Disk-Directed I/O for MIMD Multiprocessors

DAVID KOTZ
Dartmouth College

Many scientific applications that run on today's multiprocessors, such as weather forecasting and seismic analysis, are bottlenecked by their file-I/O needs. Even if the multiprocessor is configured with sufficient I/O hardware, the file system software often fails to provide the available bandwidth to the application. Although libraries and enhanced file system interfaces can make a significant improvement, we believe that fundamental changes are needed in the file server software. We propose a new technique, *disk-directed I/O*, to allow the disk servers to determine the flow of data for maximum performance. Our simulations show that tremendous performance gains are possible both for simple reads and writes and for an out-of-core application. Indeed, our disk-directed I/O technique provided consistent high performance that was largely independent of data distribution and obtained up to 93% of peak disk bandwidth. It was as much as 18 times faster than either a typical parallel file system or a two-phase-I/O library.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*access methods*; *file organization*; D.4.8 [**Operating Systems**]: Performance—*simulation*; E.5 [**Data**]: Files

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Collective I/O, disk-directed I/O, file caching, MIMD, parallel file system, parallel I/O

---

1. INTRODUCTION

Scientific applications like weather forecasting, aircraft simulation, molecular dynamics, remote sensing, seismic exploration, and climate modeling are increasingly being implemented on massively parallel supercomputers [Kotz 1996a]. Each of these applications has intense I/O demands, as well as massive computational requirements. Recent multiprocessors have pro-

---

This research was supported by Dartmouth College, by the National Science Foundation under grant number CCR 9404919, and by NASA Ames Research Center under Agreement Number NCC 2-849. Portions of this article appeared in the First Symposium on Operating Systems Design and Implementation (OSDI), November, 1994.

Author's address: 6211 Sudikoff Laboratory, Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510; email: dfk@cs.dartmouth.edu.

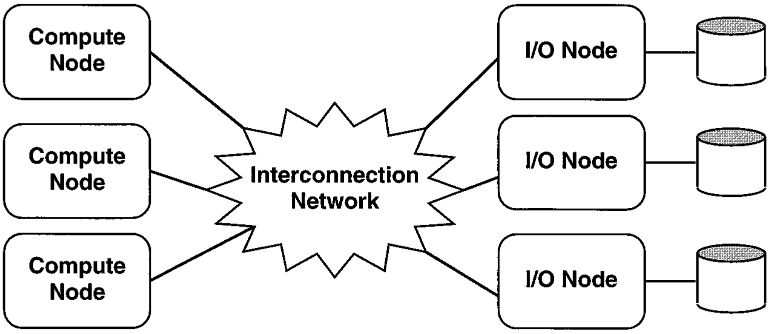

Fig. 1. A typical MIMD multiprocessor, with separate compute processors and I/O processors. Disks attach only to IOPs, which run the file system code. Applications run only on the CPs.

vided high-performance I/O hardware [Kotz 1996b], in the form of disks or disk arrays attached to I/O processors connected to the multiprocessor's interconnection network, but effective file system software has lagged behind.

Today's typical multiprocessor has a rudimentary parallel file system derived from Unix. While Unix-like semantics are convenient for users porting applications to the machine, the performance is often poor. Poor performance is not surprising because the Unix file system [McKusick et al. 1984] was designed for a general-purpose workload [Ousterhout et al. 1985], rather than for a parallel, scientific workload. Scientific applications use larger files and have more sequential access [Galbreath et al. 1993; Miller and Katz 1991; Pasquale and Polyzos 1993; 1994]. *Parallel* scientific programs access the file with patterns not seen in uniprocessor or distributed-system workloads. Although there seems to be a wide variety of access patterns [Crandall et al. 1995; Nieuwejaar et al. 1996; Smirni et al. 1996], we have noticed many patterns accessing small, discontiguous pieces of the file in several types of strided pattern [Nieuwejaar and Kotz 1996; Nieuwejaar et al. 1996]. Finally, scientific applications use files for more than loading raw data and storing results; files are used as scratch space for large problems as application-controlled virtual memory [Brunet et al. 1994; Cormen and Kotz 1993; Klimkowski and van de Geijn 1995; Womble et al. 1993]. In short, multiprocessors need new file systems that are designed for parallel scientific applications.

In this article we describe a technique, *disk-directed I/O*, that is designed specifically for high performance on parallel scientific applications. It is most suited for MIMD multiprocessors that have no remote-memory access and that distinguish between I/O Processors (IOPs), which run the file system, and Compute Processors (CPs), which run the applications. Figure 1 shows such an architecture. The IBM SP-2, Intel iPSC, Intel Paragon, KSR/2, Meiko CS-2, nCUBE/2, Thinking Machines CM-5, and Convex Exemplar all use this model. The architectures of the Paragon, the CS-2, and the SP-2 allow IOPs to double as CPs, although they are rarely so configured. Furthermore, our technique is best suited to applications written in a single-program-multiple-data (SPMD) or data-parallel programming model. With our technique, described below, CPs collectively send a single request to all IOPs, which then arrange the flow of data to optimize disk performance.

We begin by advocating that parallel file systems support noncontiguous and collective data requests. Then, in Sections 3 and 4, we consider some of the ways to support collective I/O and our implementation of these alternatives. Section 5 describes our experiments, and Section 6 examines the results. We look at some possible interfaces in Section 7, and we consider some generalizations of disk-directed I/O in Section 8. We contrast our system to related work in Section 9 and mention some existing implementations in Section 10. We summarize our conclusions in Section 11.

2. COLLECTIVE I/O

Consider (1) programs that distribute large matrices across the processor memories and (2) the common task of loading such a matrix from a file. From the point of view of a traditional file system, that is, with a Unix-like interface, each processor independently requests its portion of the data, by reading from the file into its local memory. If that processor's data are not logically contiguous in the file, as is often the case [Nieuwejaar et al. 1996], a separate file system call is needed for each contiguous chunk of the file. The file system is thus faced with concurrent small requests from many processors, instead of the single large request that would have occurred on a uniprocessor. Indeed, since most multiprocessor file systems decluster file data across many disks, each application request may be broken into even smaller requests, which are sent to different IOPs.

The problem here is that valuable semantic information has been lost. The application programmer knows that the entire matrix is to be transferred between the file and the multiple CP memories, but is forced by the traditional Unix-like interface to break that transfer into a series of small, contiguous requests from each CP. Two important pieces of semantic information have been lost in the translation: that each request is actually part of a larger data transfer and that all the CPs are cooperating in a *collective* request.

It is sometimes possible to rewrite the application to avoid making tiny, discontiguous requests, particularly if you understand the application and the I/O system well [Acharya et al. 1996]. Unfortunately, such a rewrite is often difficult, forcing the application programmer to consider issues like buffering, asynchronous I/O, prefetching, and so forth, that are better left to the file system. In this article we demonstrate a file system technique that can provide near-optimal I/O performance to applications, by allowing applications to request transfers that (1) involve noncontiguous subsets of the file and (2) involve all CPs in a collective operation.

Fortunately, there are a few file system interfaces that allow noncontiguous transfers. Vesta [Corbett and Feitelson 1996] and the nCUBE file system [DeBenedictis and del Rosario 1992] support logical mappings between the file and processor memories, defining separate "subfiles" for each processor. The Galley [Nieuwejaar and Kotz 1997] file system's *nested-batched* interface allows the programmer to specify strided, nested-strided, or list-oriented data transfers. The low-level interface proposed by the Scalable-I/O (SIO) Initiative [Corbett et al. 1996b] provides a subset of Galley's capability.

There are also a few systems that support a *collective-I/O interface*, in which all CPs cooperate to make a single, large request. Data-parallel languages, such as CM-Fortran for the CM-5 and C* [Moore et al. 1995], have a collective I/O interface by nature. The emerging MPI-IO standard includes some collective I/O support [Corbett et al. 1996a; MPI-IO Committee 1996], as does the SIO interface [Corbett et al. 1996b]. Finally, there are several libraries for collective matrix I/O [Bennett et al. 1994; Bordawekar et al. 1993; Chen et al. 1996; Foster and Nieplocha 1996; Galbreath et al. 1993; Karpovich et al. 1994; Toledo and Gustavson 1996], and at least one for more complex data structures [Gotwals et al. 1995].

These interfaces lay the groundwork for noncontiguous, collective I/O transfers. Although we return to the interface issue in Section 7, this article focuses on a high-performance implementation technique to make the best of the information provided through the interface.

3. COLLECTIVE-I/O IMPLEMENTATION ALTERNATIVES

In this article we consider collective-read and -write operations that transfer a large matrix between CP memories and a file. The matrix is stored contiguously within the file, but the file is declustered, block by block, over many IOPs and disks. The matrix is distributed among the CPs in various ways, but within each CP the data are contiguous in memory. We examine three implementation alternatives: a simple parallel file system, two-phase I/O, and disk-directed I/O. Figure 2 sketches the implementation of each. We introduce each here and discuss the details in Section 4.

Simple Parallel File System (*SPFS*). This alternative mimics a "traditional" parallel file system like Intel CFS [Pierce 1989], with IOPs that each manage a cache of data from their local disks. The interface has no support for collective I/O, or for noncontiguous requests. Thus, the application must make a separate request to the file system for each contiguous chunk of the file, no matter how small. Figure 2(a) shows the function called by the application on the CP to read its part of a file and the corresponding function executed at the IOP to service each incoming CP request.

Two-Phase I/O (*2PIO*). Figure 2(b) sketches an alternative proposed by del Rosario, Bordawekar, and Choudhary [del Rosario et al. 1993; Thakur et al. 1996], which permutes the data among the CP memories before writing or after reading. Thus, there are two phases: one for I/O and one for an in-memory permutation. The permutation is chosen so that each CP

(a) Simple Parallel File System

```
ReadCP(file, read parameters, destination address):
  for each file block needed to satisfy request
    compute which disk holds that file block
    if too many requests to that disk are outstanding,
      wait for response and deposit data into user's buffer
    send new request to that disk's IOP for this (partial) block
  end
  wait for all outstanding requests
```

```
ReadIOP(file, read parameters):
  look for the requested block in the cache
  if not there
    find or make a free cache buffer
    ask disk to read that block into cache buffer
  reply to CP, including data from cache buffer
  consider prefetching or other optimizations
```

(b) Two-Phase I/O

```
CollectiveReadCP(file, read parameters, destination address):
  Barrier (CPs using this file), to ensure that all are ready
  decide what portion of the data this processor should read
    (conforming to the file layout)
  ReadCP(file, portion, temporary buffer)
  Barrier (CPs using this file), to wait for all I/O to complete
  run permutation algorithm to send data to correct destination
  Barrier (CPs using this file), to wait for permutation to complete
```

ReadIOP (*as above*)

(c) Disk-Directed I/O

```
CollectiveReadCP(file, read parameters, destination address):
  since data will arrive later in asynchronous messages
    record destination address for use by message handler
  Barrier (CPs using this file), to ensure that all buffers are ready
  any one CP:
    multicast (CollectiveRead, file, read parameters) to all IOPs
    wait for all IOPs to respond that they are done
  Barrier (CPs using this file), to wait for all I/O to complete
```

```
CollectiveReadIOP(file, read parameters):
  determine the set of file data local to this IOP
  determine the set of disk blocks needed
  sort the disk blocks to optimize disk movement
  using double-buffering for each disk,
    request blocks from the disk
    as each block arrives from disk,
      send piece(s) to the appropriate CPs
  when complete, send message to original requesting CP
```

Fig. 2. Pseudocode for collective-read implementations. CP code is on the left; IOP code is on the right. Collective writes are similar.

makes one, large, contiguous file system request. The two-phase-I/O authors call this a "conforming" distribution; the file is logically broken into approximately equal-sized contiguous segments, one for each CP.

