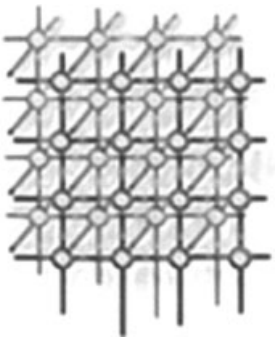
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:1433–1449
Published online 8 September 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1491



A high performance implementation of MPI-IO for a Lustre file system environment

Phillip M. Dickens*, † and Jeremy Logan

Department of Computer Science, University of Maine, Orono, ME, U.S.A.

SUMMARY

It is often the case that MPI-IO performs poorly in a Lustre file system environment, although the reasons for such performance have heretofore not been well understood. We hypothesize that such performance is a direct result 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, however, shows that this approach can actually provide the worst performance in a Lustre environment, and that the best performance may be obtained by performing a large number of small, non-contiguous I/O operations. In this paper, we provide empirical results demonstrating these non-intuitive results and explore the reasons for such unexpected performance. We present our solution to the problem, which is embodied in a user-level library termed Y-Lib, which redistributes the 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 provide a large body of experimental results, taken across two large-scale Lustre installations, demonstrating that Y-Lib outperforms MPI-IO by up to 36% on one system and 1000% on the other. We discuss the factors that impact the performance improvement obtained by Y-Lib, which include the number of aggregator processes and Object Storage Devices, as well as the power of the system's communications infrastructure. We also show that the optimal data redistribution pattern for Y-Lib is dependent upon these same factors.

Received 1 April 2009; Revised 24 June 2009; Accepted 9 July 2009

KEY WORDS: parallel I/O; grid computing; object-based file systems

*Correspondence to: Phillip M. Dickens, Department of Computer Science, University of Maine, Orono, ME, U.S.A.
†E-mail: dickens@umcs.maine.edu

Contract/grant sponsor: National Science Foundation; contract/grant number: 0702748

1. 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 (eg. Lustre [1], GPFS [2], Panasas [3]). 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 optimize 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 [4], 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 optimized for the Lustre file system, and the performance of current implementations is, by and large, quite poor [5–7]. Given the widespread 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 optimizations. 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 [8,9], and data-sieving [10]). The second problem is that the assumptions upon which these optimizations 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 optimizations are based, is that parallel I/O performance is optimized 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 explain the reasons for these non-intuitive results. In particular, we show 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. This is problematic because it redistributes application data in a way that conforms poorly with 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 across two large-scale

DOI: 10.1002/cpe

Lustre installations, showing that this alternative approach to collective I/O operations does, in fact, provide significantly enhanced parallel I/O performance. However, we also show that the magnitude of such performance improvement depends on several factors, including the number of aggregator processes and Object Storage Devices, and the power of the system's communication infrastructure. We also show that the optimal data redistribution pattern employed by Y-Lib is dependent upon these same factors.

This research is performed within the context of ROMIO [10], a high-performance implementation of the MPI-IO standard developed and maintained at the 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 two large-scale Lustre installations: Ranger, located at the University of Texas Advanced Computing Center, and BigRed, which is located at Indiana University. 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 Lustre file systems.

This paper makes two primary contributions. First, it increases our understanding of the interactions between collective I/O optimizations in a very important implementation of the MPI-IO standard, the underlying assumptions upon which these optimizations are based, and the Lustre file system 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 1000% increase in the performance. We believe that this paper will be of interest to a large segment of the high-performance computing and Grid communities given the importance of both MPI-IO and Lustre to large-scale, scientific computing.

The remainder of this paper is organized as follows. In Section 2, we provide the background information about MPI-IO and collective I/O operations. In Section 3, we discuss the Lustre object-based storage architecture. In Section 4, we discuss the architectures of two large-scale Lustre installations, our experimental set-up on both, and our experimental results. In Section 5, we discuss the related work. In Section 6, we consider the generality of our approach to high-performance parallel I/O in Lustre, and we provide our conclusions and future research in Section 7.

