Expanding the Potential for Disk-Directed I/O

David Kotz
Department of Computer Science
Dartmouth College
dfk@cs.dartmouth.edu

Abstract

As parallel computers are increasingly used to run scientific applications with large data sets, and as processor speeds continue to increase, it becomes more important to provide fast, effective parallel file systems for data storage and for temporary files. In an earlier work we demonstrated that a technique we call disk-directed I/O has the potential to provide consistent high performance for large, collective, structured I/O requests. In this paper we expand on this potential by demonstrating the ability of a disk-directed I/O system to read irregular subsets of data from a file, and to filter and distribute incoming data according to data-dependent functions.

1 Introduction

Despite dramatic improvements in parallel-computing hardware and software, many parallel programmers discover that their application's performance is limited by the rudimentary data-storage systems available on today's multiprocessors. When they find a multiprocessor that is configured with sufficient parallel-I/O hardware (unfortunately, many are not) they often discover that the file system software is not designed to meet their needs [1], or has poor performance [2].

As a result, there are several proposals for new interfaces, run-time libraries, compilers, languages, and file systems to support parallel applications on parallel computers. The focus of this paper is on a file-system technique called disk-directed I/O, which can dramatically improve the performance of reading and writing a large, regular data structure (like a matrix) between memory that is distributed across many processors and a file that is distributed across many disks [3].

There are two ways to look at this paper. One view is that we are exploring the ability of disk-directed I/O to accommodate three extensions: data-dependent distribution (Section 3), data-dependent filtering (Section 4), and working with irregular subsets (Section 5). Another view is that we are applying one idea from disk-directed I/O (shifting control from the compute nodes to the I/O nodes) to new situations (data-dependent distribution and data-dependent filtering), although our implementation of those techniques is in a disk-directed-I/O system.

This research was funded by NSF under grant number CCR-9404919, and by NASA Ames under agreement number NCC 2-849. The work was performed while the author was on sabbatical at Syracuse University.

2 Background

There are many different parallel file systems (see [4, 5] for a partial survey). Most are based on a fairly traditional Unix-like interface, in which individual processes make a request to the file system for each piece of the file they read or write. Increasingly common, however, are specialized interfaces to support multidimensional matrices or *collective I/O* ([6] is an example of both). With a collective-I/O interface, all processes make a single joint request to the file system, rather than numerous independent requests.

In this paper we assume that the multiprocessor is comprised of two types of processor nodes: those without disks, which are called compute processors (CPs), and those with disks, which are dedicated to the file system and which are called I/O processors (IOPs). Most, though not all, of the contemporary parallel file systems are designed for machines with an architecture of this type.

Some database machines have a data-dependent tuple-filtering function on the IOPs, e.g., Tandem NonStop [7]. The Super Database Computer [8] has a load-dependent data-distribution mechanism, in which disk controllers continuously produce *tasks* that are consumed and processed by CPs.

The PASSION library for scientific applications can read submatrices that can be represented as a large contiguous region with some "holes" of unwanted data, by reading the full region of data and then "sieving" out the undesired data [9]. This sieve is not data-dependent, and is used to allow the library to make larger, more efficient requests to the file system.

Disk-directed I/O. Disk-directed I/O is a technique for optimizing data transfer given a high-level, collective inter-

Appeared in the Symposium on Parallel and Distributed Processing, October 1995, pp. 490-495.
Available at URL http://www.cs.dartmouth.edu/~dfk/papers/kotz:expand.pdf
Earlier version available as Dartmouth PCS-TR95-254 at URL ftp://ftp.cs.dartmouth.edu/TR/TR95-254.pdf

face [3]. In this scheme, the complete high-level, collective request is passed to the I/O processors, which examine the request, make a list of disk blocks to be transferred, sort the list, and then use double-buffering and special remote-memory "get" and "put" messages to pipeline the transfer of data between compute-processor memories and the disks. Compared to a traditional system with caches at the I/O processors, this strategy optimizes the disk accesses, uses less memory (no cache at the I/O processors), and has less CPU and message-passing overhead.

