Performance Measurement of a Parallel Input/Output System for the Intel iPSC/2 Hypercube

James C. French
Terrence W. Pratt
Mriganka Das

IPC-TR-91-002
January 30, 1991

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, Virginia 22903

This research was supported in part by Jet Propulsion Laboratory Contract #957721 and by the Department of Energy under grant DE-FG05-88ER25063.

This paper to appear in the Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement & Modeling of Computer Systems, May, 1991.

Performance Measurement of a Parallel Input/Output System for the Intel iPSC/2 Hypercube

James C. French, Terrence W. Pratt, Mriganka Das

Institute for Parallel Computation
Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, VA 22903

Abstract

The Intel Concurrent File System (CFS) for the iPSC/2 hypercube is one of the first production file systems to utilize the declustering of large files across numbers of disks to improve I/O performance. The CFS also makes use of dedicated I/O nodes, operating asynchronously, which provide file caching and prefetching. Processing of I/O requests is distributed between the compute node that initiates the request and the I/O nodes that service the request. The effects of the various design decisions in the Intel CFS are difficult to determine without measurements of an actual system. We present performance measurements of the CFS for a hypercube with 32 compute nodes and four I/O nodes (four disks). Measurement of read/write rates for one compute node to one I/O node, one compute node to multiple I/O nodes, and multiple compute nodes to multiple I/O nodes form the basis for the study. Additional measurements show the effects of different buffer sizes, caching, prefetching, and file preallocation on system performance.

1. Introduction

Multicomputers, with many processing nodes connected through a high-speed interconnection network, are rapidly becoming a common hardware platform for high-speed scientific computation. In such systems, each node consists of a processor-memory pair together with specialized hardware for message passing across the interconnection network. The Intel Scientific Computers iPSC/2 and iPSC/860 and the nCUBE 2 are typical machines in this category.

Scientific computations often require access to large data files, for initial input, for final output, and for scratch storage during a computation. High-performance concurrent file systems for multicomputers have been available only since about 1988. The Intel Concurrent File System (CFS) discussed in this paper is one of the first delivered concurrent file systems in which read/write operations on a single file can occur in parallel.

The design choices in a concurrent file system affect the performance of the system in ways that are not well understood. Unlike a conventional sequential file system, where there is a single processor requesting data from one or more I/O channels, a multicomputer system such as the Intel CFS may have M compute nodes making concurrent I/O requests to N I/O nodes, each controlling a channel to one or more I/O devices. Our goal in this paper is to investigate the performance implications of some of the key design choices made in the Intel CFS.

1.1. Background

Input/output subsystems that utilize many inexpensive magnetic disks accessed in parallel (in contrast to the traditional reliance on a single large expensive disk, the so-called SLED design [PATT88]) have been a topic of a number of recent papers. Some of the central issues in the design of such I/O subsystems include:

1. *Architecture of the I/O subsystem.* How should the hardware of the I/O subsystem be organized? Should the disks be accessed synchronously or asynchronously? [KIM86, REDD90, SALE86] Should all disks be on a single channel, or on multiple channels? How should processing of I/O requests be distributed? [PIER89, REDD89] How should reliability be achieved? [PATT88]

2. *Disk caching.* Use of memory in the I/O processor as a cache for disk blocks is a common method for smoothing the large speed difference between physical disk devices and main memory. Can such ‘‘disk caching’’ be used effectively for prefetching input blocks or for staging output blocks in a concurrent I/O system? [KOTZ90]

3. *File storage structure.* How should the file system be organized in a concurrent I/O system? Should individual files be ‘‘striped’’ or ‘‘declustered’’? [CROC89, LIVN87, SALE86] What is the appropriate unit of decomposition—bits, bytes, or larger blocks? How should the overall file system be organized? [KOTZ90, PIER89]

Almost all the research on these design alternatives has been limited to simulation studies. A few preliminary studies that involve measurement of early versions of the Intel CFS are presented in [ASBU89, BRAD89, PIER89, PRAT89]. Our study here is based on the production software provided by Intel for the system. To our knowledge, this study represents the first published performance measurements on the production system.

The Intel CFS represents one slice from the range of design choices mentioned above. In the Intel CFS, small groups of disks (1–4) are attached to dedicated ‘‘I/O nodes’’. I/O nodes and their disks work entirely asynchronously from other I/O nodes. Memory on the I/O nodes is used for disk caching. Prefetching and ‘‘preallocation’’ are used to speed I/O operations. Individual files are split into 4KB[1] blocks which are ‘‘declustered’’ [LIVN87]

[1] In this paper, K = 2^{10} (1024), not 10^3 (1000).

and spread across all the available I/O nodes and disks. Processing of I/O requests is distributed between the ‘‘compute node’’ that initiates the request and the I/O node that services the request. I/O requests from any compute node to any I/O node proceed concurrently and asynchronously.

1.2. Organization of This Paper

In the next section, we outline the methodology used to evaluate the Intel CFS, paying particular attention to careful definition of terms such as ‘‘transfer rate’’ that are basic to the study. In Section 3, we describe the organization of the Intel IPSC/2 and Concurrent File System in enough detail to allow an understanding of the design. Section 4 presents the results of our performance studies, organized around design decisions made in the I/O subsystem and the CFS software. Section 5 summarizes the lessons learned.

2. Methodology for Evaluation

This study focuses on the overall behavior of the I/O subsystem as seen from an applications program. The performance characteristics of software systems such as the Intel CFS are transitory. Each software release or change in the hardware is likely to affect the results of performance studies. However, unless there is a major rewrite of the software, the overall behavior of the system remains in the same range. Our detailed performance measurements "snapshot" the performance of our particular configuration of the system with our particular release of the CFS software. The observed experimental values may not accurately reflect the detailed performance of other configurations, but we believe that the overall characteristics of system behavior described and measured should be common to all CFS configurations.

In studying I/O performance, surprisingly much information can be gleaned from observing a few parameters of the system. For this study we looked at the behavior of the Intel CFS from the viewpoint of a scientific applications program running in parallel on the nodes of the machine and asked questions about what the expected CFS behavior might be from this vantage point. Note that we are not asking about the overall performance of the CFS from a more global perspective, such as its overall effectiveness for multiple users. Nor are we concerned with issues such as reliability and recoverability of the file system in the event of failures.

