Y-Lib: A User Level Library to Increase the Performance of MPI-IO in a Lustre File System Environment

Phillip M. Dickens and Jeremy Logan

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

Abstract—It is widely known that MPI-IO performs poorly in a Lustre file system environment, although the reasons for such performance are currently not well understood. The research presented in this paper strongly supports our hypothesis that MPI-IO performs poorly in this environment because of the fundamental assumptions upon which most parallel I/O optimizations are based. In particular, it is almost universally believed that parallel I/O performance is optimized when aggregator processes perform large, contiguous I/O operations in parallel. Our research shows that this approach generally provides the worst performance in a Lustre environment, and that the best performance is often obtained when the aggregator processes perform a large number of small, non-contiguous I/O operations.

In this paper, we first demonstrate and explain these non-intuitive results. We then present a user-level library, termed Y-lib, which redistributes data in a way that conforms much more closely with the Lustre storage architecture than does the data redistribution pattern employed by MPI-IO. We then provide experimental results showing that Y-lib can increase performance between 300% and 1000% depending on the number of aggregator processes and file size. Finally, we cause MPI-IO itself to use our data redistribution scheme, and show that doing so results in an increase in performance of a similar magnitude when compared to the current MPI-IO data redistribution algorithms.

I. INTRODUCTION

Large-scale computing clusters are being increasingly utilized to execute large, data-intensive applications in several scientific domains. Such domains include 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 managing such massive data sets has become a significant bottleneck in parallel application performance.

This issue has led to the development of powerful parallel file systems that can provide tremendous aggregate storage capacity with highly concurrent access to the underlying data (e.g., Lustre [1], GPFS [15], Panasas [7]). This issue has also led to the development of parallel I/O interfaces with high-performance implementations that can interact 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], 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 critical to provide an MPI-IO implementation that can provide high-performance, scalable I/O to MPI applications executing in this 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 optimisations. This has led to the development of approaches (or "workarounds") that can circumvent (at least some of) the performance problems inherent in POSIX-based file systems (e.g., two-phase I/O [17, 18], and data-sieving[20]). The second problem is that the assumptions upon which these optimisations are based simply do not hold in a Lustre environment.

The most important and widely held assumption, and the one upon which most collective I/O optimisations are based, is that parallel I/O performance is optimised when application processes perform a small number of large, contiguous (non-overlapping) I/O operations concurrently. In fact, this is the assumption upon which collective I/O operations are based. The research presented in this paper, however, shows that this assumption can lead to very poor I/O performance in a Lustre file system environment. Moreover, we provide a large set of experimental results showing that the antithesis of this approach, where each aggregator process performs a large number of small (non-contiguous) I/O operations, can, when properly aligned with the Lustre storage architecture, provide significantly improved parallel I/O performance.

In this paper, we document and hypothesize the reasons for these non-intuitive results. In particular, we believe that it is the data aggregation patterns currently utilized in collective I/O operations, which result in large, contiguous I/O operations, that are largely responsible for the poor MPI-IO performance observed in Lustre file systems. We believe this

is problematic because it redistributes application data in a way that conforms poorly to Lustre's object-based storage architecture. Based on these ideas, we present an alternative approach, embodied in a user-level library termed Y-Lib, which, in a collective I/O operation, redistributes data in a way that more closely conforms to the Lustre object-based storage architecture. We provide experimental results, taken at a large-scale Lustre installation, showing that this alternative approach to collective I/O operations does, in fact, provide significantly enhanced parallel I/O performance.

This research is performed within the context of ROMIO [x], a high-performance implementation of the MPI-IO standard developed and maintained at Argonne National Laboratory. There are three reasons for choosing ROMIO as the parallel I/O implementation with which we compare our approach: It is generally regarded as 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 implemented in ROMIO on a large-scale Lustre installation at the University of Texas Advanced Computing Center. We focus on the collective write operations because they represent one of the most important parallel I/O optimisations defined in the MPI-IO standard and because they have been identified as exhibiting particularly poor performance in Lustre file systems.