An interesting application. Karpovich et al. [10] describe the problem of storing and retrieving radio-astronomy data sets. The read-mostly data set is large and multi-dimensional: each data point represents an astronomical reading at some *time* at some *frequency* on some *instrument* pointed at some *region of the sky*. Needless to say the data set is extremely sparse. They store the data set by partitioning data into buckets along a few of the dimensions, and sorting within a bucket along other dimensions. Applications rarely read the entire data set; instead, they request a subset of the data by specifying ranges for the time, frequency, and region. Using an index of buckets, only the necessary buckets must be read into memory. The buckets are then filtered to extract the items of interest, and (in a parallel application) distributed among memories of the multiprocessor according to the application's needs.

Clearly this application has different I/O needs from those imagined for the disk-directed-I/O system in [3]. It reads an irregular, discontiguous subset of data from the file. It filters out and discards some of the data it reads, after examining the data. Finally, it distributes the data among the memories in a data-dependent manner. In the remainder of the paper, we show how the concept of disk-directed I/O can also include these unusual requirements.

3 Data-dependent distributions

In the disk-directed I/O system described in [3], matrices could be read from the file into memory according to one of a variety of distribution patterns. As each block was read from disk (in whatever order was convenient for the disk), the records within that block were sent to the appropriate location in the appropriate memory, based on the distribution function. In [3] the distribution function was independent of the data; of course, a data-dependent distribution function could easily be used for the same purpose.

A traditional file system, however, is quite different. With a data-independent distribution, each processor independently computes the locations of the records it requires from the file, and reads those records. With a data-dependent distribution, however, there is no way for processors to request their own set of data. A reasonable solution is similar to two-phase I/O [11]: each processor reads some convenient subset of data from the file, examines each record to compute the distribution function, and then sends the data to the appropriate processor.

In both cases we assume that the distribution function can only decide to which processor each record belongs, and send the record to that processor. Once there, the processor appends the record to its buffer for later processing.

3.1 Experiments

To gauge the impact of data-dependent distribution on performance, we devised an experiment to compare our disk-directed I/O and traditional-caching file systems. Of course, for the purpose of this experiment it matters little what distribution function we actually use— even a data-*independent* function would do. We used a cyclic distribution (`rc` in [3]). Thus, the disk-directed system needed no change for this experiment. In the traditional caching system, the compute processors each looped reading blocks from the file, and for each record within each block, sent the record on to the appropriate destination processor. Logically, it made no difference which blocks were read by which processor, since most records would be redistributed anyway. For best I/O performance on contiguous layouts [3], we chose to have compute processors read the *blocks* in a cyclic distribution.

We ran these experiments on our simulator from [3], configured as shown in Table 1. In all cases a 10 Mbyte file was striped across disks, block by block, using one of two block layouts within each disk: contiguous or random. We repeated each experiment in this paper five times and report the mean value of each measure here. The largest coefficient of variation of any data point was 2.6%, so the trials were extremely consistent.

3.2 Results

Figure 1 shows the results of these experiments. These charts plot the execution time for the experiment, the number of bytes sent through the interconnect as messages, and the number of messages, each *normalized* against the best possible value for that measure. In all cases a smaller number is better, with 1.0 being the best. For execution time, the best possible value is computed from the amount of data read off disk and the peak throughput of the disk drives. Clearly, that execution time was only possible with no overhead and a contiguous layout. For message bytes, the best possible value is obtained when only the data itself is sent from the I/O nodes directly to the appropriate compute nodes. For message count, the best possible value is the number of records, since in our system each record is sent to its destination as a separate message. Our message protocols involve an acknowledgement for most messages, leading to message counts of 2 or more.

Each chart compares two disk layouts (contiguous and random), two record sizes (64 bytes and 8192 bytes), and two file systems (traditional caching (TC) and disk-directed

Table 1: Parameters for simulator.

