J Supercomput
DOI 10.1007/s11227-013-1060-2

Pre-execution data prefetching with I/O scheduling

Yue Zhao · Kenji Yoshigoe · Mengjun Xie

Abstract Parallel applications suffer from I/O latency. Pre-execution I/O prefetching is effective in hiding I/O latency, in which a pre-execution prefetching thread is created and dedicated to fetch the data for the main thread in advance. However, existing pre-execution prefetching works do not pay attention to the relationship between the main thread and the pre-execution prefetching thread. They just simply pre-execute the I/O accesses using the prefetching thread as soon as possible failing to carefully coordinate them with the operations of the main thread. This drawback induces a series of adverse effects on pre-execution prefetching such as diminishing the degree of the parallelism between computation and I/O, delaying the I/O access of main threads, and aggravating the I/O resource competition in the whole system. In this paper, we propose a new method to overcome this drawback by scheduling the I/O operations among the main threads and the pre-execution prefetching threads. The results of extensive experiments on four popular benchmarks in parallel I/O performance area demonstrate the benefits of the proposed approach.

Keywords Parallel application · I/O latency · Pre-execution prefetching · I/O scheduling

Y. Zhao (✉) · K. Yoshigoe · M. Xie
Department of Computer Science, University of Arkansas at Little Rock,
2801 S. University Avenue, Little Rock, AR 72204, USA
e-mail: yxzhao@ualr.edu

K. Yoshigoe
e-mail: kxyoshigoe@ualr.edu

M. Xie
e-mail: mxxie@ualr.edu

Published online: 25 December 2013

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 [1–4]. While the computational power of supercomputers keeps increasing rapidly with every generation, the same is not true for their I/O subsystems, and the data access rates of storage devices have not kept pace with the exponential growth in microprocessor performance [5]. This is called I/O wall problem, which has become a critical issue that limits the performance of parallel applications. Numerous works have been conducted to resolve this I/O wall problem. However, the I/O latency has not been relieved remarkably.

Works on prefetching [6–15] can reduce the I/O waiting time by fetching the data to be read closer to CPU in advance. However, these traditional prefetching strategies are conservative and cannot guarantee the prefetching accuracy [16]. Recently, a pre-execution prefetching approach [16,17], in which a pre-execution prefetching thread (PT) is created and dedicated to fetch the data for the main thread (MT) in advance, has been proposed to hide the I/O latency with computation. However, by simply pre-executing the I/O accesses with PT as soon as possible they not only miss much opportunities to overlap the pre-executed I/O accesses with the computation but also induce many I/O resource competitions to both local process and the whole system. In the worst scenario, all MT and PT I/O accesses collide, which results in the number of simultaneous I/O accesses in the system quite higher than that under normal execution. The competitions among large number of concurrent I/O accesses will result in the system aggregate bandwidth decreasing even I/O congestion. To our best knowledge, among both the pre-execution prefetching and the traditional prefetching works, so far there is no I/O prefetching-related work paying attention to the relationship between the pre-executed and normal executed I/O accesses.

In this work we propose a new method to overcome this drawback in the existing pre-execution prefetching works by actively scheduling the I/O accesses launched by MTs and PTs. As a preliminary step of this work, study [18] first introduces I/O scheduling into the normal and pre-executed I/O accesses in I/O prefetching direction. However, it only schedules the I/O accesses within a process, which is not enough to fully avoid the impact on normal I/O accesses induced by pre-executed I/O accesses in the whole-system scope. On the other hand, the proposed new method schedules the I/O accesses both in intra-process and the whole-system scopes.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 describes the motivation of this work. Section 4 presents pre-execution prefetching with I/O scheduling (PPIOS). Section 5 details the experiment designs and results. Section 6 concludes this paper.

2 Related work

In order to improve I/O performance quite a number of works have been done. During the last decade, a remarkable advancement in I/O parallelism has been achieved. Concretely, improvement takes place in disk-level parallelism, file-system level par-

allelism (e.g., parallel file systems such as Lustre file system [19], PVFS [20] and GPFS [21]) and application-level parallelism [22]. Elevated degree of I/O parallelism accomplishes the highly increased I/O throughput. 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. When it comes to the case in which a parallel application has a large number of isolated or small I/O accesses the effectiveness is significantly reduced. Also, to the application-level I/O parallelism the speedup is not satisfying [17,23] and file system configuration dependent. Even worse, high application-level I/O concurrency may also increase the file system workload and aggravate the I/O resources competition, which will even decrease the I/O performance conversely.

The Adaptable IO System (ADIOS) [24–26] and non-blocking I/O [27] can gain a high I/O performance improvement but they require application modification [27,28]. And the non-blocking I/O work [27] only looks at write operations; how to deal with read operation well is a challenge for non-blocking I/O.

Studies [29–32] have shown that many parallel I/O accesses are small, non-contiguous, and irregular. Motivated by this observation, a number of strategies, such as collective I/O [1,33] and data sieving [30,33], have been proposed to combine small I/O requests into large ones. By reducing the number of I/O requests and transferring more data per request these approaches can improve the I/O performance significantly in ideal cases. However, their effectiveness is application dependent. Due to the inherent nature of applications, there are still many small I/O requests that cannot be eliminated [16].

I/O prefetching for hiding I/O latency has been extensively studied [6–15]. However, these traditional prefetching strategies are conservative and most of them cannot guarantee the prefetching accuracy [16]. Incorrect prefetching not only results in the cache resource waste but also increases the I/O workload of system, which will affect the system I/O performance conversely. Although the compiler-based prefetching approach [11,13] can accomplish high accurate prefetching, it cannot deal well with the irregular applications [34], and many parallel applications are irregular.

Considering that computing power is plenty but I/O access performance is the bottleneck; recently, more and more attention has been paid to trading the excessive computing power with I/O access. Data compression and I/O pre-execution prefetching are two active research topics in this direction.

Studies [35] and [36] use data compression scheme to reduce I/O activities. 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.

