Exploiting Lustre File Joining for Effective Collective IO

Weikuan Yu, Jeffrey Vetter
Oak Ridge National Laboratory
Computer Science & Mathematics
Oak Ridge, TN, USA 37831
{wyu,vetter}@ornl.gov

R. Shane Canon
Oak Ridge National Laboratory
National Center for Computational Sci.
Oak Ridge, TN, USA 37831
canonrs@ornl.gov

Song Jiang
Wayne State University
Electrical & Computer Engineering
Detroit, MI, USA 48202
sjiang@eng.wayne.edu

Abstract

Lustre is a parallel file system that presents high aggregated IO bandwidth by striping file extents across many storage devices. However, our experiments indicate excessively wide striping can cause performance degradation. Lustre supports an innovative file joining feature that joins files in place. To mitigate striping overhead and benefit collective IO, we propose two techniques: split writing and hierarchical striping. In split writing, a file is created as separate *subfiles*, each of which is striped to only a few storage devices. They are joined as a single file at the file close time. Hierarchical striping builds on top of split writing and orchestrates the span of subfiles in a hierarchical manner to avoid overlapping and achieve the appropriate coverage of storage devices. Together, these techniques can avoid the overhead associated with large stripe width, while still being able to combine bandwidth available from many storage devices. We have prototyped these techniques in the ROMIO implementation of MPI-IO. Experimental results indicate that split writing and hierarchical striping can significantly improve the performance of Lustre collective IO in terms of both data transfer and management operations. On a Lustre file system configured with 46 object storage targets, our implementation improves collective write performance of a 16-process job by as much as 220%.

1 Introduction

Many of the scientific applications running on contemporary high end computing platforms are very data-intensive, such as those in climate modeling, fusion, fluid dynamics, and biology. For example, the Gyrokinetic Toroidal Code (GTC [12]) -- an application for fusion -- can require a throughput of several 10s of gigabytes per second in order to minimize the portion of time spent in IO and to achieve good scalability on systems with tens or hundreds of TeraFlops (10^{15}) per second.

Figure 1 shows a diagram of software layers in typical ultra-scale platform that supports data-intensive applications. Collectively, these layers provide portable abstractions for IO accesses. At the top end, scientific applications perform IO through middleware libraries such as Parallel netCDF [9], HDF [23, 24] and MPI-IO [22], often cooperatively among their processes. Parallel file systems, towards the bottom of the stack, directly serve IO requests by striping file extents and/or IO blocks across multiple storage devices. Obtaining good collective-IO performance across many processes on top of these software layers is a complex task. It requires not only awareness of the processes' collective data access patterns, but also thorough understanding of the entire software stack and, in particular, the behavior of underlying file systems.

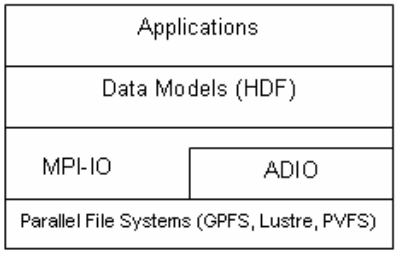

Figure 1 IO Software Stacks for HPC Applications

As shown in Figure 1, the mid-level libraries, represented by MPI-IO, are directly implemented on top of file systems. ADIO [19] is the abstract IO interface of MPI-IO, which can be specialized for specific file system implementations. Together, these programming stacks offer crucial avenues for efficient storage accesses. Numerous techniques have been investigated and implemented to improve the scalability of MPI-IO data operations, such as two-phase IO [20], data sieving [21], and data shipping [13]. Some of these techniques are designed for generic file systems, and, as such, are unable to avoid specific limitations of a particular file system. Yet other techniques exploit specific features of individual file systems, such as list IO for PVFS2 [2, 5] and data shipping for GPFS [13, 16].