Disk-Directed I/O (DDIO). Here, the collective request is passed on to the IOPs, which then arrange the data transfer as shown in Figure 2(c). This *disk-directed* model puts the disks (IOPs) in control of the order and timing of the flow of data. Disk-directed I/O has several potential advantages over two-phase I/O:

- The I/O can conform not only to the logical layout of the file, but to the physical layout on disk (since two-phase I/O is only a library on top of a SPFS, it cannot access physical layout information). Furthermore, if the disks are actually redundant disk arrays (RAIDs), the I/O can be organized to perform full-stripe writes for maximum performance.
- The disk I/O and the permutation overlap in time, rather than being separated into two phases, so the smaller of the two (usually the permutation) takes effectively no time.
- There is no need to choose a conforming distribution. The choice is difficult and is dependent on the file layout, access pattern, access size, and cache management algorithm.
- The IOPs are aware that the CPs are involved in a collective request and can work to minimize the elapsed time of the entire request. Secondary goals often implemented in an SPFS IOP's cache management and disk-scheduling policies, such as fairness to all CPs, may be abandoned. (The optimal (but unfair) schedule for some access patterns is to service one CP's request in its entirety, before the other CPs are serviced at all.)
- IOP prefetching and write-behind require no guessing and thus make no mistakes.
- Buffer management is perfect, needing little space (two buffers per disk per file) and capturing all potential locality advantages.
- No additional memory is needed at the CPs for permuting data.
- Each datum moves through the interconnect only once in disk-directed I/O and typically twice in two-phase I/O.
- Communication is spread throughout disk transfer, not concentrated in a permutation phase.
- There is no communication among the CPs, other than barriers.

Disk-directed I/O has several additional advantages over the simple parallel file system:

- There is only one I/O request to each IOP, reducing overhead.
- Disk scheduling is improved, by sorting the complete block list for each disk (Figure 2(c)), rather than dynamically scheduling only the "current" requests.

—There is no need for the file system code on the CPs to know the pattern of data declustering across disks, allowing more complex declustering schemes to be implemented.

A note about the barriers in Figure 2(c): the cost of the barriers themselves is negligible compared to the time needed for a large I/O transfer. For some applications, the waiting time at the first barrier may be a concern if the preceding computation is poorly balanced across CPs. If so, the programmer may consider using noncollective disk-directed I/O, in which each process makes its own individual disk-directed request to the IOPs. The cost of unsynchronized requests may be much larger than the saved synchronization overhead, however, particularly when the I/O-access pattern exhibits fine-grained interprocess spatial locality.

4. EVALUATION

We implemented a simple parallel file system, a two-phase-I/O system, and a disk-directed-I/O system on a simulated MIMD multiprocessor (see below). In this section, we describe our simulated implementation. Many of the parameters are shown in Table I.[1]

Files were striped across all disks, block by block. Each IOP served one or more disks, using one I/O bus. Each IOP had a thread permanently running for each local disk, that controlled access to the disk device. The disk thread communicated with threads representing CP requests through a disk-request queue.

Message-Passing and DMA. Since we assumed there was no remote-memory access, we had to depend on message passing for data transfer. We did assume, however, that the network interface had a direct-memory access (DMA) capability. Our implementation used DMA to speed message passing in several ways. Each message was encoded so that the DMA interrupt handler on the receiving processor could quickly decide where to deposit the contents of the message. For requests to the IOP, it copied the message into a free buffer and woke a sleeping thread to process the buffer. Part of each request was the address of a *reply action*, a structure on the CP which contained the address where a reply could be written, and the identity of a thread to wake after the reply arrived. The IOP included this reply-action address in its reply to a request, for the CP's interrupt handler to interpret.

In some situations we used "Memget" and "Memput" messages to read and write the user's buffer on the CPs. Every recipient CP provided a base address to its message-passing system, so that the requester only referred to offsets within each CP's buffer. Memput messages contained data and returned only an acknowledgment. Memget messages contained a reply-action address and returned a reply containing the requested data. It was

---

[1] Throughout this article, for both rates and capacities, KB means 2^{10} bytes, and MB means 2^{20} bytes.

Table I. Parameters for Simulator and Our File System Implementations

IOP buffers (DDIO)	2 per local disk
IOP buffers (SPFS, 2PIO)	2 per local disk per CP
SPFS outstanding requests to IOP	4 per CP per disk
SPFS IOP cache replacement	LRU
SPFS IOP cache prefetch	one block ahead on same disk
SPFS IOP cache write-back	WriteFull
MIMD, distributed memory	32 processors
Compute processors (CPs)	16*
I/O processors (IOPs)	16*
CPU speed, type	50MHz, 32-bit RISC
Disks	16*
Disk type	HP 97560
Disk capacity	1.3GB
Disk transfer rate	2.11MB/s, for multitrack transfers
File system block size	8KB
I/O buses (one per IOP)	16*
I/O bus type	SCSI
I/O bus peak bandwidth	10MB/s
Interconnect topology	6 × 6 rectilinear torus
Interconnect bandwidth	200×10^6 bytes/s, bidirectional
Interconnect latency	20ns per router
Routing	wormhole
Memput CPU time (approx)	5μsec + 1μsec/(50 words of data)
Memget CPU time (approx)	5μsec + 2μsec/(50 words of data)

Those marked with a * were varied in some experiments. Memput and Memget times are measurements from our code.

possible to dynamically "batch" small Memput and Memget requests, to combine many individual data transfers into larger group transfers.[2]

Two-Phase I/O. Our implementation followed the pseudocode of Figure 2(b). We chose the same conforming distribution used by the two-phase I/O authors (actually, a row-block distribution, because we store matrices in row-major order) [Thakur et al. 1996]. Thus, the application made only one, large, contiguous file system request to each CP. The data were permuted after reading, using Memputs, or before writing, using Memgets. When the matrix-element size was smaller than the maximum message size, we allowed the Memput and Memget requests to be batched into group requests. This decision nearly always led to better performance, although it was up to 5% slower in some cases [Kotz 1996c].

As in a real two-phase-I/O implementation, the code is layered above a simple parallel file system; we used the simple parallel file system described below. Since the file is striped across all disks, block by block, the

---

[2] We used a fairly naive approach with good results [Kotz 1996c]. There are more sophisticated techniques [Dinda and O'Hallaron 1996].

file system turns each CP's single contiguous file system request into a series of block requests to every disk.

Disk-Directed I/O. Each IOP received one request, which was handled by a dedicated thread. The thread computed the list of disk blocks involved, sorted the list by physical disk address, and informed the relevant disk threads. It then allocated two one-block buffers for each local disk (double buffering) and created a thread to manage each buffer. While not necessary, the threads simplified programming the concurrent activities. These buffer threads repeatedly transferred blocks using Memput and Memget messages to move data to and from the CP memories, letting the disk thread choose which block to transfer next. When possible the buffer thread sent concurrent Memget or Memput messages to many CPs. When the matrix-element size was smaller than the maximum message size, we allowed the Memput and Memget requests to be batched into group requests. This decision always led to better performance [Kotz 1996c].

Simple Parallel File System. Our code followed the pseudocode of Figure 2(a). CPs did not cache data, so all requests involved communication with the IOP. The CP sent concurrent requests to all the relevant IOPs, with up to four outstanding requests *per disk*, *per CP*, when possible. Most real systems are much less aggressive. Note that the CP file system code could only make multiple outstanding requests to the same disk when presented with a large (multistripe) request from the application.

We limited our CPs to four outstanding requests per disk, per CP, as a form of flow control, to prevent CPs from overwhelming IOP resources. Based on our experiments, four outstanding requests led to the fastest file system [Kotz 1996c]. Each IOP managed a cache that was large enough to provide two buffers for every outstanding block request from all CPs to all local disks, one for the request and one for the corresponding prefetch or write-behind request, using a total of eight buffers per CP per disk. The cache used an LRU replacement strategy, although it wrote dirty buffers to disk when they were full (i.e., after n bytes had been written to an n-byte buffer, though not necessarily in order [Kotz and Ellis 1993]). It prefetched one block ahead after each read request; this is more aggressive than many traditional Unix file systems, less aggressive than some parallel file systems (Intel CFS always reads groups of eight blocks, regardless of access pattern [Pierce 1989]), and very effective in the access patterns used in our experiments. New disk requests were placed into a per-disk priority queue, based on their disk-sector address, using the Cyclical Scan algorithm [Seltzer et al. 1990], and withdrawn from the queue by the disk thread when it completed the previous request. This algorithm was nearly always faster than an FCFS algorithm; in one case it was 16% slower [Kotz 1996c].

We transferred data as a part of request and reply messages, rather than with Memget or Memput. We tried using Memgets to fetch the data directly from the CP buffer to the cache buffer, but that was usually slower and never substantially faster [Kotz 1996c]. We nonetheless avoided most memory-memory copies by using DMA to move data directly between the network and the user's buffer or between the network and the IOP's cache buffers, if possible. At the IOP, incoming write requests containing the data to write were assigned to an idle thread, with the message deposited in the thread's stack until the thread determined where in the cache to put the data. Later, the thread copied the data into a cache buffer.

While our cache implementation does not model any specific commercial cache implementation, we believe it is a reasonable competitor for our disk-directed-I/O implementation. If anything, the competition is biased in favor of the simple parallel file system, leading to a conservative estimate of the relative benefit of disk-directed I/O, due to the following simplifications:

—The total cache provided, eight buffers *per CP*, *per disk*, *per file*, grows quadratically with system size and is thus not scalable (disk-directed I/O only needs two buffers per disk per file). This size is quite generous; for example, there is a 64MB cache per IOP per file, for IOPs with 2 local disks in a system with 512 CPs and an 8KB block size.

—The static flow control resulting from our limiting each CP to four outstanding requests per disk (made possible by our large cache) saved the extra latency and network traffic of a dynamic flow control protocol.

—We assumed that write requests from different CPs would not overlap, avoiding the need to ensure that writes were performed in the same relative order at all IOPs. Although valid for all of the access patterns in our experiments, a real system would have extra overhead needed to guarantee proper ordering, or a flag like Vesta's *reckless* mode [Corbett and Feitelson 1996].

—We arranged for the application program to transfer the largest possible contiguous pieces of the file, within the constraints of the specified access pattern, rather than to access individual matrix elements. For most access patterns this arrangement led to much better performance. Although this optimization seems obvious, a surprising number of applications read contiguous data in tiny pieces, one by one, when a single large contiguous request might have served the same purpose [Nieuwejaar et al. 1996].

4.1 Simulator

The implementations described above ran on top of the Proteus parallel-architecture simulator [Brewer et al. 1991], which in turn ran on a DEC-5000 workstation. We configured Proteus using the parameters listed in Table I. These parameters are not meant to reflect any particular machine, but a generic machine of 1994 technology.

Proteus itself has been validated against real message-passing machines [Brewer et al. 1991]. Proteus has two methods for simulating the interconnection network: an exact simulation that models every flit movement and a modeled simulation that uses stochastic techniques to estimate network contention and its effect on latency. Both methods assume that each processor has a deep hardware FIFO buffer to order incoming messages. To reduce the effect of this assumption, we added flow control to limit our use of this buffer.

We compared the effect of the network model on a subset of our experiments, some with thousands of tiny messages and some with many large messages, and found that the results of each experiment using the stochastic model differed from the same experiment using the exact network by at most 5.4%, and typically by less than 0.1%. Thus, our experiments used the stochastically modeled network.

We added a disk model, a reimplementation of Ruemmler and Wilkes' HP 97560 model [Ruemmler and Wilkes 1994]. We validated our model against disk traces provided by HP, using the same technique and measure as Ruemmler and Wilkes. Our implementation had a demerit percentage of 3.9%, which indicates that it modeled the 97560 accurately [Kotz et al. 1994].

5. EXPERIMENTAL DESIGN

We used the simulator to evaluate the performance of disk-directed I/O, with the throughput for transferring large files as our performance metric. The primary factor used in our experiments was the file system, which could be one of four alternatives: the simple parallel file system, two-phase I/O layered above the simple parallel file system, disk directed, or disk directed with block-list presort. We repeated our experiments for a variety of system configurations; each configuration was defined by a combination of the file access pattern, disk layout, number of CPs, number of IOPs, and number of disks. Each test case was replicated in five independent trials, to account for randomness in the disk layouts, disk initial rotational positions, and in the network. The total transfer time included waiting for all I/O to complete, including outstanding write-behind and prefetch requests.

