Server-Side I/O Coordination for Parallel File Systems

Huaiming Song [†*], Yanlong Yin [†], Xian-He Sun [†], Rajeev Thakur [‡], Samuel Lang [‡]

[†]Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, USA
[‡]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA
{huaiming.song, yyin2, sun}@iit.edu, {thakur, slang}@mcs.anl.gov

ABSTRACT

Parallel file systems have become a common component of modern high-end computers to mask the ever-increasing gap between disk data access speed and CPU computing power. However, while working well for certain applications, current parallel file systems lack the ability to effectively handle concurrent I/O requests with data synchronization needs, whereas concurrent I/O is the norm in data-intensive applications. Recognizing that an I/O request will not complete until all involved file servers in the parallel file system have completed their parts, in this paper we propose a server-side I/O coordination scheme for parallel file systems. The basic idea is to coordinate file servers to serve one application at a time in order to reduce the completion time, and in the meantime maintain the server utilization and fairness. A window-wide coordination concept is introduced to serve our purpose. We present the proposed I/O coordination algorithm and its corresponding analysis of average completion time in this study. We also implement a prototype of the proposed scheme under the PVFS2 file system and MPI-IO environment. Experimental results demonstrate that the proposed scheme can reduce average completion time by 8% to 46%, and provide higher I/O bandwidth than that of default data access strategies adopted by PVFS2 for heavy I/O workloads. Experimental results also show that the server-side I/O coordination scheme has good scalability.

Categories and Subject Descriptors

B.4.3 [**Interconnections**]: Parallel I/O; D.4.3 [**File Systems Management**]: Access methods

Keywords

server-side I/O coordination; parallel I/O synchronization; I/O optimization; parallel file systems

1. INTRODUCTION

Large-scale data-intensive supercomputing relies on parallel file systems, such as Lustre [1], GPFS [22], PVFS [9], and PanFS[18] for high-performance I/O. However, performance improvements in computing capacity have vastly outpaced the improvements in I/O performance in the past few decades and will likely continue in the future. Many high-performance computing (HPC) applications have become "I/O bounded", unable to scale with increasing compute power. The gap in performance between compute and I/O is amplified further when multiple applications compete for limited I/O and storage resources at the same time, as this leads to thrashing scenarios within the HPC storage system. Parallel file systems have difficulty handling I/O workloads of multiple applications for two primary reasons. First, the file servers perform data accesses in an interleaved fashion, resulting in excessive disk seeks. Second, file servers perform I/O requests independently, without knowledge of the order of requests performed at other servers, whereas HPC applications tend to coordinate I/O across all processes. This scenario leads to under-utilization of compute resources, as all compute processes are held waiting for completion of an I/O request that is delayed by the interleaved scheduling choices made by an individual file server.

In general, data files are striped across all or a part of the file servers in parallel file systems. One I/O request issued from a single client often involves data accesses on multiple servers, and the parallel I/O library has to merge the multiple data pieces from these file servers together. Moreover, collective data access from multiple clients, such as collective I/O in MPI-IO [26], has to wait for all aggregators to complete. Synchronization of I/O requests across processes is common in parallel computing, and can be classified into the following three categories (as shown in Figure 1).

- **Intra-request Synchronization**: One I/O request issued by one client accesses data in multiple file servers. It needs to gather/scatter data pieces from/to multiple storage nodes and merge them together to complete the I/O request.
- **Collective I/O Synchronization**: Multiple I/O clients access data from multiple file servers collectively within a single application. It has to wait for all aggregators to complete their collective I/O operations before continuing.
- **Inter-request Synchronization**: Multiple clients access data from a parallel file system independently, and there is explicit synchronization among these I/O clients.

Figure 1 shows the three scenarios of data synchronization. The first two categories are implicit synchronization and the third one is explicit. In a large-scale and high-performance computing system, the parallel file system is often shared by multiple applications. When these applications run simultaneously, each file server may

*This author has now joined R&D center, Dawning Information Industrial LLC, Beijing, China. Email: songhm@sugon.com

SC11, November 12-18, 2011, Seattle, Washington, USA

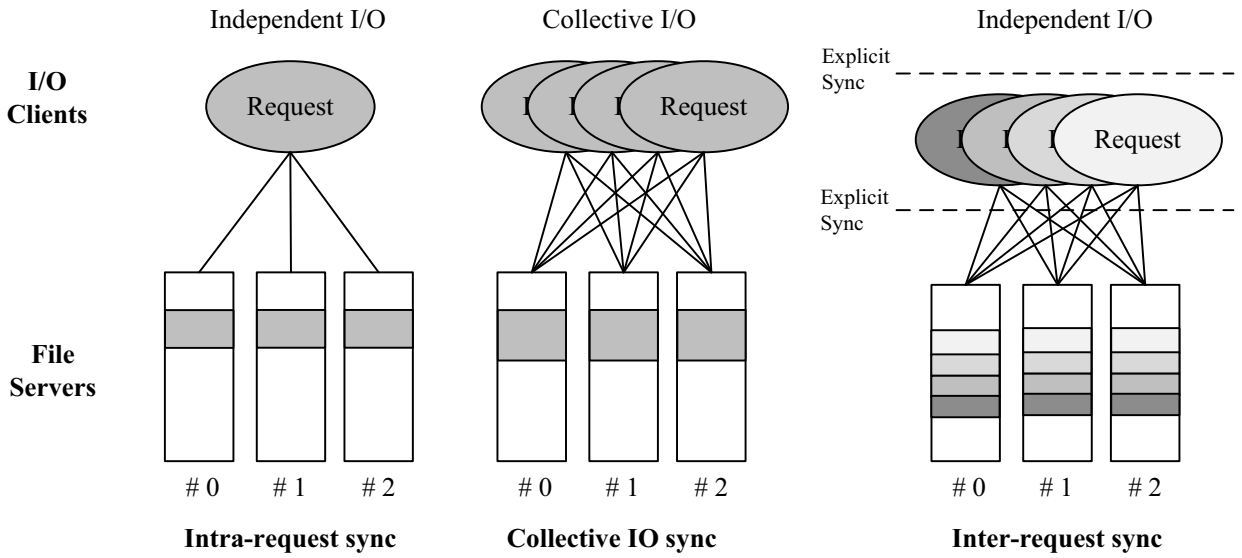


Figure 1: Three scenarios of data synchronization in parallel I/O

receive multiple I/O requests from different applications. However, these requests are likely to be served in different order on different file servers because they are scheduled independently. Figure 2 is an example of the I/O request scheduling in 4 file servers, and there are 3 applications: A, B and C. Usually, the completion time of each application depends on the completion time of the last file server to finish the request. In the left subfigure, all nodes serve the requests in different orders. The completion times of the I/O requests from the three applications are: $T_A = 4t$; $T_B = 4t$; $T_C = 4t$. Thus the average completion time is: $T_{avg} = 4t$. If we re-arrange the requests in the file servers, letting all nodes service the requests in the same order, as shown in right part of Figure 2, the completion times are: $T_A = 2t$; $T_B = 3t$; $T_C = 4t$. The average completion time is: $T_{avg} = 3t$. In other words, after requests re-ordering, the average completion time decreases from $4t$ to $3t$, which reveals a significant potential for shortening average completion time through request re-ordering at the file servers.