Chen et al. [16] first proposed the pre-execution prefetching approach to mask the I/O access with computation. In this approach, a helper thread is created and dedicated to read operations and runs ahead of main thread. Experimental results in [16] verify that pre-execution I/O prefetching approach is effective in reducing I/O access latency. Besides, it is quite practicable because the original applications can be converted to pre-execution prefetching version automatically using a pre-compiler [16]. In order to overcome the limitation of Chen's work due to read after write (RAW) dependency and further extend the computation and I/O concurrency, Zhao et al. [17] proposed

a parallel pre-execution prefetching (PPP) approach. Concretely, it employs a new thread to conduct each dependent read prefetching. Thus, the original pre-execution prefetching thread does not need to stall to wait any write operation to be finished by the main thread. When RAW dependency is encountered there are multiple pre-execution prefetching threads progressing synchronously, which achieves the parallel pre-execution prefetching.

3 Motivation

3.1 Pre-execution I/O prefetching

Chen et al. [16] proposed a pre-execution prefetching approach (PP) to hide the I/O latency of parallel applications. 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.

As Fig. 1 shows, the pre-execution prefetching is conducted via PT. Each original process is transformed into a MT. To generate pre-execution prefetching code, the original parallel application source code is transformed either by a source-to-source pre-compiler or by the programmer's intervention. PT runs ahead of MT so it can produce effective prefetching for MT.

3.2 Drawback analysis

PP aims to overlap the computation and I/O access by creating a PT to work with the MT in parallel. However, a portion of I/O accesses requested by PT may be overlapped with the MT's I/O accesses when the MT has I/O operations such as writes and those

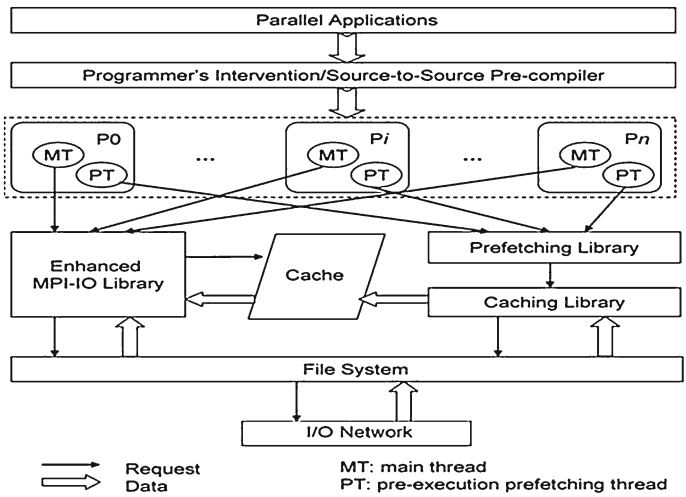

Fig. 1 Pre-execution I/O prefetching framework

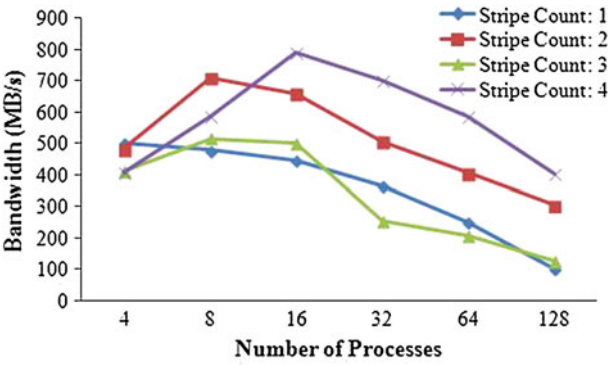

Fig. 2 Bandwidth with diverse data accessing

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 not only diminish the degree of the parallelism between computation and I/O but also induce competitions and delay the MT's I/O access. Even worse, these competitions and delay are not just limited to local MT's I/O access. They act on all MTs' I/O accesses in the system. The impact on the progress of MTs goes against the purpose of pre-execution prefetching to accelerate the execution of the original programs. And it even overwhelms the benefit brought by pre-execution prefetching when the number of processes using pre-execution prefetching is large, in which large number of concurrent I/O accesses will decrease the system aggregate sustained bandwidth and end up making the prefetcher counter-productive. Thus, I/O resource competitions introduced by PTs limit the effectiveness and scalability of the pre-execution prefetching. In the worst scenario, all MT and PT I/O accesses collide, which even results in I/O congestion.

To evaluate the relationship between aggregate sustained bandwidth and the number of concurrent I/O accesses in a system we performed an experiment on a Lustre file system detailed in Sect. 5. We conducted concurrent I/O accesses (one access per each process) on a chunk of data, which spans across different number of OSTs (stripe count) in each sub-experiment using a stripe size of 1 MB.

Figure 2 demonstrates the result with each process accessing diverse parts of data respectively. The result under all the processes accessing the same data region is shown in Fig. 3. We observed that for each case in Fig. 2 the aggregate bandwidth decreased after the number of processes is beyond a certain point. When the processes are accessing the same data region, the result is even worse. As shown in Fig. 3 the bandwidth starts to decrease when the number of processes is 2. These results confirm that when there are too many concurrent I/O accesses progressing in a system, in which the number of concurrent accesses is beyond the optimal number that system can support, the competitions among the I/O accesses will impact the system aggregate sustained bandwidth. Besides, for a small system such as ours, which has only 5 OSTs, the optimal number of concurrent I/O accesses is very small as shown in Figs. 2 and 3.

Figure 4 illustrates a program scenario and shows how this program is processed under both normal execution and PP mode, where the size of each operated segment

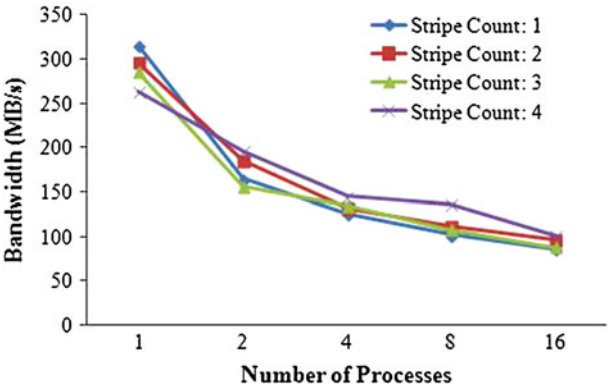

Fig. 3 Bandwidth with same data accessing

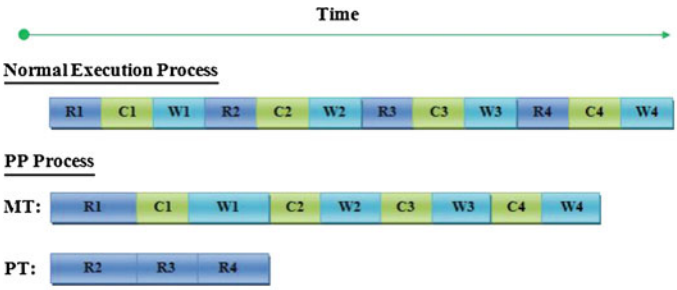

Fig. 4 Hiding I/O latency with PP. *MT* main thread, *R* read, *number after R/W/C* appearance order in original process, *W* write, *C* computation, *PT* pre-execution prefetching thread

