Exploring the use of I/O Nodes for Computation in a MIMD Multiprocessor

David Kotz and Ting Cai

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

Dartmouth Technical Report PCS-TR94-232

October 19, 1994
Revised February 20, 1995

Abstract

As parallel systems move into the production scientific computing world, the emphasis will be on cost-effective solutions that provide high throughput for a mix of applications. Cost-effective solutions demand that a system make effective use of all of its resources. Many MIMD multiprocessors today, however, distinguish between "compute" and "I/O" nodes, the latter having attached disks and being dedicated to running the file-system server. This static division of responsibilities simplifies system management but does not necessarily lead to the best performance in workloads that need a different balance of computation and I/O.

Of course, computational processes sharing a node with a file-system service may receive less CPU time, network bandwidth, and memory bandwidth than they would on a computation-only node. In this paper we examine this issue experimentally. We found that high-performance I/O does not necessarily require substantial CPU time, leaving plenty of time for application computation. There were some complex file-system requests, however, which left little CPU time available to the application. (The impact on network and memory bandwidth still needs to be determined.) For applications (or users) that cannot tolerate an occasional interruption, we recommend that they continue to use only compute nodes. For tolerant applications needing more cycles than those provided by the compute nodes, we recommend that they take full advantage of *both* compute and I/O nodes for computation, and that operating systems should make this possible.

1 Introduction

As parallel systems move into the production scientific computing world, the emphasis will be on cost-effective solutions that provide high throughput for a mix of applications. Several applications, each with different computational and I/O needs, will be simultaneously active within a single

This research was funded by NSF under grant number CCR-9404919, by NASA Ames under agreements numbered NCC 2-849 and NAG 2-936, and by Dartmouth College.

multiprocessor. Cost-effective solutions demand that a system make effective use of all of its resources.

Many MIMD multiprocessors today are configured with two distinct types of processor nodes: those that have disks attached, which are dedicated to file I/O, and those that do not have disks attached, which are used for running applications. This static division of responsibilities simplifies system management but does not necessarily lead to the best performance in workloads that need a different balance of computation and I/O. For example, a system which makes all nodes available to computational applications increases its overall computational power and may therefore be more cost effective.

Computational processes running on nodes that also serve part of the file system, however, may receive less CPU time, network bandwidth, and memory bandwidth than they would on a computation-only node. The conventional wisdom is that the CPU overhead of the file-system code running on I/O nodes, coupled with the unpredictable and erratic nature of I/O activity, would substantially disrupt the performance of computational applications. In this paper we examine this issue experimentally, focusing on the impact of a file-system server on the CPU time available to local computational processes. We found that high-performance I/O does not necessarily require substantial CPU time, leaving plenty of time for application computation. There were some complex file-system requests, however, which left little CPU time available to the application. (The impact on network and memory bandwidth still needs to be determined.) For applications (or users) which cannot tolerate an occasional interruption, we recommend that they continue to use only compute nodes. For other applications, particularly those that can adapt to changing load, we recommend that they take full advantage of *both* compute and I/O nodes for computation. After all, our results show that the I/O nodes have free cycles.

We begin in the next section with background information about multiprocessor file systems. Section 3 describes some simulations and their results and Section 4 describes some measurements on a real system. We summarize our conclusions in Section 5.

2 Background

There are many different parallel file systems [Kri94, Pie89, FPD93, Roy93, LIN+93, DdR92, CF94, Dib90, DSE88, MS94, HdC95, HER+95]. Most, though not all, are designed for machines that have dedicated I/O nodes. 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 [CFPB93, SW94, GL91, GGL93, BdC93, BBS$^+$94, Mas92], and interfaces that support *collective* I/O [GGL93, BdC93, BBS$^+$94, Mas92]. With a collective-I/O interface, all processes make a single joint request to the file system, rather than numerous independent requests.

