LACIO: A New Collective I/O Strategy for Parallel I/O Systems

Yong Chen [§], Xian-He Sun [†], Rajeev Thakur [‡], Philip C. Roth [♮], and William D. Gropp [♯]
[§] *Department of Computer Science, Texas Tech University, Lubbock, Texas, USA*
[†] *Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois, USA*
[‡] *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, USA*
[♮] *Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA*
[♯] *Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, Illinois, USA*
Email: {yong.chen@ttu.edu, sun@iit.edu, thakur@mcs.anl.gov, rothpc@ornl.gov, wgropp@illinois.edu}

***Abstract*—Parallel applications benefit considerably from the rapid advance of processor architectures and the available massive computational capability, but their performance suffers from large latency of I/O accesses. The poor I/O performance has been attributed as a critical cause of the low sustained performance of parallel systems. Collective I/O is widely considered a critical solution that exploits the correlation among I/O accesses from multiple processes of a parallel application and optimizes the I/O performance. However, the conventional collective I/O strategy makes the optimization decision based on the logical file layout to avoid multiple file system calls and does not take the physical data layout into consideration. On the other hand, the physical data layout in fact decides the actual I/O access locality and concurrency. In this study, we propose a new collective I/O strategy that is aware of the underlying physical data layout. We confirm that the new Layout-Aware Collective I/O (LACIO) improves the performance of current parallel I/O systems effectively with the help of noncontiguous file system calls. It holds promise in improving the I/O performance for parallel systems.**

***Keywords*-Parallel I/O; Collective I/O; Storage Systems; Parallel Applications; Parallel File Systems; High Performance Computing; Data Intensive Computing**

I. Introduction

With the rapid advance of semiconductor process technology and the evolvement of microarchitectures, the processor cycle times have been significantly reduced in the past decades. In addition, the widely-adopted multi-core/manycore architectures in recent years bring parallel processing on chip and significantly increase the computational performance of single processor chip. High-Performance Computing applications benefit from the processor architectural enhancement and massive computational capability considerably. However, compared to the processor performance improvement, data-access performance (latency and bandwidth) improvement has been at snail's pace. The disk drive speed has only increased by roughly 7% each year over the past two decades, which is significantly lower than the improvement speed of nearly 50% per year for processor performance [15]. This performance disparity is predicted to continually expand in next decades. Fig. 1 compares the single disk drive bandwidth improvement (left vertical axis) and the computational capability improvement of well-known supercomputers (right vertical axis) for the past decades [36]. The computational performance improvement rate is magnitudes higher than the bandwidth improvement rate of single drive. The rapid advance of processor architectures and computing capability has put ever more pressure on sluggish storage and I/O systems, especially for high-performance computing where performance is key.

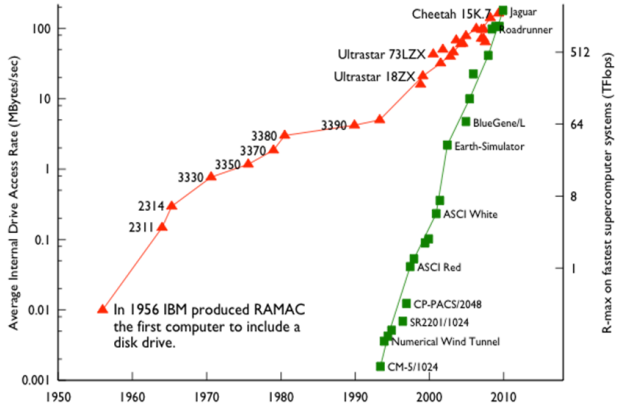


Figure 1. FLOPS of Supercomputers v.s. Single Disk Drive Bandwidth [36]. The computational performance improvement rate is magnitudes higher than the bandwidth improvement rate of single drive. Parallel I/O systems are essential to match the rapid advance of processor architectures and the fast increasing scale of computational capability.

Parallel I/O is essential to match the rapid advance of processor architectures and the fast increasing scale of computational capability. Although multi-level memory hierarchy architecture can avoid large performance loss due to long disk-access delays, the memory capacity is always limited. Furthermore, as the multicore/manycore architec-

* This research is sponsored in part by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. This research is also sponsored in part by the National Science Foundation under NSF grant CCF-0621435 and CCF-0937877. The work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC 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.

tures become universal, the available memory capacity per core even decreases, especially if the scale of HPC systems is projected to million cores or beyond. Many scientific and engineering simulations in critical areas of research, such as nanotechnology, astrophysics, climate, and high energy physics, are becoming more and more data intensive [36][26]. These applications contain a large number of I/O accesses, where large amounts of data are stored to and retrieved from storage systems. Table I shows the data requirements of several representative INCITE applications run at Argonne Leadership Computing Facility of Argonne National Laboratory in 2008 [36][21]. The data sets accessed by these applications are far beyond the memory capacity of supercomputers and need high-performance parallel I/O systems to meet their demands. There is a great need for research to improve the parallel I/O performance of high-performance computing systems.

Table I
DATA REQUIREMENTS OF SELECTED 2008 INCITE APPLICATIONS AT ALCF OF ANL [36]

Project	**On-Line Data**	**Off-Line Data**
FLASH: Buoyancy-Driven Turbulent Nuclear Burning	75TB	300TB
Reactor Core Hydrodynamics	2TB	5TB
Computational Nuclear Structure	4TB	40TB
Computational Protein Structure	1TB	2TB
Performance Evaluation and Analysis	1TB	1TB
Kinetics and Thermodynamics of Metal and Complex Hydride Nanoparticles	5TB	100TB
Climate Science	10TB	345TB
Parkinson's Disease	2.5TB	50TB
Plasma Microturbulence	2TB	10TB
Lattice QCD	1TB	44TB
Thermal Striping in Sodium Cooled Reactors	4TB	8TB
Gating Mechanisms of Membrane Proteins	10TB	10TB

Exploring parallelism and exploring physical locality have been the primary solutions to providing high-performance I/O systems to meet the ever-increasing demands of HPC applications. In this study, we take these two approaches to improve collective I/O, the most critical performance optimization strategy for parallel I/O middleware that connects users' applications and the underlying parallel file systems. We propose a new collective I/O strategy, called *Layout-Aware Collective I/O*, or *LACIO* in short. The motivation of this study comes from an observation that the existing collective I/O strategy is unaware of the physical data layout on underlying storage. The existing strategy makes the optimization decision based on the logical file layout only. The reason of this choice is that this strategy can minimize the number of file system calls, as the processing overhead of system calls is costly, and can balance the workload as well. However, the physical data layout actually decides the access locality and concurrency. The optimization decision based on the logical file layout does not necessarily translate to optimal physical locality and concurrency of accesses. The proposed new strategy LACIO enhances the existing strategy by taking the physical data layout into consideration. With the support of noncontiguous file system calls, it still keeps the overhead of system calls minimal. Furthermore, it rearranges accesses for better access locality and concurrency, and thus have a better *matched I/O*. The recent works in log-like reordering of accesses and intermediate library of rearranging accesses [2][22][46] have demonstrated the importance and significant potential of arranging data accesses in a proper manner. The primary goal of this research is to bring intelligent access awareness to parallel I/O middleware and collective I/O strategy to improve the parallel I/O system performance for high-performance computing.