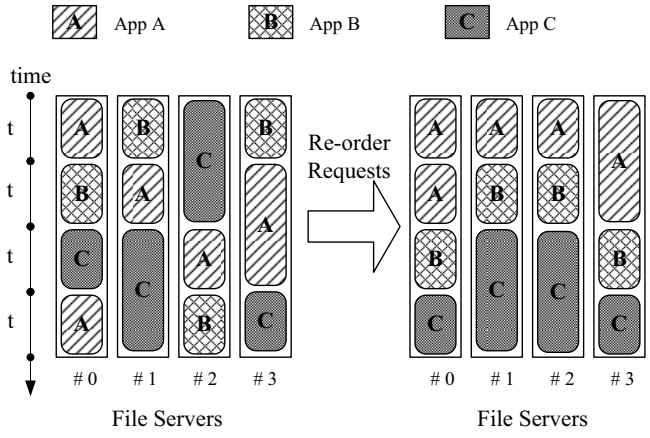


Figure 2: Order of request handling affects completion time. In the left subfigure, service order is different on different file servers, and the average completion time for the three applications is $4t$. While in the right subfigure, requests are serviced in concert, and the average completion time reduces to $3t$.

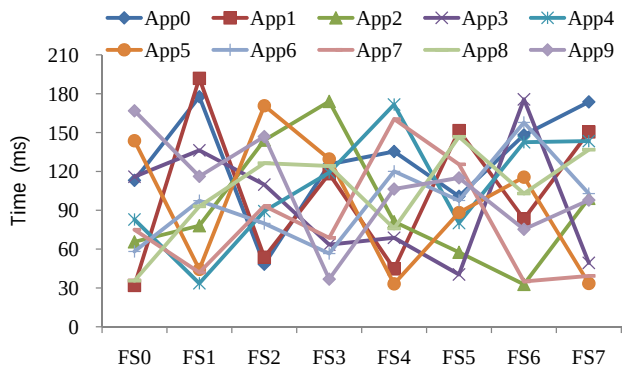
Existing scheduling algorithms in parallel file systems, such as disk-directed I/O [13], server-directed I/O [23], and stream-based I/O [11, 21], focus on reducing data access overhead on either storage nodes or network traffic, to improve throughput of each file server. These approaches have demonstrated the importance of scheduling in parallel file systems to improve performance. However, little attention has been paid to server-side I/O coordination in order to reduce average completion time of multiple applications competing for limited I/O resources. In this paper, we propose a new server-side I/O coordination scheme for parallel file systems that enables all file servers to schedule requests from different applications in a coordinated way, to reduce the synchronization time across clients for multiple applications.

The contribution of this paper is four-fold. First, we present the data synchronization problems in parallel file systems. Second, we propose an effective server-side I/O coordination scheme for parallel I/O systems to reduce the average completion time of I/O requests, and thus to alleviate the performance penalties of data synchronization. Third, we implement a prototype of the I/O coordination scheme in PVFS2 and MPI-IO. Finally, we evaluate the proposed scheme both analytically and experimentally.

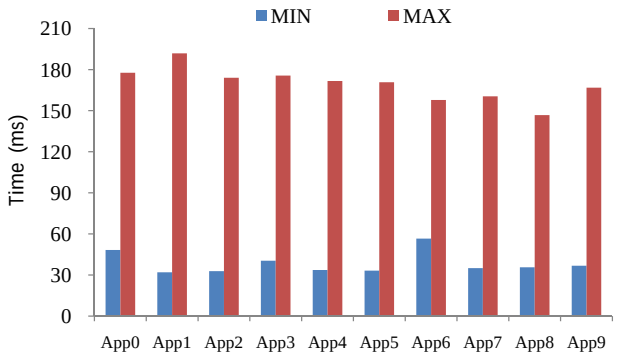
The remainder of this paper is organized as follows. Section 2 examines the overhead of data synchronization without I/O coordination. Section 3 describes the design of I/O coordination algorithm and gives an analysis of completion time. Section 4 presents the implementation of the proposed I/O scheme in PVFS2 and MPI-IO. Experimental and analytical results are discussed in Section 5. Section 6 reviews related work in server-side I/O scheduling and parallel job scheduling. Finally, Section 7 concludes this study and discusses potential future work.

2. THE IMPACT OF DATA SYNCHRONIZATION

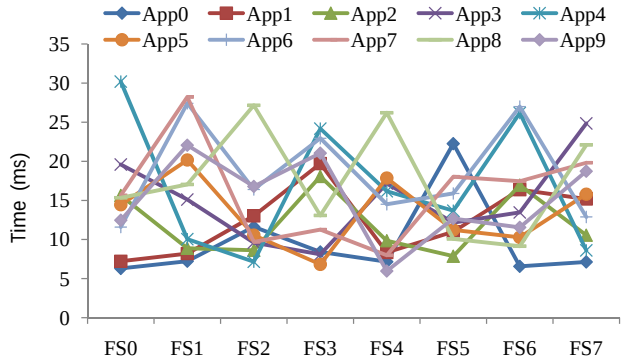
Data synchronization is common in parallel file systems, where I/O requests usually consist of multiple pieces of data access in multiple file servers and will not complete until all involved servers have completed their parts. However, due to independent scheduling strategies on file servers, I/O requests with synchronization



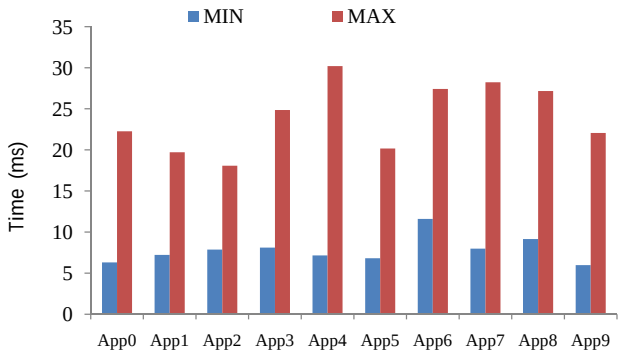
(a) Finish time on different file servers (HDD)



(b) Minimum and maximum finish time (HDD)



(c) Finish time on different file servers (SSD)



(d) Minimum and maximum finish time (SSD)

Figure 3: The finish time of I/O requests from different applications on different file servers. This set of experiments were intra-request synchronization scenario with 10 concurrent IOR instances and a 8-node PVFS2 system. The stripe size of PVFS2 was 64KB, and each IOR instance issued a 4MB contiguous read request to the PVFS2 system. Thereby every request involved all 8 file servers, and the size of requested data on one file server was 512KB. 'App$K' (k=0∼9) refers to an IOR instance, 'FS$N' (N=0∼7) refers to a file server. 'MIN' refers to the finish time of first complete file server, and 'MAX' refers to the finish time of last complete file server. The completion time of each application relies on the 'MAX' finish time for that application on all involved file servers.

needs from different applications are very likely to be served in different orders on different file servers.

Understanding the impact of data synchronization in parallel I/O systems is critical to efficiently improving completion time. In this section, we evaluate the request completion time when file servers serve requests from multiple applications simultaneously. We employed 8 nodes for PVFS2 file servers. Each file server was installed with a 7200RPM SATA II 250GB hard disk drive (HDD), a PCI-E X4 100GB solid state disk (SSD), and the interconnection was 4X InfiniBand. We adopted the IOR benchmark to simulate the intra-request synchronization scenario and measured the finish time of all requests on different file servers. The number of concurrent IOR instances was 10, to simulate 10 concurrent applications. In these experiments we show only the intra-request data synchronization case, so each instance was configured with only one process, which issued a 4MB contiguous data read request. Figure 3 shows the finish time of different requests on different file servers. From Figure 3 (a) and (c), we can see that, in both HDD and SSD environments, the finish time of every application varies a lot on different file servers. From subfigure (b) and (d), we can see that, the maximum finish time is 4.4 times of the minimum on average in the HDD environment and 3.1 times in the SSD environment. The completion time of one request is equal to the maximum value of all finish times on all involved file servers. Therefore, the significant deviation of finish time on multiple file servers leads to high completion time of data accesses.