This paper makes two primary contributions. First, it increases our understanding of the interactions between collective I/O optimisations in a very important implementation of the MPI-IO standard, the underlying assumptions upon which these optimisations are based, 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 up to a 1500% increase in performance. We believe this paper 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.

The rest of this paper is organized as follows. In Section 2, we provide background information on MPI-IO and collective I/O operations. In Section 3, we discuss the Lustre object-based storage architecture. In Section 4, we provide our experimental design, and, in Section 5, we provide our experimental results. In Section 6, we provide a discussion of our results, and provide our conclusions in Section 7.

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 the high overhead of performing I/O across a network with each such request [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).

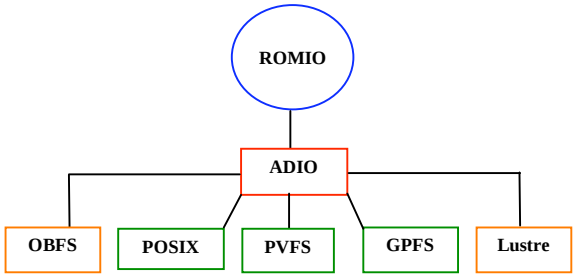

Figure 1: ROMIO is implemented on top of ADIO, which is implemented separately for each file system.

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.

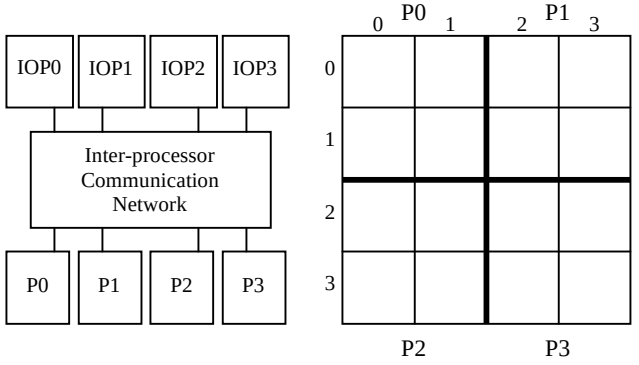

Figure 2. Example system with (a) four compute processors and four I/O processors and (b) a 4x4 array partitioned in block-block order.

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.

II. 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 [10]. 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.

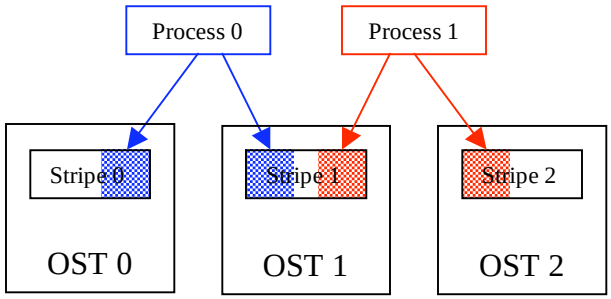

Figure 2: Crossing Stripe Boundaries with Lustre

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.

B. *Data Aggregation Patterns*

While the issues addressed by the new ADIO driver are necessary for high-performance parallel I/O in Lustre, they are not, in our view, sufficient. This is because they do not address the problems arising from multiple aggregator processes making large, contiguous I/O requests concurrently. This point may be best explained through a simple example.

Consider a two-phase collective write operation with the following parameters: four aggregator processes, a 32 MB file, a stripe size of 1 MB, 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 a primary 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 and network resources.

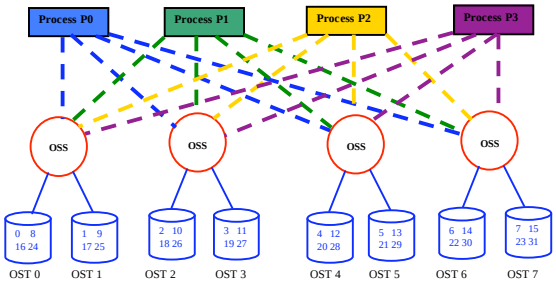

Figure 3: Communication pattern for two-phase I/O with Lustre.

