A New Data Sieving Approach for High Performance I/O

Yin Lu[1], Yong Chen[1], Prathamesh Amritkar[1], Rajeev Thakur[2], and Yu Zhuang[1]

[1] Computer Science Department, Texas Tech University, Lubbock, TX, USA
[2] Mathematics and Computer Science Division, Argonne National Lab, Argonne, IL, USA

{yong.chen, prathamesh.amritkar, yin.lu}@ttu.edu, thakur@mcs.anl.gov, yu.zhuang@ttu.edu

Abstract. Many scientific computing applications and engineering simulations exhibit noncontiguous I/O access patterns. Data sieving is an important technique to improve the performance of noncontiguous I/O accesses by combining small and noncontiguous requests into a large and contiguous request. It has been proven effective even though more data is potentially accessed than demanded. In this study, we propose a new data sieving approach namely Performance Model Directed Data Sieving, or PMD data sieving in short. It improves the existing data sieving approach from two aspects: 1) dynamically determines when it is beneficial to perform data sieving; and 2) dynamically determines how to perform data sieving if beneficial. It improves the performance of the existing data sieving approach and reduces the memory consumption as verified by experimental results. Given the importance of supporting noncontiguous accesses effectively and reducing the memory pressure in a large-scale system, the proposed PMD data sieving approach in this research holds a promise and will have an impact on high performance I/O systems.

Keywords: Data sieving; Runtime systems; Parallel I/O; Libraries; parallel file systems; high performance computing.

1 Introduction

Many scientific computing applications and engineering simulations are highly data intensive. These applications often access a large number of small and noncontiguous chunks of data [5][9]. Even though advanced parallel file systems (such as PVFS/PVFS2 [3], Lustre [4], GPFS [15]) have been developed in recent years, and they generally provide high bandwidth for large, well-formed data streams, they often perform inadequately in dealing with a large number of small and noncontiguous data requests. Data sieving is an important technique that combines small and noncontiguous I/O requests into a large and contiguous request to reduce the effect of high I/O latency caused by a noncontiguous access pattern and many small requests [17][18]. The data sieving technique has been extensively evaluated and proven effective in optimizing small and noncontiguous I/O accesses [9][14][17][18]. The current data sieving technique, however, has two potential limitations. First, the benefit of data sieving depends on specific access patterns; nevertheless, the existing data sieving technique is rather static and lacks a dynamic decision based on different access patterns. If data sieving is enabled in the parallel I/O system, the existing technique always combines requests to form a large and contiguous request, without considering specific access patterns. Even though data sieving is beneficial in many scenarios, the ignorance of access patterns can degrade the I/O performance some times. For instance, in certain access patterns, the non-requested portion between two requested portions (also called holes) could be so large that it may not be beneficial to perform data sieving any more as the sieving may not offset the overhead.

Second, the current data sieving technique has a potential problem of extensive memory requirement [17][18]. In the existing algorithm, instead of accessing each contiguous portion of the data separately, a single contiguous chunk of data starting from the first requested byte up to the last requested byte is read into a temporary buffer in memory (in an I/O read case) [17][18]. The total temporary buffer that data is read into must be as large as the total number of bytes between the first and the last byte requested by the user. As multicore/manycore architectures become universal, the available memory capacity per core is projected to decrease in high performance computing (HPC) systems. The memory requirement of the existing data sieving could be an increasingly important issue.

The detection of beneficial cases and an intelligent, dynamic adoption of the data sieving technique based on the access pattern can both improve the parallel I/O performance and reduce the memory consumption of the data sieving technique. In this study, we revisit the data sieving technique and propose a new data sieving approach, namely Performance Model Directed Data Sieving (or PMD data sieving in short). The newly proposed approach considers the I/O access pattern at run time. It improves the performance of the existing data sieving technique and reduces the memory pressure as well. To the best of our knowledge, this work is the first attempt in developing a dynamic data sieving technique based on access patterns to improve the parallel I/O performance and reduce the memory consumption.

