

Informed Prefetching of Collective Input/Output Requests

Tara M. Madhyastha

Garth A. Gibson Christos Faloutsos

tara@cse.ucsc.edu

{garth,christos}@cs.cmu.edu

Department of Computer Science

School of Computer Science

University of California

Carnegie Mellon University

Santa Cruz, CA 95064

Pittsburgh, Pennsylvania 15213

Abstract

Optimizing collective input/output (I/O) is important for improving throughput of parallel scientific applications. Current research suggests that a specialized collective application programming interface, coupled with system-level optimizations, is necessary to obtain good I/O performance. Unfortunately, collective interfaces require an application to disclose its entire access pattern to fully reorder I/O requests, and cannot flexibly utilize additional memory to improve performance. In this paper we propose and analyze a method of optimizing collective access patterns using informed prefetching that is capable of exploiting any amount of available memory to overlap I/O with computation. We compare this approach to disk-directed I/O, an efficient implementation of a collective I/O interface. Moreover, we prove that under certain conditions, a per-processor prefetch depth equal to the number of drives can guarantee sequential disk accesses for any collectively accessed file. In empirical studies, a prefetch horizon of one to two times the number of disks per processor is sufficient to match the performance of disk-directed I/O for sequentially allocated files. Finally, we develop accurate analytical models to predict the throughput of informed prefetching for collective reads as a function of the per-processor prefetch depth.

1 Introduction

Poor input/output (I/O) performance is one of the primary obstacles to effective use of high performance multiprocessor systems. In an effort to provide high throughput commensurate with compute power, current multiprocessor systems provide multiple disks or disk arrays attached to I/O processors that communicate with compute processors through a fast network (see Figure 1). A parallel file system manages parallel files that are declustered across multiple I/O processors. Unfortunately, parallel file systems have been unable to consistently deliver peak hardware performance to applications with widely varying I/O requirements, and I/O remains a critical bottleneck for a large class of important scientific applications.

Many studies have shown that parallel I/O performance is extremely sensitive to the file access pattern, which can be extremely complex. Each thread of a parallel application accesses a portion of a parallel file, declustered over multiple

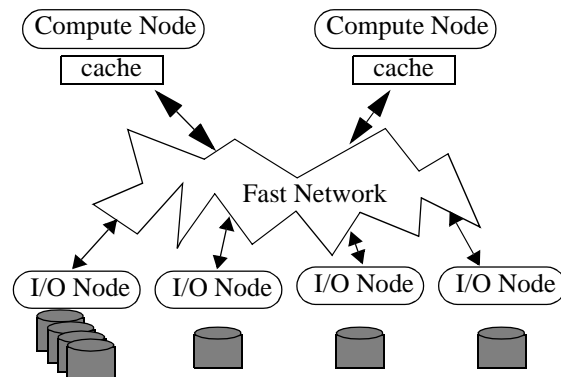


Figure 1: **System architecture.** Compute nodes and input/output nodes (each of which manages a disk or disk array) communicate through a fast network interconnect.

disks, with a local access pattern. The temporal and spatial interleaving of the local access patterns is the global access pattern. To optimize application throughput, it often helps to have information about the global access pattern; thus, an important research area in parallel file system design is determining what access pattern information needs to be specified by an application to provide good performance.

One important class of access patterns is collective I/O. Suppose the threads of a parallel application simultaneously access portions of a shared file, and no thread can proceed until all have completed their I/O. If the threads use a UNIX-style interface, a separate system call is required for each disjoint portion. This can result in non-sequential accesses, and consequently, poor performance at the I/O processors. Providing high-level access pattern information to the file system through an application programming interface (API) allows disks to reorder requests, servicing them to maximize throughput. This motivates optimizations such as two-phase [7] and disk-directed I/O [11]; given global, high-level knowledge that some data must be read or written before all the processors can proceed, the I/O operations can be reordered to occur as an efficient collective. In recognition of its importance to high-performance I/O, an interface for collective I/O is specified as an extension to the Scalable Input/Output (SIO) initiative low-level API [5].

A more general technique for improving performance by providing access pattern information to a file system is through informed prefetching. The application constructs hints describing future accesses that a prefetcher uses to issue I/O requests in advance of the demand fetch stream. Although a main advantage of prefetching is its ability to overlap I/O with computation, deep prefetching enables better disk scheduling, improving throughput analogously to a collective I/O implementation that sorts disk requests. Unlike an efficient collective I/O implementation, the quality of disk scheduling is a function of the prefetch depth.

Given that a parallel application must perform a collective read, and provided that each processor has sufficient time to overlap the I/O with computation and some amount of memory to issue prefetch requests, we investigate how to utilize this memory using general informed prefetching techniques to achieve throughput similar to an efficient implementation of collective I/O. We derive models for expected throughput using prefetching and validate these experimentally. We show that with only a small amount of additional memory that scales with the number of disks, informed prefetching can outperform an efficient collective I/O implementation.

The remainder of this paper is organized as follows. We describe the complexities of collective access patterns and our target application in §2. In §3 we present the state of the art optimizations for collective I/O. In §4 we discuss prefetching as an alternative to a collective I/O interface, and describe models for prefetching throughput based on the available memory. We present our experimental evaluation of prefetching and collective I/O using a simulation infrastructure in §5. In §6 we survey related work, and we conclude with directions for future research in §7.

2 Problem Specification: Collective Access Patterns

We define a collective access pattern to be a special global access pattern where processors simultaneously access portions of a parallel file. A parallel file has several layers of abstraction. At the application level, processors may impose some structure on the logical file to determine how to partition it among themselves. At the system level, there is usually a canonical order for logical file blocks or records. These units are in turn stored on disk drives according to some distribution. Finally, the placement of blocks on a single drive may vary from sequential to random, depending on the placement strategy.

Depending on how the logical file blocks are stored on disk and distributed among processors, the global access pattern, and consequently, queues at each drive, can vary dramatically. Figure 2 illustrates how logical file blocks may be striped round-robin across drives, and distributed among processors according to three different logical file views (interleaved sequential, 1-D block, and 3-D block) that have been used in scientific applications.

Using a UNIX-style interface, processors must at least issue a separate request for each contiguous portion. We assume that in a collective read, processors synchronize and each processor begins reading its portion, block by block, in increasing logical block order. Figure 3 depicts an abstraction of how the 1-D block decomposition of Figure 2 evolves over time. We can see that disk utilization, and consequently throughput, is poor. This example is very small, so individual drive accesses are still sequential. However, when processors simultaneously access different locations of a larger parallel file, it becomes difficult to exploit sequentiality at the drives.