III. Data Redistribution with Y-Lib

The aggregation pattern shown in Figure 3 is what we term an *all-to-all* OST pattern because it involves all aggregator processes communicating will all of the OSTs. The simplest solution is to limit the number of OSTs across which a file is striped. In fact, the recommended (and default) stripe width is four. While this certainly reduces contention, it also severely limits the parallelism of file accesses, which, in turn, limits parallel I/O performance. However, we believe it is possible to both reduce contention and maintain a high degree of parallelism, by implementing an alternative data aggregation pattern. This is accomplished via a user-level library termed Y-Lib.

The basic idea behind Y-Lib is to minimize the number of OSTs with which a given aggregator process communicates. In particular, it seeks to redistribute data in what we term a *"one-to-one"* OST pattern, where the data is arranged such that each aggregator process communicates with exactly one OST. Once the data is redistributed in this fashion, each process performs a series of non-contiguous I/O operations (in parallel) to write the data to disk. We provide a simple example to help illustrate these ideas.

Assume there are four application processes that share a 16 MB file with a stripe size of 1 MB and a stripe width of four (i.e., it is striped across four OSTs). Given these parameters, Lustre distributes the 16 stripes across the four OSTs in a round-robin pattern as shown in Figure 4. Thus stripes 0, 4, 8, and 12 are stored on OST 0, stripes 1, 5, 9, and 13 are stored on OST 1, and so forth.

Figure 4: Lustre File Layout

Figure 5(a) shows the data blocks residing on the four processes in a way that is termed the *conforming distribution* where each process can write its data to disk in a single, contiguous I/O operation. This is the distribution pattern that results from the first phase of ROMIO's collective write operations, based on the assumption that performing large, contiguous I/O operations provides optimal parallel I/O performance.

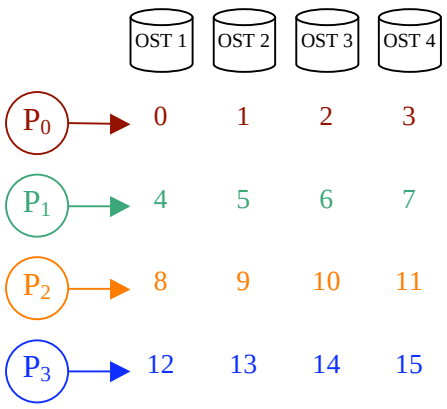

Figure 5(a): Conforming Distribution

Figure 5(b) shows how the same data would be distributed by Y-Lib to create the one-to-one OST pattern. As can be seen, the data is rearranged to reflect the way it is striped across the individual OSTs, resulting in each process having to communicate with only a single OST.

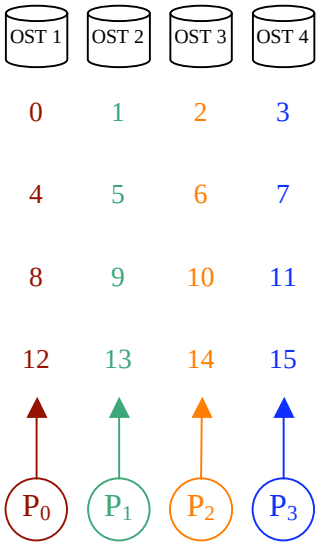

Figure 5(b): The one-to-one OST pattern

It is interesting to consider the trade-offs in these two approaches. When the data is redistributed to the conforming distribution, each process can write its data to disk in a single, contiguous, I/O operation. However, this creates a great deal of background activity as the file system client must communicate with all OSTs. In the one-to-one OST distribution, there is significantly less contention for system resources, but each process must perform a (potentially) large number of small I/O requests, with a disk seek between each such request.

Thus the relative performance of the two approaches is determined by the particular overhead costs associated with each. In the following sections, we provide extensive experimentation showing that the costs associated with contention for system resources (OSTs, lock managers, network) significantly dominates the cost of performing multiple, small, and non-contiguous I/O operations.

IV. Experimental Design

We were interested in the impact of the data aggregation patterns on the throughput obtained when performing collective I/O operations in a large-scale Lustre file system. The Lustre installation we used in this research 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, 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 three key parameters in these experiments: The implementation of the collective I/O operation, the number of processors that participated in the operation, and the file size. In particular, we varied the number of processors from 128 to 1024, where each processor wrote one Gigabyte of data to disk. Thus the file size varied between 128 Gigabytes and one Terabyte. We kept the number of OSTs constant at 128, and maintained a stripe size of one MB. Each data point represents the mean value of 50 trials taken over a five-day period.

