Pre-execution Data Prefetching with Inter-thread I/O Scheduling

Yue Zhao, Kenji Yoshigoe, and Mengjun Xie

Department of Computer Science, University of Arkansas at Little Rock
2801 S. University Avenue, Little Rock, Arkansas 72204, USA
{yxzhao,kxyoshigoe,mxxie}@ualr.edu

Abstract. With the rate of computing power growing much faster than that of storage I/O access, parallel applications suffer more from I/O latency. I/O prefetching is effective in hiding I/O latency. However, existing I/O prefetching techniques are conservative and their effectiveness is limited. Recently, a more aggressive prefetching approach named pre-execution prefetching [19] has been proposed. In this paper, we first identify the drawback of this pre-execution prefetching approach, and then propose a new method to overcome the drawback by scheduling the I/O operations between the main thread and the prefetching thread. By careful I/O scheduling, our approach further extends the computation and I/O concurrency and avoids the I/O competition within one process. The results of extensive experiments, including experiments on real-life applications such as big matrix manipulation and Hill encryption, demonstrate the benefits of the proposed approach.

1 Introduction

Parallel applications execution suffers from large latency of I/O accesses. The poor I/O performance has been attributed as a critical cause of the low sustained performance of parallel systems ([1], [2]). In order to improve I/O performance numerous works have been conducted. However, their effectiveness and practicability are limited by their inherent drawbacks.

A remarkable advancement in I/O parallelism ([5], [6], [7], [18]) has been achieved. However, this advancement in I/O parallelism is accompanied with a much more expeditious development of parallel processing both on hardware and software, so it is still not capable of reducing the I/O latency effectively. The Adaptable IO System (ADIOS) ([22], [23]) and non-blocking I/O [21] can gain a high I/O performance improvement but they require application modification ([21], [24]). The effectiveness of collective I/O and data sieving ([8], [9]) is application dependent. Due to the inherent nature of applications, there are still many small I/O requests that cannot be eliminated [19]. Studies [3] and [4] use data compression scheme to reduce the amount of I/O traffic. However, limited by the data condensability, compression rate and extra overhead on the system management, the exploitation of data compression approach in practice is restricted. Traditional prefetching strategies ([10], [11], [12], [13], [14], [15],

[16], [17]) are conservative and most cannot guarantee the prefetching accuracy and timeliness.

Considering computing power is plenty but data access is the bottleneck and most of existing I/O prefetching techniques are conservative and their effectiveness is limited, Chen et al. [19] proposed a pre-execution I/O prefetching approach. Pre-execution I/O prefetching approach is promising in reducing I/O access latency and it can convert original applications to prefetching version automatically. Following this direction, in order to overcome the limitation due to read after write (RAW) dependency and further extend the computation and I/O concurrency, Zhao et al. [20] proposed a parallel pre-execution prefetching (PPP) approach. However, both [19] and [20] do not pay attention to the relationship among the I/O accesses conducted by diverse threads. And they failed to further extend the I/O and computation concurrency by carefully coordinating the I/O accesses. Our work aims to resolve this issue by developing a new approach named pre-execution prefetching with inter-thread I/O scheduling (PPIS). With PPIS we extend the computation and I/O concurrency while avoiding the I/O competition caused by multiple concurrent I/O operations requested by the main thread and prefetching thread in one process.

The rest of the paper is organized as follows. Section 2 describes the motivation of this work. Section 3 presents PPIS. Section 4 details the experiment designs and results. Section 5 concludes this paper and states our future work.

2 Motivation

In [19], Chen et al. proposed a pre-execution prefetching approach (PP). The basic idea is to pre-execute a portion of code on each process to identify future I/O references, and then fetch the data closer to CPU in advance in order to overlap the computation and I/O access.