3 Collective I/O Implementations

As we have described, collective access to parallel files is often inefficient from an I/O performance standpoint. Each process accesses its own disjoint portion, but to fully exploit disk bandwidth, the file system must have knowledge of the global, coordinated access pattern. If the application can specify that the component transfers of the collective operation may be executed in any order, because the application will not progress until the entire operation has been completed, the

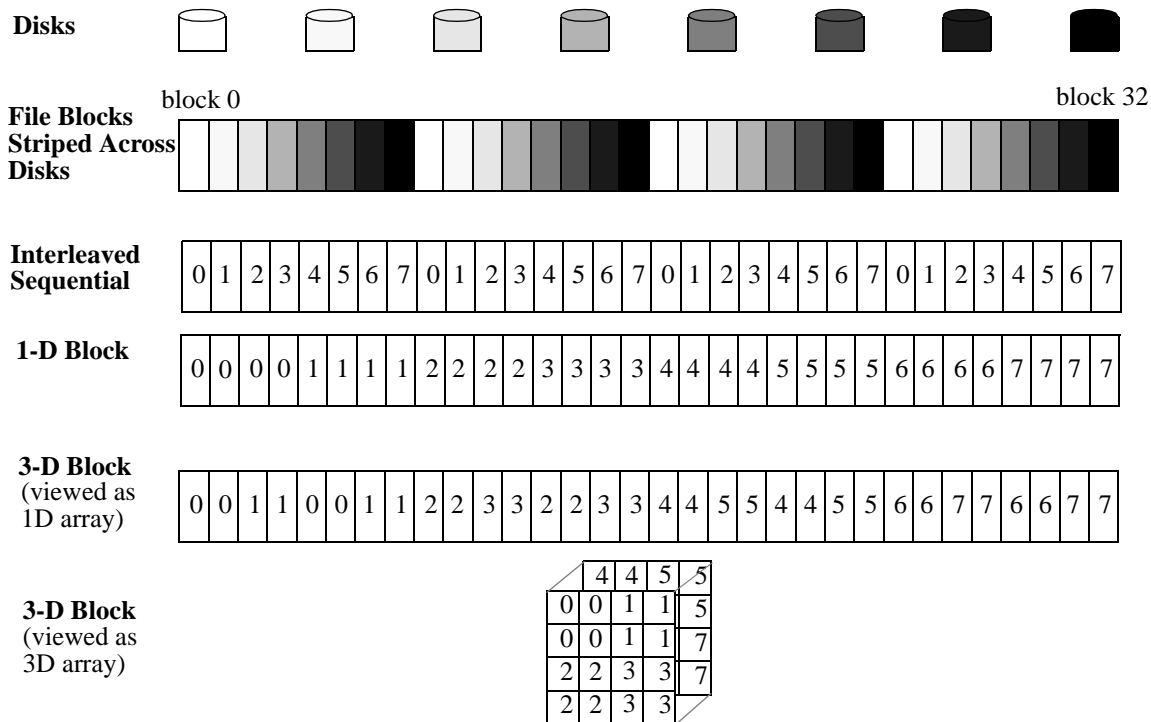


Figure 2: **Three examples of logical file views.** A file, viewed as an array of blocks, may be allocated to processors many ways. The diagrams above show three such logical file views. The numbers identify processors that access the denoted file regions and processors access blocks in logically increasing order. For example, in the interleaved sequential access pattern, each processor will access all the blocks stored on a single disk.

file system can minimize disk transfer time by reordering requests to match the file layout.

The SIO low-level API [5] includes an extension interface for collective I/O. One task of the application creates a collective I/O, specifying the number of participants, whether the operation is a read or write, the number of iterations, and (optionally) what portions will be operated on. Participating application tasks issue calls to join the collective; each specifies a list of corresponding file regions and memory locations that will be involved in the transfer. The transfer is asynchronous; the file system can either wait for all participants to join the collective before initiating the transfer, or service each request immediately.

There are several implementation alternatives for collective I/O that can be categorized as user-level or system-level. User-level libraries to support collective I/O, such as two-phase I/O [7] have processors access files in long contiguous portions and permute the data among themselves to correspond to the processor data decomposition. Two major performance problems with this approach are that the user-level library cannot exploit information about the physical disk layout, and the permutation phase is not overlapped with I/O. System-level implementation approaches provide system support for collective operations; one approach that addresses these problems is disk-directed I/O [11]. Disk-directed I/O allows the I/O processors to sort the physical block requests and transfer the requested blocks directly to the requesting processors. Besides application buffer space to receive the outstanding request, no additional memory is necessary.

4 Proposed Method: Informed Prefetching

The confluence of application-specified information and system-level support allows optimal scheduling of collective I/O requests. However, the underlying principle (gathering and sorting requests from processors in a globally optimal manner) can improve the performance of many parallel I/O access patterns. Prefetching can provide these benefits for collective I/O using a more general framework that does not rely on specialized system optimizations.

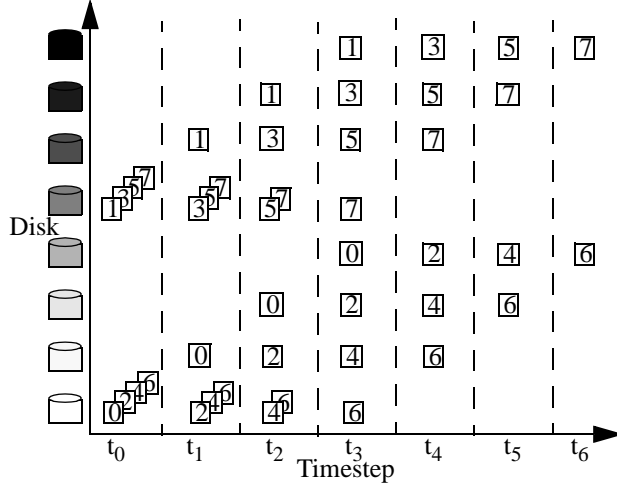


Figure 3: **Disk queues over time.** Assuming unit time to service disk requests, the 1-D block pattern evolves over time as shown above. The identifier of the requesting processor labels each block request. Note that disk utilization is only 50%. Ideally this set of accesses could be completed in four timesteps.

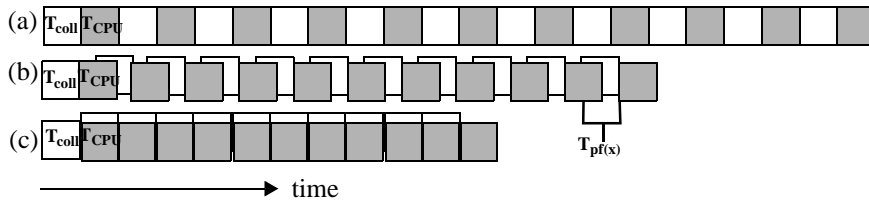


Figure 4: **Collective input/output benchmark.** We consider a benchmark which iteratively reads data (the collective operation) and computes on it (a). Read time and compute time are balanced so that all read time can be partially (b) or fully (c) overlapped with computation.

4.1 Overview

Prefetching is a general technique that improves I/O performance by reading data into cache before it is requested, overlapping I/O with computation. Although much research has focussed on anticipating data requests based on access history, informed prefetching advocates using application-supplied hints to inform the file system of future requests [14].

