# Disk-directed I/O for MIMD Multiprocessors

David Kotz

Department of Computer Science
Dartmouth College
Hanover, NH 03755
dfk@cs.dartmouth.edu

Dartmouth PCS-TR94-226

July 22, 1994
**Revised November 8, 1994**

**Abstract**

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. Indeed, disk-directed I/O provided consistent high performance that was largely independent of data distribution, obtained up to 93% of peak disk bandwidth, and was as much as 18 times faster than traditional parallel file systems.

## 1 Introduction

Scientific applications like weather forecasting, aircraft simulation, seismic exploration, and climate modeling are increasingly being implemented on massively parallel supercomputers. Applications like these have intense I/O demands, as well as massive computational requirements. Recent multiprocessors have provided high-performance I/O hardware, 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 yet to be built.

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

This research was funded by Dartmouth College. A condensed form of this text appears in the First Symposium on Operating Systems Design and Implementation (OSDI), November 1994. This revised technical report updates a few of the traditional-caching numbers from the OSDI paper (see page 13), but makes no qualitative changes.

is often poor. Poor performance is not surprising because the Unix file system was designed for a general-purpose workload [OCH+85], rather than for a parallel, scientific workload. Scientific applications use larger files and have more sequential access [MK91, GGL93, PP93]. *Parallel* scientific programs access the file with patterns not seen in uniprocessor or distributed-system workloads, in particular, complex strided access to discontiguous pieces of the file [KN94, NK94]. Finally, scientific applications use files for more than loading raw data and storing results; files are used as scratch space for very large problems as application-controlled virtual memory [CK93]. In short, multiprocessors need new file systems that are designed for parallel scientific applications.

In this paper we describe a technique 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 do file-system processing, and Compute Processors (CPs), which do mostly application processing. Figure 1 shows such an architecture. The IBM SP-2, Intel iPSC, Intel Paragon, KSR/2, Meiko CS-2, nCUBE/2, and Thinking Machines CM-5 all use this model; the CS-2 and the SP-2 allow IOPs to double as CPs. Furthermore, our technique is best suited to applications written in a single-program-multiple-data (SPMD) or data-parallel programming model. With our technique, *disk-directed I/O*, CPs collectively send a single request to all IOPs, which then arrange the flow of data to optimize disk, buffer, and network resources.

We begin by advocating a "collective-I/O" interface for parallel file systems. 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 contrast our system to related work in Section 7, and summarize our conclusions in Section 8.

## 2 Collective I/O

Consider programs that distribute large matrices across the processor memories, and the task of loading such a matrix from a file.[1] From the point of view of a traditional file system, each processor independently requests its portion of the data, by reading from the file into its local memory. If that processor's data is not logically contiguous in the file, as is often the case [KN94], a separate file-system call is needed for each contiguous chunk of the file. The file system is thus faced with

[1]This scenario arises in many situations. The file may contain raw input data or may be a scratch file written in a previous phase of the application. The matrix may be the whole data set, or may be a partition of a larger data set, for example, a 2-d slice of a 3-d matrix. Furthermore, the operation may be synchronous, with the application waiting for I/O to complete, or asynchronous, perhaps as the result of a compiler-instigated prefetch request.

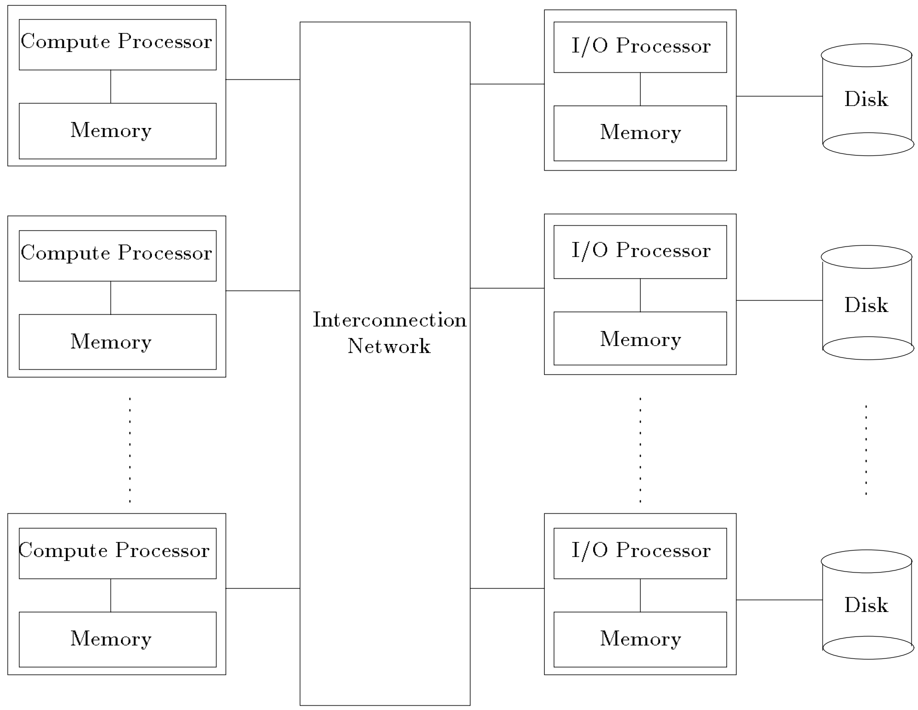


Figure 1: Parallel Independent Disks (PID) in an MIMD multiprocessor, with separate compute processors (CP) and I/O processors (IOP).

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 [CF94, FPD93, Pie89, Roy93, DdR92, LIN$^+$93, BGST93, Dib90, DSE88] decluster file data across many disks, each application request may be broken into even smaller requests that are sent to different IOPs. It is difficult for the file system, which is distributed across many I/O processors, to recognize these requests as a single coordinated request, and to use that information to optimize the I/O. Valuable semantic information — that a large, contiguous, parallel file transfer is in progress — is lost through this low-level interface. A *collective-I/O interface*, in which all CPs cooperate to make a single, large request, retains this semantic information, making it easier to coordinate I/O for better performance [dBC93, Nit92, PGK88].

Collective I/O need not involve matrices. Many out-of-core parallel algorithms do I/O in "memoryloads," that is, they repeatedly load some subset of the file into memory, process it, and write it out [CK93]. Each transfer is a large, but not necessarily contiguous, set of data. Traditional caching and prefetching policies, geared for sequential access, would be ineffective or even detrimental for this type of I/O.

Unfortunately, few multiprocessor file systems provide a collective interface. Most have an interface based on simple parallel extensions to the traditional read/write/seek model, focusing on coordination of the file pointer. Vesta [CF94] and the nCUBE file system [DdR92] support logical mappings between the file and processor memories, defining separate "subfiles" for each processor. Although these mappings remove the burden of managing the file pointer from the programmer, and allow the programmer to request noncontiguous data in a single request, there is no support for collective I/O. CM-Fortran for the CM-5 does provide a collective-I/O interface, which leads to high performance through cooperation among the compiler, run-time, operating system, and hardware. ELFS [KGF94] provides an object-oriented interface that encourages operations on large objects, and could lead to support for collective I/O. Finally, there are several interfaces for collective matrix I/O [GGL93, BdC93, BBS+94]. For example, to read a two-dimensional matrix of integers in the notation of [GGL93], every processor executes the following code:

```
/* describes my part of matrix */
PIFArrayPart mypart[2] =  ... ;
/* memory for my part */
int *A = malloc(...);
PIFILE *fp = PIFOpen(...);
PIFReadDistributedArray(fp, NULL, sizeof(int), mypart, 2, A, MSG_INT);
```

Thus, the groundwork for collective I/O exists. The challenge is to provide mechanisms that use the semantic-information content of collective operations to improve performance.

## 3 Collective-I/O implementation alternatives

In this paper we consider collective-read and -write operations that transfer a large matrix between CP memories and a file that 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 is contiguous in memory. We discuss three implementation alternatives: traditional caching, two-phase I/O, and disk-directed I/O. The latter two require a collective-I/O interface similar to that of Galbreath et al [GGL93], above.

**Traditional caching.** This alternative mimics a "traditional" parallel file system like Intel CFS [Pie89], with no explicit collective-I/O interface and with IOPs that each manage a file cache. Figure 2a 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. Recall that each application process must call ReadCP once for each contiguous chunk of the file, no matter how small. Each IOP attempts to dynamically optimize the use of the disk, cache, and network interface.