PP approach aims to overlap the computation and I/O access by creating a pre-execution prefetching thread (PT) to work with the main thread (MT) in parallel. However, a portion of I/O accesses requested by PT may be overlapped with MT's I/O accesses when MT has I/O operations such as writes and those reads that cannot be conducted by PT early enough. In other words, this portion of pre-executed I/O accesses fails to be hidden by computation. This issue will result in a series of adverse effects. First, it diminishes the degree of the parallelism between computation and I/O, which affects the effectiveness of pre-execution prefetching. Second, the I/O resource competition between the simultaneous I/O accesses of MT and PT can delay MT's I/O access, which goes against the purpose of pre-execution prefetching to accelerate the execution of the original program. Third and most importantly, PP does not take into account the global I/O network and file system source competition. Simply launching more concurrent I/O requests within local process will result in high I/O competition even I/O congestion in the whole system, and end up overwhelming the network and the file system, which not just limits the scalability of the pre-execution prefetching, but makes the prefetcher counter-productive as well.

Figure 1 illustrates an application scenario and shows how it runs under normal execution mode and PP mode, where the size of each operated segment represents time duration. This application scenario is typical in real applications such as big matrix manipulation and big file encryption where the process in normal execution mode sequentially processes a large volume of data. For each piece of data, data reading, computation, and writing are executed in sequence. Under PP execution mode, the process contains two threads, MT and PT. Since I/O access is the focus of PT and in this scenario dominates PT's execution time, we can safely ignore the time incurred by computation conducted by PT in Fig. 1. As under PP mode, PT is designed to do data prefetching as fast as possible, I/O overlap between PT and MT is easy to occur. The scenario in Fig. 1 shows that under PP mode, a high portion of I/O accesses of PT overlaps with that of MT. They are R_2 overlapping with R_1 and R_4 overlapping with W_1. Only the I/O operation R_3 is successfully overlapped with the computation of MT. In this scenario, the computation and I/O access concurrency achieved by PP is much limited. Worse, the I/O overlap between MT and PT results in the I/O resource competition, which makes the I/O access latency (R_1 , R_2 , W_1 and R_4) longer, and then delays the normal execution of MT. When the impact induced by prefetching operations conducted by other processes is taken into account the outcome will be even worse. In case hundreds of processes are employed for a large computing job, which is common for high-performance computing applications, doubled number of concurrent I/O accesses induced by PP can even cause I/O congestion.