Unlike a collective API, using hints to expose future requests to the file system does not explicitly give the system the ability to reorder them. However, deep prefetching promotes deep disk queues, enabling implicit request reordering. The deeper the queues, the more reordering is possible, and the higher the throughput. In a spectrum where demand fetching is at one extreme (no reordering) and collective I/O is at the other (perfect reordering), prefetching with gradually increasing depths describes the performance points in between. In other words, an informed prefetching system permits more flexible use of memory; if there is not enough memory to fully optimize the transaction, the system can still obtain partial benefits, degrading gracefully.

Thus, prefetching improvements have two components; I/O is overlapped with computation, and deep prefetch queues allow the disk to reorder requests dynamically, improving throughput. Collective I/O is an optimization that relies solely upon request reordering. One can overlap I/O with computation by issuing collective I/O calls asynchronously; however, this approach requires allocating enough memory to hold both the collective in progress and the collective in memory.

To formalize this intuition, we consider a parallel benchmark that iteratively performs collective I/O operations and computes on the data. A parallel application must read and process some number of files, each with B blocks. The files are striped round-robin, across D disks; this aspect of the layout is not under application control. There are P processors, each with finite buffer space, and the disks and processors are connected with a zero latency network. For each file, the processors synchronize, each processor reads a disjoint portion of the current file (P/B blocks), and when all processors have completed the read, they begin to compute on the data. Figure 4a shows this I/O-compute cycle. Each processor has

Parameter	Value
sector size	512 bytes
cylinders	1962
revolution speed	4002 RPM
transfer time (8 KB)	3.3316 ms
rotational latency	14.992 ms/revolution
seek time (C cylinders)	if $C < 383$, $3.24 + .4\sqrt{C}$ else $8.00 + 0.008C$

Table 1: HP 97560 disk parameters.

x blocks available for prefetching, called the prefetch depth (or prefetch horizon), and may issue up to x prefetch requests in advance of the demand fetch stream, to overlap I/O and computation as shown in Figure 4b. This functionality might be provided by a very basic prefetching system that could be implemented at the application level using asynchronous I/O.

Drawing from terminology set forth in [14], we describe the elapsed time T for such a benchmark as

$$T = N_{coll}(T_{CPU} + T_{coll}) \quad (1)$$

where N_{coll} is the number of collective I/O accesses, T_{CPU} is the CPU time between collective operations and T_{coll} is the time to perform a collective I/O access.

By default, T_{coll} is not overlapped with T_{CPU} ; the application sees all I/O time (Figure 4a). We define $T_{pf(x)}$ as the time to perform a hinted collective I/O with some per-processor prefetch depth x . In other words, we can overlap some fraction of the I/Os in the collective with compute time (T_{CPU}) to improve performance, as shown in Figure 4b. Prefetching will outperform non-overlapped collective I/O if

$$T_{pf(x)} \leq T_{CPU} + T_{coll} \quad (2)$$

At this point, we make an important observation. $T_{pf(x)}$ is highly dependent both on the global access pattern (the temporal and spatial interleaving of each processor’s logical access pattern) and on the disk allocation strategy (the mapping of logical blocks to physical disk locations). Therefore, the choice of prefetch depth will change the time necessary to service the collective using prefetching. We investigate the impact of the global access pattern on prefetch horizon in §5.3.

We use memory as the common currency to compare performance of collective I/O and informed prefetching. A collective operation requires B blocks without overlapping, and $2B$ blocks with potentially full overlap of I/O and computation. In contrast, informed prefetching can utilize any available memory between B to $2B$ blocks. To outperform non-overlapped collective I/O using B blocks, prefetching will require $B + x$ blocks, where x is the prefetch depth. We determine bounds on x in §4.3.

4.2 Performance Model

We consider the throughput of an application with the read/compute cycle of collective operations shown in Figure 4 as a function of the prefetch depth x . Assume that disk block placement is random, so the global access pattern, determined by the distribution of blocks to processors, does not significantly affect overall throughput.

The time for each read/compute cycle is the time to service $(B/P) - x$ I/O requests, plus the compute time for the collective, or $(B/P)T_C$. We assume that the initial x prefetch requests are fully overlapped with the previous cycle compute time. Equation (3) expresses the time for a single read/compute cycle.

$$T_{cycle} = \left(\frac{B}{P} - x\right) \cdot T_d + \left(\frac{B}{P}\right) \cdot T_C \quad (3)$$

The time T_d to service a single block varies with the prefetch depth x . We can model the disk service time more accurately with (4), as the sum of the independent variables for the expected transfer time, rotational latency, and seek time as a function of the per-processor prefetch depth x .

$$T_d = E[TransferTime] + E[RotationalLatency] + E[SeekTime(x)] \quad (4)$$

We can calculate T_d for the HP97560 disk drive (see Table 3), used in our experimental evaluation, as follows. The expected transfer time is a constant for 8 KB blocks, expected rotational latency is equal to one half a rotation, and the

Symbol	Meaning
T_{cycle}	read/compute cycle time
B	number 8 KB blocks
P	number processors (equal to number of disks)
T_d	time to service block read
T_C	compute time per block
x	per-processor prefetch horizon
M	number tracks in desired horizon
r	prefetch requests per disk

Table 2: Symbol definitions.

expected seek time is formulated in (5). If a processor has a prefetch depth of x , it may have at most x outstanding prefetch requests. Although these prefetch requests are issued in logical order within the file, the physical disk blocks will be randomly distributed, and reordered at the disk queues. A processor with x outstanding requests cannot prefetch another block until its first prefetch request has been demand fetched. On average, with random placement, the first request will be in the middle of some disk queue, so half of the requests must be serviced before the processor can issue more prefetches. Consequently, on average, only $x/2$ requests are in the queue at any time. We count the current disk position as an additional request. The average seek distance, in cylinders C , between r uniformly distributed requests is given by $C/(r + 1)$; thus, our model approximates the average seek distance as $C/((x/2) + 2)$.

$$E[SeekTime(x)] = 3.24 + .04\left(\sqrt{\frac{1962}{\frac{x}{2} + 2}}\right) \quad (5)$$

For sequential disk placement, it is more difficult to estimate T_d , because it is dependent upon both the prefetch depth x and the access pattern. We simplify the access pattern variations by modeling a global sequential access pattern created by allocating each block in the file to a processor drawn from a uniform distribution. Our intuition is as follows: the optimal schedule for each drive is to read file blocks sequentially, which means that ideally each drive should know which processors require the blocks they are reading. Within a particular file region, each disk will have partial information about which processors require which blocks, through prefetch requests. As the prefetch depth x increases, the probability of having complete information for this region increases. Within each track, requests are sequential, but the cost to access a different track incurs rotational latency plus transfer time.