Seventh IEEE International Symposium on Cluster Computing and the Grid(CCGrid'07)
0-7695-2833-3/07 $20.00 © 2007 IEEE

Lustre [8] is a file system that has been deployed on many supercomputers, such as Jaguar at Oak Ridge National Laboratory, Thunderbird at Sandia National Laboratory, Tera-10 at CEA in Europe, and TSUBAME at Tokyo Tech in Japan [3]. Because many of the applications running on these systems use (or could use) collective IO operations, developing scalable collective-IO operations over Lustre will benefit a large number of applications. Not surprisingly, there has been an effort to provide a Lustre-specific ADIO implementation, in which file hints are introduced to specify striping pattern on a per-file basis. However, the research on how to leverage Lustre-specific file system features to optimize MPI-IO design and implementation is still at a nascent stage.

In this paper, we show that excessively wide striping in Lustre can cause performance degradation. An important question, then, is how to continue aggregating IO bandwidth available from many storage devices while avoiding the performance hit of wide striping.

Consequently, Lustre provides a file joining feature for joining multiple files into a single combined file. Using this feature, a file with mixed striping patterns can be created by joining multiple files with different striping parameters. We believe that leveraging this feature could help mitigate the detrimental drawbacks of wide striping while still retain the benefits from aggregating bandwidth of many storage devices.

In this paper, we investigate the feasibility of using Lustre file joining to benefit of collective IO. To this end, we propose and evaluate two techniques: split writing and hierarchical striping. In split writing, different processes are allowed to open/create individual files, each with a small stripe width for data input/output. These files are also referred to as *subfiles*. In addition, only one IO process opens the first subfile for initial IO accesses, so bursts of metadata traffic by a parallel open are avoided. Hierarchical striping determines striping parameters for the subfiles so that: (1) each of them is aggregating bandwidth from an appropriate set of IO storage devices; (2) there is no overlapping of storage devices between subfiles; (3) and a good coverage of storage devices is achieved. We have designed and prototyped a Lustre-specific collective IO approach based on split writing and hierarchical striping. In our experiments, when using the proposed techniques, Lustre collective IO performance is significantly improved for both data transfer and management operations. Split writing is able to provide highly scalable collective management operations such as MPI_File_Open() and MPI_File_Set_size(); hierarchical striping can dramatically improve collective IO bandwidth by aggregating IO bandwidth available to multiple files. We also show that these optimizations in collective IO benefit scientific applications' common IO patterns using MPI-Tile-IO [14] and a modified NAS BT/IO program [25].

The rest of the paper is organized as follows. In the next section, we provide our motivation. Section 3 provides the detailed design of split writing and hierarchical striping, and discusses how they accomplish the intended goals. We evaluate our proposed techniques in Section 4. Finally, we give an overview of related work in Section 5 before concluding the paper in Section 6.

2 Motivation

In this section, we motivate the work of exploring Lustre file joining functionality for collective IO optimizations. An overview of Lustre is provided first. Then, we show the performance trends of Lustre with the varying stripe widths.

2.1 An overview of Lustre

Lustre [8] is a POSIX-compliant, object-based parallel file system. It provides fine-grained parallel file services with its distributed lock management. Lustre separates essential file system activities into three components: clients, metadata servers, and storage servers. These three components are referred to as Object Storage Client (OSC), Meta-Data Server (MDS) and Object Storage Targets (OST), respectively. Figure 2 shows a diagram of the Lustre system architecture. An OSC opens and creates a file through an MDS (step 1), which creates objects in all OSTs (step 2). IO is then performed in a striped manner to all OSTs (step 3). By decoupling metadata operations from IO operations, data access in Lustre is carried out in a parallel fashion, directly between the OSCs and OSTs. This allows Lustre to aggregate bandwidth available from many OSTs. Lustre provides other features such as read-ahead and write-back caching for performance improvements. Here, we discuss a few relevant features: file consistency, file striping, and file joining.

File Consistency and Locking To guarantee file consistency, Lustre serializes data accesses to a file or file extents using a distributed lock management mechanism. Because of the need for maintaining file consistency, all processes first have to acquire locks

before they can update a shared file or an overlapped file block. Thus, when all processes are accessing the same file, their IO data performance is dependent not only on the aggregated physical bandwidth from the storage devices, but on the amount of lock contention that exists among them.