**Two-phase I/O.** Figure 2b sketches an alternative proposed by del Rosario, Bordawekar, and Choudhary [dBC93, BdC93], 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 requests to the IOPs "conform" to the layout of the file, that is, the requests are for large contiguous chunks.

**Disk-directed I/O.** We go further by having the CPs pass the collective request on to the IOPs, which then arrange the data transfer as shown in Figure 2c. This *disk-directed* model, which essentially puts the disks (IOPs) in control of the order and timing of the flow of data, has several potential performance advantages:

- The I/O can conform not only to the logical layout of the file, as in two-phase I/O, but to the physical layout on disk.
- The disk-I/O phase is integrated with the permutation phase.
- There is only one I/O request to each IOP; subsequent communication uses only low-overhead data-transfer messages.
- Disk scheduling is improved, possibly across megabytes of data: in Figure 2c, the IOPs presort the block list for each disk.
- 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 or memory-memory copying is needed at the CPs for buffering, message-passing, or permuting data.
- There is no communication among the IOPs and none, other than barriers, among the CPs. The cost of these barriers is negligible compared to the time needed for a large file transfer.

**a) Traditional caching**

```
ReadCP(file, read parameters, destination address):
    for each file block needed to satisfy request
        compute which disk holds that file block
        if our previous request to that disk is still 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)
    for each contiguous chunk of the file this processor should read
        ReadCP(file, one chunk)
    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):
    arrange for incoming data to be stored at destination address
    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 finished
    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
```

Figure 1: Pseudo-code for collective-read implementations. Collective writes are similar.

## 4 Evaluation

We implemented both a traditional-caching system and a disk-directed-I/O system on a simulated MIMD multiprocessor (see below). We did not implement two-phase I/O because, as we discuss in Section 7.1, disk-directed I/O obtains all the benefits of two-phase I/O, and more. In this section, we describe our simulated implementation.

Files were striped across all disks, block by block. Each IOP served one or more disks, using one I/O bus. Each disk had a thread permanently running on its IOP, that controlled access to the disk. The disk thread corresponded 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 interrupt handler on the receiving processor could quickly decide where to deposit the contents of the message, using DMA. For requests to the IOP, it created a new thread and deposited the message in the thread's stack. 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 addition to the request/reply messages, the IOP could use "Memget" and "Memput" messages to read and write the user's buffer on the CPs. Every CP provided a base address to its message-passing system, so that the IOPs only referred to offsets within each CP. Memput messages contained data, and returned only an acknowledgement. Memget messages contained a reply-action address, and returned a reply containing the requested data.

**Disk-directed I/O.** Each IOP received one request, creating one new thread. The new thread computed the list of disk blocks involved, sorted the list by location, and informed the relevant disk threads. It then allocated two one-block buffers[2] for each local disk, and created a thread to manage each buffer. While not absolutely necessary, the threads simplified programming the concurrent activities. These buffer threads repeatedly transferred blocks, letting the disk thread

[2] Larger buffers could be used, but with today's track-buffering disk devices, they are not particularly helpful.

choose which block to transfer next. When reading, they used Memput messages to move data from the IOP memory to the CP memories. When writing, they sent Memget messages to the CPs, causing them to reply with a message containing the requested data. When possible the thread sent concurrent Memget or Memput messages to many CPs.

**Traditional caching.** Our code followed the pseudo-code of Figure 2a. CPs did not cache or prefetch data, so all requests involved communication with the IOP. The CP sent concurrent requests to all the relevant IOPs, with up to one outstanding request per disk per CP. This limit was a compromise between maximizing concurrency and the need to limit the potential load on each IOP.[3]

Each IOP managed a cache that was large enough to double-buffer an independent stream of requests from each CP to each disk.[4] The cache used an LRU-replacement strategy, prefetched one block ahead after each read request, and flushed dirty buffers to disk when they were full (i.e., after $n$ bytes had been written to an $n$-byte buffer [KE93]).

As described above, we transferred data as a part of request and reply messages, using DMA to avoid most extraneous copies. At the IOP incoming write requests, containing the data to write, were stored in the new thread's buffer until the thread determined where in the cache to put the data. Later, the thread copied the data into a cache buffer, the only memory-memory copy we used.[5]

While our cache implementation does not model any specific commercial cache implementation, we believe it is reasonable and better than most, and thus a fair competitor for our disk-directed-I/O implementation.

## 4.1 Simulator

The implementations described above ran on top of the Proteus parallel-architecture simulator [BDCW91], which in turn ran on a DEC-5000 workstation. We configured Proteus using the parameters listed in Table 1. These parameters are not meant to reflect any particular machine, but a generic machine of current technology.

[3] More aggressive strategies would require either more buffer space or the addition of dynamic flow control, without a substantial improvement in parallelism.

[4] While two cache buffers per disk *per CP* is not scalable, it is reasonable in most situations (e.g., only 16 MB per IOP for 2 local disks, 512 CPs, and an 8 KB block size). Note that this is much more than the space needed for disk-directed I/O, two buffers per disk.

[5] We chose this design because it was similar to traditional systems. In any case, we believe that avoiding the memory-memory copy by using Memgets and dataless request messages would be unlikely to justify the extra round-trip message traffic, particularly for small writes.

Table 1: Parameters for simulator. Those marked with a * were varied in some experiments.

| | |
|---|---|
| MIMD, distributed-memory | 32 processors |
| Compute processors (CPs) | 16 * |
| I/O processors (IOPs) | 16 * |
| CPU speed, type | 50 MHz, RISC |
| Disks | 16 * |
| Disk type | HP 97560 |
| Disk capacity | 1.3 GB |
| Disk peak transfer rate | 2.34 Mbytes/s |
| File-system block size | 8 KB |
| I/O buses (one per IOP) | 16 * |
| I/O bus type | SCSI |
| I/O bus peak bandwidth | 10 Mbytes/s |
| Interconnect topology | $6 \times 6$ torus |
| Interconnect bandwidth | $200 \times 10^6$ bytes/s bidirectional |
| Interconnect latency | 20 ns per router |
| Routing | wormhole |

Proteus itself has been validated against real message-passing machines [BDCW91]. 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.[6] We compared the effect of this choice on a subset of our experiments, some with thousands of very short messages, and some with many large messages, and found that the results of each experiment using the modeled network 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 modeled network.

We added a disk model, a reimplementation of Ruemmler and Wilkes' HP 97560 model [RW94, KTR94]. 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.

[6] Both methods assume that each processor has a deep hardware FIFO for incoming messages. To reduce the effect of this assumption, we added flow control to limit our use of this FIFO.

## 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 three alternatives: traditional caching, disk-directed, or disk-directed with block-list presort (defined in Figure 2c). We repeated this experiment 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 and in the network. To be fair, 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 10 MB (1280 8-KB blocks). While 10 MB is not a large file, preliminary tests showed qualitatively similar results with 100 and 1000 MB files (see page 23). Thus, 10 MB 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.[7] A real file system would be somewhere between the two. As a 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 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 [HPF93, dBC93], 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

[7] We chose five random layouts, one for each trial, and used the same set of five layouts for all *random-blocks* experiments. Of course, there was only one *contiguous* layout, used in all trials.