Disk-directed I/O is a promising new technique that takes advantage of a collective-I/O interface, and leads to much better performance than file systems based on traditional caching strategies [Kot94]. With disk-directed I/O, compute nodes make a collective request to the file system, which forwards the request to all I/O nodes. Each I/O node examines the request to determine which file blocks are on its disks, sorts the file blocks by physical location to determine an efficient schedule, and then begins a series of transfers. In effect, the I/O nodes are in charge of the data transfer, which is organized to best suit the disks' performance characteristics. Each I/O node uses two buffers to overlap disk transfer and network transfer. For example, when reading, one buffer is filled by reading a block from disk while another buffer is emptied by scattering its contents among the compute-node memories according to the requested distribution. Data transfers between compute nodes and I/O nodes use low-overhead "Memput" and "Memget" messages that move data directly to and from the application buffer. The experiments in [Kot94] show that disk-directed I/O obtains nearly the peak disk bandwidth across many data distributions and system configurations.

There have been no similar studies of CPU activity on the I/O nodes of multiprocessors. A ten-year old study of diskless workstations [LZCZ86] found that file-server CPU load can be extremely high. To be able to provide high performance during periods of intense I/O activity, however, a balanced multiprocessor spreads its disks across many I/O nodes so that the I/O-node CPUs will not be a performance bottleneck. This configuration leaves open the possibility that the I/O nodes will be underutilized during other periods.

3 Simulation Experiments

We wanted to measure the worst-case impact of unpredictable I/O interruptions on a computational application, so we devised an experiment involving two 16-processor applications on a 32-node multiprocessor, in which one application did nothing but I/O, and the other did nothing but computation. The I/O application either read or wrote a file that was striped across disks attached to the *computational application's* processors. Thus, the computational application was occasionally interrupted so that the file system could service I/O requests for the other application. These interruptions slowed the computational application in two ways. First, every cycle spent servicing the I/O request was another cycle delay for the interrupted application. Second, delaying one process in the computational application indirectly delayed other processes that waited for the process at a future synchronization point [MCD+91].

In our experiments we used two different kinds of computational applications, 36 different kinds of I/O applications, and two different kinds of file systems, all on a parallel file-system simulator.

3.1 Computational applications

Our two computational applications did nothing but computation. The first application, designed to measure the effect of interruptions on raw computational performance, had no synchronization or other communication between processes. The second application was designed to measure the effect of load imbalance caused by I/O-related interruptions, by having all processes meet at a barrier every 5 msec of virtual time. With no interruptions, all processes would meet at every barrier at precisely the same physical times, and thus would never wait. An interruption of the computation on one processor, however, delayed both that process and all other processes that had to wait for it at the next barrier. Thus, a small perturbation of the execution time of one process could have a ripple effect that was much larger than the original.

We chose to use barriers because they have the most drastic effects on performance if the processors become unbalanced: all processes must wait for the slowest process. Similarly we chose a tight 5 msec interval to represent a challenging case (several NASA benchmarks on the Paragon and an SGI cluster were measured with inter-barrier times of 6, 17, or 64 msec [Nit94]).

Note that our barrier experiment also represents a computational application that is running on many processors, only some of which are involved in serving I/O, while others are left to run at full speed. All other things being equal, those without I/O interruptions will always have to wait for those with I/O interruptions. If those slow processors run at 95% of the speed, then the whole application runs at 95% of full speed, regardless of the number of uninterrupted processors.

3.2 I/O applications

Our I/O applications did nothing but I/O. They each transferred a one- or two-dimensional array of records, but in either case the file size 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. Thus, 10 MB was a compromise to save simulation time. The file was striped, block by block, across the 16 disks attached to the *computational application's* processors. The matrix was distributed