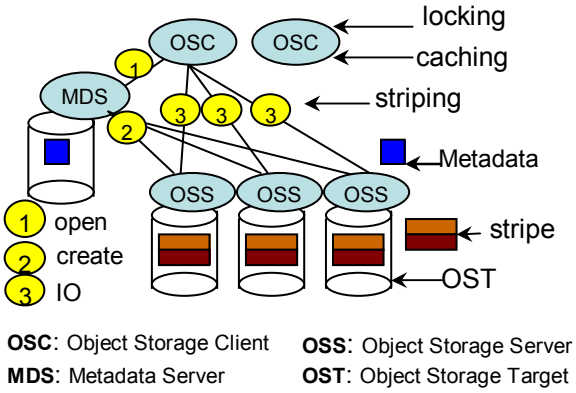

Figure 2 Lustre System Architecture

Flexible Striping and Joining As shown in Figure 2, file IO is striped across a number of OSTs. Striping pattern parameters can be specified on a per-file or per-directory basis. Such parameters include stripe size, stripe width, and stripe index (the index of the first storage device). In addition, Lustre also supports a file joining feature. It allows files with different striping patterns to be joined into a single file. This feature allows a file to use different striping patterns for its extents.

Table 1 Commands for Stripe Width Analysis

Program	Detail Command
dd write	time dd if=/dev/zero if=/Lustre/file count=4M
dd read	time dd of=/dev/null if=/Lustre/file count=4M
IOzone write	iozone –e –i 0 –j 8 –L 128 –S 2048 –s 4194304 –r 64K –f /Lustre/file
IOzone read	iozone –e –i 1 –j 8 –L 128 –S 2048 –s 4194304 –r 64K –f /Lustre/file

2.2 Impacts of Lustre Striping Width

We conducted several experiments to evaluate the Lustre performance with various striping patterns. The first two experiments measured Lustre read/write performance using the UNIX dd command and the IOzone benchmark [1]. These experiments were conducted with a Lustre file system composed of one MDS and 64 OSTs, as described in Section 4, using Lustre version 1.4.7. Table 1 lists the commands used for these experiments.

Figure 3 shows the performance of Lustre read/write as measured from the UNIX dd command and the IOzone benchmark. The file stripe width ranges from 1 to 64 OSTs. These IO performance results suggest that wider stripe width does not help IO accesses with small request block sizes (512B, dd), and the performance trend is improved with larger request sizes (64KB, IOzone), up to only 4 or 8 OSTs for reads and writes, respectively. Beyond that, even wider stripe width gradually brings the IO bandwidth down because of the overhead of striping data to more OSTs.

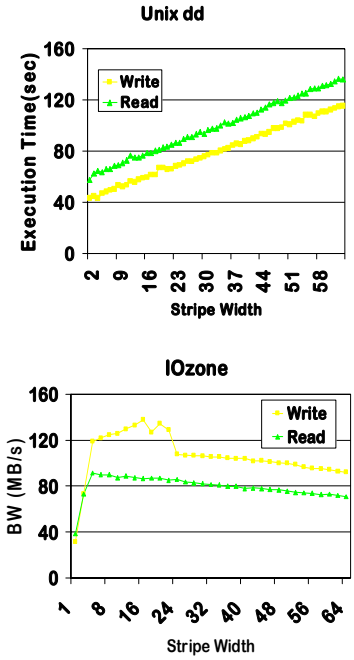

Figure 3 Impact of Stripe Width to dd and IOzone Performance

Note that these experiments were done with rather small IO requests from a single client to reveal the striping cost. It is not to be interpreted as direct contrast to the earlier report [15], in which Lustre achieves good performance with many processes making large IO requests. Based on the Lustre architecture shown in Figure 3, we speculate the performance degradation of wide striping may come from at least two aspects. One is the increased protocol processing overhead when an OSC is communicating with more OSTs; the other is the reduced memory cache locality when the client's communication buffer space is multiplexed for more OSTs.