The size of the file region we consider is $2P$ blocks, expressed as tracks in (6). This is two times the prefetch depth necessary for theoretical sequentiality on the drives, as we will show in §4.3, and is a good estimate of the practical bound for sequential access (as we show experimentally in §5.4).

$$M = \frac{2 \cdot P \cdot sectorsPerBlock}{sectorsPerTrack} \quad (6)$$

We estimate the average number of tracks hit (\overline{tracks}) by r requests using Cardenas's formula [2] in (7). The number of requests at each disk, r , is related to the prefetch depth x by the relationship $r = (Px)/D$. Since the number of drives and processors are the same, r is equal to x .

$$\overline{tracks} = \left(1 - \left(1 - \frac{1}{M}\right)^r\right) \cdot M \quad (7)$$

Thus, we have $frac_{rand} = (M - \overline{tracks})/M$, and the formula for T_d is given by (8), where T_{seq} is the time to service a sequential disk request. From (8) and (3) we can calculate the throughput for sequential disk layout for a global sequential access pattern.

$$T_d = frac_{rand} \cdot (E[TransferTime] + E[RotationalLatency]) + (1 - frac_{rand}) \cdot T_{seq} \quad (8)$$

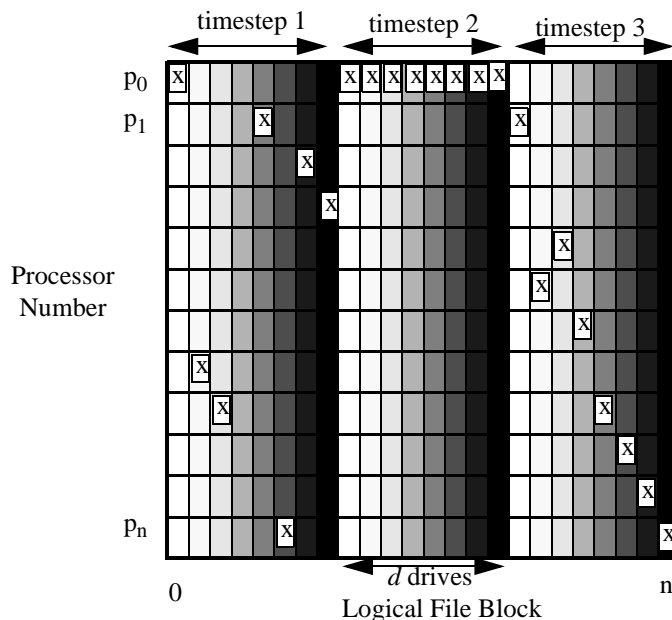


Figure 5: **Bounding the prefetch depth for collective access patterns.** If the network is infinitely fast and disk access occur in unit time, a prefetch depth per processor equal to the number of disks is sufficient to ensure sequential accesses at each drive.

4.3 Determining the Prefetch Depth

Collective I/O implementations optimize collective access patterns by requiring each processor to disclose all intended accesses in advance, and globally sorting this request list for optimal throughput. It might seem that informed prefetching is incapable of generating this optimal I/O request schedule for any file size and access pattern, because the amount of reordering possible is bounded by the size of each processor's prefetch horizon.

However, we can show that independent of file size and access pattern, with certain disk placement assumptions, the prefetch depth necessary to generate an optimal schedule for collective accesses in an ideal system is theoretically bounded and is equal to D , the number of drives. Figure 5 encapsulates our assumptions and sketches a proof of this bound. Blocks of a file are striped, round-robin, across D drives (8 in this figure). Within a drive, logically increasing file blocks are placed sequentially. Each column represents a logical file block, and each row represents a single processor's logical block allocations, or ownerships. Every block is owned by exactly one processor; ownership is represented by an x in the row/column intersection. Each processor requests its blocks in increasing order. We assume I/O requests are perfectly synchronized at the disks and occur in unit time. We also assume the rest of our system is infinitely fast, so that the instant a block has been prefetched, the processor that owns that block can issue a new prefetch request.

The optimal schedule is predetermined by the file layout; at timestep 1, blocks 0 through $D - 1$ must be serviced, then in the next timestep, blocks D through $2D - 1$, and so on. At each timestep, the disks need to know which processor owns each block (i.e., the processor must have prefetched that block). As shown in Figure 5, at any timestep t a prefetch depth of D at each processor will convey this information. This is because all the prefetches for earlier blocks have been serviced, so to service the next D blocks it is sufficient for each processor to reveal its next D requests. For example, in timestep 2, processor p_0 owns all the D blocks that will be accessed; it must have prefetched these blocks.

Intuitively, as requests for blocks in the beginning of the file are serviced, the processors that were waiting on those blocks issue more prefetch requests for the next requests in sequence. In this way, the scheduling algorithm at each drive can dynamically translate an arbitrary decomposition into sequential accesses. Note that the assumption that all blocks of a file will be accessed is crucial to this result, but not necessarily true of collective access patterns in general. For the rest of this paper, we consider only files that are accessed in their entirety.

Parameter	Value
Number of disks	8
Disk type	HP 97560
Disk capacity	1.3 GB
Disk transfer rate	2.11 MB/sec
File system block size	8 KB
Number of processors	8
Network latency	25 ns
Network bandwidth	400 MB/sec

Table 3: Simulator parameters.

5 Experimental Evaluation

In the previous sections, we introduced a model for prefetching throughput and proposed a theoretical bound on the prefetch depth necessary to obtain sequential disk request scheduling. We expect that with equal amounts of memory, informed prefetching should behave slightly worse than a collective interface. However, performance of informed prefetching will increase gracefully as additional prefetch buffers are available, quickly surpassing an efficient collective I/O implementation. In this section we compare collective I/O and informed prefetching experimentally via simulation and measure the accuracy of our model.

5.1 Simulator Architecture

We simulated a simple parallel file system for the system architecture shown in Figure 1 that supports disk-directed I/O and informed prefetching. The major components of our system are processors, the network, and I/O nodes, and our emphasis is on I/O and queuing effects. Files are striped, block by block, across the I/O nodes, and each I/O node services one disk. The network model has point to point communication with a fixed startup overhead and a per-byte transfer cost; we assume a very fast interconnect. Each processor has its own cache and prefetcher; we assume no overhead for memory copies incurred by cache hits. The I/O nodes have no cache beyond the disk cache. The disk model we have used is Kotz’s reimplementation of Ruemmler and Wilkes’ HP 97560 model [12, 15]. Table 3 lists the significant simulator parameters. Although this disk is not modern, advances in disk technology do not qualitatively affect the queuing effects we are addressing, and this particular model has been well tested and validated.

There are several possible implementation strategies for collective I/O; we limit our comparison to disk-directed I/O, which generally outperforms application-level collective implementations such as two-phase collective I/O [7]. In our simulation of disk-directed I/O, the processors synchronize at a barrier, issue a collective call, and submit the collective requests to the I/O nodes. Each I/O node then knows what blocks it must read, and which processors have requested those blocks. The I/O nodes sort the block lists to minimize transfer time, and send each block to the requesting processor as it is read from disk. Throughput for the collective using disk-directed I/O is optimal.

