# Server-side scheduling in cluster parallel I/O systems

**Robert B. Ross* and Walter B. Ligon III****

**Mathematics and Computer Science Division*
*Argonne National Laboratory*
*Argonne, IL 60439, USA*
*rross@mcs.anl.gov*

***Holcombe Department of Electrical and Computer Engineering*
*Clemson University*
*Clemson, SC 29634, USA*
*walt@clemson.edu*

ABSTRACT. *Parallel I/O has become a necessity in the face of performance improvements in other areas of computing systems. Studies have shown that peak performance is infrequently realized, and work in parallel I/O optimization strives to achieve peak performance for applications. In this paper we revisit one area of performance optimization in parallel I/O, that of server-side scheduling of service. With the wide variety of systems and workloads seen today, multiple server-side scheduling algorithms are necessary to match potential workloads. We show through experimentation that performance gains can be seen in practice through the use of alternative scheduling algorithms, but that no single algorithm provides the best performance across the board. Finally we discuss the potential for automatic matching of server-side scheduling algorithms to workloads in real-time.*

RÉSUMÉ. *TBD*

KEYWORDS: *parallel I/O, PVFS, parallel file system, scheduling*

MOTS-CLÉS : *TBD*

e soumission à *Calculateurs Parallèles*, le 9th November 2001.

## 1. Introduction

Performance improvements in computing technology have vastly out-paced improvements in storage technology. This trend has led to the adoption of parallel I/O systems as a solution. By combining large numbers of storage devices and providing the system software to utilize them in concert, parallel I/O has extended the range of problems that may be solved on high performance computing platforms. However, it is obvious from workload studies that peak performance is rarely attained from these coordinated storage devices.

In order to address this, a collection of techniques for more efficiently utilizing these resources has been researched and developed. The collection includes traditional I/O enhancements such as prefetching and caching as well as novel approaches to organizing storage access and data transfer in parallel systems. The majority of these approaches were tailored for the environment in which they were originally applied, that of commercial supercomputers. In commercial parallel machines the network typically has much lower latency and much higher throughput than the storage system, and accounting for this disparity is the goal of much parallel I/O work.

More recently new parallel computing platforms have emerged, including PC clusters such as Beowulf computers [RID 97]. Beowulfs are constructed from commodity components and commonly include fast ethernet networks and IDE disks. These components are of roughly the same order of magnitude of performance. When coupled with varying workloads it becomes even less obvious what resources, if any, will consistently outperform the other components in the system. The widespread adoption of clusters as a high performance computing platform has seen the same I/O techniques developed for commercial supercomputers applied again, but no work has thus far examined the viability of these approaches in this new arena.

In this work we focus on the application of server-side scheduling algorithms in parallel I/O workloads on cluster systems. First we cover previous work related to scheduling of server operations. Second we describe the parallel file system in which these algorithms will be implemented and tested. Third we describe the set of scheduling algorithms we have implemented for the purpose of this study. Next we describe the set of workloads used in our testing to provide a variety of access patterns and resource demands. Finally we examine the results of our experiments. From these experiments we note that the application of scheduling on the server side does have noticeable effects on performance, and that a correlation between resource demand and optimal algorithm choice can be seen. We go on to describe how the results of this work might be used to implement a scheduling system that can dynamically apply scheduling algorithms, in real-time, based on workload information.

## 2. Server-side scheduling

The networks in the machines in use during the majority of early parallel I/O work were of much higher performance than the the storage subsystems provided in these

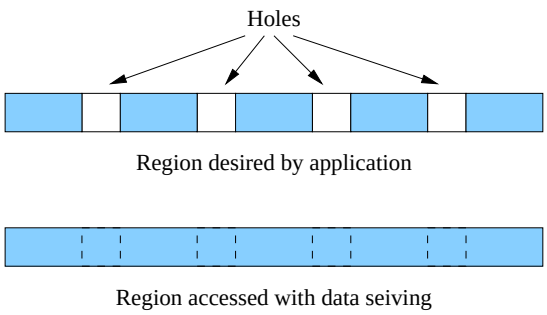



**Figure 1.** *Data sieving example*

systems. For example, the iPSC/860 system at NAS [NIT 92] had a total of ten I/O nodes utilizing SCSI disks with a peak of 1 Mbyte/sec providing storage for the CFS parallel file system. The hypercube-based network provided 2.8 Mbytes/sec connections between processors. At least in part because of this environment, early techniques for optimizing access tended to focus on optimizing disk performance. The first four techniques covered in this section, data sieving, two-phase I/O, disk-directed I/O, and server-directed I/O, were all originally applied in or designed for these types of systems. The last technique, stream-based I/O, was designed instead with cluster systems in mind. We will discuss each of these in turn.

### 2.1. *Data sieving*

*Data sieving* is a technique for efficiently accessing noncontiguous regions of data in files when noncontiguous accesses are not provided as a file system primitive [CHO 94]. It is presented here because it is a building block for two-phase access, which is discussed in the next section.

Workload studies on a number of platforms have shown that noncontiguous accesses are a common occurrence in parallel I/O workloads [KOT 95]. The naive approach to accessing noncontiguous regions is to utilize a separate I/O call for each contiguous region in the file. This results in a large number of I/O operations, each of which is often for a very small amount of data. The added network cost of performing an I/O operation across the network, as in parallel I/O systems, is often high due to latency. Thus this naive approach typically performs very poorly because of the overhead of multiple operations. In the data sieving technique, a number of noncontiguous regions are accessed by reading a block of data containing all of the regions including the unwanted data between them (called “holes”). Figure 1 shows an example of how data sieving might access a number of noncontiguous regions by reading a single block. The regions of interest are then extracted from this large block by the client. This has the advantage of a single I/O call, but additional data is read from the disk

and passed across the network. The implementors found for their test system, an Intel Touchstone Delta, that the reduction of operations outweighs the added data transfer for a large percentage of accesses.

This technique can in fact also benefit systems with high latency networks as well in that it reduces the number of requests, for which there is often significant startup time. However, the percentage of data transferred that is desired must be high for this to pay off. A more appropriate technique for reducing the overhead of multiple requests in network bound systems is the use of more descriptive requests. These allow the I/O server to either perform noncontiguous accesses, if the capability is available, or to perform this sieving on the server side, reducing network traffic. However, most I/O systems only support contiguous accesses.

### 2.2. *Two-phase I/O*

The *two-phase access* strategy is described in [BOR 93]. This strategy attempts to avoid the performance penalties often incurred when directly mapping from the distribution of data on disks to the distribution in processor memories. Data is first read from disk, in the arrangement it is stored in on disk, by a subset of processors. The data is then redistributed to the processors in the final, processor-memory distribution. The strategy was tested on the Intel Touchstone Delta using the Concurrent File System (CFS). The 512 processor Delta has a limited number of I/O nodes (32) and substantial network bandwidth between processors, which are arranged in a mesh topology.

In the first phase of access, the number of processors involved is chosen to match the I/O nodes. Each chosen processor typically requests all the needed data from a single disk and uses data sieving to reduce the number of requests. In the second phase, the processors who previously read data from the disks calculate the final destination for each block of data and perform the necessary transfers. This takes advantage of the additional bandwidth between compute processors to more quickly complete the I/O process.

In order for this technique to be of use, compute processes must communicate and organize the transfer, which means that collective I/O must be available. The collective component of two-phase access can be implemented above the file system layer (i.e. on top of a file system that does not support collective accesses). Possibly the most popular implementation of two-phase I/O at this time is in the ROMIO MPI-IO implementation [ROM], which implements two-phase accesses on top of a variety of parallel file systems with only independent access primitives.

