File-Access Characteristics of Parallel Scientific Workloads

Nils Nieuwejaar and David Kotz *
Apratim Purakayastha and Carla Schlatter Ellis †
Michael Best‡
Dartmouth Technical Report: PCS-TR95-263

March 29, 1996

Abstract

Phenomenal improvements in the computational performance of multiprocessors have not been matched by comparable gains in I/O system performance. This imbalance has resulted in I/O becoming a significant bottleneck for many scientific applications. One key to overcoming this bottleneck is improving the performance of parallel file systems.

The design of a high-performance parallel file system requires a comprehensive understanding of the expected workload. Unfortunately, until recently, no general workload studies of parallel file systems have been conducted. The goal of the CHARISMA project was to remedy this problem by characterizing the behavior of several production workloads, on different machines, at the level of individual reads and writes. The first set of results from the CHARISMA project describe the workloads observed on an Intel iPSC/860 and a Thinking Machines CM-5. This paper is intended to compare and contrast these two workloads for an understanding of their essential similarities and differences, isolating common trends and platform-dependent variances. Using this comparison, we are able to gain more insight into the general principles that should guide parallel file-system design.

Keywords: parallel I/O, file systems, workload characterization, file access patterns, multiprocessor file systems.

---

*Dartmouth College, email: {nils,dfk}@cs.dartmouth.edu.
†Duke University, email: {ap,carla}@cs.duke.edu.
‡M.I.T., email: mikeb@media.mit.edu; also affiliated with Thinking Machines Corporation.

This work was supported in part by the National Science Foundation under grant number CCR-9113170, the National Center for Supercomputing Applications, NASA Ames Research Center under agreement number NCC 2-849, and Thinking Machines Corporation.

1 Introduction

There is a growing imbalance between the computational performance and the I/O subsystem performance in multiprocessors. This imbalance has resulted in I/O becoming a significant bottleneck for many scientific applications. Thus, there is a clear need for improvements in the design of high-performance parallel file systems to enable them to meet the I/O needs of these applications.

To be successful, a system designer must possess a thorough understanding of how the system is likely to be used. Only with such an understanding can a system's policies and mechanisms be optimized for the cases expected to be most common in that system's workload. Designers have so far been forced to rely on speculation about how parallel file systems would be used, extrapolating from file-system characterizations of general-purpose workloads on uniprocessor and distributed systems or of scientific workloads on vector supercomputers.

To address this limitation, we initiated the CHARISMA project in June 1993 to CHARacterize I/O in Scientific Multiprocessor Applications from a variety of production parallel computing platforms and sites.[1] While some work has been done in studying the I/O needs of parallel scientific applications (typically by examining a small number of selected applications), the CHARISMA project is unique in recording individual read and write requests in live, multiprogramming, parallel workloads. We have so far completed characterization studies on an Intel iPSC/860 at NASA's Ames Research Center [1] and on a Thinking Machines CM-5 at the National Center for Supercomputing Applications [2]. On both systems we addressed a similar set of questions:

- What did the job mix look like: How many jobs were run concurrently? How many processors did each job use?
- How many files were read and written? What were their sizes?

[1] More about CHARISMA may be found at `http://www.cs.dartmouth.edu/research/charisma.html`.

- What were typical read and write request sizes, and how were they spaced in the file? Were the accesses sequential and, if so, in what way?

- What are the overall implications for parallel file-system design?

In this paper we address the final question by integrating results and observations across multiple platforms. To that end, we use the results from the two machine-specific studies to try to identify observations that hold across various multiprocessor platforms, and to pinpoint characteristics that appear to be specific to a single platform or environment.

In the next section we describe previous studies of multiprocessor file systems and file-system workloads, and we describe the two platforms examined in this study. In Section 3 we outline our research methods, and in Section 4 present our results. Section 5 draws some overall conclusions.

2 Background

In this section, we review many of the previous studies of file-system workloads and outline the basic design of some current multiprocessor file systems. Finally, we describe the design of the two platforms we traced: the Intel iPSC/860 and the Thinking Machines CM-5.

2.1 Workload Characterizations

We classify previous file-system workload studies as characterizing general-purpose workstations or workstation networks, scientific vector applications, or scientific parallel applications.