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	6×6 torus
Interconnect bandwidth	200×10^6 bytes/s bidirectional
Interconnect latency	20 ns per router
Routing	wormhole

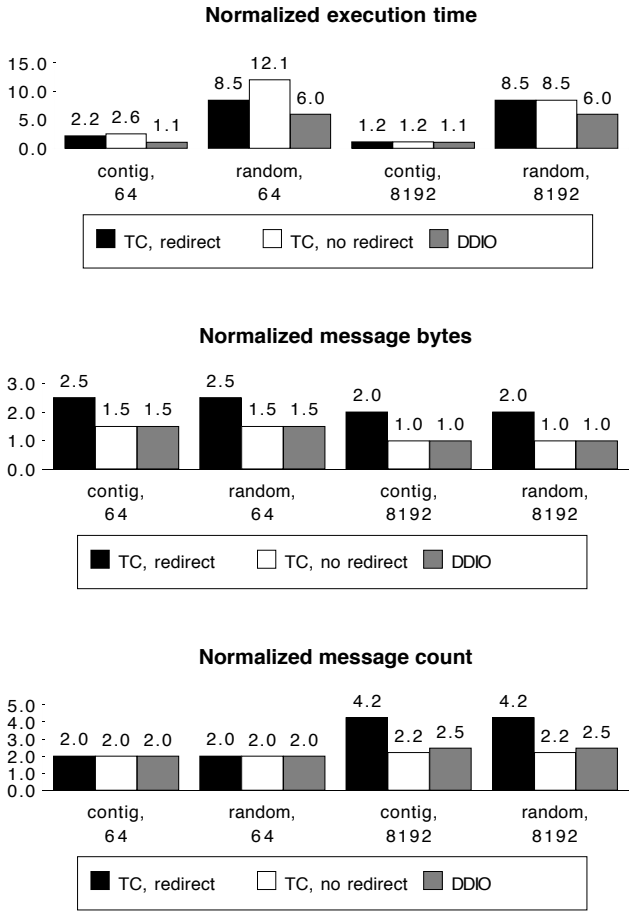

Figure 1: Results of data-dependent distribution experiments. Each graph is normalized against the "best" value for that measure. TC is traditional caching, and DDIO is disk-directed I/O. "contig" is a contiguous disk layout, and "random" is a random disk layout. We used both 64- and 8192-byte records.

I/O (DDIO)). For comparison, we add "TC, no redirect", in which each processor directly requested the data it needed; this would be impossible with a truly data-dependent distribution, but provides an interesting comparison.

Consider first the 8192-byte records. On the contiguous layout, all three systems performed similarly (which is why we chose this distribution pattern), but using traditional caching to fetch and then redirect the data pushed all of the data through the network twice. The number of messages nearly doubled as each block needed four messages: I/O request, I/O reply, send to another node, and acknowledgement from the other node. On the random layout, disk-directed I/O was faster because it could schedule the I/Os for better disk performance. Disk-directed I/O used slightly more messages here; these were startup overhead and would be negligible with a larger number of records, as can be seen in the bars for 64-byte records.

Consider the 64-byte records, which present an interesting picture. Despite nearly doubling the amount of message traffic, traditional caching with data-dependent redistribution was *faster* than the direct-access version. Here we were essentially using a pipelined form of two-phase I/O [11]. Compute processors requested whole blocks from the I/O nodes, rather than small records. The larger requests reduced the overhead at the I/O nodes, which in this case more than offset the extra work at the compute nodes and in the network. Disk-directed I/O was always better, however, because it could not only avoid the extra messages, but had less overhead and could optimize the disk traffic.

Our distribution function was simple to compute, so there was little difference whether it was computed by CPs or IOPs. With a more complex distribution function, systems with fewer spare IOP cycles may find the IOPs slow enough to offset the gains due to reduced network usage.