2 Conventional Data Sieving and Implementation

Data sieving was first used in the PASSION system to access sections of strided arrays [18]. This technique has been extended in ROMIO to handle general noncontiguous I/O accesses [14][17]. The main advantage of data sieving is that it requires very few I/O requests compared with the direct method in which the number of I/O requests made is equal to the number of times data is requested. With the data sieving technique, the I/O performance increases because the number of I/O calls is reduced and if a large and contiguous request outweighs the penalty of reading and extracting extra data. The data sieving technique has been extensively evaluated and proven beneficial for many applications [17][18].

The implementation of the data sieving technique is straightforward in general. In the read case, the data sieving approach first reads the entire contiguous chunk starting from the lowest offset of all requests to the highest offset of all requests. This contiguous chunk includes non-requested data, also called holes. After the entire chunk is read into temporary memory, the data sieving approach sieves out non-requested data. Only the demanded data are kept and copied into user buffer. Given a large number of I/O requests, and the possible wide distribution of these requests, the temporary memory requirement of the conventional data sieving technique could be high. In the write case, it is slightly more complicated than in the read case because the data sieving needs to perform a read-modify-write operation. In addition, as other processes can try to access the same region, an atomic read-modify-write is needed. The data sieving technique has been well implemented in ROMIO, the most popular implementation of the MPI-IO middleware [14][17].

3 Performance Model Directed Data Sieving

We propose a new performance model directed data sieving strategy to improve the I/O performance and to reduce the memory consumption of the existing approach. The essential idea of this strategy is that we model the performance of I/O requests, and based on the performance model, the new strategy dynamically determines the way to perform data sieving based on access patterns and the performance estimated from the model. It is essentially a heuristic data sieving approach that adapts to different I/O access patterns and makes the decision dynamically.

3.1 Performance Model

The purpose of a performance model is to estimate the performance of I/O requests and thus direct data sieving dynamically. The performance model does not need to be exactly accurate but provides useful heuristic direction. In our model, the time consumption for each data access primarily contains two parts, the time spent on accessing storage and the time spent on the network establishment and transmission. Table 1 lists the parameters considered in the performance model and the descriptions of them. The performance model is simple but effective. It has been verified that it has clear benefits of improving the performance and reducing the memory requirement for data sieving via experimental tests.

Table 1. Parameters and Descriptions

Parameters	Description
p	Number of I/O client processes in a client node
n	Number of storage nodes (file servers)
te	Time of establishing network connection for single node
tt	Network transmission time of one unit of data
cud	Time reading/writing one unit of data
lq_{dep}	The latency for outstanding I/Os
$size_{rd}$	Read data size of one I/O request
$size_{wr}$	Write data size of one I/O request

The basic idea of constructing network time is as follows. For each data access, the time spent on network, $T_{network}$, consists of the time spent on establishing the connection and the time spent on transferring the data. The storage access time, $T_{storage}$, consists of the start up time for one storage node I/O operation (s) and the time spent on the actual data read/write (T_{rw}). The latency for outstanding I/Os (lq_{dep}) also affects the overall time of data access and hence we consider it in the performance model as well. As storage node performance varies for read and write requests, we consider these two operations separately in the model. Thus, the total time can be written as a function of the above workload characteristics as:

$$T_{total} = function\ (T_{network}\ ,\ T_{storage}\ ,\ lq_{dep})$$

In practice, we can find the relations between above workload characteristics and can derive formulas that guide the performance estimation of I/O requests in a data sieving approach as shown in Table 2.

Table 2. Formula of Deriving I/O Performance

Total time required for establishing network connection	$te * p$
The total time spent on the network transmission	$\frac{tt * size_{rd}}{n}$ Or $\frac{tt * size_{wr}}{n}$
The total start up (s) time for I/O operations	$p * (seek\ time + system\ IO\ call)$
Total time spent on the actual data read/write (T_{rw})	$\frac{size_{rd} * cud}{n}$ Or $\frac{size_{wr} * cud}{n}$

Hence, the total required time to access a requested data from the storage node can be calculated from the above discussed parameters and formula as:

$$T_{total} = te * p + \frac{tt * size_{rd}}{n} + p * (seek\ time + system\ IO\ call) + \frac{size_{rd} * cud}{n} + lq_{dep}$$

The performance model contains those most critical parameters that determine the performance of an I/O system. With such a performance model, we are able to analyze the performance of a data sieving approach and perform data sieving dynamically based on different access patterns. In the proposed PMD data sieving approach, we explore two levels of improvements over the existing data sieving approach: 1) with the performance model, the PMD data sieving approach can dynamically determine when would be good to perform data sieving depending on specific access patterns; and 2) if it is determined to perform data sieving, the PMD data sieving approach will also determine how to perform data sieving to achieve the maximum benefits. We introduce two algorithms to achieve these goals, as discussed below.