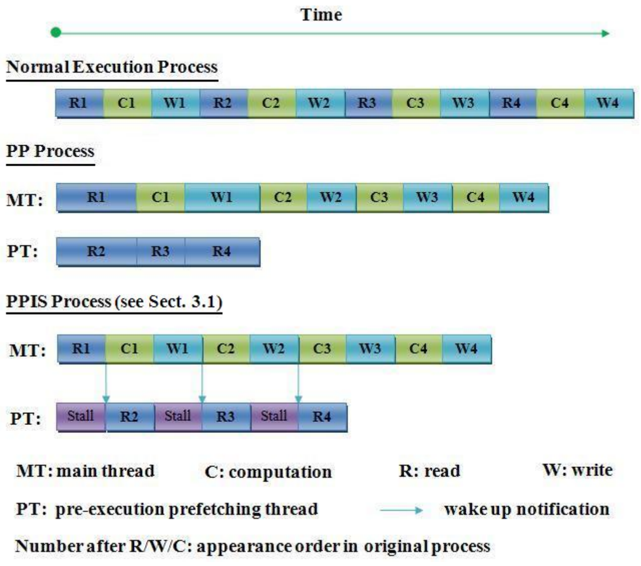

Fig. 1. Hiding I/O Latency with PP and PPIS

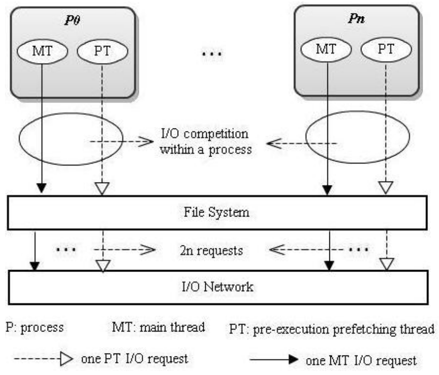

Fig. 2. I/O Workflow of PP

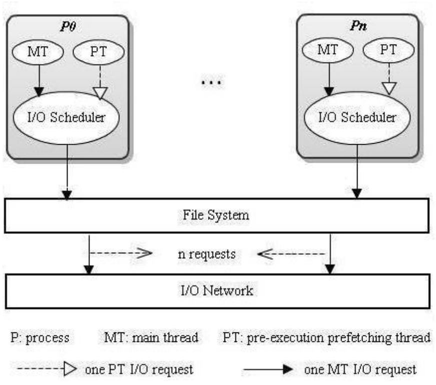

Fig. 3. I/O Workflow of PPIS

3 Pre-execution Prefetching with Inter-thread I/O Scheduling

3.1 Description

We propose a new approach, pre-execution prefetching with inter-thread I/O scheduling (PPIS), to further improve the pre-execution prefetching strategy. The benefit of our approach is twofold. First, we extend the degree of computation and I/O concurrency of a parallel application, and further hide the I/O latency. Second, we improve the scalability of the pre-execution prefetching by avoiding multiple concurrent I/O operations conducted within each process.

In PPIS, we assign a higher priority to MT's I/O accesses to make sure they gain the maximal system I/O resources. Concretely, only when MT is not performing I/O operation can PT launch an I/O operation. In case MT needs to perform I/O operations while at this moment PT is still doing I/O prefetching, we suspend PT's prefetching and record its current prefetching status information, for example the identifier of the block that has just been completely prefetched. After MT finishes its I/O access, MT notifies PT to continue prefetching. By scheduling the I/O accesses of MT and PT in a coordinate manner, PPIS can maximize the parallelism of I/O access and computation, and meanwhile avoid the I/O competition within one process. Figure 2 and Fig. 3 show the I/O workflow of threads when both MT and PT are conducting I/O operation under PP and PPIS scenarios respectively.

The advantages of PPIS over PP are illustrated in Fig. 1, which compares how PPIS and PP progress in the scenario mentioned in Sect. 2. In this ideal case, PPIS maximizes PT's I/O access and MT's computation concurrency. Moreover, it avoids the I/O competition between the two threads, which not only optimizes the completion time of MT's I/O access, but also avoids the potential I/O congestion caused by PP when the number of concurrent processes is large.

3.2 Implementation

Software Stack. We employ MPI protocol and its parallel API to actualize the execution of parallel applications. In order to implement PT co-working with MT within each process, we adopt the POSIX Threads (Pthreads) multi-threaded programming standard. We conduct the parallel file system access through the ROMIO MPI-IO implementation in Open MPI. Figure 4 shows the software stack to implement our approach.

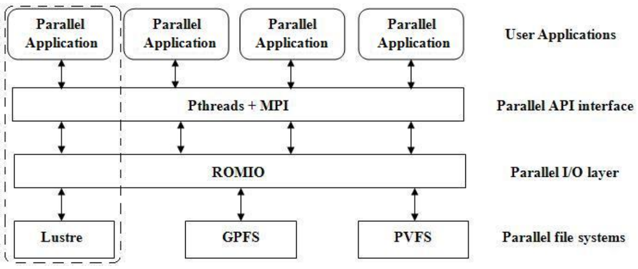

Fig. 4. Software Stack (Module inside the dashed line box represents the experiment environment used in Sect. 4)