3 Collective IO with File Joining

As discussed in Section 2.2, wide striping leads to performance degradation beyond certain stripe width

for small IO requests. We attempt to investigate how Lustre file joining can help mitigate this problem while still achieving wide striping when necessary. In this paper, we focus on the feasibility of using file joining by examining a higher-level parallel IO library, MPI-IO [22]. We propose two techniques: split writing and hierarchical striping to address this issue. Both techniques are built on top of file joining. Split writing allows processes to create/open separate subfiles for IO. Hierarchical striping computes the striping parameters for the subfiles to ensure high aggregated IO bandwidth from a sufficient number of OSTs.

3.1 Split Writing

Writing file contents to multiple subfiles is not a new idea. However, one needs to be aware of Lustre characteristics to take advantage of this idea and the strength of Lustre file joining. For example, we need to answer a variety of questions, such as when to create all the subfiles, when to join them together, which process is to manage what subfiles, as well as how to maintain MPI-IO semantics for file consistency.

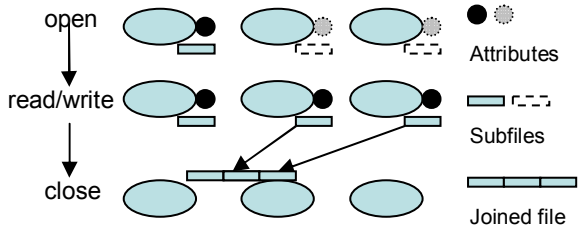

Figure 4 Diagram of Split Writing

Figure 4 shows a diagram of split writing among three MPI processes. At the beginning of the file creation, only one process, rank 0, creates an actual subfile. All other processes open their portion of file extents as a *ghost subfile*. Ghost subfiles (marked with dashed boxes) are actually created only when applications write to them. This optimization avoids a burst of metadata operations for the same file to the metadata server, a problem known as *parallel open* to large scale applications. By delaying the creation of other subfiles, metadata operations are amortized across the course of the entire file IO. At the end of parallel open, all processes exchange and synchronize on the file attributes, including those of the ghost subfiles. File attributes are also exchanged and synchronized in every MPI_File_sync() operation to ensure consistency during run-time. Note that this does not violate MPI-IO consistency semantics as MPI-IO specification requires all IO processes to call sync operations if global data consistency is desired.

When a file is to be closed, all processes exchange information such as the number of its subfiles, their real names, and their representative file domains. Remaining ghost files are left as holes in the final file. No alteration is introduced on the Lustre internal storage management of file holes. These subfiles are then joined into a single file. Instead of joining files in a tree fashion through many processes, the subfiles are joined by Rank 0. Interestingly, this does not increase the processing overhead because all metadata operations are essentially serialized by the Lustre MDS as a single thread per request. More clients will only add to its processing burden.

3.2 Hierarchical Striping

Lustre stripes file objects across its object storage targets (OSTs). The striping pattern is determined by a combination of three parameters: stripe size, stripe width (or stripe count), and stripe index.

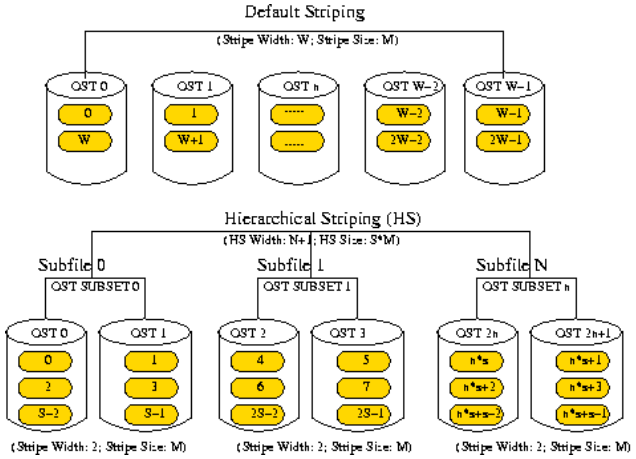

Figure 5 Comparison of Default Striping and Hierarchical Striping

As discussed earlier, split writing creates multiple subfiles. To achieve the best IO rate for the collection of subfiles, one needs to ensure the following for the subfiles: (1) a small subfile, (i.e., a file smaller than 1 MB, does not need to be striped); (2) each subfile should not stripe too wide (Our experiments (c.f. Figure 2) used a stripe width of 2); (3) the subfiles should not have any overlap; and (4) the subfiles together should cover a sufficient number of OSTs for a good aggregated bandwidth. To achieve this, we apply a technique called hierarchical striping to determine the striping parameters for a subfile, including (a) *subfile_size*: the size of a subfile; (b) *subset_size*: the number of OSTs for this subfile; and (c) *subset_index*: the index of the OST subset. Figure 5