The I/O requirements of large scientific programs usually include both low volume data transfers (formatted character data) and high volume transfers (unformatted binary data). Sequential access is the predominant mode of file access. We focus here on the high volume I/O component: sequential access to very large data files (1MB—1GB). Typical data files might be satellite imagery, measurement data from experiments, or graphical output from simulations.

We use the term *buffer size* to refer to the size of the block of data involved in a read/write request by an application program. For high volume I/O, large buffer sizes (4 KB – 500KB) are often used in I/O requests. These requirements are distinctly different from other computing environments such as personal workstations (small files, small buffer sizes) and commercial applications (large files, small buffer sizes, random access).

For this study, we target I/O behavior typical of large scientific programs. Files are read/written in sequential portions, where each of several application processes may be concurrently reading/writing disjoint portions of the same file. The files are large (at least 1 MB in most cases), and the buffer size used in the application is also large (buffers of at least 4 KB and always a multiple of the physical block size used by the file system). Where the I/O system has several modes of operation, we are concerned primarily with just the modes that offer the highest I/O performance. Our measurement programs generate I/O requests at a rate sufficient to saturate the I/O system, with little or no computation between requests. This means our programs are entirely I/O bound in general.

For a single application program, the pervasive questions are "how fast can I move my data into and out of the file system?" and "how should I organize the I/O operations within the program to achieve the most effective performance?" Because of the multicomputer environment, the application program may itself be running on many compute nodes. We assume that from 1 to 32 nodes (the size of our machine) may be involved in I/O for this study. This environment broadens the questions of interest, because K nodes may be involved in reading/writing a single data file concurrently. Now it is also of interest to ask "how should the I/O operations be distributed among the computational nodes for most effective performance?"

Let’s consider first the view from a parallel process running in a single computational node. The primary measurement of interest is the *data transfer rate* observed for that process. For multiple nodes, we also are concerned with *aggregate* rates. These terms need careful definition.

Suppose that we have p processes reading (writing) a file of N bytes in parallel, where each process resides on a separate processor. Each process i reads (writes) n bytes in time t_i where $n = \frac{N}{p}$. The *individual* data transfer rate of a particular processor i is given by $r_i = \frac{n}{t_i}$. The *average* individual data transfer rate is given by $\bar{r} = \frac{1}{p}\sum_{i=1}^{p} r_i$.

There are at least two reasonable measures of the *aggregate* data transfer rate of the p processors. In the first case, we sum the data rates of the individual processors. This gives rise to the quantity $\sum_{i=1}^{p} r_i$ called the *maximum sustained aggregate rate* (max_SAR). We call this measure the ‘‘maximum’’ rate because, by construction, it assumes that each processor i contributes a rate r_i and all processors contribute during the same time interval. From the definition of $\bar{r}$ above, we see that

$$\text{max\_SAR} = \sum_{i=1}^{p} r_i = p\bar{r} \ . \qquad (1)$$

In the second case, we consider that all N bytes move through the system in $\tau = \max_i t_i$ time units. That is, the entire file is not transferred until the slowest processor finishes reading (writing) its partition of the file. This gives rise to the quantity $\frac{N}{\tau}$ called the *minimum sustained aggregate rate* (min_SAR). We call this a ‘‘minimum’’ rate since this is the rate that an outside observer will perceive as the rate at which the entire processor ensemble is operating. From the definitions above, we see that

$$\text{min\_SAR} = \frac{N}{\tau} = \frac{np}{\max t_i}$$

$$= np \min\left(\frac{1}{t_i}\right) = p \min\left(\frac{n}{t_i}\right) = p \min r_i \ . \qquad (2)$$

Note that min_SAR ≤ max_SAR with equality when $t_i = \tau$ for all i. Both min_SAR and max_SAR are useful measures of the

absolute performance of an I/O subsystem.

3. The Intel iPSC/2 and Concurrent File System

The Intel iPSC/2 system used in this study is composed of 32 compute nodes interconnected in a hypercube geometry. A ''host node'' provides the interface to the outside world. The host node manages its own disk (separate from the CFS-managed I/O subsystem) and is the target for low-volume I/O from applications programs running on the compute nodes. Each of the compute nodes is actually a composite of two processors, one for computation (an 80386 chip) and one for handling message traffic. The message co-processor assures that node-to-node communication is independent of the number of links traversed, if there is no contention for those links. Each compute node has 4 MB of memory.

There are 4 I/O nodes in the system, each connected to one port of a different compute node. Each I/O node has 4 MB of internal memory and is connected to one port of a SCSI bus which serves as the conduit to a Maxtor disk with around 330 MB of memory available for user files. Up to four I/O devices can be connected to each bus; our system has only a single disk/bus. Figure 1 illustrates the architecture. The I/O nodes are connected to the compute nodes 2,6,10 and 14 in the system. Each physical disk corresponds to a logical ''volume'' in the file system. Thus there are four volumes in this system. Each I/O node uses a part of its internal memory for a disk cache, in which it can temporarily store blocks whose permanent copies are in the associated volume.

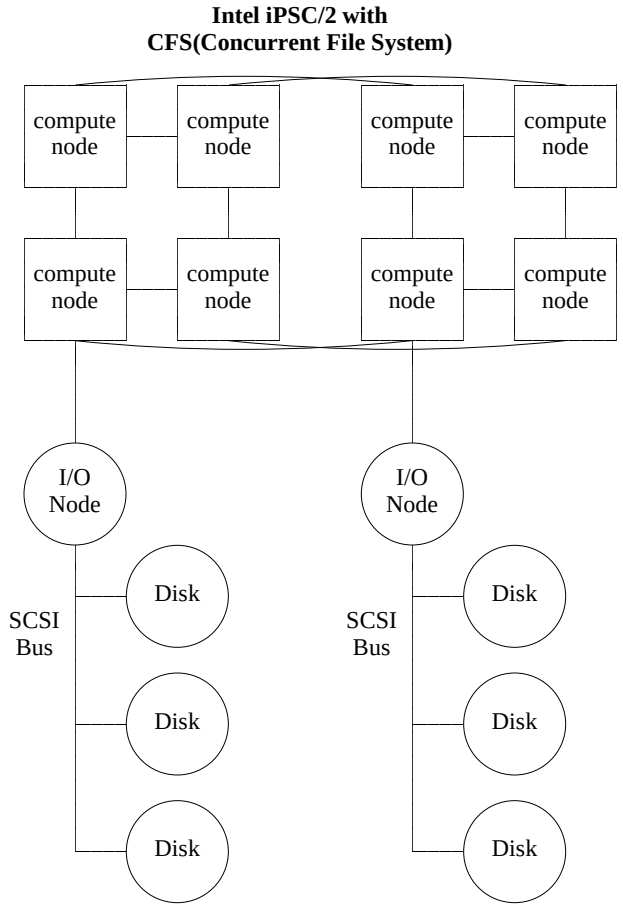


Figure 1

3.1. File Organization

The CFS maintains the UNIX view of a file from an application program. Each file has a file body and header. Short files are packed into the header. For larger files, block locations are tracked by a tree-structured index whose root is in the header block. These index blocks are termed *indirect blocks*. The depth of the tree is unrestricted.