The rest of this paper is organized as follows. Section II briefly reviews the parallel I/O software stack and the most important optimization strategy, collective I/O, in parallel I/O middleware. Section III introduces the idea, design and implementation methodology of the proposed LACIO. Section IV presents the experimental results of the proposed strategy. Section V discusses important related works in parallel I/O optimization and compares our work with existing works. Section VI concludes this study and discusses potential future work.

II. MPI-IO, COLLECTIVE I/O AND TWO-PHASE IMPLEMENTATION

In this study, we assume parallel applications are written in MPI (Message Passing Interface) [14][28], the dominant parallel programming model on all large-scale parallel machines, such as IBM Blue Gene/P and Cray XT5, as well as on many clusters of various sizes. We briefly review the MPI-IO, the I/O interface for parallel applications, and its common implementation as a middleware sitting between applications and the underlying parallel file systems in this section. We also review the most critical performance optimization strategy, collective I/O, in MPI-IO middleware and its commonly adopted two-phase implementation.

MPI-IO is a subset of the MPI-2 specification [14][26][28]. It defines an I/O access interface for parallel applications. The primary motivation for MPI-IO specification came from the observation that parallel I/O optimizations require two basic abstractions: the ability to define a set of processes (MPI communicators), and the ability to define complex data access patterns (MPI datatypes). MPI interface has already equipped the ability for these two abstractions. Therefore, built upon communicators and datatypes from MPI, the MPI-IO designers created an interface that supports many parallel I/O operations and optimizations. The implementation of MPI-IO is usually a middleware connecting parallel applications and underlying various parallel file systems, providing the code-level portability across many different machine architectures and operating systems. ROMIO is a popular MPI-IO implementation [39][34]. It

provides an abstract-device interface called ADIO [40] for implementing the portable parallel I/O API. It performs various optimizations, including collective I/O and data sieving, for common access patterns of parallel applications [39].

Collective I/O is one of the most important I/O access optimizations for parallel applications. It stands in contrast to independent I/O, in which each process of a parallel application issues I/O requests independently of all other processes. Although independent I/O is a straightforward form of I/O and is widely used in many applications, this form of I/O is not recommended for parallel applications because it does not capture the complete data access information of a parallel application. The independent I/O can be implemented directly with I/O system calls depending on specific underlying file systems. However, the implementation has no idea of what other processes might do and therefore have to service the I/O requests of each process individually.

With collective I/O, requests from all processes of a parallel application can be serviced together, allowing the middleware to take advantage of correlations between those requests. The motivation of collective I/O is several-fold. First, collective I/O can filter overlapping and redundant requests from multiple processes. Second, for many parallel applications, even though each process may access several noncontiguous portions of a file, the requests of multiple processes are often interleaved and may constitute a large contiguous portion of a file together [39]. Third, the collective I/O can reduce the number of system calls by combining small and noncontiguous requests into large and contiguous ones. Note that the collective I/O is a general idea that exploits the correlations among accesses from multiple processes of a parallel application and optimizes its I/O accesses. It can be applied at many levels, such as disk level [19], server level [37] or client level [39]. In this study, we focus on parallel I/O middleware level, i.e. the MPI-IO level. If the user chooses collective I/O semantics and provides the entire access information of a group of processes to the underlying MPI-IO middleware, the MPI-IO implementation can improve I/O performance significantly by combining the requests of different processes and servicing the combined aggregate requests.

The most popular method of implementing collective I/O is a two-phase strategy [35] (and its extension - generalized two-phase I/O [41]). This strategy separates the servicing of I/O requests into an I/O phase and data exchange phase (or communication phase). Fig. 2 shows the strategy of a two-phase collective I/O read for four processes. In this example, we assume two processes participate in the I/O phase (the processes participating the I/O phase are termed as *aggregators* and the number of aggregators can be specified by users) and each aggregator has sufficient memory for temporary buffer. The two-phase I/O implementation has

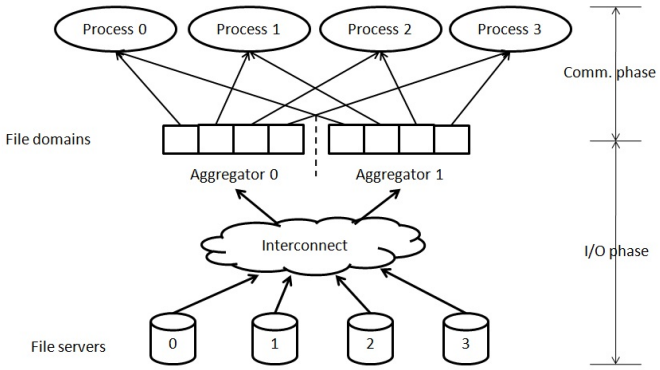


Figure 2. Collective I/O and Two-phase Implementation. Collective I/O optimizes parallel I/O performance by combining the requests of multiple processes and servicing the combined aggregate request. The two-phase implementation strategy separates the servicing of I/O requests into an I/O phase and data exchange phase (or communication phase). This figure demonstrates the strategy of a two-phase collective I/O read for four processes with two processes participating as aggregators.

a first-round of communication to let each aggregator know the aggregated span of the I/O requests of all processes. The implementation then partitions the aggregated span of requests into multiple file domains with each aggregator responsible for carrying out I/O requests for its own file domain. This phase is called the I/O phase. In the data exchange phase, each aggregator sends data to the requesting processes, and each process receives its required data from corresponding aggregators that fetch the data on behalf of it.