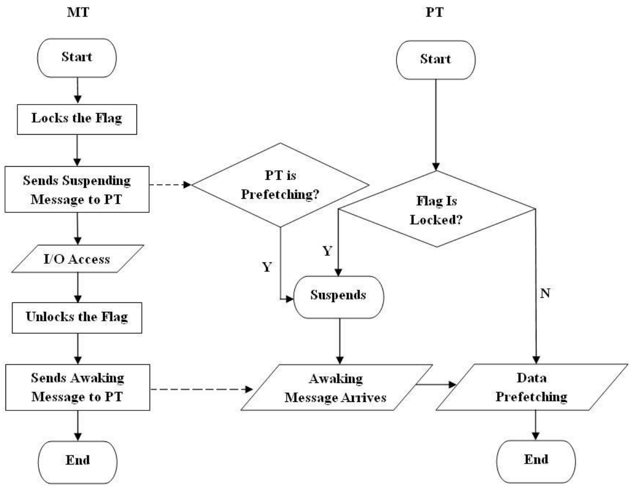

Fig. 5. Threads I/O Related Behavior of PPIS

Threads I/O Related Behavior to Implement I/O Scheduling. We employ a condition variable and Pthreads inter-thread message passing mechanism to accomplish the inter-thread I/O scheduling. The condition variable is used as a flag managed by MT. Initially the flag is set as unlocked. When MT starts performing I/O access it first locks the flag. Locked flag indicates that PT cannot launch any I/O accesses. Otherwise, the prefetching is allowed. When PT encounters a read function, it has to check the flag's status first. If the flag is locked, then PT goes into the suspend status. When MT finishes its I/O access it unlocks the flag and sends a notification to wake up and allow PT to prefetch data into cache. Figure 5 shows the I/O related behavior of each thread in PPIS.

3.3 Analysis

Optimal Analysis. PP introduces extra I/O accesses over normal execution. In the worst scenario, all MT and PT I/O accesses collide and result in I/O congestion. Then the original application execution can be delayed infinitely. By introducing I/O scheduling PPIS not only avoids this issue but also further extends the degree of I/O access and computation concurrency of a parallel program. In this section we analyze the optimal speedup achieved by PPIS over normal execution.

Table 1. Notations

N_C	the number of segments of computation in the original application
N_W	the number of segments of write operation in the original application
N_R	the number of segments of read operation in the original application
T_{Normal}	the total execution time of the application under normal execution
T_{PPIS}	the total execution time of the application under PPIS mode
$T_{TH\_OP\_SN\_MO}$	The execution time of a certain operation (TH: thread; OP: operation; SN: serial number; Mode: execution mode (e.g., $T_{MT\_R\_i\_Normal}$ refers to the execution time of MT's ith segment of read under normal execution mode))
$N_{TH\_OP\_MO}$	The number of segments of a certain operation (e.g., $N_{PT\_R\_PPIS}$ refers to the number of segments of read conducted by PT under PPIS mode)

The total execution time of the program under a certain mode is actually MT's execution time, so

$$T_{Mode} = \sum_{i=1}^{N_{MT\_R\_Mode}} T_{MT\_R\_i\_Mode} + \sum_{j=1}^{N_{MT\_W\_Mode}} T_{MT\_W\_j\_Mode} + \sum_{k=1}^{N_{MT\_C\_Mode}} T_{MT\_C\_k\_Mode}. \quad (1)$$

Here, $Mode \in \{Normal, PP, PPIS\}$. Under normal execution mode MT conducts all the operations exactly identical to how the original application does. And under PPIS all write and computation operations are conducted by MT with the same progress time as those under normal execution mode assuming the I/O accesses of other processes in the system are not disturbed by the prefetching. Thus,

$$T_{Normal} = \sum_{i=1}^{N_R} T_{MT\_R\_i\_Normal} + \sum_{j=1}^{N_W} T_{MT\_W\_j\_Normal} + \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}. \quad (2)$$

$$T_{PPIS} = \sum_{i=1}^{N_{MT\_R\_PPIS}} T_{MT\_R\_i\_PPIS} + \sum_{j=1}^{N_W} T_{MT\_W\_j\_Normal} + \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}. \quad (3)$$

In the optimal case, all the data to be read in the application is prefetched by PT with completely overlapping with the computation of MT, then,

$$T_{PPIS} = \sum_{j=1}^{N_W} T_{MT\_W\_j\_Normal} + \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}. \quad (4)$$

The speedup achieved by PPIS over the normal execution mode is:

$$\begin{aligned} &Speedup_{(PPIS/Normal)} \\ &= T_{Normal}/T_{PPIS} \\ &= \frac{\sum_{i=1}^{N_R} T_{MT\_R\_i\_Normal} + \sum_{j=1}^{N_W} T_{MT\_W\_j\_Normal} + \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}}{\sum_{j=1}^{N_W} T_{MT\_W\_j\_Normal} + \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}} \\ &<= \frac{\sum_{i=1}^{N_R} T_{MT\_R\_i\_Normal} + \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}}{\sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}}. \end{aligned} \quad (5)$$

To hide all the read latency by computation, there must be

$$\sum_{i=1}^{N_R} T_{MT\_R\_i\_Normal} < \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}. \quad (6)$$

With

$$\lim \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal} = \sum_{i=1}^{N_R} T_{MT\_R\_i\_Normal}. \quad (7)$$

So,

$$Speedup_{PPIS/Normal} <= 2\sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal} / \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal} = 2. \quad (8)$$

Namely,

$$\max\{Speedup_{(PPIS/Normal)}\} = 2. \quad (9)$$

So, in the optimal scenario, PPIS can hide all the read latency suffered by an application and achieve 50% total execution time reduction of that application over normal execution.