General-purpose workstations. Uniprocessor file access patterns have been measured many times. Floyd and Ellis [3, 4] and Ousterhout *et al.* [5] measured isolated Unix workstations, and Baker *et al.* measured a distributed Unix system (Sprite) [6]. Ramakrishnan *et al.* [7] studied access patterns in a commercial computing environment on a VAX/VMS platform. These studies all cover general-purpose (engineering and office) workloads with uniprocessor applications. These studies identify several characteristics that are common among uniprocessor file-system workloads: files tend to be small (only a few kilobytes), they tend to be accessed with small requests, and they tend to be accessed both completely and sequentially (i.e., each byte in the file is accessed in order — from beginning to end).

Scientific vector applications. Some studies specifically examined scientific workloads on vector machines. Del Rosario and Choudhary provide an informal characterization of grand-challenge applications [8]. Powell measured a set of static characteristics (file sizes) of a Cray-1 file system [9]. Miller and Katz traced specific I/O-intensive Cray applications to determine the per-file access patterns [10], focusing primarily on access rates. Miller and Katz also measured secondary-tertiary file migration patterns on a Cray [11], giving a good picture of long-term, whole-file access patterns. Pasquale and Polyzos studied I/O-intensive Cray applications, focusing on patterns in the I/O rate [12, 13]. All of these studies are limited to single-process applications on vector supercomputers. These studies identify several characteristics that are common among supercomputer file-system workloads. Unlike workstation file-system workloads, files tend to be large (many megabytes or gigabytes) and they tend to be accessed with large requests. Like workstation workloads, files are typically accessed both completely and sequentially.

Scientific parallel applications. Experimental studies of I/O from parallel scientific programs running on multiprocessors are rare. Crockett [14] and Kotz [15] hypothesize about the character of a parallel scientific file-system workload. Reddy and Banerjee chose five sequential scientific applications from the PERFECT benchmarks and parallelized them for an eight-processor Alliant, finding only sequential file-access patterns [16]. This study is interesting, but far from what we need: the sample size is small; the programs are parallelized sequential programs, not parallel programs *per se*; and the I/O itself was not parallelized. Cypher *et al.* [17] studied individual parallel scientific applications, measuring temporal patterns in I/O rates. Galbreath *et al.* [18] present a useful high-level characterization based on anecdotal evidence. Bagrodia *et al.* [19] have proposed using *Pablo* to analyze and characterize specific applications, and Crandall *et al.* performed such an analysis on three scientific applications [20]. As part of the CHARISMA project, we have traced parallel I/O requests by a live, production mix of user programs on an Intel iPSC [1] and on a CM-5 [2]. No other study has included more than one machine or programming platform.

2.2 Existing Parallel File Systems

A single, coherent model of parallel file-access has not yet emerged. Parallel-I/O models are often closely tied to a particular machine architecture as well as to a programming model. Nonetheless, there are some common characteristics. To increase parallelism, most parallel file systems decluster blocks of a file across many disks, which are accessed in parallel. Most extend a traditional file abstraction (a growable, addressable, linear sequence of bytes) with some parallel file-access methods. The most common provide I/O *modes* that specify whether and how parallel processes share a common file pointer [14, 21, 22, 23, 24, 25]. Some systems are based on a memory-mapped interface [26, 27], and two provide a way for the user to specify per-process logical views of the file [28, 29]. Some provide SIMD-style transfers [30, 31, 25, 18]. Finally, in addition to shared file pointers, MPI-IO allows applications to describe a mapping from a linear file to the compute nodes running the application in terms of higher-level data structures [32].

Clearly, the industrial and research communities have not yet settled on a single new model for file access. Thus, some aspects of a parallel file-system workload are dependent on the particular model provided to the user. The implications of this fact for our study are discussed throughout this paper, whenever such a dependency is apparent.

2.3 Systems Under Study

To be useful to a system designer, a workload characterization must be based on a realistic workload similar to that which is expected to be used in the future. For our purposes, this meant that we had to trace multiprocessor file systems that were in use for *production* scientific computing. The Intel iPSC/860 at NASA Ames' Numerical Aerodynamics Simulation (NAS) facility met this criterion, as did the Thinking Machines CM-5 at the National Center for Supercomputing Applications (NCSA).