The basic unit of file allocation and data transfer in the CFS is a block of 4096 bytes. Paul Pierce, the designer of CFS, describes this block size as the smallest block size where the request/response performance over the communication network matches the sustained disk transfer rate of about 1MB/sec. [PIER89] He also notes that a larger block size would increase contention on communication links and increase internal fragmentation when blocks are not filled completely.

Each file is split into 4KB blocks when it is initially created. A file can be made to reside on one or more volumes. When a file is on multiple volumes it is ''declustered,'' — the blocks of the file are allocated in round-robin manner among the allowable volumes (unless one of them runs out of space in the course of allocation, in which case it is subsequently bypassed). A file is spread across all available volumes by default. However the *restrictvol* system call may be used to restrict a file's allocation to certain volumes.

3.2. File System Interface

The user interface to the CFS follows the standard UNIX design, with some extensions. The pathname component "/cfs" is what distinguishes CFS files from those on the disk managed by the host node. For most operations on CFS files standard UNIX I/O calls are used. *Lsize* is used to preallocate blocks to a file before writing them. *Restrictvol*, as mentioned earlier, allows the user to exercise control over which volumes are used to store the blocks of the file. It also allows the user to explicitly control where the header (and the indirect blocks) reside. *Cread* and *cwrite* are high performance synchronous read/write commands which are used by our measurement programs.

3.3. Distributed Processing of I/O Requests

The CFS software is distributed among the compute nodes and the I/O nodes. Each I/O node runs a server process called a *disk process* whose main function is to send and receive file blocks in response to read/write requests from the compute nodes. One I/O node also runs a second server process called the *name process* whose major function is to handle the directory tree and to respond to I/O requests that involve manipulation of directory entries. In CFS the entire directory tree is represented as a single file rather than as a tree of separate files.

Processing of I/O requests is distributed between the compute node running the application program, where the request is initiated, and the I/O nodes themselves. An application program process that makes I/O requests is linked to a runtime I/O library routine on the compute node. The library routine interprets the I/O request, breaks it into smaller I/O service requests, and sends the service requests as messages to the appropriate I/O nodes. This organization allows concurrency between the interpretation of the file structure and the actual handling of block fetches or stores. Since file structure interpretation is done on each compute node, each compute node has to store the header and indirect blocks to avoid getting them from the servers upon each request. No file data blocks are cached at compute nodes between I/O operations.

When a read or write operation is executed by an application process on a compute node, the call activates the appropriate library routine. If the data involved resides within a single block of the file, the library routine sends an I/O request to the I/O node responsible for that block. If the data crosses the boundaries of one or more file blocks, then the library routine breaks the I/O request into requests for individual blocks (whole or partial) and sends each request to the appropriate I/O node. Thus a single read/write operation by an application program may spawn more than one I/O request from its compute node to various I/O nodes. In all cases, at most one block (4 KB) is transferred per I/O request received by an I/O node.

At each I/O node, the disk process manages a large (1 MB) cache for file blocks. When reading a file, blocks are prefetched from the disk into the cache. When writing a file, blocks are deposited into the cache as they arrive from compute nodes, and then they are moved from the cache to the disk itself as a background activity of the I/O node.

4. CFS Design Decisions and Their Performance Implications

The organizing principle in the sub-sections that follow is to explore the effects of several key design decisions on the performance of the CFS. In this way the CFS behavior can be characterized within the context of the design framework.

Recall that the CFS software is organized as a single name process and one disk process per I/O node. The physical placement of these processes requires that the name process occupy the same node as one of the disk processes. Throughout the measurement process, we sought to minimize (or eliminate) the effects of the name process by concentrating on read/write activity rather than open/close activity. We have measured the cost of opening (closing) a file to be approximately 30 ms (1.5 ms). As long as this cost is reasonable, it is largely irrelevant in our perceived application domain.

In the following sections, our concern is primarily to measure the maximum achievable I/O rates in the system. Our measurement programs are designed to saturate the I/O system with I/O requests, with little or no computational load. In the CFS, ''preallocation'' of the blocks of an output file always leads to higher output rates, so all output files are preallocated for the tests discussed below (Section 4.5 provides details on the performance benefits of preallocation).

4.1. Single Compute Node to Single I/O Node

Our first measurements are used to determine the point-to-point I/O rates achievable in the absence of contention in the system. The tests use a single application process on a compute node interacting with a single I/O node. The same studies are used to show that I/O rates are unaffected by the number of inter-node ''hops'' that I/O requests must make in going from a compute node to an I/O node. Recall from Figure 1 that the I/O nodes of the CFS are directly connected to individual compute nodes. On the machine used for these measurements, the four I/O nodes are attached to compute nodes 2, 6, 10, and 14.

To measure the greatest achievable transfer rate from each node, we placed a single reader (writer) in turn on each node of a quiescent cube and measured the time required to read (write) files of 1 MB and 8 MB. These two file sizes were chosen to help assess any effects due to disk cache size. The disk processes each use about 1 MB of memory for the disk block cache. Thus for a writer, the 1 MB file can often be written from memory to memory, while the 8 MB file must involve actual disk accesses. To restrict interactions to a single I/O node, the files were restricted to lie wholly on a single disk volume. The results of these measurements are shown in the plots of Figure 2. These measurements show the point-to-point transfer rate attainable when a single reader (writer) reads (writes) a file to a single I/O node.

A quick glance at the plots shows that the placement of a reader (writer) within the compute nodes has no effect on the attainable point-to-point data transfer rate, so the number of hops to the I/O node has no observable effect.