The experimental results also indicate that, due to the independent scheduling strategy on each file server, data accesses are finished in different orders for concurrent applications. The difference of service orders on different file servers will become much greater in the inter-request or collective I/O synchronization cases, where each application has multiple processes. As a result, the independent scheduling strategy on file servers introduces a large number of idle CPU cycles waiting for data synchronization on computing nodes, and the case will become even worse for large-scale HPC clusters. The results also reveal that there is a significant potential to shorten completion time by coordinated I/O scheduling on file servers.

3. I/O COORDINATION

In order to reduce the overhead of data synchronization, we propose a server-side I/O coordination scheme which re-arranges I/O requests on file servers, so that requests are serviced in the same order in terms of applications on all involved nodes. Data synchronization usually explicitly or implicitly exists in parallel processes of parallel applications. The re-ordering aims at scheduling

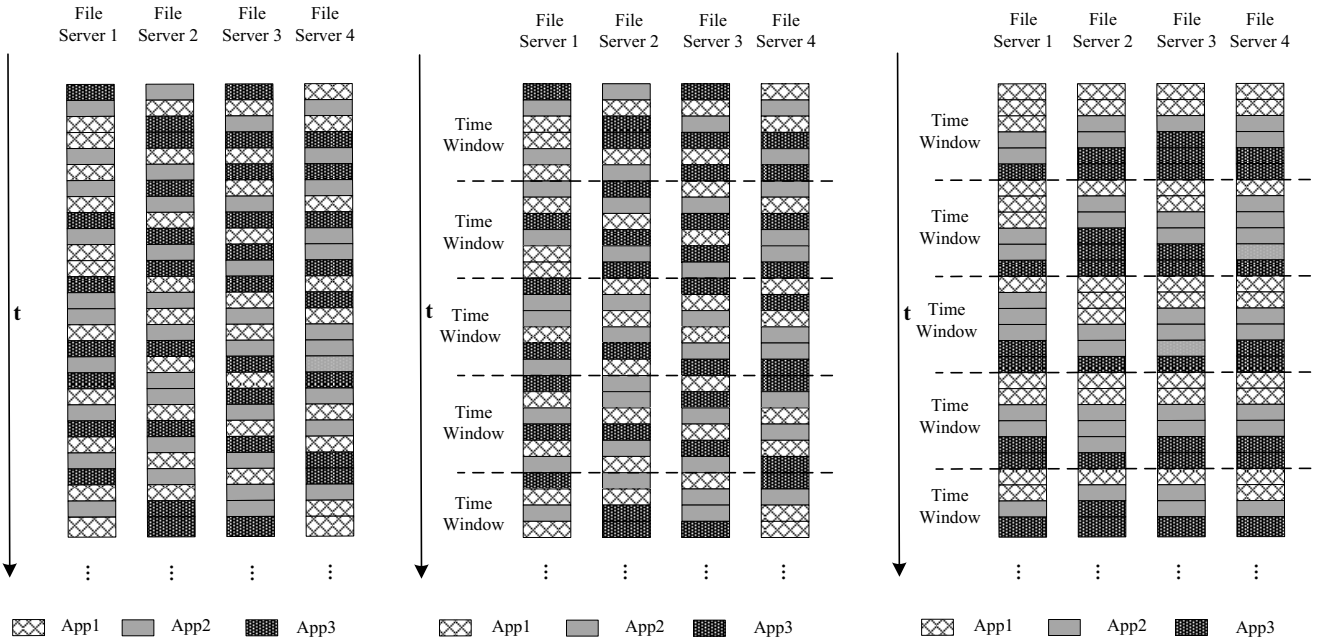


Figure 4: I/O coordination scheme in parallel file systems

the parallel I/O requests that need to be synchronized to run together, which can benefit the system with a shorter average completion time of all I/O requests. A good scheduling algorithm should take into account both performance and fairness. A good practical scheduling algorithm also requires simplicity in implementation. The proposed I/O coordination is no exception. For fairness, all I/O requests should be serviced within an acceptable period to avoid starvation. To provide the right balance of performance and fairness, the concept of '*Time Window*' and '*Application ID*' are introduced to support the server-side I/O coordination approach.

Time Window. All I/O requests issued to a file server can be regarded as a time series. The time series is then divided into successive segments by a fixed time interval. Here each segment of the time series is referred to as *Time Window*. Thus, one Time Window consists of a number of I/O requests. The value of the time interval can be regarded as *Time Window Width*.

Application ID. We allocate an integer value for each application running on the cluster. The integer is an identification of "which application an I/O request belongs to", and is referred to as *Application ID*. For each I/O request, it will pass on this integer to the file servers.

According to the definition, all I/O requests from one application have the same 'Application ID'. For applications with multiple parallel processes, such as MPI programs, there might be large amounts of data synchronization. In order to alleviate the performance penalties of synchronization, I/O requests from all processes should have the same 'Application ID', and they should be served in concert in multiple file servers. The 'Application ID' is generated automatically in the parallel I/O library and it is transparent to users. It can be implemented in parallel I/O client libraries or the middleware layer, without modifying application programs.

For fairness, requests in an earlier 'Time Window' will be serviced prior to those in a later one, to avoid starvation. The request time can use either file-server-side time (the arrival time of a request) or client-side time (the issue time of a request). Because of network latency and load imbalance issues, one client side request may have different arrival time on different file servers. In a system with many concurrent clients, a request issued earlier might get a later arrival time on some file servers. For these reasons, in our implementation, we choose client-side time as the request time.

3.1 Algorithm

It is not difficult to imagine that in a parallel file system, a large number of I/O requests might be queued on each file server at a time. These I/O requests might come from multiple applications. As all arriving requests are attached with a request time and an 'Application ID', the I/O coordination algorithm can be described as follows. In the same 'Time Window', I/O requests are ordered by the value of 'Application ID'; while in different 'Time Windows', requests in an earlier window would be serviced prior to those in a later one.

The proposed I/O scheduling algorithm is based on the observation that requests from the same application have a better locality and, equally important, the execution will be optimized if these requests finish at the same time. It takes both performance and fairness into consideration. In each time window, requests are served one application at a time in order to reduce the overhead of data synchronization. In addition, none of the requests will be starved, because requests in an earlier time window will always be performed first.

Figure 4 illustrates how the I/O coordination algorithm works in parallel file systems. In this example, there are 4 file servers and three concurrent applications. The original request arrival orders are inconsistent on different file servers, such as in subfigure (a). The series of I/O requests are split into successive 'Time Windows' by a fixed time interval on all file servers, as shown in subfigure (b). The scheduler on each file server then reorders the requests in each 'Time Window' by 'Application ID', so that requests from one application can be serviced in the same time on all file servers, as shown in subfigure (c).

The scheduler on each file server maintains a queue for all requests, which determines the service order of I/O requests. When a new I/O request arrives, if the queue is empty, the request will be scheduled immediately. If the queue is not empty, the scheduler will insert the request into the queue in terms of 'Time Window' and 'Application ID'. The scheduler keeps issuing request with the highest priority (i.e. the head of the queue) to the low-level storage devices in current queue on each file server. Since the 'Application ID' and request time are generated at the client side and then passed to the file servers, there is no communication between different file servers while scheduling the requests. The use of 'Application ID' and 'Time Window' has significantly simplified the implementation of the coordination and paved the foundation for good scalability as the number of file servers increases.

3.2 Completion Time Analysis

Assume that the number of file servers is n, the number of concurrent applications is m, and that each application needs to access data on all file servers (for simplicity). A collective data access from one application is mapped into n sub-parts to all file servers, and each sub-part is also a request in a file server. The service time on each file server for each sub-part is t.