2.3.1 Intel iPSC/860 and the Concurrent File System

The iPSC/860 is a distributed-memory, message-passing, MIMD machine. The compute nodes are based on the Intel i860 processor and are connected by a hypercube network. I/O is handled by dedicated I/O nodes, which are each connected to a single compute node rather than directly to the hypercube interconnect. The I/O nodes are based on the Intel i386 processor and each controls a single SCSI disk drive. There may also be one or more service nodes that handle such things as Ethernet connections or interactive shells [33]. At the time of our study, the iPSC/860 at NAS had 128 compute nodes and 10 I/O nodes. Each compute node had 8 MB of memory, and each I/O node had 4 MB of memory and a single 760 MB disk drive [34]. There was also a single service node that handled a 10-Mbit Ethernet connection to the host computer. The total I/O capacity was 7.6 GB and the total bandwidth was less than 10 MB/s.

Intel's Concurrent File System (CFS) stripes each file across all disks in 4 KB blocks. Requests are sent directly from the compute node that issues a request to the appropriate I/O node for service. Since the iPSC is a MIMD machine, the compute nodes operate independently of one another. To assist the programmer in coordinating accesses from these independent compute nodes to a single, shared file, CFS provides four *I/O modes*. Mode 0, the default mode, gives each process its own file pointer while mode 1 shares a single file pointer among all processes. Mode 2 is like mode 1, but enforces a round-robin ordering of accesses across all nodes, and mode 3 is like mode 2 but restricts the access sizes to be identical. More details about CFS, and its performance, can be found in [21, 35, 36].

2.3.2 Thinking Machines CM-5 and the Scalable File System

The CM-5 is a distributed-memory machine with many (tens to thousands) SPARC-based Processing Nodes, and a small number of Control Processors (CPs). Processing nodes are logically grouped into *partitions*, each of which is managed by a CP. Each job executes on a single partition. Generally, each processing node in a partition executes the same program, although they may execute different instructions (SPMD-style). Within individual partitions, jobs are timeshared. The processing nodes communicate via two scalable interprocessor communication networks [37]. Although it is possible for users' jobs running in different partitions to communicate with one another, it is rarely done in practice.

The CM-5 supports a variety of I/O devices [37, 38]. This study focuses on the Scalable Disk Array (SDA), as it was the primary high-volume, high-bandwidth storage device on the CM-5 at NCSA. The SDA is an expandable RAID-3 disk system that typically provides I/O bandwidths of 33-264 MB/sec. The Scalable File System (SFS) is an enhancement of the Unix file system with extensions to support parallel I/O and large files. Although it is a fully general file system, the SFS is optimized for parallel high-volume transfer.

During the tracing project, the CM-5 at NCSA had 512 nodes, and was generally divided into 5 static partitions of size 32, 32, 64, 128 and 256 nodes. The partitions on a CM-5 are reconfigurable, and at times the machine was reconfigured as a single 512-node partition. Each node had a single CPU, a network interface, and 4 vector units with a collective memory size of 32 MB/node. The SDA had a single file system distributed across 118 data disks and 1 parity disk, for a total capacity of about 138 GB. The logical block size of this file system was 29.5 KB and the physical disk block size is 59 KB.

The CM-5 supports two primary programming models, data-parallel and control-parallel, each with its own I/O model. In this paper we characterize I/O from programs written in

CMF, a data-parallel Fortran dialect, and CMMD, a control-parallel messaging library. The CMF programming model presents a single thread of control to the user; all nodes appear to be executing identical code though they may be operating on different data. CMF I/O is a library of support routines that are layered on top of SFS and allow users to read and write arrays (or portions thereof) to the SDA via either special library calls or normal Fortran READ and WRITE statements. Since there is only a single thread of control, every I/O request is *collective*. That is, whenever the application issues an I/O request, every node in the application must participate in that request. Issues of data distribution and I/O parallelization are hidden from the user.

The CMMD library may be used from a variety of familiar programming languages (e.g., C, C++, and f77) and, like the iPSC, provides the user with an independent thread of control for each processing node. CMMD I/O is also layered on top of SFS and, like CFS, provides a variety of I/O modes [39, 23]. CMMD's *local-independent* mode, like mode 0 in CFS, gives each process its own view of the file, and allows each process to make arbitrary requests to the file. In *global-independent* mode each process has a private file pointer, but all other state is shared. For example, if one process performs an `ioctl()` to change the blocking mode, the blocking mode will be changed for every process. CMMD's *synchronous-sequential* mode is like CFS's mode 2. Every node must participate in an I/O request, but each may request a different amount of data. The data will be read from or written to a contiguous region of the file, and the nodes' requests will be satisfied in round-robin order. In the final mode, *synchronous-broadcast*, every node accesses the exact same region of the file. While it is possible to write data in this mode, it is most likely to be used to read header information or a shared configuration file.