Using informed prefetching, each processor constructs a hint containing the list of blocks that it intends to read using a standard UNIX-style interface. A hint contains a flag indicating whether it is a read or a write and an ordered sequence of blocks to be accessed. This matches the specification SIO low-level API [5]. A prefetching thread associated with each processor issues up to x outstanding prefetches, to be cached at the processor, where x is the prefetch horizon. This approach allows us to hide network latency as well as disk latency. For fair comparison with disk-directed I/O, processors issue a barrier before reading the file.

One of the advantages of deep disk queues created by prefetch requests is the ability to reorder requests, thereby improving throughput. To this end, we use a shortest-seek time first (SSTF) algorithm to schedule physical I/O requests. Although in a real system, SSTF scheduling could lead to starvation, this is not an issue in our experimental design.

5.2 Collective Input/Output Benchmark

Studies of application-level I/O access patterns reveal that iterative I/O compute loops where processors collaborate to read or write and process multiple data sets over the course of execution are quite common [6, 17]. In our experiments, we consider a hypothetical application with a read/compute loop that can be restructured to read (and process) files of varying sizes, maintaining a fixed ratio of compute time to I/O volume, as illustrated in Figure 4. The compute time for

Access Pattern	File Size (MB)	Prefetch Throughput		Collective Throughput	Crossover Prefetch Depth
		Min	Max		
Interleaved Sequential	4	9.018	17.988	9.066	1
	8	9.286	18.132	9.314	1
	32	9.505	18.132	9.509	1
	64	9.534	18.132	9.541	2
	128	9.555	18.132	9.558	2
	256	9.564	18.132	9.566	2
1D Block Sequential	4	3.091	16.894	9.066	16
	32	2.411	18.132	9.509	16
	256	2.344	18.132	9.566	16
3D Block Sequential	4	2.584	17.198	9.066	8
	32	2.708	18.132	9.509	16
	256	2.473	18.132	9.566	16
Global Sequential	4	3.123	16.145	9.066	7
	32	2.623	18.132	9.509	10
	256	2.408	18.132	9.566	14

Table 4: Sequential disk block placement.

File Size (MB)	Prefetch Throughput		Collective Throughput	Crossover Prefetch Depth	
	Min	Max		Unsorted	Sorted
4	1.264	2.926	1.551	16	2
8	1.314	3.033	1.593	32	2
32	1.336	3.036	1.642	56	2
64	1.303	3.036	1.643	88	2
128	1.323	3.036	1.660	136	6
256	1.332	3.069	1.697	296	1

Table 5: Random disk block placement for interleaved sequential access pattern.

<p>for N iterations:</p> <ol style="list-style-type: none"> 1. one processor defines the collective request, describing data to be read by all processors 2. barrier synchronization 3. each processor joins collective, reading its portion of data 4. compute on data 	<p>for N iterations:</p> <ol style="list-style-type: none"> 1. barrier synchronization 2. each processor reads its portion of the data, one block at a time 3. each processor gives the prefetcher hints for the next iteration read, begin prefetching with available memory 4. compute on data
--	---

(a)

(b)

Figure 6: **Collective input/output pseudocode.** Processors collectively read and process N different data files one at a time, using a collective input/output interface (a) and informed prefetching (b).

each collective read is equal to the time required to read the file at the throughput obtained with a 4 MB disk-directed I/O operation, ensuring that I/O can be completely overlapped with computation. There is enough lead compute time that the initial collective can also be fully overlapped. Although iterative collective I/Os are typical, these particular benchmark parameters are fairly arbitrary, and are chosen solely to simplify presentation of results.

We describe results from experiments where files are striped over 8 disks and accessed by 8 processors. We obtained qualitatively identical results with larger balanced systems (larger numbers of disks and processors, and larger file sizes); the smaller configuration size was chosen to limit simulation time.

Figure 6 shows the pseudocode for this benchmark using a collective I/O interface and informed prefetching. With a collective interface, processors request all their assigned blocks at once in step 3. Using prefetching, each processor requests its portion, a block at a time, with no interrequest latency. We consider only 8 KB transfers, a common file system block size. Larger requests increase the degree of sequentiality, and are subsumed by access patterns with larger sequential runs. In this paper we do not consider collective writes or the additional caching and prefetching issues involved in clustering extremely small, noncontiguous requests.

5.3 The Significance of Global Access Patterns

We have explained that throughput using informed prefetching is dependent upon the global access pattern. A collective API allows disks to sort the total amount of work in advance, fully exploiting physical sequentiality. In contrast, the amount of reordering possible through informed prefetching depends upon the prefetch depth.

To illustrate this, we compare disk service times for a file accessed with two different global access patterns: one which results in balanced sequential disk accesses, and one where the prefetch depth determines the length of globally sequential runs. Clients issue prefetches; we vary the number of simultaneously outstanding prefetches per client, but the client cache size is large enough that demand fetches do not flush prefetched buffers. In this experiment, a 4 MB file is striped over 8 disks, and blocks are contiguous on each disk.

Our first access pattern is a simple interleaved sequential pattern. There are 8 processors; each processor, p , reads every 8th block beginning with block p . This processor decomposition corresponds perfectly to the disk decomposition; unfortunately, such a perfect data decomposition is relatively unusual. In contrast, we also consider a 3-D block decomposition. Figure 7 compares the average disk response time for these access patterns at different prefetch horizons. Disk response times for the interleaved sequential access pattern are uniformly good because the stream of accesses to each disk is already sequential. Therefore, deeper prefetch queues cannot further improve performance by enabling request reordering; the only benefit of prefetching is overlapping of I/O and computation. In contrast, for the 3-D block decomposition, larger per-processor prefetch-horizons increase queue depths and allow reordering for improved performance.

5.4 A Comparison of Informed Prefetching and Collective Input/Output

As described in §4, we compare performance of informed prefetching and disk-directed I/O (an efficient implementation of collective I/O) with respect to their memory requirements. Both approaches improve throughput through request reordering. Collective I/O does this with a minimal amount of memory using a specialized API and disk request sorting (collective throughput is independent of access pattern). Prefetching requires additional buffers to store outstanding prefetched blocks

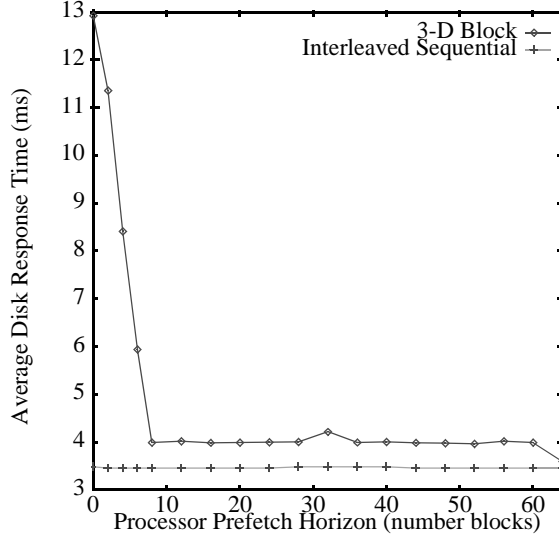


Figure 7: **Selection of prefetch depth.** Average disk response time at different prefetch horizons for different access patterns varies substantially due to queue reordering.