Without I/O coordination, the sub-parts are served in different file servers independently. As requests are issued simultaneously, the sub-parts may be served randomly without order on all file servers. Hence for each sub-part, the finish time on each file server can randomly fall in $\{t, 2t, 3t, ..., mt\}$, and the finish time of data access for one application depends on the latest finish time of all nodes. The expectation of completion time of one data access is equal to the expectation of the maximum finish time on all n file servers. The average completion time can be represented as Formula (1), where $F(k)$ means the probability distribution function and $f(x)$ represents the probability density function. From the formula, we observe that, if there is only 1 file server, the expectation of completion time is $\frac{m+1}{2}t$, which conforms to the distribution of our assumption. The formula also indicates that the completion time increases as the number of file servers n increases, and also as the concurrent applications number m increases. When the file server number n is very large, $\frac{t}{m^n}\sum_{k=1}^{m-1} k^n$ would be close to 0, and then the average completion time would be close to mt.

$$\begin{aligned} T_{avg} &= E(Max(T)) = (\sum_{k=1}^{m} kf(k))t \\ &= (\sum_{k=1}^{m} k(F(k) - F(k-1)))t \\ &= (\sum_{k=1}^{m} k((\frac{k}{m})^n - (\frac{k-1}{m})^n))t \\ &= mt - \frac{t}{m^n}\sum_{k=1}^{m-1} k^n \end{aligned} \quad (1)$$

With the I/O coordination strategy, all file servers serve applications one at a time. I/O requests with synchronization needs will be served at the same time on all file servers. Therefore, the completion times for these applications are: t, $2t$, ..., mt, and the average completion time can be represented as Formula (2). The formula indicates that the average completion time is independent of n, the number of file servers. That means the average completion time of the I/O coordination scheme is much more scalable than that of existing independent scheduling strategies. Currently, parallel file systems usually reach up to hundreds of storage nodes or even beyond. The proposed I/O coordination strategy is a practical way to reduce the request completion time for data-intensive applications.

$$T'_{avg} = \frac{1}{m}\sum_{k=1}^{m} kt = \frac{m+1}{2}t \quad (2)$$

From Formula (1) and (2), we can calculate the reduction of the average completion time as follows.

$$\begin{aligned} T_{diff} &= T_{avg} - T'_{avg} \\ &= \frac{m-1}{2}t - \frac{t}{m^n}\sum_{k=1}^{m-1} k^n \end{aligned} \quad (3)$$

As can be seen in Formula 3, when the number of file servers n is very big, the reduction of completion time would be close to $\frac{m-1}{2}t$, and the decrease rate would be approaching $\frac{m-1}{2m}$. As the number of concurrent applications m increases, the decrease rate is approaching 50%.

4. IMPLEMENTATION

We have implemented the server-side I/O coordination scheme under PVFS2[9] and MPI-IO. PVFS2 is an open source parallel file system developed jointly by Clemson University and Argonne National Laboratory. It is a virtual parallel file system for Linux clusters based on underlying native file systems on storage nodes. The prototype implementation includes modifications to the PVFS2 request scheduling module and the PVFS2 driver package in ROMIO [26] MPI-IO library.

4.1 Implementation in PVFS2

We modified the client interface and server side request scheduler in PVFS2. The client interface passes 'Application ID' and 'Request Time' to the file servers, and then the file servers rearrange requests service orders based on the two parameters.

We utilize the '`PVFS_hint`' mechanism to pass the two parameters between I/O clients and file servers. Two new hint types are defined in the PVFS2 source code: '`PINT_HINT_APP_ID`' and '`PINT_HINT_REQ_TIME`', representing the Application ID and request time respectively. We made a modification of the client-side interface `PVFS_sys_read/write()`, adding `PVFS_hint` as a parameter, so that the hint could be passed to the PVFS2 server side.

When a file server receives a request, the scheduler first calculates its priority, and then inserts the request to the request queue in the ascending order of their priorities. The smaller the priority number a request gets, the earlier it would be scheduled. The request priority is calculated as follows.

```
req_prior = req_time / interval * 32768
            + app_id;
```

Here req_time is the issue time of the I/O request from the client side, and it is an integer value referring to the number of milliseconds since '1970-01-01 00:00:00 UTC'. $Interval$ is the width of the 'Time Window', which can be defined as a startup parameter in the PVFS2 configuration file. If $interval$ is not configured, it will use the default value (1000ms for HDD and 250ms for SSD). App_id represents 'Application ID', and it is an integer value in the range 0 to 32767. From the formula we observe that the req_prior

of a request in an earlier 'Time Window' is guaranteed to be smaller than a request in a later one. Also in one 'Time Window', a request with a small Application ID will be scheduled prior to that with a large one. Therefore, all the I/O requests in file servers are ordered by the value of req_prior.

In current PVFS2, each file server maintains a set of request queues for different file handles, and services requests in each queue in the FCFS (First Come First Serve) way. A file handle corresponds to a data file on one file server, which is usually a subfile of a whole PVFS2 file. We designed a global shared request queue to store all I/O requests of different jobs in the requests scheduler module. In request post function `PINT_req_sched_post()`, instead of adding an I/O request to the tail of the request queue of each file handle, the I/O scheduler inserts the request into the shared request queue according to the value of req_prior. The $trove$ module of PVFS2 handles read/write operations on block devices one by one from the head of the shared queue. Therefore, all I/O requests are serviced in the order of req_prior.

4.2 Implementation in MPI-IO Library

We also modified the PVFS2 driver in ROMIO [26] to pass 'Request Time' and 'Application ID' via '`PVFS_hint`'. 'Application ID' is generated the first time when an MPI program calls function `MPI_File_open()`, and then it is broadcast to all MPI processes. 'Application ID' is a global variable shared by all MPI processes, so that all processes of an MPI program get the same value of 'Application ID'. It is an unsigned integer value, which is generated randomly between 0 and 32767 by default. For system performance tuning, we also provide a configuration interface for parallel file system administrators. Administrators can specify the value of the 'Application ID' in a global configuration file, either as a fixed number or a range. If it is specified as a range, the value will be generated randomly in the range.

ROMIO[26] is a high-performance, portable implementation of MPI-IO, providing applications with a uniform interface in the top layer, and dealing with data access to various file systems by an internal abstract I/O device layer called ADIO. It provides various types of file system drivers in its internal abstract I/O device layer, including PVFS2. We modified the PVFS2 driver package in ROMIO: for every data access function, it first generates a request time, and adds the request time and global 'Application ID' into a variable of `PVFS_hint` type, and then passes the hint to file servers by calling modified data access functions. Following is an example of calling the PVFS2 data read interface.

```
...
PVFS_hint chint = PVFS_HINT_NULL;
int appid = app_id;
struct timeval rtime;
gettimeofday(&rtime, NULL);
long int req_time = rtime.tv_sec;

/* add application id and request time to hint */
PVFS_hint_add(&chint, "pvfs.hint.app_id", sizeof(int),
              &appid);
PVFS_hint_add(&chint, "pvfs.hint.req_time",
              sizeof(long int), &req_time);

/* call new read/write function with the hint
   parameters.*/
ret = PVFS_sys_read2(pvfs_fs->object_ref, file_req,
                     offset, buf, mem_req,
                     &(pvfs_fs->credentials),
                     &resp_io, chint);
...
```

These code modifications in the MPI-IO library are transparent to application programmers and users. There is no need to modify the source code of application; the user can simply relink the program using the modified MPI-IO library.