This technique indirectly affects the behavior of I/O servers by altering the request pattern from many small accesses into single large accesses per server. This in effect forces the server into a mode where it is sequentially accessing a single large region (assuming the server returns the bytes from the request in order). It also constrains the server to servicing requests for a single client, since only one client makes a request. When disk is the bottleneck, this technique is often a win. Additionally it has been

shown that some I/O systems perform best when the number of simultaneous accesses is limited [KRY 93, NIT 92]. These systems in particular benefit from this approach.

### 2.3. *Disk-directed I/O*

Disk-directed I/O (DDIO) is a combination of a number of other techniques for data transfer in parallel I/O systems [KOT 97]. DDIO was developed after both the data sieving and two-phase techniques, and it relies on both noncontiguous and collective I/O primitives in the file system. Additionally the I/O servers must be able to map file locations into disk block positions and must be capable of reasonably predicting the optimal disk access pattern. The disk-directed technique uses the information passed to it about the data requested in the collective request to determine a list of physical blocks to retrieve from the disk. It sorts these blocks into some optimal access ordering and uses double-buffering to overlap disk and network I/O, sending data directly to the final destination.

From the server scheduling point of view the disk-directed approach is superior to two-phase in that it passes a great deal more information on the total access along to the server. This allows the server to determine the access ordering utilizing both information on what is being accessed and also information on where that data is located on disk. Furthermore the disk-directed approach gives the server the opportunity to schedule access across multiple network connections as well, as data is moved directly from the server to the appropriate clients.

As far as contributions to a well-rounded I/O transfer method, DDIO has a number of things to offer. First, it makes use of noncontiguous requests, generally resulting in fewer, larger packets. Second, it promotes the use of an ordering scheme for optimization on the server side. While it might not always make sense to optimize for disk access, and a static scheme such as the one used in their examples might not help in a complex system, it does make sense to have a system capable of determining the cost of transferring particular packets and ordering transfers accordingly. Unfortunately most parallel I/O systems do not meet the requirements for implementing DDIO. Server-directed I/O relaxes these requirements.

### 2.4. *Server-directed I/O*

A derivative of disk-directed I/O, called server-directed I/O, was proposed and implemented in the PANDA library [SEA 95]. This technique utilizes a high-level multidimensional data set interface, performs array chunking, and uses disk-directed techniques at the logical, or file, level. Instead of determining physical block locations, they use logical file offsets to determine their optimal ordering. File data is stored on underlying local file systems, and block arrangement information was unavailable. The developers found that they were able to utilize almost the full capacity of the

disk subsystems in their test system for a range of array sizes and numbers of nodes, despite the lack of disk block layout information.

### 2.5. *Stream-based I/O*

Stream-based I/O (SBIO) attempts to address network bottlenecks in parallel I/O systems [LIG 96]. The SBIO technique was developed as part of the Parallel Virtual File System (PVFS) project [LIG 96], which is described in Section 3. With SBIO, this concept of combining small accesses into more efficient, large ones is applied to data transfer over the network. Data being moved between clients and servers is considered to be a stream of bytes regardless of the location of data bytes within a file. This is similar to a technique known as message coalescing in interprocessor communication. These streams are packetized by underlying network protocols (e.g. TCP) for movement across the network. Control messages are placed only at the beginning and end of the data stream in order to minimize their effects on packetization. This is accomplished by calculating the stream data ordering on both client and server.

This is strictly a technique for optimizing network traffic. When coupled with a server that focuses on the network (almost "network directed I/O"), peak performance can be maintained for a variety of workloads, particularly when network performance lags behind disk performance or when most data on I/O servers is cached.

## 3. PVFS design

One area in which commercial machines still maintain great advantage is that of parallel file systems. A production-quality high-performance parallel file system has not been available for Linux clusters, and without such a file system, Linux clusters cannot be used for large I/O-intensive parallel applications. To fill this need, we have developed a parallel file system for Linux clusters called the Parallel Virtual File System (PVFS). PVFS is being used by a number of groups, including ones at Argonne National Laboratory and the NASA Goddard Space Flight Center. Other researchers are using the PVFS system as a research tool [TAK 99].

Details on PVFS can be found in [CAR 00]. The rest of this section focuses on the components of PVFS which are of importance to this study, namely how PVFS manages data storage, how PVFS processes requests for data, and how this request processing might be modified to study server-side scheduling algorithms.

### 3.1. *PVFS metadata*

PVFS files are striped across a set of I/O nodes in round-robin fashion. The specifics of a given file distribution are described with three metadata parameters: starting I/O node number (*base*), number of I/O nodes (*pcount*), and strip size (*ssize*).

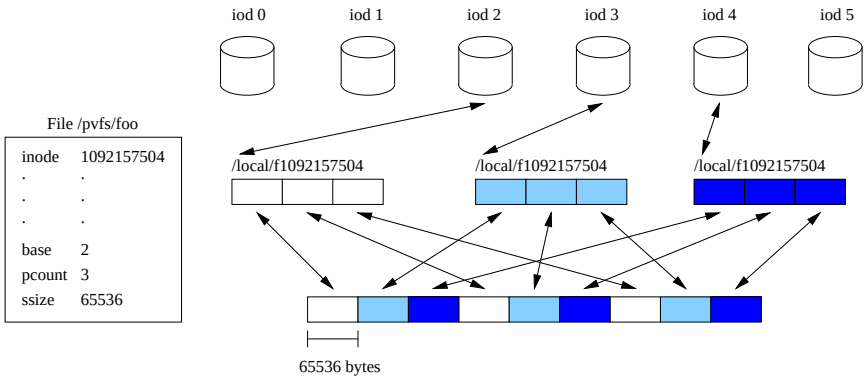



**Figure 2.** *Example metadata and file distribution*

These parameters, together with an ordering of the I/O nodes for the file system, allow the file distribution to be completely specified.

An example of some of the metadata fields for a file `/pvfs/foo` is given in Figure 2. The *pcount* field specifies that the data is spread across three I/O nodes, *base* specifies that the first (or base) I/O node is node 2, and *ssize* specifies that the strip size—the unit by which the file is divided among the I/O nodes—is 64 Kbytes. The application can set these parameters when the file is created, or, if not specified, PVFS will use a default set of values.

This file metadata, including locations of I/O nodes, is obtained by the application from the PVFS system when a file is opened. This information allows applications to communicate directly with I/O nodes when file data is accessed.

### 3.2. *I/O daemons and data storage*

An ordered set of I/O daemons (iods) run on the I/O nodes in the cluster. The I/O nodes are specified by the administrator when the file system is installed. These daemons are responsible for using the local disks on each I/O node for storing data for PVFS files, and they do so by using a local file system to store data. For each PVFS file handled by the daemon, a local file is created on an existing local file system. These files are accessed using standard UNIX `read()`, `write()`, and `mmap()` operations. This means that all data transfer occurs through the kernel block and page caches and is scheduled by the kernel I/O subsystem.