In our experiments the network was fast enough to not be a bottleneck. Thus, redistributing the data from the compute nodes was not a performance problem. Systems with slower networks, or with networks being shared by other applications, may find it especially valuable to be able to distribute the data directly from the IOPs, rather than indirectly through the CPs. An important example is a workstation cluster being used as a parallel computer.

4 Data-dependent filtering

If the disk-directed file system can make distribution decisions based on the data in the record, it is easy to see how it could filter out records according to some function

of the data in the record. (Again, we assume that we do not send the data to a specific location within the destination processor; instead, the destination appends newly arriving records to its current buffer.) No less I/O is needed, but there may be substantially less network traffic.

In a traditional system, each compute processor reads its data in chunks, filtering and compacting each chunk before reading more from the file system. Here filtering should neither reduce the I/O nor reduce message traffic. Thus, traditional caching is essentially unaffected by filtering.

The main point of these experiments is not to show that disk-directed I/O is better than traditional caching, but that I/O-node filtering may be helpful. Such filtering fits naturally into a disk-directed system structure, however.

4.1 Experiments

We again used the cyclic (`rc`) distribution pattern. We filtered out either 0% or 90% of the records, according to a random filtration function (since again it does not matter specifically what function we use). Other system parameters are the same.

4.2 Results

Figure 2 shows the results of these experiments with two disk layouts (contiguous and random), two record sizes (64 bytes and 8192 bytes), two file systems (traditional caching and disk-directed I/O), and two filtration ratios (0% and 90%). Since we used a trivial filtration function, traditional caching has the same performance regardless of the filtration ratio. On random layouts disk-directed I/O can optimize the disk schedule, and on small records it has lower overhead, so it had better performance in those cases than did traditional caching. These differences are nothing new. The chart shows that adding filtration does not affect performance, which is no surprise, but that it dramatically reduces the network traffic. Thus, filtering at the I/O processor rather than at the compute processor will have less impact on other applications, or on other simultaneous communication in the same application, and would perform better in systems where the network may be a bottleneck, e.g., workstation clusters.

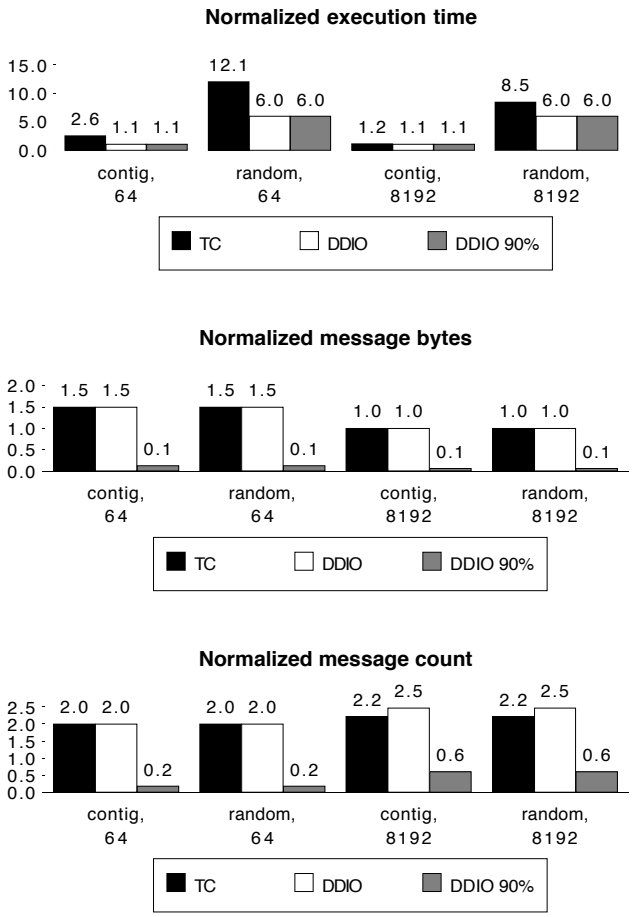

Figure 2: Results of data-dependent filtration experiments. Traditional caching is unchanged by filtering. We compare normal DDIO with a DDIO that filtered 90% of all records. The message metrics are normalized against the amount of traffic necessary with no filtration.