2. 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 [11–13]. However, it is often the case that taken together, the processes are performing large, contiguous I/O operations, which historically have made much better use of the parallel I/O hardware.

MPI-IO [4], the I/O component of the MPI2 standard was developed (in part at least) to take advantage of such global information to enhance the parallel I/O performance. One of the most

DOI: 10.1002/cpe

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 [10], which is integrated into the MPICH2 MPI library developed and maintained at the Argonne National Laboratory. ROMIO provides key optimizations for the 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 [14] upon which ROMIO implements the MPI-IO interface. ADIO implements the file system dependent features, and is thus implemented separately for each file system.

ROMIO implements the collective I/O operations using a technique termed *two-phase I/O* [9,10]. 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. A simple example may help clarify these ideas.

Consider an application with four processes, each of which must write two megabytes to a file. Figure 1(a) shows how the file data is partitioned among the four processes. In this example, each

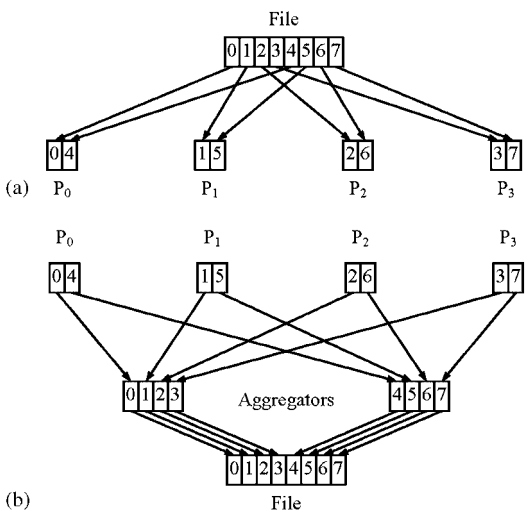


Figure 1. (a) Two-phase I/O example—initial distribution and (b) two-phase I/O example—redistribution using two aggregator processes.

DOI: 10.1002/cpe

process operates on two non-contiguous file regions. Because the data is non-contiguous in the file, writing it to disk requires that each process perform two separate I/O operations with a seek operation between them.

Figure 1(b) depicts this same data being written to disk using two-phase I/O. This example assumes two aggregator processes, one of which will write data blocks 0–3 and the other will write blocks 4–7. In the first phase, each process sends its data blocks to the appropriate aggregator process. In the second phase, the aggregator processes each perform one larger, contiguous write operation.

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

3. 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 [15]. 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 [16], we have never observed such improvement by doing so.

Previous research efforts with parallel I/O on the Lustre file system have shed some light on the 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 2 helps to illustrate the problem that arises when I/O accesses cross stripe boundaries.

Assume that 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.

An ADIO driver for Lustre has recently been added to ROMIO, appearing in the 1.0.7 release of MPICH2 [19]. This new Lustre driver adds support via hints for user settable features such as

DOI: 10.1002/cpe

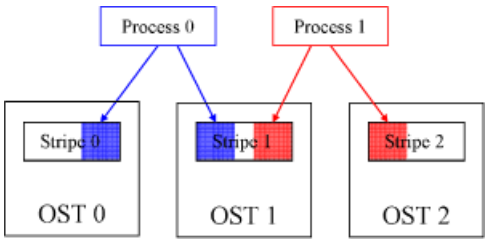


Figure 2. Crossing stripe boundaries with Lustre.

Lustre striping and direct I/O. In addition, the driver insures that disk accesses are aligned on Lustre stripe boundaries.

3.1. 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 Megabyte file, a stripe size of 1 Megabyte, 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 8 Megabyte 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.

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 the 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.

