Collective Prefetching for Parallel I/O Systems

Yong Chen
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA
cheny@ornl.gov

Philip C. Roth
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA
rothpc@ornl.gov

***Abstract*—Data prefetching can be beneficial for improving parallel I/O system performance, but the amount of benefit depends on how efficiently and swiftly prefetches can be done. In this study, we propose a new prefetching strategy, called *collective prefetching*. The idea is to exploit the correlation among I/O accesses of multiple processes of a parallel application and carry out prefetches collectively, instead of the traditional strategy of carrying out prefetches by each process individually. The rationale behind this new collective prefetching strategy is that the concurrent processes of the same parallel application have strong correlation with respect to their I/O requests. We present the idea, initial design and implementation of the new collective prefetching strategy in this study. The preliminary experimental results show that this new collective prefetching strategy holds promise for improving parallel I/O performance.**

Keywords-parallel I/O; prefetching; collective prefetching; MPI-IO; middleware; parallel file systems; storage; performance; exascale computing; high-performance computing

I. MOTIVATION

High-performance computing (HPC) has crossed the Petaflop mark and is reaching for the Exaflop range [17]. Although computing resources are making rapid progress, there is a significant gap between processing capacity and data-access performance. Due to this gap, available computing devices often have to stay idle waiting for data to arrive, which leads to a severe overall performance degradation. Figure 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 [14]. The computational performance improvement rate is magnitudes higher than the bandwidth improvement rate of disk drives. 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. In order to match the rapid advance of processor architectures and the fast increasing scale of computational capability, parallel I/O is essential to address this problem. Many high-performance computing applications and scientific simulations in critical areas of research, such as nanotechnology, astrophysics, climate, and high energy physics, are becoming more and more data intensive [14]. These applications contain a large number of I/O accesses, where large amounts of data are stored to and retrieved from disks. They 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 and in investigating novel and intelligent solutions such as data prefetching.

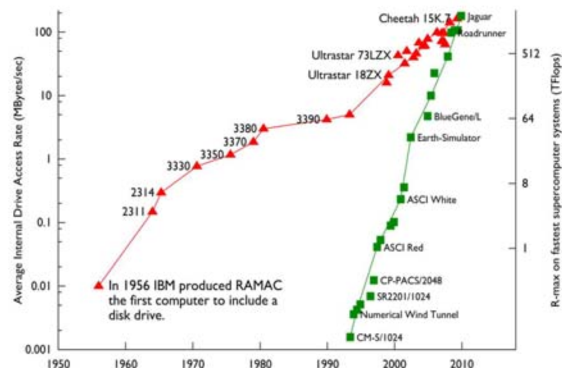


Figure 1. FLOPS of Supercomputers v.s. Single Disk Drive Bandwidth

The fundamental idea of data prefetching is to observe data access patterns, and then predict future accesses and fetch that data from underlying storage device so it is available when it is needed by the computation. It is recognized as a critical and promising technique that improves parallel I/O access performance for many applications [2][3][11][12][13]. Many scientific applications can benefit from prefetching because such applications have been shown that they access structured data (such as two-dimensional/three-dimensional array of single-precision/double-precision floating point data) with predictable and regular patterns. The data are accessed regularly and periodically for processing, and the processed data are written into storage. In these applications, regular patterns of I/O accesses can be identified and I/O prefetching is effective in speeding up parallel I/O performance. However, the effectiveness of parallel I/O prefetching depends on carrying out prefetches efficiently and moving data swiftly. The current I/O prefetching strategy uses an independent approach, where each process of a parallel application issues prefetches independently to move the required data in advance. We term this form of I/O prefetching as an *independent prefetching* strategy. In this study, we propose a new form of

This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed 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.