Access Pattern	File Size (MB)	Crossover Prefetch Depth
Interleaved Sequential	256	1
1-D Block	256	96
3-D Block	256	32

Table 6: Random disk block placement with sorted block allocation.

until they are fetched; the exact number of buffers varies with the access pattern and the physical block layout. At the same time, these additional prefetch buffers allow I/O to be overlapped with computation, further improving performance. Collective I/O can only utilize memory in multiples of the collective file size.

Thus, the comparison is one of determining how much additional memory (in prefetch buffers) is required to equal or surpass the performance of a collective interface through these two factors: improved I/O throughput because of request reordering, and overlapping of I/O and computation. In this section we show that prefetching with only a modest per-processor prefetch depth enables disk scheduling optimizations similar to a specialized collective I/O interface.

5.4.1 A Memory-Based Comparison

We compare the throughput of our benchmark using different I/O techniques with varying amounts of memory. For each collective read size B , there are two data points for collective I/O throughput; either the application allocates just enough memory to read the file (B), leaving no room for overlapping I/O and computation, or it allocates twice as much memory ($2B$), permitting potentially full overlap. In contrast, using prefetching, we obtain a full throughput curve as we vary the amount of memory allocated to prefetch buffers from B to $2B$.

Figure 8a shows application throughput for sequentially allocated files accessed using an interleaved sequential access pattern and the 3-D block decomposition illustrated in Figure 2. The total memory available for prefetching varies from one to two times the file size (4 MB to 8 MB) divided among the 8 processors. With prefetching, we can gradually increase the per-processor prefetch horizon, utilizing available extra memory to improve performance. With a collective interface, either none of the I/O can be overlapped with compute time, or the collective operation can be performed asynchronously with twice as much total memory. Thus, there are only two points (circled) for collective throughput. The choice of access pattern does not affect collective throughput, because blocks are optimally sorted.

The interleaved sequential access pattern matches the file layout perfectly; additional memory improves throughput only because prefetching overlaps I/O with computation. In the 3-D block access pattern, additional prefetch buffers not

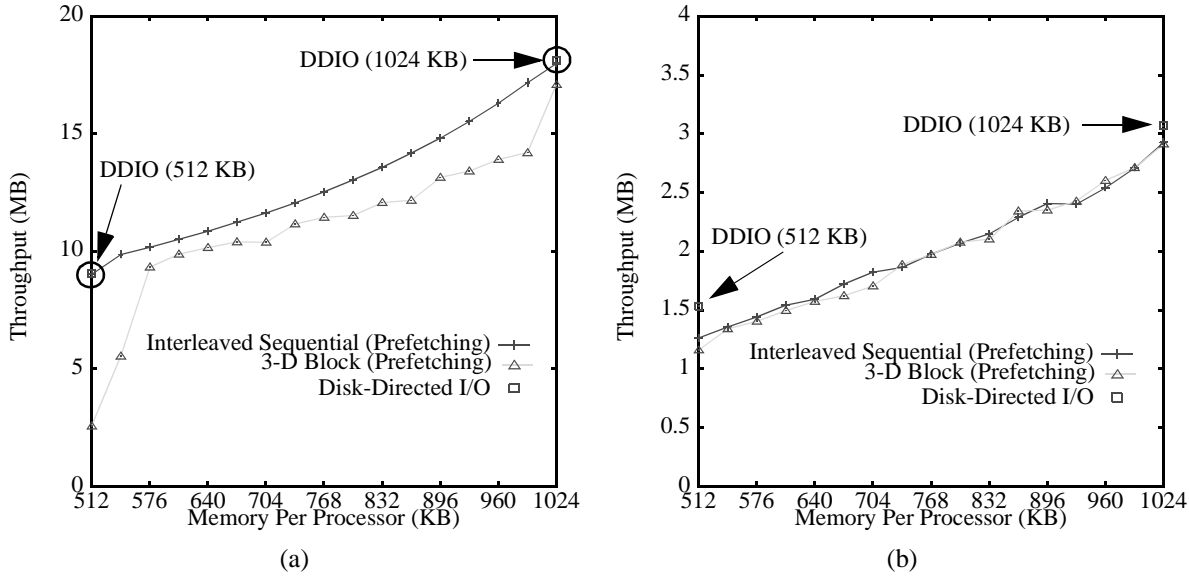


Figure 8: **Throughput vs. memory using collective input/output and prefetching.** A 4 MB file is striped across 8 disks and accessed by 8 processors using different access patterns. In (a), block allocation is sequential and in (b) it is random. With sequential allocation, only a modest amount of additional memory per processor suffices to surpass the throughput obtained using a collective input/output interface. With random allocation, all access patterns exhibit similar behavior because the correlation between logical sequentiality and physical sequentiality is lost. Prefetch depths must be larger to surpass collective input/output throughput.

only overlap I/O and computation, but improve throughput by reordering disk requests and balancing disk load, resulting in sharp performance improvements that level off. Performance of the 3-D block access pattern is not as good as the interleaved sequential pattern, even with deep prefetch queues. This is because shortest seek time first (SSTF) scheduling does not reorder requests to be perfectly sequential, and a single stall can have a cascading effect on throughput when it causes other prefetches not to be issued.

Figure 8b shows the same throughput curves for a file with random block allocation. In this extreme, the application access pattern has little bearing upon the shape of the prefetch horizon curve; deeper prefetch queues improve throughput according to the same function regardless of access pattern as long as the I/O load is balanced among disks. We also observe that because request reordering is so important to optimize random access patterns, collective I/O always provides higher throughput than prefetching with the same amount of memory. Recall that using disk-directed I/O we sort the disk blocks, and with prefetching we use an SSTF scheduling algorithm. Even when each processor prefetches its entire portion, SSTF does not perform as well as sorting the list and making a single disk sweep.

Table 4 summarizes throughput for different file sizes, allocated sequentially, using from one to two times the file size total memory for prefetching. The crossover prefetch depth is the number of outstanding prefetch requests per processor (in 8 KB blocks) necessary to surpass throughput obtained using a synchronous collective I/O interface and disk-directed I/O. In addition to the access patterns shown in Figure 2, we also evaluated a “global sequential” access pattern, created by assigning each logical file block a processor drawn from a uniform distribution until each processor has been assigned its share of the file. As processors read their allocated file blocks, the global access pattern is sequential, but it does not correspond to any standard decomposition.

Table 5 summarizes throughput for an interleaved sequential collective access for a randomly allocated file; other access patterns have similar characteristics. The crossover prefetch depth increases linearly with the file size. It is clear that to exploit the high bandwidth afforded by multiple disks, file blocks that will be accessed together should be stored sequentially.