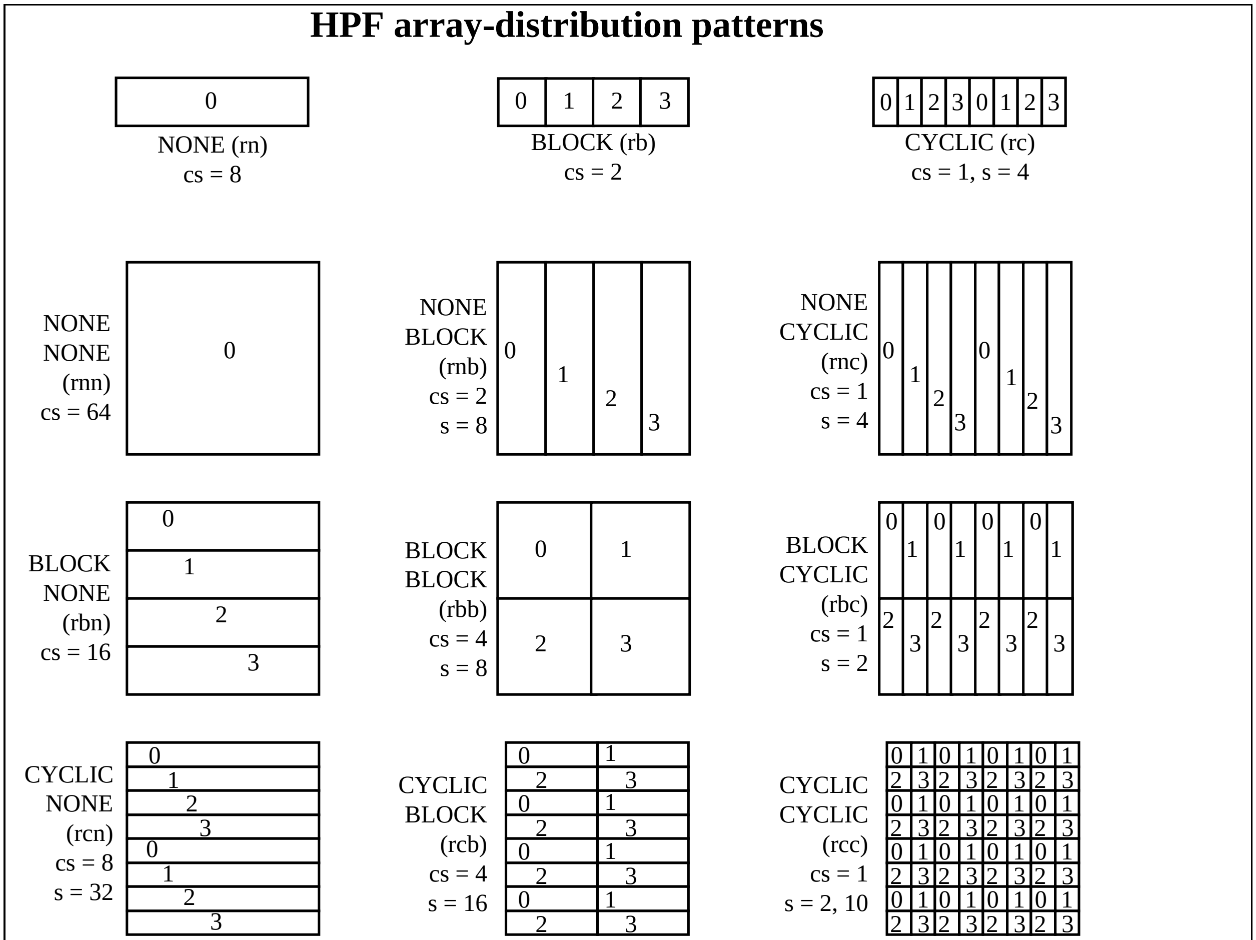


Figure 3: Examples of matrix distributions, which we used as file-access patterns in our experiments. These examples represent common ways to distribute a 1x8 vector or an 8x8 matrix over four processors. Patterns are named by the distribution method (NONE, BLOCK, or CYCLIC) in each dimension (rows first, 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 2.

beginning with `r` for reading and `w` for writing; 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. Table 2 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

Table 2: Summary of file-access patterns (smaller examples of these patterns are shown in Figure 3). We list only the read patterns here. All numbers are for a 10 MB 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 1 K record is one block).

| Pattern name | Row distribution | Column distribution | Record size (bytes) | Rows | Cols | Chunk size (records) | Stride (records) | Same as |
|---|---|---|---|---|---|---|---|---|
| ra | ALL | - | - | - | - | 1280 blocks | - | |
| rn | NONE | - | - | - | - | 1280 blocks | - | |
| rb | BLOCK | - | 8 | 1310720 | - | 80 K | - | |
| rc | CYCLIC | - | 8 | 1310720 | - | 1 | 16 | |
| rnn | NONE | NONE | 8 | 1280 | 1024 | 1280 K | - | rn |
| rnb | NONE | BLOCK | 8 | 1280 | 1024 | 64 | 1 K | |
| rnc | NONE | CYCLIC | 8 | 1280 | 1024 | 1 | 16 | rc |
| rbn | BLOCK | NONE | 8 | 1280 | 1024 | 80 K | - | rb |
| rbb | BLOCK | BLOCK | 8 | 1280 | 1024 | 256 | 1 K | |
| rbc | BLOCK | CYCLIC | 8 | 1280 | 1024 | 1 | 4 | |
| rcn | CYCLIC | NONE | 8 | 1280 | 1024 | 1 K | 16 K | |
| rcb | CYCLIC | BLOCK | 8 | 1280 | 1024 | 256 | 4 K | |
| 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 | - | rn |
| rnb | NONE | BLOCK | 8192 | 40 | 32 | 2 | 32 | |
| rnc | NONE | CYCLIC | 8192 | 40 | 32 | 1 | 16 | rc |
| rbn | BLOCK | NONE | 8192 | 40 | 32 | 80 | - | rb |
| 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 | |

to work in the most-convenient unit, with little interprocess locality or contention. The small record size was 8 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 [KN94]. We also tried 1024-byte and 4096-byte records (Figure 12), leading to results between the 8-byte and 8192-byte results; we present only the extremes here.

# 6 Results

*A note on the results: the numbers have been updated since the earlier version of this TR and since the OSDI paper. Earlier, the traditional-caching code did not include some obvious optimizations, leading to an unfair comparison. In this revision we update the traditional-caching numbers to incorporate the optimizations; while many cases did not change, a few cases are substantially faster than before. Thus, these numbers represent a better comparison between disk-directed I/O and traditional caching. Overall, however, there is no qualitative difference or change in the conclusions.*

Figures 4 and 5 show the performance of our disk-directed-I/O approach and of the traditional-caching method.[8] Each figure has two graphs, one for 8-byte records and one for 8192-byte records. Disk-directed I/O was usually at least as fast as traditional caching, and in one case was *18 times* faster.[9]

Figure 4 and Table 3 display the performance on a random-blocks disk layout. Three cases are shown for each access pattern: traditional caching (TC), and disk-directed I/O (DDIO) with and without a presort of the block requests by physical location. Throughput for disk-directed I/O with presorting consistently reached 6.2 Mbytes/s for reading and 7.4–7.5 Mbytes/s for writing. In contrast, traditional-caching throughput was highly dependent on the access pattern, was never faster than 5 Mbytes/s, and was particularly slow for many 8-byte patterns. Cases with small chunk sizes were the slowest, as slow as 0.7 Mbytes/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 10.6 times faster than traditional caching.

Figure 4 and Table 3 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 traditional caching or, for that matter, in two-phase I/O. Nonetheless, disk-directed I/O without presorting was still faster than traditional caching in most cases. At best, it was 7.1 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.

[8] Because the `ra` pattern broadcasts the same 10 MB data to all 16 CPs, its apparent throughput was inflated. We have normalized it in our graphs by dividing by the number of CPs.

[9] In the worst case where disk-directed I/O was slower than traditional caching by a statistically significant amount, disk-directed I/O was slower by 1.1%.