The File and Disk Layout. Our experiments transferred a one- or two-dimensional array of records. Two-dimensional arrays were stored in the file in row-major order. The file was striped across disks, block by block. The file size in all cases was 10MB (1280 8KB blocks). While 10MB is not a large file, preliminary tests showed qualitatively similar results with 100 and 1000MB files (see Section 6.1). Thus, 10MB was a compromise to save simulation time.

Within each disk, the blocks of the file were laid out according to one of two strategies: *contiguous*, where the logical blocks of the file were laid out in consecutive physical blocks on disk, or *random-blocks*, where blocks were placed at random physical locations. We used the same set of five layouts (one for each trial) for all *random-blocks* experiments. A real file system would be somewhere between the two. For validation, however, we experimented with a compromise *random-tracks* layout. In this layout, we chose a random set of physical tracks and placed blocks consecutively within each track. We found our results to be qualitatively similar, and quantitatively

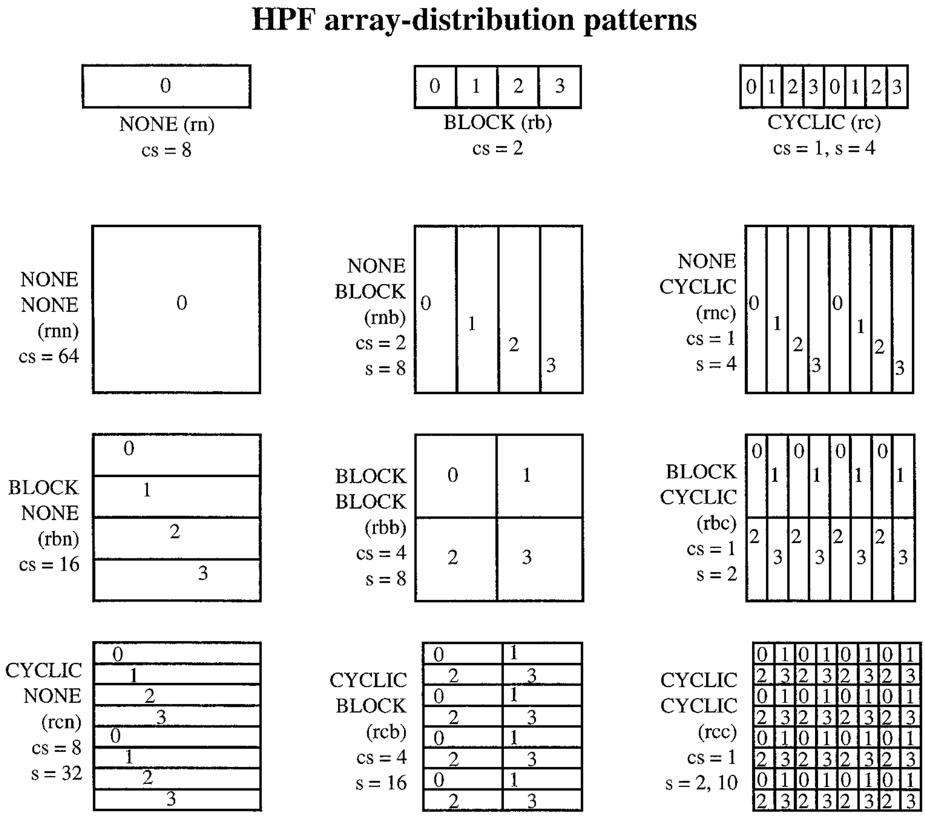

Fig. 3. Examples of matrix distributions, which we used as file access patterns in our experiments. These examples represent common ways to distribute a 1 × 8 vector or an 8 × 8 matrix over four processors. Patterns are named by the distribution method (NONE, BLOCK, or CYCLIC) in each dimension (rows first, in the case of matrices). Each region of the matrix is labeled with the number of the CP responsible for that region. The matrix is stored in row-major order, both in the file and in memory. The *chunk size* (cs) is the size of the largest contiguous chunk of the file that is sent to a single CP (in units of array elements), and the *stride* (s) is the file distance between the beginning of one chunk and the next chunk destined for the same CP, where relevant. The actual shapes used in our experiments are listed in Table II.

between the contiguous and random-blocks layouts, so we only treat the two extremes here.

The Access Patterns. Our read- and write-access patterns differed in the way the array elements (records) were mapped into CP memories. We chose to evaluate the array distribution possibilities available in High-Performance Fortran [High Performance Fortran Forum 1993], as shown in Figure 3. Thus, elements in each dimension of the array could be mapped entirely to one CP (NONE), distributed among CPs in contiguous blocks (BLOCK; note this is a different “block” than the file system “block”), or distributed round-robin among the CPs (CYCLIC). We name the patterns using a shorthand beginning with r for reading an existing file or w for

Table II. Summary of File Access Patterns

Pattern Name	Row Distribution	Column Distribution	Record Size (bytes)	Rows	Cols	Chunk Size (records)	Stride (records)
ra	ALL	—	—	—	—	1280 blocks	—
rn	NONE	—	—	—	—	1280 blocks	—
rb	BLOCK	—	8	1310720	—	80K	—
rc	CYCLIC	—	8	1310720	—	1	16
rnn	NONE	NONE	8	1280	1024	1280K	—
rnb	NONE	BLOCK	8	1280	1024	64	1K
rnc	NONE	CYCLIC	8	1280	1024	1	16
rbn	BLOCK	NONE	8	1280	1024	80K	—
rbb	BLOCK	BLOCK	8	1280	1024	256	1K
rbc	BLOCK	CYCLIC	8	1280	1024	1	4
rcn	CYCLIC	NONE	8	1280	1024	1K	16K
rcb	CYCLIC	BLOCK	8	1280	1024	256	4K
rcc	CYCLIC	CYCLIC	8	1280	1024	1	4, 3K+4
rb	BLOCK	—	8192	1280	—	80	—
rc	CYCLIC	—	8192	1280	—	1	16
rnn	NONE	NONE	8192	40	32	1280	—
rnb	NONE	BLOCK	8192	40	32	2	32
rnc	NONE	CYCLIC	8192	40	32	1	16
rbn	BLOCK	NONE	8192	40	32	80	—
rbb	BLOCK	BLOCK	8192	40	32	8	32
rbc	BLOCK	CYCLIC	8192	40	32	1	4
rcn	CYCLIC	NONE	8192	40	32	32	512
rcb	CYCLIC	BLOCK	8192	40	32	8	128
rcc	CYCLIC	CYCLIC	8192	40	32	1	4, 100

Smaller examples of these patterns are shown in Figure 3. We list only the read patterns here. All numbers are for a 10MB file distributed over 16 CPs. Two-dimensional matrices are stored in the file in row-major order. A dash (—) indicates "not applicable." Chunks and strides are given in *records*, not bytes (for 8-byte records, notice that 1K records are one block). The rcc pattern has two different strides.

writing a new file; the r names are shown in Figure 3. There was one additional pattern, ra (ALL, not shown), which corresponds to all CPs reading the entire file, leading to multiple copies of the file in memory. Note that rb and wb are the "conforming distributions" used by two-phase I/O. Table II shows the exact shapes used in our experiments. A few patterns are redundant in our configuration (rnn ≡ rn, rnc ≡ rc, rbn ≡ rb) and were not actually used.

We chose two different record sizes: one designed to stress the system's capability to process small pieces of data, with lots of interprocess locality and lots of contention, and the other designed to work in the most convenient unit, with little interprocess locality or contention. The small record size was eight bytes, the size of a double-precision floating-point number. The large record size was 8192 bytes, the size of a file system block and cache buffer. These record size choices are reasonable [Nieuwejaar et al. 1996]. We also tried 1024-byte and 4096-byte records (Figure 15),

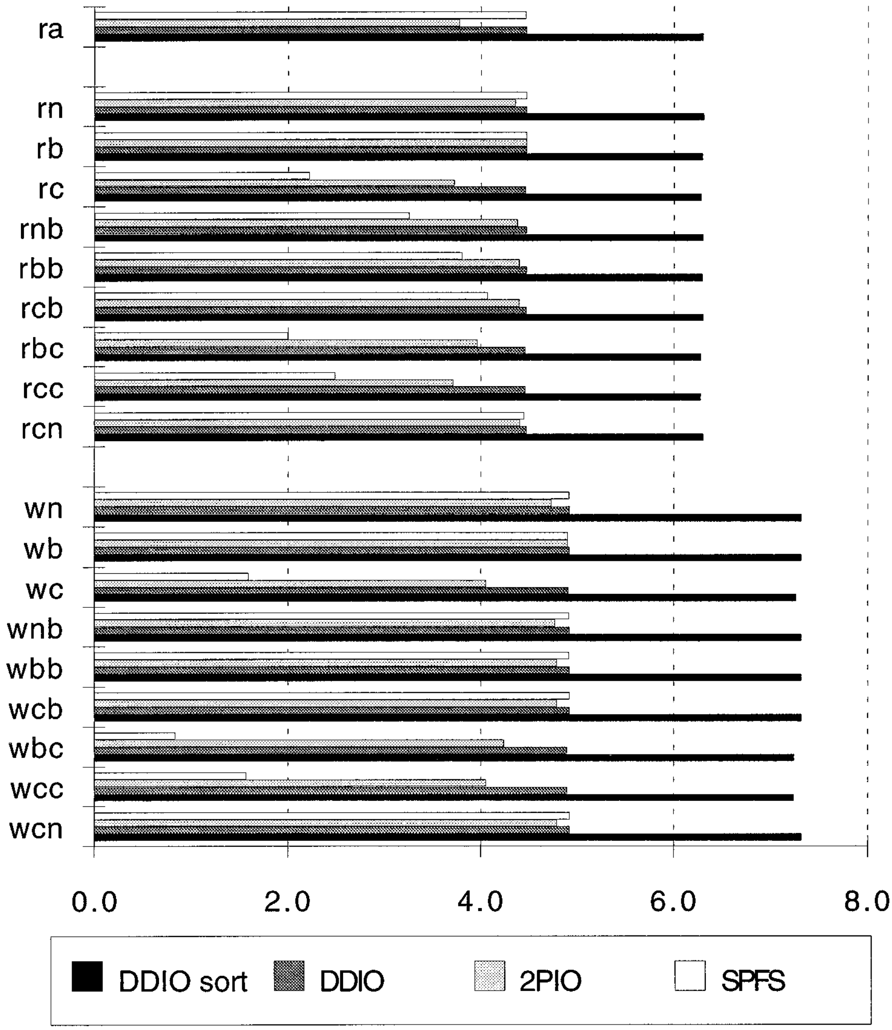

Fig. 4. Comparing the throughput of disk-directed I/O (DDIO) to that of two-phase I/O (2PIO) and the simple parallel file system (SPFS), on a random-blocks disk layout using patterns with an 8-byte record size. ra throughput has been normalized by the number of CPs. Each point represents the average of five trials of an access pattern (the maximum coefficient of variation (*cv*) is 0.042, except for 0.32 on wc on SPFS).

leading to results between the 8-byte and 8192-byte results; we present only the extremes here.

In the simple-system case, recall that the application makes file system requests for whole chunks, which may be much larger than individual records (Table II).