III. Layout-Aware Collective I/O (LACIO) Design and Methodology

In this section, we present the idea and the design of the Layout-Aware Collective I/O (LACIO) strategy. We first present a motivating example, and then introduce the new LACIO strategy. We also present the implementation methodology of the LACIO and analyze its impact on I/O accesses.

A. A Motivating Example

Fig. 3 demonstrates a representative example of current collective I/O strategy and the data layout on file servers. We assume to have four aggregators serving the I/O requests from all processes of a parallel application collectively in this example. We illustrate the access details and show the relationship between I/O requests and the actual data layout on file servers in this figure. The logical block number, i.e. LB#, shows the position of I/O requests. The file server number, i.e. S#, represents the file server where the requested data reside on. We assume there are four file servers/storage nodes. The data is assumed to be distributed across all file servers with the most common round-robin data layout strategy.

The current collective I/O strategy adopts a file domain partitioning and request scheduling method by evenly partitioning the aggregate request into equal-length sub-requests based on the logical file layout. Each aggregator is then responsible for carrying out the I/O request for its own file domain. For instance, the aggregator 0 carries out the I/O request for LB#0-2, while aggregator 1, 2 and 3 carry out I/O requests for LB#3-5, LB#6-8 and LB#9-11 respectively. The goal of this strategy is to balance the workload for each aggregator, as the performance of a collective I/O will be decided by the slowest aggregator. In addition, the partitioning method based on the logical file layout can minimize the number of file system calls and the overhead involved. However, since the file is striped across multiple server nodes, this existing file domain partitioning and request scheduling strategy would make each aggregator generates many concurrent and burst requests to all file servers (not the metadata server) simultaneously. Such a strategy without being aware of the physical data layout could in turn generate access contention and lose the physical access locality. For instance, as shown in Fig. 5b with the detailed communication and I/O pattern, the existing strategy leads to file server 0 serving aggregator 0, 1 and 2 respectively with one striping unit size of request, while file servers 1, 2 and 3 serving three different aggregators with one striping unit size of request respectively. This access method loses the access locality to physical storage and increases the access contention. As demonstrated in the following subsection, the proposed new LACIO strategy rearranges accesses to increase the access locality and reduce the access contention.

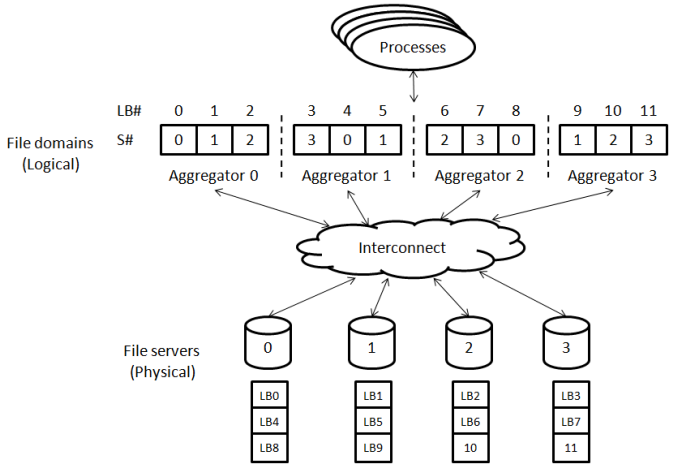


Figure 3. A Representative Example of Collective I/O and Data Layout. In this example, four aggregators serve a group of compute processes. The data is distributed across four file servers following the round robin layout strategy. The existing collective I/O strategy adopts a file domain partitioning and request scheduling method by evenly partitioning the aggregate request into equal-length sub-requests based on the logical file layout with each aggregator responsible for carrying out the I/O request for its own file domain.

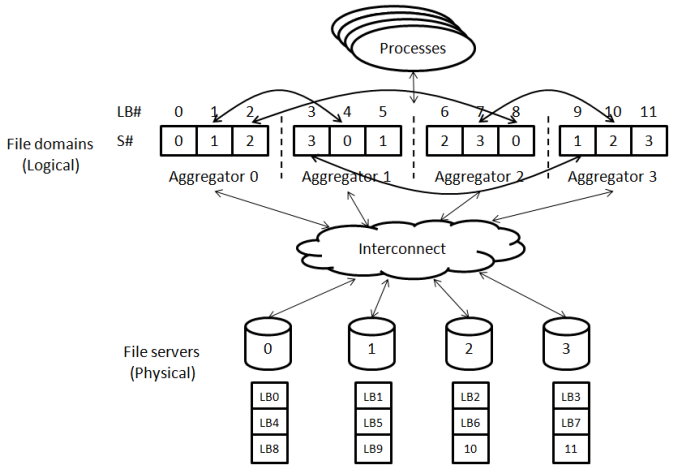
B. LACIO Design, Methodology, and Analysis

1) LACIO Design: As shown in Fig. 3 and explained in the previous section, a limitation of the current collective I/O strategy and the implementation is that they are unaware of the physical layout of data on file servers and storage devices. The calculation of the span of the combined I/O requests and the creation of the file domains are based on the logical file layout only. Even though the file domain that each aggregator is responsible for carrying out I/O requests is logically contiguous, it does not translate to physical contiguity, as which is decided by the layout strategy of the underlying parallel file systems. We propose to incorporate the physical layouts of data distribution among servers, the information from parallel file systems, with parallel I/O middleware, and rearrange file domain's partition and the requests from aggregators in a fashion that matches with the physical layout on servers. We call this new collective I/O methodology a *Layout-Aware Collective I/O (LACIO)* strategy.

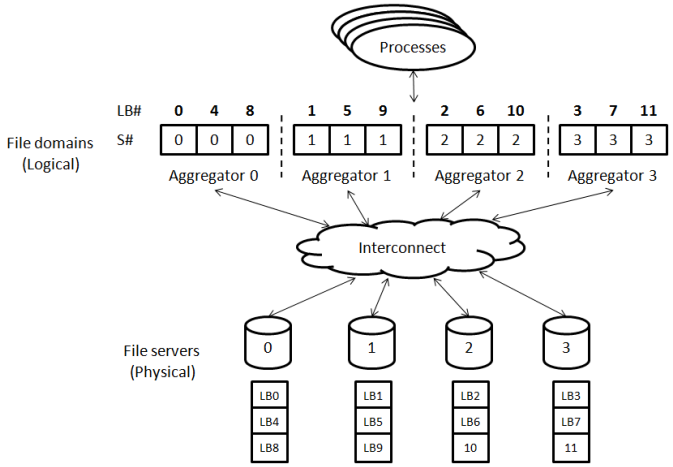
Following the previous example, Fig. 4 illustrates the idea of the proposed LACIO strategy. The proposed strategy rearranges the partitions of file domains and the requests of aggregators such that: 1) the requests are grouped and need as few file servers as possible to reduce access contention and exploit better concurrency; 2) the requests are reordered to be physically contiguous as much as possible to exploit better locality.