Cost. Over the existing pre-execution prefetching approach, in which I/O related operations conducted by PT do not involve communication with other processes in general [19], PPIS requires an inter-thread I/O scheduling, which is quite light-weight in cost. Throughout the implementation of the I/O scheduling, only a condition variable and several inter-thread messages are added to the existing pre-execution prefetching implementation. Also, the messages are thread control messages with no additional data transported, which are quite small in size. Thus, the overhead caused by I/O scheduling is negligible, especially, when it is compared to the huge workload of parallel applications. The cost does not impact the effectiveness of PPIS, which is also verified by the performance improvement achieved by PPIS as shown in Sect. 4.

Correctness. First, the existing pre-execution prefetching approach can guarantee the correctness of the original program. Second, the I/O scheduling between PT and MT only deals with the prefetching time of PT, so it does not affect the logical behavior and accuracy of MT. Thus, the MT in a program running with and without PPIS will logically behave identically. In summary, PPIS does not affect the correctness of the original program.

Thread Safety. First, only one global variable is added on top of PP to implement PPIS. Since only MT has the access to write it, there is no concurrent read/write by multiple threads on this global variable. Thus, no additional thread safety risk is induced to the existing pre-execution prefetching. Second, by introducing a prefetching file pointer as a hidden file offset pointer within the non-transparent MPI file handle object in order to track the prefetching thread file offset, the thread safety can be guaranteed naturally by PP [19]. Therefore, PPIS is thread-safe.

4 Experiment

Our experiments were conducted on a 66-node 528 processors Linux-based cluster. Each compute node has 16 GB of RAM and 2 CPU sockets, each with quad-core Intel Xeon 2.66GHz CPU. Depending on the number of processes in experiment, we used the subset of this cluster with size ranging from 1 to 16 compute nodes. We dynamically assign the buffer size as demand in each node, which can be large enough for our experiments as each node has 16 GB of RAM. Software environment refers to the dashed line box in Fig. 4.

4.1 Design

We evaluated the benefits of our approach on big matrix searching operation and Hill encryption application respectively. The former was tested under the system with light I/O workload. The later was tested under the system with light and heavy I/O workload, respectively. For the light I/O workload, only our experiment benchmark accessed the disk in the system. To achieve a heavy workload environment, we conducted multiple simultaneous large file I/O operations in the system. We evaluated the results with three metrics, total execution time, aggregate sustained bandwidth and I/O latency, which are the most important performance metrics in practice.

Experiment #1: Big Matrix Searching. Big matrix searching is the fundamental operation of many real parallel applications. In this experiment we conducted searches in a big integer matrix, which is 4GB in size, to find its top 30 maximum items. The matrix was split into 4 sub-matrices with equal size to process in sequence.

Experiment #2: Hill Encryption. Hill encryption is a real application to encrypt data with Hill cipher, in which the key is a matrix. When the plaintext data is large it will be partitioned into smaller chunks, and then these chunks are encrypted in sequence. In this experiment we encrypted a big file of 6GB with the key matrix size was set as 100 by 100. First, we tested the case in which the chunk size is 2GB. Then we further tested the I/O latency reduction achieved by PPIS over normal execution mode under different chunk sizes.