3.2. Aligning data with the Lustre object storage model

The aggregation pattern shown in Figure 3 is what we term an *all-to-all* OST pattern because it involves all aggregator processes communicating with all OSTs. The simplest approach to reducing contention caused by such aggregation patterns is to limit the number of OSTs across which a file is striped. In fact, the generally recommended (and often the default) stripe width is four. While this certainly reduces contention, it also severely limits the parallelism of file accesses, which, in turn,

DOI: 10.1002/cpe

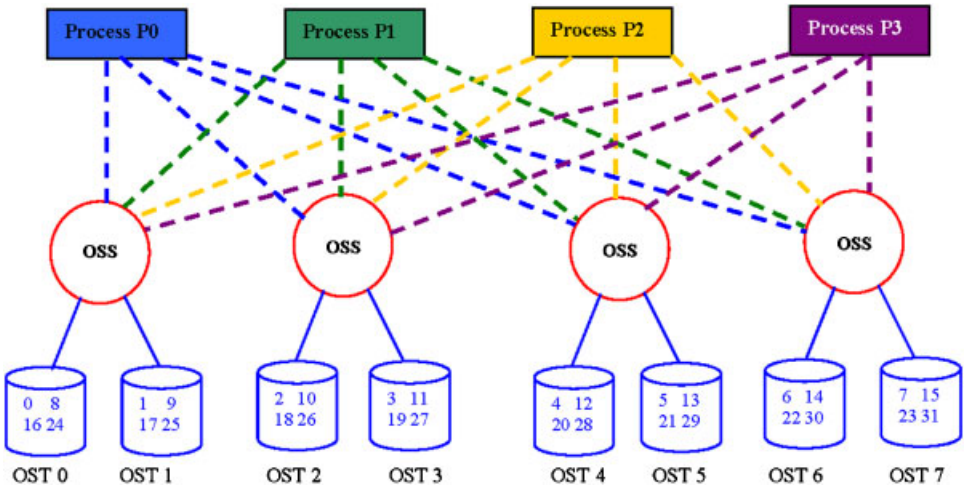


Figure 3. Communication pattern for two-phase I/O with Lustre.

limits parallel I/O performance. However, we believe that 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 contention for file system resources by controlling the number of OSTs with which each aggregator process communicates. On Ranger, we found the optimal data redistribution pattern to be what we term a *'one-to-one'* OST pattern, where the data is arranged such that each aggregator process communicates with exactly one OST. On BigRed, however, we found that a 'one-to-two' OST pattern, where each aggregator process communicates with two OSTs, provided the best performance. 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.

A simple example should help to clarify these ideas. Assume that there are four application processes that share a 16 Megabyte file with a stripe size of 1 Megabyte 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 5(a) shows how the data would be distributed to the aggregator processes in what 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, and it is based on the assumption that performing large, contiguous I/O operations provides optimal parallel I/O performance.

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(c) and (d) shows the data redistribution patterns for the conforming distribution (c) and the one-to-two OST pattern (d).

DOI: 10.1002/cpe

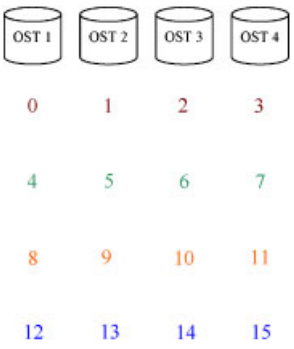


Figure 4. Lustre file layout.