5 Irregular subsets

There are some applications that need to request an irregular, discontiguous subset of a file, such as the radio-astronomy data set described above. How might disk-directed I/O support these kinds of applications?

5.1 Experiments

We simulate applications that request an irregular subset of blocks from the file, although we do assume that they could specify the entire subset in a list (e.g., in a batch request [12]). The list was in logical sorted order. In the disk-directed I/O system, each compute processor sent its list of requests to the appropriate I/O processors. The I/O processors converted the logical offsets to a list of physical block numbers, sorted the list for the best schedule, and processed the blocks much as before. In the traditional-caching system, each compute processor requested one block at a time, according to the list.

We used a 100 Mbyte file and selected an irregular subset of about 22% of the file blocks; of that subset, each processor requested those blocks that would also be distributed to it if the whole file were distributed cyclically, block by block, to all processors. Thus, this pattern was a subset of the `rc` pattern. The subset was arranged so that each processor received about the same number of blocks, and each disk received about the same number of read requests.

5.2 Results

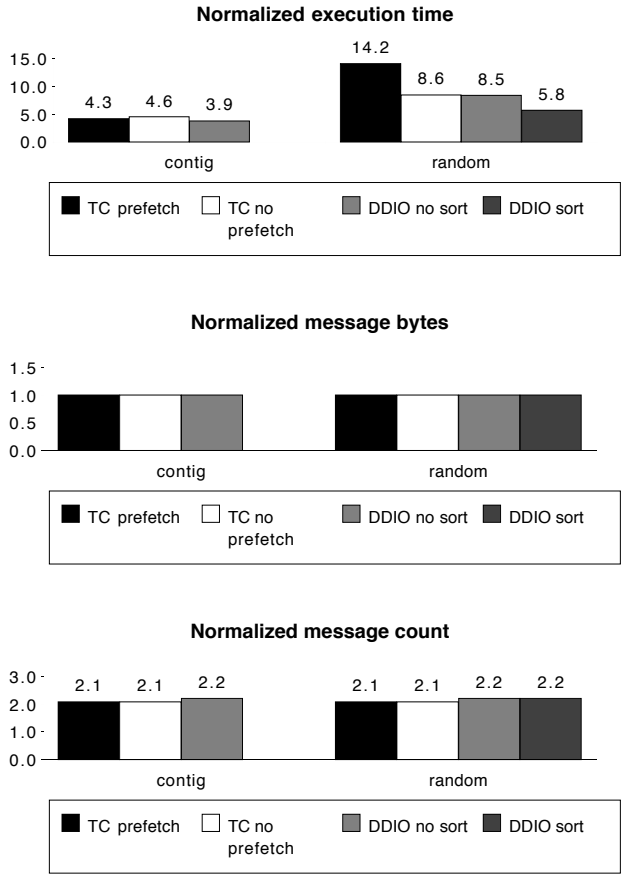

Figure 3: Normalized measures for experiments requesting an irregular subset of the file. Best execution time is computed from the peak disk bandwidth and the actual amount of data transferred. Best message count and message bytes are computed assuming that only the requested blocks will flow through the network.

Figure 3 displays the results of these experiments. Our traditional caching has an admittedly naive prefetching policy (at a request for file block i, prefetch file block $i + 1$), so we include some results with prefetching disabled. In the random layout our prefetching made performance dramatically *worse*, by prefetching useless blocks on the mistaken assumption of sequential access. In the contiguous layout those mistakes made little difference because those prefetches were quickly completed by the drive's own prefetching. A production system would, of course, use smarter prefetching policies (although something like the Unix prefetching policy does not adapt well to parallel access patterns) [13, 14].

