Improving Parallel I/O Performance with Data Layout Awareness [♯]

Yong Chen [§], Xian-He Sun [‡], Rajeev Thakur [*], Huaiming Song [‡], Hui Jin [‡]
[§] *Computer Science and Mathematics Division, Oak Ridge National Laboratory, USA*
[‡] *Department of Computer Science, Illinois Institute of Technology, USA*
[*] *Mathematics and Computer Science Division, Argonne National Laboratory, USA*
Email: {cheny@ornl.gov, sun@iit.edu, thakur@mcs.anl.gov, hsong20@iit.edu, hjin6@iit.edu}

Abstract

Parallel applications can benefit greatly from 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 computing systems. In this study, we propose a data layout-aware optimization strategy to promote a better integration of the parallel I/O middleware and parallel file systems, two major components of the current parallel I/O systems, and to improve the data access performance. We explore the layout-aware optimization in both independent I/O and collective I/O, two primary forms of I/O in parallel applications. We illustrate that the layout-aware I/O optimization could improve the performance of current parallel I/O strategy effectively. The experimental results verify that the proposed strategy could improve parallel I/O performance by nearly 40% on average. The proposed layout-aware parallel I/O has a promising potential in improving the I/O performance of parallel systems.

Keywords: parallel I/O, parallel file systems, parallel I/O middleware, collective I/O, independent I/O, data layout, I/O performance, data access optimization

1. Introduction

Supercomputers have crossed the Petaflop performance mark and are moving forward to reach the Exaflop range [43]. However, while computing resources are making rapid progress, there is a significant gap between processing capacity and data-access performance. Due to this gap, although processing resources are available, they have to stay idle waiting for data to arrive, which leads to a severe overall performance degradation. Fig. 1 shows the number of CPU cycles required to access cache memory (SRAM), main memory (DRAM), and disk storage [6]. It can be seen that the number of cycles for accessing storage is hundreds of thousands of times larger. This trend is predicted to continue in the near future. In the meantime, 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 [28]. These applications contain a large number of I/O accesses, where large amounts of data are stored to and retrieved from disks. However, the disparity of technology growth is causing a gap between processor performance and storage performance that has been increasing over the last few decades. This poor I/O performance has been attributed as the cause of low sustained performance of existing parallel computing systems.

Data layout describes how data is distributed among multiple file servers. It is a crucial factor that determines the data access latency and the I/O subsystem performance for parallel computing systems. The recent works in log-like reordering of accesses and intermediate library of rearranging accesses [2][25] have demonstrated the importance and significant potential of arranging data accesses in a proper manner. However, historically, parallel I/O middleware and parallel file systems, two critical components to provide high data-access bandwidth for parallel systems, are developed separately with a separated modular design in mind. This separation enhances the transparency between different parallel I/O components at distinct layers and eases the software implementation. Nev-

[♯] *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.*

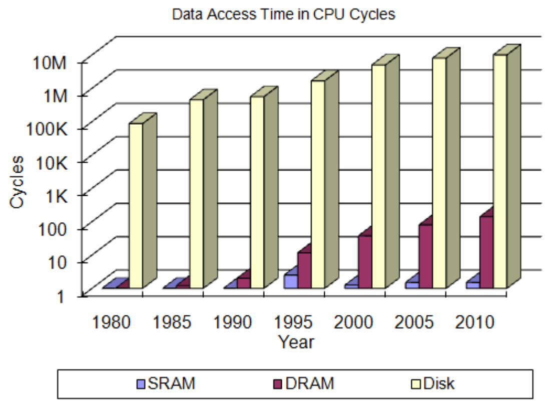


Figure 1. Comparison of Data Access Latency. This chart shows the data access time of SRAM, DRAM and disk in CPU cycles from year 1980 to 2010. The disk I/O is hundreds of thousands of times slower and this trend is predicted to continue in the next decades.

ertheless, this separation can lose the potential optimization opportunity that could benefit the overall performance of parallel I/O systems. For instance, the collective I/O, one of the most important optimization strategies in parallel I/O, often relies on the logical layout of file accesses from multiple processes, other than the physical data layout on file servers, because of lacking data layout information. On the other hand, it is the parallel file system that determines the physical layout of data among multiple file servers and thus determines the access latency and concurrency. With aware of such layout information, the overall parallel I/O system could perform better. However, the current parallel I/O strategy does not explore such layout-aware optimization well.