The request time is one of the primary factors used for request reordering on file servers in the proposed I/O coordination strategy. For this reason, the clock of all machines in the large-scale system must be synchronized. In our implementation, the request time is generated in MPI-IO library at the client side, so all the client machines must adopt the same clock. Clock skew of client nodes may lead to unexpected requests service orders, especially for the collective I/O synchronization and inter-request synchronization cases. Currently, most of the high-performance computing clusters have synchronized clocks using either NTP service or hardware clock synchronization(for example in Blue Gene/P).

5. EXPERIMENTAL EVALUATION

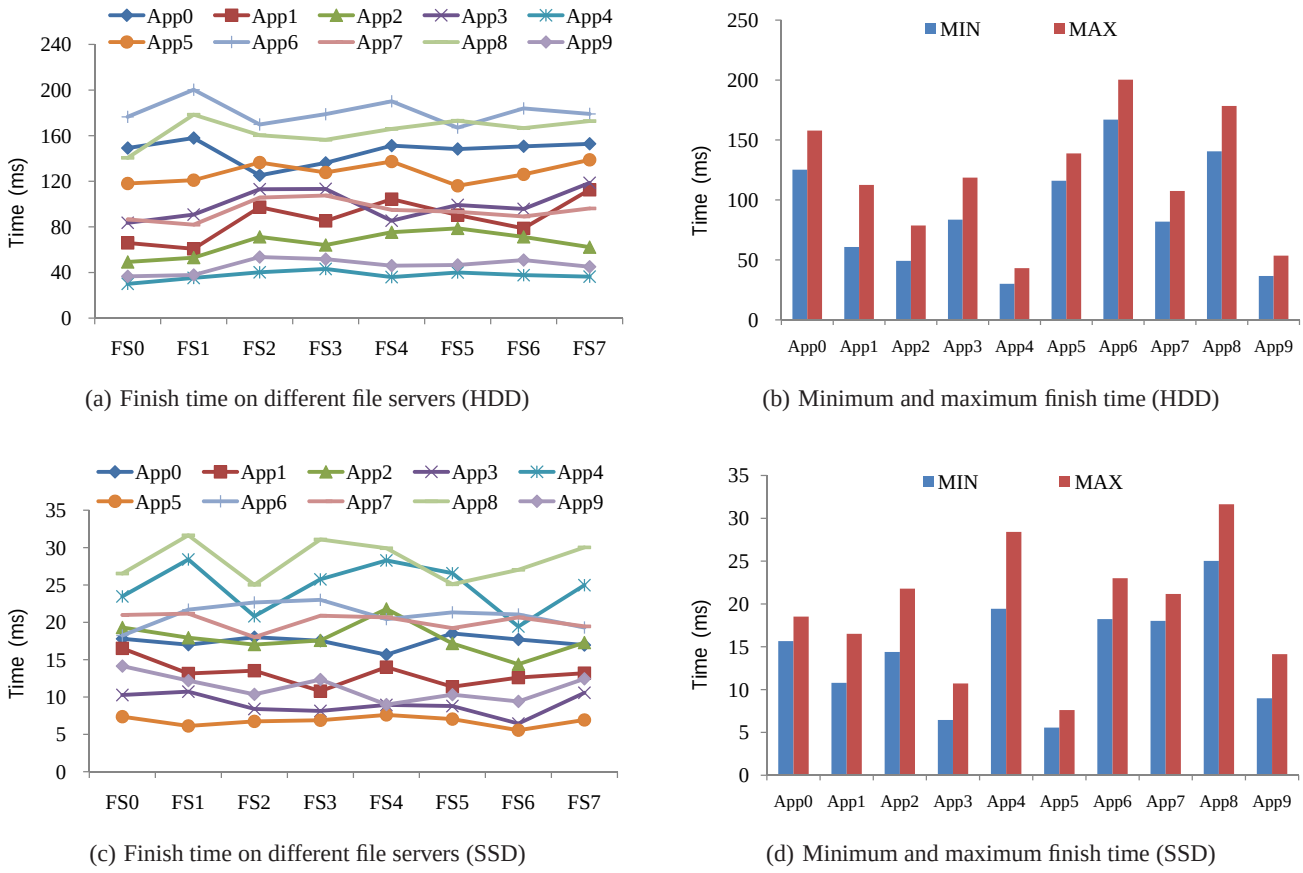
5.1 Experiments Setup

Our experiments were conducted on a 65-node SUN Fire Linux-based cluster, with one head node and 64 computing nodes. All nodes were equipped with Gigabit Ethernet interconnection. The model of head node was Sun Fire X4240, installed with dual 2.7 GHz Opteron quad-core processors, 8GB memory, and 12 500GB 7200RPM SATA II disk drives configured as RAID5 disk array. The computing nodes were Sun Fire X2200 servers, each with dual 2.3GHz Opteron quad-core processors, 8GB memory, and a 250GB 7200RPM SATA hard drive. All 65 nodes were connected with Gigabit Ethernet. In addition, 17 of these nodes (including the head node) were connected with 4X InfiniBand network, and had a PCI-E X4 100GB SSD. All these nodes ran Ubuntu 9.04 (Linux kernel 2.6.28.10) operating system. We implemented the I/O coordination strategy in MPICH2-1.1.1p1 and PVFS2 2.8.1 file system.

We evaluated the proposed I/O coordination strategy in both 'Gigabit Ethernet + HDD' and 'InfiniBand + SSD' environments. We measured average completion time, system scalability, and bandwidth with IOR, PIO-Bench, MPI-TILE-IO, and Noncontig benchmarks. IOR benchmark is a software used to test random and sequential I/O performance of parallel file systems. PIO-Bench provides a flexible framework for standardized testing of multiple file access methods. MPI-TILE-IO and Noncontig are designed to test the performance of MPI-IO for non-contiguous access workloads. All the tests were repeated 3 times. Before each run, we flushed memory to avoid the impact of memory cache and buffer.

5.2 Results and Analysis

First we conducted experiments to evaluate the completion time of I/O requests with the proposed I/O coordination strategy, by comparing with original scheduling strategy (without I/O coordination) in PVFS2. We used the same application scenarios shown in Figure 3. Figure 5 shows the completion time of different applications on different file servers. From the results we see that the I/O requests from one application were served together, and different applications finished one by one on all file servers. The maximum finish time is reduced from 4.4 to 1.3 times of the minimum finish time in the HDD environment, and from 3.1 to 1.2 times in the SSD environment. The completion time of one application relies on the maximum finish time of all file servers. From the results, we observe that the average completion time of all applications is reduced around 29.8% in the HDD environment and 19.5% in the SSD environment. Compared with the results in Figure 3, the proposed I/O coordination lets requests from the same application complete together, one at a time, rather than mixed random. We also notice some crossover in the completion time of the requests in subfigure (a) and (c). The reason is that, due to nonuniform network delays,



(a) Finish time on different file servers (HDD)

(b) Minimum and maximum finish time (HDD)

(c) Finish time on different file servers (SSD)

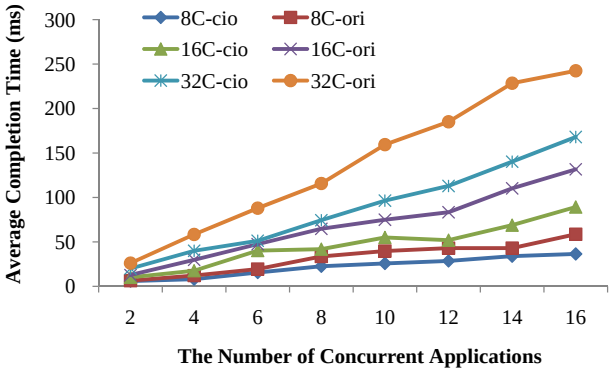
(d) Minimum and maximum finish time (SSD)

Figure 5: The finish time of I/O requests on different file servers with the proposed I/O coordination strategy. All file servers serve one application at a time together, and they serve I/O requests from multiple applications in the same order. The application scenarios are the same as in Figure 3.