There are several other interesting observations which can be made. Consider first the write behavior as shown in Figures 2(c) and 2(d):

1. The data transfer rate for the 1 MB file is approximately 1240 KB/sec. The maximum point-to-point communication transfer rate of the iPSC/2 is specified by Intel [PIER88] to be 2800 KB/sec and has been measured by others [BOMA89] at 2660 KB/sec. Thus this output rate is slightly less than 50% of the basic message-passing communication rate.
2. The data transfer rate for the 8 MB file is 750–800 KB/sec or 610 KB/sec depending on where the file header is relative to the file body. The higher rate is achievable when the header resides on a different volume than the file body; when the header and body are on the same volume, contention between accesses to indirect blocks and writes to the file result in the lower figure.
3. The lower overall transfer rate associated with the 8M file is due to the physical disk activity necessary to flush cache blocks to disk as the cache fills.

Now consider the read behavior as shown in Figures 2(a) and 2(b):

1. Again we see two modes of behavior for the 8 MB file. The data transfer rate (Fig. 2(b)) is 670–685 when the file header is on a different volume than the body of the file and 660–675 when the header and body reside on the same volume. Notice that the magnitude of the difference is smaller for reads (10–15 KB/sec) than for writes (140–190 KB/sec).
2. The data transfer rate for the 8 MB file is slightly higher than for the 1 MB file. This apparently counterintuitive effect is due to a large initial startup cost. Our data shows that while a typical read transfers 4 KB in 3–4 ms, one or two early I/O operations incur times of around 100 ms. The larger file transfers 8 times as many blocks so the effect of these large spikes is amortized over a larger number of I/O operations.

A summary of these measurements is shown in Table 1. As can be seen, the write rates are usually higher than the read

CFS Point-to-Point Data Transfer Rates (KB/sec)

Operation	File Size		
	1 MB	8 MB	
		Normal	Header separate
Write	1240	610	775
Read	650	670	680

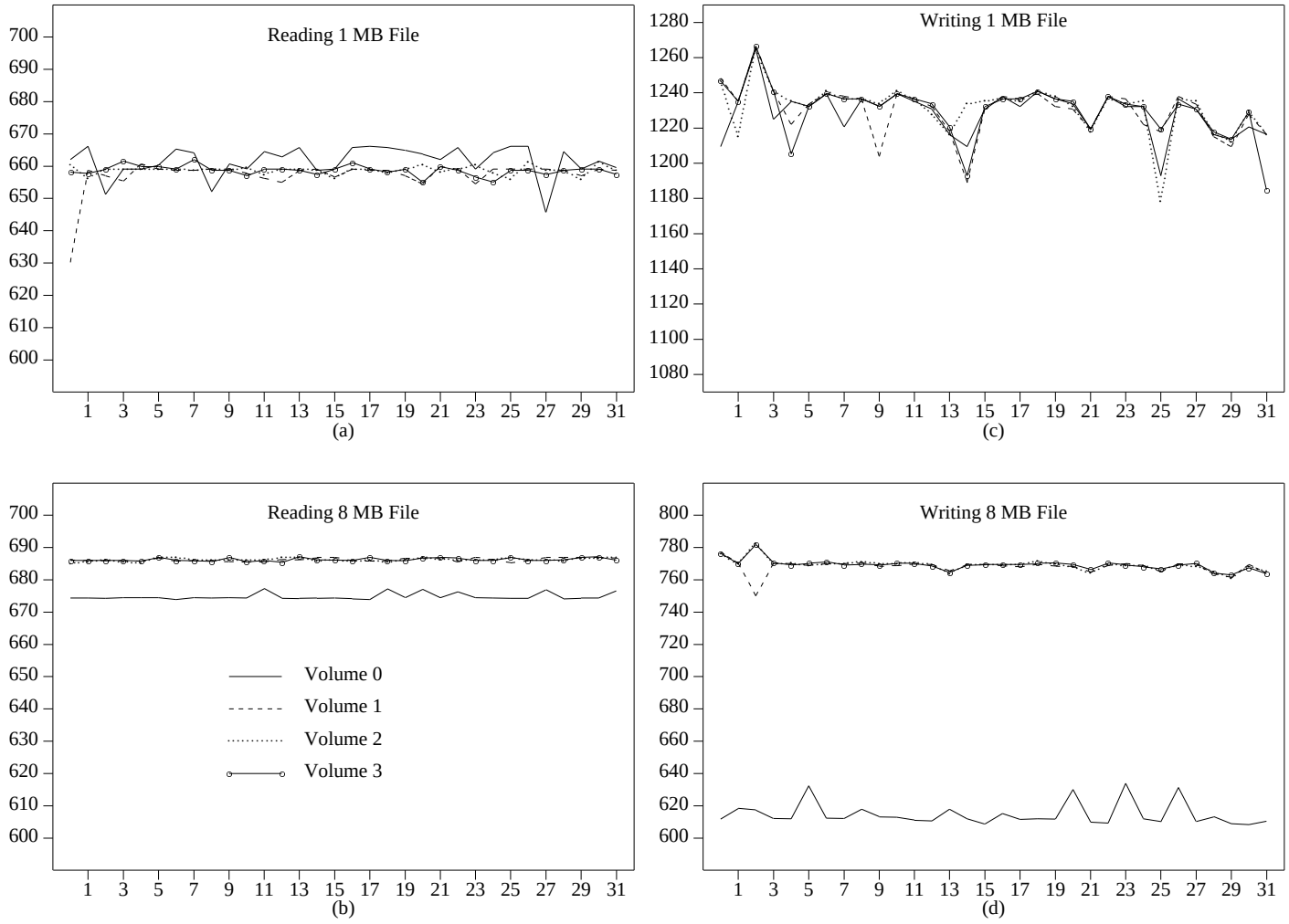
Table 1

rates. This is explained by the fact that when writing, almost all of the physical disk I/O can often be overlapped with the

Figure 2

Point-to-Point Transfer Rates Reading(Writing) a 1(8) MB File

File body continuously restricted to volume zero
File header restricted to each volume in turn (see legend in (b))
X-axis : Compute Node, Y-axis : Transfer Rate (KB/sec)



movement of data from the application to the disk process. Readers incur the wait for physical reads far more frequently. For example, when a file is small enough to fit entirely in the disk process cache, a writer will effectively be doing memory to memory copies. However, a reader must wait, at least initially, for disk blocks to be brought into the cache.