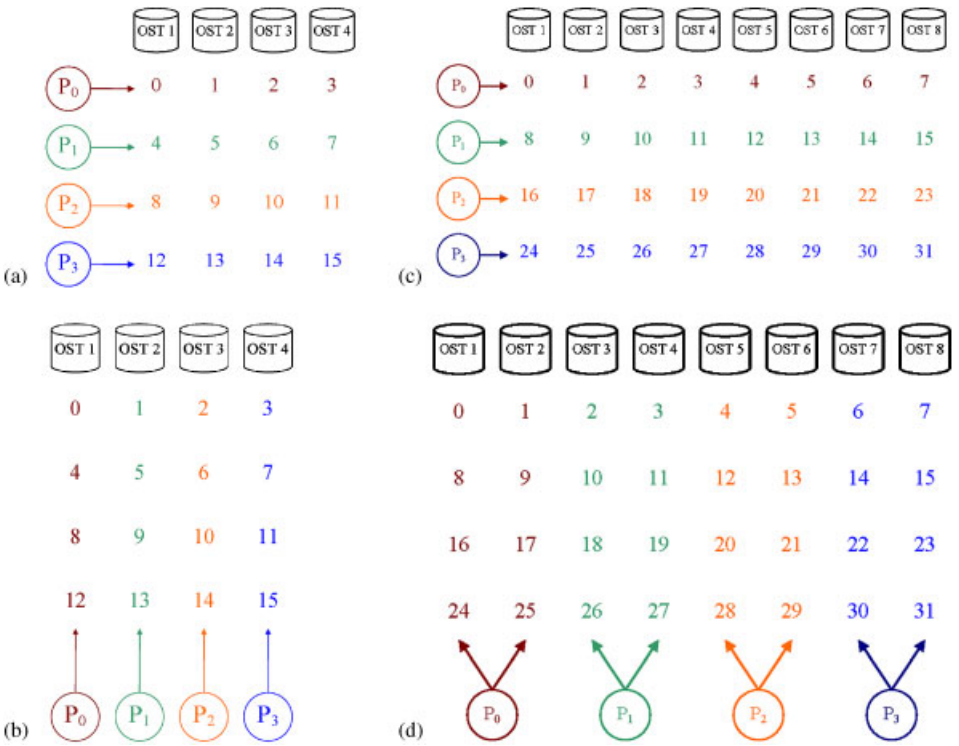


Figure 5. (a) Each process has its data in the conforming distribution; (b) the one-to-one OST pattern; (c) each process has its data in the conforming distribution; and (d) the one-to-two OST pattern.

DOI: 10.1002/cpe

3.3. Tradeoffs in the redistribution patterns

It is interesting to consider the tradeoffs 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.

4. 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 [20]: 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 Gigabyte of memory. The compute nodes were connected to Lustre through 24 Myricom 10-Gigabit Ethernet cards. The Lustre file system (Data Capacitor) was mounted on Big Red, and consisted of 52 Dell servers running Red Hat Enterprise Linux, 12 DataDirect Networks S29550, and 30 DataDirect Networks 48 bay SATA disk chassis, for a capacity of 535 Terabytes. There were a total of 96 OSTs on the Data Capacitor, and there was a total aggregate transfer rate of 14.5 Gigabits per second. The MPI implementation used on BigRed was MPICH2.

The other Lustre installation was Ranger, located at the Texas Advanced Computing Center (TACC) at the University of Texas. There are 3936 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, for a total of 300 OSTs. The bottleneck in the system was a 1-Gigabyte per second throughput from the OSSs to the network.

Both Ranger and the Data Capacitor are production file systems that are heavily utilized by the scientific research community, and we were unable to obtain exclusive access to either file system for our testing. Thus, we were unable to control the number of other jobs, the I/O characteristics of such jobs, and the level of network contention during our experimentation. The primary problem with not having exclusive access is the potential for large variability in experimental results making them very difficult to interpret. However, as will be seen below, the level of variability in these results is not large, and we thus believe that the results obtained here are reflective of what a user would experience when accessing these file systems.

DOI: 10.1002/cpe

It is important to note that the experimental environment is quite different on these two systems. BigRed has a smaller number of nodes (768 versus 3936), and a significantly longer maximum runtime (2 days to 2 weeks on BigRed versus 24 h on Ranger). This resulted in very lengthy queues, where the number of waiting jobs often exceeded 1000 and was rarely less than 700. Thus it was difficult to obtain a large number of nodes, and the time between the experiments could be quite large, often taking between 4 days and 1 week.

For these reasons, we were able to complete a larger set of experiments, with a larger number of processes and OSTs, on Ranger than we were on BigRed. We thus begin by discussing our results on Ranger.

4.1. Experimental study on ranger

We varied three key parameters in the experiments conducted on Ranger: 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 1 Megabyte. Each data point represents the mean value of 50 trials taken over a five-day period.

4.1.1. 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.