6. RESULTS

Figures 4–7 show the performance of the three techniques. All experiments used 16 CPs, 16 disks, and 16 IOPs. Because the ra pattern broadcasts the

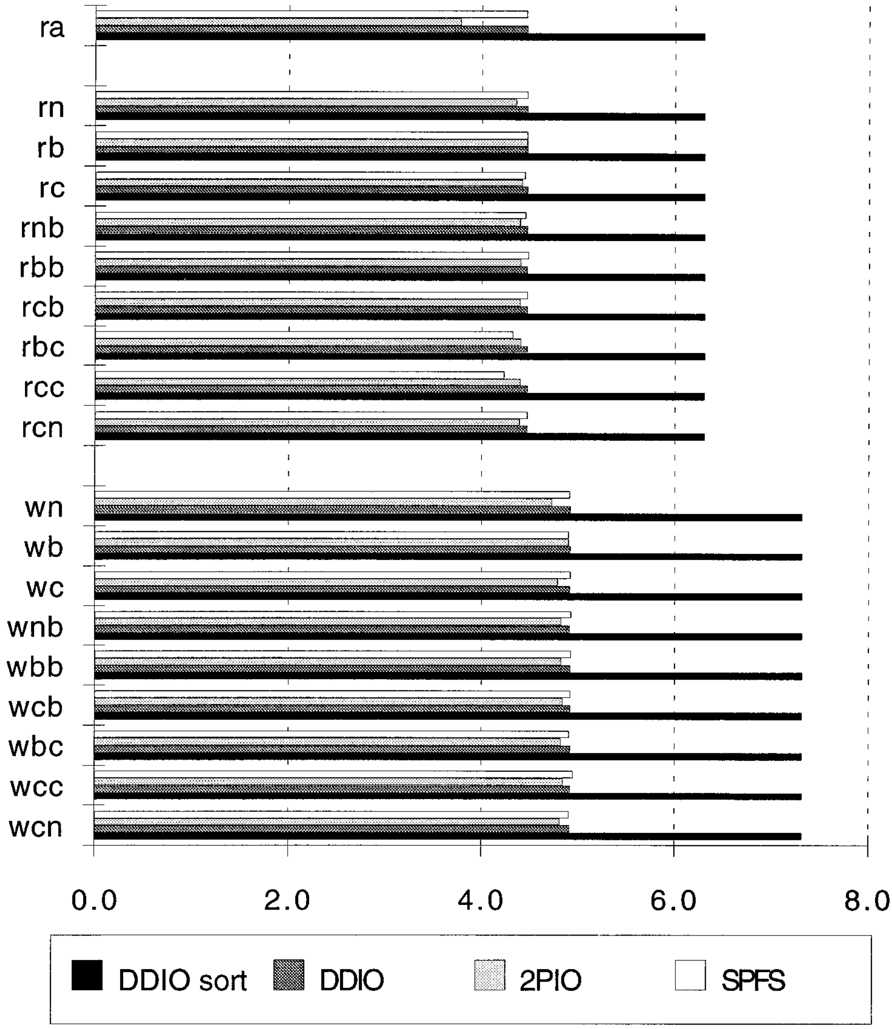

Fig. 5. Comparing the throughput of disk-directed I/O (DDIO) to that of two-phase I/O (2PIO) and the simple parallel file system (SPFS), on a random-blocks disk layout using patterns with an 8192-byte record size. ra throughput has been normalized by the number of CPs. Each point represents the average of five trials of an access pattern (the maximum *cv* is 0.031).

same 10MB data to all 16 CPs, its apparent throughput was inflated. We have normalized ra throughput in all of our graphs by dividing by the number of CPs.

Figures 4 and 5 display the performance on a random-blocks disk layout: Figure 4 for patterns with 8-byte records and Figure 5 for patterns with 8192-byte records. On each graph, four cases are shown for each access pattern: simple parallel file system (SPFS), two-phase I/O (2PIO), and disk-directed I/O (DDIO) with and without a presort of the block requests by physical location. Note that the disks' peak multitrack transfer rate was 33.8MB/s, but with a random-blocks disk layout it was impossible to come close to that throughput. Throughput for disk-directed I/O with presorting consistently reached 6.3MB/s for reading and 7.3MB/s for writing. In contrast, SPFS throughput was highly dependent on the access pattern, was never faster than 5MB/s, and was particularly slow for many 8-byte patterns. Cases with small chunk sizes were the slowest, as slow as 0.8MB/s, due to the tremendous number of requests required to transfer the data. As a result, disk-directed I/O with presorting was up to 8.7 times faster than the simple parallel file system.

Figures 4 and 5 also make clear the benefit of presorting disk requests by physical location, an optimization available in disk-directed I/O to an extent not possible in the simple parallel file system or in two-phase I/O. Even so, disk-directed I/O *without* presorting was faster than the simple parallel file system in most cases. At best, it was 5.9 times faster; at worst, there was no noticeable difference. Disk-directed I/O thus improved performance in two ways: by reducing overhead and by presorting the block list.

Figures 4 and 5 demonstrate the mixed results of two-phase I/O. It was slightly *slower* than the simple parallel file system for most patterns with 8KB records, because it did not overlap the permutation with the I/O. Of the statistically significant differences, most were only 2–4%, although in the ra pattern two-phase I/O was 16% slower. Two-phase I/O did substantially improve performance (by as much as 5.1 times) on small-chunk-size patterns. Two-phase I/O matched the performance of disk-directed I/O without presorting in most patterns, although disk-directed I/O was still about 20% faster in ra and some 8-byte cyclic patterns, because it could overlap the costly permutation with the disk I/O. With disk-directed I/O's additional advantage of presorting the block list, it was 41–79% faster than two-phase I/O.

To test the ability of the different file system implementations to take advantage of disk layout, and to expose other overheads when the disk bandwidth could be fully utilized, we compared the two methods on a contiguous disk layout (Figures 6 and 7). I/O on this layout was much faster than on the random-blocks layout, by avoiding the disk-head movements caused by random layouts and by benefiting from the disks' own read-ahead and write-behind caches. In most cases disk-directed I/O moved about 31.4MB/s, which was a respectable 93% of the disks' peak multitrack transfer rate of 33.8MB/s. The few cases where disk-directed I/O did not get as close to the peak disk-transfer rate were affected by the overhead of moving individual 8-byte records to and from the CPs. (In our earlier results [Kotz 1994], the performance was worse: the "batched" Memput and Memget operations used here improved performance by 10–24% on these patterns [Kotz 1996c].)

Discussion. The simple parallel file system was often unable to obtain the full disk bandwidth and had particular trouble with the 8-byte patterns. Although there were cases where the simple parallel file system could match disk-directed I/O, disk-directed I/O was as much as 18.1 times

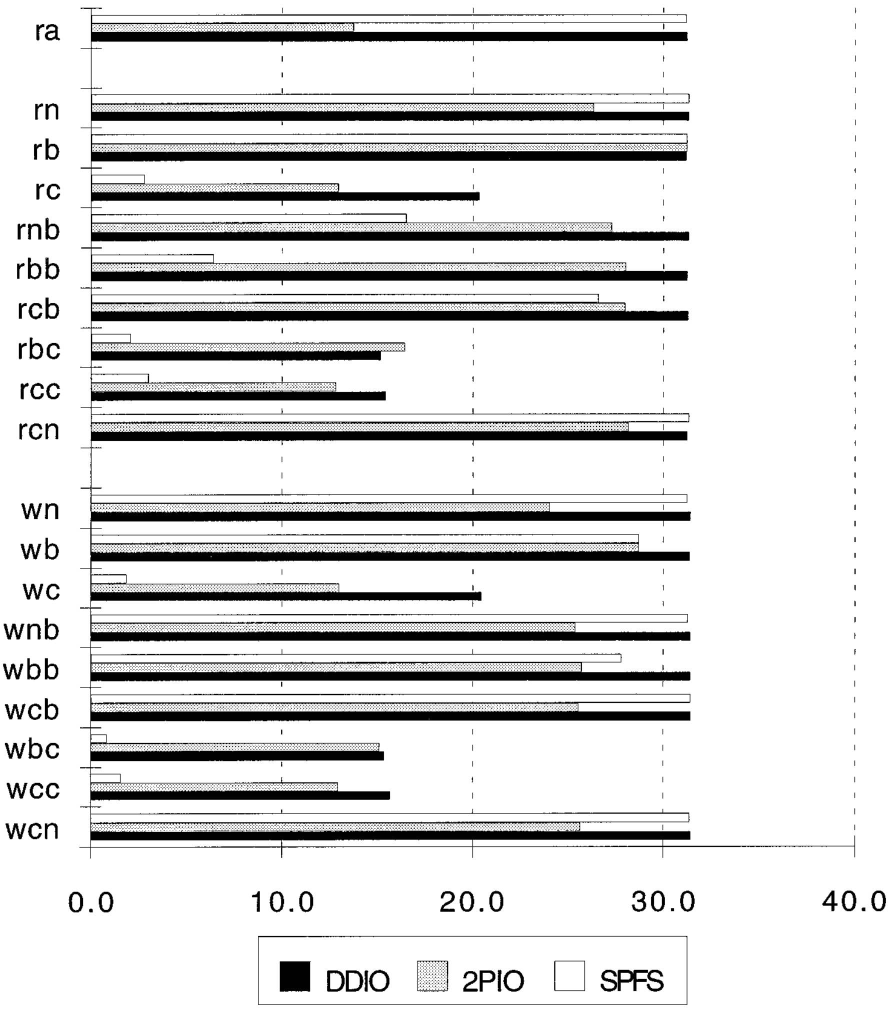

Fig. 6. Comparing the throughput of disk-directed I/O (DDIO), two-phase I/O (2PIO), and the simple parallel file system (SPFS), on a contiguous disk layout, using access patterns with 8-byte records. Note that "DDIO" and "DDIO sort" are identical here, because the logical block numbers are identical to the physical block numbers, so the sort is a no-op. ra throughput has been normalized by the number of CPs. Each point represents the average of five trials of an access pattern (the maximum *cv* is 0.052, except for 0.25 on 8-byte wc on SPFS). Note that the peak disk throughput was 33.8MB/s.

faster than the simple parallel file system. The simple parallel file system had several difficulties:

—When the CPs were using patterns with 8-byte chunks (rc, rbc, rcc, wc, wbc, and wcc), many IOP-request messages were necessary to transfer the small noncontiguous records, requiring many expensive IOP-cache accesses. It could have been worse: the cache successfully caught the interprocess spatial locality of these patterns; if the CPs had been poorly synchronized the cache would have thrashed.

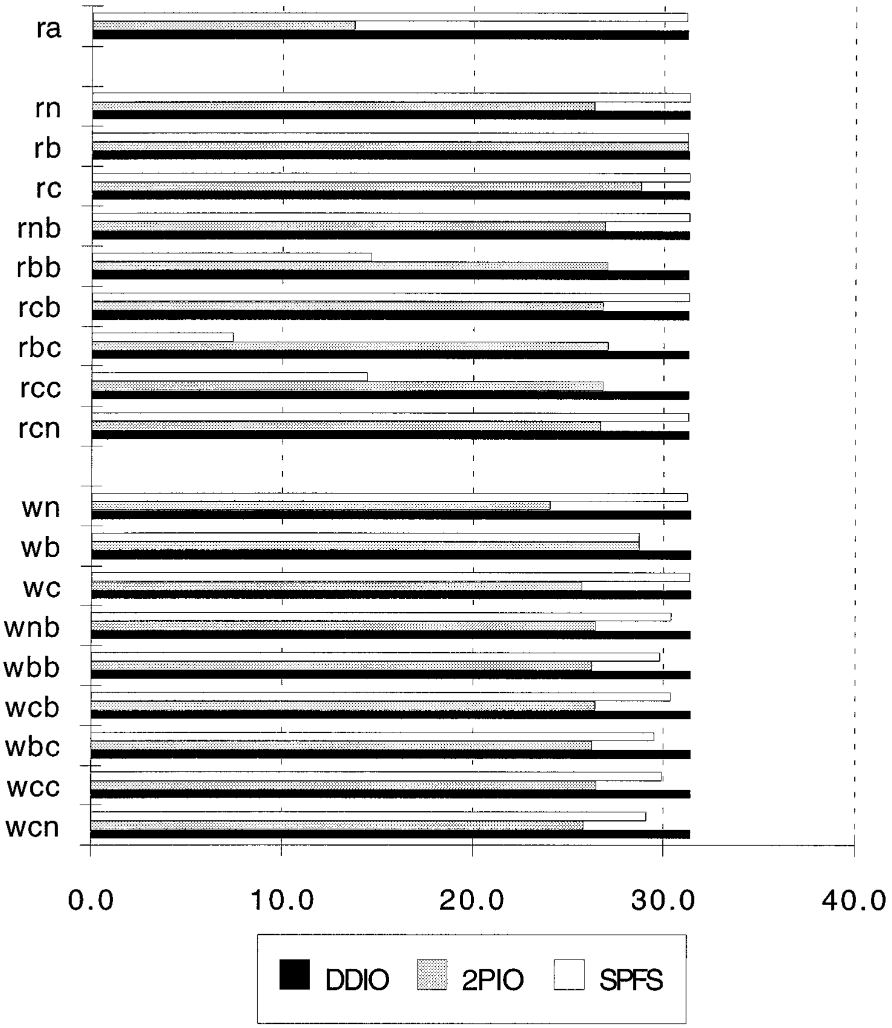

Fig. 7. Comparing the throughput of disk-directed I/O (DDIO), two-phase I/O (2PIO), and the simple parallel file system (SPFS), on a contiguous disk layout, using access patterns with 8192-byte records. Note that "DDIO" and "DDIO sort" are identical here, because the logical block numbers are identical to the physical block numbers, so the sort is a no-op. ra throughput has been normalized by the number of CPs. Each point represents the average of five trials of an access pattern (the maximum *cv* is 0.024). Note that the peak disk throughput was 33.8MB/s.