I/O prefetching strategy, called *collective prefetching*. The rationale for collective prefetching is that the processes of many parallel applications have a strong correlation with each other with respect to I/O accesses. This correlation has been used to optimize parallel I/O performance in many strategies, such as collective I/O [16], one of the most critical performance optimization strategies for parallel I/O systems, data sieving, etc. We argue that taking advantage of this correlation is beneficial for I/O prefetching too: prefetching should be done in a collective way with global awareness rather an ad hoc individual and independent way. In the mean while, the traditional concerns with prefetching strategies, such as increased memory pressure, buffer cache pollution and increased communication congestion, have been remedied well by new technologies such as much larger memory at low cost, dedicated memory portions for buffer cache, and higher bandwidth and disk-level buffer cache. In this paper, we introduce a collective prefetching framework to the parallel I/O system. We illustrate the design of MPI-IO with collective prefetching functionality, and present the implementation strategy. Initial experimentation has shown the potential benefit of the collective prefetching. The primary goal of this research is to bring intelligent prefetching strategies to parallel I/O systems to improve the I/O performance for high-performance computing.

II. COLLECTIVE PREFETCHING FRAMEWORK

The fundamental idea of the proposed collective prefetching is to take advantage of the correlation among I/O accesses of multiple processes of the same parallel application and to optimize prefetching in a collective and global-aware way. The potential benefits of collective prefetching are three fold.

- Collective prefetching can filter overlapping and redundant prefetch requests from multiple processes. As the system size increases, the likelihood of overlapping and redundant prefetch requests increases, especially when we consider petascale/exascale systems. These overlapping and redundant requests considerably waste limited I/O bandwidth. Filtering out redundant prefetch requests helps alleviate the bottleneck due to limited I/O bandwidth.
- For many parallel applications, each process accesses data in a non-contiguous fashion in each iteration. However, when combining the demand I/O request in one iteration with the prefetch requests from future iterations, the aggregated request often comprises a contiguous data region. Furthermore, when combining the prefetch requests from multiple processes with the collective prefetching strategy, we can explore the possibility of contiguous data region across the entire application, making parallel I/O prefetching more efficient.
- As with collective I/O, the collective prefetching strategy can reduce the number of parallel file system calls by combining small and noncontiguous requests from the same application iteration into large and contiguous ones. Furthermore, with collective prefetching, we can combine the prefetch requests with demand requests to improve the parallel I/O prefetching efficiency. The reduced number of system calls can decrease the system call overhead.

Many parallel applications exhibit strong correlation among the I/O accesses of multiple processes [1][8][9][14][16], and this correlation has been well exploited for collective I/O design [16]. For instance, many parallel applications have processes that access data with the same and constant stride. Another representative example is that many parallel processes access the border of data array/matrix in an overlapping way and redundantly. In these scenarios, the proposed collective prefetching can be of great potential in carrying out parallel I/O prefetching in a better and more efficient way.

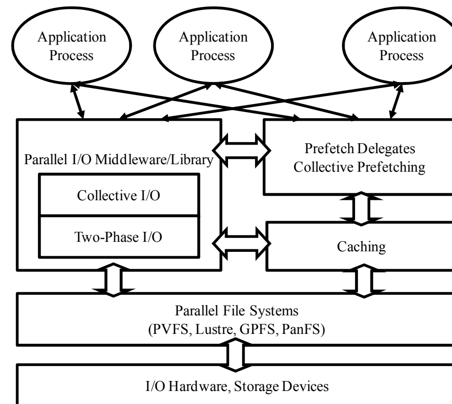


Figure 2. Collective Prefetching Framework

Figure 2 illustrates the high-level structure of the proposed collective prefetching framework. We introduce *prefetch delegates* that explore correlations among prefetch requests. The prefetch delegates find large contiguous data to prefetch by combining multiple prefetch requests and carry out prefetches collectively. For merging of multiple requests, we use the correlation to identify overlapped data accesses. With the global awareness brought by collective prefetching, we can also detect redundant prefetches among multiple processes, and utilize the precious bandwidth wisely. The implementation methodology of prefetch delegates is similar to that of aggregators in a collective I/O implementation, such as ROMIO [16]. The collective prefetching component directly interacts with the parallel I/O middleware/library to merge and filter prefetch requests, and interacts with the caching component to bring the prefetched data into the caching component. The regular parallel I/O library, where optimizations like collective I/O happen, interacts with the caching component to take advantage of the prefetched data in the cache buffer. If the requested data are not in the cache buffer, the parallel I/O library still requests the data via the underlying parallel file system. The caching component also interacts with the parallel file system to fetch data into the cache buffer. The parallel file system manages data on physical storage devices and provides data access to upper layer parallel I/O library and caching component via file system calls.