4.1.2. Data aggregation patterns without redistribution

The next set of experiments assumed that 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. Thus there was no need to perform data redistribution, but we disabled two-phase I/O nonetheless to ensure fairness. In the case of Y-Lib, the data redistribution phase was not executed, and each process performed the independent write operations assuming that the data was already in the one-to-one pattern.

4.1.3. 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

DOI: 10.1002/cpe

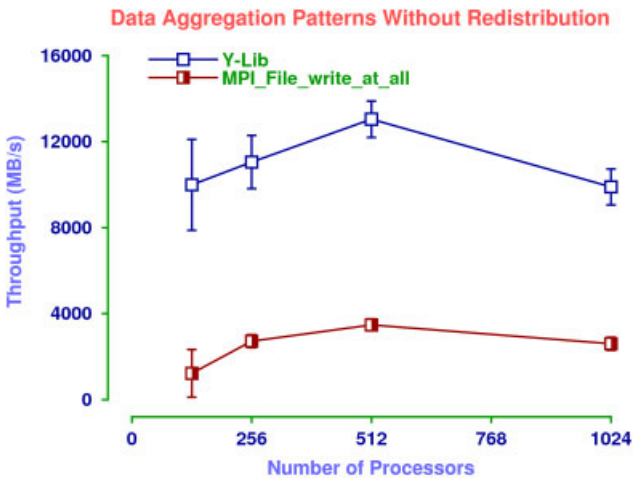


Figure 6. This figure shows the mean throughput obtained by each approach as a function of the number of processors when data redistribution is not required.

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.

4.2. Experimental results

The experimental results are shown in Figures 6–8. Each data point represents the measured throughput averaged over 50 trials and 95% confidence intervals around the means. 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 with 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 the I/O performance by up to a factor of 10. This is particularly impressive given that each process performed 1024 independent write operations.

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 the performance by up to a factor of three.

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.

DOI: 10.1002/cpe

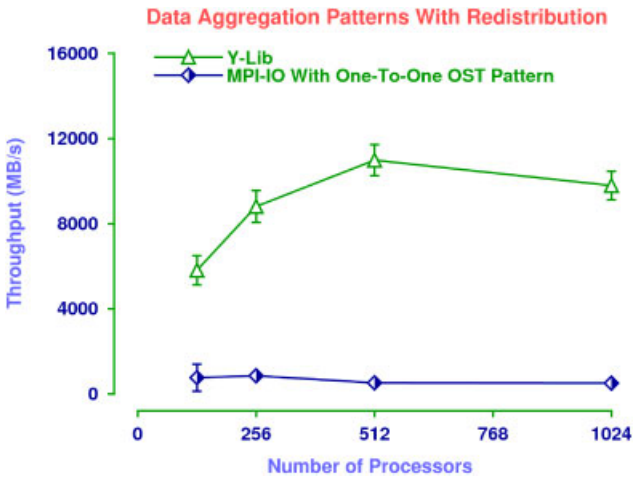


Figure 7. This figure shows the mean throughput obtained by each approach as a function of the number of processors when data redistribution is required.

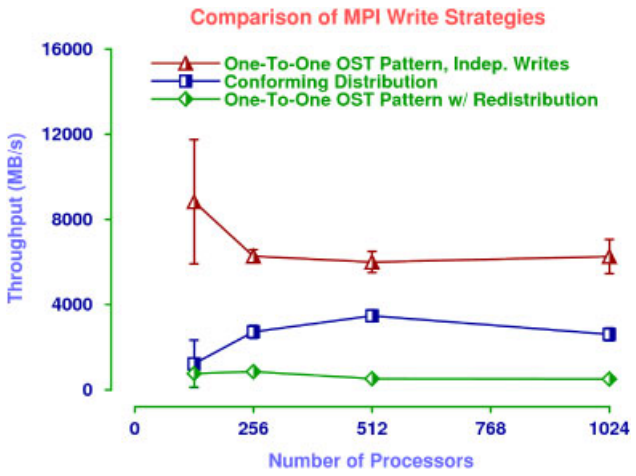


Figure 8. The mean throughput obtained by MPI-IO using three different OST patterns as a function of the number of processes. The performance of MPI-IO itself can be greatly improved by using the one-to-one OST pattern.