—When the CPs were active at widely different locations in the file (e.g., in rb, rbb, rbc, or rcc, with 8KB records), there was little interprocess spatial locality. In the contiguous layout, these multiple localities defeated the disk's internal caching and caused extra head movement, both a significant performance loss. Fortunately, disk scheduling and the ability to request up to four blocks per CP per disk allowed the rb pattern (which transfers data in large chunks) to avoid most of this problem

[Kotz 1996c]. In doing so, it used a schedule that allowed some CPs to progress much more quickly than others; this is an example of an instance where load imbalance and service that is unfair to some CPs can lead to much better collective performance.

—Patterns reading medium-sized chunks (rbb, rbc, rcc with 8KB records) were slow because the application made only one request at a time (to each CP), and the small chunk size prevented the CPs from issuing many requests to the IOPs. The IOPs' disk queues thus had few requests, and thus the disk was forced to seek from one region to another. The rbb pattern, when mapped onto a larger file (1000MB), had larger chunks and thus was able to fill the disk queues and realize the full bandwidth (not shown).

The corresponding write patterns (wbb, wbc, wcc), however, were more successful. The IOP caches were large enough (4MB) to hold much of the file (10MB). The numerous small CP writes completed quickly, filling the cache and thus filling the disk queues, leading to a disk schedule nearly as efficient as that used in disk-directed I/O. This effect would be negligible in a huge file.

—The high data rates of the contiguous disk layout expose the cache management overhead in the simple parallel file system, particularly in the access patterns with small chunks.

Two-phase I/O usually helped avoid the worst troubles of the simple parallel file system, particularly for small records. It had several problems of its own, however:

—Despite making larger requests to the file system, the flow control limitations prevented it from making enough requests to the IOPs to fill the disk queues as well as disk-directed I/O, so it was less able to optimize the disk accesses in the random-blocks layout.

—The additional permutation step prevented it from matching disk-directed-I/O performance in most patterns, even with 8192-byte records and a contiguous layout. Indeed, the cost of the permutation occasionally resulted in lower throughput than the simple parallel file system, even for 8-byte records.

Disk-directed I/O was not perfect, of course. Note that disk-directed I/O chose the same (optimal) disk schedule for all access patterns. Thus, any difference in performance between two access patterns was due to the time spent delivering the data to CPs when reading, or gathering the data from CPs when writing. IOP double-buffering allowed this communication to overlap the I/O. The I/O time was sufficiently high in the random-blocks layout to cover the communication overhead of all access patterns. The I/O time was low in the contiguous layout, but still large enough to cover the communication time in most access patterns. The patterns with 8-byte chunks (rc, rbc, rcc, wc, wbc, and wcc), however, required a lot of communication and computation from the IOP, which became the limiting factor in the performance.

Table III. Performance Relative to SPFS

Method	Minimum	Geometric Mean	Maximum
DDIO sort	1.00	1.64	18.10
DDIO no sort	1.00	1.18	5.88
2PIO	0.44	1.20	17.83

Indeed, in one case (8-byte rbc in the contiguous layout), disk-directed I/O was 8% slower than two-phase I/O. In this situation, where communication was the limiting factor, the optimal I/O pattern was not the optimal communication pattern. The optimal I/O pattern read the file from beginning to end, which meant that the rows of the matrix were read in increasing order. In our rbc distribution (see Figure 3 and Table II) this ordering meant that the IOPs were communicating with only four CPs at a time, leading to network congestion. In the two-phase-I/O permutation phase, however, all 16 CPs were communicating simultaneously, with less congestion. The solution would be to have each IOP rotate its I/O list by a different amount, so as to start its I/O pattern at a different point, costing one disk seek but staggering the communications and reducing congestion.

Summary. The above results give a rough picture of the kind of performance improvements possible in a workload that reads and writes matrices in files. To summarize, consider the ratio of the throughput offered by disk-directed I/O, or two-phase I/O, to that offered by the simple parallel file system on a particular access pattern. A ratio greater than one indicates that the method was faster than the simple parallel file system on that access pattern. We summarize that "improvement factor" across all access patterns, looking at the minimum, geometric mean [Jain 1991, p. 191], and maximum (Table III).

We include "DDIO no sort" only for comparison, as one would always want to use the sorting feature; these numbers apply only to the random disk layout. We can see that although DDIO (with sorting) sometimes makes no difference (ratio 1.00), it is on average 64% faster and was up to 18 times faster. Although two-phase I/O was also about 18 times faster in one case (8-byte wbc), it was otherwise no more than 8.24 times faster, sometimes much slower than the simple parallel file system, and only 20% faster on average.

6.1 Sensitivity

To evaluate the sensitivity of our results to some of the parameters, we independently varied the number of CPs, number of IOPs, number of disks, file size, and record size. It was only feasible to experiment with a subset of all configurations, so we chose a subset that would push the limits of the system by using the contiguous layout, and exhibit most of the variety shown earlier, by using the patterns ra, rn, rb, and rc with 8KB records. ra throughput was normalized as usual (see the beginning of Section 6). Since the conclusions from two-phase I/O were nearly always the same as those

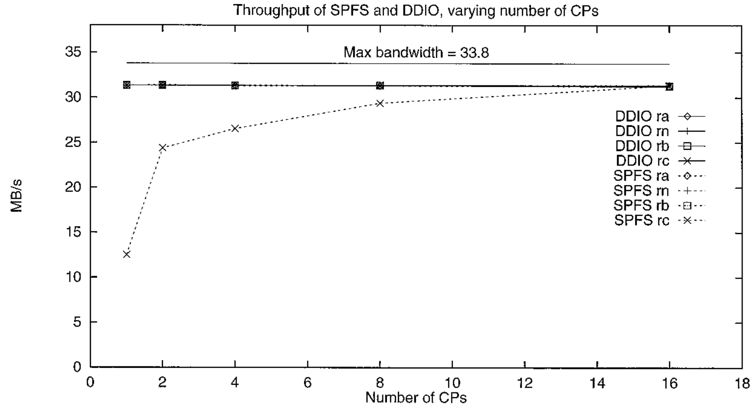

Fig. 8. A comparison of the throughput of disk-directed I/O (DDIO) and the simple parallel file system (SPFS), as the number of CPs varied, for the ra, rb, and rc patterns (ra throughput has been normalized by the number of CPs). All cases used the contiguous disk layout, and all used 8KB records. See Figure 9 for 2PIO results.

from the simple parallel file system, we plot two-phase I/O only where the conclusions differ from the simple parallel file system.

We first varied the number of CPs (Figure 8), holding the number of IOPs and disks fixed and maintaining the cache size for the simple parallel file system at eight buffers per disk *per CP*. It may seem unusual to consider a configuration with fewer CPs than IOPs. Most multiprocessors are shared, however, so it is not unlikely for an application to occasionally run on a small subset of CPs, while accessing files that are declustered across the larger, complete set of IOPs.

Most cases were unaffected; the most interesting effect was the poor performance of the simple parallel file system on the rc pattern. Recall that in the simple parallel file system all the parallelism is generated by the CPs, either from splitting large requests into concurrent smaller requests or from several CPs making concurrent requests. With one-block records and no buffers at the CP, each file system call could only use one disk, and then with only one outstanding request. With fewer CPs than IOPs, the full disk parallelism was not used.

Unlike in our other variations, below, two-phase I/O behaved quite differently from the simple parallel file system. Results from the contiguous layout are shown in Figure 9. Similar results were found with the random-blocks layout (not shown). As with the simple parallel file system, the rb throughput was unaffected by the number of CPs. Since rb was the I/O access pattern always used by two-phase I/O, the reduced throughput seen for ra, rn, and rc was due entirely to slowness in the permutation. With one CP, the permutation was local to one CP and was thus fairly fast (it would have matched rb if the code was changed to test for this special case, avoiding the permutation). Otherwise, the permutation throughput steadily improved for rn and rc, as more CPs provided more CPUs, memories, and network interfaces for moving the data. The normalized permuta-

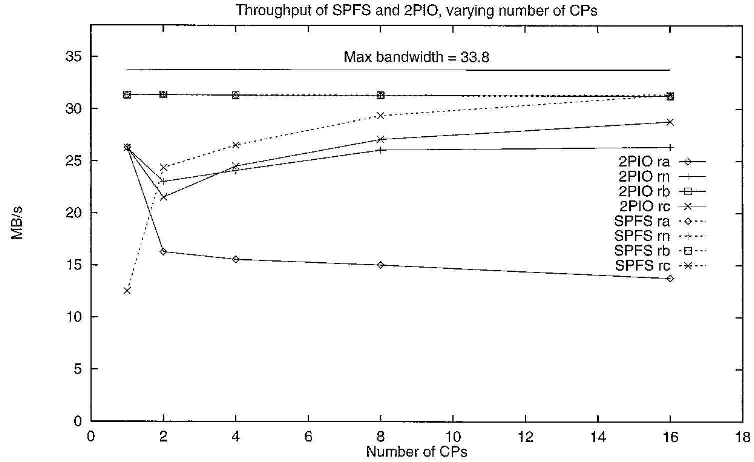

Fig. 9. A comparison of the throughput of two-phase I/O (2PIO) and the simple parallel file system (SPFS), as the number of CPs varied, for the ra, rn, rb, and rc patterns (ra throughput has been normalized by the number of CPs). All cases used the contiguous disk layout, and all used 8KB records. See Figure 8 for DDIO results.

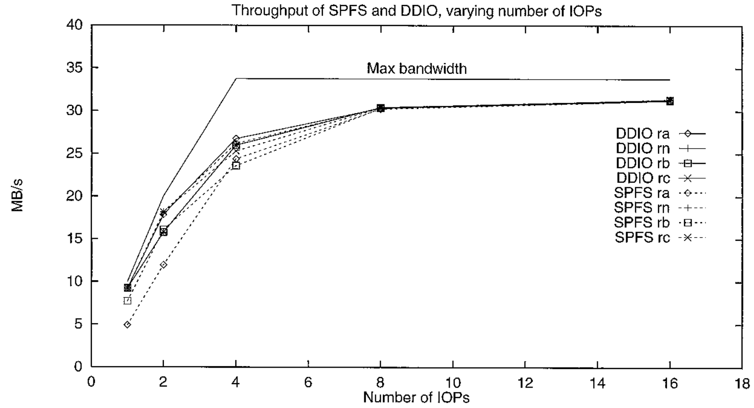

Fig. 10. A comparison of the throughput of disk-directed I/O (DDIO) and the simple parallel file system (SPFS), as the number of IOPs (and busses) varied, for the ra, rn, rb, and rc patterns (ra throughput has been normalized by the number of CPs). All cases used the contiguous disk layout, and all used 8KB records. The maximum bandwidth was determined by either the busses (1–2 IOPs) or the disks (4–16 IOPs).

tion throughput decreases for ra, due to increasing contention in this all-to-all permutation (recall that for ra the amount of data moved increases with the number of CPs).