Prefetching aside, there were no unusual differences between disk-directed I/O and traditional caching that could be attributed to the irregularity of access. The main point here is that disk-directed I/O was perfectly capable of dealing with irregular access patterns as well as regular access patterns.

6 Conclusions

While our earlier work demonstrated the value of disk-directed I/O for large, collective requests, the examples were limited to regular structures with a data-independent distribution [3]. In this paper we show that disk-directed I/O could accommodate irregularly structured requests, data-dependent distributions, and data-dependent filtering, with no loss in performance. These features, which fit easily and efficiently into a disk-directed I/O structure but not into a traditional-caching structure, give disk-directed I/O a further advantage over traditional caching: traditional caching doubled the network traffic when doing a data-dependent distribution function, and was not able to reduce the network traffic when doing data-dependent filtering. Although the difference in network traffic did not affect the execution time in our experiments, because we used a very fast network, it may affect performance on systems where the network is slower or shared (as in a workstation cluster). On the other hand, if the data-dependent functions are computationally expensive (they were trivial in our experiments), and the network is fast, there may be no advantage to trading IOP cycles for network bandwidth.

Given that disk-directed I/O has the potential for all of these capabilities, the crucial remaining question is how to make these capabilities available to the programmer. In particular, how does the user (or compiler) tell the I/O processor about its distribution function, filtration function, and which file data to read? We are beginning to study this issue.

Availability

Our simulator, the full technical-report version of this paper [15], and many of the papers below, are available at `http://www.cs.dartmouth.edu/~dfk/`.

References

[1] David Kotz and Nils Nieuwejaar, "Dynamic file-access characteristics of a production parallel scientific workload", in *Proceedings of Supercomputing '94*, Nov. 1994, pp. 640–649.

[2] Bill Nitzberg, "Performance of the iPSC/860 Concurrent File System", Tech. Rep. RND-92-020, NAS Systems Division, NASA Ames, Dec. 1992.

[3] David Kotz, "Disk-directed I/O for MIMD multiprocessors", in *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, Nov. 1994, pp. 61–74, Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[4] Thomas H. Cormen and David Kotz, "Integrating theory and practice in parallel file systems", in *Proceedings of the 1993 DAGS/PC Symposium*, Hanover, NH, June 1993, Dartmouth Institute for Advanced Graduate Studies, pp. 64–74.

[5] Dror G. Feitelson, Peter F. Corbett, Yarson Hsu, and Jean-Pierre Prost, "Parallel I/O systems and interfaces for parallel computers", in *Multiprocessor Systems — Design and Integration*, C.-L. Wu, Ed. World Scientific, 1995, To appear.

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

[7] 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*, May 1990, pp. 245–246.

[8] 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*, 1992, vol. I, pp. 308–319.

[9] Rajeev Thakur, Rajesh Bordawekar, Alok Choudhary, Ravi Ponnusamy, and Tarvinder Singh, "PASSION runtime library for parallel I/O", in *Proceedings of the Scalable Parallel Libraries Conference*, Oct. 1994, pp. 119–128.

[10] John F. Karpovich, James C. French, and Andrew S. Grimshaw, "High performance access to radio astronomy data: A case study", in *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, Sept. 1994, Also available as Univ. of Virginia TR CS-94-25.

[11] 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*, 1993, pp. 56–70, Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[12] Nils Nieuwejaar and David Kotz, "Low-level interfaces for high-level parallel I/O", in *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, Apr. 1995, pp. 47–62.

[13] David Kotz and Carla Schlatter Ellis, "Practical prefetching techniques for multiprocessor file systems", *Journal of Distributed and Parallel Databases*, vol. 1, no. 1, pp. 33–51, Jan. 1993.

[14] R. Hugo Patterson and Garth A. Gibson, "Exposing I/O concurrency with informed prefetching", in *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, Sept. 1994, pp. 7–16.

[15] David Kotz, "Expanding the potential for disk-directed I/O", Tech. Rep. PCS-TR95–254, Dept. of Computer Science, Dartmouth College, Mar. 1995.