shows a comparison between the default striping and the hierarchical striping. Compared to the default striping pattern of *(stripe_size: M, stripe_width: W, stripe_index: 0)*, the provided example has a hierarchical striping pattern of *(subfile_size: S*M, subset_size: 2, subset_index: 0)*, with each subfile's striping parameters as (M, 2, 2*i). The index, i, is the rank of a subfile. Note that, by default, Lustre has a load balancing mechanism to select an OST for the *stripe_index*. Hierarchical striping maintains the same feature to use this index as the first OST of the first subset.

Hierarchical striping also reshapes IO access pattern to the OSTs. A file created to stripe across all OSTs forces the process to communicate with all the OSTs. Hierarchical striping reduces the number of connections between Lustre clients and OSTs. Communication memory buffers are more frequently recycled among a few connections, resulting in better memory locality. Hence, hierarchical striping could both reduce striping overhead and enhance the communication scalability [26].

4 Performance Evaluation

We have implemented a prototype of the proposed techniques in the ROMIO of MPICH2-1.0.3 release. In this section, we describe its performance evaluation. Our experiments are conducted on a cluster of 80 Ciara VXB-7520J blades: each with dual Intel Xeon 3.4 GHz processors, 2MB cache, 800MHz Front Side Bus, 4GB physical memory, and a Gigabit Ethernet card connected to PCI-X 100 Mhz bus. On the nodes configured as Lustre OSTs, a 7200 RPM, ATA/100 Western Digital hard disk WD800JB is used for disk storage. These nodes are running CentOS 4.4 Linux operating system and Lustre version 1.4.7. Out of the eighty-node cluster, a Lustre file system is configured with 46 OSTs and one MDS, unless specified otherwise.

4.1 Collective Management Operations

Management operations, such as MPI_File_open() and MPI_File_set_size(), do not involve massive data transfer, but they do require support for scalable metadata operations from the underlying file system [8]. To evaluate the benefits of our optimizations to these management operations, we have performed the following experiments using a microbenchmark available in the PVFS2 [2] distribution. The first experiment measures the average time to create a file using collective MPI_File_open(). As shown in Table 2, compared to the original ADIO implementation, our implementation, denoted as *New,* significantly improves the time to create an MPI file. Furthermore, the creation time does not increase as the number of processes increase. This improvement is due to two benefits of split writing. First, split writing is able to reduce the striping width when creating new files, therefore reducing the striping cost. Secondly, although it presents an abstraction of a shared file to all processes, there is only one process that actually creates the file, therefore reducing the amount of metadata requests to the Lustre MDS.

The second experiment measures the average time to perform a resize operation using collective MPI_File_Set_size(). As shown in Table 2, our implementation brings down the cost of resize operations dramatically. This is because split writing allows only one process to update the attributes (such as size) of a file. When the *subfiles* are to be closed, their attributes, such as file size, are committed to the physical storage. Our approach essentially eliminates the contention of metadata operations from many processes. The remaining cost is due to a single system call and the need of synchronization among parallel processes.

Table 2 Comparison of the Scalability of Management Operations

No. of Processes	Original	New
Create (Milliseconds)		
4	8.05	8.75
8	11.98	8.49
16	20.81	8.63
32	37.37	8.98
Resize (Milliseconds)		
4	182.67	0.56
8	355.28	0.81
16	712.68	1.03
32	1432.5	1.36

4.2 Concurrent Read/Write

To measure the concurrent read/write performance, we use a parallel program that iteratively performs concurrent read and write to a shared file. Each process writes and then reads a contiguous 256MB data at disjoint offsets based on its rank in the program. At the end of each iteration, the average time taken for all processes is computed and recorded. Twenty-one iterations are performed, and the lowest and highest values are discarded.