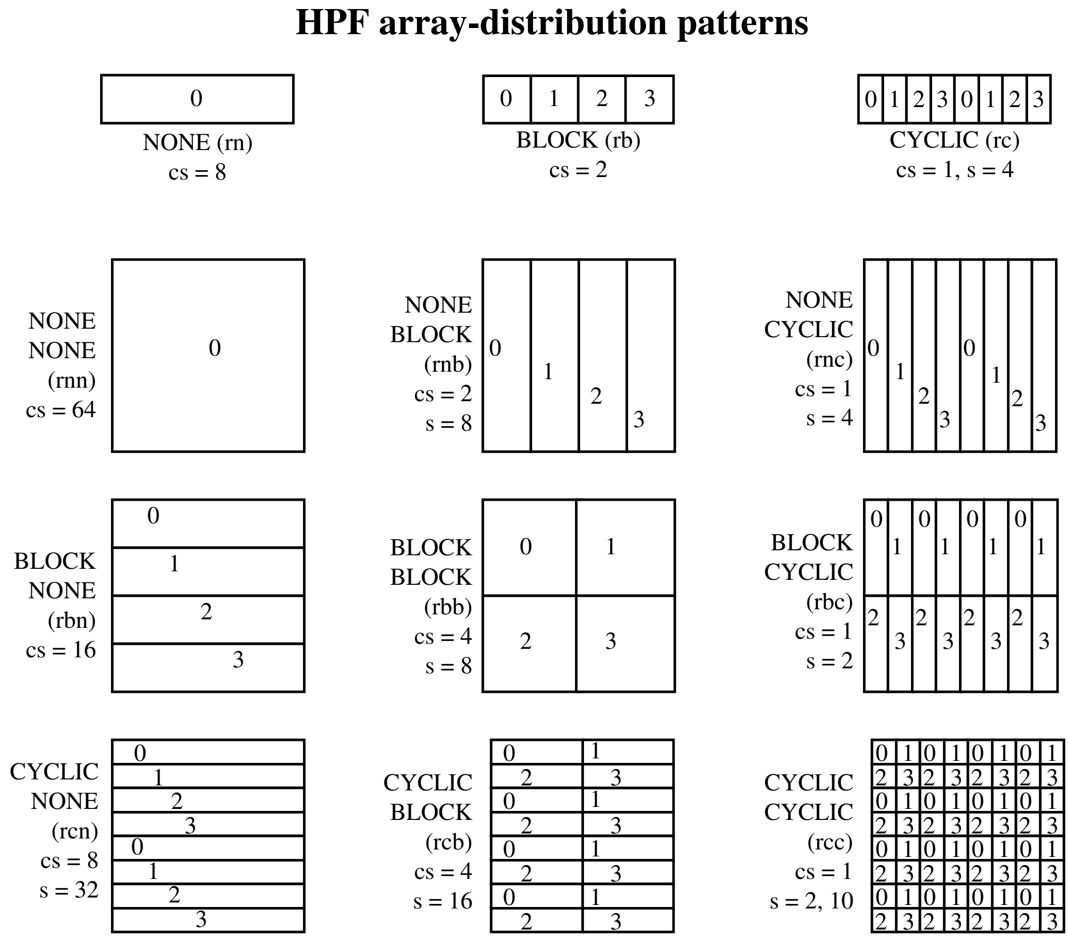

Figure 1: Examples of matrix distributions, which we used as file-access patterns in our experiments. These examples represent common ways to distribute a 1x8 vector or an 8x8 matrix over four processors. Patterns are named by the distribution method (NONE, BLOCK, or CYCLIC) in each dimension (rows first, in the case of matrices). Each region of the matrix is labeled with the number of the compute node 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 compute node (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 compute node, where relevant.

across the 16 memories of the I/O application according to one of the HPF distributions [HPF93], as shown in Figure 1. Each matrix element was either 8 bytes or 8 Kbytes. Clearly, patterns that use 8-byte elements and a column-cyclic distribution lead to a fine-grained data distribution, and typically to more I/O overhead.

3.3 File-system implementations

The file accessed by the I/O applications was striped across all 16 disks. Within each disk the blocks of the file were laid out contiguously, that is, the logical blocks of the file were laid out in consecutive physical blocks on disk. We chose this layout because it provides the highest I/O throughput, thus keeping the file-system code the most busy. Any other layout would transfer data more slowly, requiring interruptions less often.