In this study, we argue that it could be beneficial if we have such a layout-aware parallel I/O optimization to get a better integration of parallel I/O middleware and parallel file systems. We propose to consider the physical data layout and data locality into parallel I/O strategy, and thus have a better matched I/O. We exploit the layout-aware optimization in both independent I/O and collective I/O, two forms of I/O that are widely used in parallel applications. The rest of this paper is organized as follows. We first review important related works in parallel I/O optimization in Section 2. Section 3 introduces the idea and the design of the layout-aware parallel I/O strategy, including both independent I/O and collective I/O. Section 4 presents the experimental results of the proposed strategy. Section 5 concludes this study and discusses potential future work.

2. Related Work

While a layout-aware parallel I/O optimization is innovative, there are numerous studies that have focused on improving 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 Peta-bytes. Research in software optimizations can be roughly classified into the areas of runtime I/O libraries [5, 19, 21, 36, 40, 41], parallel file systems [7, 10, 30, 37, 44], caching, prefetching and data distribution strategies [1, 9, 12, 15, 16, 20, 22–24, 26, 27, 29, 31, 34, 38, 39, 44].

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

There has been a significant amount of research effort in optimizing I/O performance using runtime libraries, such as collective I/O [19, 40], two-phase I/O [5], extended two-phase I/O [41], data sieving [40], server-directed I/O [36] and disk-directed I/O [21]. These strategies collect and merge small requests into larger requests at the I/O client/middleware/server level.

Parallel file systems, such as Lustre [10], GPFS [37], PanFS [30], PVFS [7], and PPFS2 [44], 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, but perform relatively poorly on other less-ideal access patterns. PPFS2 [44] offers better runtime optimization compared to other file systems. However, the optimizing techniques in parallel file systems lack aggressiveness in reading, writing, and moving data around fast enough to avoid severe performance bottlenecks.

2.2. Parallel I/O Caching and Prefetching

Many research efforts have been devoted at caching optimizations for parallel I/O, such as collective buffering [31], active buffering [29], and collective caching [26]. Patterson et al. [34], Kuenning et al. [24], Griffioen and Appleton [15] proposed prefetching strategies using compiler, runtime, and access pattern information.

Various pattern-based file prefetching methods have been proposed [1, 16, 27, 44] 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. Prediction algorithms, such as One-Block-Lookahead (OBL) prefetching, sequential prefetching [12], stride prefetching[14], Markov prefetching [20], and distance prefetching [22], have been proposed for identifying patterns in memory accesses. Most of these algorithms can be applied to the prediction in parallel I/O domain too. Speculative prefetching methods include Chang and Gibson's SpecHint [11], Patterson et. al.'s informed prefetching TIP [34] and Yang et al.'s AASFP approach [45]. PPFS2 [44] offers runtime optimization for caching, prefetching, data distribution, and sharing. 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].

2.3. 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 [10], GPFS [37] 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 [23, 38] have been developed for accessing data in a way that improves disk access parallelism by modifying application code. The problem is that these strategies are not transparent to developers and often introduce extra programming burden. Library-level optimizations [4, 40, 42], such as data sieving and two-phase I/O, are helpful in reducing the number of requests to the file system. 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]. There also exist numerous studies that utilize data layout information to optimize caching and prefetching strategies in sequential I/O research domain, such as Diskseen prefetching [13] and DULO buffer cache management scheme [18]. While many optimizations exist, there lacks sufficient study that investigates a better integration of parallel I/O middleware and parallel file systems, two major components of parallel I/O systems. In this study, we demonstrate the potential and real benefits of a better integration, layout-aware parallel I/O strategy, with the goal of improving disk access performance and task parallelism for reducing overall execution time of applications.

3. Layout-Aware Parallel I/O Design and Methodology

In this section, we present the idea and the prototype design of the data layout aware parallel I/O strategy. We first briefly review the independent I/O and collective I/O, two major forms of I/O that are used in parallel applications. We assume parallel applications are written in MPI (Message Passing Interface) in this study. We then introduce the proposed strategy that fosters a better integration of layout awareness to parallel I/O to explore physical locality and reduce contention better.

3.1. Independent I/O and Collective I/O

Independent I/O is a straightforward form of I/O and is widely used in parallel applications. This form of I/O can be called independently by an individual process or any subset of processes of a parallel application. It is different from the case that all processes within the communicator that is associated with the file handle carry out I/O requests together. In the context of this study, the independent I/O refers to the MPI-IO non-collective function family. The advantage of independent I/O is that users have the freedom to perform I/O for each individual process or any subset of the processes that open the file. However, the disadvantage is that the implementation has no idea of what other processes might do and therefore have to service the I/O requests of each process individually. This shortcoming loses the opportunity that optimizes the I/O performance with the knowledge of other processes. The independent I/O can be implemented directly with I/O system calls depending on specific file systems.