4.2 Results

Experiment #1. Figure 6 shows the total execution time results. The execution time under the PPIS mode is reduced in all the cases showed in Fig.6 compared to normal execution and PP, respectively. Over normal execution and PP the maximal reduction is 28.6% and 28.2% when the number of processes is 16 and 64, respectively. As a reference, Fig. 6 also shows the application's execution time under the theoretically optimal scenario, in which the I/O latency would be completely masked. The computation dominates the whole application when the number of processes is small (e.g., 1 and 2). Thus, even if a large amount of I/O latency was hidden by PPIS the reduction percentage of the whole application execution time is quite marginal. For large applications which run

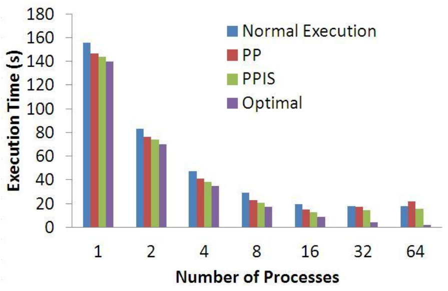

Fig. 6. Execution Time

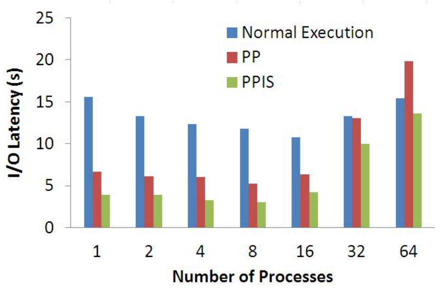

Fig. 7. I/O Latency

tens of days or even months, the reduction of the execution time is significant as the amount of I/O time being hidden would constitute tens of hours of the applications. When the number of the processes is large the computing workload assigned to each process is quite low, which limits the amount of I/O latency hidden by computation. Thus, high execution time reduction percentage under PPIS mode can be achieved with moderate number of parallelisms as observed in [20]. In Fig. 6, when the number of processes exceeds 16 and 32 respectively, the execution time under the PP mode starts to increase. With the number of processes is 64, it is even larger than that of the normal execution. PPIS, on the other hand, can achieve execution time reduction compared to the other two modes in all the cases.

Figure 7 shows the corresponding results of I/O latency during the whole execution of the application. In most of cases, a considerable I/O latency reduction percentage has been achieved by PPIS over the other two modes with the maximal reduction being 75.1% over normal execution mode when the number of processes is 1 and 54.3% over PP mode when the number of processes is 4. Most importantly, the PPIS outperforms the normal and PP modes for all process sizes being evaluated while the normal and PP modes outperform/underperform over one another at certain process sizes being evaluated.

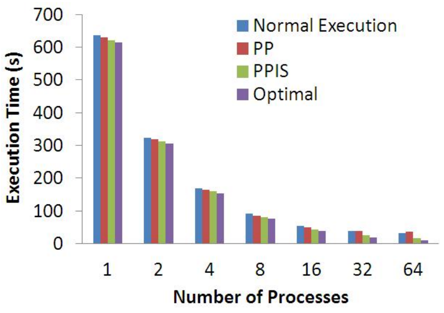

Fig. 8. Execution Time (light I/O)

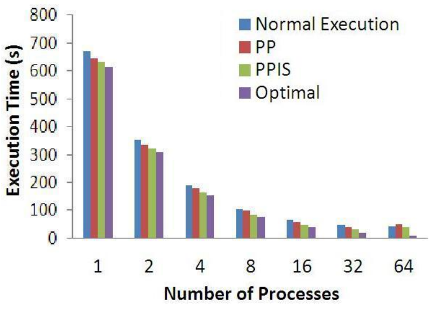

Fig. 9. Execution Time (heavy I/O)

Experiment #2. Figure 8 and Fig. 9 show the execution time of the Hill encryption with the chunk size is 2GB. Likewise, PPIS achieves execution time reduction in all cases shown in the figures compared to normal execution and PP, respectively. Under the light workload system, the best execution time reduction achieved by PPIS is 47.4% and 54.6% over normal execution and PP with the number of processes is 32 and 64, respectively. Under the heavy workload system, the corresponding results turn out to be 27.6% and 24.0% with the number of processes is 16 and 64, respectively. Besides, better scalability was achieved by PPIS compared to PP in both circumstances.

Figure 10 and Fig. 11 show the I/O latency during the whole encryption process. Quite high I/O latency reduction percentage acquired by PPIS is observed with the maximal reduction being 67.0% over normal execution and 62.9% over PP under the number of processes being 1 and 32, respectively.