We then varied the number of IOPs (and SCSI busses), holding the number of CPs, number of disks, and total cache size fixed (Figure 10). Performance decreased with fewer IOPs because of increasing bus contention, particularly when there were more than two disks per bus and was ultimately limited by the 10MB/s bus bandwidth. Indeed, with two IOPs

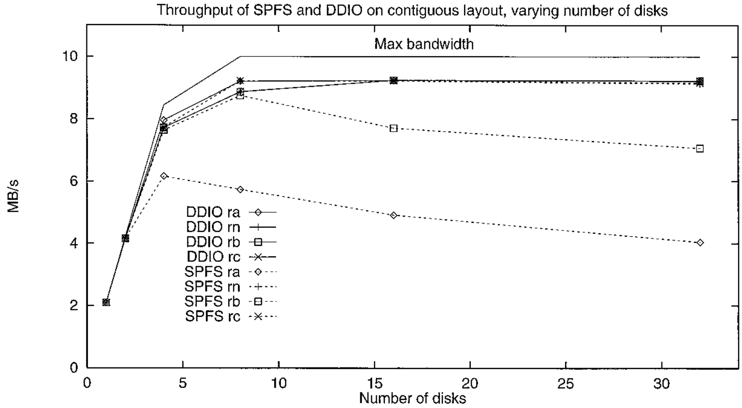

Fig. 11. A comparison of the throughput of disk-directed I/O (DDIO) and the simple parallel file system (SPFS), as the number of disks varied, for the ra, rn, rb, and rc patterns (ra throughput has been normalized by the number of CPs). All cases used the contiguous disk layout, and all used 8KB records. The maximum bandwidth was determined either by the disks (1–4) or by the (single) bus (8–32 disks).

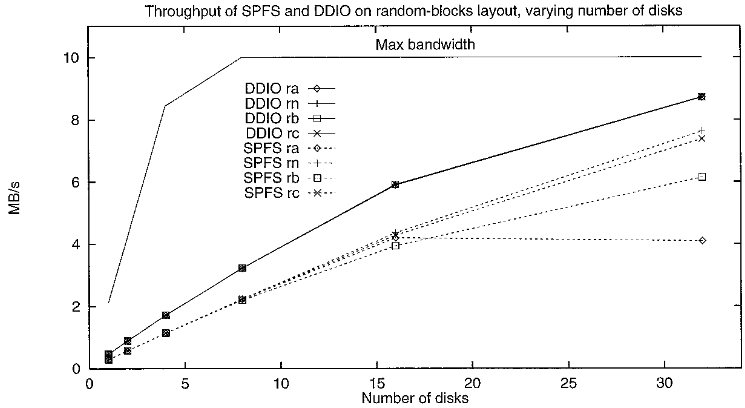

Fig. 12. Similar to Figure 11, but here all cases used the random-blocks disk layout. DDIO used the block-list presort.

the simple parallel file system was 13% faster than disk-directed I/O in the rn and rc patterns, due to a subtle implementation issue.[3]

We then varied the number of disks, using one IOP, holding the number of CPs at 16, and maintaining the simple-system cache size at eight buffers per CP *per disk* (Figures 11 and 12). Performance scaled with more disks,

---

[3] Disk-directed I/O used three bus transactions when reading, rather than two: first, the host asked the disk to prefetch the desired block into its cache; second, the host asked the disk to transfer the data to the host; third, the disk transferred the data to the host. The first request is unusual, but our implementation sometimes knew the identity of the next block before it knew the location of the buffer that would hold the data. This scheme normally improved performance, but when the bus was congested, the extra delay slowed down the file system.

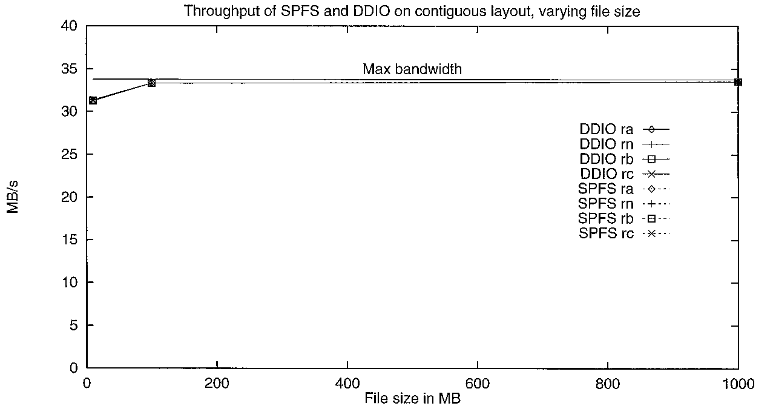

Fig. 13. A comparison of the throughput of disk-directed I/O (DDIO) and the simple parallel file system (SPFS), as the file size varied, for the ra, rn, rb, and rc patterns (ra throughput has been normalized by the number of CPs). All cases used the contiguous disk layout, and all used 8KB records.

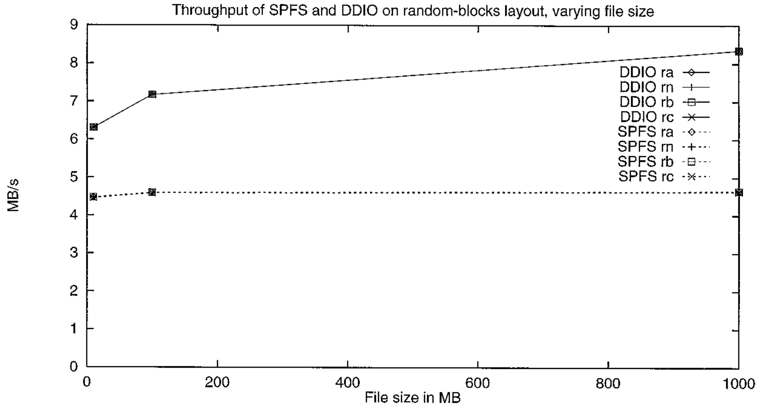

Fig. 14. A comparison of the throughput of disk-directed I/O (DDIO) and the simple parallel file system (SPFS), as the file size varied, for the ra, rn, rb, and rc patterns (ra throughput has been normalized by the number of CPs). All cases used the random-blocks disk layout, and all used 8KB records. Here the disk-directed I/O includes a presort; similar conclusions were obtained without the presort.

approaching the 10MB/s bus-speed limit. The simple parallel file system had particular difficulty with the rb and ra patterns. The large chunk sizes in these patterns sent a tremendous number of requests to the single IOP, and it appears that throughput was degraded by the overhead on the IOP CPU.

In most of this article we simulate 10MB files. To examine the effect of this choice, Figures 13 and 14 compare throughputs for files 10 and 100 times larger. Though the maximum throughputs were reached with files 100MB or larger, we chose 10MB for simulation efficiency. The *relative*

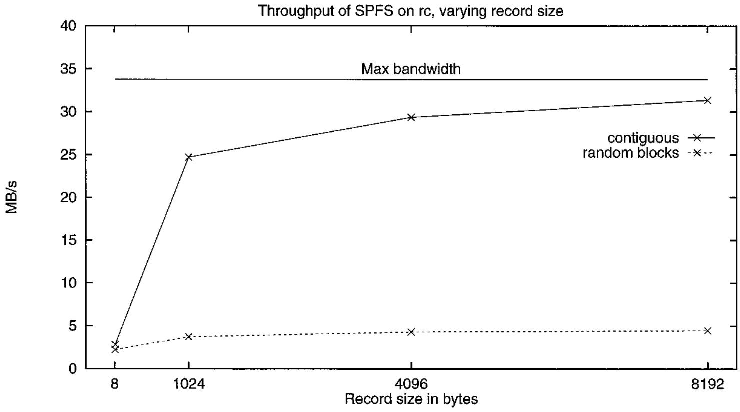

Fig. 15. The throughput of the simple parallel file system on rc patterns of various record sizes, for both the contiguous and random-blocks layouts.

order of test cases remained the same. The maximum throughput attained was 33.5MB/s, which is 99% of the peak disk transfer bandwidth.

In this article we focus on 8- and 8192-byte record sizes. Figure 15 shows the effect of other record sizes in situations where the record size was expected to make the most difference: in the simple parallel file system on rc, using both contiguous and random-blocks layouts. This plot justifies our focus on the extremes; 8-byte records limited throughput through excessive overhead, while 8192-byte records reduced overhead and exposed other limits (here, the disk bandwidth in the random-blocks layout).

Summary. These variation experiments showed that while the relative benefit of disk-directed I/O over two-phase I/O or the simple parallel file system varied, disk-directed I/O consistently provided excellent performance, almost always better than the simple parallel file system, often independent of access pattern, and often close to hardware limits.

7. INTERFACES FOR DISK-DIRECTED I/O

There are two interfaces that are important to consider when implementing a disk-directed I/O system: the application programmer's interface (API) and the internal CP-IOP interface. Although we do not propose any specific interfaces in this article, it should be possible to use any of several existing interfaces in the construction of a disk-directed I/O system.

7.1 Application Programmer's Interface

The interesting characteristic of an API is its capability to specify which parts of the file are desired and how the data are distributed among the CPs' buffers. Perhaps the most common behavior is to collectively transfer a data set that is contiguous within the file, but distributed among processor memories in some interesting way. There are at least three fundamental styles of API for parallel I/O, each of which provides a different kind of solution to this problem.

The first style allows the programmer to directly read and write data structures such as matrices; Fortran provides this style of interface, as do many libraries [Bennett et al. 1994; Galbreath et al. 1993; Karpovich et al. 1994; Seamons et al. 1995; Thakur et al. 1996]. Some object-oriented interfaces go even further in this direction [Karpovich et al. 1994; Krieger and Stumm 1996; Seamons et al. 1995]. As long as your data structure can be described by a matrix, and the language or library also provides ways to describe distributed matrices, this interface provides a neat solution.

The second style provides each processor its own "view" of the file, in which noncontiguous portions of the file appear to be contiguous to that processor. By carefully arranging the processor views, the processors can use a traditional I/O transfer call that transfers a contiguous portion of the file (in their view) to or from a contiguous buffer in their memory, and yet still accomplish a nontrivial data distribution. The most notable examples of this style include a proposed nCUBE file system [DeBenedictis and del Rosario 1992], Vesta [Corbett and Feitelson 1996], and MPI-IO [Corbett et al. 1996a; MPI-IO Committee 1996].

The third style has neither an understanding of high-level data structures, like the first, nor per-process views of the file, like the second. Each call specifies the bytes of the file that should be transferred. This interface is common when using the C programming language in most MIMD systems, although many have special file pointer modes that help in a few simple situations (Intel CFS [Pierce 1989] and TMC CMMD [Best et al. 1993], for example). None of these allow the processor to make a single file system request for a complex distribution pattern. More sophisticated interfaces, such as the nested-batched interface [Nieuwejaar and Kotz 1996], can specify a list, or a strided series, of transfers in a single request. This latter interface is perhaps the most powerful (efficient and expressive) of this style of interface.

Any of the above interfaces that support collective requests and can express nontrivial distributions of data among the processor memories would be sufficient to support disk-directed I/O. These include (at least) HPF and other SPMD languages, the nested-batched interface [Nieuwejaar and Kotz 1996] with collective extensions, Vesta [Corbett and Feitelson 1996], MPI-IO [Corbett et al. 1996a; MPI-IO Committee 1996], and most of the matrix libraries [Bennett et al. 1994; Galbreath et al. 1993; Karpovich et al. 1994; Seamons et al. 1995; Thakur et al. 1996].

7.2 CP-IOP Interface

Once the application programmer has expressed the desired data transfer, how do the compute processors communicate that information to all of the IOPs, and how do the IOPs use the information to arrange the data transfer?

In the experiments of Section 5, all of the possible data distribution patterns (e.g., block-cyclic) were understood by the IOPs, so the CPs needed only to request a particular distribution pattern and to provide a few parameters. A more realistic system should be more flexible: it should support the common matrix distributions easily, and it should support arbitrary distributions and irregular data structures.

Fortunately, several compiler groups have developed compact parameterized formats for describing matrix distributions [Brezany et al. 1995; Thakur et al. 1994]. This compact description of the distribution pattern, generated by a compiler or a matrix support library, can be passed to the IOPs. A few calculations can tell the IOP which file blocks it should be transferring, and for each file block, the in-memory location of the data (CP number and offset within that CP's buffer).

To support complex or irregular distributions, each CP can send a single nested-batched request [Nieuwejaar and Kotz 1996] to each IOP. Such requests can capture complex but regular requests in a compact form, but can also capture completely irregular requests as a list. These compact requests can be easily converted into a list of blocks, for I/O, and later used for mapping each block into the in-memory location (CP number, CP offset) of the data [Kotz 1995b].