For many parallel applications, even though each process may access several non-contiguous portions of a file, the requests of multiple processes are often interleaved and may constitute a large contiguous portion of a file together [40]. In order to achieve better I/O performance, a group of processes may cooperate with each other in reading or writing data in a collective and efficient way, which is known as *collective I/O*. 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 [21], server level [36] or client level [40]. In this study, we focus on parallel I/O middleware level. The collective I/O has been well implemented in the most popular MPI-IO middleware implementation, ROMIO [40]. 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.

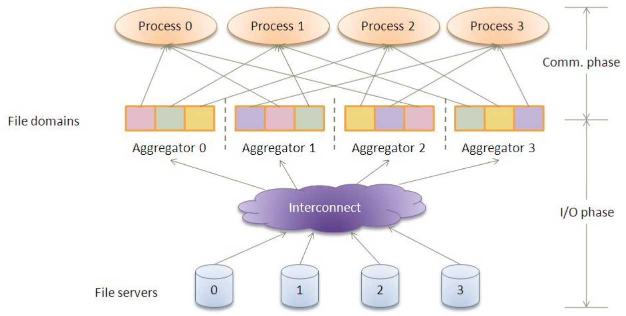


Figure 2. Collective I/O and Two-phase Implementation. The two-phase implementation has a communication phase to exchange data between I/O client processes and aggregators and an I/O phase to carry out I/O requests via aggregators collectively.

The most popular method of implementing collective I/O is a two-phase strategy [35] (and its extension - a generalized two-phase I/O [41]). This strategy carries out collective I/O with separated I/O phase and data exchange phase (or communication phase). Fig. 2 shows an example of two-phase collective I/O read. In this example, we assume all processes participate in the I/O phase (the processes participating I/O phase are termed as aggregators and the number of aggregators can be specified by users) and each process (also aggregator) has sufficient memory for temporary buffer. The two-phase I/O implementation has a first-round communication to let each aggregator knows 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.

3.2. Independent I/O with Layout Awareness

Although the independent I/O is simple and straightforward, the current strategy is not optimal. Due to the independent nature, the existing independent I/O strategy in parallel I/O merely utilizes the underlying file system calls without the consideration of other processes. However, in this study, we argue that the ignorance of other processes could destroy the locality of requests from each individual process. The reason is that all these processes compete with each other to be serviced, which essentially damages the locality and hurts the overall performance, as we know that the request of a specific process usually has strong locality (i.e., the principle of locality). Without considering the accesses from other processes for the independent I/O, we not only lose the potential benefit of collective I/O, but also deteriorate the I/O performance if without considering the data layout and locality well.

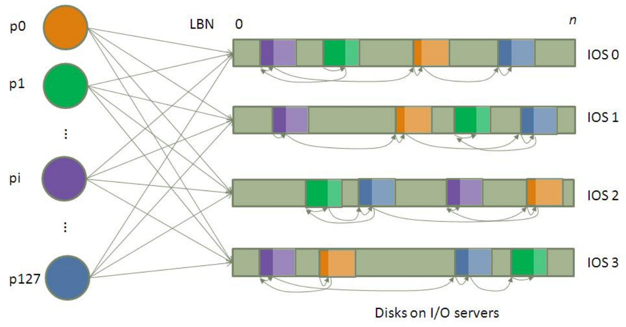


Figure 3. Independent I/O from Parallel Applications. Each I/O client process carries out I/O requests independently without considering the requests of other processes. However, from the data layout point of view, this ignorance of other processes could destroy the access locality of each individual process due to contention.

Fig. 3 demonstrates a scenario where 128 processes of a parallel application issue independent I/O requests simultaneously to four I/O servers. This scenario is quite common in real applications, such as in application-level checkpointing/restart. The result of such a parallel I/O strategy is that all processes compete with each other and destroy the locality of each process's request. For instance, it could happen that one I/O server (IOS#0) serves the partial request from P1 first, then interrupted by Pi and serve the request from Pi partially, followed by the interruption and serving the request from P0 and P127 partially too (note that the requests from other processes are omitted here). Similarly, it could happen to other processes too that the requests from these processes compete with each other and are serviced without considering physical layout. Such an existing form of independent I/O of parallel applications can lose the benefit of physical data locality considerably.