III. MPI-IO WITH COLLECTIVE PREFETCHING

In this section, we present the design and implementation methodology of providing collective prefetching at MPI-IO. We first briefly review MPI-IO, the ROMIO implementation of MPI-IO, and collective I/O optimizations in this section since collective prefetching at the MPI-IO layer is built upon them. We then introduce the implementation of collective prefetching in ROMIO at the MPI-IO layer.

A. MPI-IO, Collective I/O and ROMIO

MPI-IO defines an I/O access interface for parallel applications and is a subset of the MPI-2 specification [7]. 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. The implementation of MPI-IO usually uses many features of MPI. ROMIO is a popular MPI-IO implementation [16]. It provides an abstract-device interface called ADIO 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 [16].

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. This shortcoming makes the MPI-IO middleware loses the opportunity for optimizing I/O performance with the knowledge of multiple parallel processes. 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.

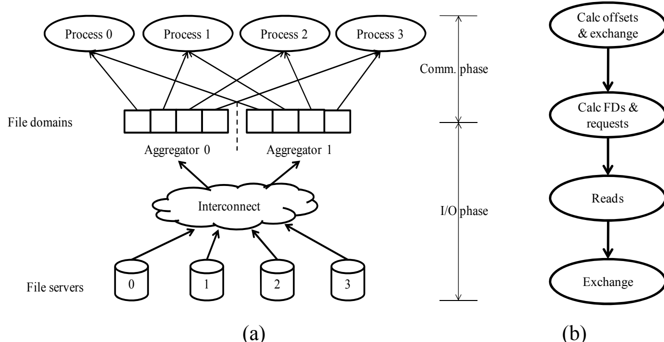


Figure 3. (a) Collective I/O and Two-phase Implementation. (b) Two-phase Read Protocol in ROMIO. The protocol consists of four steps: 1) each aggregator calculates the I/O requests span and exchange; 2) partitions the aggregated span into file domains; 3) each aggregator carries out I/O requests for its own file domain; and 4) all aggregators send data to the requesting processes, and each process receives its required data from corresponding aggregators that fetch the data on behalf of it.

The most popular method of implementing collective I/O is a two-phase strategy (and its extension - generalized two-phase I/O [16]). This strategy separates the servicing of an I/O request into an I/O phase and data exchange phase (or communication phase). Figure 3 (a) shows the strategy of a two-phase collective I/O read for four processes, where two processes participate in the I/O phase (aggregators). The two-phase I/O implementation has 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 (FD) 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. Figure 3 (b) illustrates the collective I/O two-phase protocol in the ROMIO implementation.

B. MPI-IO with Collective Prefetching

Figure 4 illustrates the current design and prototype of collective prefetching in MPI-IO based on the existing collective I/O mechanism that is available in ROMIO and its internal implementation, ADIO. To simplify the design and implementation, we currently constrain prefetch delegates of collective prefetching to be same as aggregators in the existing collective I/O mechanism. Users can configure the number of aggregators/prefetch delegates (APD) and specify which processes to be the APD with user supplied hints. By default, all processes are aggregators/prefetch delegates.