Figure 2 shows how the example file `/pvfs/foo` is distributed in PVFS based on the metadata. Note that although there are six I/O nodes in this example, the file is striped across only three I/O nodes, starting from node 2, because the metadata file specifies such a distribution. Each I/O daemon stores its portion of the PVFS file in a

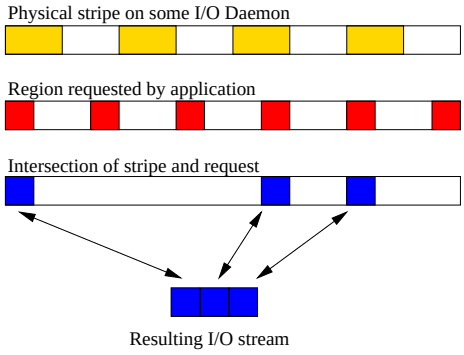



**Figure 3.** *I/O stream example*

file on the local file system on the I/O node. The name of this file is based on the inode number that the PVFS system assigned to the file (in our example, 1092157504).

As mentioned above, when application tasks (clients) open a PVFS file they are returned the locations of the I/O daemons. The clients then establish connections with the I/O daemons directly. These connections are used strictly for data requests. When a client wishes to access file data, the client library sends a description of the file region being accessed to the I/O daemons holding data in the region. The daemons determine what portions of the requested region they have locally and perform the necessary data transfers using TCP/IP.

Figure 3 shows an example of how one of these regions, in this case a simple-strided region, might be mapped to the data available on a single I/O node. The intersection of the two regions defines what we call an *I/O stream*. This stream of data is transferred in logical file order across the network connection. By retaining the ordering implicit in the request and allowing the underlying stream protocol to handle packetization, no additional overhead is incurred with control messages at the application layer.

Figure 4 shows in greater detail what happens when a client accesses data from PVFS I/O daemons. In red (gray) we see the data that was requested, which corresponds to the region described in Figure 3. In the two shades of blue (black and light gray) we see the portions of this data that are stored on the two I/O nodes across which this file is striped.

When this region is accessed, the I/O daemons each send back an I/O stream containing the requested data that they possess. These two streams are then merged into the application data buffer on the client.

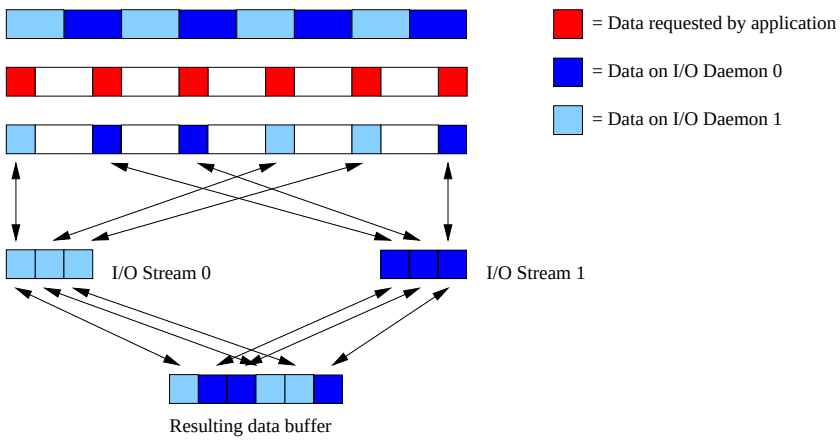



**Figure 4.** *File access example*

### 3.3. *PVFS request processing*

The PVFS request processing mechanism is the component of PVFS that implements the scheduling policy. Here we provide a short overview of the PVFS request processing implementation. First we cover how PVFS receives requests, how these are processed, and the structure in which they are stored for service. Next we discuss how the system handles multiple simultaneous requests. We concentrate on read operations in this discussion, and we detail the actual system calls used by the iods to perform local file system and socket accesses.

#### 3.3.1. *Receiving and queuing requests*

Since all PVFS I/O is currently performed over TCP, all PVFS communication is through the UNIX sockets interface. PVFS I/O servers maintain a set of open sockets that are checked for activity in a loop. One of these sockets is an "accept" socket that is used by clients to establish connections for service. The other two possible states for an open socket are that it is connected but has no outstanding request, or that it is in active use for servicing a request.

PVFS I/O servers are single-threaded entities that rely on the `select()` call to identify connections that are ready for service. One of the sockets that the server queries is the accept socket. When this socket (or any other open socket not involved in a request) is ready for reading, the I/O server attempts to receive an I/O *request*. Requests are messages sent by application tasks (clients) asking that some operation be performed on their behalf. The request is parsed after reception, and if the request requires data transfer a *job* is created to perform the necessary I/O. Figure 5 shows the job data structure as it is being created to service a request. The job is associated with a socket and file, and attached to the job is a list of *accesses*. Accesses are data

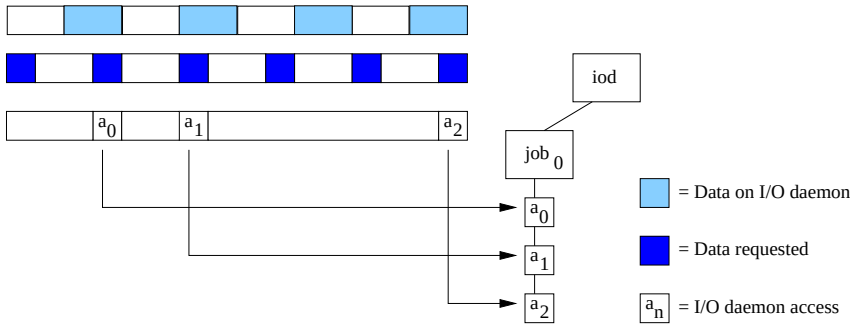



**Figure 5.** *Creating accesses from a request*

transfers that must be performed to service the request. A job may have as many as tens or hundreds of accesses.

First the I/O server allocates the job structure and breaks the request into contiguous accesses based on the intersection of the requested data and the data available locally on the server. This is the process diagrammed in Figure 5. Following this an acknowledgment is prepended to the access list for passing status information back to the client (i.e. EOF reached). This job is then added to the collection of jobs that the I/O server is processing.

Figure 6 shows multiple jobs in service simultaneously. As the I/O server processes these jobs, accesses associated with the job are updated and removed when completed. In this example, job 1 has already had its acknowledgment sent, so it is no longer present on the access list.

### 3.3.2. *Servicing requests*

Typically each task in a parallel application will send a request to each I/O server when performing an I/O operation, resulting in a job on each server. It is easy to imagine that for a large parallel application a large number of jobs might be in service on an I/O server at one time. This large number of jobs, for which there are by definition no inter-job dependencies, provide us with an opportunity to optimize by selecting the order in which jobs will be serviced. I/O servers, when not idle, sit in a loop servicing requests:

```
while (job list not empty) {
   select a job to service
   make progress on accesses for selected job
}
```

Selection of a job can be performed by any means we wish, including examining the characteristics of the job such as size or next access type and position. This gives

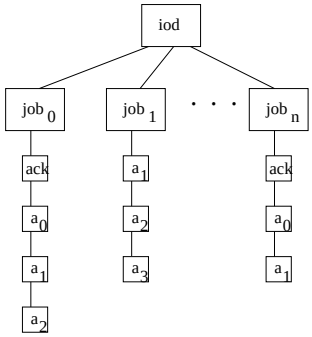



**Figure 6.** *Jobs in service*

us the flexibility to implement our scheduling algorithms. Our scheduling algorithms will influence the system by choosing what jobs will be serviced and in what order.

Once a job is selected the server can perform all or part of the sequence of accesses for the selected job before selecting another job to service. Generally, however, the server should not block waiting for any single job to complete when other jobs could be serviced. Thus the server normally performs only the accesses or parts of acceses for a selected job that will not cause the server to block.