represents time duration. This program scenario is typical in real applications such as big matrix manipulation and big file encryption where the process in the normal execution mode sequentially processes a large volume of data. For each piece of data, data reading, computation, and writing are executed in sequence. For simplicity, Fig. 4 only shows the progress situation of one process. Other processes in the system experience equally in logic. After applying PP on the original program, the process contains two threads, MT and PT. Since I/O access is the focus of PT and in this scenario dominates its execution time, we can safely ignore the time incurred by computation conducted by PT in Fig. 4. As PT is designed to do data prefetching as fast as possible, I/O overlap between PT and MT is easy to occur. Figure 4 shows that under the PP mode, a high portion of I/O accesses of PT overlap with the MT I/O accesses. They are R_2 overlapping with R_1 and R_4 overlapping with W_1. Only the I/O operation R_3 is partially overlapped with the computation of MT. In this scenario, the computation and I/O access concurrency achieved by PP are much limited. Worse, I/O resource competitions induced both by local PT and other PTs in the system make all the I/O latencies under PP mode longer, and then delay the normal execution of MT.

Considering the competitions introduced to MTs' I/O accesses by PTs and the potential to further extend the computation and I/O access concurrency, a careful

I/O scheduling scheme for normal executed and pre-executed I/O accesses for pre-execution prefetching is necessary.

4 Pre-execution prefetching with I/O scheduling

4.1 Description

Motivated by the conclusion of Sect. 3, we propose PPIOS approach to carefully schedule the I/O accesses launched by the MTs and the PTs in the system. PPIOS uses two classes of I/O scheduling: (1) intra-process I/O scheduling for scheduling MT and PT I/O accesses within a process, and (2) inter-process I/O scheduling for scheduling MT and PT I/O accesses launched by all processes in the system.

PPIOS assigns a higher priority to MTs' I/O accesses to make sure they gain the maximal system I/O resources. Under the intra-process level, within a process only one I/O access is allowed to progress at any time and MT's I/O accesses are privileged. By this design, PPIOS forces the pre-executed I/O accesses to overlap with the MT's computation, which not only extends the computation and I/O access concurrency but also avoids the I/O competition within a process. Under the inter-process level, among different processes, PTs' I/O accesses give way to MTs' I/O accesses when the number of concurrent I/O accesses is beyond a certain value. Concretely, assuming the optimal number of concurrent I/O accesses a system supports for a certain access pattern, which we call the threshold value of a system for that pattern, is n and there are m MTs' I/O accesses progressing. Then, only $n - m$ PTs' I/O accesses are allowed to progress. In this work, we assume that all I/O accesses follow the same access pattern for simplicity. In case, the total number of ongoing I/O accesses has already reached the threshold value and a new MT's I/O access is launched, one of the ongoing PTs' I/O accesses, if there are, will be chosen by PPIOS to suspend and give way to the new launched MT's I/O access. The first choice is the local PT's I/O access. The reason has been discussed at the beginning of this paragraph. If no local PT's I/O access is ongoing, the selection criteria would be cost minimizing. Figure 6 shows the working scenario of PPIOS when it forces PT I/O accesses to give way to MT I/O accesses. For comparison we show the corresponding working scenario of PP in Fig. 5.

The advantage of PPIOS over PP is illustrated in Fig. 7, which compares how PPIOS and PP progress in the scenario mentioned in Sect. 3. In Fig. 7, the size of each operated segment represents time duration. In this ideal case, PPIOS maximizes the PT I/O accesses and the MT computation concurrency. Also, it avoids the I/O competitions introduced by both local PT and other PTs in system.

In summary, the benefit of PPIOS is twofold. First, it extends the degree of computation and I/O concurrency of an application, and then further hides the I/O latency. Second, PPIOS avoids the impact on system I/O throughput introduced by PTs to achieve the best effectiveness and scalability of pre-execution prefetching approach.

The logic and action of I/O scheduling proposed by PPIOS are realizable. The intra-process I/O scheduling can be conducted in the same way as Zhao et al. [18]