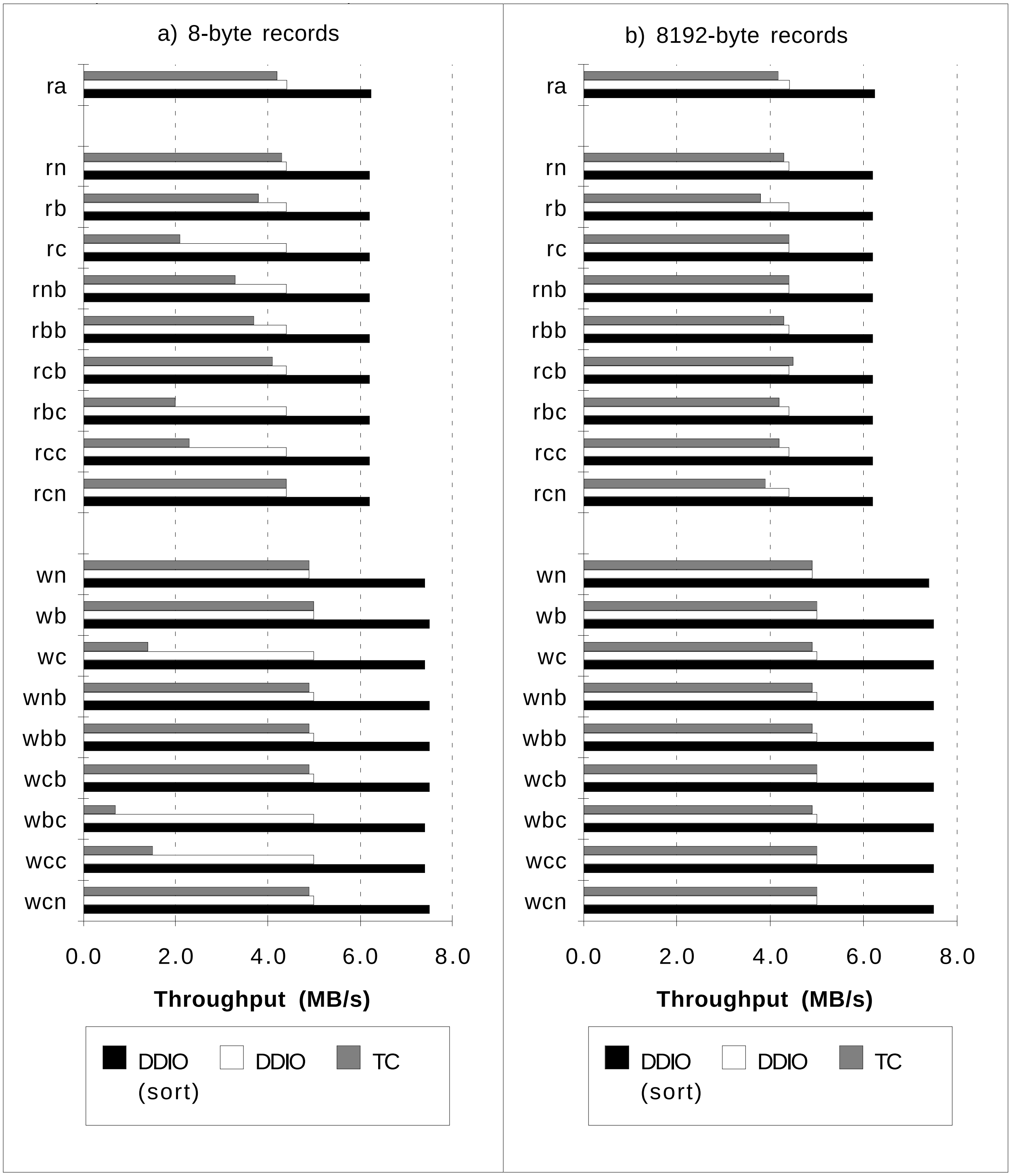


Figure 4: Two graphs comparing the throughput of disk-directed I/O (DDIO) to that of traditional caching (TC), on a **random-blocks** disk layout. `ra` throughput has been normalized by the number of CPs. Each point represents the average of five trials of an access pattern on both methods (maximum coefficient of variation ($cv$) is 0.11). These data are also presented in Table 3.

Table 3: Throughput in MB/s for traditional caching (TC) and disk-directed I/O ("DD" for disk-directed I/O without presort, and "DDs" for disk-directed I/O with presort), on all patterns, for both record sizes, for the **random-blocks layout**, averaged over five trials. Along with each pair is the average of the throughput ratios ($r$); those in italics do not represent a difference that is statistically significant at the 95% confidence level. Disk-directed I/O was never substantially slower than traditional caching. Pattern `ra` is not scaled as it is in the graphs. Patterns `ra`, `rn`, and `wn` are independent of record size, and are listed in the 8192 column. These data are graphed in Figure 4.

| | Random-blocks layout | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 8-byte records | | | | | 8192-byte records | | | | |
| Pattern | TC | DD | $r$ | DDs | $r$ | TC | DD | $r$ | DDs | $r$ |
| `ra` | - | - | - | - | - | 66.7 | 70.7 | 1.1 | 99.8 | 1.5 |
| `rn` | - | - | - | - | - | 4.3 | 4.4 | 1.0 | 6.2 | 1.4 |
| `rb` | 3.8 | 4.4 | 1.2 | 6.2 | 1.7 | 3.8 | 4.4 | 1.2 | 6.2 | 1.7 |
| `rc` | 2.1 | 4.4 | 2.1 | 6.2 | 2.9 | 4.4 | 4.4 | *1.0* | 6.2 | 1.4 |
| `rnb` | 3.3 | 4.4 | 1.3 | 6.2 | 1.9 | 4.4 | 4.4 | *1.0* | 6.2 | 1.4 |
| `rbb` | 3.7 | 4.4 | 1.2 | 6.2 | 1.7 | 4.3 | 4.4 | 1.0 | 6.2 | 1.5 |
| `rcb` | 4.1 | 4.4 | 1.1 | 6.2 | 1.5 | 4.5 | 4.4 | *1.0* | 6.2 | 1.4 |
| `rbc` | 2.0 | 4.4 | 2.2 | 6.2 | 3.1 | 4.2 | 4.4 | 1.1 | 6.2 | 1.5 |
| `rcc` | 2.3 | 4.4 | 1.9 | 6.2 | 2.6 | 4.2 | 4.4 | 1.0 | 6.2 | 1.5 |
| `rcn` | 4.4 | 4.4 | *1.0* | 6.2 | 1.4 | 3.9 | 4.4 | 1.1 | 6.2 | 1.6 |
| `wn` | - | - | - | - | - | 4.9 | 4.9 | *1.0* | 7.4 | 1.5 |
| `wb` | 5.0 | 5.0 | *1.0* | 7.5 | 1.5 | 5.0 | 5.0 | *1.0* | 7.5 | 1.5 |
| `wc` | 1.4 | 5.0 | 3.7 | 7.4 | 5.4 | 4.9 | 5.0 | 1.0 | 7.5 | 1.5 |
| `wnb` | 4.9 | 5.0 | 1.0 | 7.5 | 1.5 | 4.9 | 5.0 | *1.0* | 7.5 | 1.5 |
| `wbb` | 4.9 | 5.0 | *1.0* | 7.5 | 1.5 | 4.9 | 5.0 | *1.0* | 7.5 | 1.5 |
| `wcb` | 4.9 | 5.0 | 1.0 | 7.5 | 1.5 | 5.0 | 5.0 | *1.0* | 7.5 | 1.5 |
| `wbc` | 0.7 | 5.0 | 7.1 | 7.4 | 10.6 | 4.9 | 5.0 | *1.0* | 7.5 | 1.5 |
| `wcc` | 1.5 | 5.0 | 3.3 | 7.4 | 4.9 | 5.0 | 5.0 | *1.0* | 7.5 | 1.5 |
| `wcn` | 4.9 | 5.0 | 1.0 | 7.5 | 1.5 | 5.0 | 5.0 | *1.0* | 7.5 | 1.5 |

Maximum coefficient of variation on average of ratios was 0.11.

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 (Figure 5 and Table 4). 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 caches when using the contiguous layout. In most cases disk-directed reading moved about 32.8 Mbytes/s, and disk-directed writing moved 34.8 Mbytes/s, which was an impressive 93% of the disks' peak transfer rate of 37.5 Mbytes/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. Further tuning of the disk-directed-I/O code may alleviate this problem, but the real solution would be to use gather/scatter Memput and Memget operations.

Traditional caching was rarely able to obtain the full disk bandwidth, and had particular trouble with the 8-byte patterns. Although there were cases where traditional caching could match disk-directed I/O, traditional caching was as much as 18.2 times slower than disk-directed I/O. Traditional caching failed in a few critical ways:

- When the CPs were active at widely different locations in the file (e.g., in `rb` or `rcn`), there was little interprocess spatial locality. In the contiguous layout, the multiple localities defeated the disk's internal caching and caused extra head movement, both a significant performance loss. Furthermore, the lost locality hampered the performance of IOP caching and prefetching, causing extraneous disk I/O.

- In some patterns, IOP-prefetching mistakes caused extraneous disk reads. At the end of the `rb` pattern, for example, one extra block is prefetched for each CP on each disk; these extra blocks are negligible in large files (see page 23), but account for most of traditional caching's poor performance on `rb` in Figure 4. `rcn` with 8 KB records and `rbb` had similar problems.