Fig. 4a demonstrates how file domain partitions and access requests are rearranged following the new LACIO strategy with the previous example. Fig. 4b illustrates the accesses of each aggregator after rearranging and reordering. These examples demonstrate that the LACIO strategy rearranges requests of aggregators to have each aggregator accesses data on file servers contiguously, and multiple aggregators access file servers concurrently. In this design, we assume that the data layout information can be obtained from the API provided by the underlying parallel file systems. It is not unusual that parallel file systems provide the interface to inquire the data layout on file servers, such as in PVFS2 [7][32]. Note that the rearrangement here is to change the requests that each aggregator carries out on behalf of the processes. In other words, the rearrangement changes the responsible portion of the aggregators and the way the aggregators access data. It is critical to note that the rearrangement does not exchange data themselves among aggregators. Also, note that the I/O requests in this example can be either reads or writes. The LACIO strategy is designed for both I/O reads and writes.

2) LACIO Implementation Methodology: The implementation methodology of the proposed LACIO is not complicated with the underlying parallel file system support. Since LACIO is a collective operation over all processes that open the file (associated with the file handle), the implementation



(a) Collective I/O with Layout Awareness



(b) Collective I/O with Layout Awareness After Reordering

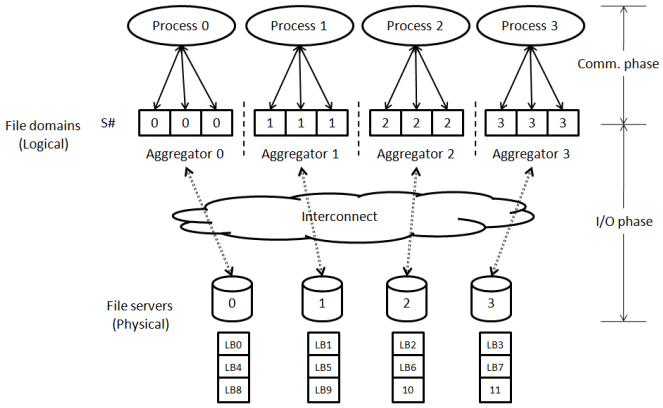
Figure 4. Layout-aware Collective I/O (LACIO). This new LACIO strategy incorporates the physical layouts of data distribution among servers and rearranges file domain's partition and the requests from aggregators in a fashion that matches with the physical layout on servers. It is designed to have each aggregator accesses data on file servers contiguously, and multiple aggregators access file servers concurrently. The new LACIO strategy reduces the access contention and improves the request locality.

obtains the data layout information (including the striping width, striping factor, and layout strategy) via parallel file systems API and then broadcast to all aggregators, instead of letting each aggregator retrieve the data layout information respectively. The obtaining of data layout knowledge happens at the very first time a collective I/O occurs. The data layout information is then cached in each aggregator. Such a caching is safe as the data layout of a specific file is determined when it is created and will be static except deleted or explicitly changed.

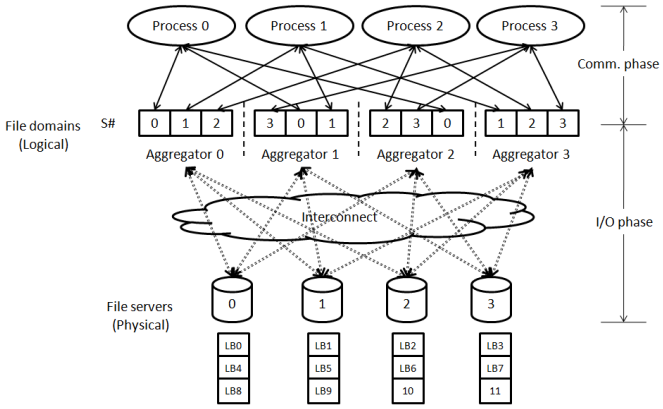
After obtaining the layout knowledge, all aggregators partition file domains and rearrange requests follow the design in each collective I/O operation. Depending on the number of aggregators configured and the striping factor, three mappings between aggregators and file servers are possible, including multiple-to-one, one-to-one, and one-to-multiple mappings. The desired case would be the one-to-one mapping, as shown in the previous examples, which is similar to the software I/O forwarding layer currently under active exploration of parallel I/O researchers. In the case of multiple-to-one or one-to-multiple mapping, the implementation can balance the load by considering the ratio between the number of aggregators and the number of file servers.

A common concern for the LACIO strategy is that it might result in many file system calls and incur extra overhead, as the partitioned requests are not logically contiguous anymore. However, with the advanced parallel file system development and the support of noncontiguous system call access [11], the implementation can still reduce the number of system calls and the overhead incurred. As the I/O requests of aggregators rearranged, the communication between aggregators and compute processes (I/O client processes) are changed correspondingly. This can be done by exchanging the requests information among aggregators and compute processes at the very beginning of each collective I/O operation. Such information will be used to decide how data is exchanged in the communication phase. Fig. 5 compares the communication and I/O pattern of the LACIO strategy and the conventional strategy. It can be seen that the new LACIO strategy groups accesses with the consideration of data layout information and results in the accesses in a neater and matched way.

3) LACIO Analysis: The proposed LACIO strategy might raise an interesting question - why do we need all these hassles by combining I/O requests from parallel processes and then splitting and carrying out them? Note that, considering the noncontiguous accesses from multiple processes of an application, the conventional strategy combines these noncontiguous accesses and then split them in a logically contiguous way. The new LACIO is essentially combining noncontiguous accesses and split them in a logically noncontiguous way but with better physical locality and reduced access contention. A special case would be, if the underlying file system only utilizes one file server/storage node, the proposed LACIO would have the same way of combining noncontiguous accesses from multiple processes and carrying out them contiguously as the conventional strategy does. The reason why the new LACIO strategy is desired and could be beneficial is several-fold. First, this strategy still performs collective I/O, which means the overlapping and redundant requests are still removed. Second, the number of requests to the parallel file system are still controlled by taking advantage of the noncontiguous file system calls supported by advanced parallel file systems. Third, the access rearranging and reordering can exploit better locality and reduce access contention and thus achieve better performance.