The I/O server refers to the access list of the selected job in order to determine what operation should be performed next. In the case of a read operation the I/O server first uses `mmap()` to map the data region into its address space. This is performed on a region of 128 Kbytes in the implementation tested, which through testing was found to be a reasonable trade-off between mapping too large a region and performing too many mapping operations on this particular system. Once the region is mapped into memory, `send()` is used to send the data from the desired region to the remote host. The `O_NONBLOCK` flag is set on the socket using `fcntl()` prior to sending data so that the server will not block on the socket. When a new region of the file is needed by this connection for I/O, the old region is unmapped with `munmap()` before the new region is mapped.

### 3.4. *Limitations*

While PVFS is the most complete open-source parallel file system available and our best option for experimentation, its architecture does place some limitations on our ability to make scheduling decisions and to implement previous schemes.

First and foremost PVFS servers operate at the user level. This means that all scheduling actions are in some sense indirect; we can put data into a socket buffer or perform a `write()` call to request that data be stored on disk, but in the end the kernel

makes the final decision on when operations happen. This is particularly troublesome in the case of writes; it is very difficult to force the Linux kernel to write data to disk.

Similarly, since iods store data in files, it is more natural to make spatial locality decisions based on file offsets. It is possible that blocks in the file are not placed sequentially, but previous work has shown this technique to be effective [SEA 95]. In our experiments we perform all operations on a single file in order to make the most of file offset information. The use of `mmap()` for reading data and `write()` for writing data prevents servers from truly knowing when they will block. They instead rely on the kernel buffering of data to help provide overlap between disk and network I/O.

At the time these tests were performed, PVFS did not support generalized non-contiguous requests. Thus we were unable to utilize noncontiguous requests for our random access workload. Instead multiple contiguous requests were used, and this limits the ability of the server to organize the data movement. Finally, the stream-based data transfer method implemented in PVFS constrains the order in which data may be returned to the client. This places an additional constraint on the server.

These characteristics do inhibit our ability to extract the highest performance from the underlying components. However, our goal here is not to show the highest possible performance but instead to show that matching scheduling algorithm to workload can provide a performance win. This list of limitations then serves as a starting point for further improvement of the PVFS system.

## 4. Scheduling algorithms

For our experiments we have implmented four scheduling algorithms that work with PVFS. We will designate these algorithms *Opt 1 - 4*. The first algorithm is the default algorithm for PVFS and is optimized for network access. The other three are increasingly disk-oriented, focusing on reducing disk access time. As previously mentioned, PVFS cannot send data out of order for a given job, but it can control the order that multiple jobs are serviced, and these different algorithms reflect this.

*Opt 1* is the PVFS default stream-based algorithm. Using this algorithm, the I/O server first checks to see which network connections are ready for service using a `select()` call and then services each ready socket in FCFS order until the connection is no longer ready for service. When all ready sockets have been serviced this process is repeated. Due to limited buffering in the network subsystem, this algorithm tends to do a pretty good job of load balancing service to the various jobs. It does not consider disk access order at all, and may result in significant disk head movement when servicing multiple requests. *Opt 1* has the advantage of being the only algorithm that does not need to sort the jobs, making its processing phase the fastest of the four algorithms.

*Opt 2* is a modification of *Opt 1*. As in *Opt 1*, first the server selects all sockets ready for service, then it sorts the sockets based on the offset of the next access from

| Round | $J_0$ | $J_1$ | $J_2$ | $J_3$ | $J_4$ | Schedule(in order) |
|---|---|---|---|---|---|---|
| $r_0$ | | 100 | 400 | | 300 | $J_1$, $J_2$, $J_4$ |
| $r_1$ | 900 | | 500 | 200 | | $J_0$, $J_2$, $J_3$ |
| $r_2$ | 1000 | 300 | | | 400 | $J_0$, $J_1$, $J_4$ |

**Table 1.** *Example of* Opt 1 *scheduling*

| Round | $J_0$ | $J_1$ | $J_2$ | $J_3$ | $J_4$ | Schedule(in order) |
|---|---|---|---|---|---|---|
| $r_0$ | | 100 | 400 | | 300 | $J_1$, $J_4$, $J_2$ |
| $r_1$ | 900 | | 500 | 200 | | $J_2$, $J_3$, $J_0$ |
| $r_2$ | 1000 | 300 | | | 400 | $J_0$, $J_1$, $J_4$ |

**Table 2.** *Example of* Opt 2 *scheduling, starting with* $O_{last}$*=100*

the start of the file. Sorting starts at the last offset accessed for the file ($O_{last}$), continues to the largest offset for a ready job, and then continues with the jobs whose offsets are smaller than the last offset. In our experiments, all requests are accessing the same file, and if the operating system does a reasonable job of clustering file data on the disk, access to disk should be in a more efficient order than in *Opt 1*.

*Opt 3* further modifies *Opt 2* by removing some jobs that are ready for network service because their file offset differs too much from the other jobs. Under this option, a logical window is defined that spans file offsets in a range around last offset accessed. The center of the window is defined to be the last offset accessed, so for a given window size $W_{sz}$ and last offset $O_{last}$, values in the range of $O_{last} \pm \frac{W_{sz}}{2}$ are inside the window. The value of $W_{sz}$ is selectable at compile time. With this optimization all requests that are "close" together are serviced, while requests that would access a distant part of the file are not. In the event that no requests fall into this window, the closest request is serviced instead.

*Opt 3* is based on a window scan scheduling algorithm (WSCAN) and tends to allow the system to more effectively use spatial locality and caching, especially when the operating system is prefetching based on file offset. On the other hand this algorithm might not load balance as well as *Opt 1* or *Opt 2* because some jobs that are ready for service might not get serviced for some time. In addition, one has to consider the potential for starvation when a job is specifically excluded from service. Starvation is possible with this algorithm; however, this algorithm will never wait on a job that is not ready as long as a ready job is available.

*Opt 4* is the only algorithm that does not consider whether a job is ready for network service. This algorithm sorts all of the jobs based on the offset of the next access, starting from the offset of the last access and services the jobs in that order, waiting for a network connection to become ready when neccessary. This is an implementation of the shortest seek time first (SSTF) algorithm [DEN 67]. This algorithm is extremely likely to starve some jobs, and we will see the results of this when we later discuss fairness. The purpose of including *Opt 4* in our study is to ascertain the max-

| Round | $J_0$ | $J_1$ | $J_2$ | $J_3$ | $J_4$ | Schedule(in order) |
|---|---|---|---|---|---|---|
| $r_0$ | | 100 | 400 | | 300 | $J_1$, $J_4$, $J_2$ |
| $r_1$ | 900 | | 500 | 200 | | $J_3$, $J_2$ |
| $r_2$ | 900 | 300 | | | 400 | $J_1$, $J_4$ |
| $r_3$ | 900 | | | | | $J_0$ |
| $r_4$ | 1000 | | | | | $J_0$ |

**Table 3.** *Example of* Opt 3 *scheduling with $W_{sz}$=600, starting with $O_{last}$=100*

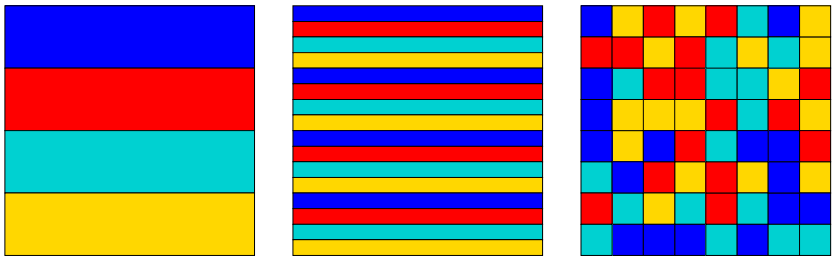
| Round | $J_0$ | $J_1$ | $J_2$ | $J_3$ | $J_4$ | Schedule(in order) |
|---|---|---|---|---|---|---|
| $r_0$ | | 100 | 400 | | 300 | $J_1$ |
| $r_1$ | 900 | | 400 | 200 | 300 | $J_3$ |
| $r_2$ | 900 | 300 | 400 | | 300 | $J_1$, $J_4$ |
| $r_3$ | 900 | | 400 | | 400 | $J_2$, $J_4$ |
| $r_4$ | 900 | | 500 | | | $J_2$ |
| $r_5$ | 900 | | | | | $J_0$ |
| $r_6$ | 1000 | | | | | $J_0$ |

**Table 4.** *Example of* Opt 4 *scheduling, starting with $O_{last}$=100*

imum possible benefit we can obtain from disk-oriented scheduling within the PVFS system.