- When the CPs were using 8-byte CYCLIC patterns, many IOP-request messages were necessary to transfer the small non-contiguous records, requiring many (expensive) IOP-cache accesses. In addition, the success of interprocess spatial locality was crucial for performance.

- The high data rates of the contiguous disk layout expose the cache-management overhead in traditional caching, unable to match disk-directed I/O's performance except for `wn`.

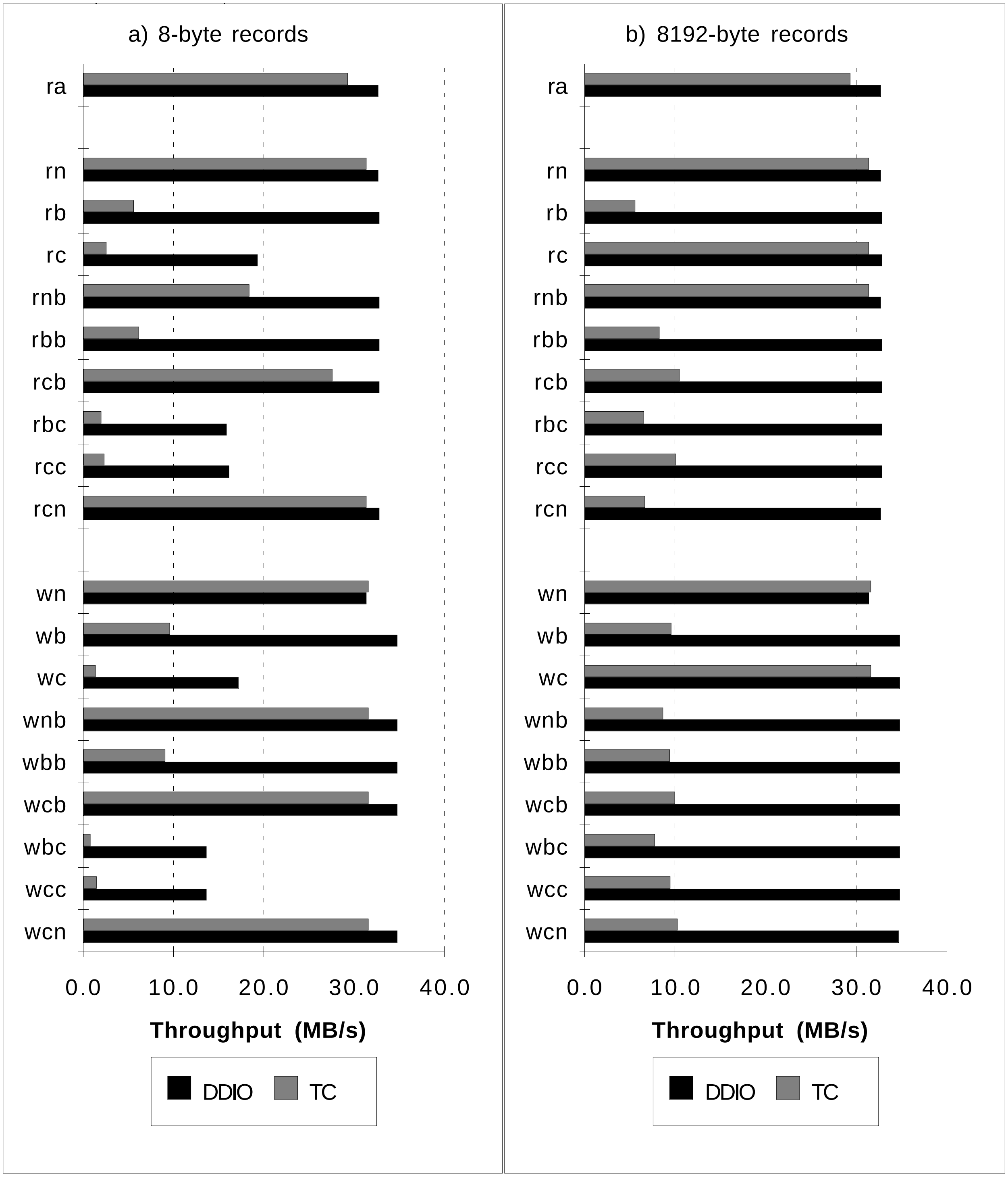


Figure 5: Two graphs comparing the throughput of disk-directed I/O (DDIO) and traditional caching (TC), on a **contiguous** disk layout. `ra` throughput has been normalized by the number of CPs. Each point represents the average of five trials of an access pattern on both methods (maximum *cv* is 0.12). Note that the peak disk throughput was 37.5 Mbytes/s. These data are also presented in Table 4.

Table 4: Throughput in MB/s for traditional caching (TC) and disk-directed I/O (DD), on all patterns, for both record sizes, for the **contiguous layout**, averaged over five trials. Along with each pair is the average of the throughput ratios ($r$); all are statistically significant at the 95% confidence level. Disk-directed I/O was in all cases faster than traditional caching. Pattern `ra` is not scaled as it is in the graphs. Patterns `ra`, `rn`, and `wn` are independent of record size, and are listed in the 8192 column. These data are graphed in Figure 5.

| | Contiguous layout | | | | | |
|---|---|---|---|---|---|---|
| | 8-byte records | | | 8192-byte records | | |
| Pattern | TC | DD | $r$ | TC | DD | $r$ |
| `ra` | - | - | - | 469.1 | 522.9 | 1.1 |
| `rn` | - | - | - | 31.4 | 32.7 | 1.0 |
| `rb` | 5.6 | 32.8 | 5.8 | 5.6 | 32.8 | 5.8 |
| `rc` | 2.6 | 19.3 | 7.4 | 31.4 | 32.8 | 1.0 |
| `rnb` | 18.4 | 32.8 | 1.8 | 31.4 | 32.7 | 1.0 |
| `rbb` | 6.2 | 32.8 | 5.3 | 8.3 | 32.8 | 4.0 |
| `rcb` | 27.6 | 32.8 | 1.2 | 10.5 | 32.8 | 3.1 |
| `rbc` | 2.0 | 15.9 | 7.8 | 6.6 | 32.8 | 4.9 |
| `rcc` | 2.4 | 16.2 | 6.8 | 10.1 | 32.8 | 3.3 |
| `rcn` | 31.4 | 32.8 | 1.0 | 6.7 | 32.7 | 4.9 |
| `wn` | - | - | - | 31.6 | 31.4 | 1.0 |
| `wb` | 9.6 | 34.8 | 3.6 | 9.6 | 34.8 | 3.6 |
| `wc` | 1.4 | 17.2 | 12.3 | 31.6 | 34.8 | 1.1 |
| `wnb` | 31.6 | 34.8 | 1.1 | 8.7 | 34.8 | 4.0 |
| `wbb` | 9.1 | 34.8 | 3.8 | 9.4 | 34.8 | 3.7 |
| `wcb` | 31.6 | 34.8 | 1.1 | 10.0 | 34.8 | 3.5 |
| `wbc` | 0.8 | 13.7 | 18.2 | 7.8 | 34.8 | 4.4 |
| `wcc` | 1.5 | 13.7 | 9.1 | 9.5 | 34.8 | 3.7 |
| `wcn` | 31.6 | 34.8 | 1.1 | 10.3 | 34.7 | 3.4 |

Maximum coefficient of variation on average of ratios was 0.12.

## 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, and number of disks. 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 8 KB records. `ra` throughput was normalized as usual.

We first varied the number of CPs (Figure 6), holding the number of IOPs and disks fixed, and maintaining the cache size for traditional caching at two buffers per disk *per CP*. Note that disk-directed I/O was unaffected. Multiple localities hurt `rb` as before, but the most interesting effect was the poor performance of traditional caching on the `rc` pattern. With 1-block records and no buffers at the CP, each CP request can only use one disk. With fewer CPs than IOPs, the full disk parallelism was not used. Finally, cache-management overhead, which grew with cache size and contention by multiple CPs, reduced the performance of traditional caching on all four patterns.

We then varied the number of IOPs (and SCSI busses), holding the number of CPs, number of disks, and total cache size fixed (Figure 7). 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 10 MB/s bus bandwidth. As always, traditional caching had difficulty with the `rb` pattern. Cache-management overhead contributed to traditional caching's inability to match disk-directed I/O.