5.5 Selecting the Prefetch Depth

We proved in §4.3 that for a theoretical system, if logical sequentiality is preserved in the physical sequentiality of the block layout, and file blocks are striped round robin across D disks, a per-processor prefetch depth of D suffices to permit optimal disk scheduling. However, real systems are not infinitely fast, nor do I/O requests occur in constant time, so we should

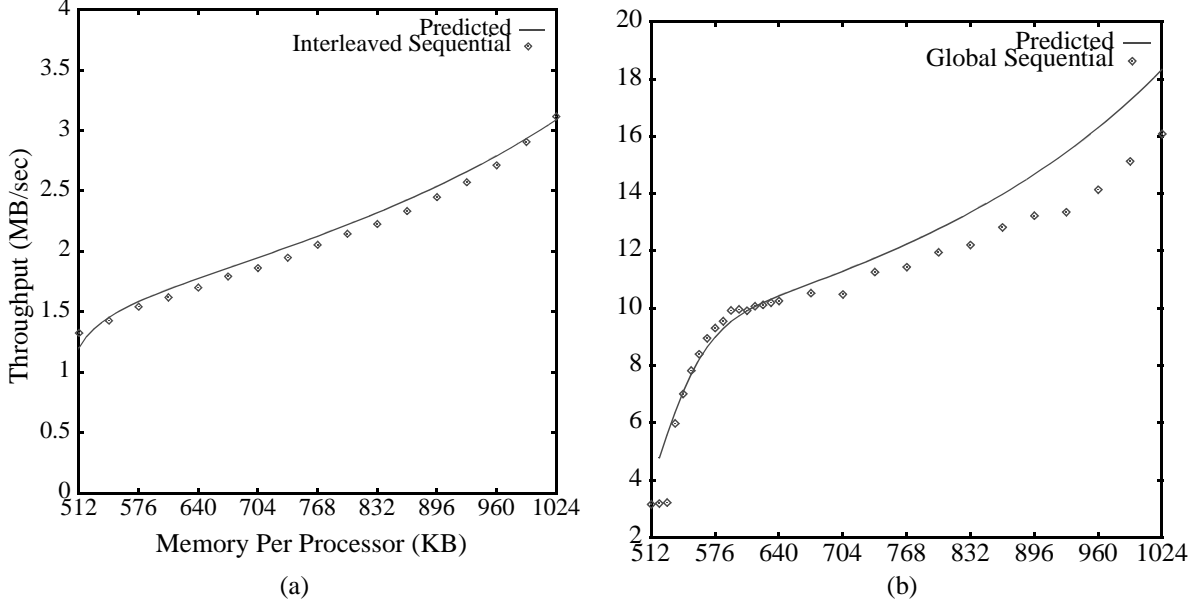


Figure 9: **Predicted throughput vs. experimental throughput.** Our model for random disk placement (a) is accurate to within 5%. The model for sequential disk placement (b) is accurate to within 20% for prefetch depths greater than two.

expect the actual crossover prefetch depth, x , to increase depending on network speed, disk placement, and interrequest latency.

In our experiments, interrequest latency of blocks within the collective is zero, and therefore not an issue. Our network is fast enough that the turnaround time for a prefetched block to be returned to the requesting processor and a new prefetch issued is much smaller than the nearly constant cost of a sequential I/O; however, it is not instantaneous. Thus, we observe in Table 4 that the maximum crossover prefetch depth for sequential block placement is $2D$; the extra factor of D allows processors to indicate which blocks they own far enough in advance that at each timestep the drives will have complete information for the next timestep without instantaneous turnaround.

In a real file system, block placement is usually somewhere between sequential and random. We noted from Table 5 that the crossover prefetch depth increases linearly with the file size; our prefetch depth bound does not apply because the physical disk layout does not reflect the logical block order. However, even if physical blocks are not sequential, as long as they are consecutive (there may be gaps between them) and they are stored on disk in increasing logical order, a limited prefetch buffer depth will suffice to obtain maximum throughput. We demonstrate this by sorting the randomly allocated blocks used to determine the default crossover prefetch depths of Table 4 (so that logically increasing blocks are physically increasing on disk) and recomputing throughput. The new depths, shown in the final column of Table 5, and for large files in Table 6, are significantly smaller.

Note that the optimal depths are not bounded by the $2D$ threshold as they are for sequential access; this is because disk service times for randomly allocated blocks are irregular, and processors must issue more outstanding prefetches to compensate for the lack of synchronicity.

Finally, although we present results only from a balanced system ($P = D$), simulation shows that the dependence of prefetch threshold on D also holds when the number of processors and disks are different.

5.6 Model Accuracy

We have demonstrated that one of the benefits of informed prefetching is its ability to provide throughput improvements that increase gradually with additional memory. In §4.2 we described models for the throughput of informed prefetching as a function of the per-processor prefetch depth. In this section we assess the accuracy of these models compared to our simulation.

Throughput predictions provided by the model for random block placement described in §4.2 are accurate to within 5%, as shown in Figure 9. Throughput predictions provided by the model for sequential block placement are accurate to within 20% for prefetch depths greater than two. Predictions for very small depths are inaccurate because requests are so scattered that the expected rotational latency will be one revolution, as opposed to the half revolution used to calculate

in the model. The model also loses accuracy at larger depths for the same reason that the curve for the 3-D block access pattern in Figure 8a is lower than for the interleaved sequential access pattern; the model assumes perfect sequentiality with high prefetch depths, but SSTF scheduling does not reorder requests to be perfectly sequential. The model is more accurate for larger files, when the effect of a single poor scheduling decision is smaller.

6 Related Work

Several implementations of collective I/O have been proposed. Two of the most popular are two-phase I/O [7] and disk-directed I/O [11]. In two-phase I/O, the application uses a particular interface to alert the file system to a collective operation. In a two-phase read, for example, data is read from disks sequentially and redistributed to the processors involved in the collective. A variation of two-phase I/O called extended two-phase I/O [20] uses collective I/O in conjunction with dynamic partitioning of the I/O workload among processors to balance load and improve performance. As we have described, disk-directed I/O is another method for optimizing collective requests, where the complete request is passed to the I/O processors to be sorted and transferred to application memory.

Many I/O libraries have been developed that rely upon collective I/O optimizations to improve performance of multiprocessor I/O systems. Jovian [1] is a runtime library for Single Process Multiple Data (SPMD) codes (processors execute the same code on different data). The programmer chooses the appropriate data distribution and I/O operations are implicitly collective. A variable number of coalescing processes aggregate the requests to perform them efficiently under the chosen distribution. PASSION [4, 21, 19] (Parallel And Scalable Software for Input-Output) is another runtime library targeted for SPMD applications with routines to efficiently perform out-of-core operations. The Panda [16, 3] library utilizes server-directed I/O, a variation of disk-directed I/O, and a high-level collective interface to achieve high performance on array accesses.

Patterson *et al* [14] demonstrated the potential of hinting accesses to guide prefetching and caching of files that will be accessed in the future. A cost-benefit analysis evaluates the reduction in I/O service time per buffer access to decide when to allocate buffers. This is an elegant approach for dealing with multiple applications that compete for shared caching and prefetching buffers. Kimbrel and Tomkins *et al* [10] compare several integrated caching and prefetching policies using hints for systems with multiple disks. However, these approaches do not take into account the effects of disk queue reordering that are essential to prefetching collective I/O operations.