4.2. Single Compute Node to Multiple I/O Nodes

The last section confined its discussion to files that had been restricted to a single volume. The plots of Figure 3 show the transfer rate achievable by a single reader and a single writer respectively when the file is fully declustered. The transfer rate increases with increasing file size until reaching a steady state.

Our system has four I/O nodes. We can use *restrictvol* to simulate a system with from one to four I/O nodes to gauge the effect on system performance of increasing the number of I/O nodes. A second experiment involved a single writer writing an 8 MB file four separate times, to a file spread across 1, 2, 3, and 4 volumes. This is logically equivalent to writing the file using only 1, 2, 3, and 4 I/O nodes respectively. The results of these tests are shown in Table 2.

Transfer Rate as Number of I/O Nodes Varies (KB/sec)				
Number I/O Nodes	Reading		Writing	
	4 KB buffer	16 KB buffer	4 KB buffer	16 KB buffer
1	683	684	698	713
2	935	1314	727	975
3	957	1480	1139	1760
4	962	1492	1179	1764

Table 2

Table 2 indicates that declustering gives substantial improvement in transfer rates for a single reader/writer. When either reading or writing, three I/O nodes are needed to provide maximum transfer rates to a single compute node. Moving to four I/O nodes provides only a small additional gain.

4.3. Multiple Compute Nodes to Multiple I/O Nodes

One of the major potential advantages of the CFS architecture is that multiple application processes, running on different compute nodes, can be performing concurrent I/O operations on

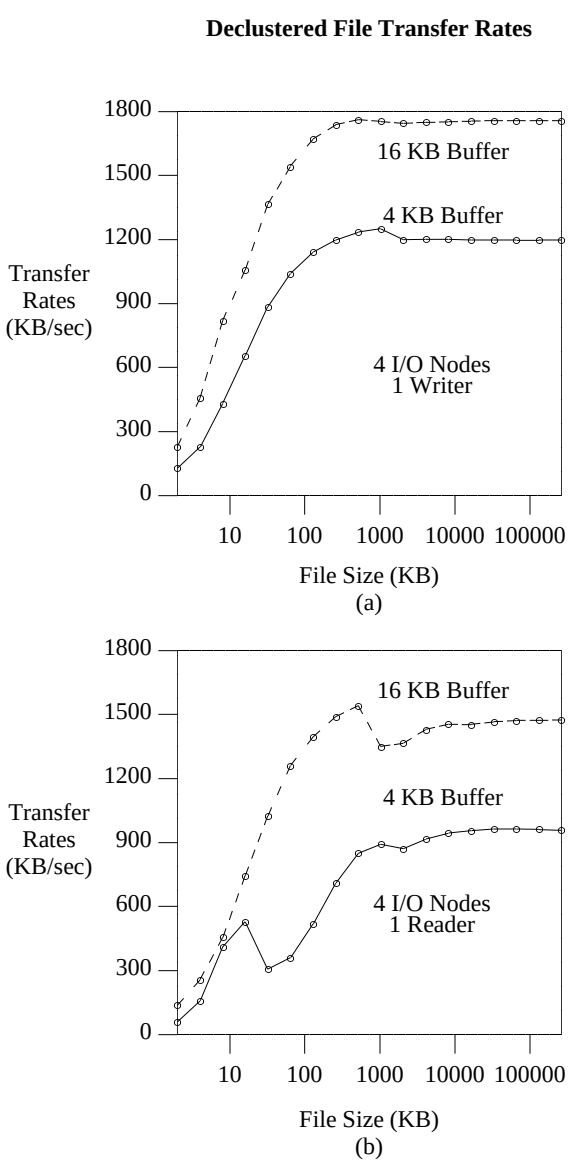


Figure 3

the same file (or different files at the same time). Figure 4 shows the transfer rate obtained for an 8 MB file as the number of concurrent readers/writers varies from 1 to 16. All four I/O nodes are involved in handling the I/O requests. The lower part of the graphs shows that contention for I/O services reduces the read/write transfer rate available to individual nodes as the number of concurrent workers increases. The aggregate rates (maxSAR and minSAR) shown in the upper graphs indicate that the I/O nodes become saturated when the number of concurrent readers/writers reaches about four. After that point additional readers/writers do not increase the aggregate rates. The rather large variation in the aggregate write rates remains unexplained (the same behavior is exhibited over many runs).

Overall, increasing the number of concurrent readers/writers increases the aggregate transfer rate, from the approximately 1500 KB/sec available to a single reader/writer to an aggregate rate of 2500 KB/sec for reading and approximately 3000 KB/sec for writing. The apparent saturation of the I/O nodes with more than four readers/writers indicates that increasing the number of I/O nodes would increase the number of concurrent readers/writers that could be effectively employed, with an attendant increase in the aggregate transfer rate.

4.4. Effects of Buffer Size on Transfer Rate

To maximize I/O rates, an application might choose a large buffer size in making I/O requests. However, in this setting, a larger buffer size does not translate into a larger data transfer