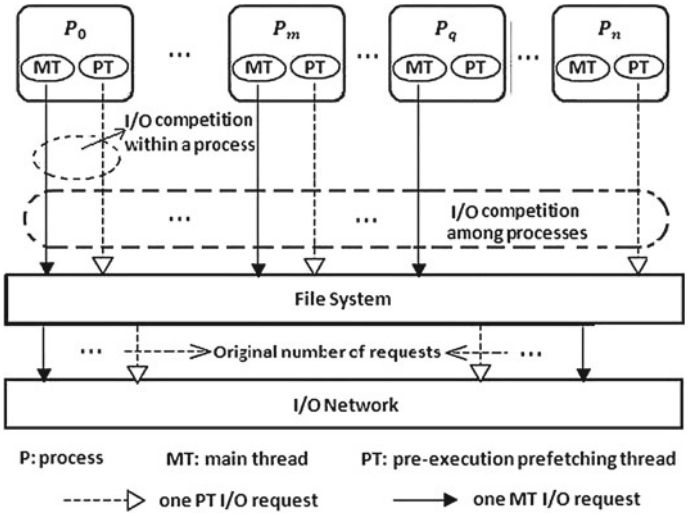

Fig. 5 I/O workflow of PP

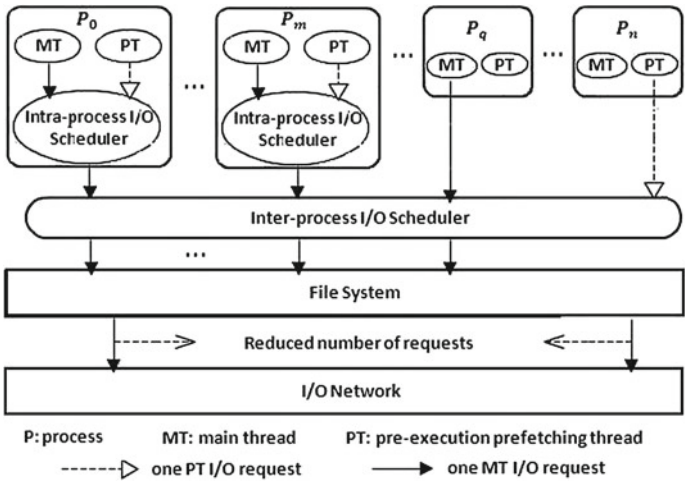

Fig. 6 I/O workflow of PPIOS

does, in which a conditional variable is employed. By extending the functionality of existing I/O schedulers the inter-process I/O scheduling proposed is easy to be implemented as well. We take the I/O scheduler proposed by Vishwanath et al. [37] as instance and briefly describe how to implement the proposed inter-process I/O scheduling on it, which is depicted in Fig. 8. In order to mitigate resource contention study [37] introduces an I/O work-queue model to control the number of ongoing I/O accesses in the system. To implement the inter-process I/O scheduling of PPIOS on this work-queue model, when en-queuing a job we assign a higher priority to the MT's I/O accesses than those launched by PTs. I/O request APIs are updated by appending an additional parameter indicating the category (MT or PT) of the thread,

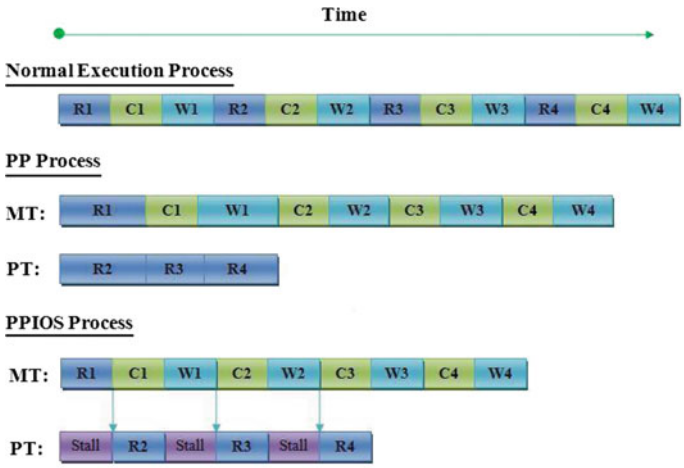

Fig. 7 Hiding I/O latency with PP and PPIOS. *MT* main thread, *C* computation, *R* read, *W* write, *PT* pre-execution prefetching thread, *arrows* wake up notification, *number after R/W/C* appearance order in original process

which launches the I/O request. The number of I/O accesses in the executing pool is equal to the system threshold value and it can be informed via an environment variable during job submission as done in [37].

4.2 Dependency consideration

In pre-execution I/O prefetching approach, PT runs a fragment of code ahead of MT to prefetch data into the prefetching cache in advance. It is possible that the PT's I/O access relies on previous writes from MT. In order to address this RAW dependency issue we adopt the method proposed in [17]. Concretely, a waiting thread (WT) is created by PT for each dependant I/O access of PT. And the PT itself skips current dependent access and keeps running ahead to prefetch future reference files. The scheduler controls the number of concurrent prefetching threads according to the current I/O workload and system resources. Higher priority is assigned to nearer used data prefetching.

4.3 Conceptualization

As a preliminary step for implementing the PPIOS, we have conceptualized it. 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 9 shows the software stack we use to implement PPIOS.

We preliminarily implement the I/O scheduling action of PPIOS in the user application layer by employing inter-thread communication and several global variables visible for all processes in the system.

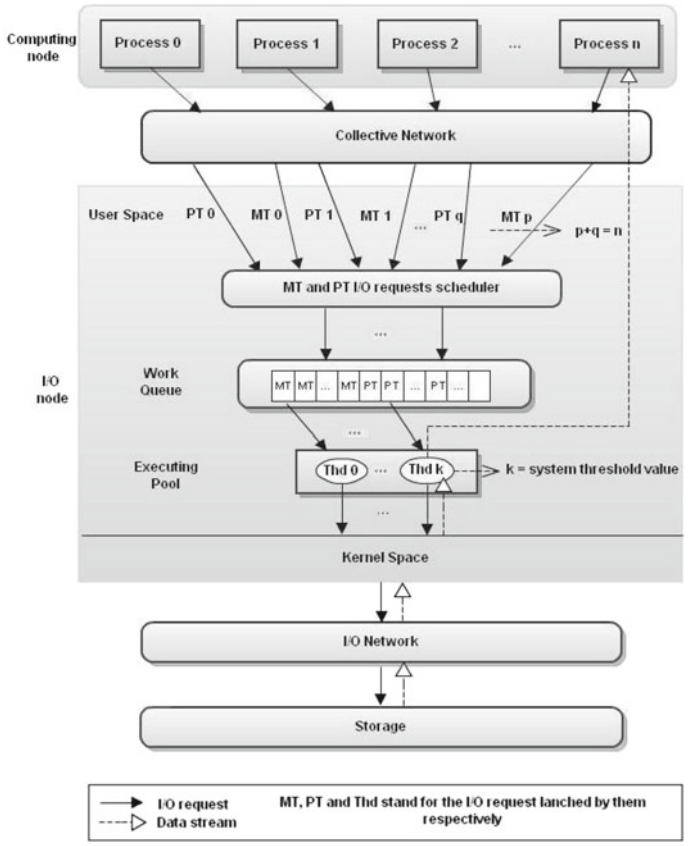

Fig. 8 Inter-process I/O scheduling implement mechanism

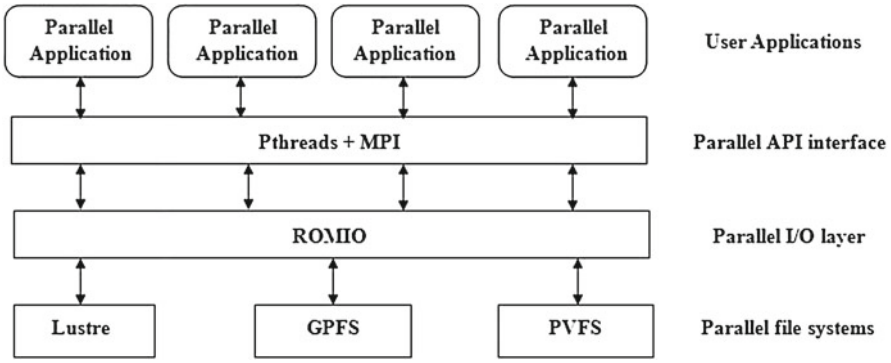

Fig. 9 Software stack

Intra-process I/O scheduling We employ a condition variable and Pthreads inter-thread message passing mechanism to accomplish the intra-process 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.