(a) LACIO Strategy



(b) Conventional Collective I/O Strategy

Figure 5. Communication and I/O Pattern: LACIO Strategy v.s. Conventional Collective I/O Strategy. The new LACIO strategy results in I/O accesses in a neater and better matched way.

IV. Experimental Results and Analysis

We have performed experimental tests for the new LACIO and compared it with the existing collective I/O strategy. The experiments were conducted with several benchmarks and one user-level checkpointing/restart application. We first briefly describe the experimental environment and then present the experimental testing results.

A. Experimental Setup

Our experiments were conducted on a 65-node Sun Fire Linux-based cluster. This cluster is composed of one Sun Fire X4240 head node, with dual 2.7 GHz Opteron quad-core processors and 8GB memory, and 64 Sun Fire X2200 compute nodes with dual 2.3GHz Opteron quad-core processors and 8GB memory. The head node has 12 500GB 7.2K-RPM SATA-II drives configured as RAID-5 system. Each compute node has a 250GB 7.2K-RPM SATA hard drive. All 65 nodes are connected with Gigabit ethernet. The experiments were tested with MPICH2-1.0.5p3 release and PVFS 2.8.1 file system on Ubuntu 4.3.3-5 system with kernel 2.6.28.10. We configured PVFS2 with 32 I/O server nodes. The rest of 32 nodes were used as client nodes.

B. Experimental Results of A Synthetic Benchmark

This set of tests were carried out with a synthetic benchmark to measure the performance of LACIO and the existing collective I/O strategy. This synthetic benchmark does strided reads for each process, and the aggregated requests of all processes are sequential reads over the file. We performed a series of tests on the Sun Fire cluster to compare the performance of LACIO and the existing strategy. The total size of the data accessed by all processes are 64MB, 160MB, 320MB, 800MB and 4000MB respectively. The results are shown in Fig. 6.

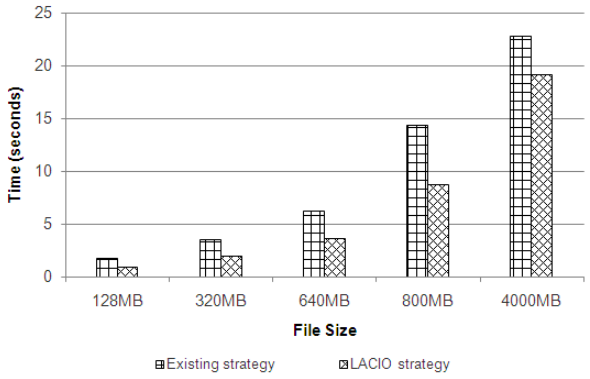


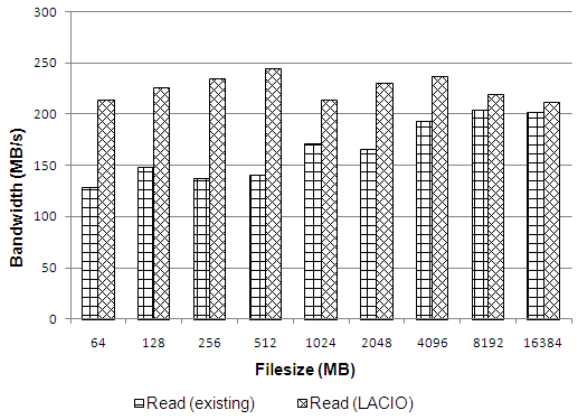
Figure 6. Comparison of LACIO and Existing Collective I/O with Synthetic Benchmark Testing. The benchmark does strided reads for each process, and the aggregated requests of all processes are sequential reads over the file with varying the total amount of data accessed.

It can be observed that the LACIO strategy with layout awareness could have a considerable impact on the performance of parallel I/O system. The performance variation was up to 48.8% and the average performance improvement was nearly 38%.

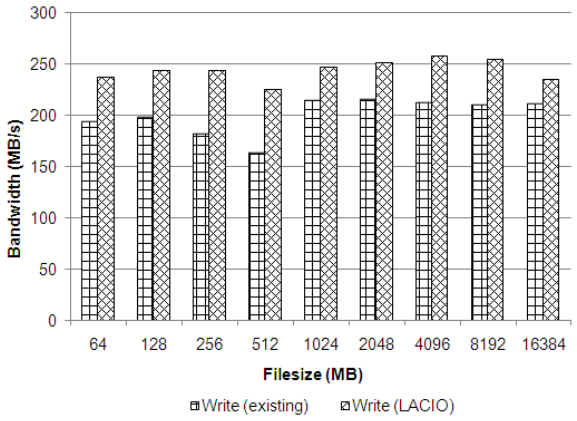
C. Experimental Results of IOR Benchmark

Fig. 7 and Fig. 8 report the testing results with IOR-2.10.2 benchmark from Lawrence Livermore National Laboratory [17]. In these experiments, we performed the test with 64 processes on 32 client nodes (client nodes are separate from I/O server nodes) with half of them configured as aggregators. We performed both interleaved reads/writes and random reads/writes tests, and varied the file size from 64MB to 16GB. Fig. 7a and Fig. 7b report the bandwidth of read and write testing respectively with the existing collective I/O strategy and the proposed LACIO in the random access test case. It can be observed that the LACIO strategy can clearly improve the I/O access performance. The performance improvement of read tests are more sensitive to the new strategy, as it varied from 4.8% improvement to 74.1% improvement. The performance improvement of write

tests are less sensitive - the performance speedup varied from 11.3% to 38.4%. The average performance improvement for read and write tests are 40.4% and 22.7% respectively.