3.2 When to Perform Data Sieving

The first algorithm we present for dynamically determining whether to perform data sieving or not depending on specific access patterns. This algorithm takes requests (with possible holes), and the rest of the parameters in the performance model as input, and outputs whether it is beneficial to perform data sieving for such requests.

The algorithm makes the decision of when to perform data sieving dynamically based on the performance model, access patterns, network, and the storage system performance. It compares the overhead of accessing holes (time of accessing a hole) and the time savings with the data sieving (reduced storage and network startup time and latency due to combined requests), and if the savings outperforms the overhead, then a data sieving approach is determined to be performed.

The first algorithm is a building block for the second algorithm. The second algorithm scans all requests from the lowest offset to the highest offset, and dynamically determines whether to perform data sieving for any two consecutive requests as discussed in the next subsection.

3.3 How to Perform Data Sieving

If it is determined that the data sieving technique is beneficial then the challenge is how to do it. The second algorithm solves this issue. The aim of this algorithm is to find different groups in which data sieving technique can be beneficial.

This algorithm starts grouping these I/O requests from the lowest offset to the largest one. This algorithm scans all requests and applies the first algorithm to determine whether each request should be handled with data sieving or not. For instance, this algorithm comes across the first request, and then it follows the first algorithm. Let us assume that it does not follow data sieving technique. Hence, it will treat the first I/O request independently and will not group with any other requests. After that it will need to make the decision between second and third request and assume, again it cannot group them together. In this case also, the second I/O request will be treated independently and won't be grouped with any other. Now, it comes across the third request. Assume this time the algorithm determines to adopt data sieving. Then, it will be grouped with the consecutive data request. The algorithm will keep grouping consecutive noncontiguous requests unless the decision from the first algorithm comes out to be NO. This process will be continued and the second algorithm will terminate at the end of the last I/O request.

4 Experimental Results and Analysis

In this section, we present the experimental results of the proposed PMD data sieving. We also compare it with the existing data sieving approach and the direct method where no data sieving is applied.

4.1 Experimental Environment

The experiments were conducted on a 65-node Sun Fire Linux-based cluster test bed, with Ubuntu 4.3.3-5 operating system with kernel 2.6.28.10, PVFS 2.8.1 file system and MPICH2-1.0.5p3 library and runtime environment. The tests were conducted with three I/O benchmark scenarios, one with all requests and holes among them have different sizes, one with sparse noncontiguous I/O requests and large holes exist among requests, and one with dense noncontiguous I/O requests where small size holes exist among requests. We performed the tests with three I/O access scenarios and combinations of them to measure the performance. The actual values of the parameters used in the performance model were obtained through measurement on the experimental platform. The values are, te: 0.0003sec, tt: 1/120 MB, s: 0.0003sec and cud: 1/120 MB. The run time measured for each scenario was obtained from the average of 100 runs.

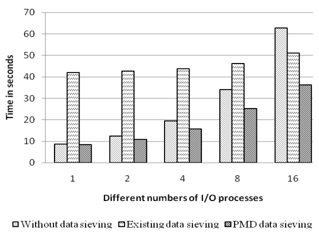

Fig. 1. Execution time for access scenario 1 (fixed storage nodes)

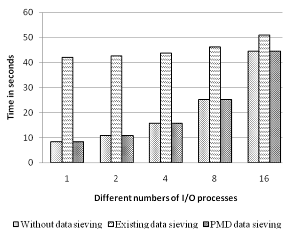

Fig. 2. Execution time for access scenario 2 (fixed storage nodes)