In order to implement the preempting logic of MT's I/O access over PT's I/O access, we split the original PT's I/O access into sequentially executed sub-accesses (the access size of each sub-access is far less than the access size of most MT's I/O accesses), which does not impact the prefetching rate proved by our test. By blocking a certain sub-access of PT's I/O access we achieve that MT's I/O access preempts the original PT's I/O access.

Inter-process I/O scheduling By employing global variables, which indicates the number of related I/O accesses and can be accessed by all processes, we make it visible to all processes the number of ongoing MT's I/O accesses and PT's I/O accesses in the system, respectively. Based on this information and the system threshold value each process controls its prefetching according to the policy described in Sect. 4. The global variables can be kept by a small file located either in memory or storage. The preempting of MT's I/O access over PT's I/O access is implemented in the same way stated in former subsection. Thus, we implement the scheduling of normal executed and pre-executed I/O accesses in the whole-system scope.

4.4 Analysis

Optimal analysis PP introduces extra competitions to MTs' I/O accesses of the original processes. In the worst scenario, all MT and PT I/O accesses collide and result in I/O congestion. Then, the original processes execution can be delayed infinitely. By introducing I/O scheduling PPIOS not only avoids this issue but also further extends the degree of I/O access and computation concurrency of a parallel application. In this section, we analyze the optimal speedup achieved by PPIOS over normal execution. Table 1 describes the notations.

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

Table 1 Notations

N_C	The number of segments of computation in the original process
N_W	The number of segments of write operation in the original process
N_R	The number of segments of read operation in the original process
T_{Normal}	The total execution time of the process under normal execution
T_{PPIOS}	The total execution time of the process under PPIOS 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., $N_{PT\_R\_PPIOS}$ refers to the number of segments of read conducted by PT under PPIOS mode)

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

Here, $Mode \in \{Normal, PP, PPIOS\}$. Under normal execution mode MT conducts all the operations exactly identical to how the original process does. And under PPIOS all write and computation operations are conducted by MT with the same progress time as those under normal execution mode, which is true when PPIOS completely avoids the disturbance introduced by prefetching to the I/O accesses of all MTs in the system. Thus,

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

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

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

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

The speedup achieved by PPIOS over the normal execution mode is

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

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

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

With

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

So,

$$Speedup_{PPIOS/Normal} \leq 2 \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal} \Big/ \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal} = 2 \tag{8}$$

Namely,

$$\max \left\{ Speedup_{(PPIOS/Normal)} \right\} = 2 \tag{9}$$

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

The 2-time speedup can be also achieved solely by intra-process I/O scheduling under the ideal circumstance as shown in [18], except that in practical system, intra-process I/O scheduling is still insufficient to eliminate the cost of other processes' I/O access rate and the decrease of system I/O throughput induced by aggressive prefetching. Intra-process I/O scheduling can only avoid the impact on MT's normal I/O requests induced by pre-execution prefetching within single-process scope. Thus, the prefetching activities of local process may still introduce competitions to normal I/O requests of other processes. On the other hand, inter-process I/O scheduling coordinates scheduling of prefetching I/O requests and normal I/O requests among processes in a distributed system to help optimize the progress order of the normal I/O and pre-execution I/O requests issued by diverse processes. This feature not only guarantees assigning the limited system I/O resource to the more urgently demanded I/O request further but also avoids the system I/O throughput cost caused by pre-execution prefetching.

Worst case analysis Once MT's I/O requests are continuously launched the PT's I/O requests will encounter starvation. This is consistent to the system resource assignment requirement. In case the system instantaneous I/O workload is high I/O requests prefetching the future accessed data should give way to the demanded I/O requests asking for immediately needed data. This prefetching request starvation will not affect the normal execution of its master process. Failing to find the demanded data in memory buffer the process will issue normal I/O requests through MT. Thus, in the worst case the speedup of PPIOS is 1.

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 [16], PPIOS requires an I/O scheduling, which is quite light-weight in cost. Throughout our implementation of the I/O scheduling, only several variables and 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 PPIOS, which is also verified by the performance improvement achieved by PPIOS as shown in Sect. 5.

Correctness First, the existing pre-execution prefetching approach can guarantee the correctness of the original programs. Second, the I/O scheduling introduced by PPIOS only reschedule the I/O accesses time, so it does not affect the logical behavior and accuracy of MT. Thus, the MT in the system running with and without PPIOS will logically behave identically. In summary, PPIOS does not affect the correctness of the original programs.

5 Experiment

5.1 Platform

Our experiments were conducted on a 66-node 528 processors Linux-based cluster. This cluster is composed of 1 frontend node that runs TORQUE resource manager and Moab scheduler, 1 login node and 64 compute nodes. Each compute node has 16 GB of RAM and 2 CPU sockets, each with quad core Intel Xeon 2.66 GHz 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. File system is Lustre parallel file system. There are five storage targets (OST), each with a RAID5 set with 8 internal 1TB disks. In this experiment we striped the input files across 4 OSTs with stripe size being 1 MB. 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 Fig. 9.

5.2 Benchmarks

In order to evaluate the effectiveness of PPIOS, we measured its performance on four benchmarks shown in Table 2. We chose these benchmarks because they are representative kernels from data statistical and data mining applications.

5.3 Design and results

Based on the chosen benchmarks we designed two experiments. In the first experiment, we evaluated the benefits of PPIOS on single benchmark accessing the file system. To simulate the real HPC working scenario, in which there are multiple jobs being conducted synchronously, in the second experiment we ran four benchmarks

Table 2 Benchmarks

Name	Description
SUM	Data statistical operation that reads the input file and returns the sum of all its items
AVG	Data statistical operation that reads the input file and returns the average of all its items
DSCAL	Data mining operation that multiplies every item in the input file by a scalar [38]
GREP	Data mining operation that searches matching string patterns from the input file

concurrently. All the benchmarks processed the input file chunk by chunk with the chunk size being 1 GB except for special statement. For each benchmark the input file is 6 GB in size and the data type is integer. We compared the experimental results measured under normal execution, PP and PPIOS mode.