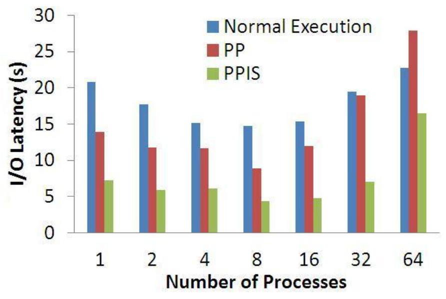

Fig. 10. I/O Latency (light I/O)

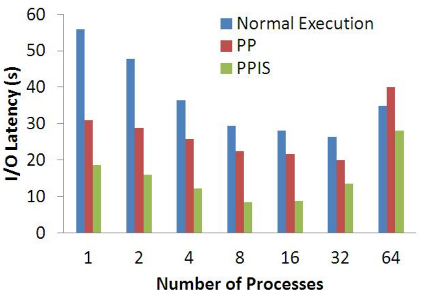

Fig. 11. I/O Latency (heavy I/O)

Figure 12 shows the I/O latency reduction achieved by PPIS over normal execution mode under light I/O workload as the chunk size changes. In some cases, the I/O latency reduction is close to 100%. It demonstrates that in these cases PPIS can almost hide the entire I/O latency of the Hill encryption application by scheduling the pre-executed I/O operation to strictly overlap with computation. When the number of process is larger than or equal to 32, the reduction drops. The reason is that the computation workload assigned to each process is too small to hide all the I/O latency.

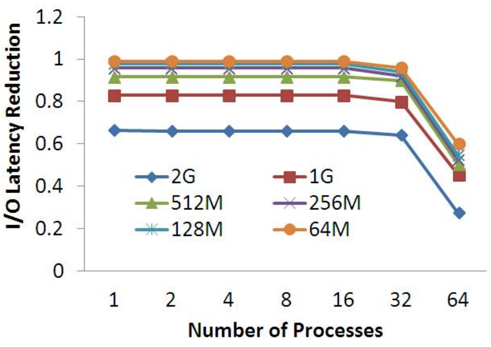

Fig. 12. I/O Latency Reduction

In experiments #1 and #2 when the number of processes is larger such as 128 or 256 the advantage of PPIS in terms of scalability is more remarkable. For instance, the execution time reduction achieved by PPIS compared to PP for 256 processes under the heavy I/O workload system in Experiment #2 is 42.1%. It is larger than those under the number of processes being 128 and 64, which are 31% and 24.0%, respectively. The effectiveness of PPIS under heavy workload system also indicates that PPIS benefits the application, which is running simultaneously with other multiple applications in the system as well.

5 Conclusion and Future Work

Parallel applications continue to suffer from I/O latency. In this study, by proposing PPIS approach, we enhanced the existing pre-execution prefetching strategy to further hide I/O latency. Meanwhile, the new pre-execution prefetching approach is more scalable. The main contribution of this study is that we employed the active scheduling or careful coordination on the normal and pre-executed I/O accesses to maximum the overlap between the pre-executed I/O accesses and computation, which is the first work to the best of our knowledge in this research direction. Compared to the existing pre-execution prefetching approach PPIS extends the degree of computation and I/O concurrency, and also avoids the I/O congestion caused by multiple I/O operations requested by one process synchronously. The extensive evaluation results, including one from Hill encryption as a real-life application, have verified that the proposed approach has more potential and better scalability to hide I/O access delay than the existing approach. In order to further decrease or avoid impact on all processes in system introduced by prefetching our future work is to schedule the normal and pre-executed I/O accesses in inter-process level.

Acknowledgement. This work is supported in part by the National Science Foundation under Grant CRI CNS-0855248, Grant EPS-0701890, Grant EPS-0918970, and Grant MRI CNS-0619069.