Figure 6 shows the performance of concurrent read and write. Compared to the original, our implementation

improves the aggregated bandwidth by 220% and 95% for writes and reads, respectively. Note that the measured write bandwidth is close to the aggregated peak write bandwidth for all 46 IDE disks. As it reaches the plateau, the available bandwidth for 32 processes drops slightly for writes. Read bandwidth is further improved with the increase of processes. The aggregated read/write bandwidth of the original implementation remains much lower compared to our optimized implementation. In addition, we have also tested the performance of concurrent read and write to an existing join file. It is interesting to note that both implementations report low IO rates. These low rates are due to the manipulation and placement of extent attributes for a joined file. An optimization on the management of a joined file's extent attributes is needed for a proper fix. Nonetheless, this performance degradation is mostly avoided during file IO to a new file because a file is not joined until it is to be closed.

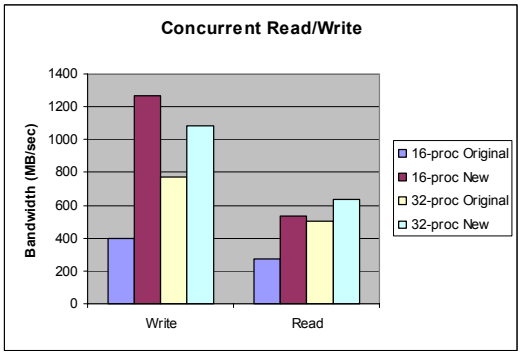

Figure 6 Performance of Concurrent Read/Write

4.3 NAS BT-IO

NAS BT-IO [25] is an IO benchmark that tests the output capability of NAS BT (Block-Tridiagonal) parallel benchmark. It is developed at NASA Ames Research Center. Its data set undergoes diagonal multi-partitioning and is distributed among MPI-processes. The data structures are represented as structured MPI datatypes and written to a file periodically, which is typically every 5 timesteps. There are several different BT-IO implementations, which vary how its file IO is carried out among all the processes. In our experiments, we used an implementation that performs IO using MPI-IO collective IO routines, so called *full mode* BT-IO.

Figure 7 shows the performance of BT-IO. Compared to the original implementation, our implementation actually has longer execution time for BT-IO if we use a default *subset_size* 2 in hierarchical striping. Further analysis reveals that data in BT-IO is written sequentially in 40 different steps. With a subset_size of 2 for a Lustre file system of 46 OSTs, the output file's extents are divided into 23 subfiles, each for two OSTs. In every step, the data falls into only $1/40^{th}$ of the entire file, i.e. one subfile. Therefore, the effective bandwidth is close to what's available from 2 OSTs. We have also tested BT-IO with 8 timesteps. This scenario investigates whether the collective IO pattern in BT-IO, even with a reduced number of repetitions, can benefit from hierarchical striping. The last set in Figure 7 shows the performance of modified BT-IO. Hierarchical striping does indeed reduce the IO time by 5.21 and 3.87 seconds, for class B, 16 and 25 processes, respectively. This suggests that hierarchical striping is beneficial to the scientific IO pattern as exhibited by BT-IO, when the file access is no longer limited to a single subfile. But further investigation is needed to make the benefits generally applicable to output files of arbitrary lengths.

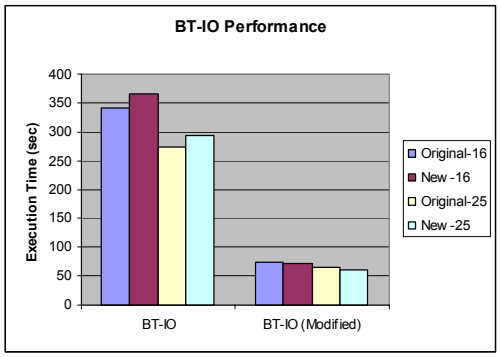

Figure 7 BT-IO Performance

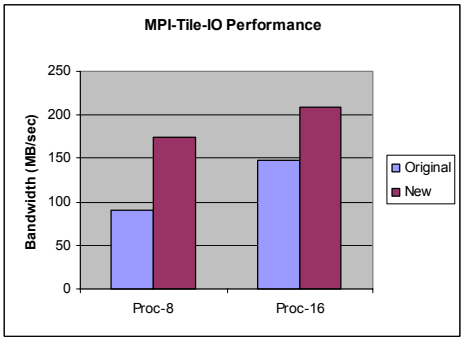

Figure 8 MPI-Tile-IO Performance

4.4 MPI-Tile-IO

MPI-Tile-IO [14] is an MPI-IO benchmark testing the performance of tiled data accesses. In this application, data IO is non-contiguous and issued in a single step