(a) Reads

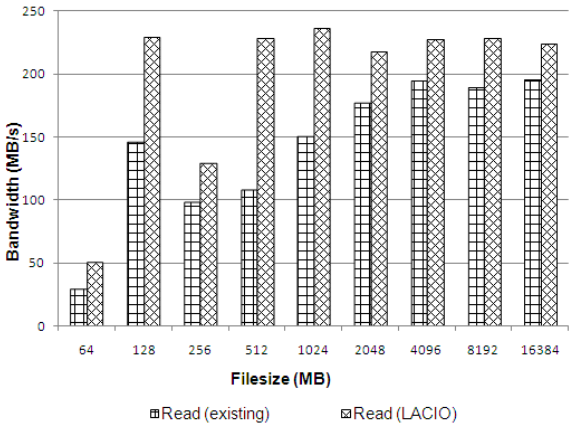


(b) Writes

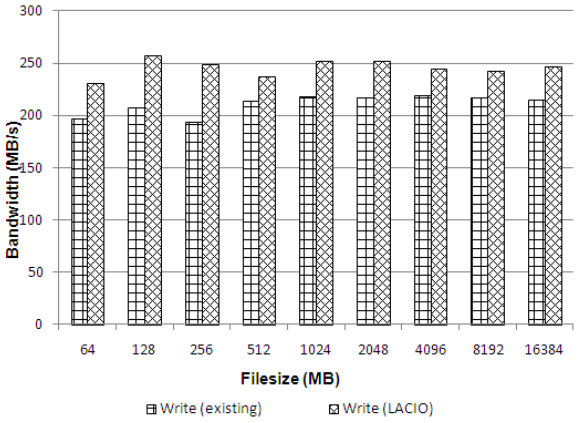
Figure 7. Comparison of LACIO and Existing Collective I/O with IOR Benchmark Testing with Random Accesses. The results are the reported bandwidth of reads and writes respectively with the existing collective I/O strategy and the new LACIO strategy and varying the total amount of data accessed.

Fig. 8a and Fig. 8b report the bandwidth of read and write testing respectively in the interleaved test case. While the access pattern is quite different from the previous testing, the LACIO strategy can overcome the impact of different application access patterns as it rearranges and reorders accesses, and improves the I/O access performance over the existing strategy constantly. The average performance improvement for read and write tests are 45.3% and 16.7% respectively in this series of tests. The performance improvement trend is close to the previous series of tests. The reads are more sensitive to the optimization while the performance improvement of writes is relatively stable in most cases.

As can be seen from these results, the LACIO strategy can affect the IOR benchmark testing performance considerably.



(a) Reads



(b) Writes

Figure 8. Comparison of LACIO and Existing Collective I/O with IOR Benchmark Testing with Interleaved Accesses. The results are the reported bandwidth of reads and writes respectively with the existing collective I/O strategy and the new LACIO strategy and varying the total amount of data accessed.

The LACIO could improve the I/O bandwidth up to 74%, 38%, 112% and 28% for random reads, random writes, interleaved reads and interleaved writes, respectively. The average potential improvement of the LACIO strategy with different file sizes was 40%, 23%, 45% and 16% for random reads, random writes, interleaved reads and interleaved writes respectively as shown in Fig. 9.

D. Experimental Results of User-Level Checkpointing Application

Fig. 10 reports the results of a user-level checkpointing/restart application, where the user's program explicitly checkpoints the program data structures to the underlying storage. The tests were carried out with 64 and 128 processes respectively. The total checkpoint image size remains the same as 6GB in total for all processes in each test. By utilizing the new LACIO, the performance was improved by 15% and 28% when tested with 64 and 128 processes

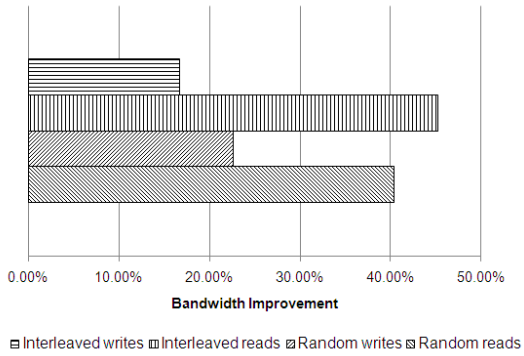


Figure 9. Average Bandwidth Improvement of LACIO

respectively. Note that, even though the total image size remains the same, the performance varied with different number of processes due to the different access contention and request locality. However, in each case, the LACIO strategy constantly improves the overall performance.

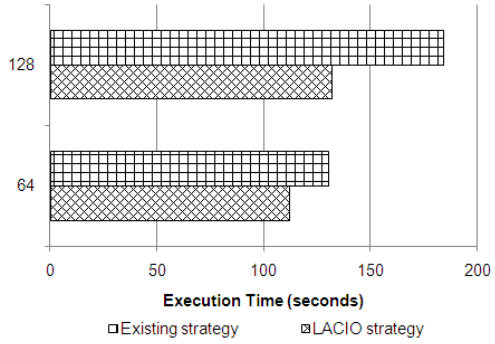


Figure 10. Application-level Checkpointing with Existing Collective I/O and LACIO

V. Related Work

Extensive studies have focused on improving parallel I/O access performance at various levels. At the hardware level, disk bandwidth has only improved at a very slow pace, while the capacity has been increasing rapidly to petabytes and even beyond. Research in software optimizations can be roughly classified into the areas of runtime I/O libraries [26][39][6][41][30][37][19], parallel file systems [7][9][38][42], caching, prefetching and data distribution strategies[29][27][23][31][13][25][16][42][20].

A. Parallel I/O Runtime Library and File System Optimizations

There has been significant amount of research effort in optimizing I/O performance using runtime libraries, such as collective I/O [26][39], two-phase I/O [6], extended two-phase I/O [41], data sieving [39], server-direct I/O [37] and disk-directed I/O [19]. Recently, Zhang et. al. proposed to find and match the I/O request pattern with the striping pattern to make I/O resonance a common case [46]. Their study demonstrated the great performance improvement with considering proper I/O request arrangement. Ali et. al. proposed a new I/O forwarding software layer sitting between the MPI-IO and parallel file systems to ship the I/O calls to dedicated I/O nodes and improves the scalability of parallel I/O systems [1]. Iskra et. al. introduced an I/O forwarding component in the ZeptoOS for petascale architectures such as IBM Blue Gene systems [18]. All these strategies collect and merge small requests into larger requests at the I/O client/middleware/server level, or aggregates and ships I/O calls to dedicated processes/nodes. This study addresses the issues in the existing collective I/O and proposes a new LACIO. It advances the state of the art in these areas.