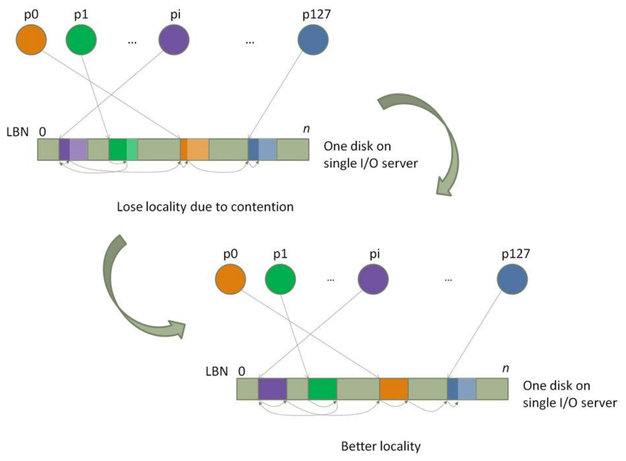

p0
p1
pi
p127
LBN 0
n
One disk on single I/O server
Lose locality due to contention
p0
p1
pi
p127
LBN 0
n
One disk on single I/O server
Better locality

Figure 4. Layout-Aware Independent I/O from Parallel Applications. With data layout awareness, the request from a specific I/O client process is serviced continuously in whole instead of being interrupted randomly by other processes. The layout-aware independent I/O exploits better locality, reduces the performance loss due to the access contention and thus improves the I/O performance.

We propose to consider physical layout for the existing parallel I/O strategy, as shown in Fig. 4. In this new form, the request from a specific process is serviced continuously in whole, instead of interrupted randomly by other processes. The advantage of this newly optimized strategy is that it considers the physical locality and avoids the interruption caused by the contention from other processes. Though this strategy does not combine and optimize accesses with other processes as the collective I/O does, it avoids and reduces the performance loss due to the contention of other processes. Thus it can improve the performance effectively too as the experiments demonstrate. There might exist two concerns for this new strategy. The first concern is that this approach may result in serialized accesses to I/O servers. The second concern is that this strategy may result in imbalanced response time for different processes (or fairness concern). Note that the total response time, or the time-to-solution, is what users really care for a parallel application. As the experimental testing shows, the total execution time of a parallel application can be considerably improved with the new layout-aware strategy, even though the response time for individual processes are not well balanced. In addition, such an optimization strategy is devised from the perspective of the data layout on each single disk. It also works in an environment where each I/O server has multiple disks. In this case, the new strategy can still benefit from the parallelism provided by multiple disks within a server or across servers. Furthermore, in order to reduce the imbalance and avoid I/O serialization, we decouple the network communication and I/O operations. As shown in the experiments , the decoupled approach can achieve an even better result together with layout-aware optimization.

3.3. Collective I/O with Layout Awareness

Collective I/O combines the requests and issues the aggregated request via aggregators on behalf of all processes. In addition to applying the same idea in independent I/O into aggregators to exploit better locality and reduce contention, we apply an additional dimension of layout awareness optimization in collective I/O when the file domains are partitioned and the requests are carried out.

As we have discussed in previous sections, a separated design of parallel I/O middleware and parallel file systems makes the current collective I/O strategy lacking the physical data layout information. Specifically, the calculation of the span of the combined I/O requests and the creation of the file domains are based on logical partitions, not physical layout that actually determines data access latency. Even though the file domain that each aggregator is responsible for carrying out I/O requests is logically contiguous, it does not guarantee physically continuity. The ignorance of the physical layout is subject to limit the performance improvement space of collective I/O. We demonstrate that it would be beneficial by the incorporation of layout awareness into collective I/O. We propose to integrate the physical layout information of data distribution among servers and rearrange file domain's partition and the requests from aggregators in a fashion that matches the physical layout on servers.