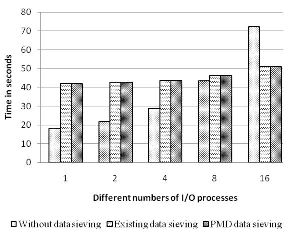

Fig. 3. Execution time for access scenario 3 (fixed storage nodes)

4.2 Experimental Results

The above and following figures report the time comparison between the three methods, the direct method, the existing data sieving approach and the proposed PMD data sieving approach. The Y-axis represents the run time in seconds. The X axis represents different number of I/O client processes, ranging from 1, 2, 4, 8 and 16. We fixed the number of storage nodes as 16 in these tests. Figure 1 plots the run time results of all three strategies for the first access scenario. In all cases, the PMD data sieving approach performed better than the current data sieving approach and the direct method.

Figure 2 and Figure 3 are similar to Figure 1, and plot the run time results for the access scenarios 2 and 3, respectively. The PMD approach performed almost equally well with the direct method, whereas the existing data sieving technique had worse performance in this case as shown in Figure 2. Figure 3 demonstrates that, as the number of I/O client processes increased, the run time of the direct method increased drastically. In the case of one process, the direct method performed better than both data sieving approaches; whereas the direct method performed worse than the other two in the case of 16 processes.

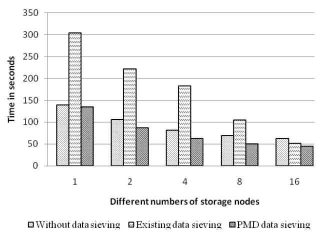

Fig. 4. Execution time for access scenario 1 (fixed client processes)

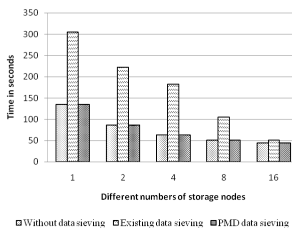

Fig. 5. Execution time for access scenario 2 (fixed client processes)

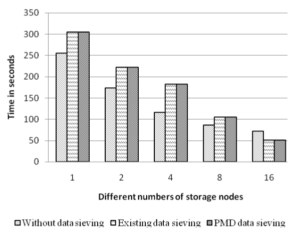

Fig. 6. Execution time for access scenario 3 (fixed client processes)

We have also fixed the number of processes at 16 and varied the number of storage nodes to observe the performance variations. In this set of tests, the storage nodes were varied from 1, 2, 4, 8 and 16. Figures 4, 5, and 6 report the results of these tests for access scenarios 1, 2 and 3 respectively. In these figures, the X axis represents the number of storage nodes while Y axis represents the run time in seconds. As the number of storage nodes increased, the distribution of requested data also increased. The transmission time and the time spent on the actual data read/write were decreased. All the three graphs confirmed the decreasing trend. In these tests, the existing data sieving performed better than the direct method for access scenario 3, while the PMD data sieving achieved the best performance in most cases because of its capability of making data sieving decisions dynamically based on different access patterns.

5 Related Work

There has been significant amount of research effort in optimizing parallel I/O performance, such as collective I/O [17], two-phase I/O [2], extended two-phase I/O [18], data sieving [17], and ADIOS library [8], resonant I/O, I/O forwarding [6]. These strategies demonstrate that data sieving is one of the most successful and widely used approaches to collect and merge requests into a large and contiguous one to carry out more efficiently. This study further improves the data sieving approach and proposes an intelligent performance model directed data sieving that dynamically makes the decision for when and how to conduct data sieving. It advances the state of the art in these areas.

Many research efforts have also been devoted to caching and prefetching optimizations for high performance I/O systems [11][12][10][19][1][7][16][13], While prefetching optimizations can hide I/O access latency and caching optimizations can reduce the I/O requests to underlying storage devices, they cannot completely eliminate small and noncontiguous I/O requests. The data sieving and the proposed performance model directed data sieving approach are complementary to them and are critical for providing a high performance I/O system.