Consider the following example. Assume there are 5 jobs, each accessing the same file. Assume that the last access was at file position 100. In each row, if a number is present, then the socket for that job is ready for service, and the number indicates the file offset being accessed. The schedule shows the jobs that will be serviced, in order. Note that the rounds shown are not intended to represent a fixed amount of time, thus the number of rows does not imply that the resulting schedule neccessarily takes longer to complete. Rather, this shows the different scheduling rounds corresponding to passes through the jobs.

Under *Opt 1*, the ready jobs are scheduled FCFS in each round (Table 1). Under *Opt 2*, the ready jobs are scheduled in offset order in each round (Table 2). The schedule for *Opt 3* (Table 3) bears some explanation. In the first round all three ready jobs fall within the window, so all they are scheduled in offset order. In the second round, Jobs 2 and 3 are within the window , but Job 0 is not. These jobs are serviced in file offset order. In the third round Jobs 1 and 4 are within the window and are scheduled for service. In the fourth round no jobs are within the window, so the closest (and only) job, Job 0, is serviced. In the fifth round Job 0 is still within the window and service is completed. Finally, under *Opt 4* (Table 4), the job with the next offset larger than the previous is scheduled, even if that job isn't ready yet. Thus all blocks are processed in order unless a job arrives after processing has already passed its first offset.



a) Contiguous b) Strided c) Random

**Figure 7.** *Test workloads*

## 5. Workloads

In Section 3 we covered PVFS, the parallel file system used in our experiments. In this section we will discuss the workloads we examined in order to ascertain the effectiveness of our scheduling algorithms at servicing three different test patterns:

– single block accesses

– strided accesses

– random block accesses

Three test applications were used in this workload study. These workloads represent a number of access patterns seen in some traditional applications, particularly ones operating on dense multidimensional matrices. We include both single contiguous block access, strided access, and multiple random block access workloads in order to cover a wide range of possible workloads. Figure 7 shows the general pattern of access for these workloads shown as row-major two-dimensional structures.

In all cases a MPI application was used to create a set of application tasks that independently access a PVFS file system using the native PVFS libraries. In all tests a single PVFS file was used to store data.

In the single block access tests, contiguous regions of the data file were simultaneously accessed by each task. The tasks synchronize before the access, and the times to complete access were recorded. We consider the longest service time of any one task to be the application service time, as this is the time the application as a whole would have to wait if operations were collective. We also calculate the mean service time for all tasks and the variance of task service time. Patterns such as these are seen in applications accessing dense matrices in a block manner, in checkpoint applications, and in some out of core applications.

In the strided access tests, multiple noncontiguous regions of the data file were simultaneously accessed by each task using a single, simple-strided, operation. Again

the tasks synchronize before the access, and the times to complete the operation were recorded. As before we consider the longest task service time to be the application service time, and we also calculate mean service time and the variance. Strided accesses are often seen as a result of row cyclic distribution of data sets and access to portions of records of a fixed size.

The purpose of the random block access tests is to observe the system serving an application with an irregular access pattern. The file is logically divided into a number of blocks, and these blocks are randomly assigned to the tasks in the application such that each task will access an equal number of blocks. Tasks are synchronized before any accesses begins, and tasks access all blocks in random order using a native PVFS operation to access each block, one block at a time. The time to access all blocks is recorded for each task, the largest of these total times is considered the application service time, and mean service time and variance are also calculated. This pattern might be created by an application accessing pieces of a multidimensional data set or reading arbitrary records from a large database.

In all cases, our goal is to analyze the effects of the four scheduling algorithms on the performance of the system, using three metrics: application service time, mean task service time, and task service time variance. All of these metrics are important in one situation or another. For a system serving single parallel applications, application service time might be the most appropriate metric. For a system running multiple parallel applications or many serial ones, mean task service time might be a more appropriate metric. Task service time variance is the variance observed between the service times of tasks concurrently accessing the system. This value is an indicator of fairness; a low value indicates that service is distributed fairly between jobs, while a high variance often indicates that starvation is occurring. This is of particular importance in real-time applications.

## 6. Experimental results

The Beowulf machine on which this work was performed is a 17-node cluster at Clemson University. The cluster was configured as follows. Each node has a single Pentium 150 MHz CPU, 64 Mbytes EDO DRAM, 64 Mbytes local swap space, a 2.1 GByte IDE disk, and Tulip-based 100 Mbit Fast Ethernet card. The nodes are connected by an Intel Fast Ethernet switch in full-duplex mode.

One node runs the PVFS manager daemon and handles interactive connections while the other nodes are used as compute nodes, I/O nodes, or both. Each node runs Linux v2.2.13 with tulip driver v0.89H. The IDE disks provide approximately 4.5 Mbytes/sec with sustained writes and 4.2 Mbytes/sec with sustained reads, as reported by Bonnie, a popular UNIX file system performance benchmark [BRA ]. When idle, approximately 8 Mbytes of memory are used on each node by the kernel and various system processes, including PVFS, leaving approximately 56 Mbytes of space

that could be used by the system for I/O buffers. The window size for tests with *Opt 3* was set to 56 Mbytes. Our test applications were compiled with MPICH v1.2.0.

For these tests two nodes were used as I/O nodes, and 14 nodes were used for computation. This combination allowed us to separate the I/O nodes from the compute ones, to provide a number of simultaneous jobs that the I/O nodes can schedule, and to ensure that no single-disk optimizations are used on the I/O nodes (mapping PVFS file locations to local file ones is simpler in the single-disk case).

Varying scheduling algorithms for write workloads showed only limited benefit, so this data is not presented here. Interested readers are directed to [ROS 00] for this data. For read tests, before each run a local data file larger than the size of a node's memory was read in its entirety on each I/O node to remove all PVFS file data from cache. A separate run of read tests was additionally performed in which we allowed file data to remain in cache (i.e. did not read a local data file between runs). These results are compared to small accesses without cache in the following sections as well.

We first discuss the application and task service time metric for each of the tests, presenting output for all four of the algorithms described earlier in this work. The total data accessed on a single I/O node is shown on the X axis, and service time is provided on the Y axis. The data presented is the average of three test runs. Following this discussion we cover the issue of fairness with respect to our algorithms and the tested workloads.

### 6.1. *Single block accesses*

The results for single block accesses are shown in Figure 8. When data is uncached we see that *Opt 4* provides by far the lowest average task service time, beating the worst performers by as much as 28%. Recall that *Opt 4* is our algorithm most similar to disk-directed I/O; only the request closest to the last accessed file position will be serviced. It is apparent that we are more effectively utilizing disk resources with *Opt 4* in this case. Application service times are consistent across all algorithms.