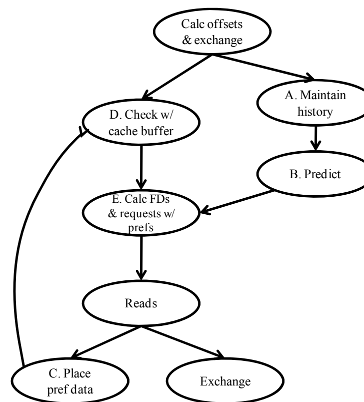


Figure 4. An Extended Collective I/O Two-phase Protocol in ROMIO for Collective Prefetching. The protocol is extended with four more steps: A. Maintain history; B. Predict; C. Place prefetched data; D. Check requests with cache buffer, and one revised step, E. Calculate file domains and requests together with prefetch requests, to carry out collective prefetching.

To provide collective prefetching at the MPI-IO layer, the two-phase protocol is extended with four more steps and revised within one step. The four new steps are:

A) All APDs maintain the past I/O request history. Currently, a history window of 8 is kept, which means the 8 most recent I/O requests (offset and length lists) are kept in memory. The history window size can be tunable and can also be made configurable with user hints.

B) All APDs generate prefetch requests based on predictions. In theory, any prediction, machine learning and data mining algorithm can be used here for generating prefetches. To simplify the initial experimentation and to verify the framework, we currently only provide a simple streaming and strided prediction algorithm. These patterns are common patterns observed in many scientific applications.

C) All APDs place prefetched data in cache buffer. With collective prefetching, the extended two-phase protocol separates the fetched data into two categories: demanded data and prefetched data. The demanded data are used to satisfy the demand requests; such data is moved to the user supplied buffer space when issuing the I/O function call. The prefetched data are kept in an internal cache buffer to satisfy future requests. We use collective caching proposed by Liao et al. [8] as the internal cache buffer. This code is implemented at ADIO layer within the ROMIO and maintains a global buffer cache among multiple processes at the client side. Each client

contributes part of its memory to construct the global cache pool. We customize the collective caching for APDs instead of for all client processes.

D) When calculating spanned request and exchanging data, the request is checked against the cache buffer. The data residing in the cache buffer are used to service I/O requests directly.

E) The revised step is that when the file domain is partitioned and calculated, the prefetch requests are combined with demand requests to carry out collectively. In this step, the overlapping and redundant prefetch requests are detected and filtered. In addition, the prefetch requests of multiple prefetch delegates are combined and partitioned together with demand requests to form large contiguous accesses to improve the prefetching efficiency.

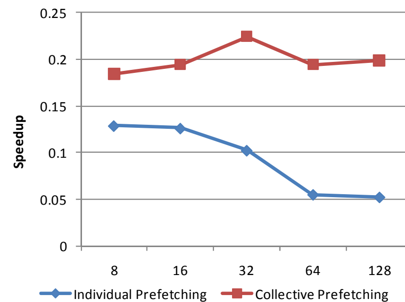
```
Algorithm cpf /* Collective Prefetching at MPI-IO */
Input: I/O request offset list, I/O request length list
Output: none
Begin
 1.   Each aggregator maintains recent access history of
      window size w
 2.   Aggregators/prefetch delegates run prediction or
      mining algorithms on all tracked global access
      history
              a.   Algorithms can be as streaming, strided,
                   Markov, or advanced mining algorithms
                   such as PCA/ANN
 3.   Generate prefetch requests and enqueue them in PFQ
 4.   Process requests in PFQs together with demand
      accesses
 5.   Filter out overlapping and redundant requests
 6.   Perform extended two-phase I/O protocol with
      prefetch requests
              a.   Prefetched data are kept in cache buffer
                   to satisfy future requests
              b.   Exchange data to satisfy demand requests
                   (move data to user buffer)
End
```

Figure 5. Collective Prefetching Algorithm with An Extended Collective I/O Two-phase Protocol