4.3. Discussion

These results strongly support the hypothesis that MPI-IO does, in fact, perform very poorly in a Lustre file system when the all-to-all communication pattern creates significant contention for

DOI: 10.1002/cpe

system resources. They also show that it is possible to utilize all of the system resources quite profitably when utilizing the data redistribution pattern employed by Y-lib. These results also lend strong support to other studies on Lustre showing that the maximum performance is obtained when individual processes write to independent files concurrently [7,16]. It also helps to 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 the performance [21].

4.4. Experimental studies on BigRed

In our initial exploration of Y-lib on BigRed, we did not obtain the improvement in the I/O performance that we observed on Ranger. Further investigation revealed that we were under-utilizing the powerful parallel I/O subsystem by having each process communicate with only one OST. We then experimented with other OST patterns, and found that the best performance was obtained when each process communicated with exactly two OSTs (a *one-to-two OST* pattern). Thus all of the experiments discussed in this section utilized this data redistribution pattern. It should also be noted that we had not yet completed the implementation of the one-to-two OST redistribution pattern at the time of this publication, and thus the experiments discussed here assumed that the data was already in the correct distribution for each approach.

4.4.1. Data aggregation patterns without redistribution

In these experiments, we compared the I/O performance obtained when the data was arranged according to the conforming distribution or the two-OST distribution. We varied the number of aggregator processes between 32 and 256, and the stripe width between 32 and 96 (the maximum number of OSTs available). We scaled the file size between 32 and 256 Gigabytes (i.e. one Gigabyte times the number of processes), and, for 32 to 96 processes, set the stripe width equal to the number of processes. In the case of 192 processes, we utilized 96 OSTs. In the 256-process case, however, we utilized only 64 OSTs. This was because the number of processes must be a multiple of the number of OSTs to ensure that each process always communicates with the same two OSTs. In all cases, the stripe size was one Megabyte, and the writes were aligned on stripe and lock boundaries.

As noted above, these experiments assumed that the data was in the correct distribution for each approach, and thus neither performed the first phase of the two-phase I/O algorithm. When the data was distributed according to the conforming distribution, the aggregators performed a single large, contiguous I/O operation. Because of the one-to-two OST pattern employed by Y-lib, each process made 512 separate I/O requests.

4.4.2. Experimental results

The results of these experiments are shown in Figure 9. It shows the mean throughput and 95% confidence intervals around the means, as a function of the number of processes and I/O strategy. As can be seen, the Y-lib distribution pattern starts to significantly outperform the conforming distribution once the number of processes exceeds 32. The largest improvement comes with 96 processes (and OSTs), where a 36% improvement in performance is observed. The relative improvement in

DOI: 10.1002/cpe

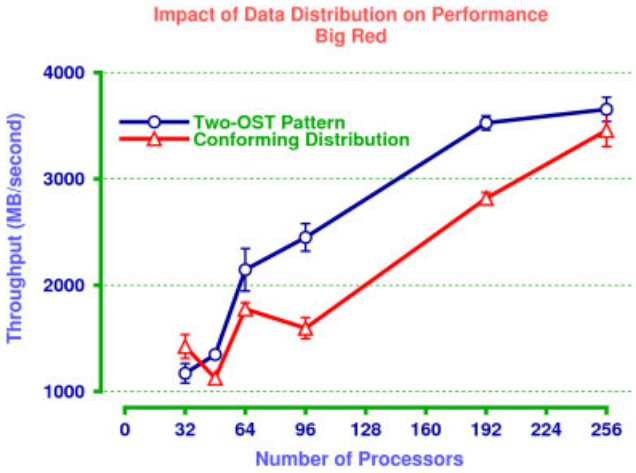


Figure 9. The comparison of mean I/O performance as a function of the redistribution pattern and the number of processors.

performance was approximately 32% with 192 processes (96 OSTs), and was on the order of 5% in the 256-process case (64 OSTs).