Experiment #1: Single benchmark running in the system We selected benchmarks SUM and GREP to execute separately as a single benchmark accessing the file system. For GREP the item of string is an integer number and the length of string pattern to search is 30.

Figure 10 shows the execution time results of benchmarks SUM and GREP, respectively. The execution time under PPIOS mode is reduced by up to 42.6 and 20.9 % over normal execution and PP mode, respectively. As a reference, they also show the application's execution time under the theoretically optimal scenario, in which the I/O latency would be completely masked. 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 PPIOS mode can be achieved with moderate number of parallelisms as observed in [17]. Due to the I/O competitions introduced by PP when the number of processes is large the execution time under PP even overwhelms that of the normal execution. PPIOS, on the other hand, can achieve execution time reduction compared to the other two modes in all cases.

Figure 11 shows the results of I/O latency during the whole execution of the benchmarks. In most of cases, a considerable I/O latency reduction percentage has been achieved by PPIOS over the other two modes, which is up to 83.3 % over normal execution mode and 42.7 % over PP mode. When the number of processes is equal to

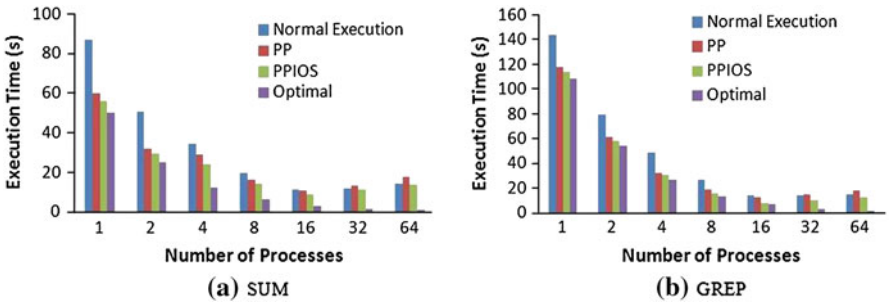

Fig. 10 Execution time: **a** SUM, **b** GREP

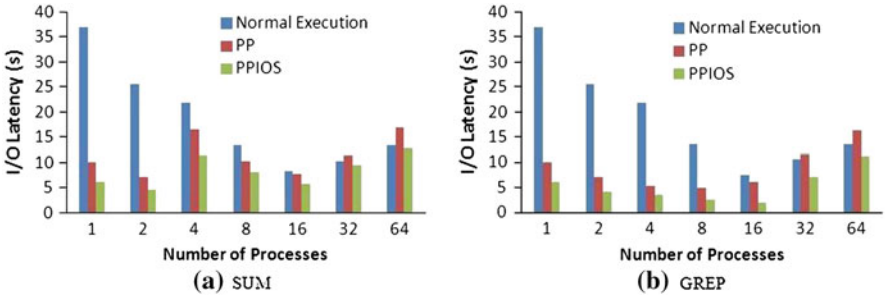

Fig. 11 I/O latency: **a** SUM, **b** GREP

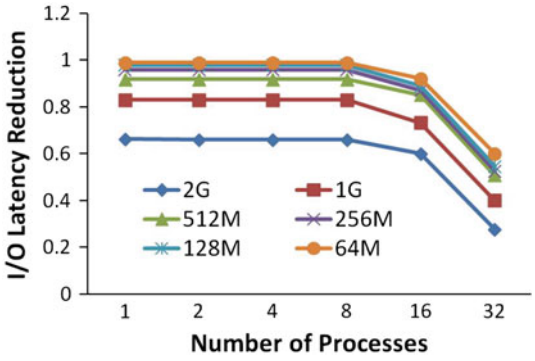

Fig. 12 I/O latency reduction of GREP

4 and 32 in benchmark SUM and GREP, respectively, the I/O latency starts to dominate the execution time of process. Thus, the percentage of I/O latency that can be hidden by computation decreases, which results a zoom of I/O latency under both PP and PPIOS mode. Most importantly, the PPIOS 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.

Figure 12 shows the I/O latency reduction for GREP achieved by PPIOS over normal execution mode as the chunk size changes. In some cases, the I/O latency reduction is close to 100 %. It demonstrates that in these cases PPIOS can almost completely hide the I/O latency suffered by GREP by scheduling the pre-executed I/O operation to strictly overlap with computation. When the number of processes is larger than or equal to 16, the reduction drops. The reason is the same with former analysis that the computation executing time percentage starts to decrease and I/O latency dominates in each process.

Experiment #2: Multiple benchmarks concurrently running in the system We ran all the four benchmarks in Table 2 concurrently with different files as input respectively. As shown in Fig. 2, under striped count being 4, for our system the highest bandwidth is achieved when the number of processes is 16 for diverse data accessing. Thus, in this experiment we set the threshold value of system as 16.

Figure 13 shows the execution time results, in which the number of processes refers to total number of processes employed, which are equally distributed to each bench-

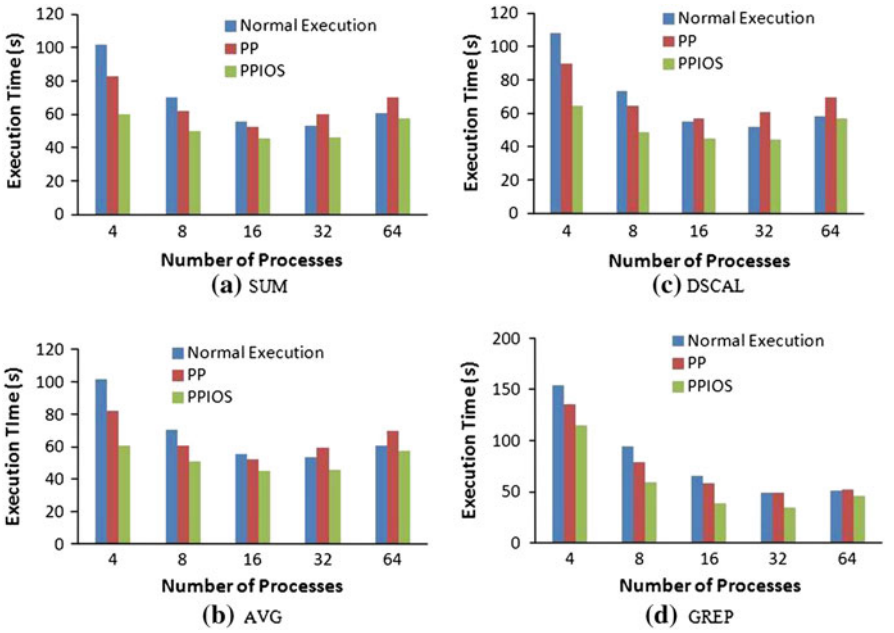

Fig. 13 Execution time: **a** SUM, **b** DSCAL, **c** AVG, **d** GREP

mark. The execution time under PPIOS mode is reduced by up to 41.1 and 27.7 % over normal execution and PP, respectively.