Parallel file systems, such as Lustre [9], GPFS [38], PanFS [45], PVFS [7], and PPFS2 [42], enable concurrent I/O accesses from multiple clients to files. All these file systems provide high bandwidth for large, well-formed parallel I/O requests. In this study, we propose to reveal layout information to parallel I/O middleware via the underlying file system, which fosters a better integration of the parallel I/O middleware and parallel file systems, the two major components of parallel I/O systems, and improves the overall parallel I/O performance.

B. Parallel I/O Caching and Prefetching

Many research efforts have also been devoted at caching and prefetching optimizations for parallel I/O systems. Liao et al. proposed collective caching at MPI-IO layer to construct a global cache pool to enhance parallel I/O accesses performance [23]. Nitzberg et al. proposed collective buffering [29], and Ma et al. proposed active buffering [27] to boost the output performance of many scientific applications. Vilayannur et al. proposed discretionary caching for parallel I/O to use compilation and runtime support to bypass caching if the caching hurts the performance [43]. Eshel et al. designed a cluster file system cache named Panache that exploits parallelism in many aspects of its design and has been proven effective and scalable as a parallel file system cache [12]. Several parallel or distributed file systems, such as PanFS [45] and Ceph [44], also provide client-side caching to improve file system performance.

Patterson et al. [31], Griffioen and Appleton [13] proposed prefetching strategies using compiler, runtime, and access pattern information. Various pattern based file prefetching methods have been proposed [25][16][42] to improve I/O performance of applications with regular data access. Many other I/O prefetching strategies have been proposed in both heuristic prediction based approaches and speculative execution based approaches including Chang and Gibson's SpecHint [10] and Patterson's informed prefetching TIP [31]. Tran et al. proposed time series modeling to provide

efficient adaptive prefetching [42]. Recently, a more aggressive pre-execution based prefetching [8], where a prefetching thread runs ahead of main computing thread to prefetch data, was introduced. Besides, a signature based prefetching with post-execution analysis and runtime adjustment was introduced in [3]. Blas et al. proposed multiple-level caching and one-level prefetching for Blue Gene systems based on ROMIO [5].

C. Data Layout and Access Optimization

In parallel file systems, file data is distributed among multiple I/O servers and disks to provide higher degree of parallelism. Parallel file systems, including Lustre [9], GPFS [38], PanFS [45] and PVFS2 [7], implement a simple striping data distribution function, that data is distributed using a fixed block size in a round-robin manner among available I/O servers and disks. Considering applications' access information and data layout information on file servers to optimize accesses is possible at various levels including application level, middleware level, and file system level. At application level, many techniques have been developed for accessing data in a way that improves disk access parallelism by modifying application code [20]. The problem is that these strategies are not transparent to developers and often introduce extra programming burden. Library-level optimizations, such as data sieving and two-phase I/O, are helpful in reducing the number of requests to the file system [4][39]. However, as demonstrated in this study, when combined and optimized with physical layout information of file systems, we can achieve an even better result.

The data layout optimization at file system level primarily focuses on providing variant data distribution strategies for a variety of I/O workloads and user requirements. For instance, PVFS2 [7] uses simple striping by default, and provides two more data distribution strategies called variable striping and two dimensional striping [33]. While many optimizations exist, there lacks sufficient study that investigates a better collective I/O strategy with aware of the physical data layout. In this study, we propose a new LACIO strategy with the goal of improving request locality and reducing access contention, and have confirmed its benefits.

VI. Conclusion and Future Work

With the tremendous advance in processor architectures and the computational capability, I/O has been widely recognized as the bottleneck in high-performance computing for many applications. In this study, we propose a new collective I/O strategy, called *Layout-Aware Collective I/O (LACIO)*, to optimize parallel I/O performance and foster a better integration of parallel I/O middleware and parallel file systems. While both of the parallel I/O middleware and parallel file systems technologies have made their success, little has been done to investigate a layout-aware strategy for collective I/O and a better integration of these two parallel I/O subsystems to improve the overall performance.

The contribution of this study is three-fold. First, we demonstrate that it is beneficial to integrate layout awareness to collective I/O strategy, one of the most important optimizations in parallel I/O systems. Second, we propose a new LACIO to match the partitioning of file domains and the request scheduling with physical layout to achieve a better *matched I/O* form. This optimization is transparent to users and benefits users' applications automatically as long as the collective MPI-IO API is used in the applications. This new strategy improves request locality and reduce access contention. In addition, although the existing parallel I/O systems provide high bandwidth for simple, well-formed, and generic I/O access characteristics, the performance varies from application to application due to different access patterns. The proposed new collective I/O strategy rearranges and reorders accesses at MPI-IO layer and can potentially overcome the impact of various application patterns on the I/O performance. Third, as the experimental results demonstrate, the new LACIO can clearly improve the parallel I/O system performance. It is a promising new strategy for collective I/O. Furthermore, this study reveals that an API to obtain the relevant physical information about the file system and data storage is essential to the performance optimization of the overall parallel I/O systems and should be standardized as much as possible. A standardized API can also make these optimizations apply to non-proprietary or vertically integrated systems.

In the near future, we plan to further explore the potential of layout awareness optimization in parallel I/O. For instance, the layout aware strategy can rearrange accesses and align them to locking boundary, which has been demonstrated beneficial in [24]. We plan to continue exploring the optimization opportunity in reducing both access and locking contention to further improve parallel I/O system performance.

Acknowledgment

The authors are thankful to anonymous reviewers' valuable suggestions and comments that help the further improvement of this work.

References

[1] N. Ali, P. H. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. B. Ross, L. Ward, P. Sadayappan. Scalable I/O Forwarding Framework for High-performance Computing Systems. Proceedings of the 2009 IEEE International Conference on Cluster Computing, 2009.

[2] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In Proc. of ACM/IEEE Supercomputing Conference, 2009.

[3] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp. Parallel I/O Prefetching Using MPI File Caching and I/O Signatures. In Proc. of the ACM/IEEE Supercomputing Conference, 2008.

[4] J. G. Blas, F. Isaila, D. E. Singh, and J. Carretero. View-based Collective I/O for MPI-IO. In Proc. of IEEE International Symposium on Cluster Computing and the Grid, 2008.

[5] J. G. Blas, F. Isaila, J. Carretero, R. Latham, R. Ross. Multiple-Level MPI File Write-Back and Prefetching for Blue Gene Systems. In Proc. of PVM/MPI 2009.

