.

We used the Tungsten cluster at NCSA, which consisted of 2,560 Intel IA-32 Xeon 3.2 GHz processors divided into five sub-clusters, each containing 256 dual-processor nodes. Each of the sub-clusters had a 20-gigabit per second connection to the OSTs via a common switch, where each OST was connected to the switch via one gigabit Ethernet.

The Lustre file system on Tungsten was served by a total of 104 Object Storage Devices (OSTs), split over two mount points. The Lustre partition used in these experiments consisted of 32 OSTs. We used the ChaMPIon MPI implementation on Tungsten, which is a proprietary implementation of the MPI standard developed at MPI Software Technology.

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 used on Big Red was MPICH, which was developed at Argonne National Laboratory and utilizes ROMIO for the I/O operations. There were 96 OSTs on the Data Capacitor.

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 8 would be termed a 1-OST pattern, Figure 7 would be categorized as a 2-OST pattern, and Figure 6 would be categorized as the 4-OST pattern. We varied the write pattern between 1-OST and 32-OST.

On Tungsten, there were a total of 54 OSTs available on the partition to which we were assigned. We utilized 32 of these OSTs to simplify the experimentation (that is, to more easily modify the data aggregation patterns). We varied the number of aggregator processes from 8 to 256. On Big Red, we were able to obtain 104 nodes, and thus used 52 OSTs, and experimented with 13, 26, 52, and 104 aggregator processes. The file was 8 Gigabytes on Tungsten and 13 GB on Big Red (again chosen to simplify the experimentation). All of the tests used one processor per node, because we found that the connections between nodes and OSTs were quickly saturated, and very little performance increase was gained by using multiple processors per node for I/O.

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 (ChaMPIon on Tungsten, and MPICH on Big Red) 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 9 and 10. 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

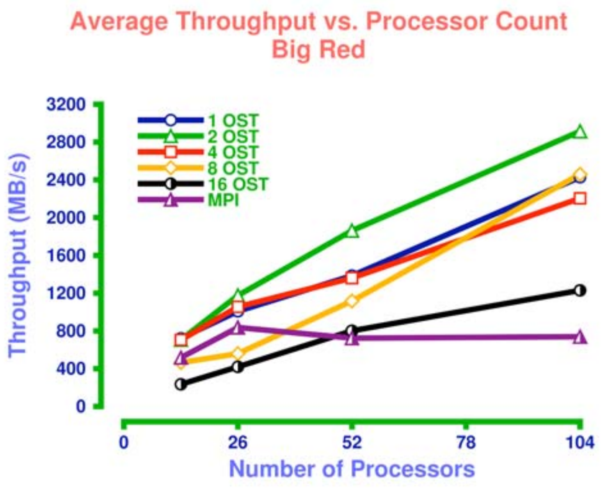

Figure 9: Performance results from Big Red at Indiana University

performed quite poorly, especially compared to the other OST aggregation patterns.

It is also interesting to consider the results obtained when there were 104 aggregator processes. In particular, in the 16-OST pattern, each aggregator process performed 16 individual I/O requests, each of which wrote 16 MB. In the 2-OST pattern, each aggregator performed 128 separate I/O requests, each of which wrote 2 MB.

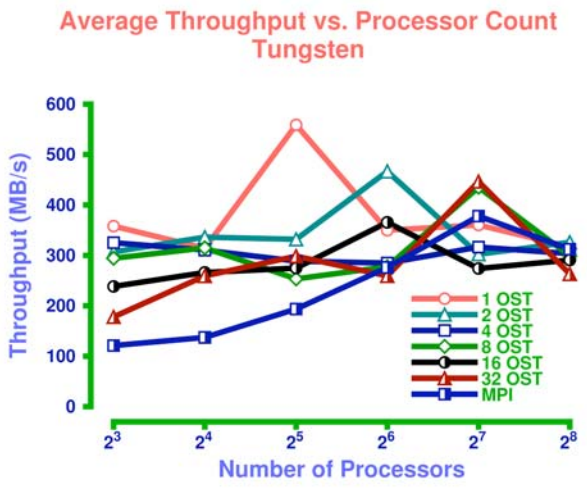

Figure 10: Performance results from Tungsten at NCSA

Now consider the results obtained on Tungsten shown in Figure 10. As can be seen, the best performance by far was obtained using the 1-OST aggregation pattern with 32 aggregator processes. It can also be observed that MPI provided the worst performance in this configuration. As the number of aggregator processes was increased beyond 32, I/O performance began to degrade. With 64 processes, the best performance was observed with the 2-OST pattern. The difference in performance between the 2-OST pattern and other patterns was still significant, but was not of the magnitude of that observed with the 1-OST pattern. When the number of aggregators was increased to 128, other aggregation patterns began to provide the best performance. However, overall performance was significantly lower than that obtained with 32 aggregator processes, and the differences in performance became much less pronounced. When the number of aggregator processes was increased to 256, I/O performance plummeted, and there was very little difference in the performance obtained from any configuration.

We believe the results obtained with up to 64 aggregation processes are the most reliable results. This is because of the way the scheduler assigns nodes to requesting jobs. In particular, it tends to allocate the vast majority of nodes within the same sub-cluster. As noted above, each such sub-cluster only had a single 20-bit gigabit connection to the OSTs. Thus once the number of aggregator processes increased beyond 64, contention for this 20-gigabit connection began to dominate the cost of performing I/O.