Figure 14 shows the results of I/O latency suffered by the benchmarks. In most of cases, a considerable I/O latency reduction percentage has been achieved by PPIOS over the other two modes, which is up to 80.7 % over normal execution mode and 29.8 % over PP mode.

For all tested benchmarks, the performance gained by PPIOS over normal execution mode shrinks when the number of processes employed is large (e.g. 32, 64). There are two reasons. First, it is due to that within each process the ratio of computation to I/O latency decreases, which results in less I/O latency can be hidden by computation. Second, under heavy I/O workload scenario PT's I/O accesses are more likely to be forced by PPIOS to give way to those launched by MTs to avoid aggravating the I/O competition in the system.

In Experiments #1 and #2, compared to PP, though PPIOS decreases the prefetching rate by sleeping some prefetching operations to give way to the I/O accesses launched by MTs, PPIOS still achieves the I/O performance improvement over PP. First reason is that it successfully hides all the pre-executed I/O accesses latency with computation within a process. Second, it avoids the impact on MTs' I/O accesses introduced by prefetching conducted both by local and other processes in the system, which guarantees the complete time of MTs' I/O accesses.

For Experiments #1 and #2 when the number of processes employed is larger (e.g. 128, 256), in the original program the I/O latency will highly dominate the execution time of each process. Take SUM in Experiment #1 for instance, when 128

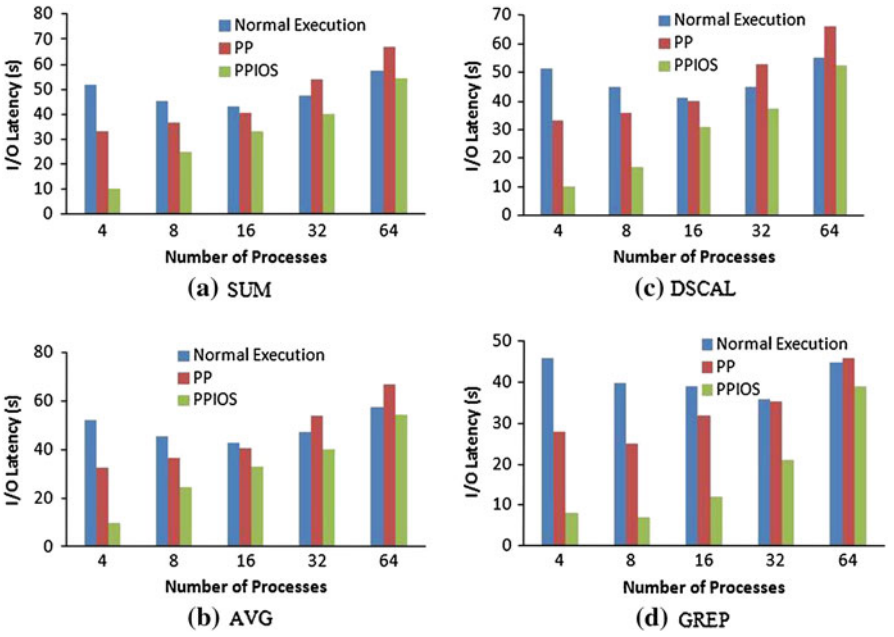

Fig. 14 I/O latency: **a** SUM, **b** DSCAL, **c** AVG, **d** GREP

processes are employed, the computing duration of each process is only about 0.36 s. It is far less than that of I/O latency, which is more than 10 s. This is true for other three benchmarks as well. Pre-execution prefetching approach is designed to hide I/O latency with computation [16] and performs well in applications with moderate ratio of I/O latency to computation. Thus, results under larger process size in above experiments for our small system are meaningless.

Results of above experiments confirm that PPIOS has more potential to hide I/O latency suffered by parallel applications and can successfully eliminate the impact on the normal executed I/O accesses introduced by prefetching in the whole-system scope. In summary, PPIOS achieves better effectiveness and scalability than the existing pre-execution prefetching.

6 Conclusion

With the rate of computing power growing much faster than that of storage I/O access, parallel applications suffer more from I/O latency. In this study, we propose the PPIOS approach to improve the I/O performance of parallel applications. 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 in both intra- and inter-process levels to maximize the overlap between the pre-executed I/O accesses and computation and eliminate the impact on normal executed I/O accesses introduced by prefetching, which is the first work to the best of our knowledge in the research direction of

I/O prefetching. The extensive evaluation results verified the benefits of the proposed approach.