Fig. 5 illustrates the idea of such a collective I/O strategy with layout awareness. In Fig. 5, we illustrate the details of the existing collective I/O operation example and the physical layout of requested data (distinguished by the logical block number, i.e. LB#) on file servers. The file server number, i.e. S#, represents which file server the requested data reside on. Since the proposed strategy focuses on optimizing the I/O phase of the two-phase I/O strategy, we omit the details of communication phase here. The data layout strategy of file servers is assumed to be the most common round-robin mechanism. The proposed new collective I/O strategy rearranges the partitions of file domains and the requests of aggregators in a way that the requests are physically contiguous as much as possible, not only logically contiguous, as shown in Fig. 5. This example demonstrates that we rearrange requests of aggregators to have each aggregator accesses data on file servers contiguously, and multiple aggregators can access file servers concurrently. The data layout information can usually be obtained from the API provided by the underlying parallel file systems. For instance, PVFS2 provides the interface to inquire the data layout on file servers [7][32]. Note that the rearrangement here is to change the requests that each aggregator carries out on behalf of the processes, or the way the aggregators access data. The rearrangement does not exchange data themselves among aggregators. The communication overhead involved in the proposed design is low, not as exchanging data among aggregators, especially for large I/O requests.

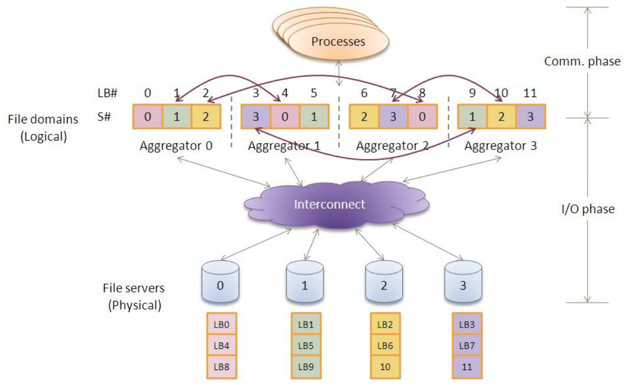


Figure 5. Collective I/O with Layout Awareness. This new strategy rearranges the partitions of file domains and the requests of aggregators to be physically contiguous as much as possible. This strategy exploits better locality, reduces contention and achieves better I/O performance.

4. Experimental Results and Analysis

We have performed experimental tests with the proposed layout-aware parallel I/O strategy for both independent I/O and collective I/O operations with several benchmarks and one user-level checkpointing/restart application. We first briefly describe the experimental environment and then present the experimental testing results.

4.1. 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. In addition to the ethernet interconnection, a subset of 16 client nodes and the head node are also connected with 10Gigabit InfiniBand interconnection. The experiments were tested on MPICH2-1.0.5p3 release and PVFS 2.8.1 file system. We varied PVFS2 system configurations to test the performance under different scenarios.

4.2. Experimental Results and Analysis of Independent I/O with Layout Awareness

The first set of experiments is to compare the performance of layout-aware independent I/O with the existing independent I/O strategy. In order to better understand the performance improvement of the proposed strategy, we distinguish two different cases of the layout-aware strategy. The first case is layout-aware reordering optimization only, which means we only apply layout awareness to all processes but do not decouple communication and I/O. The second case is a complete optimization strategy with both layout-aware optimization and communication and I/O decoupling. In this set of experiments, we tested on the InfiniBand subset cluster. We configured PVFS2 system with four I/O server nodes and eight client nodes.

Fig. 6 demonstrates the results of the layout-aware strategy and the normal parallel I/O strategy for a user-level checkpointing application. In this application, all processes follow a coordinated checkpointing protocol by reaching a global consistent state first, then issue application-level checkpointing by writing the runtime data into persistent data storage. We vary the number of client processes to test the performance under different scenarios, but keep the same total image size of the whole application in all cases. For instance, the first group of three bars in the figure represents the case with 8 processes and each process writes an image of 2000MB. As can be observed from the figure, even though the total amount of I/O requests is the same for all cases, the execution time was increased when the number of processes increased. This time increase is primarily due to the contention from multiple processes and the degraded locality because of the contention. With the complete layout-aware reordering and communication and I/O decoupling, the execution time was decreased by 35.2% on average. In this case, the communication and I/O decoupling contributed considerably to the overall performance, while the strategy with only layout-aware reordering decreased the execution time by 7.61% on average. Furthermore, we can observe that the complete proposed optimization strategy can achieve stable performance under various cases, which clearly demonstrates that the layout-aware optimization exploits better physical locality and reduces contention considerably. In addition, we can observe that the benefit of the optimized strategy increased as the system size scaled up. For instance, the performance speedup with the complete optimization strategy was 24.3%, 33.2%, 38.9%, 38.4% and 41.2% respectively when we tested with 8, 16, 32, 64 and 128 processes. This is a great merit of the proposed strategy, as it is scalable and can achieve better performance speedup when the system size is increased.