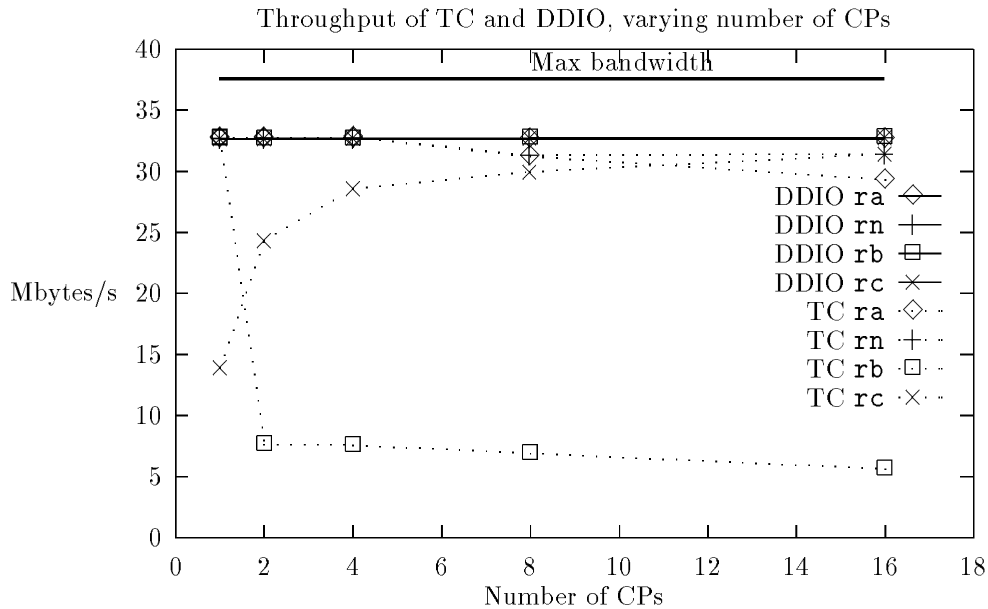


Figure 6: A comparison of the throughput of disk-directed I/O (DDIO) and traditional caching (TC), as the **number of CPs varied**, for the `ra`, `rn`, `rb`, and `rc` patterns (`ra` throughput has been normalized by the number of CPs). All cases used the contiguous disk layout, and all used 8 KB records.

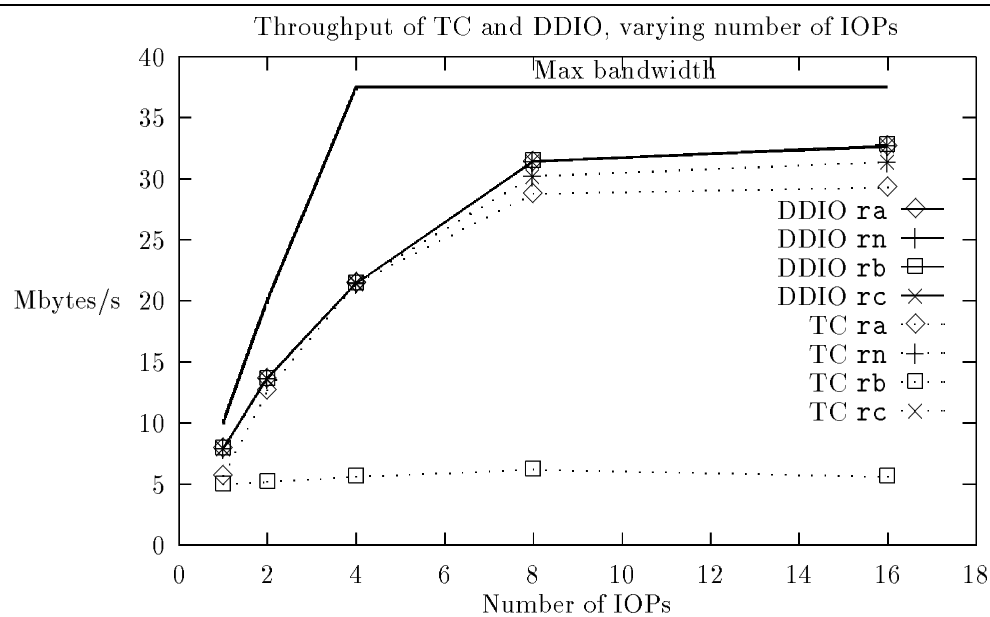


Figure 7: A comparison of the throughput of disk-directed I/O (DDIO) and traditional caching (TC), as the **number of IOPs (and busses) varied**, for the `ra`, `rn`, `rb`, and `rc` patterns (`ra` throughput has been normalized by the number of CPs). All cases used the contiguous disk layout, and all used 8 KB records. The maximum bandwidth was determined by either the busses (1–2 IOPs) or the disks (4–16 IOPs).

We then varied the number of disks, using one IOP, holding the number of CPs at 16, and maintaining the traditional-caching cache size at two buffers per CP *per disk* (Figures 8 and 9). Performance scaled with more disks, approaching the 10 MB/s bus-speed limit. The relationship between disk-directed I/O and traditional caching was determined by a combination of factors: disk-directed I/O's lower overhead and better use of the disks, and traditional caching's better use of the bus (sometimes the "synchronous" nature of disk-directed I/O caused bus congestion on the contiguous layout).

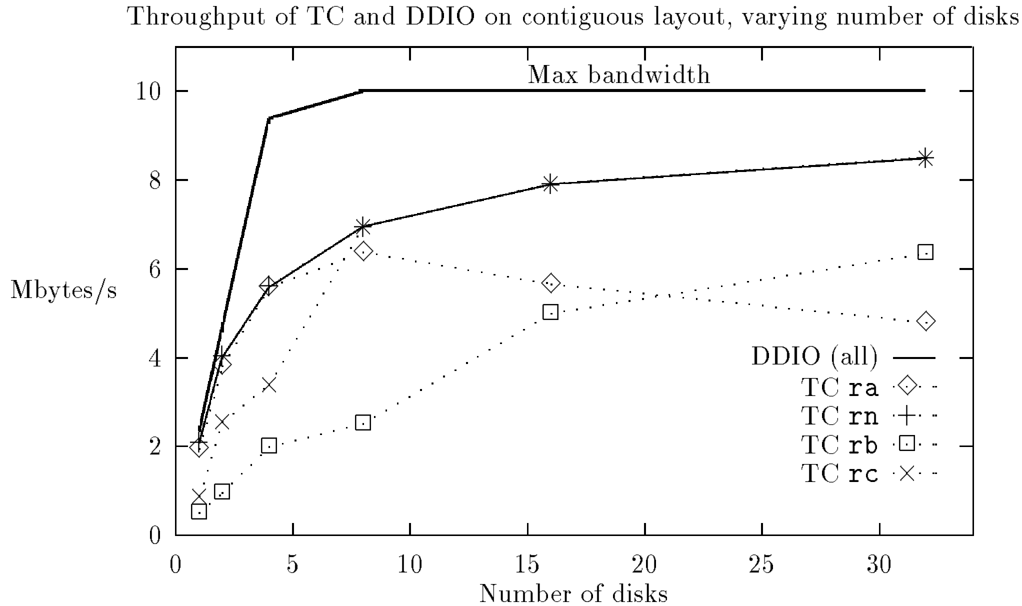


Figure 8: A comparison of the throughput of disk-directed I/O (DDIO) and traditional caching (TC), as the **number of disks varied**, for the `ra`, `rn`, `rb`, and `rc` patterns (`ra` throughput has been normalized by the number of CPs). All cases used the contiguous disk layout, and all used 8 KB records. All the DDIO cases fell on the same line, so only one curve is plotted. The maximum bandwidth was determined either by the disks (1–4 disks) or by the (single) bus (8–32 disks).