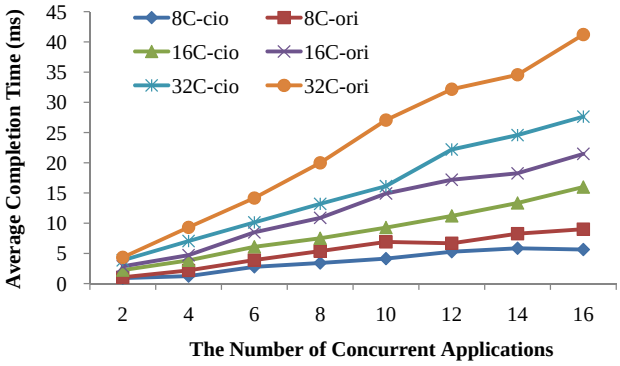
some requests with low priority were already issued to the storage devices in cases when some requests with high priority arrived late on some file servers. The proposed I/O coordination scheme always issues requests with the highest priority to low-level storage devices in the request queue on each file server. Therefore, a small percent crossover is expected.

We then compared the average completion time with different number of concurrent applications. We employed 16 file servers, and tested in both HDD and SSD environments. We used multiple instances of IOR to simulate concurrent applications. The numbers of concurrent instances were 2, 4, 6, 8, 10, 12, 14 and 16 respectively, and the number of MPI processes for each IOR instance were 8, 16 and 32, respectively. The width of 'Time Window' was set as 1000 milliseconds. The I/O request size was 128 KB, and the stripe size of PVFS2 was 4 KB. We added an `MPI_Barrier` operation between two requests and measured the completion time of each I/O request. Figure 6 shows the performance results. The prefix in the legend indicates the number of processes in each application, e.g. '32C' means 32 processes per application. The suffix 'cio' means the proposed I/O coordination strategy, and 'ori' means the original scheduling strategy in PVFS2. From the results we observe that the proposed I/O coordination always achieves lower average completion time, and the decrease in completion time is about 8% to 42% in HDD environment and 11% to 43% in SSD environment. Moreover, as the number of concurrent applications increases, the decrease rate of completion time rises, which matches our previous analysis.

Next we conducted experiments to evaluate the scalability of the proposed I/O coordination strategy. We configured PVFS2 with 2, 4, 8, 16, 32 and 64 file servers in HDD environment and 2, 4, 8, and 16 file servers in SSD environment, respectively. We adopted PIO-Bench instances for applications, and the number of MPI processes for each application were 8, 16, and 32, respectively. In this set of experiments, we measured the completion time of sequential read and write. The request sizes were $8KB*n$ (the number of file servers) for different runs, so that for each request the data size on all file servers is the same (8KB). We ran 10 concurrent PIO-Bench instances together. Figure 7 shows the results; the X axis represents the number of MPI processes for I/O coordination and original data access strategies. The figure demonstrates that I/O coordination can get a sustained steady completion time as the number of file servers increases, while with the original data access strategy, the average completion time grows as system scale increases. In the case of 2 file servers, the I/O coordination could obtain about 10% reduction of average completion time compared to original scheduling strategy in both HDD and SSD environments. While the number of file servers increases, the completion time decrease is around 46% for 64-node HDD environment and 39% for 16-node SSD environment. The results indicate that, the proposed I/O coordination

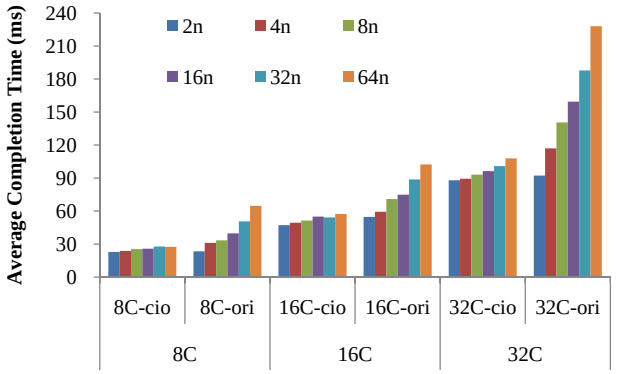


(a) Average completion time (HDD)

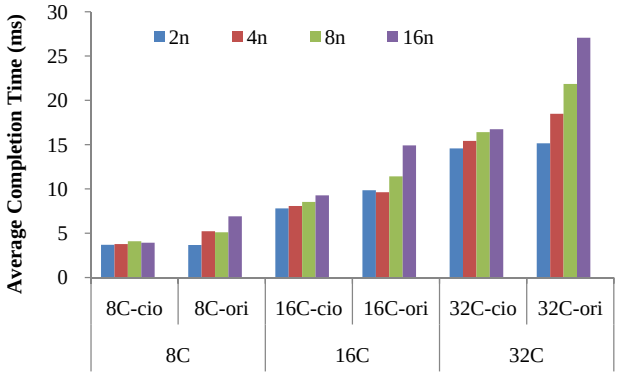


(b) Average completion time (SSD)

Figure 6: Average completion time with different numbers of concurrent applications. Prefix '8C', '16C' and '32C' mean each application has 8, 16 and 32 MPI processes, respectively. Suffix 'cio' means the I/O coordination scheme, and 'ori' means the original data access strategy. Labels in Figure 7 are similarly defined. We used multiple instances of IOR to simulate concurrent applications.



(a) Average completion time (HDD)



(b) Average completion time (SSD)

Figure 7: Average completion time with different numbers of file servers. In HDD tests, the number of file servers was configured as 2, 4, 8, 16, 32, or 64. In SSD tests, we configured the number of file server as 2, 4, 8, or 16, respectively. We adopted PIO-Bench instances to simulated concurrent applications.

strategy is effective and even more appropriate for large-scale parallel file systems.

We also conducted experiments to evaluate the effect of different lengths of the time window of the proposed I/O coordination scheme. We set time window sizes as 250ms, 500ms, 1000ms, and 2000ms, and compared their completion time and I/O bandwidth without I/O coordination. The number of file servers was 16 in both HDD and SSD experiments. We adopted 3 IOR, 3 PIO-Bench, 2 MPI-TILE-IO, and 2 Noncontig instances to simulate 10 concurrent applications. The numbers of MPI processes for each application were 8, 16, and 32, respectively (labelled as '8C', '16C', and '32C'). The request sizes of all programs were 128 KB, and the stripe size was 4 KB. Figure 8 shows the experimental results, where subfigures (a) and (c) show average completion time and subfigures (b) and (d) show the aggregate I/O bandwidth. From subfigures (a) and (c) we observe that, for all time window sizes, the completion times with the proposed I/O coordination strategy are lower than that with original strategy without I/O coordination. In addition, the window size 1000ms results in the lowest completion time in almost all cases in the HDD tests, and the 250ms window size results in the lowest completion time in SSD tests. From subfigure (b) we can observe that, in HDD tests, the I/O bandwidth increases as the window size is increased from 250ms to 1000ms, and the I/O bandwidth with window size 1000ms and 2000ms are similar. From subfigure (d) we observe that, in SSD tests the window size 250ms obtained the highest bandwidth. From the results in HDD environment we see that, when the number of processes in each application is 8, the bandwidth of original I/O scheduling strategy is little higher (up to 1.2%) than I/O coordination scheme in some cases. But for 32 processes, the I/O coordination strategy achieved about 10.9% higher aggregate bandwidth than the original strategy. The results in SSD show that the I/O coordination scheme always obtains the highest I/O bandwidth. The results indicate that the I/O coordination strategy can achieve comparable bandwidth to the original strategy when the I/O workload of a parallel file system is heavy. The experimental results also indicate that, the size of time windows affect the completion time and I/O bandwidth. Generally, parallel file systems consisting of high performance storage devices should set a short time window, and those consisting of lower performance storage devices should set a relative large window size. Based on the results in this set of experiments, we recommend to set the window size to 1000ms in an HDD environment and 250ms in an SSD environment.