using collective IO. It tests the performance of tiled access to a two-dimensional dense dataset, simulating the type of workload that exists in some visualization applications and numerical applications. In our experiments, each process renders a 1x1 tile with 2048x1536 pixels. The size of each element is 64 bytes, leading to a file size of 192*N MB, where N is the number of processes.

MPI-Tile-IO requires an existing file for read. We focus our experiments on tiled write because writes are the focus of our study. Figure 8 shows the write performance of MPI-Tile-IO with 8 and 16 processes. Our implementation improved MPI-Tile-IO write bandwidth by 91% and 42%, respectively. These results also indicate that our optimizations are able to effectively aggregate write bandwidth for applications with non-contiguous collective IO as exhibited in MPI-Tile-IO [14].

5 Related Work

Numerous researchers have studied techniques to optimize data access for parallel scientific applications. ROMIO [4] provides the most popular implementation of a parallel IO interface, MPI-IO. One of its important features for collective IO is extended two-phase IO [20], which employs a two-phase strategy to distribute IO requests amongst a set of IO aggregator processes, thus consolidating many small, noncontiguous IO requests into a small number of large, contiguous requests for effective collective-IO. Extended two-phase IO [20] optimizes collective IO only for cases with overlapped, non-contiguous accesses. Liao *et .al.* [10, 11] have carried out a series of studies on improving collective IO by caching application data at the user level. It has also been shown as beneficial at the MPI-IO layer [10]. We believe this user-level caching optimization is complementary to our work because it uses write-back to commit the data to the storage, which results in large IO request sizes.

Parallel netCDF (PnetCDF) [9] is a project that provides collective IO optimizations for scientific applications on top of netCDF. Its main purpose is to enable parallelism for netCDF. Its design strategies do not take the parallel file system features into account. Rather it leaves that to lower programming layers, such as MPI-IO.

Tatebe *et al.* have exploited the concepts of local file view in the design of a distributed file system for Grid [17, 18]. The idea of local file view is similar to split writing in this paper, except our technique is an abstraction at the user-level inside MPI-IO, which does not require instrumentation into the implementation of file system client architecture. Our hierarchical striping technique is similar in concept to another technique: two-level striping. Two-level striping is a disk striping technique used in the implementation of the Panasas [7] file system, and is used as an internal storage organization policy. Our hierarchical striping is built on top of the user-level file joining feature. It works at the level of IO middleware, aimed to reduce the overhead of excessive striping. Nonetheless, these two techniques are similar in the way they both provide another level of striping to reshape the communication pattern between storage clients and devices.

MPI-IO/GPFS [13] is an implementation that is similar to our work in that it introduces file-system specific optimizations to ADIO. It provides an optimized MPI-IO implementation on top of IBM General Parallel File System (GPFS) [16]. Collective data access operations in MPI-IO/GPFS are optimized by minimizing message exchanges in sparse accesses and by overlapping communication with file operations. MPI-IO/GPFS also takes advantage of GPFS programming features, such as data shipping, to achieve effective collective IO.

6 Conclusions

In this paper, we have shown that it is feasible to exploit the Lustre file joining feature for effective collective IO. We first show that IO middleware and programming libraries over Lustre need to be aware of its characteristics such as stripe width because excessive stripe width may incur significant striping overhead for both metadata and file read/write operations. We propose split writing and hierarchical striping to mitigate the striping cost while still being able to cover many storage devices. We have prototyped and evaluated these techniques inside a Lustre-specific ADIO implementation. Experimental results have shown our techniques are able to provide effective collective-IO and scalable management operations. The performance evaluation on other application IO benchmarks, such as BT-IO and MPI-Tile-IO, suggests that the benefits of Lustre file joining can be beneficial to scientific IO patterns, but further investigation is needed to increase the applicability of our techniques to general workloads and overcome its drawback of low performance with an existing joined file.