We modeled two different file systems: traditional caching and disk-directed I/O. *Traditional caching* was meant to simulate a typical parallel file system where compute nodes, on behalf of application processes, made independent requests to the appropriate I/O nodes. Each application request to a compute node was for some contiguous range of bytes in the file, but because the file was striped by blocks, each compute-node request to an I/O node could be for at most one block. The I/O nodes each maintained a block cache, with LRU replacement and support for prefetching and write-behind. The I/O node was multithreaded, with a new thread created for each incoming request. Threads shared a data structure describing the LRU buffer list, blocking when waiting for a buffer to be flushed for re-use, or for a buffer to be filled with new data from disk. This choice led to a clean design with plenty of concurrency, at the cost of some thread-switching overhead. More importantly, the distribution of I/O-request service times was highly variable, depending on whether it was a cache hit or miss, could easily locate a free buffer, and so forth.

Disk-directed I/O is a new technique that takes advantage of a collective-I/O interface, and leads to much better performance than traditional caching [Kot94]. As described above, it works by giving control over the order and pace of data transfer to the I/O nodes, who optimize the transfer for maximum disk performance. After an initial burst of CPU activity to determine the disk schedules, the only ongoing CPU overhead is to compute the distribution of each block's data among the compute-node memories. When reading, for example, some blocks coming off of disk must be split into several smaller pieces, which are sent to the remote compute-node memories. Some distributions, particularly when the matrix-element size is small, involve substantial computations to determine the ultimate location of each element.

3.4 Measurement methodology

Rather than actually running a computational application, we measured the fraction of CPU time available for running a computational application on one set of processors, during the period the I/O application was running on the other set of processors. Before and after the I/O application ran, of course, there were no interruptions and so the computational application received 100% of the CPU's time; since we were interested in the effect of the I/O requests, we only measured the period when the I/O application was running. Note that this methodology means that the I/O interruptions had priority over the computation; again, this experiment was designed to expose the worst-case effects on the computational application.

To make this measurement, we collected traces of the CPU activity on the I/O nodes of our two file systems, under load from one of the I/O applications. We processed the traces to count idle cycles as a proportion of total cycles (i.e., the inverse of the CPU utilization). However, not all idle cycles would be available to a real computation, due to the overhead for switching context between the application and the file system. For each interruption, therefore, we deducted 50 μsec.[1] Idle intervals shorter than 50 μsec were therefore useless to the computation, and so were not counted.

3.5 Simulator

Our traces were collected from the STARFISH parallel file-system simulator [Kot94], which 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.

3.6 Results

Figure 2 compares the impact of all 36 I/O applications on our first computational application, as well as showing the I/O bandwidth achieved by the I/O application. Ideally, all points would be in the upper-right corner, indicating high I/O throughput and computational performance. Most of the disk-directed-I/O points are there, except for six "hard" patterns on the left. Traditional caching had much poorer I/O performance, and its CPU needs were slightly smaller (to some extent the CPU needs appear smaller because the CPU impact was spread over a longer physical time, due to the poor I/O performance).

To get a better understanding of Figure 2, we selected two representative patterns for more detailed presentation: one that was extremely easy and fast in both file systems, and another that was extremely complex and slow in both file systems. The easy pattern (representing points in the upper right) distributed a one-dimensional matrix of 8-KB records cyclically among the memories (recall that 8 KB was the file-system block size). The hard pattern (representing points

[1]This is a moderate context-switch time [ALBL91], even when cache effects are considered. In any case, preliminary experiments showed that our results were not sensitive to this parameter.

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

in the lower left) distributed a two-dimensional matrix of 8-byte records among the memories in a BLOCK-CYCLIC layout, to use HPF terminology. We look at both the read and write versions of these two patterns, for a total of four cases.[2]

Table 2 shows the results in detail for each of these four access patterns and each file system. The "easy" access patterns took little CPU time, leaving 90–95% of the CPU for the computational application. Nonetheless, they sustained 32–33 MB/s, which is 86–89% of the disks' peak bandwidth. Of the two file systems, disk-directed I/O had higher I/O throughput and less CPU demand.