The combination of the compact parameterized descriptions for common matrix distributions and the fully general nested-batched interface [Nieuwejaar and Kotz 1996] is sufficient to support disk-directed I/O efficiently.

8. EXPANDING THE POTENTIAL OF DISK-DIRECTED I/O

The idea of disk-directed I/O can be expanded to include several other interesting possibilities [Kotz 1995a]. Assuming some mechanism exists to run application-specific code on the IOPs, the IOPs could do more with the data than simply transfer it between CP memories and disk.

Data-Dependent Distributions. In some applications, the data set must be divided among the CPs according to the *value* of the records, rather than their position in the file. Using a simple parallel file system, it is necessary to use a two-phase I/O approach. The CPs collectively read all of the data into memory. As each record arrives at a CP, the CP examines the record, determines the actual destination of that record, and sends the record to the appropriate destination CP. By moving this distribution function to the IOPs, the data could be sent directly to the destination CP, halving the total network traffic (for experimental results, see Kotz [1995a]). Unless the additional work overloads the IOPs, reduced network traffic would lead to better throughput in systems with slow or congested networks.

Data-Dependent Filtering. Some applications wish to read a subset of the records in a file, where the subset is defined by the *value* of the data in the records, rather than their position in the file. Using a simple parallel file system, the CPs must read all of the data and then discard the undesired records. By moving this record-filtering function to the IOPs, undesired records would never be sent to CPs, reducing network traffic (for experimental results, see Kotz [1995a]). In systems with slow or congested networks, that lower traffic would lead to better throughput. A similar technique has already been demonstrated in some database systems [Borr and Putzolu 1988].

9. RELATED WORK

Disk-directed I/O is somewhat reminiscent of the PIFS (Bridge) “tools” interface [Dibble 1990], in that the data flow is controlled by the file system rather than by the application. PIFS focuses on managing *where* data flows (for memory locality), whereas disk-directed I/O focuses more on *when* data flows (for better disk and cache performance).

Some parallel database machines use an architecture similar to disk-directed I/O, in that certain operations are moved closer to the disks to allow for more optimization. By moving some SQL processing to the IOPs, one system was able to filter out irrelevant tuples at the IOPs, reducing the data volume sent to the CPs [Borr and Putzolu 1988].

Some matrix-I/O libraries significantly improve performance by changing the underlying matrix storage format [Karpovich et al. 1994; Sarawagi and Stonebraker 1993; Seamons and Winslett 1994; Toledo and Gustavson 1996]. These libraries could use a disk-directed file system to obtain even better performance, transparently to the end user.

The Jovian collective-I/O library [Bennett et al. 1994] tries to coalesce fragmented requests from many CPs into larger requests that can be passed to the IOPs. Their “coalescing processes” are essentially a dynamic implementation of the two-phase-I/O permutation phase.

Transparent Informed Prefetching (TIP) enables applications to submit detailed hints about their future file activity to the file system, which can then use the hints for accurate, aggressive prefetching [Patterson et al. 1995]. Aggressive prefetching serves to provide concurrency to disk arrays and deeper disk queues to obtain better disk schedules. In this sense TIP and disk-directed I/O are similar. TIP, however, has no explicit support for parallel applications, let alone collective I/O, and thus would need to be extended. Furthermore, once an application provides hints to TIP it uses the traditional Unix-like file system interface, retaining the overhead of processing many tiny requests. The application requests I/O in the same sequence, limiting the potential for reordering within the disk queues due to limited buffer space. Finally, TIP offers no benefits for writing, only for reading.

Our model for managing a disk-directed request, that is, sending a high-level request to all IOPs which then operate independently under the assumption that they can determine the necessary actions to accomplish the task, is an example of *collaborative execution* like that used in the TickerTAIP RAID controller [Cao et al. 1994].

Finally, our Memput and Memget operations are not unusual. Similar remote-memory-access mechanisms are supported in a variety of distributed-memory systems [Culler et al. 1993; Hayashi et al. 1994; Wheat et al. 1994].

10. IMPLEMENTATIONS OF DISK-DIRECTED I/O

The original appearance of this research [Kotz 1994] inspired several other research projects.

ENWRICH [Purakayastha et al. 1996] uses our simulator to investigate the viability of CP caching in write-only access patterns. In ENWRICH, CPs using a traditional Unix-like application interface accumulate small writes in local buffers, then use disk-directed I/O to collectively flush the buffers when they become full.

The Panda library for collective matrix I/O uses a variant of disk-directed I/O they call *server-directed I/O* [Chen et al. 1996; Seamons et al. 1995]. Panda is implemented on top of a traditional Unix file system, so they cannot obtain information about the physical disk layout to use in their preliminary sort. Otherwise, Panda's technique is like ours. Results from their implementation on an IBM SP-2 validate the benefits of disk-directed I/O over a noncollective, client-directed approach.

The Galley parallel file system [Nieuwejaar and Kotz 1997] provides a compromise interface: it has no collective requests, but it has structured requests that allow strided chunks of the file to be transferred in a single request. The implementation essentially uses a noncollective version of disk-directed I/O: a single complex request is sent from each CP to the IOP in the form of a list of contiguous chunks to be transferred from that IOP's disk to that CP. The IOP converts the list of chunks into a list of blocks. First, it checks the cache to transfer any data that needs no disk I/O. Then it passes a list of block-transfer requests to the disk thread, which sorts them into a disk schedule based on the disk layout. As the disk works through the schedule, it sends data to (or fetches data from) the CP. Notice that if many CPs are simultaneously requesting complementary chunks from the file, as one would expect in a collective operation, their requests will dynamically meet each other in the cache and the disk queue. (Note that it is important for the CPs to be approximately synchronized in their file-access patterns, to avoid cache thrashing.) The performance is often similar to that of a pure disk-directed I/O implementation [Nieuwejaar and Kotz 1997].

11. CONCLUSIONS

Our simulations show that disk-directed I/O avoided many of the pitfalls inherent in the simple parallel file system (SPFS), such as cache thrashing, extraneous disk-head movements, extraneous prefetches, excessive request-response traffic between CP and IOP, inability to use all the disk parallelism, inability to use the disks' own caches, overhead for cache management, and memory-memory copies. Furthermore, disk-directed I/O was able to schedule disk requests across the entire access pattern, rather than across a smaller set of "current" requests. As a result, disk-directed I/O could provide consistent performance close to the limits of the disk hardware. Indeed, it was in one case more than 18 times faster than the SPFS, despite the fact that our SPFS implementation included simplifying assumptions that should overestimate its performance. Finally, the performance of disk-directed I/O was nearly independent of the distribution of data to CPs.

Our results also show that while two-phase I/O could substantially improve performance over the simple parallel file system, it could also reduce performance. Furthermore, it was often unable to match the performance of disk-directed I/O, largely because it did not overlap the I/O with the permutation.

As presented here, disk-directed I/O would be most valuable when making large, collective transfers of data between multiple disks and multiple memories, whether for loading input data, storing result data, or swapping data to a scratch file in an out-of-core algorithm. Indeed, the data need not be contiguous [Kotz 1995a], and the Galley results show that the interface need not be collective [Nieuwejaar and Kotz 1997]. The concept of disk-directed I/O can also be extended to other environments. Our Memput and Memget operations would be easily implemented on a shared-memory machine with a block-transfer operation, for example. Although our patterns focused on the transfer of 1D and 2D matrices, we expect to see similar performance for higher-dimensional matrices and other regular structures. Finally, there is potential to implement transfer requests that are more complex than simple permutations—for example, selecting only a subset of records whose data values match some criterion, or distributing records to CPs based on their value, rather than file position.

Our results emphasize that simply layering a new interface on top of a simple parallel file system will not suffice. For maximum performance the file system interface must allow CPs to make large, noncontiguous requests and should support collective-I/O operations. The file system software (in particular, the IOP software) must be redesigned to use mechanisms like disk-directed I/O. Nonetheless, there is still a place for caches. Irregular or dynamic access patterns involving small, independent transfers and having substantial temporal or interprocess locality will still benefit from a cache. The challenge, then, is to design systems that integrate the two techniques smoothly. Despite not having explicit support for collective I/O, the Galley Parallel File System [Nieuwejaar and Kotz 1997] is one such system; its disk-directed approach to serving complex requests from individual CPs leads to excellent performance under many collective access patterns.

11.1 Future Work

There are many directions for future work in this area:

—integrate with I/O-optimizing compilers [Cormen and Colvin 1994; Thakur et al. 1996],

—optimize concurrent disk-directed activities, and

—explore the possibility of “programmable” IOPs [Kotz and Nieuwejaar 1996].

11.2 Availability

The full simulator source code is available on the WWW at URL

http://www.cs.dartmouth.edu/research/starfish/

The disk model software can be found at

http://www.cs.dartmouth.edu/cs_archive/diskmodel.html.

Many of the references below are available at

http://www.cs.dartmouth.edu/pario/bib/.

ACKNOWLEDGMENTS

Thanks to Song Bac Toh and Sriram Radhakrishnan for implementing and validating the disk model; to Chris Ruemmler, John Wilkes, and Hewlett Packard Company for allowing us to use their disk traces to validate our disk model and for their help in understanding the details of the HP 97560; to Denise Ecklund of Intel for help understanding the Paragon interconnection network; to Eric Brewer and Chrysanthos Dellarocas for Proteus; to Tom Cormen, Keith Kotay, Nils Nieuwejaar, the anonymous reviewers, and especially Karin Petersen for feedback on drafts of this article.

REFERENCES

ACHARYA, A., UYSAL, M., BENNETT, R., MENDELSON, A., BEYNON, M., HOLLINGSWORTH, J. K., SALTZ, J., AND SUSSMAN, A. 1996. Tuning the performance of I/O intensive parallel applications. In *Proceedings of the 4th Workshop on Input/Output in Parallel and Distributed Systems.* ACM Press, New York, 15–27.

BENNETT, R., BRYANT, K., SUSSMAN, A., DAS, R., AND SALTZ, J. 1994. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference.* IEEE Computer Society Press, Los Alamitos, Calif., 10–20.

BEST, M. L., GREENBERG, A., STANFILL, C., AND TUCKER, L. W. 1993. CMMD I/O: A parallel Unix I/O. In *Proceedings of the 7th International Parallel Processing Symposium.* IEEE Computer Society Press, Los Alamitos, Calif., 489–495.

BORDAWEKAR, R., DEL ROSARIO, J. M., AND CHOUDHARY, A. 1993. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93.* IEEE Computer Society Press, Los Alamitos, Calif., 452–461.

BORR, A. J. AND PUTZOLU, F. 1988. High performance SQL through low-level system integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* ACM Press, New York, 342–349.

BREWER, E. A., DELLAROCAS, C. N., COLBROOK, A., AND WEIHL, W. E. 1991. Proteus: A high-performance parallel-architecture simulator. Tech. Rep. MIT/LCS/TR-516, MIT, Cambridge, Mass.

BREZANY, P., MUECK, T. A., AND SCHIKUTA, E. 1995. Language, compiler and parallel database support for I/O intensive applications. In *Proceedings of the International Conference on High Performance Computing and Networking.* Lecture Notes in Computer Science, vol. 919. Springer-Verlag, Berlin, 14–20.

BRUNET, J.-P., PEDERSEN, P., AND JOHNSSON, S. L. 1994. Load-balanced LU and QR factor and solve routines for scalable processors with scalable I/O. In *Proceedings of the 17th IMACS World Congress.* Also available as Harvard Univ. Computer Science Tech. Rep. TR-20-94.

CAO, P., LIM, S. B., VENKATARAMAN, S., AND WILKES, J. 1994. The TickerTAIP parallel RAID architecture. *ACM Trans. Comput. Syst. 12,* 3 (Aug.), 236–269.

CHEN, Y., WINSLETT, M., SEAMONS, K. E., KUO, S., CHO, Y., AND SUBRAMANIAM, M. 1996. Scalable message passing in Panda. In *Proceedings of the 4th Workshop on Input/Output in Parallel and Distributed Systems.* ACM Press, New York, 109–121.