When file data is cached we see that *Opt 4* still results in the best task service time (beating the worst performer by 30$); however, this comes at the cost of a substantially higher application service time than the other algorithms. In this case *Opt 3* seems to be a more appropriate overall choice in this case. Recall that *Opt 3* relaxes the strict ordering of requests, allowing for jobs within a window to be serviced and always allowing the nearest ready job to be serviced. When cached data is available, *Opt 3* provides a better application service time while also resulting in a competitive mean task service time.

When examining the small, cached service time graphs one can see a uniform change in performance at approximately the 56 Mbyte point. This is the point at which we begin to exceed our cache size and start hitting disk. This trend will be seen throughout all the test results.

### 6.2. *Strided accesses*

Results are shown for strided accesses in Figure 9. For our strided access pattern we arbitrarily chose to access 16 disjoint regions with each task access. The size of the disjoint regions was varied throughout the tests.

When servicing uncached strided read requests we see that *Opt 1* provides the lowest mean task service time by as much as 14%, most likely due to the fact that file data is interleaved between the application tasks, resulting in *Opt 1* performing similarly to *Opt 4*, but without its ordering restrictions. When data does reside in cache we see less benefit from using *Opt 1* over other algorithms, although it does appear to still be the best choice. All algorithms result in approximately the same application read service times over the wide range of access sizes.

The file byte ordering limitation imposed by PVFS is particularly inhibiting for disk-oriented algorithms servicing this type of workload. Since in practice all jobs are not started simultaneously, it is likely that the first job to arrive will be partially serviced before others are started. These new jobs might have data located near the data accessed for the first job, but those portions of the new jobs may not be serviced until their point in the byte ordering is reached.

### 6.3. *Random block accesses*

We chose to study random block access in addition to tests focusing on known patterns. An interesting characteristic of these tests is that only a fraction of the total data to be accessed is being requested by jobs in service at any one time because multiple operations are required to access the randomly distributed blocks (using native PVFS calls). This is in contrast to the previous tests, where all data to be accessed is requested in single calls. The result of this is that the total size of requests at any point in time is no more than $S_{tot}/N_{blks}$, where $S_{tot}$ is the total amount of data that will be accessed and $N_{blks}$ is the number of blocks into which the data is split (per task). Tests were run for $N_{blks}$ values of 16 and 32. Results for both were similar, so only the results from 32 blocks per task are presented here. Interested readers are directed to [ROS 00] for this data.

Figure 10 presents the results for 32 blocks per task. This is the first set of tests for which we see a significant difference in uncached application service times between algorithms. This is a clear indicator that we are reducing the amount of work performed by the underlying I/O system using the disk-oriented algorithms *Opt 3* and *Opt 4*. These two algorithms also outperform the others in task service time for large accesses. When caching is in effect we again note that *Opt 1* becomes extremely competitive, outperforming *Opt 4* by as much as 10%.

### 6.4. *Fairness*

For some applications it is highly desirable for service times to be predictable. For these applications fairness is of extreme importance, as starvation will lead to unpredictable service times. In Figure 11 we show the task service time variance for our tests. We show the results for small, cached data and large accessses only; small uncached results followed the trends of the large accesses.

When looking at the small, cached graphs we observe a uniform trend of spikes in the variance graphs as we enter the 56-64 Mbyte range. This is to be expected as it is at this point that accesses first begin to result in cache misses.

Additionally we see that in general *Opt 1* and *Opt 2* provide better (lower) variance than *Opt 3* and *Opt 4*. We expected this, as *Opt 1* and *Opt 2* both service all ready jobs on each pass, while *Opt 3* and *Opt 4* allow certain jobs to starve. *Opt 2* also tends to provide more predictable performance than *Opt 1*; this is undoubtedly due to its more fair method of cycling through file locations.

The notable exception to this is large, strided access. In this case *Opt 3* and *Opt 4* provide the most predictable performance. This is due to their more strict enforcement of ordering; by ordering accesses by their file locations they enforce fair service when job data happens to be organized in a strided manner.

As noted previously, *Opt 3* and *Opt 4* both introduce the possibility of starvation. Particularly in the case of *Opt 4* workarounds to avoid this situation would need to be added if the algorithm were to be used in a production system. The *Opt 3* algorithm already implements a degree of starvation-avoidance, and based on this we believe that the addition of such workarounds would have a minimal impact of performance in common workloads.

## 7. Conclusions and future work

As a whole we see that we are able to affect application service time in only a small number of cases, and in general our scheduling changes had little effect on write workloads. This is not completely surprising considering the limitations discussed in Section 3.4.

However, we consistently see benefits to applying certain algorithms in read cases with respect to task service time. In particular, for situations where uncached contiguous regions are being serviced, *Opt 3* and *Opt 4* show the best performance. On the other hand, for cases where significant fractions of data are cached we see that *Opt 1* performs the best. This is likely to be in part due to the algorithm itself and in part due to the time it saves by not sorting jobs in service.

For our strided read workload we see that *Opt 1* performs best as well. This is partially due to the implicit interleaving in the requests. However, it is likely that our predefined ordering of request data is also a factor. By this we mean that the order

in which request data is returned to the requesting task by PVFS is always in order of monotonically increasing file byte offset. This constrains the order in which we can service the pieces that make up strided requests, which in turn limits the ability of more disk-oriented algorithms to best access the disk.

If one is most concerned with "fair" service, *Opt 2* would be the best choice. It provided the lowest variance for almost all the tested cases with little loss in application or task service time. The only exception to this was the strided case with large accesses, in which *Opt 3* would be the best choice.

Overall we see that no single algorithm performs best over the range of workloads, but instead that algorithms tend to be appropriate to a workloads fitting certain characteristics. This indicates that rather than relying on a single algorithm, servers should instead have a collection of algorithms at their disposal. These algorithms could be selected by administrators based on expected usage, but a more effective approach would be to automatically select algorithms in response to workload characteristics at run-time.

Adaptive selection of policies for caching and prefetching have already been developed [MAD 96, MAD 97]. Our intention is to build a complementary system for selecting scheduling algorithms. A behavioral model incorporating workload characteristics such as the extent and size of requests in service and system effects such as available cache would be coupled with heuristics for algorithm selection. One concern with such a solution is that the overhead of performing the calculations at run-time might outweigh the potential gains, but previous work in kernel-level scheduling on similar machines indicates that performing additional calculations at run-time should be feasible [GEI 97].

We have recently implemented noncontiguous requests for PVFS. This capability extends the application's ability to describe the overall desired I/O pattern to the server, which should enable us to better schedule service. Additional studies are necessary to validate this claim.

## Acknowledgements

This work was funded in part by NASA grants NAG5-3835 and NGT5-21 and in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

## 8. References

[BOR 93] BORDAWEKAR R., DEL ROSARIO J. M., CHOUDHARY A., "Design and Evaluation of Primitives for Parallel I/O", *Proceedings of Supercomputing '93*, Portland, OR, 1993, IEEE Computer Society Press, p. 452–461.

[BRA ] BRAY T., "Bonnie File System Benchmark", `http://www.textuality.com/bonnie/`.

[CAR 00] CARNS P. H., LIGON III W. B., ROSS R. B., THAKUR R., "PVFS: A Parallel File System for Linux Clusters", *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, October 2000, USENIX Association, p. 317–327.