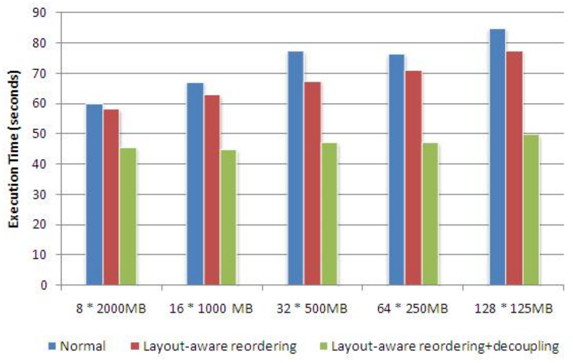


Figure 6. Layout-Aware Independent I/O Optimization Result on PVFS2

Fig. 7 demonstrates the results of the similar test as in the previous case, but on NFS file system. The performance trend of the test on NFS file system is not as stable as in the PVFS2 test case, but we can still observe a clear performance improvement with the proposed layout-aware optimization strategy. Although the communication and I/O decoupling seems contributed little in this series of tests, the overall performance was still improved by 28.6% on average. The layout-aware reordering alone achieved roughly 19.4% performance speedup on average.

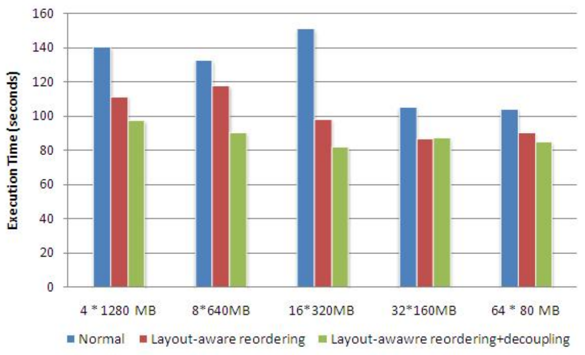


Figure 7. Layout-Aware Independent I/O Optimization Result on NFS

4.3. Experimental Results and Analysis of Collective I/O with Layout Awareness

Next we present the experimental results of the proposed layout-aware optimization in collective I/O. The tests were carried out with one synthetic benchmark that tested the performance of collective I/O and the IOR collective I/O testing. In this set of experimental tests, we varied the PVFS2 settings. We configured PVFS2 with 32 I/O server nodes. The rest of 32 nodes were used as client nodes.

4.3.1. Synthetic Benchmark. We have coded a synthetic benchmark in which each process does strided reads but 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 layout-aware collective I/O and the original one. The total size of the data accessed by all processes are 128MB, 320MB, 640MB, 800MB and 4000MB respectively. The results are shown in Fig. 8 with both current collective I/O strategy and the optimized strategy with layout awareness. It can be observed that the optimization strategy with layout awareness could have a considerable impact on the performance of parallel I/O system. The performance variation and the performance improvement was up to 48.8% and 38% on average.

4.3.2. IOR Benchmark. Fig. 9 reports 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). We performed both sequential reads/writes and random reads/writes tests, and varied the file size. As can be seen from these results, the layout-aware strategy can affect the IOR benchmark testing performance

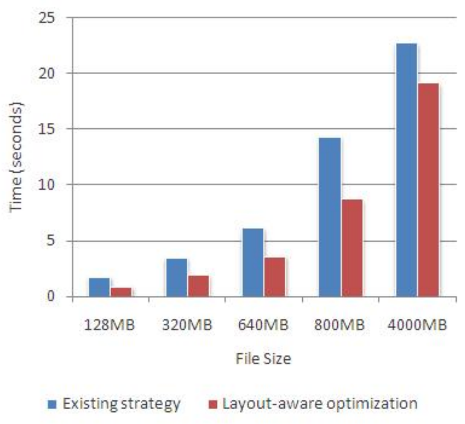
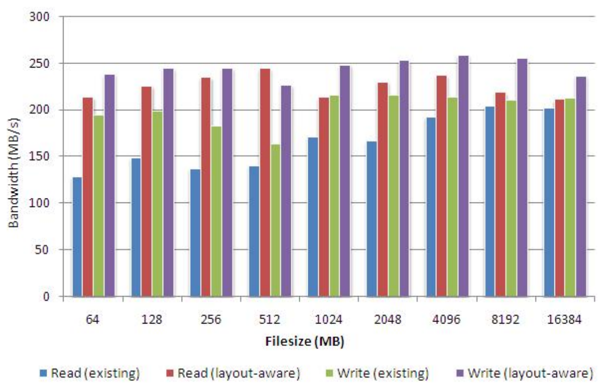
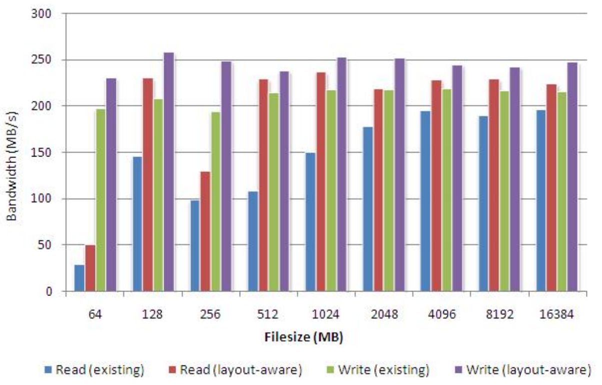