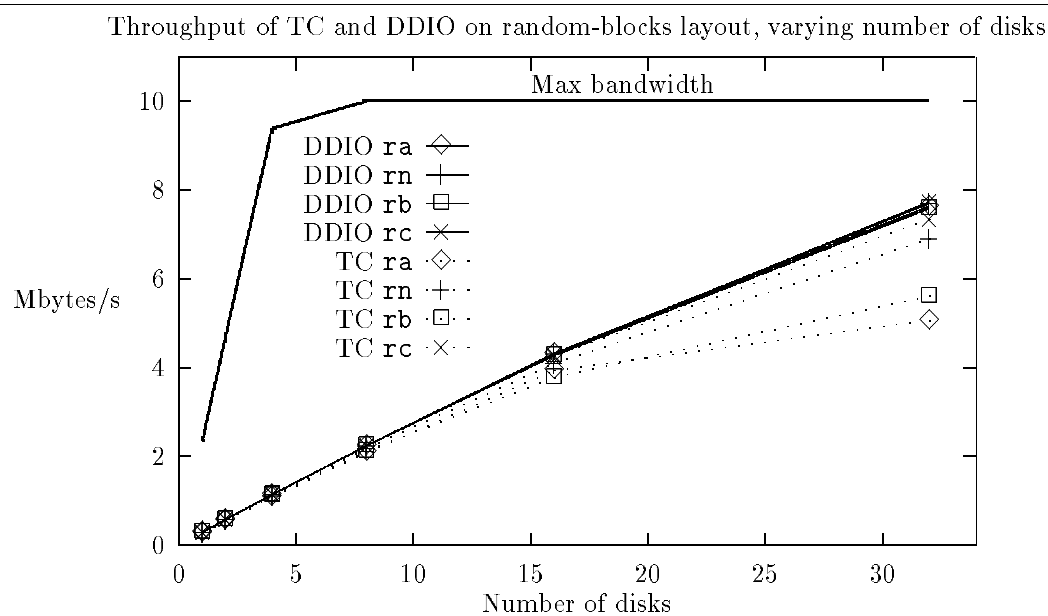


Figure 9: Similar to Figure 8, but here all cases used the random-blocks disk layout.

In most of this paper we simulate 10-MB files. To examine the effect of this choice, Figures 10 and 11 compare throughputs for files 10 and 100 times larger. Though the maximum throughputs were reached with files 100 MB or larger, we chose 10 MB for simulation efficiency. The *relative order* of test cases remained the same, with one exception: `rb` had much lower throughput on 10 MB files than on 100 MB files. This was due to the relatively large cost of prefetching mistakes committed at the end of the pattern, since their number was independent of the file size.

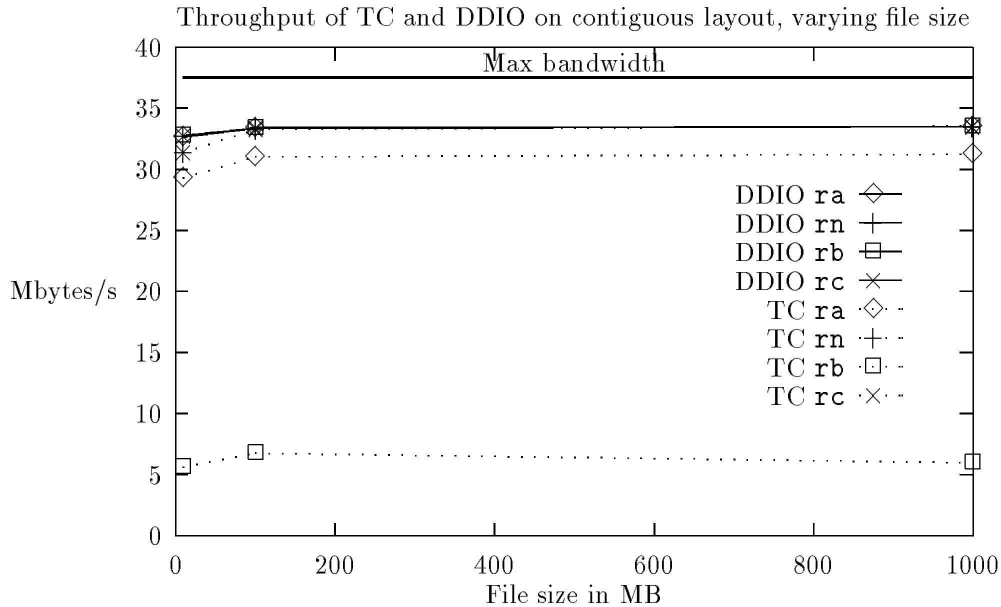


Figure 10: A comparison of the throughput of disk-directed I/O (DDIO) and traditional caching (TC), as the **file size varied**, for the `ra`, `rn`, `rb`, and `rc` patterns (`ra` throughput has been normalized by the number of CPs). All cases used the contiguous disk layout, and all used 8 KB records.

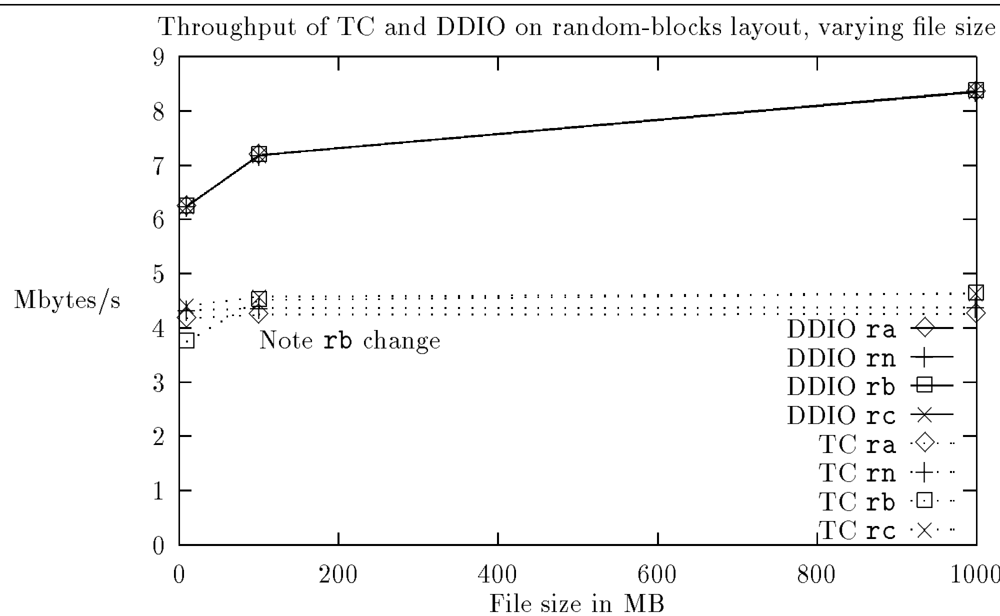


Figure 11: A comparison of the throughput of disk-directed I/O (DDIO) and traditional caching (TC), as the **file size varied**, for the `ra`, `rn`, `rb`, and `rc` patterns (`ra` throughput has been normalized by the number of CPs). All cases used the random-blocks disk layout, and all used 8 KB records. Here, disk-directed I/O includes a presort; similar conclusions were obtained without the presort.

In this paper we focus on 8- and 8192-byte record sizes. Figure 12 shows the effect of other record sizes in situations where the record size was expected to make the most difference: traditional caching 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).

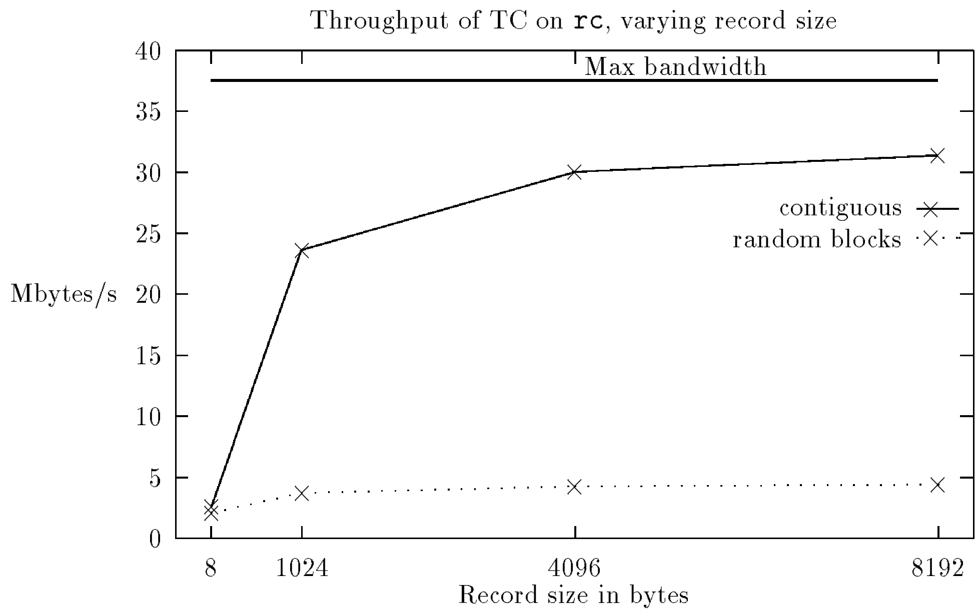


Figure 12: The throughput of traditional caching on `rc` patterns of various record sizes, for both the contiguous and random-blocks layouts.