For the "hard" access patterns, however, the situation was quite different. I/O performance suffered, in traditional caching because it managed the disks and cache poorly, and in disk-directed I/O because of the amount of overhead in handling thousands of 8-byte messages.[3] Nonetheless, this example points out a situation where the I/O benefits of disk-directed I/O were enormous. It came at a cost, however, in terms of the amount of CPU overhead required, which in the worst case left only 3.4% of the CPU cycles available for the computational application. The CPU overhead of traditional caching does not seem to be so bad, but this was again partially due to the poor I/O

[2]In [Kot94], the easy patterns are called `rc` and `wc` with 8-KB records, and the hard patterns are called `rbc` and `wbc` with 8-byte records.

[3]We suspect the latter may be improved with a gather/scatter message-passing mechanism.

Table 2: Percent of CPU time available to the computational application (100% is ideal), and the amount of data throughput achieved by the I/O application.

	Traditional Caching		Disk-directed I/O	
	CPU available (percent)	I/O throughput (MBytes/s)	CPU available (percent)	I/O throughput (MBytes/s)
easy read	95.	32.2	95.	33.5
easy write	90.	32.4	95.	32.2
hard read	60.	2.2	3.4	16.2
hard write	87.	0.7	5.1	14.2

performance spreading out the overhead over many cycles.

When we added barrier synchronizations to the computational application, the I/O activity of course had a bigger effect. Figure 3 plots the effect of all 36 access patterns on this synchronizing application. Table 3 focuses on the same representative cases as before. First, note that there was only minimal effect on the easy access patterns. The interruptions were short and rare, leading to little disturbance. On the "hard" patterns in the traditional-caching file system, however, there was a dramatic effect due to the highly variable amount of computation needed for cache-management operations (for example, a cache miss took much more computation than a cache hit), leading to load imbalance within the computational application.

Table 3: A comparison of the amount of CPU time usable by the computation, with and without barrier synchronization. In the presence of load imbalance caused by I/O interruptions, barriers cause some processors to idle, reducing the percentage of CPU that was "usable."

	Traditional Caching		Disk-directed I/O	
	no barriers	barriers	no barriers	barriers
easy read	95.	92.	95.	93.
easy write	90.	87.	95.	93.
hard read	60.	11.	3.4	2.4
hard write	87.	7.1	5.1	3.4

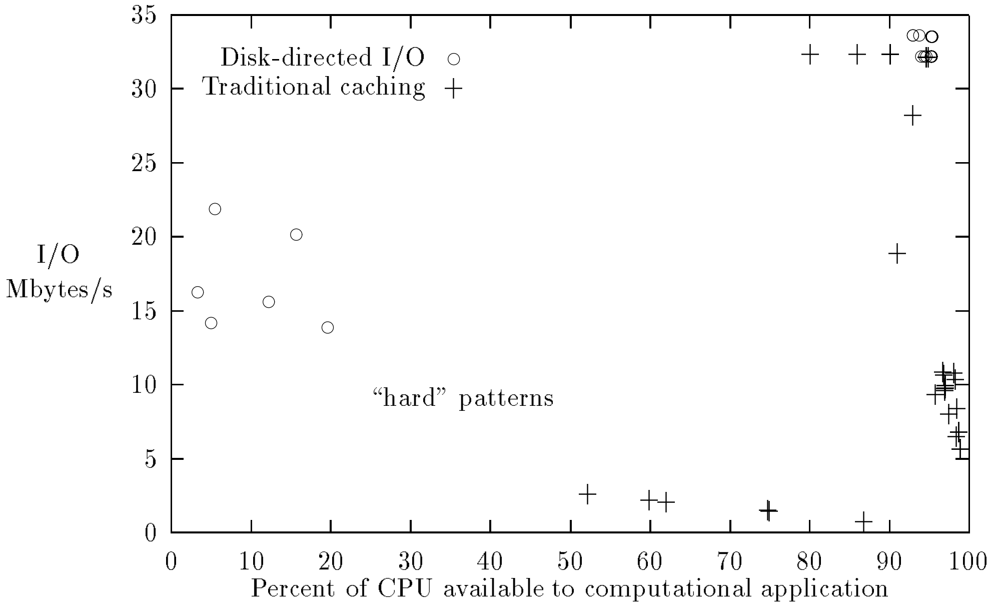

Figure 2: I/O throughput *vs.* computational performance for all 36 different access patterns, and both file-system implementations. The upper-right corner represents the best cases; there are actually 41 points above 30 MB/s, many of which overlap in this picture.