In the future, we intend to investigate how different striping policy can reshape the communication pattern between IO clients and storage devices, particularly in

an ultra-scale environment. We also plan to further exploit potential benefits of the two proposed techniques by applying a dynamic striping policy in Lustre, possibly with hints from applications.

Acknowledgment

This manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

We are very thankful to Dr. Philip Roth and Dr. Sarp Oral from Oak Ridge National Laboratory for their valuable technical comments and suggestions.

References

[1] IOzone Filesystem Benchmark. http://www.iozone.org/

[2] The Parallel Virtual File System, version 2. http://www.pvfs.org/pvfs2.

[3] TOP 500 Supercomputers. http://www.top500.org/.

[4] Argonne National Laboratory. ROMIO: A High-Performance, Portable MPI-IO Implementation. http://www-unix.mcs.anl.gov/romio/.

[5] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp. Noncontiguous I/O through PVFS. In *Proceedings of the IEEE International Conference on Cluster Computing*, Chicago, IL, September 2002.

[6] Cluster File System, Inc. Lustre: A Scalable, High Performance File System. http://www.Lustre.org/docs.html.

[7] A. M. David Nagle, Denis Serenyi. The Panasas ActiveScale Storage Cluster . Delivering Scalable High Bandwidth Storage. In *Proceedings of Supercomputing '04*, November 2004.

[8] R. Latham, R. Ross, and R. Thakur. The Impact of File Systems on MPI-IO Scalability. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2004)*, pages 87.96, September 2004.

[9] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, and R. Latham. Parallel netCDF: A High Performance Scientific I/O Interface. In *Proceedings of the Supercomputing '03*, November 2003.

[10] W. Liao, A. Ching, K. Coloma, A. Choudhary and L. Ward. Implementation and Evaluation of Client-side File Caching for MPI-IO. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium '07, March 2007.*

[11] W. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and S. Tideman. In *Proceedings of the Symposium on High Performance Distributed Computing 2005.*

[12] Z. Lin, S. Ethier, T. S. Hahm, and W. M. Tang. Size scaling of turbulent transport in magnetically confined plasmas. *Phys. Rev. Lett.*, 88(19):195004, Apr 2002.

[13] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an Optimized Implementation of MPI-IO on Top of GPFS. In *Proceedings of Supercomputing '01*, November 2001.

[14] R. Ross. Parallel I/O Benchmarking Consortium. http://www-unix.mcs.anl.gov/rross/pio-benchmark/html/.

[15] R. Ross, J. Moreira, K. Cupps and W. Pfeiffer. Parallel I/O on the IBM BlueGene/L System.

[16] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST '02*, pages 231.244. USENIX, Jan. 2002.

[17] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, S. Sekiguchi, "Grid Datafarm Architecture for Petascale Data Intensive Computing," *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*,

[18] O. Tatebe, N. Soda, Y. Morita, S. Matsuoka, S. Sekiguchi, "Gfarm v2: A Grid file system that supports high-performance distributed and parallel data computing," *Proceedings of the 2004 Computing in High Energy and Nuclear Physics (CHEP04)*, Interlaken, Switzerland, September 2004.

[19] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301.317, Winter 1996.

[20] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of Frontiers '96: The Sixth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, Oct 1996.

[21] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182.189. IEEE Computer Society Press, 1999.

[22] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23.32. ACM Press, May 1999.

[23] The National Center for SuperComputing. HDF Home Page. http://hdf.ncsa.uiuc.com/hdf4.html.

[24] The National Center for SuperComputing. HDF5 Home Page. http://hdf.ncsa.uiuc.com/HPD5/.

[25] P. Wong and R. F. Van der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division.

[26] W. Yu, Q. Gao, and D. K. Panda. Adaptive Connection Management for Scalable MPI over InfiniBand. In *International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, April 2006.