**Summary.** These variation experiments showed that while the relative benefit of disk-directed I/O over traditional caching varied, disk-directed I/O consistently provided excellent performance, at least as good as traditional caching, often independent of access pattern, and often close to hardware limits.

# 7 Related work

Disk-directed I/O is somewhat reminiscent of the PIFS (Bridge) "tools" interface [Dib90], in that the data flow is controlled by the file system rather by than 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. In the Tandem NonStop system [EGKS90] each query is sent to all IOPs, which scan the local database partition and send only the relevant tuples back to the requesting node. The Super Database Computer [KHH+92] has disk controllers that continuously produce *tasks* from the input data set, which are consumed and processed by CPs as they become available. While this concept is roughly similar to our disk-directed I/O, it is primarily a speed-matching buffer used for load balancing.

The Jovian collective-I/O library [BBS+94] 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.

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 [CLVW93].

Finally, our Memput and Memget operations are not unusual. Similar remote-memory-access mechanisms are supported in a variety of distributed-memory systems [WMR+94, CDG+93].

## 7.1 Comparison to Two-phase I/O

The above results clearly show the benefits of disk-directed I/O over traditional caching. Two-phase I/O [dBC93] was designed to avoid the worst of traditional caching while using the same IOP software, by reading data in a "conforming distribution," then permuting it among the CPs. At first glance, disk-directed I/O is two-phase I/O implemented by rewriting IOP software so the IOPs do both phases simultaneously. In fact, disk-directed I/O has many advantages over two-phase I/O:

- There is no need to choose a conforming distribution. Our data indicates that it would be a difficult choice, dependent on the file layout, access pattern, record size, and cache management algorithm. The designers of two-phase I/O found that an `rb` distribution was appropriate for a matrix laid out in row-major order, but our results show that `rb` was rarely the best choice.

- There is the opportunity to optimize disk access with disk-request presorting, in our case obtaining a 41–50% performance boost.
- Smaller caches are needed at the IOPs, there are no prefetching mistakes, and there is no cache thrashing.
- No extra memory is needed for permuting at the CPs.
- No extra time is needed for a permutation phase; the "permutation" is overlapped with I/O.
- 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.

Thus, although we did not simulate two-phase I/O, it should be slower than disk-directed I/O because it cannot optimize the I/O as well and because the I/O and permutation phases are not overlapped. Two-phase I/O could be faster than disk-directed I/O in some patterns if the network were much slower than the disks, *and* two-phase I/O were able to use a smart permutation algorithm not available to the more dynamically scheduled disk-directed I/O.

## 8 Conclusions

Our simulations showed that disk-directed I/O avoided many of the pitfalls inherent in the traditional caching method, such as cache thrashing, extraneous disk-head movements, 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 presorted disk requests to optimize head movement, and had smaller buffer space requirements. 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 caching method, and was never substantially slower. More importantly, its performance was nearly independent of the distribution of data to CPs.

Our results also reaffirm the importance of disk layout to performance: throughput on the contiguous layout was about 5 times that on a random-blocks layout. Multiprocessor file systems for scientific applications should definitely consider extent-based layouts or other techniques to increase physical contiguity.

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; our random-blocks layout also simulates a request for an arbitrary subset of blocks from a large file. The concept of disk-directed I/O can be extended to other environments, however. Non-collective I/O access (e.g., our `rn` and `wn` patterns) can benefit, although the gain is not as dramatic. Our Memput and Memget operations would fit in well on a shared-memory machine with a block-transfer operation. Although our patterns focused on the transfer of 1-d and 2-d 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 that match some criterion.

Our results emphasize that simply layering a new interface on top of a traditional file system will not suffice. For maximum performance the file-system interface must include collective-I/O operations, and the file-system software (in particular, the IOP software) must be redesigned to use mechanisms like disk-directed I/O to support collective 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.

## Future work

There are many directions for future work in this area:

- design an appropriate collective-I/O interface,
- find a general way to specify a collective, disk-directed access request to IOPs,
- reduce overhead by allowing the application to make "strided" requests to the traditional caching system,
- optimize network message traffic by using gather/scatter messages to move non-contiguous data, and
- optimize concurrent disk-directed activities.

### Acknowledgements

Thanks to Song Bac Toh and Sriram Radhakrishnan for implementing and validating the disk model; to Chris Ruemmler, John Wilkes, and Hewlett Packard Corporation 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 paper.


## References


[BBS+94] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*. IEEE Computer Society Press, October 1994. To appear.

[BdC93] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993.

[BDCW91] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR–516, MIT, September 1991.

[BGST93] Michael L. Best, Adam Greenberg, Craig Stanfill, and Lewis W. Tucker. CMMD I/O: A parallel Unix I/O. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 489–495, 1993.

[CDG+93] David E. Culler, Andrea Drusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–283, 1993.

[CF94] Peter F. Corbett and Dror G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

[CK93] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies. Revised from Dartmouth PCS-TR93-188.

[CLVW93] Pei Cao, Swee Boon Lim, Shivakumar Venkataraman, and John Wilkes. The TickerTAIP parallel RAID architecture. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 52–63, 1993.

[dBC93] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[DdR92] Erik DeBenedictis and Juan Miguel del Rosario. nCUBE parallel I/O software. In *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC)*, pages 0117–0124, April 1992.

[Dib90] Peter C. Dibble. *A Parallel Interleaved File System*. PhD thesis, University of Rochester, March 1990.

[DSE88] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[EGKS90] Susanne Englert, Jim Gray, Terrye Kocher, and Praful Shah. A benchmark of NonStop SQL Release 2 demonstrating near-linear speedup and scaleup on large databases. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 245–246, May 1990.

[FPD93] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of the Concurrent File System of the Intel iPSC/2 hypercube. *Journal of Parallel and Distributed Computing*, 17(1–2):115–121, January and February 1993.

[GGL93] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, 1993.

[HPF93] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 1.0 edition, May 3 1993.

[KE93] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. *Journal of Parallel and Distributed Computing*, 17(1–2):140–145, January and February 1993.

[KGF94] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file systems ELFS: An object-oriented approach to high performance file I/O. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–204, October 1994.

[KHH+92] Masaru Kitsuregawa, Satoshi Hirano, Masanobu Harada, Minoru Nakamura, and Mikio Takagi. The Super Database Computer (SDC): System architecture, algorithm and preliminary evaluation. In *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences*, volume I, pages 308–319, 1992.

[KN94] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, November 1994. To appear.

[KTR94] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical Report PCS-TR94-220, Dept. of Computer Science, Dartmouth College, July 1994.

[LIN+93] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Conference*, pages 291–305, 1993.

[MK91] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputer applications. In *Proceedings of Supercomputing '91*, pages 567–576, November 1991.

[Nit92] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[NK94] Nils Nieuwejaar and David Kotz. A multiprocessor extension to the conventional file system interface. Technical Report PCS-TR94-230, Dept. of Computer Science, Dartmouth College, September 1994.

[OCH+85] John Ousterhout, Hervé Da Costa, David Harrison, John Kunze, Mike Kupfer, and James Thompson. A trace driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24, December 1985.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[Pie89] Paul Pierce. A concurrent file system for a highly parallel mass storage system. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, 1989.

[PP93] Barbara K. Pasquale and George C. Polyzos. A static analysis of I/O characteristics of scientific applications in a production workload. In *Proceedings of Supercomputing '93*, pages 388–397, 1993.

[Roy93] Paul J. Roy. Unix file access and caching in a multicomputer environment. In *Proceedings of the Usenix Mach III Symposium*, pages 21–37, 1993.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[WMR+94] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: An operating system for massively parallel systems. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, 1994.

Many of these papers can be found at

`http://www.cs.dartmouth.edu/pario.html`

The disk-model software can be found at

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