Acknowledgments This work was 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 XH, Thakur R, Roth PC, Gropp W (2011) LACIO: a new collective I/O strategy for parallel I/O systems. In: Proceedings of international parallel and distributed processing symposium (IPDPS). IEEE, New York, pp 794–804
2. Sun X-H, Chen Y, Wu M (2005) Scalability of heterogeneous computing. In: Proceedings of 34th international conference on parallel processing
3. Liu N, Fu J, Carothers CD (2010) Massively parallel I/O for partitioned solver systems. Parallel Process Lett 6:1–17
4. Kesavan M, Gavrilovska A, Schwan K (2010) On disk I/O scheduling in virtual machines. In: WIOV '10, March 2010
5. Ali N, Carns PH, Iskra K, Kimpe D, Lang S, Latham R, Ross RB, Ward L, Sadayappan P (2009) Scalable I/O forwarding framework for high-performance computing systems. In: CLUSTER. pp 1–10
6. Ding X, Jiang S, Chen F, Davis K, Zhang X (2007) DiskSeen: exploiting disk layout and access history to enhance I/O prefetch. In: Proceedings of USENIX annual technical conference
7. Kotz DF, Ellis CS (1990) Prefetching in file systems for MIMD multiprocessors. In: IEEE transactions on parallel and distributed systems, vol 1, no 2
8. May J (2001) Parallel I/O for high performance computing. Morgan Kaufmann Publishing, Los Altos
9. Papathanasiou A, Scott M (2005) Aggressive prefetching: an idea whose time has come. In: Proceedings of the 10th workshop on hot topics in operating systems
10. Patterson RH (1997) Informed prefetching and caching. Carnegie Mellon Ph.D. Dissertation CMU-CS-97-204
11. Son SW, Kandemir M, Karakoy M, Chakrabarti D (2009) A compiler-directed data prefetching scheme for chip multiprocessors. In: Proceedings of the 14th symposium on principles and practice of parallel programming. pp 209–218
12. Ravichandran N, Paris JF (2005) Making early predictions of file accesses. In: Proceedings of 4th International Inf. Telecommun. Technol. pp 122–129
13. Brown AD, Mowry TC, Krieger O (2001) Compiler-based I/O prefetching for out-of-core applications. ACM Trans Comput Syst 19(2):111–170
14. Seelam S, Chung IH, Bauer J, Wen HF (2010) Masking I/O latency using application level I/O caching and prefetching on Blue Gene systems. In: Proceedings of IEEE international symposium on parallel distributed processing (IPDPS). pp 1–12
15. He J, Sun X-H, Thakur R (2012) KNOWAC: I/O prefetch via accumulated knowledge. In: Proceedings Of IEEE international conference on cluster computing. pp 429–437
16. Chen Y, Byna S, Sun XH, Thakur R, Gropp W (2008) Hiding I/O latency with pre-execution prefetching for parallel applications. In: Proceedings of SC 2008. pp 1–10
17. Zhao Y, Yoshigoe K (2012) Hiding I/O latency with parallel pre-execution prefetching. In: Proceedings of the 24th IASTED international conference on parallel and distributed computing and systems (PDCS 2012), November 2012. pp 162–169
18. Zhao Y, Yoshigoe K, Xie M (2013) Pre-execution data prefetching with inter-thread I/O scheduling. In: Proceedings of the 2013 international supercomputing conference. Lecture notes in computer science (LNCS), vol 7905. Springer, Berlin, pp 395–407
19. Schwan P (2003) Lustre: building a file system for 1000-node clusters. In: Proceedings of Linux. Symposium, July 2003
20. Ligon W, Ross R (2003) Parallel I/O and the parallel virtual file system. In: Beowulf cluster computing with Linux. MIT Press, Cambridge, pp 493–534
21. Schmuck F, Haskin R (2002) GPFS: a shared-disk file system for large computing clusters. In: Proceedings of the 1st USENIX conference on file and storage technologies
22. Chen Y, Byna S, Sun X-H, Thakur R, Gropp W (2008) Exploring parallel I/O concurrency with speculative prefetching. In: Proceedings of 37th international conference on parallel processing (ICPP 08)

23. Margo MW, Kovatch PA, Andrews P, Banister B (2004) An analysis of state-of-the-art parallel file systems for linux. In: The 5th international conference on Linux clusters: the HPC revolution 2004. Austin, TX
24. Lofstead JF, Klasky S, Schwan K, Podhorszki N, Jin C (2008) Flexible io and integration for scientific codes through the adaptable io system (adios). In: Proceedings of the 6th international workshop on Challenges of large applications in distributed, environments. pp 15–24
25. 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 (2008) Adaptive io system (adios). Cray Users Group
26. Lofstead J, Klasky S, Booth M, Abbasi H, Zheng F, Wolf M, Schwan K (2009) Petascale io using the adaptable io system. Cray Users Group
27. Buettner D, Kunkel J, Ludwig T (2009) Using non-blocking I/O operations in high performance computing to reduce execution times. Recent advances in parallel virtual machine and message passing interface. Lecture notes in computer science, vol 5759. pp 134–142
28. Bent J, Gibson G, Grider G, McClelland B, Nowoczynski P, Nunez J, Polte M, Wingate M (2009) Plfs: a checkpoint filesystem for parallel applications. In: Proceedings of conference on high performance computing, networking, storage and analysis (SC' 2009)
29. Kotz DF, Nieuwejaar N (1994) Dynamic file-access characteristics of a production parallel scientific workload. In: Proceedings of Supercomputing'94. pp 640–649
30. Reed D (2003) Scalable Input/Output: achieving system balance. The MIT Press, Cambridge
31. Madhyastha TM, Reed DA (2002) Learning to classify parallel Input/ Output access patterns. In: Proceedings of IEEE transactions on parallel and distributed systems, vol 13, no 8
32. Smirni E, Reed DA (1997) Workload characterization of Input/Output intensive parallel applications. In: Proceedings of the 9th international conference on computer performance evaluation: modeling techniques and tools
33. Thakur R, Gropp W, Lusk E (1999) Data sieving and collective I/O in ROMIO. In: Proceedings of the 7th symposium on the frontiers of massively parallel computation
34. Solihin Y, Lee J, Torrellas J (2002) Using a user-level memory thread for correlation prefetching. In: Proceedings of the 29th annual international symposium on computer architecture (ISCA), Anchorage. Alaska, May 2002
35. Makatos T, Klonatos Y, Marazakis M, Flouris MD, Bilas A (2010) Using transparent compression to improve SSD-based I/O caches. In: Proceedings of the 5th European conference on computer systems, EuroSys 10, NY, USA. ACM, New york, pp 1–14
36. Welton B, Kimpe D, Cope J, Patrick C, Iskra K, Ross R (2011) Improving I/O forwarding throughput with data compression. In: International conference on cluster computing, CLUSTER '11. IEEE, New York, pp 438–445
37. Vishwanath V, Hereld M, Iskra K, Kimpe D, Morozov V, Papka ME, Ross RB, Yoshii K (2010) Accelerating i/o forwarding in ibm blue gene/p systems. In: SC. pp 1–10
38. Piernas J, Nieplocha J, Felix EJ (2007) Evaluation of active storage strategies for the lustre parallel file system. In: Proceedings of Supercomputing, 2007 (SC '07)