Figure 8. Layout-Aware Collective I/O Optimization with Synthetic Benchmark Testing



(a) Random Accesses



(b) Sequential accesses

Figure 9. Layout-aware Collective I/O Optimization with IOR Benchmark Testing

considerably. The layout-aware strategy could improve the I/O bandwidth up to 74%, 38%, 112% and 28% for random reads, random writes, sequential reads and sequential writes, respectively. The average potential improvement of layout-aware strategy with different file sizes was 40%, 23%, 45% and 16% for random reads, random writes, sequential reads and sequential writes respectively.

5. Conclusion and Future Work

Poor I/O performance has been a bottleneck in many parallel computing systems and data-intensive high-end/high-performance computing applications. In this study, we propose a new layout-aware I/O strategy to optimize parallel I/O performance and foster a better integration of parallel I/O middleware (independent I/O and collective I/O) and parallel file systems (data layout information). 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 parallel I/O strategy 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 could be beneficial to integrate layout awareness to parallel I/O strategy. Second, we propose a new layout-aware parallel I/O optimization strategy and exploit this optimization strategy for both independent I/O and collective I/O. Third, as the experimental results demonstrate, the layout-aware optimization can clearly improve the parallel I/O system performance. The proposed optimization strategy can exploit physical data locality and reduce contention better than the existing parallel I/O strategy.

Parallel I/O systems have been designed as one-set-for-all and have been static. There is a great need for research into next-generation parallel I/O architectures to support data layout awareness, applications' access characteristics and intelligence. Although our current research effort is just one step toward the goal of an intelligent next-generation I/O architecture, our prototyping system has demonstrated the great potential in improving parallel I/O access performance via layout awareness optimization. In the near future, we plan to continue our current research investigation, especially the investigation on layout awareness and access awareness optimizations, to further improve parallel I/O system performance.

Acknowledgment

The authors are thankful to Dr. Philip C. Roth of Oak Ridge National Laboratory for his helpful suggestions. The authors are also grateful to anonymous reviewers for their constructive comments and suggestions that help the further improvement of this work.

References

[1] Ahmed Amer, Darell Long, Jehan-Francios Paris, and Randal Burns, File Access Prediction with Adjustable Accuracy, International Performance Conference on Computers and Communication, Phoenix, AZ, April 2002.

[2] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, M. Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. in Proc. of ACM/IEEE Supercomputing'09, 2009.

[3] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, W. Gropp. Parallel I/O Prefetching Using MPI File Caching and I/O Signatures, in Proc. of the ACM/IEEE SuperComputing Conference (SC'08), Nov. 2008.

[4] Javier Garcła Blas, Florin Isaila, David E. Singh and Jess Carretero, View-based Collective I/O for MPI-IO, IEEE International Symposium on Cluster Computing and the Grid (CCGRID). Lyon, France. 2008.

[5] Rajesh Bordawekar, Juan Miguel del Rosario, Alok N. Choudhary: Design and Evaluation of primitives for Parallel I/O. SC 1993: 452-461

[6] R. E. Bryant and D. O'Hallaron. Computer Systems: A Programmer's Perspective. Prentice-Hall, 2003.

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

[8] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, W. Gropp. Hiding I/O Latency with Pre-execution Prefetching for Parallel Applications, in Proc. of the ACM/IEEE SuperComputing Conference (SC'08), Nov. 2008.