We also investigated three different factors that impacted the performance of the collective I/O operations, which we discuss in turn.

C. Data Aggregation Patterns with Redistribution

In this set of experiments, we assigned the data to the processors in a way that required it to be redistributed to reach the desired aggregation pattern. Thus, in the case of MPI-IO, we set a file view for each process that specified the one-to-one OST pattern, and set the hint to use two-phase I/O to carry out the write operation. Similarly, we assigned the data to the processors in the conforming distribution, and made a collective call to Y-Lib to redistribute the data to the one-to-one OST pattern. Once Y-Lib completed the data redistribution, it wrote the data to disk using independent (but concurrent) write operations.

D. Data Aggregation Patterns without Redistribution

The next set of experiments assumed the data was already assigned to the processors in the required distribution. Thus in the case of MPI-IO, the processors performed the collective MPI_File_write_at_all operation, and passed to the function a contiguous one Gigabyte data buffer.

In the case of Y-Lib, the data redistribution phase was not executed, and each process performed the independent write operations assuming the data was already in the one-to-one pattern.

E. MPI-IO Write Strategies

The final set of experiments was designed to determine if we could improve the performance of MPI itself by forcing it to use the one-to-one OST pattern with independent writes. We accomplished this by setting a file view specifying the one-to-one OST pattern, and disabling both two-phase I/O and data sieving. We then compared the performance of this approach with that of MPI-IO assuming the conforming distribution, and MPI-IO assuming the one-to-one OST distribution using two-phase I/O.

V. Experimental Results

The experimental results are shown in Figures 6, 7, and 8. Figure 6 shows the throughput obtained when Y-Lib started with the data in the conforming distribution, used message passing to put it into the one-to-one OST distribution, and then wrote the data to disk with multiple, POSIX write operations. This is compared to the throughput obtained by the MPI-IO MPI_File_write_all operation when the data is initially placed in the one-to-one OST pattern. As can be seen, Y-Lib improves I/O performance by up to a factor of ten. This is particularly impressive given that each process performed 1024 independent write operations.

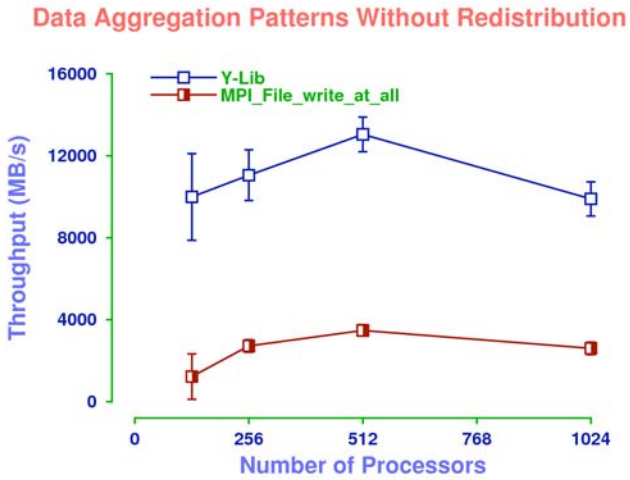

Figure 6: Data aggregation patterns without redistribution

Figure 7 shows the throughput obtained assuming the optimal data distribution for each approach. That is, the data was in the conforming distribution for MPI-IO, and in the one-to-one OST distribution for Y-Lib. Thus neither approach required the redistribution of data. As can be seen, the one-to-one pattern, which required 1024 independent write operations, significantly outperformed the MPI_File_write_at_all operation, where each process wrote a contiguous one Gigabyte buffer to disk. In this case, Y-Lib improved performance by up to a factor of three.