4.5. Discussion

These results are very different from those obtained on Ranger, and it is interesting to consider the causes for such differences. There are really two separate questions: Why did the performance of ROMIO increase with increasing numbers of processes, and why did the rate of performance increases begin to slow for Y-lib in this scenario? We address each question in turn.

We believe the increasing performance observed on BigRed was due to the very powerful parallel I/O subsystem available from the Data Capacitor, combined with an aggregate bandwidth of 240 Gigabits per second between the two systems provided by the 24 10-Gigabit Myricom connections. Clearly, this infrastructure was powerful enough to handle the all-to-all communication pattern required by the conforming distribution. However, the number of processes we were able to test was relatively small (at least compared to Ranger), and it would be very interesting to execute the same tests with 512 processes.

The reduction in the rate of increasing performance observed in Y-lib was, we believe, related to the ratio of OSTs to aggregator processes. That is, the overhead of performing a large number of small I/O operations becomes more pronounced as contention for OSTs and network services begins to increase. In the case of 256 aggregators and 64 OSTs, each OST is communicating with eight processes (even though each process is only communicating with two OSTs). Thus, while the level of contention in this case is not as significant as that resulting from the conforming distribution, it is apparently enough to begin to impact the performance of Y-lib.

DOI: 10.1002/cpe

5. RELATED WORK

The most closely related work is from Yu *et al.* [7], 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 [6] provide an excellent analysis of Lustre's performance on the Cray XT3/XT4, and, based on such analysis, provide some guidelines to maximize the 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 I/O 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 [17], it was shown that aligning the data to be written with the basic striping pattern improves the 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.

6. GENERALITY OF APPROACH

While Y-Lib outperformed ROMIO in the two file systems studied here, we do not believe it will necessarily do so in all Lustre file system configurations. In fact, our own experimentation showed relatively little increase in the performance when executing on 256 processors on BigRed, and, judging by Figure 9, the relative performance of the two approaches may cross farther out in the curve (i.e. 512–1024 processors).

In order to extend Y-Lib to the general case, we are developing two mechanisms that can be used to guide the decision of the best OST pattern for Y-Lib to use, or whether Y-Lib should be used at all. The first approach is to develop an analytic model to provide such information, and the second approach is to develop a set of heuristics that can provide similar information. We discuss each in turn.

DOI: 10.1002/cpe

We are developing the analytical model as a function that takes a set of parameters and returns the most appropriate I/O strategy. This research has identified some of the most important parameters to such a model including the bandwidth between the clients and the OSSs, the bandwidth between the OSSs and the network, and the capabilities of the storage devices. Another important factor is the time required to redistribute the data between processes, which would be a function of the interconnection network. There are likely to be other contributing factors of which we are currently unaware.

The second approach is to develop a tool that can empirically determine the optimal I/O strategy when moving to a new Lustre file system, or when important changes are made to the current system. This tool would execute a set of I/O benchmarks involving different redistribution patterns, stripe sizes, stripe widths, numbers of aggregator processes, and numbers of OSTs. The results of the benchmarks could then be compared to determine the most appropriate I/O strategy.

Another important question is how Y-Lib would scale to a Grid environment where the application processes themselves could be geographically distributed. To be successful in such an environment requires the ability to dynamically evaluate the optimal I/O strategy based on the currently allocated resources. While it would be infeasible to run the suite of benchmark tests dynamically, it would certainly be feasible to dynamically re-evaluate the analytic model assuming that the run-time system were able to provide updated values for model parameters. For example, if the costs of redistributing data were too expensive (because of the high latency between processors) the model would suggest not using Y-Lib at all.

7. 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 way the data was redistributed in the first phase of a two-phase I/O operation, resulting in an all-to-all communication pattern that can cause (perhaps significant) contention for system resources. We then implemented a new (and non-intuitive) data redistribution pattern 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 ROMIO by up to a factor of 10.