7 Conclusions and Future Work

We have demonstrated that informed prefetching is a viable alternative to a specialized collective I/O interface for a large class of access patterns. Unlike an efficient collective I/O implementation, informed prefetching requires limited system support beyond the ability to use memory to store asynchronous reads until they are demand fetched. It may be a more natural programming interface for a parallel application programmer who is accustomed to sequential I/O. While performance of informed prefetching improves as additional buffer space becomes available, a collective I/O interface cannot exploit additional memory unless there is enough to hold the subsequent collective read.

The main perceived advantage to a collective I/O interface over prefetching is the ability to sort the collective request pool for optimal throughput. We have proven that a bounded prefetch horizon will suffice to permit global request reordering similar to a collective interface, regardless of access pattern, assuming logical file sequentiality is preserved in physical block layout. We have validated these theoretical results with simulated experiments. Finally, we have developed and validated analytical models that match our experiments to within 20%, while reflecting the important trends and discontinuities.

This research leaves many directions for future work. We intend to investigate how hints can be used to tailor write policies to similarly improve performance of collective writes.

Acknowledgments

I would like to thank Steven Rudich for his help proving a bound on the prefetch depth. Also, thanks to Khalil Amiri, Mike Bigrigg, Joan Digney, Tom Kroeger, David Petrou and David Rochberg for their detailed comments on the text. This research was partially funded by the National Science Foundation CISE Postdoctoral Research Associate Award CDA-9704704, by DARPA/ITO Order D306 issued by Indian Head Division, NSWC, under contract N00174-96-0002, by NSF ERC grant ECD-8907068, by NSF Grants No. IRI-9625428 and DMS-9873442, and by the NSF, ARPA and NASA under NSF Cooperative Agreement No. IRI-9411299. Also, by generous contributions from the member companies of the Parallel Data Consortium, including: Hewlett-Packard Laboratories, Intel, Quantum, Seagate Technology, Storage Technology, Wind River Systems, 3Com Corporation, Compaq, Data General/Clariion, and Symbios Logic.

References

- [1] BENNETT, R., BRYANT, K., SUSSMAN, A., DAS, R., AND SALTZ, J. Jovian: A Framework for Optimizing Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference* (Mississippi State, MS, October 1994), IEEE Computer Society Press, pp. 10–20.
- [2] CARDENAS, A. Analysis and Performance of Inverted Data Base Structures. *CACM 18*, 5 (May 1975), 253–263.
- [3] CHEN, Y., FOSTER, I., NIEPLOCHA, J., AND WINSLETT, M. Optimizing Collective I/O Performance on Parallel Computers: A Multisystem Study. In *Proceedings of the 11th ACM International Conference on Supercomputing* (July 1997), ACM Press, pp. 28–35.
- [4] CHOUDHARY, A., BORDAWEKAR, R., HARRY, M., KRISHNAIYER, R., PONNUSAMY, R., SINGH, T., AND THAKUR, R. PASSION: Parallel and Scalable Software for Input-Output. Tech. Rep. SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.
- [5] CORBETT, P., PROST, J.-P., DEMETRIOU, C., GIBSON, G., RIEDEL, E., ZELENKA, J., CHEN, Y., FELTEN, E., LI, K., HARTMAN, J., PETERSON, L., BERSHAD, B., WOLMAN, A., AND AYDT, R. Proposal for a Common Parallel File System Programming Interface Version 1.0, 1996. Available at <http://www.cs.arizona.edu/sio>.
- [6] CRANDALL, P. E., AYDT, R. A., CHIEN, A. A., AND REED, D. A. Characterization of a Suite of Input/Output Intensive Applications. In *Proceedings of Supercomputing '95* (Dec. 1995).
- [7] DEL ROSARIO, J. M., BORDAWEKAR, R., AND CHOUDHARY, A. Improved Parallel I/O via a Two-Phase Run-Time Access Strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems* (1993), pp. 56–70. Also published in *Computer Architecture News* 21(5), December 1993, pages 31–38.
- [8] GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the Eighth Conference on Architectural Support for Programming Languages and Operating Systems* (1998), ACM Press.
- [9] HUBER, J., ELFORD, C. L., REED, D. A., CHIEN, A. A., AND BLUMENTHAL, D. S. PPFS: A High Performance Portable Parallel File System. In *Proceedings of the 9th ACM International Conference on Supercomputing* (Barcelona, July 1995), ACM Press, pp. 385–394.
- [10] KIMREL, T., TOMKINS, A., PATTERSON, R. H., BERSHAD, B., CAO, P., FELTEN, E. W., GIBSON, G. A., KARLIN, A. R., AND LI, K. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Proceedings of the USENIX Association Second Symposium on Operating Systems Design and Implementation* (1996), pp. 19–34.
- [11] KOTZ, D. Disk-directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems* 15, 1 (February 1997), 41–74.
- [12] KOTZ, D., TOH, S. B., AND RADHAKRISHNAN, S. A Detailed Simulation Model of the HP 97560 Disk Drive. Tech. rep., Department of Computer Science, Dartmouth College, 1994.
- [13] NEEDLEMAN, S. B., AND WUNSCH, C. D. An Efficient Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins. *Journal of Molecular Biology* 48 (1970), 444–453.
- [14] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), ACM Press, pp. 79–95.
- [15] RUEMLER, C., AND WILKES, J. Modelling Disks. Tech. Rep. HPL-93-68, Hewlett Packard Laboratories, July 1993.
- [16] SEAMONS, K. E., CHEN, Y., JONES, P., JOZWIAK, J., AND WINSLETT, M. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing '95* (San Diego, CA, December 1995), IEEE Computer Society Press.

- [17] SMIRNI, E., AYDT, R. A., CHIEN, A. A., AND REED, D. A. I/O Requirements of Scientific Applications: An Evolutionary View. In *Fifth International Symposium on High Performance Distributed Computing* (1996), pp. 49–59.
- [18] STEERE, D. C. Using Dynamic Sets to Speed Search in World Wide Information Systems. Tech. Rep. CMU-CS-95-174, School of Computer Science, Carnegie Mellon University, March 1995.
- [19] THAKUR, R. *Runtime Support for In-Core and Out-of-Core Data-Parallel Programs*. PhD thesis, Department of Electrical and Computer Engineering, Syracuse University, May 1995.
- [20] THAKUR, R., AND CHOUDHARY, A. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. Tech. Rep. CACR-103, Scalable I/O Initiative, Center for Advanced Computing Research, Caltech, June 1995. Revised November 1995.
- [21] THAKUR, R., CHOUDHARY, A., BORDAWEKAR, R., MORE, S., AND KUDITIPUDI, S. Passion: Optimized I/O for Parallel Applications. *IEEE Computer* 29, 6 (June 1996), 70–78.