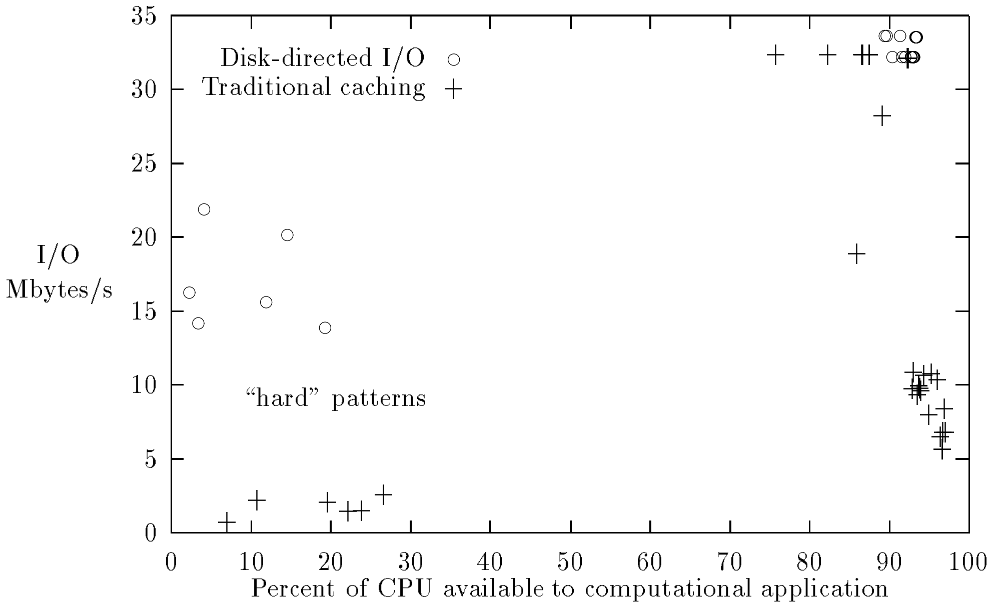

Figure 3: Similar to Figure 2, but with a computational application that includes a barrier synchronization every 5 msec of virtual time. Again, many of the points in the upper right overlap.

4 Measurement Experiments

The simulations in the previous section allowed us to examine the effects of a variety of workloads on two very different file systems in a controlled setting. To support these results, we have also measured the effects of a real file system on a real computation, using a cluster of eight IBM RS/6000-250 workstations in Dartmouth's FLEET lab.[4] We used a LINPACK benchmark program as a computational application. We ran several copies of this program in parallel, one on each of six workstations. Each process ran 15 iterations of the LINPACK computation, stopping for a barrier after each iteration.[5] Needless to say, this synthetic parallel application is perfectly load balanced. Then, we had one of the other two workstations run a simple program that either read or wrote a 400 MB file with 1 KB requests, sequentially or randomly, where the file was served through NFS from one of the hosts running the LINPACK program. Due to periodic barriers, any slowdown experienced by that node caused the entire application to slow down. (As a control, we ran a similar test with six workstations running the LINPACK program while the other two did I/O, one as client and one as server; despite the network traffic, the I/O had no effect on the LINPACK program's barriers.)

Table 4 presents the results. Although we cannot fully explain the differences in the effects of the I/O access patterns, it is clear that the application was able to run at 60–97% efficiency despite the CPU impact of the I/O. Faster processors, which would be found in any substantial parallel machine, should experience even less impact. Given the heavyweight nature of this operating system and the NFS file system, these results corroborate those in the previous section.