B. *Discussion*

In these experiments, all MPI process communicated with all OSTs. This was because each MPI process wrote a contiguous block of data to disk, the stripe size was 1 MB, the file was striped across all OSTs, and the size of the blocks was never less than the number of 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 observed poor performance. This was further supported by the fact that the 1- and 2-OST patterns provided the best performance on Tungston and Big Red respectively. Thus while it is not currently clear as to whether the 1- or 2-OST pattern should be used (we believe this is an architectural issue, where the more powerful networking infrastructure of Big Red enabled each process to communicate with more than one OST), it seems clear that communicating with a small number of OSTs is better than communicating with a large number of OSTs.

These results also lend strong support to other studies of Lustre showing that maximum performance is obtained when individual processes write to independent files concurrently [4, 26]. 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.

It is our belief that MPI-IO can provide excellent, and perhaps optimal, performance in a Lustre environment. This is because it offers considerable support for collective operations and provides tremendous flexibility. For example, the results obtained on Big Red show that a 2-OST pattern provides the best performance. Such a pattern is not possible without a parallel I/O API and supporting infrastructure. Other studies may show that the optimal pattern is, to some extent, architecture dependent. MPI-IO has the flexibility to adapt to such findings.

However, these results do indicate that it is worthwhile to go down a (somewhat) different path in developing a high-performance ADIO driver for Lustre. While it is certainly critical to ensure that all I/O requests are properly aligned with the data striping patterns, these results show that doing so is not sufficient to significantly improve the performance of MPI-IO. We base this conclusion on the fact that every experiment we performed, regardless of the aggregation pattern, was aligned with the striping patterns.

Perhaps the weakest part of this study was the size of the clusters on which we were able to experiment. The TeraGrid systems on which this study was performed are very heavily used, and it is extremely difficult to grab a large number of processors for an extended period of time. However, we have had some success in obtaining dedicated time for a short duration, and will strive to perform similar studies with significantly larger Lustre systems.

VII. Related Work

The most closely related work is from Yu et al. [26], 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.

Our approach 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. The performance analysis presented in this paper complements and extends the analysis performed by Yu et al.

Larkin and Fahey [15] 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 [18], 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 sub-optimal performance. We also show how to modify ROMIO's collective I/O algorithms to achieve significantly improved performance.

There has also been a significant research effort focused on various techniques to enhance the performance of MPI-IO. DAChe [14] is a user-space client side cache that was shown to improve the performance of ROMIO on both Lustre and GPFS [14]. Active Buffering [16] has also been shown to improve the performance of parallel write operations using local buffering and performing the I/O in the background. In [19], an object-based caching system is shown to improve the performance of ROMIO on the FLASH I/O benchmark [5]. There has also been significant research related to improving the performance of non-contiguous I/O requests in ROMIO [24], independent writes [17], and workarounds to the POSIX API (e.g., data sieving [22], two-phase I/O [23]). Our research is similarly focused on improving the performance of ROMIO in a particular file system environment. We show that many of these techniques, which provide significant improvement in other file system environments, do not perform well in an object-based environment such as Lustre. Thus this research is focused on identifying the current approaches that do not work, explaining the reason for poor performance, and showing approaches to solving the performance issues.

VIII. Conclusions and Future Research

This research was motivated by the fact that MPI-IO performs poorly in Lustre file systems, and the reasons for such performance have been largely unknown. We hypothesized that the problem was related to the high overhead associated with writing large, contiguous blocks of data to the file system, which can require the multiplexing of many concurrent communication channels. We devised a series of experiments to test our hypothesis, and, based on the results, believe that this provides a very plausible explanation.

Based on these results, we believe it is worthwhile to further explore these ideas, and to develop 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.

The results presented in this paper are limited by the relatively small size of the systems on which the experimental studies were conducted. Thus another focus of future research is to obtain access to larger systems for longer periods of time. Also, 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 not currently known. Finally, we are working to develop a methodology for determining the particular OST pattern that would provide the best performance for a given architecture.

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]. M. Zingale. FLASH I/O Benchmark Routine - Parallel HDF5, March 2001. http://flash.uchicago.edu/~zingale/flash_benchmark_io

[6]. MPI-2: Extensions to the Message-Passing Interface. Message Passing Interface Forum http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html

[7]. MPICH2 Home Page. http://www.mcs.anl.gov/mpi/mpich

[8]. The Panasas Home Page. http://www.panasas.com

[9]. The Teragrid Project http://www.teragrid.org

[10]. 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.

[11]. 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.

[12]. Bramm, P.J. The Lustre Storage Architecture

[13]. 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).

[14]. Kenin Coloma, Alok Choudhary, Wei-keng Liao, Lee Ward and ., S.T., DAChe: Direct Access Cache System for Parallel I/O. In the *Proceedings of the T the Proceedings of the 2005 International Supercomputer Conference.*

[15]. Larkin, J. and Fahey, M. Guidelines for Efficient Parallel I/O on the Cray XT3/XT4 *CUG 2007*, 2007.

[16]. Lee, J., Ma, X., Ross, R., Thakur, R. and Winslett, M., RFS: Efficient and Flexible Remote File Access for MPI-IO. In the *Proceedings of the The 2004 IEEE International Conference on Cluster Computing*, (2004).

[17]. 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).

[18]. 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 Distried Processing Symposium (IPDPS '07)*, (2007).

[19]. Logan, J. and Dickens, P., Using Object Based Files for High Performance Parallel I/O In the *Proceedings of the To Appear: IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*., (Dortmund, Germany, 2007).

[20]. 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).

[21]. 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*.

[22]. Thakur, R., Gropp, W. and Lusk, E., Data Sieving and Collective I/O in ROMIO. In the *Proceedings of the Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 182-189.

[23]. 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.

[24]. Thakur, R., Gropp, W. and Lusk, E. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing*, *28* (1). 83-105. January, 2002.

[25]. 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, Revised May 2004.

[26]. 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).