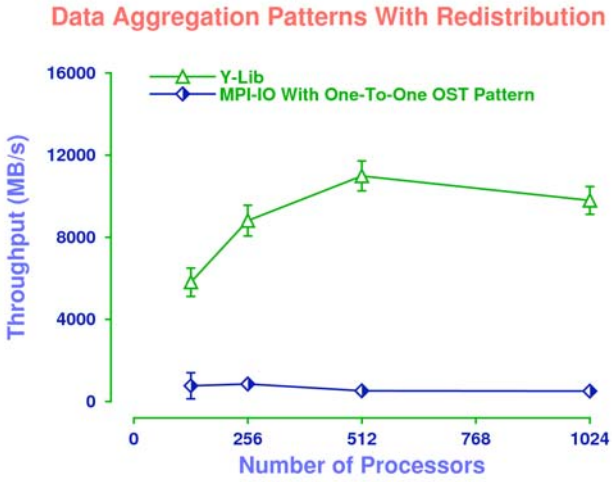

Figure 7:Data aggregation patterns with redistribution

Figure 8 depicts the performance of three different MPI-IO collective operations. It includes the two previously described approaches, and compares them with the performance of MPI-IO when it was forced to use independent writes. As can be seen, we were able to increase the performance of MPI-IO itself by over a factor of two, by forcing it to use the one-to-one OST pattern.

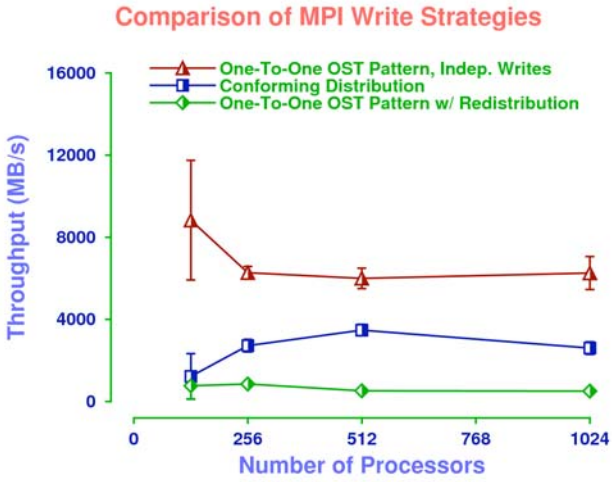

Figure 8: Comparison of MPI write strategies

VI. Discussion of Results

These results strongly support the hypothesis that the poor performance of MPI-IO in a Lustre file system environment is due in large part to the contention for system resources caused by the all-to-all communication pattern between processors and OSTs. This phenomenon arises because of the underlying assumption that parallel I/O performance is optimised by distributing data to achieve the conforming distribution, and writing the data to disk in a small number of large, contiguous I/O operations.

These results also lend strong support to other studies on Lustre showing that maximum 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]. Based on this research, we now believe that parallel I/O is, in fact, critical to high performance I/O in Lustre, but must be done in a way that is more closely aligned with the Lustre object-based storage architecture.

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

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 sub-optimal performance.

VIII. Conclusions and Future Research

This research was motivated by the fact that MPI-IO has been shown to perform poorly in a Lustre environment, the reasons for which have been heretofore 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 implemented a new approach to collective I/O operations in Lustre that significantly reduced such contention. This new approach was embodied in a user-level library termed Y-Lib, which was shown to outperform the current implementation of collective I/O operations by up to a factor of ten.

These results were obtained on one large-scale Lustre installation, and current research is focusing on implementing and evaluating Y-Lib on other Lustre file systems. If, as we expect, the results are consistent across several installations, we will implement the new data aggregation patterns in an ADIO driver that is optimised for Lustre file systems.

Another focus of current research is 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.

Reference

[1]. Cluster File Systems, Inc. http://www.clustrefs.com

[2]. Frequently Asked Questions.

[3]. I/O Performance Project http://wiki.lustre.org/index.php?title=IOPerformanceProjec

[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]. Scalable Implementation for Overlapping File Access in MPI-IO http://www.ece.northwestern.edu/~wkliao/Sandia/

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

[9]. Avery Ching, Choudhary, A., Coloma, K., Liao, W.-k., et al., 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]. Bramm, P.J. The Lustre Storage Architecture
[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., et al., 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., et al., 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).
[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 Proc. 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, Revised 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).