Table 4: Execution time of a synthetic parallel computation, in seconds. In the "No I/O" case, this application runs alone, and represents the ideal execution time for this application. In the other cases one of the nodes is burdened with heavy NFS traffic. "Efficiency" represents the performance relative to the ideal execution time.

	Execution time (sec)	Efficiency
No I/O	130.8	
Sequential read	219.9	59.5%
Random read	181.4	72.1%
Sequential write	135.1	96.8%
Random write	193.8	67.5%

[4] For more information see `http://www.cs.dartmouth.edu/research/fleet/`.

[5] We used MPI [Wal94] for the communication support.

5 Discussion and conclusions

Large multiprocessors with many processors and disks have great potential for fast computations and high I/O throughput. Since they typically cost a lot of money, it is important to utilize the resources efficiently. To provide the high-performance I/O needed by some applications, many multiprocessors today dedicate a subset of their nodes to I/O. Our results show that for some complex file-request patterns, these dedicated nodes were saturated. For many simpler patterns, however, the I/O-node CPUs were largely idle, that is, with 80–95% available that could be used for running applications. Furthermore, even applications that synchronized at a barrier every 5 msec could profitably obtain about 89–93% of the I/O node's CPU time for computation. Disk-directed I/O usually needed less CPU time than a traditional caching file system. Measurement results from a real file system on a cluster of workstations corroborated these results.

We therefore encourage the development of parallel operating systems that do not enforce a static partitioning of I/O nodes and compute nodes, and of applications that are willing to run all or in part on the I/O nodes of a system. Clearly, applications that can adapt to changing load conditions are best suited to run in this environment.

Future work. We have only considered the impact of I/O service on the CPU utilization of an I/O node. File-I/O traffic may also substantially impact the communication performance of a computation-only application, depending on the nature of the network interface, so further study is required.

Acknowledgements

Thanks to Eric Brewer and Chrysanthos Dellarocas for Proteus, and to Nils Nieuwejaar, Bill Nitzberg, and Mike Harry for feedback on drafts of this paper.

Availability

The STARFISH simulator can be found at `http://www.cs.dartmouth.edu/~dfk/STARFISH/`

Information about Dartmouth's FLEET lab can be found at
`http://www.cs.dartmouth.edu/research/fleet/`

Many of these papers can be found at `http://www.cs.dartmouth.edu/pario.html`

References

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, 1991.

[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*, pages 10–20. IEEE Computer Society Press, October 1994.

[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.

[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.

[CFPB93] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra Johnson Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, 1993.

[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.

[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.

[GL91] Andrew S. Grimshaw and Edmond C. Loyot, Jr. ELFS: object-oriented extensible file systems. Technical Report TR-91-14, Univ. of Virginia Computer Science Department, July 1991.

[HdC95] Michael Harry, Juan Miguel del Rosario, and Alok Choudhary. VIP-FS: A virtual, parallel file system for high performance parallel and distributed computing. In *Proceedings of the Ninth International Parallel Processing Symposium*, April 1995. To appear.

[HER+95] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. Technical Report UIUCDCS-R-95-1903, University of Illinois at Urbana Champaign, January 1995.

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

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[Kri94] Orran Krieger. *HFS: A flexible file system for shared-memory multiprocessors*. PhD thesis, University of Toronto, October 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.

[LZCZ86] Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenepoel. File access performance of diskless workstations. *ACM Transactions on Computer Systems*, 4(3):238–268, August 1986.

[Mas92] Parallel file I/O routines. MasPar Computer Corporation, 1992.

[MCD+91] Evangelos Markatos, Mark Crovella, Prakash Das, Cezary Dubnicki, and Tom LeBlanc. The effects of multiprogramming on barrier synchronization. In *Proceedings of the 1991 IEEE Symposium on Parallel and Distributed Processing*, pages 662–669, 1991.

[MS94] Steven A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[Nit94] Bill Nitzberg. Time between barriers. Personal communication, 1994.

[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.

[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.

[SW94] K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In *Proceedings of Supercomputing '94*, pages 650–659, November 1994.

[Wal94] D. W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, April 1994.