[CHO 94] CHOUDHARY A., BORDAWEKAR R., HARRY M., KRISHNAIYER R., PONNUSAMY R., SINGH T., THAKUR R., "PASSION: Parallel And Scalable Software for Input-Output", report num. SCCS-636, September 1994, ECE Dept., NPAC and CASE Center, Syracuse University.

[DEN 67] DENNING P. J., "Effects of scheduling on file memory operations", *AFIPS Spring Joint Computer Conference*, April 1967.

[GEI 97] GEIST R., ROSS R., "Disk Scheduling Revisited: Can $O(N^2)$ Algorithms Compete?", *Proceedings of the 35th Annual ACM Southeast Conference*, April 1997.

[KOT 95] KOTZ D., NIEUWEJAAR N., "File-System Workload on a Scientific Multiprocessor", *IEEE Parallel and Distributed Technology*, vol. 3, num. 1, 1995, p. 51–60, IEEE Computer Society Press.

[KOT 97] KOTZ D., "Disk-directed I/O for MIMD Multiprocessors", *ACM Transactions on Computer Systems*, vol. 15, num. 1, 1997, p. 41–74, ACM Press.

[KRY 93] KRYSTYNAK J., NITZBERG B., "Performance Characteristics of the iPSC/860 and CM-2 I/O Systems", *Proceedings of the Seventh International Parallel Processing Symposium*, Newport Beach, CA, 1993, IEEE Computer Society Press, p. 837–841.

[LIG 96] LIGON W. B., ROSS R. B., "Implementation and Performance of a Parallel File System for High Performance Distributed Applications", *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, August 1996, p. 471–480.

[MAD 96] MADHYASTHA T. M., REED D. A., "Intelligent, Adaptive File System Policy Selection", *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, IEEE Computer Society Press, October 1996, p. 172–179.

[MAD 97] MADHYASTHA T. M., REED D. A., "Input/Output Access Pattern Classification Using Hidden Markov Models", *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, San Jose, CA, November 1997, ACM Press, p. 57–67.

[NIT 92] NITZBERG B., "Performance of the iPSC/860 Concurrent File System", report num. RND-92-020, December 1992, NAS Systems Division, NASA Ames.

[RID 97] RIDGE D., BECKER D., MERKEY P., STERLING T., "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs", *Proceedings of the 1997 IEEE Aerospace Conference*, 1997.

[ROM] "ROMIO: A High-Performance, Portable MPI-IO Implementation", `http://www.mcs.anl.gov/romio`.

[ROS 00] ROSS R. B., "Reactive Scheduling for Parallel I/O Systems", PhD thesis, Dept. of Electrical and Computer Engineering, Clemson University, Clemson, SC, December 2000.

[SEA 95] SEAMONS K. E., CHEN Y., JONES P., JOZWIAK J., WINSLETT M., "Server-Directed Collective I/O in Panda", *Proceedings of Supercomputing '95*, San Diego, CA, December 1995, IEEE Computer Society Press.

[TAK 99] TAKI H., UTARD G., “MPI-IO on a Parallel File System for Cluster of Workstations”, *Proceedings of the First IEEE International Workshop on Cluster Computing*, 1999.