Parallel file systems [4][15][20][3][16], enable concurrent I/O accesses from multiple clients to files. While parallel file systems perform well for large and well-formed data streams, they often perform inadequately in dealing with many small and noncontiguous data requests. The data sieving and the enhanced performance model directed data sieving approach proposed in this study address these issues well. This research will have an impact for high performance parallel I/O system.

6 Conclusion

Poor I/O performance is a critical hurdle in HPC systems, especially for data-intensive applications. These applications often exhibit small and noncontiguous accesses, and it is important to deliver high performance for these accesses. Data sieving remains a critical approach to improve the performance of small and noncontiguous accesses [17][18]. The existing data sieving strategy, however, suffers large memory requirement pressure and is static. This study proposes a new performance model directed (PMD) data sieving approach and addresses the drawbacks. The proposed approach is essentially a heuristic data sieving approach directed by the performance estimation given from a performance model. The PMD data sieving approach dynamically makes the decision on when and how to perform data sieving based on different access patterns. The experimental results have confirmed its benefits and advantages over the widely used conventional method. It improves the I/O performance and reduces the memory requirement. Given the importance of a data sieving approach to improve the performance of small and noncontiguous I/O, the PMD data sieving approach will have an impact on high performance I/O systems.

References

1. J. G. Blas, F. Isaila, J. Carretero, R. Latham and R. Ross. Multiple-Level MPI File Write-Back and Prefetching for Blue Gene Systems. *In Proc. of PVM/MPI, 2009.*
2. 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.
3. 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.*
4. Cluster File Systems Inc., “Lustre: A scalable, high performance file system”, Whitepaper, http://www.lustre.org/docs/whitepaper.pdf
5. P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed, “Input/output characteristics of scalable parallel applications”, in *Proc. of the ACM/IEEE Conf. on Supercomputing,* pp.59-es, 1995.
6. K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman. “ZOID: I/O Forwarding Infrastructure for Petascale Architectures.” In *Proc. of the 13th ACM PPoPP*, 2008.
7. H. Lei and D. Duchamp. “An Analytical Approach to File Prefetching.” In *Proceedings of the 1997 USENIX Annual Technical Conference,* pages 275–288, Jan. 1997.
8. J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki and C. Jin. “Flexible I/O and Integration for Scientific Codes Through the Adaptable I/O System (ADIOS).” In *Proc. of the 6th International Workshop on Challenges of Large Applications in Distributed Environments,* 2008.
9. J. May. “Parallel I/O for High Performance Computing.” *Morgan Kaufmann,* 2001.
10. X. S. Ma, M. Winslett, et. al. “Faster Collective Output through Active Buffering.” *IPDPS,* 2002.
11. A. Nisar, W.-K. Liao, A. Choudhary. “Scaling Parallel I/O Performance through I/O Delegate and Caching System.” *SC*, 2008.
12. B. Nitzberg, et. al. “Collective Buffering: Improving Parallel I/O Performance.” *HPDC,* 1997.
13. M. M. Rafique, A. R. Butt and D. S. Nikolopoulos. “DMA-based Prefetching for I/O-Intensive Workloads on the Cell Architecture.” *Conf. Computing Frontiers,* 23-32, 2008.
14. ROMIO website. http://www-unix.mcs.anl.gov/romio/.
15. F. Schmuck and R. Haskin, “GPFS: A Shared-Disk File System for Large Computing Clusters”, in *Proceedings of the First USENIX FAST*, pp. 231-244, USENIX, January 2002.
16. N. Tran and D. A. Reed. “Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching.” *IEEE Trans. Parallel Distrib.* Syst. 15(4): 362-377 (2004).
17. 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.
18. R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. “Passion: Optimized I/O for Parallel Applications.” *Computer*, 29(6):70–78, June 1996.
19. M. Vilayannur, A. Sivasubramaniam, M. T. Kandemir, R.Thakur and R. Ross. “Discretionary Caching for I/O on Clusters.” *Cluster Computing* 9(1): 29-44, 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.” USENIX FAST*,* 2008.
21. X. Zhang, S. Jiang, and K. Davis. “Making Resonance a Common Case: A High-performance Implementation of Collective I/O on Parallel File Systems.” IPDPS 2009.