To extend these results to the general Lustre environment, we have proposed the development of an analytic model and a set of heuristics to help guide the choice of the most appropriate I/O strategy. The other primary focus of the current research is the integration of Y-lib into ROMIO.

ACKNOWLEDGEMENTS

This material is based on the work supported by the National Science Foundation under Grant No. 0702748. Portions of this paper are reprinted with permission from 'Y-Lib: A User Level Library to Increase the Performance of MPI-IO in a Lustre File System Environment', by Phillip M. Dickens and Jeremy Logan, published in the International ACM Symposium on High Performance Distributed Computing.

DOI: 10.1002/cpe

REFERENCES

1. Cluster File Systems, Inc. Available at: http://www.clusterfs.com [4 October 2007].
2. Schmuck F, Haskin R. GPFS: A shared-disk file system for large computing clusters. *The Proceedings of the Conference on File and Storage Technologies*, IBM Almaden Research Center, San Jose, CA, 2002.
3. The Panasas Home Page. Available at: http://www.panasas.com [15 September 2007].
4. MPI-2: Extensions to the Message-Passing Interface. Message Passing Interface Forum. Available at: http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html [14 April 2009].
5. I/O Performance Project. Available at: http://wiki.lustre.org/index.php?title=IOPerformanceProject [16 October 2007].
6. Larkin J, Fahey M. Guidelines for Efficient Parallel I/O on the Cray XT3/XT4. *CUG 2007*, Seattle, WA, U.S.A., 2007.
7. Yu W, Vetter J, Canon RS, Jiang S. Exploiting Lustre file joining for effective collective I/O. *The Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid* (*CCGrid '07*), Rio de Janeiro, Brazil, 2007.
8. Thakur R, Gropp W, Lusk E. Data sieving and collective I/O in ROMIO. *The Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, Annapolis, MD, U.S.A., 1999; 182–189.
9. Thakur R, Gropp W, Lusk E. On Implementing MPI-IO portably and with high performance. *The Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, Atlanta, GA, U.S.A., 1999; 23–32.
10. Thakur R, Ross R, 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.
11. Ching A, Choudhary A, Coloma K, Liao W-K, Ross R, Gropp W. Noncontiguous I/O accesses through MPI-IO. *The Proceedings of the Third International Symposium on Cluster Computing and the Grid* (*CCGrid*), Berlin, Germany, 2002; 104–111.
12. Isaila F, Tichy WF. View I/O: Improving the performance of non-contiguous I/O. *The Proceedings of the IEEE Cluster Computing Conference*, Hong Kong, 2003.
13. Thakur R, Gropp W, Lusk E. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing* 2002; **28**(1):83–105.
14. Thakur R, Gropp W, Lusk E. An abstract-device interface for implementing portable parallel-I/O interfaces. *The Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, Annapolis, MD, U.S.A., 1996.
15. Bramm PJ. The Lustre Storage Architecture. Available at: http://www.lustre.org [16 October 2007].
16. Lustre: Scalable, secure, robust, highly-available cluster file system. An offshoot of AFS, CODA, and Ext2. Available at: www.lustre.org/ [16 October 2007].
17. Liao W-K, Ching A, Coloma K, Choudhary A, Ward L. An implementation and evaluation of client-side file caching for MPI-IO. *The Proceedings of the International Parallel and Distributed Processing Symposium* (*IPDPS '07*), Long Beach, CA, U.S.A., 2007.
18. Liao W-K, Ching A, Coloma K, Choudhary A, Kandemir M. Improving MPI independent write performance using a two-stage write-behind buffering method. *The Proceedings of the Next Generation Software* (*NGS*) *Workshop*, Long Beach, CA, U.S.A., 2007.
19. MPICH2 Home Page. Available at: http://www.mcs.anl.gov/mpi/mpich [14 April 2009].
20. The Teragrid Project. Available at: http://www.teragrid.org [20 April 2009].
21. Frequently Asked Questions. Available at: http://www.lustre.org [16 October 2007].

DOI: 10.1002/cpe