[9] Y. Chen, X.-H. Sun and M. Wu. Algorithm-System Scalability of Heterogeneous Computing. Journal of Parallel and Distributed Computing (JPDC), Vol. 68, No. 11, 1403-1412, 2008.

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

[11] F. Chang and G. A. Gibson. Automatic I/O Hint Generation Through Speculative Execution, Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI), 1999.

[12] F. Dahlgren, M. Dubois, and P. Stenstrom, Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors, Proc. 1993 Int'l Conf. Parallel Processing, CRC Press, 1993, pp. I56-I63

[13] X. Ding, S. Jiang, F. Chen, K. Davis, X. Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch, in Proceedings of USENIX Annual Technical Conference 2007, pp. 261-274, 2007.

[14] J. Fu, J.H. Patel, Data Prefetching in Multiprocessor Vector Cache Memories, Proc. 17th annual Int'l Symposium, Computer Architecture, pp. 54-63, May 1991.

[15] J. Griffioen and R. Appleton, Performance Measurements of Automatic Prefetching, In Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems, pages 237-242. IASTED, Sept. 1995.

[16] T. Highley and P. Reynolds, Marginal Cost-Benefit Analysis for Predictive File Prefetching, Proceedings of the 41st Annual ACM Southeast Conference (ACMSE 2003), Savannah, GA.

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

[18] S. Jiang, X. Ding, F. Chen, E. Tan, X. Zhang. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities, in Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST), 2005.

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

[20] D.Joseph and D. Grunwald. Prefetching Using Markov Predictors, Proceedings of the 24th Annual Symposium on Computer Architecture, Denver-Colorado, pp 252-263, June 2-4 1997.

[21] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. ACM Transactions on Computer Systems, 15(1):41C74, 1997.

[22] Gokul Kandiraju, Anand Sivasubramaniam, Going the Distance for TLB Prefetching: An Application-Driven Study, In Proceedings of the ISCA '02, 2002.

[23] Meenakshi A. Kandaswamy , Mahmut Kandemir , Alok Choudhary , David Bernholdt, An Experimental Evaluation of I/O Optimizations on Different Applications, IEEE Transactions on Parallel and Distributed Systems, v.13 n.12, p.1303-1319, December 2002.

[24] G. H. Kuenning, The Design of the SEER Predictive Caching System, In Proceedings IEEE Workshop on Mobile Computing Systems and Applications, 1994, pages 37-43, IEEE, 1994.

[25] 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.

[26] Wei-keng Liao, Avery Ching, Kenin Coloma, Alok N. Choudhary, Lee Ward, An Implementation and Evaluation of Client-Side File Caching for MPI-IO, IPDPS 2007: 1-10

[27] H. Lei and D. Duchamp, An Analytical Approach to File Prefetching, In Proceedings of the 1997 USENIX Annual Technical Conference, pages 275-288. USENIX, Jan. 1997.

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

[29] Xiaosong Ma, Marianne Winslett, Jonghyun Lee, Shengke Yu, Faster Collective Output through Active Buffering, IPDPS 2002.

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

[31] B. Nitzberg, Virginia Mary Lo, Collective Buffering: Improving Parallel I/O Performance, HPDC 1997

[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] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, Informed Prefetching and Caching, In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95), ACM, 1995.

[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 93, 1993.

[36] K. Seamons, Y. Chen, P. Jones, J. Jozwiak and M. Winslett, Server-Directed Collective I/O in Panda, in Proc. of Supercomputing95. ACM Press, 1995.

[37] F. Schmuck and R. Haskin, GPFS: A Shared-Disk File System for Large Computing Clusters, In First USENIX Conference on File and Storage Technologies, pages 231–244. USENIX, Jan. 2002.

[38] S. W. Son, G. Chen, M. Kandemir, Disk Layout Optimization for Reducing Energy Consumption, Proceedings of the 19th annual international conference on Supercomputing, June 20-22, 2005.

[39] X.-H. Sun, Y. Chen and Y. Yin. Data Layout Optimization for Petascale File Systems. In Proc. of The 4th Petascale Data Storage Workshop (in conjunction with ACM/IEEE Supercomputing'09), 2009.

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

[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, Winter 1996.

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

[43] Top 500 Supercomputing Website. http://www.top500.org.

[44] N. Tran, D. A. Reed. Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching, IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 4, pp. 362-377, April, 2004.

[45] C.K. Yang, T. Mitra and T. Chiueh, A Decoupled Architecture for Application-Specific File Prefetching, Freenix Track of USENIX 2002 Annual Conference, 2002.