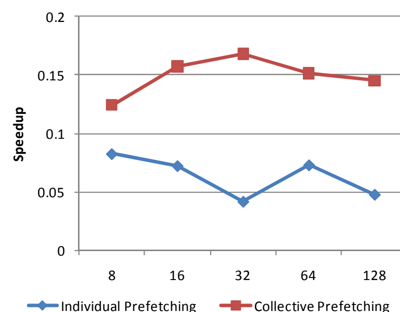
Figure 5 explains the algorithm and the flow of collective prefetching at the MPI-IO layer. In the implementation, a prefetch queue (PFQ) is maintained for each prefetch delegate. This PFQ accommodates prefetch requests and is used to combine them with demand requests and to carry them out collectively. In addition, the requests are checked with the offset and length lists, and the overlapping and redundant requests are filtered out to improve prefetching efficiency and use I/O bandwidth wisely. The fetched data are either placed into the cache buffer if they are prefetched data to satisfy future requests, or are supplied to the process' requests directly if they are demanded data.

IV. PRELIMINARY EXPERIMENTAL RESULTS AND ANALYSIS

The initial experiments were tested with a revised synthetic pio-bench benchmark [15]. The revision simulates both computation and I/O access behavior of parallel applications, and the original only characterizes I/O behavior. The original benchmark is usually used for measuring the peak I/O performance with different access patterns, while the revision is suitable for studying the sustained performance and the impact of different optimization techniques. The experiments were tested on MPICH2-1.0.5p3 release and PVFS 2.8.1 file system. The number of processes varied from 8 to 128 in each test. The number of APDs was configured as 8 in all tests. The total data size accessed was fixed as 16GB in each test.



(a) With 1MB stride



(b) With 4MB stride

Figure 6. Speedup with Strided Access Pattern

The experimental tests compared the sustained bandwidth with the standard MPI-IO library, collective prefetching and individual prefetching respectively. Figure 6 reports the speedup of the sustained bandwidth of collective prefetching and individual prefetching over the standard case respectively. These two figures cover two cases of the tests, the strided access pattern with 1MB and 4MB stride respectively. The sustained bandwidth was decreased as the number of processes was increased, which was found due to the reason of increased contention [4]. These tests showed that the collective strided prefetching outperformed the individual strided prefetching. The former outperformed the latter by over two fold on average. In addition, we observe that the individual strided prefetching is not as stable as collective strided prefetching – there is a decreasing trend of the individual prefetching speedup, while the collective prefetching combines prefetch requests, reduces the contention at a large scale and achieved stable speedup.

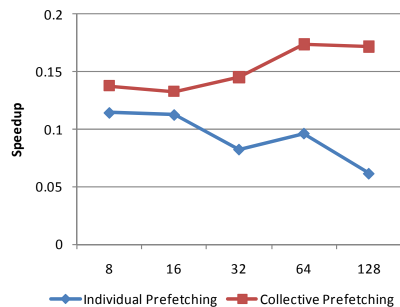


Figure 7. Speedup with Nested Strided Access Pattern

Figure 7 reports the result of one of the tests with nested strided pattern with (1MB, 3MB) stride pair. It can be observed the trend is similar to the tests with strided patterns. The collective prefetching was more effective than individual prefetching and was able to achieve more stable improvement.

V. RELATED WORK

Many research efforts have 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 [8]. Nisar et al. proposed to delegate a small set of compute nodes called I/O delegates to perform caching collectively, resolving caching coherence and reducing lock contention [10]. 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 [18]. 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 [6]. Several parallel or distributed file systems, such as PanFS [20] and Ceph [19], also provide client-side caching to improve file system performance.

Prefetching algorithms, such as One-Block-Lookahead prefetching, sequential prefetching, stride prefetching, Markov prefetching, and distance prefetching, have been widely used for identifying patterns in memory accesses. Most of these algorithms can be applied to prediction problem in the parallel I/O domain also. Patterson et al. proposed informed prefetching strategy using compiler, runtime, and access pattern information [13]. Tran et al. proposed time series modeling to provide efficient adaptive prefetching [11]. Recently, a more aggressive pre-execution based prefetching, where a prefetching thread runs ahead of main computing thread to prefetch data, was introduced [5]. A signature based prefetching with post-execution analysis and runtime adjustment was introduced by Byna et al [3]. Blas et al. proposed multiple-level caching and one-level prefetching for Blue Gene systems based on ROMIO [2]. In this study, we propose collective prefetching to exploit the correlation among multiple processes, and to explore globally coordinated prefetching for parallel I/O systems, which has not been exploited in existing literature.