References

1. Chen, Y., Sun, X.H., Thakur, R., Roth, P.C., Gropp, W.: LACIO: a new collective I/O strategy for parallel I/O systems. In: Proc. of IEEE IPDPS 2011, pp. 794–804 (2011)
2. Sun, X.-H., Chen, Y., Wu, M.: Scalability of heterogeneous computing. In: Proc. of 34th International Conference on Parallel Processing (2005)
3. Makatos, T., Klonatos, Y., Marazakis, M., Flouris, M.D., Bilas, A.: Using transparent compression to improve ssd-based i/o caches. In: Proc. of the ACM EuroSys 2010, pp. 1–14 (2010)
4. Welton, B., Kimpe, D., Cope, J., Patrick, C., Iskra, K., Ross, R.: Improving I/O forwarding throughput with data compression. In: Proc. of IEEE CLUSTER 2011, pp. 438–445 (2011)
5. `http://www.oracle.com/us/products/serversstorage/storage/storage-software/031855.htm` (accessed: March 09, 2010)
6. Ligon, W., Ross, R.: Parallel I/O and the parallel virtual file system. In: Beowulf Cluster Computing with Linux, Cambridge, MA, pp. 493–534 (2003)
7. Schmuck, F., Haskin, R.: GPFS: A shared-disk file system for large computing clusters. In: Proc. of the 1st USENIX Conference on File and Storage Technologies (2002)
8. Reed, D.: Scalable Input/Output: achieving system balance. The MIT Press (2003)
9. Thakur, R., Gropp, W., Lusk, E.: Data sieving and collective I/O in ROMIO. In: Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation (1999)

10. Ding, X., Jiang, S., Chen, F., Davis, K., Zhang, X.: DiskSeen: exploiting disk layout and access history to enhance I/O prefetch. In: Proc. of USENIX Annual Technical Conference (2007)
11. Kotz, D.F., Ellis, C.S.: Prefetching in file systems for MIMD multiprocessors. IEEE Transactions on Parallel and Distributed Systems 1(2) (1990)
12. May, J.: Parallel I/O for high performance computing. Morgan Kaufmann Publishing (2001)
13. Papathanasiou, A., Scott, M.: Aggressive Prefetching: an idea whose time has come. In: Proc. of the 10th Workshop on Hot Topics in Operating Systems (2005)
14. Patterson, R.H.: Informed prefetching and caching, Carnegie Mellon Ph.D. Dissertation, CMU-CS-97-204 (1997)
15. Son, S.W., Kandemir, M., Karakoy, M., Chakrabarti, D.: A compiler-directed data prefetching scheme for chip multiprocessors. In: Proc. of the 14th Symposium on Principles and Practice of Parallel Programming, pp. 209–218 (2009)
16. Ravichandran, N., Paris, J.F.: Making early predictions of file accesses. In: Proc. of the 4th Int. Inf. Telecommun. Technol., pp. 122–129 (2005)
17. Brown, A.D., Mowry, T.C., Krieger, O.: Compiler-based I/O prefetching for out of- core applications. ACM Transactions on Computer Systems 19(2) (2001)
18. Chen, Y., Byna, S., Sun, X.-H., Thakur, R., Gropp, W.: Exploring parallel I/O concurrency with speculative prefetching. In: Proc. of the ICPP (2008)
19. Chen, Y., Byna, S., Sun, X.H., Thakur, R., Gropp, W.: Hiding I/O latency with pre-execution prefetching for parallel applications. In: Proc. of SC (2008)
20. Zhao, Y., Yoshigoe, K.: Hiding I/O latency with parallel pre-execution prefetching. In: Proc. of the 24th IASTED International Conference on Parallel and Distributed Computing and Systems, PDCS 2012 (2012)
21. Buettner, D., Kunkel, J., Ludwig, T.: Using non-blocking I/O operations in high performance computing to reduce execution times. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/MPI. LNCS, vol. 5759, pp. 134–142. Springer, Heidelberg (2009)
22. Lofstead, J.F., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: 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, pp. 15–24 (2008)
23. Jin, C., Klasky, S., Hodson, S., Yu, W., Lofstead, J., Abbasi, H., Schwan, K., Wolf, M., Liao, W., Choudhary, A., Parashar, M., Docan, C., Oldfield, R.: Adaptive io system (adios). In: Cray User's Group (2008)
24. Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., Wingate, M.: Plfs: A checkpoint filesystem for parallel applications. In: Proc. of SC 2009 (2009)