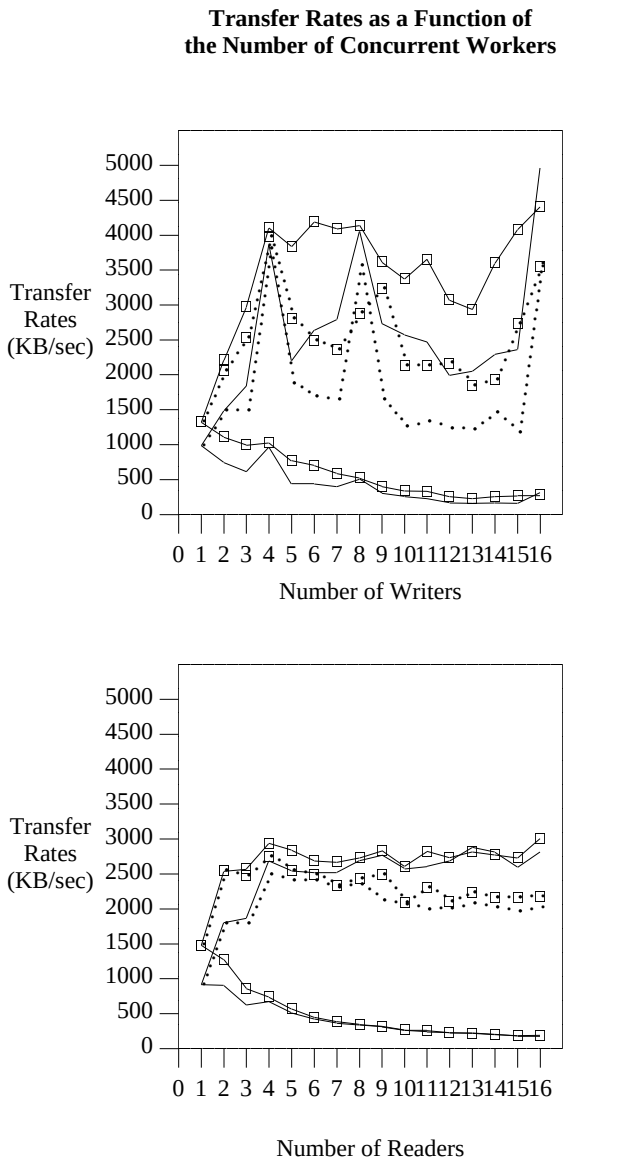


Figure 4

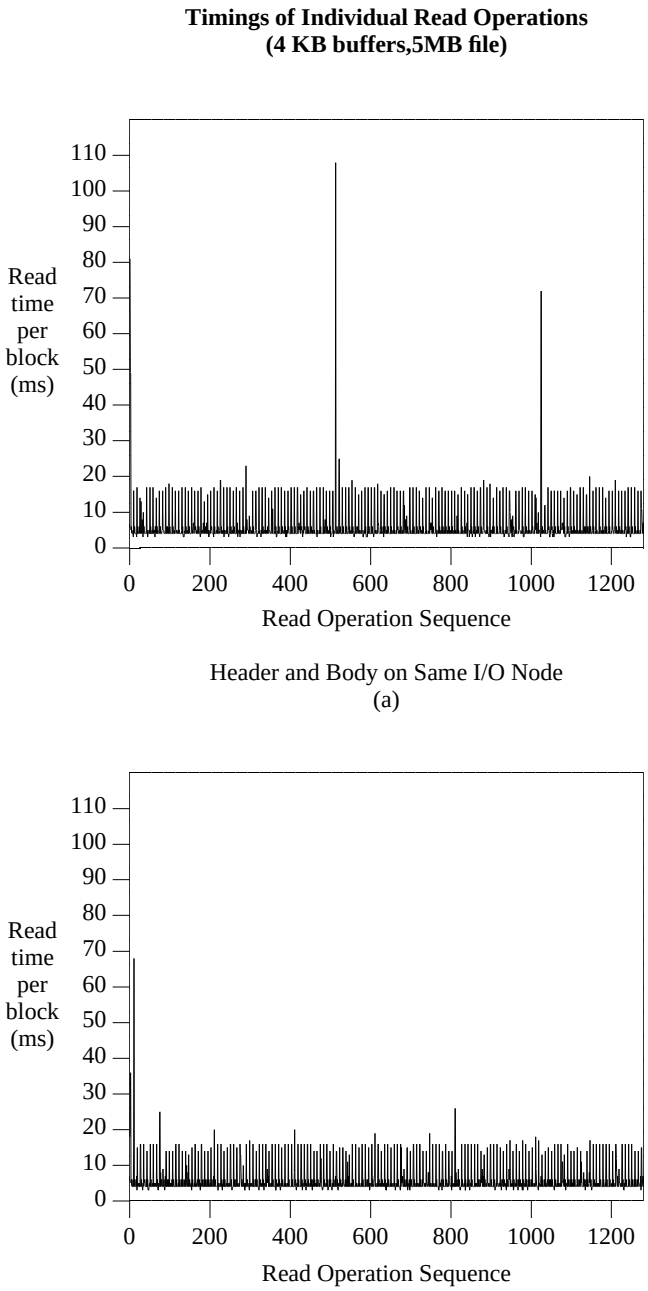


Figure 5

between an I/O node and the compute node. Recall that the CFS decomposes large buffer reads/writes into 4 KB (or smaller) pieces — the block size of the file system. This enables the asynchronous processing of the entire I/O request. Using a larger buffer size in an I/O request allows a single entry to the library routine on the compute node to spawn a larger number of 4 KB I/O requests, spread across multiple I/O nodes. To measure the effect of larger buffer sizes, we measured I/O rates with both 4 KB and 16 KB buffer sizes. Table 2 shows that 16 KB buffers achieve about 1.5 times the data rate of 4 KB buffers when a single reader (writer) is reading (writing) a file.

Earlier measurements of the effects of buffer size on transfer rate [FREN89] showed that most improvement occurred when the buffer size was increased from 4 KB to 16 KB. After that essentially no further improvement is observed. With our four I/O node system, a 16 KB buffer translates into four 4 KB block transfer requests, one from the compute node to each of the four I/O nodes. With larger buffer sizes, multiple 4 KB requests would be directed to each I/O node, and the speed of the I/O node would become the limiting factor. It appears that a system with more I/O nodes would allow larger buffer sizes to be more effective.

4.5. Effects of Caching and Prefetching on Read Operations

Caching and prefetching of disk blocks have an important effect on read performance. In this section, we explore further the ramifications of the CFS caching and prefetching design.

The read behavior of the CFS can be characterized by a spectral analysis of the timing of individual read operations as an entire file is read. Figures 5(a) and 5(b) demonstrate the two characteristic behaviors exhibited by the CFS. The figures show the read times for 4 KB buffers as an entire file is read. That is, if the k^{th} read operation takes t ms, then the k^{th} vertical line in the figure is proportional to t. The data file has been restricted to a single CFS volume.

There are three separate frequency components, each with a characteristic amplitude. The components are:

1. The lowest amplitude component (3–4 ms) represents the effect of a cache hit, where the block is simply moved from the disk process cache to the compute node.
2. The next component has approximately a 15 ms amplitude. It occurs every 8 read operations and represents the time associated with CFS prefetch activity. The first read request causes up to 8 blocks to be prefetched by the disk process. After 4 blocks have been read by the application, the CFS initiates another prefetch.
3. The highest amplitude component (70–105 ms) occurs every 512 reads in Figure 5(a) and only at the beginning of Figure 5(b). This component represents the cost of fetching indirect blocks. In Figure 5(a) the header is on the same volume as the body of the file. In this situation, any effort to prefetch the indirect blocks will be defeated when incoming data forces the blocks from the cache. Thus, when the next indirect block is needed it will not be in the cache and must be read from the disk, incurring the large delay. In Figure 5(b), the header resides on a volume distinct from the file body. In this situation, the indirect blocks are already in the cache of another I/O node and can be fetched directly as a memory-to-memory transfer, without waiting for a disk access.