[6] R. Bordawekar, J. M. Rosario, and A. N. Choudhary. Design and Evaluation of primitives for Parallel I/O. In Proc. of ACM/IEEE Supercomputing Conference, 1993.

[7] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In Proc. of the 4th Annual Linux Showcase and Conference, 2000.

[8] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, and W. Gropp. Hiding I/O Latency with Pre-execution Prefetching for Parallel Applications. In Proc. of the ACM/IEEE Supercomputing Conference, 2008.

[9] Cluster File Systems Inc. Lustre: A Scalable, High Performance File System. Whitepaper, http://www.lustre.org/docs/whitepaper.pdf.

[10] F. Chang and G. A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In Proc. of the 3rd Symposium on Operating Systems Design and Implementation, 1999.

[11] A. W. Ching, A. Choudhary, W.-K. Liao, R. Ross, and W. Gropp. Noncontiguous I/O through PVFS. Proceedings of 2002 IEEE International Conference on Cluster Computing, 2002.

[12] M. Eshel, R. L. Haskin, D. Hildebrand, M. Naik, F. B. Schmuck, R. Tewari. Panache: A Parallel File System Cache for Global File Access. In Proc. of the 8th USENIX Conference on File and Storage Technologies, 2010.

[13] J. Griffioen, and R. Appleton. Performance Measurements of Automatic Prefetching. In Proc. of the 8th International Conference on Parallel and Distributed Computing Systems, 1995.

[14] W. D. Gropp, E. Lusk, and R. Thakur. Using MPI-2. MIT Press, 1999.

[15] J. Hennessy, and D. Patterson. Computer Architecture: A Quantitative Approach, 2006.

[16] T. Highley, and P. Reynolds. Marginal Cost-Benefit Analysis for Predictive File Prefetching. In Proc. of the 41st Annual ACM Southeast Conference, 2003.

[17] Interleaved or Random (IOR) Benchmark. http://sourceforge.net/projects/ior-sio/.

[18] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman. ZOID: I/O Forwarding Infrastructure for Petascale Architectures. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 153 - 162, 2008.

[19] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. ACM Transactions on Computer Systems, 15(1):41-74, 1997.

[20] M. A. Kandaswamy, M. Kandemir, A. Choudhary, and D. Bernholdt. An Experimental Evaluation of I/O Optimizations on Different Applications. IEEE Transactions on Parallel and Distributed Systems, 13(12), 2002.

[21] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O Performance Challenges at Leadership Scale. In Proc. of ACM/IEEE Supercomputing Conference, 2009.

[22] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and Integration for Scientific Codes Through the Adaptable IO System (ADIOS). In Proc. of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, 2008.

[23] W.-K. Liao, A. Ching, K. Coloma, A. Choudhary, and L. Ward. An Implementation and Evaluation of Client-Side File Caching for MPI-IO. In Proc. of IEEE International Parallel and Distributed Processing Symposium, 2007.

[24] W.-K Liao, and A. N. Choudhary. Dynamically Adapting File Domain Partitioning Methods for Collective I/O based on Underlying Parallel File System Locking Protocols. In Proc. of ACM/IEEE Supercomputing Conference, 2008.

[25] H. Lei, and D. Duchamp. An Analytical Approach to File Prefetching. In Proc. of the 1997 USENIX Annual Technical Conference, 1997.

[26] J. May. Parallel I/O for High Performance Computing. Morgan Kaufmann Publishing, 2001.

[27] X. Ma, M. Winslett, J. Lee, and S. Yu. Faster Collective Output through Active Buffering. In Proc. of IEEE International Parallel and Distributed Processing Symposium, 2002.

[28] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org/docs/docs.html, 1996.

[29] B. Nitzberg, and V. M. Lo. Collective Buffering: Improving Parallel I/O Performance. In Proc. of High-performance Parallel and Distributed Computing, 1997.

[30] R. Oldfield, L. Ward, R. Riesen, A. B. Maccabe, P. Widener, and T. Kordenbrock. Lightweight I/O for Scientific Applications. In Proc. of IEEE International Conference on Cluster Computing, 2006.

[31] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In Proc. of the 15th ACM Symposium on Operating Systems Principles, 1995.

[32] PVFS2 Development Team. PVFS Developer's Guide. http://www.pvfs.org/cvs/pvfs-2-8-branch-docs/doc//pvfs2-guide.pdf.

[33] PVFS Development Team. PVFS2 Tuning. http://www.pvfs.org/cgi-bin/pvfs2/viewcvs/viewcvs.cgi/pvfs2/doc/pvfs2-tuning.tex#rev1.2.

[34] ROMIO website. http://www-unix.mcs.anl.gov/romio/.

[35] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In Proc. of the Workshop on I/O in Parallel Computer Systems at IPPS, 1993.

[36] R. Ross, R. Latham, M. Unangst, and B. Welch. Paralell I/O in Practice, tutorial in the ACM/IEEE Supercomputing Conference, 2009.

[37] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-Directed Collective I/O in Panda. In Proc. of Supercomputing Conference, 1995.

[38] F. Schmuck, and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In First USENIX Conference on File and Storage Technologies, 2002.

[39] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation, 1999.

[40] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation, 1996.

[41] R. Thakur, and A. Choudhary. An Extended Two-Phase Method for Accessing S ections of Out-of-Core Arrays. Scientific Programming, (5)4:301-317, 1996.

[42] N. Tran, and D. A. Reed. Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching. IEEE Transactions on Parallel and Distributed Systems, 15(4):362-377, 2004.

[43] M. Vilayannur, A. Sivasubramaniam, M. T. Kandemir, R. Thakur, R. Ross. Discretionary Caching for I/O on Clusters. Cluster Computing 9(1): 29-44, 2006.

[44] S. A. Weil, S. A. Brandt, E. L. Miller, D. D.E. Long, C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In Proc. of USENIX Symposium on Operating Systems Design and Implementation, 2006.

[45] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B.Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In Proc. of the 6th USENIX Conference on File and Storage Technologies, 2008.

[46] X. Zhang, S. Jiang, and K. Davis. Making Resonance a Common Case: A High-performance Implementation of Collective I/O on Parallel File Systems. In Proc. of the 23rd IEEE International Symposium on Parallel and Distributed Processing, 2009.