VI. CONCLUSION AND FUTURE WORK

With the tremendous advances in processor architectures and the computational capability, I/O has been widely recognized as the performance bottleneck for many applications. In this research, we propose a new form of prefetching specifically for parallel applications, called collective prefetching, to improve parallel I/O prefetching efficiency and to enhance parallel I/O performance. This investigation is motivated by the fact that existing I/O prefetching strategies are not coordinated even though parallel applications have correlation in their I/O accesses. Although existing parallel I/O strategies, such as collective I/O, two-phase I/O, and data sieving, take advantage of the correlation and have been demonstrated to be beneficial in many scenarios, no studies have explored the correlation and optimization for I/O prefetching strategies. This research and the proposed collective prefetching strategy address the limitation of existing studies. It exploits the correlation among accesses from multiple processes of a parallel application and optimizes parallel I/O prefetching. This is a general idea that can be applied at many levels, such as the storage device level or server level. In this study, we focus on the middleware level, i.e. the MPI-IO level. We will continue working on the investigation of collective prefetching at MPI-IO layer. In the future, we plan to investigate the potential of collective prefetching strategy at the server and storage level as well.

VII. ACKKNOWLEDGEMENT

The authors are grateful to Prof. Xian-He Sun of Illinois Institute of Technology and Dr. Rajeev Thakur of Argonne National Laboratory for their constructive and helpful suggestions toward this study. The authors are also thankful to Prof. Wei-Keng Liao and Prof. Alok Choudary of Northwestern University for their collective caching code.

REFERENCES

[1] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, M. Wingate, "PLFS: A Checkpoint Filesystem for Parallel Applications," SC 2009.

[2] J. G. Blas, F. Isaila, J. Carretero, R. Latham, R. Ross, "Multiple-Level MPI File Write-Back and Prefetching for Blue Gene Systems", PVM/MPI 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," SC 2008.

[4] Y. Chen, X.-H. Sun, R. Thakur, H. Song and H. Jin, "Improving Parallel I/O Performance with Data Layout Awareness," Cluster 2010.

[5] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, W. Gropp, "Hiding I/O Latency with Pre-execution Prefetching for Parallel Applications," SC08.

[6] M. Eshel, R. L. Haskin, D. Hildebrand, M. Naik, F. B. Schmuck, R. Tewari, "Panache: A Parallel File System Cache for Global File Access", FAST 2010.

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

[8] W.K. Liao, A. Ching, K. Coloma, A. Choudhary and L. Ward, "An Implementation and Evaluation of Client-Side File Caching for MPIIO", IPDPS 2007.

[9] 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)," CLADE 2008.

[10] A. Nisar, W.-K. Liao, A. Choudhary. "Scaling Parallel I/O Performance through I/O Delegate and Caching System", SC 2008.

[11] N. Tran, D. A. Reed, "Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching," IEEE TPDS, 15(4), pp. 362-377, 2004.

[12] A. Papathanasiou and M. Scott, "Aggressive Prefetching: An Idea Whose Time Has Come", HotOS X, 2005.

[13] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," SOSP 1995.

[14] R. Ross, R. Latham, M. Unangst, B. Welch, "Paralell I/O in Practice", tutorial in the ACM/IEEE Supercomputing Conference (SC'09), 2009.

[15] F. Shorter, "Design and Analysis of a Performance Evaluation Standard for Parallel File Systems", Master Thesis, Clemson University. 2003.

[16] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO", Frontiers 1999.

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

[18] 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.

[19] S. A. Weil, S. A. Brandt, E. L. Miller, D. D.E. Long, C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," OSDI 2006.

[20] 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," FAST 2008.