Point-to-Point Data Transfer Rates
800 KB file
(percentages show improvement due to prefetching and caching)
(KB/sec)

Direction	Rate	Improvement
Reverse	208	—
Forward	661	217%
Reread	1037	57%

Table 3

Reverse Read and Reread of File

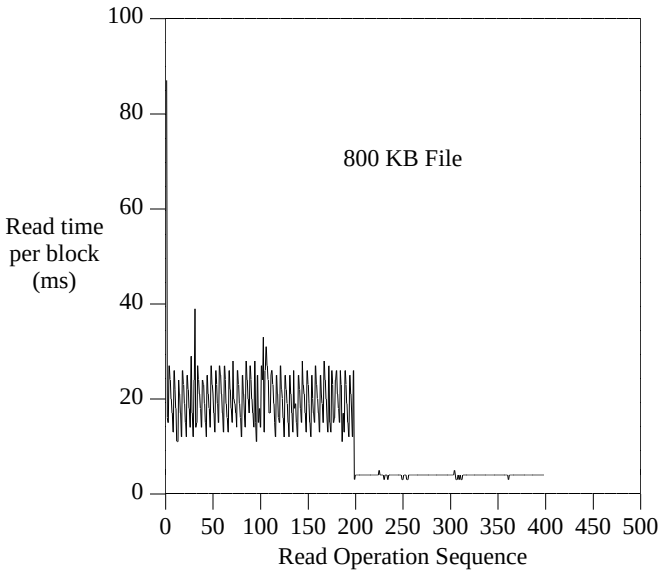


Figure 6

To help assess the effects of prefetching and caching on reads, we ran a test which read an 800 KB file in reverse order, then read it in forward order, and finally reread it in forward order. The time taken for each read operation was recorded and the resulting transfer rates calculated. These are shown in Table 3. By reading the file in reverse order we completely defeat the prefetch mechanism and are forced to the disk for every read request. When reading the file in the forward direction prefetching will improve performance. Note that care was taken to ensure that no blocks of the file remained in the cache before reading the file in the forward direction. Finally, after reading the file in the forward direction the file was again read in reverse — any direction would have had the same effect — and the effect of the cache is clearly seen. These effects are shown graphically in Figures 6 and 7.

Read and Reread of File

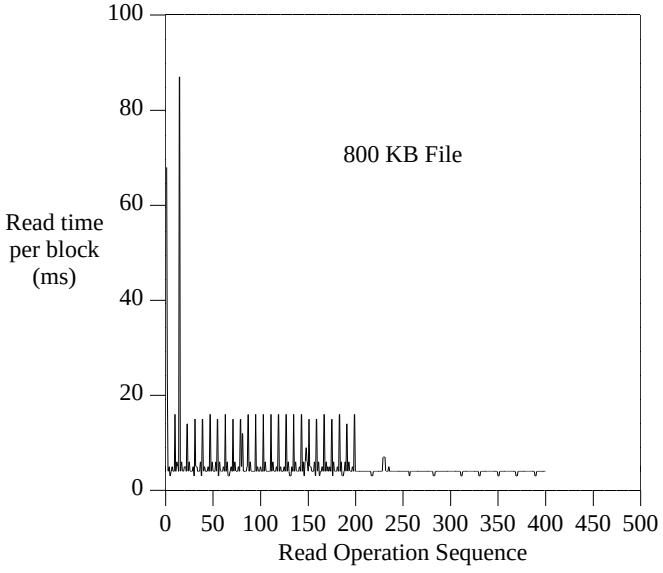


Figure 7

Figure 6 shows a mean block read time of about 20 ms when reading in reverse, dropping to less than 5 ms per block when the file is reread from the cache. Figure 7 shows the prefetch behavior described in the last section (small spike every 8 read operations) followed by a fairly constant block read time when the file is entirely in the cache.

4.6. Effect of File Preallocation on Write Operations

As we said earlier, to achieve maximum performance we used the CFS *lsize* call to preallocate files before writing them. In this section, we discuss the actual performance benefit accruing from that decision.

To examine the effects of preallocation on write times, we created, preallocated, and wrote an 8 MB file under two scenarios — first as a file restricted to one volume and then as a file declustered over all volumes. The results of these tests are summarized in the first two lines of Table 4. All times shown in the table are given in seconds and for the preallocated write times, the transfer rate is shown in parentheses.

Effect of File Preallocation on File Write Times (8 MB file) (KB/sec)

File Type	Preallocation Time	Preallocated Write Time		Unallocated Write Time
Restricted	.49	10.7	(748)	30.1
Declustered	.45	7.2	(1111)	30.2

Table 4

The table clearly shows that preallocation results in a major performance gain. The cost of preallocation is about 55–60 ms/MB and resulted in speedups 3–4 times the unallocated rate. Stated another way, the sum of preallocation time and the time to write a preallocated file is 1/4 – 1/3 the time to write the equivalent unallocated file.

4.7. An Experiment with Four Writers

We conclude this discussion of the behavior of the CFS with a simple experiment combining many of the dynamics of the CFS. This should be considered illustrative of the complex behavior of the CFS rather than an attempt to isolate some particular feature.

We assume that an application has four writers reading (writing) disjoint pieces of a 32 MB file. There are at least three obvious strategies that may be employed: (A) operate on a single 32 MB file declustered across all volumes, (B) assign one fourth of the file (8 MB) to each volume and restrict it to reside wholly on that volume, or (C) use four 8 MB files, one per writer, but spread all four files across all volumes. Option (A) uses the default behavior of the CFS as it was intended to be used. Option (B) simulates a situation where each reader (writer) has a dedicated I/O node (and disk). Option (C) simulates a situation where each reader (writer) has a dedicated file, but not a dedicated I/O node. The results of these measurements are summarized as Table 5.