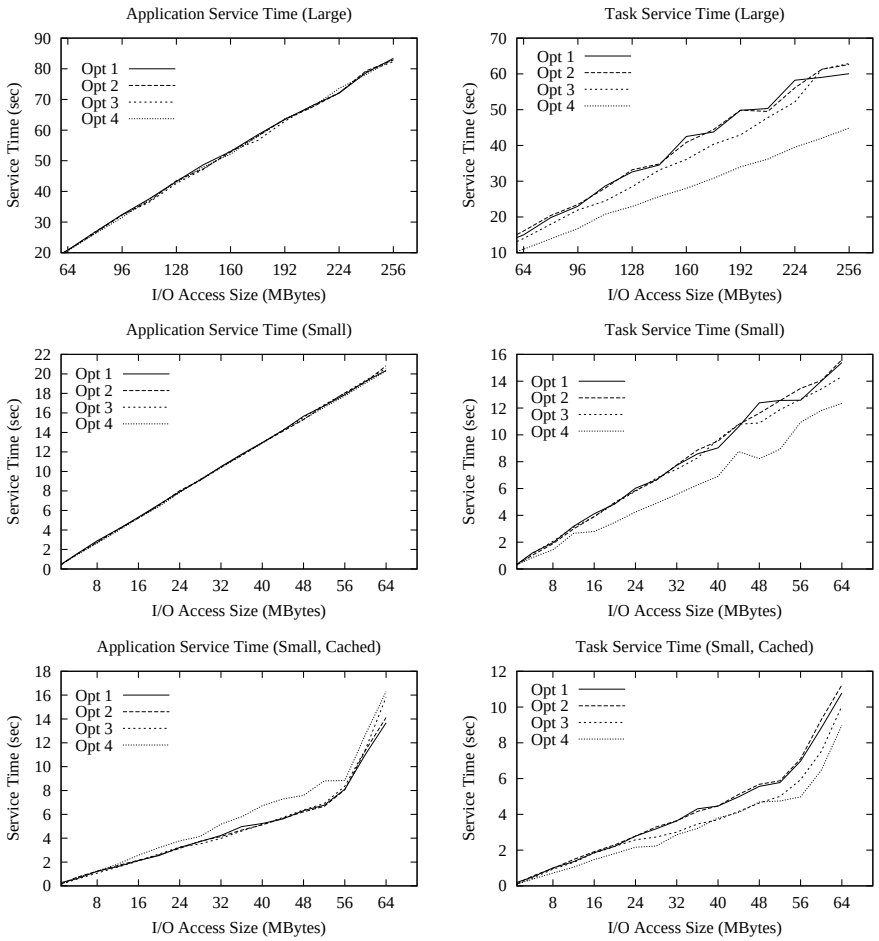



**Figure 8.** *Single block read performance*

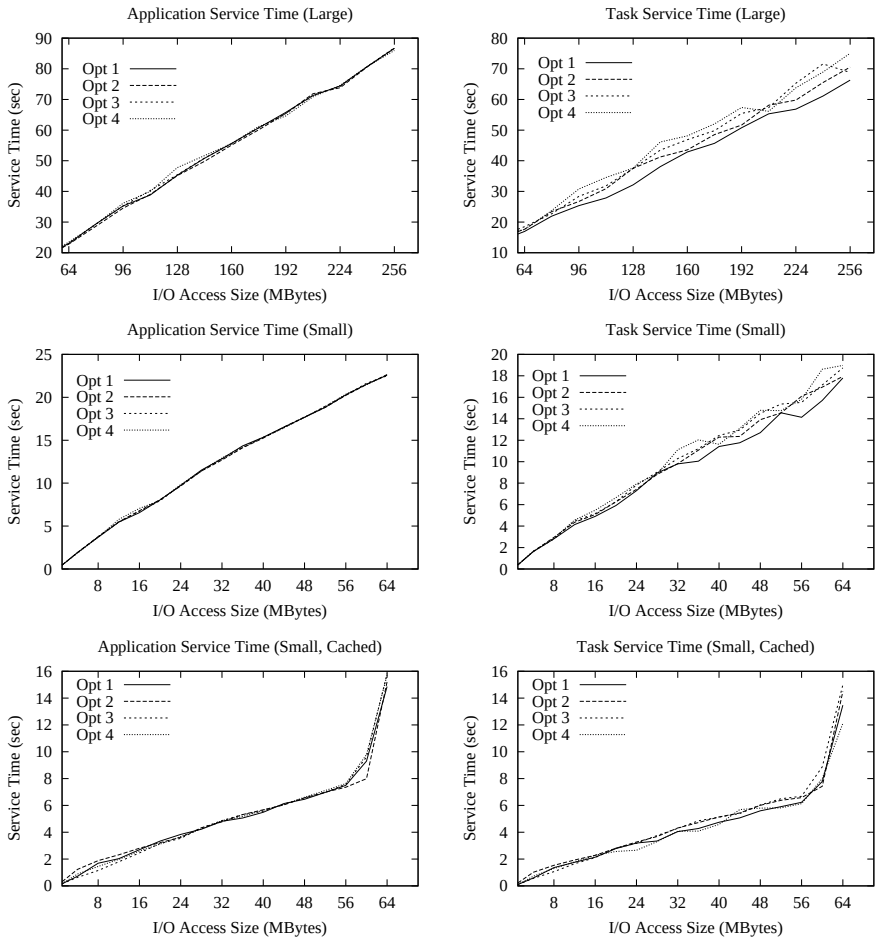



**Figure 9.** *Strided read performance*

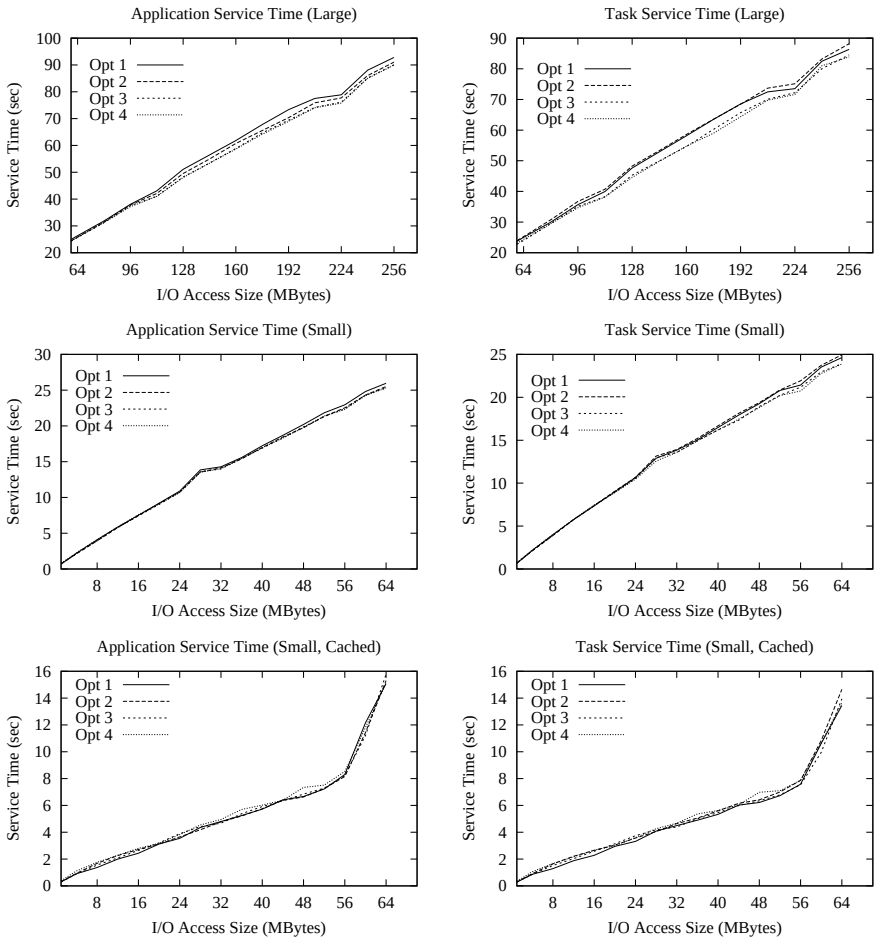



**Figure 10.** *Random (32 block) read performance*

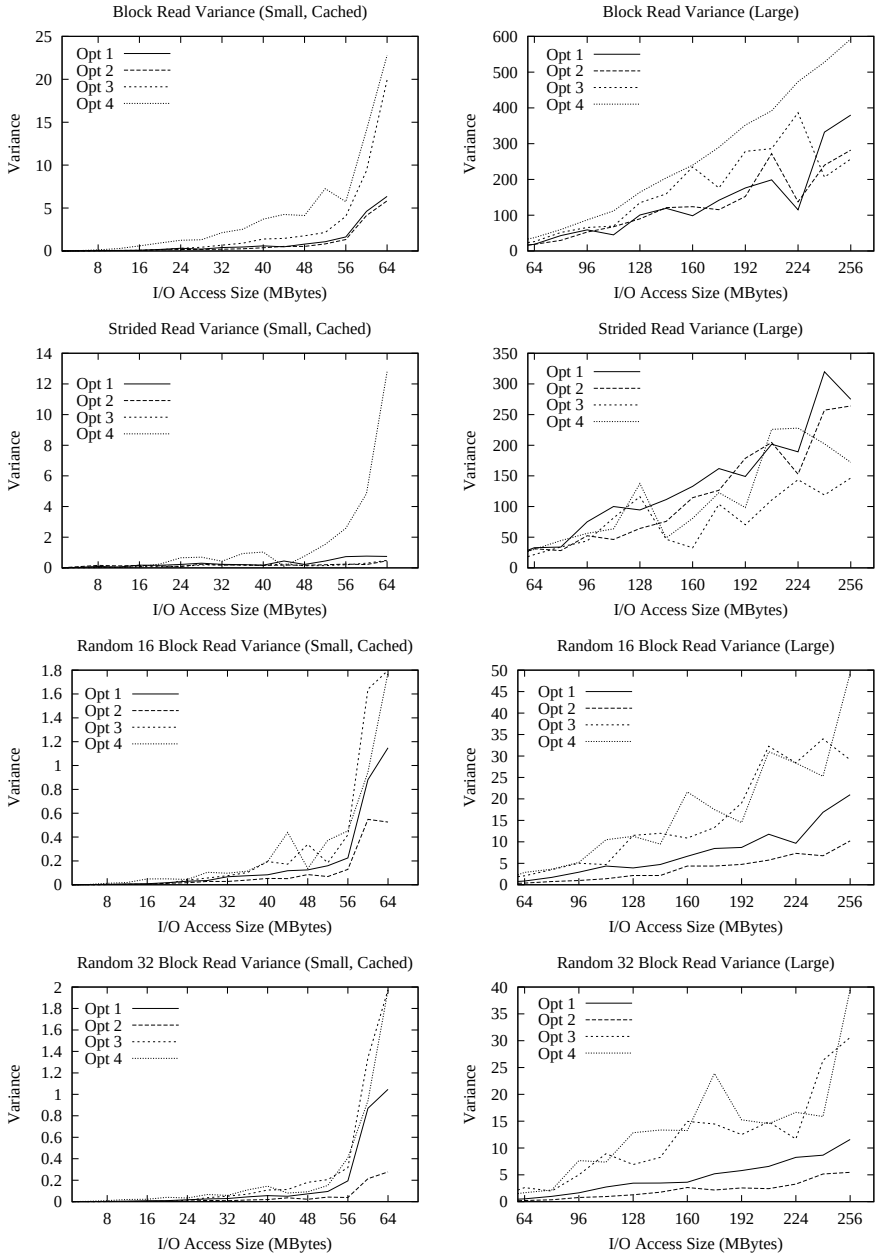



**Figure 11.** *Task service time variance for reads*