CORBETT, P., FEITELSON, D., FINEBERG, S., HSU, Y., NITZBERG, B., PROST, J.-P., SNIR, M., TRAVERSAT, B., AND WONG, P. 1996a. Overview of the MPI-IO parallel I/O interface. In *Input/Output in Parallel and Distributed Computer Systems,* R. Jain, J. Werth, and J. C. Browne Eds. The Kluwer International Series in Engineering and Computer Science, vol. 362. Kluwer, Amsterdam.

CORBETT, P. F. AND FEITELSON, D. G. 1996. The Vesta parallel file system. *ACM Trans. Comput. Syst. 14,* 3 (Aug.), 225–264.

CORBETT, P. F., PROST, J.-P., DEMETRIOU, C., GIBSON, G., REIDEL, E., ZELENKA, J., CHEN, Y., FELTEN, E., LI, K., HARTMAN, J., PETERSON, L., BERSHAD, B., WOLMAN, A., AND AYDT, R. 1996b. Proposal for a common parallel file system programming interface. Dept. of Computer Science, Univ. of Arizona, Tucson, Ariz. Available via http://www.cs.arizona.edu/sio/api1.0.ps. Version 1.0.

CORMEN, T. H. AND COLVIN, A. 1994. ViC*: A preprocessor for virtual-memory C*. Tech. Rep. PCS-TR94-243, Dept. of Computer Science, Dartmouth College, Hanover, N.H.

CORMEN, T. H. AND KOTZ, D. 1993. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium.* Dartmouth College, Hanover, N.H., 64–74. Revised as Dartmouth PCS-TR93-188.

CRANDALL, P. E., AYDT, R. A., CHIEN, A. A., AND REED, D. A. 1995. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95.* IEEE Computer Society Press, Los Alamitos, Calif.

CULLER, D. E., DRUSSEAU, A., GOLDSTEIN, S. C., KRISHNAMURTHY, A., LUMETTA, S., VON EICKEN, T., AND YELICK, K. 1993. Parallel programming in Split-C. In *Proceedings of Supercomputing '93.* IEEE Computer Society Press, Los Alamitos, Calif., 262–288.

DEBENEDICTIS, E. AND DEL ROSARIO, J. M. 1992. nCUBE parallel I/O software. In *Proceedings of the 11th Annual IEEE International Phoenix Conference on Computers and Communications.* IEEE Computer Society Press, Los Alamitos, Calif., 117–124.

DEL ROSARIO, J. M., BORDAWEKAR, R., AND CHOUDHARY, A. 1993. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems.* Rice University, Houston, Tex., 56–70. Also available in *Comput. Arch. News 21,* 5 (Dec. 1993), 31–38.

DIBBLE, P. C. 1990. A parallel interleaved file system. Ph.D. thesis, Univ. of Rochester, Rochester, N.Y.

DINDA, P. AND O'HALLARON, D. 1996. Fast message assembly using compact address relations. In *Proceedings of the 1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems.* ACM Press, New York, 47–56.

FOSTER, I. AND NIEPLOCHA, J. 1996. ChemIO: High-performance I/O for computational chemistry applications. Argonne National Laboratories, Argonne, Ill. Available as http://www.mcs.anl.gov/chemio/.

GALBREATH, N., GROPP, W., AND LEVINE, D. 1993. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93.* IEEE Computer Society Press, Los Alamitos, Calif., 462–471.

GOTWALS, J., SRINIVAS, S., AND GANNON, D. 1995. pC++/streams: A library for I/O on complex distributed data structures. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM Press, New York, 11–19.

HAYASHI, K., DOI, T., HORIE, T., KOYANAGI, Y., SHIRAKI, O., IMAMURA, N., SHIMIZU, T., ISHIHATA, H., AND SHINDO, T. 1994. AP1000+: Architectural support of PUT/GET interface for parallelizing compiler. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM Press, New York, 196–207.

HIGH PERFORMANCE FORTRAN FORUM. 1993. High Performance Fortran. *Sci. Program. 2,* 1–2 (Spring–Summer), 1–170.

JAIN, R. 1991. *The Art of Computer Systems Performance Analysis.* Wiley, New York.

KARPOVICH, J. F., GRIMSHAW, A. S., AND FRENCH, J. C. 1994. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications.* ACM Press, New York, 191–204.

KLIMKOWSKI, K. AND VAN DE GEIJN, R. 1995. Anatomy of an out-of-core dense linear solver. In *Proceedings of the 1995 International Conference on Parallel Processing.* CRC Press, Boca Raton, Fla., III: 29–33.

KOTZ, D. 1994. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation.* USENIX Assoc., Berkeley, Calif. Updated as Dartmouth Tech. Rep. PCS-TR94-226.

KOTZ, D. 1995a. Expanding the potential for disk-directed I/O. In *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing.* IEEE Computer Society Press, Los Alamitos, Calif., 490–495.

KOTZ, D. 1995b. Interfaces for disk-directed I/O. Tech. Rep. PCS-TR95-270, Dept. of Computer Science, Dartmouth College, Hanover, N.H.

KOTZ, D. 1996a. Applications of parallel I/O. Tech. Rep. PCS-TR96-297, Dept. of Computer Science, Dartmouth College, Hanover, N.H.

KOTZ, D. 1996b. Introduction to multiprocessor I/O architecture. In *Input/Output in Parallel and Distributed Computer Systems,* R. Jain, J. Werth, and J. C. Browne Eds. The Kluwer International Series in Engineering and Computer Science, vol. 362. Kluwer, Amsterdam, 97–123.

KOTZ, D. 1996c. Tuning STARFISH. Tech. Rep. PCS-TR96-296, Dept. of Computer Science, Dartmouth College, Hanover, N.H.

KOTZ, D. AND ELLIS, C. S. 1993. Caching and writeback policies in parallel file systems. *J. Parallel Distrib. Comput. 17,* 1–2 (Jan.–Feb.), 140–145.

KOTZ, D. AND NIEUWEJAAR, N. 1996. Flexibility and performance of parallel file systems. In *Proceedings of the 3rd International Conference of the Austrian Center for Parallel Computation.* Lecture Notes in Computer Science, vol. 1127. Springer-Verlag, Berlin, 1–11.

KOTZ, D., TOH, S. B., AND RADHAKRISHNAN, S. 1994. A detailed simulation model of the HP97560 disk drive. Tech. Rep. PCS-TR94-220, Dept. of Computer Science, Dartmouth College, Hanover, N.H.

KRIEGER, O. AND STUMM, M. 1996. HFS: A performance-oriented flexible file system based on building-block compositions. In *Proceedings of the 4th Workshop on Input/Output in Parallel and Distributed Systems.* ACM Press, New York, 95–108.

MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. 1984. A fast file system for UNIX. *ACM Trans. Comput. Syst. 2,* 3 (Aug.), 181–197.

MILLER, E. L. AND KATZ, R. H. 1991. Input/output behavior of supercomputer applications. In *Proceedings of Supercomputing '91.* IEEE Computer Society Press, Los Alamitos, Calif., 567–576.

MOORE, J. A., HATCHER, P. J., AND QUINN, M. J. 1995. Stream*: Fast, flexible, data-parallel I/O. In *Parallel Computing: State-of-the-Art and Perspectives* (*ParCo '95*). Elsevier Science, Amsterdam.

MPI-IO COMMITTEE. 1996. MPI-IO: A parallel file I/O interface for MPI. Version 5. The MPI-IO Committee, NASA Ames, Ames, Iowa. Available as http://lovelace.nas.nasa.gov/MPI-IO/mpi-io-report.0.5.ps.

NIEUWEJAAR, N. AND KOTZ, D. 1996. Low-level interfaces for high-level parallel I/O. In *Input/Output in Parallel and Distributed Computer Systems,* R. Jain, J. Werth, and J. C. Browne Eds., The Kluwer International Series in Engineering and Computer Science, vol. 362. Kluwer, Amsterdam, 205–223.

NIEUWEJAAR, N. AND KOTZ, D. 1997. The Galley parallel file system. *Parallel Comput.* To be published.

NIEUWEJAAR, N., KOTZ, D., PURAKAYASTHA, A., ELLIS, C. S., AND BEST, M. 1996. File-access characteristics of parallel scientific workloads. *IEEE Trans. Parallel Distrib. Syst. 7,* 10 (Oct.), 1075–1089.

OUSTERHOUT, J., COSTA, H. D., HARRISON, D., KUNZE, J., KUPFER, M., AND THOMPSON, J. 1985. A trace driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles.* ACM Press, New York, 15–24.

PASQUALE, B. K. AND POLYZOS, G. C. 1993. A static analysis of I/O characteristics of scientific applications in a production workload. In *Proceedings of Supercomputing '93.* IEEE Computer Society Press, Los Alamitos, Calif., 388–397.

PASQUALE, B. K. AND POLYZOS, G. C. 1994. Dynamic I/O characterization of I/O intensive scientific applications. In *Proceedings of Supercomputing '94.* IEEE Computer Society Press, Los Alamitos, Calif., 660–669.

PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. 1995. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles.* ACM Press, New York, 79–95.

PIERCE, P. 1989. A concurrent file system for a highly parallel mass storage system. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications.* Golden Gate Enterprises, Los Altos, Calif., 155–160.

PURAKAYASTHA, A., ELLIS, C. S., AND KOTZ, D. 1996. ENWRICH: A compute-processor write caching scheme for parallel file systems. In *Proceedings of the 4th Workshop on Input/Output in Parallel and Distributed Systems.* ACM Press, New York, 55–68.

RUEMMLER, C. AND WILKES, J. 1994. An introduction to disk drive modeling. *IEEE Comput. 27,* 3 (Mar.), 17–28.

SARAWAGI, S. AND STONEBRAKER, M. 1993. Efficient organization of large multidimensional arrays. Tech. Rep. S2K-93-32, Univ. of California, Berkeley, Calif.

SEAMONS, K. E., CHEN, Y., JONES, P., JOZWIAK, J., AND WINSLETT, M. 1995. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95.* IEEE Computer Society Press, Los Alamitos, Calif.

SEAMONS, K. E. AND WINSLETT, M. 1994. An efficient abstract interface for multidimensional array I/O. In *Proceedings of Supercomputing '94.* IEEE Computer Society Press, Los Alamitos, Calif., 650–659.

SELTZER, M., CHEN, P., AND OUSTERHOUT, J. 1990. Disk scheduling revisited. In *Proceedings of the 1990 Winter USENIX Conference.* USENIX Assoc., Berkeley, Calif.

SMIRNI, E., AYDT, R. A., CHIEN, A. A., AND REED, D. A. 1996. I/O requirements of scientific applications: An evolutionary view. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing.* IEEE Computer Society Press, Los Alamitos, Calif., 49–59.

THAKUR, R., BORDAWEKAR, R., AND CHOUDHARY, A. 1994. Compiler and runtime support for out-of-core HPF programs. In *Proceedings of the 8th ACM International Conference on Supercomputing.* ACM Press, New York, 382–391.

THAKUR, R., CHOUDHARY, A., BORDAWEKAR, R., MORE, S., AND KUDITIPUDI, S. 1996. PASSION: Optimized I/O for parallel applications. *IEEE Comput. 29,* 6 (June), 70–78.

TOLEDO, S. AND GUSTAVSON, F. G. 1996. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Proceedings of the 4th Workshop on Input/Output in Parallel and Distributed Systems.* ACM Press, New York, 28–40.

WHEAT, S. R., MACCABE, A. B., RIESEN, R., VAN DRESSER, D. W., AND STALLCUP, T. M. 1994. PUMA: An operating system for massively parallel systems. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences.* IEEE Computer Society Press, Los Alamitos, Calif., 56–65.

WOMBLE, D., GREENBERG, D., WHEAT, S., AND RIESEN, R. 1993. Beyond core: Making parallel computer I/O practical. In *Proceedings of the 1993 DAGS/PC Symposium.* Dartmouth Inst. for Advanced Graduate Studies, Hanover, N.H., 56–63.

Received January 1995; revised February 1996; accepted October 1996