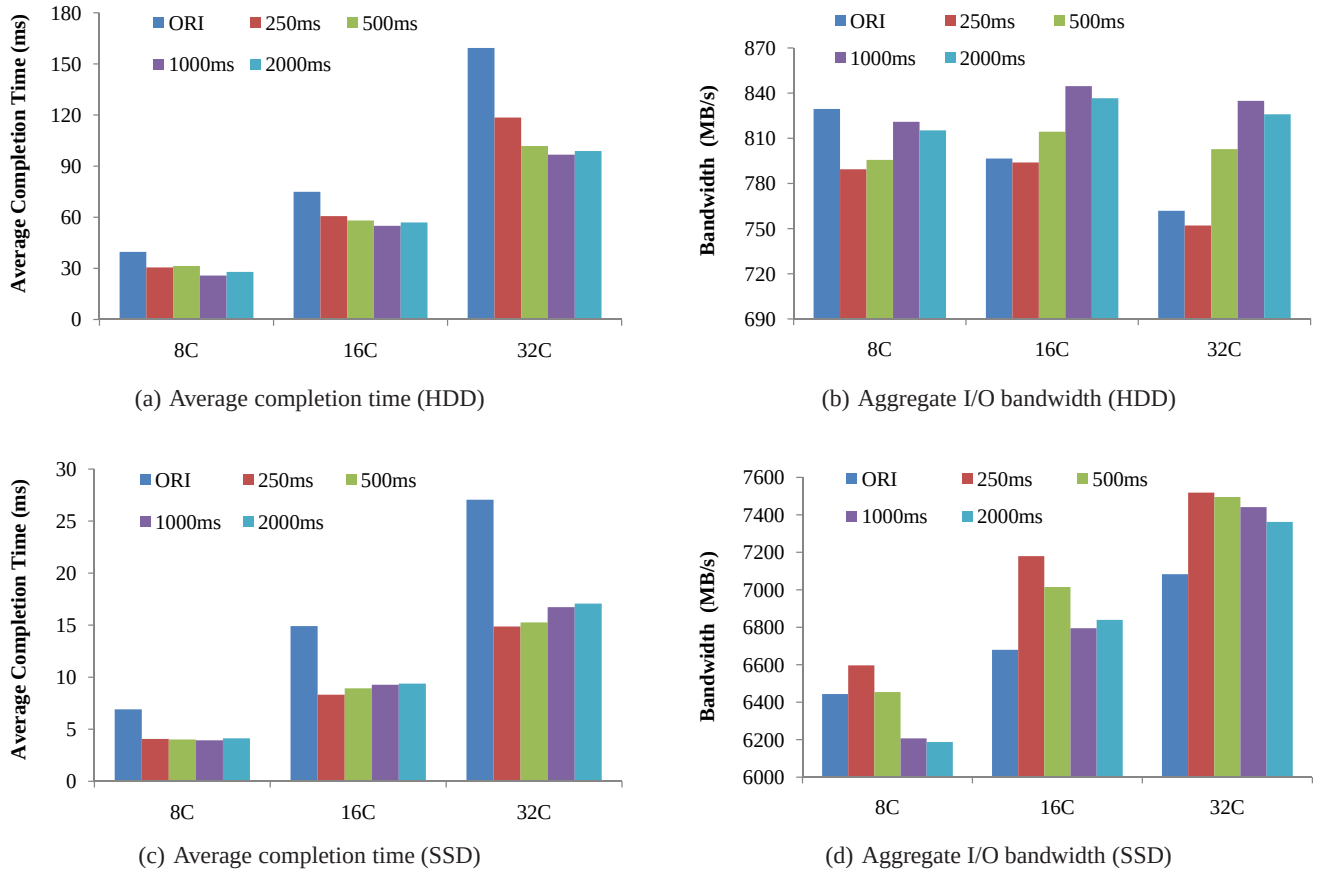


Figure 8: Average completion time and aggregate I/O bandwidth under different window sizes. We run 10 concurrent applications (3 IOR, 3 PIO-Bench, 2 MPI-TILE-IO, and 2 Noncontig instances) in both HDD and SSD environments. The window sizes of I/O coordination scheme were 250ms, 500ms, 1000ms, and 2000ms, respectively. We also measured the results without I/O coordination strategy (labelled as 'ORI' in the figure).

6. RELATED WORK

We discuss related work in the area of scheduling in parallel I/O and parallel file systems, and we also discuss coordinated scheduling techniques, and discuss how our work differs from those efforts.

6.1 Server-side I/O Scheduling in Parallel File Systems

In order to obtain sustained peak I/O performance, a collection of I/O scheduling techniques have been developed for the server side I/O scheduling of parallel file systems, such as disk-directed I/O [13], server-directed I/O [23], and stream-based I/O [11, 21]. These techniques succeed in achieving high bandwidth in disks and networks of file servers, by reducing either the frequency of disk seeks, or the waiting time of socket connections. However, to the best of our knowledge, little effort has been devoted to reducing the average completion time of I/O requests of multiple applications for multiple file servers.

Numerous research efforts have been devoted to improving quality of service (QoS) of I/O requests in distributed or parallel storage systems [4, 8, 10, 12, 19, 29]. Some of them adopted deadline-driven strategies [12, 19, 31], that allow the upper layer to specify latency and throughput goals of file servers and schedule the requests based on Earliest Deadline First(EDF) [16] or its variants [19, 20, 31]. Some approaches employed proportional-sharing scheduling strategies [8, 29] between competing I/O workloads. These strategies aim to either provide fairness of sharing of bandwidth for clients, or to control the requests issue queue lengths of I/O clients to guarantee a moderate latency. Our approach is different from these works in that we do not use explicit deadline-driven or throttling control algorithms, making it completely transparent to I/O clients. Moreover, our approach takes into consideration multiple file servers.

6.2 Coordinated scheduling

Coordinated scheduling has been recognized as an effective approach to obtain efficient execution for parallel or distributed environments. It has been achieved with gang scheduling [5, 6, 7, 14, 17, 27, 28, 30] and co-scheduling [2, 3, 15, 25, 24]. A large body of research has been devoted to reducing the synchronization time for communication between threads/processes, by scheduling related threads/processes to run simultaneously on different processors in a parallel or distributed system. The scheduler packs synchronized processes into gangs and schedules them simultaneously, to alleviate performance penalties of communicative synchronization. Feitelson et al. [5] made a comparison of various packing schemes for gang scheduling, and evaluated them under different cases. Wiseman et al. [30] matched gangs that make heavy use of the CPU with gangs that make light use of the CPU, and scheduled such pairs

together, to improve the throughput by making better utilization of the system resources. Wang et al. [27, 28] presented a mathematical model that can measure system performance with different scheduling parameters, to guide the design of scheduling policies. All these coordinated scheduling techniques focus on the thread, process or job level, either to reduce synchronization time or to better utilize the system resources. However, none of these works focused on I/O request scheduling. Moreover, current coordinated scheduling policies employed centralized or distributed schedulers, both of which are based on communication itself among processors/nodes. In our approach no central control mechanism exists, nor communication between file servers, making it much more suitable for large-scale parallel file systems.