Option (B), dedicated I/O nodes, provides the worst performance for both reads and writes. Option (A), the simple, direct use of the CFS operations, provides an 8% improvement for writes and a 7% improvement for reads. Option (C), separate files for each compute node, provides even better write performance, a 12% improvement, but only a 3% improvement in read performance. These results suggest that for large files and large buffer

	One 32 MB Declustered File (A) vs. Four 8 MB Restricted Files (B) vs. Four 8 MB Declustered Files (C) (Average transfer rate with 16 KB buffers) Percentages = improvement over option (B)			
	Write Rate (KB/sec)		Read Rate (KB/sec)	
A	690	(8%)	663	(7%)
B	645	(-)	621	(-)
C	723	(12%)	639	(3%)

Table 5

sizes, the basic CFS design provides balanced and effective use of the available hardware resources.

5. Conclusions

To summarize these results, we would conclude that:

(1) Declustering of large files in the CFS provides performance improvement for a single reader/writer. Depending on buffer size, improvements for declustering across four I/O nodes (disks) are measured at 40–120% for reading and 70–150% for writing.

(2) With multiple reader/writers, declustering and concurrent access provide additional performance improvements until the I/O nodes become saturated. For our system, the improvements provided by concurrent access to four I/O nodes were measured at about 70% for reading and 100% for writing. Saturation occurred with about four readers/writers. After saturation, additional concurrent readers/writers experienced degraded performance, with the overall aggregate transfer rate from the I/O subsystem remaining approximately constant.

(3) The choice of buffer size used by an application program in an I/O request affects the I/O transfer rate, in spite of the fact that local processing of an I/O request on a compute node splits the request into 4 KB or smaller block requests. Larger buffers improved performance within a certain range, with about a 50% improvement measured between 4 KB buffers and 16 KB buffers. Beyond 16 KB, larger buffers did not affect performance in a four I/O node system. The number of I/O nodes appears to be the critical parameter here.

(4) Disk caching and prefetching on the I/O nodes improved performance of read operations. For a 4 KB read request, a cache hit allowed the operation to complete in 3–4 ms. Prefetching caused about every 8th read operation to take about 15 ms. When indirect blocks had to be fetched from the disk, the read operation might take more than 70 ms. In our tests, prefetching during sequential reads provided about a 220% performance improvement over random access, and caching provided an additional 60% improvement.

(5) Preallocation of file blocks before beginning a sequence of write operations provided major performance improvements, measured at 300–400%.

(6) In a comparison of I/O strategies for applications programs, we compared the use of a single large declustered file accessed by four processes concurrently, four smaller files restricted to single disks and accessed only by a single application process, and four smaller declustered files accessed by single application processes. The single large declustered file provided the simplest interface and the best performance for reads; the four smaller declustered files provided the best performance for writes. The dedicated files option had the worst performance in all cases.

Acknowledgements

This work was supported in part by JPL contract #957721 and by the Department of Energy under grant DE-FG05-88ER25063. We also thank Brent Laster who helped collect and plot some of the measurements reported here.

References

[ASBU89] R. K. Asbury and D. S. Scott, ‘‘Fortran I/O on the iPSC/2: Is There Read after Write?’’, *Proc. 4th Conf. on Hypercube Concurrent Computers and Applications*, Monterey, Mar. 1989, 129-132.

[BOMA89] L. Bomans and D. Roose, ‘‘Benchmarking the iPSC/2 Hypercube Multiprocessor’’, *Concurrency: Practice and Experience 1*, 1 (Sep. 1989), 3-18.

[BRAD89] D. K. Bradley and D. A. Reed, ‘‘Performance of the Intel iPSC/2 Input/Output System’’, *Proc. 4th Conf. on Hypercube Concurrent Computers and Applications*, Monterey, Mar. 1989, 141-144.

[CROC89] T. W. Crockett, ‘‘File Concepts for Parallel I/O’’, *Proc. Supercomputing ’89*, Reno, Nevada, Nov. 1989, 574-579.

[FREN89] J. C. French and T. W. Pratt, ‘‘Performance Measurement of Two Parallel File Systems’’, Tech. Rep. IPC-TR-89-13 (*Supercomputing ’89* poster presentation), Institute for Parallel Computation, Thornton Hall, University of Virginia, Charlottesville, VA, Nov. 1989.

[KIM86] M. Y. Kim, ‘‘Synchronized Disk Interleaving’’, *IEEE Trans. on Computers C-35*, 11 (Nov. 1986), 978-988.

[KOTZ90] D. Kotz and C. S. Ellis, ‘‘Prefetching in File Systems for MIMD Multiprocessors’’, *IEEE Trans. on Parallel and Distributed Systems 1*, 2 (Apr. 1990), 218-230.

[LIVN87] M. Livny, S. Khoshafian and H. Boral, ‘‘Multi-Disk Management Algorithms’’, *Proc. of the 1987 ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, May 1987, 69-77.

[PATT88] D. Patterson, G. Gibson and R. Katz, ‘‘A Case for Redundant Arrays of Inexpensive Disks (RAID)’’, *Proc. of the Inter. Conf. on Management of Data*, Chicago, IL, June 1988, 109-116.

[PIER88] P. Pierce, ‘‘The NX/2 Operating System’’, *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications*, Pasadena, Jan. 1988, 384-390.

[PIER89] P. Pierce, ‘‘A Concurrent File System for a Highly Parallel Mass Storage Subsystem’’, *Proc. 4th Conf. on Hypercube Concurrent Computers and Applications*, Monterey, Mar. 1989, 155-160.

[PRAT89] T. W. Pratt, J. C. French, P. M. Dickens and S. A. Janet, ‘‘A Comparison of the Architecture and Performance of Two Parallel File Systems’’, *Proc.*

4th Conf. on Hypercube Concurrent Computers and Applications, Monterey, CA, Mar. 1989, 161-166.

[REDD89] A. L. N. Reddy and P. Banerjee, ‘‘An Evaluation of Multiple-Disk I/O Systems’’, *IEEE Trans. on Computers 38*, 12 (Dec. 1989), 1680-1690.

[REDD90] A. L. N. Reddy and P. Banerjee, ‘‘Design, Analysis, and Simulation of I/O Architectures for Hypercube Multiprocessors’’, *IEEE Trans. on Parallel and Distributed Systems 1*, 2 (Apr. 1990), 140-151.

[SALE86] K. Salem and H. Garcia-Molina, ‘‘Disk Striping’’, *Proc. of the Second Inter. Conf. on Data Engineering*, 1986, 336-342.