Zhang et al. [32] proposed an inter-server coordination technique in parallel file systems to improve the spatial locality and program reuse distance. They calculated the access distances and group the requests with small distance together, but this optimization does not apply to SSDs. Our approach, on the other hand, is based on the observation that the requests with synchronization needs will be optimized if they finish at the same time. We coordinate among file servers so that they work on one application at a time together. The motivation and methodology of the design and implementation of our and their approaches are very different. In addition, our approach does not require a central control, is simple in implementation, and can be extended to SSDs.

7. CONCLUSIONS AND FUTURE WORK

Parallel file systems are widely used for data-intensive and high-performance computing applications. However, I/O performance lags far behind the computing capacities in current systems, resulting in processors wasting large numbers of cycles waiting for data to arrive. The situation becomes even worse when multiple applications try to access data concurrently. Existing server-side scheduling algorithms focus on tapping the potential capacity of each single file server to achieve a higher throughput of each storage node, thus to improve overall system throughput. Little has been done to investigate coordinated data access on multiple file servers to reduce the average completion time from the parallel data access point-of-view. This paper targets the problem of reducing the average completion time of I/O requests from multiple applications.

This paper proposes a novel server-side I/O coordination scheme in which all file servers serve requests in step to alleviate the impact of data synchronization, and also maintain fairness and simplicity. The proposed scheme lets all file servers work on one application at a time based on a automatically created chronological order recognized within the whole cluster, rather than all servers working independently. This paper makes the following contributions. First, we describe the I/O synchronization problems in parallel I/O systems, and demonstrate that re-arranging service orders on multiple file servers is beneficial. Second, we propose an I/O coordination scheme to let all file servers work in concert. Third, we have implemented the proposed I/O strategy in PVFS2 and MPI-IO. The experimental results demonstrate that, compared to the conventional data access strategy, the proposed I/O coordination scheme can reduce the I/O completion time by up to 46% and provide a comparable I/O bandwidth. Analytical and experimental results confirm that the control mechanism of the proposed I/O coordination scheme is simple and effective, and it is an appropriate choice for large-scale parallel file systems with heavy I/O workloads.

In the future, we plan to investigate optimization of the I/O coordination strategy based on application data access patterns. We also plan to add a minimum group communication in I/O coordination, to explore its feasibility for imbalanced data access workloads.

8. ACKNOWLEDGMENTS

The authors are thankful to Jibing Li from Illinois Institute of Technology and Robert Ross from Argonne National Laboratory for their constructive and thoughtful suggestions toward this study. This research was supported in part by National Science Foundation under NSF grant CCF-0621435 and CCF-0937877, and in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

9. REFERENCES

[1] High-performance Storage Architecture and Scalable Cluster File System. Lustre File System White Paper, Dec. 2007.

[2] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, New York, NY, USA, 1995. ACM.

[3] A. C. Arpaci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*, 19(3):283–331, 2001.

[4] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance Virtualization for Large-scale Storage Systems. In *SRDS '03: Proceedings of the 22th International Symposium on Reliable Distributed Systems*, pages 109–118, 2003.

[5] D. G. Feitelson. Packing Schemes for Gang Scheduling. In *IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 89–110, London, UK, 1996. Springer-Verlag.

[6] D. G. Feitelson and M. A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 238–261, London, UK, 1997. Springer-Verlag.

[7] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[8] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST '09: Proccedings of the 7th Conference on File and Storage Technologies*, pages 85–98, Berkeley, CA, USA, 2009. USENIX Association.

[9] I. F. Haddad. PVFS: A Parallel Virtual File System for Linux Clusters. *Linux Journal*, page 5, 2000.

[10] L. Huang, G. Peng, and T.-c. Chiueh. Multi-dimensional Storage Virtualization. *SIGMETRICS Perform. Eval. Rev.*, 32(1):14–24, 2004.

[11] W. B. Ligon III and R. B. Ross. Implementation and Performance of a Parallel File System for High Performance Distributed Applications. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 471, Washington, DC, USA, 1996. IEEE Computer Society.

[12] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance Differentiation for Storage Systems Using Adaptive Control. *ACM Trans. Storage*, 1(4):457–480, 2005.

[13] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. *ACM Trans. Comput. Syst.*, 15(1):41–74, 1997.

[14] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of I/O for Gang Scheduled Workloads. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 215–237, London, UK, 1997. Springer-Verlag.

[15] S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Algorithms. In *SIGMETRICS '90: Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 226–236, New York, NY, USA, 1990. ACM.

[16] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Dard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[17] J. E. Moreira, W. Chan, L. L. Fong, H. Franke, and M. A. Jette. An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–14, Washington, DC, USA, 1998. IEEE Computer Society.

[18] D. Nagle, D. Serenyi, and A. Matthews. The Panasas Activescale Storage Cluster: Delivering Scalable High Bandwidth Storage. In *Supercomputing '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53, Washington, DC, USA, 2004. IEEE Computer Society.

[19] A. Povzner, D. Sawyer, and S. Brandt. Horizon: Efficient Deadline-driven Disk I/O Management for Distributed Storage Systems. In *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 1–12, 2010.

[20] A. L. Narasimha Reddy and J. C. Wyllie. Disk Scheduling in a Multimedia I/O System. In *MULTIMEDIA '93: Proceedings of the First ACM International Conference on Multimedia*, pages 225–233, New York, NY, USA, 1993. ACM.

[21] R. B. Ross and W. B. Ligon III. Server-Side Scheduling in Cluster Parallel I/O Systems. *Calculateurs Parallèles Special Issue on Parallel I/O for Cluster Computing*, 2001.

[22] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 19, Berkeley, CA, USA, 2002. USENIX Association.

[23] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed Collective I/O in Panda. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 57, New York, NY, USA, 1995. ACM.

[24] A. Snavely and D. M. Tullsen. Symbiotic Job Scheduling for a Simultaneous Multithreaded Processor. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 234–244, New York, NY, USA, 2000. ACM.

[25] P. G. Sobalvarro. *Demand-Based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors*. PhD thesis, 1997. Supervisor-Weihl, William E.

[26] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *FRONTIERS '99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, page 182, Washington, DC, USA, 1999. IEEE Computer Society.

[27] F. Wang, H. Franke, M. C. Papaefthymiou, P. Pattnaik, L. Rudolph, and M. S. Squillante. A Gang Scheduling Design for Multiprogrammed Parallel Computing Environments. In *IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 111–125, London, UK, 1996. Springer-Verlag.

[28] F. Wang, M. C. Papaefthymiou, and M. S. Squillante. Performance Evaluation of Gang scheduling for Parallel and Distributed Multiprogramming. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 277–298, London, UK, 1997. Springer-Verlag.

[29] Y. Wang and A. Merchant. Proportional-share Scheduling for Distributed Storage Systems. In *FAST '07: Proceedings of the 5th USENIX Conference on File and Storage Technologies*, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

[30] Y. Wiseman and D. G. Feitelson. Paired Gang Scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 14(6):581–592, 2003.

[31] T. M. Wong, R. A. Golding, C. Lin, and R. A. Becker-Szendy. Zygaria: Storage Performance as a Managed Resource. In *In IEEE Real Time and Embedded Technology and Applications Symposium (RTAS 06*, pages 125–134, 2006.

[32] X. Zhang, K. Davis, and S. Jiang. IOrchestrator: Improving the Performance of Multi-node I/O Systems via Inter-Server Coordination. In *Supercomputing '10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2010. IEEE Computer Society.