At NCSA, CMF users outnumber CMMD users by a factor of about 7 to 3 [40].

3 Methods

Given the diversity of multiprocessor file systems, it is not possible to construct an architecture-independent workload study. Thus, it is important to study a variety of platforms. By comparing and contrasting results from production workloads on multiple platforms, we may derive several benefits. First, if there are strong common trends one can confidently make some generalizations that can be used in parallel file-system design. Second, studying various platforms pinpoints platform- or environment-dependent characteristics that may be useful when designing a new file system for a similar platform or environment. In this section we describe our methods for collecting and analyzing data on two different platforms.

3.1 iPSC/860 Trace Collection

A CHARISMA trace file begins with a header record containing enough information to make the file self-descriptive, and continues with a series of event records, one per event. On the iPSC/860, one trace file was collected for the entire file system. We traced only the I/O that involved the Concurrent File System. This means that any I/O which was done through standard input and output or to the host file system (all limited to sequential, Ethernet speeds) was not recorded. We collected data for about 156 hours over a period of 3 weeks in February of 1994. While we did not trace continuously for the whole 3 weeks, we tried to get a realistic picture of the whole workload by tracing at all different times of the day and of the week, including nights and weekends. The period covered by a single trace file ranges from 30 minutes to 22 hours. The longest continuously traced period was about 62.5 hours. Tracing was usually initiated when the machine was idle. For those few cases in which a job was running when we began tracing, the job was not traced. Tracing was stopped in one of two ways: manually or by a full system crash. The machine was usually idle when a trace was manually stopped.

On the iPSC/860, high-level CFS calls are implemented in a run-time library that is linked with the user's program. We instrumented the library calls to generate an event record each time they were called. Since our instrumentation was almost entirely within a user-level library, there were some jobs whose file accesses were not traced. These included system programs (e.g., `cp`, and `ftp`) as well as user programs that were not relinked during the period we were tracing. While our instrumented library was the default, users that did not wish to have their applications traced had the option of linking with the uninstrumented library. Regardless of whether an application was traced or not, we were able to record all job starts and ends through a separate mechanism. While we were tracing, 3016 jobs were run, of which 2237 were only run on a single node. We actually traced at least 429 of the 779 multi-node jobs and at least 41 of the single-node jobs. As a tremendous number of the single-node jobs were system programs it is not surprising nor necessarily undesirable that so many were untraced. In particular, there was one single-node job that was run periodically, and which accounted for over 800 of the single-node jobs, simply to check the status of the machine. There was no way to distinguish between an untraced job and a traced job that did no CFS I/O, so the numbers of traced jobs are a lower bound.

One of our primary concerns was to minimize the degree that our measurement perturbed the workload. To reduce network contention and local per-call overhead, we buffered event records on each node and sent them to a central trace collector only when the buffer was full. Since large messages on the iPSC are broken into 4 KB blocks, we chose that as our buffer size. This buffering allowed us to reduce the number of messages sent to the collector by well over 90% without stealing much memory from user jobs. As our trace records were written to the same file system we were tracing, we were careful to minimize our effects on its performance as well, by creating a large buffer for the data collector and writing the data to CFS in large, sequential blocks. Since our data collector was linked with the non-instrumented library, our use of the file system was not recorded.

Simple benchmarking of the instrumented library revealed that the overhead added by our instrumentation was virtually undetectable in most cases. The worst case we found was a 7% increase in execution time on one run of the NAS NHT-1 Application-I/O Benchmark [41]. After the instrumented library was put into production use, anecdotal evidence suggests that there was no noticeable performance loss. Although we collected about 700 MB of data, our trace files accounted for less than 1% of the total CFS traffic.

Since each node buffered 4 KB of data before sending it to the central data collector, the raw trace file contained only a partially ordered list of event records. Ordering the records was complicated by the lack of synchronized clocks on the iPSC/860. Each node maintains its own clock; the clocks are synchronized at system startup but each drifts significantly after that [42]. We partially compensated for the asynchrony by timestamping each block of records when it left the compute node and again when it was received at the data collector. From the difference between the two, we attempt to adjust the event order to compensate for each node's clock drift relative to the collector's clock. While this technique results in a better estimation of the actual event order, it is still an approximation, so much of our analysis is based on spatial, rather than temporal, information.

3.2 CM-5 Trace Collection

On the CM-5 we traced programs from two different programming models: data-parallel CM Fortran (CMF) programs and control-parallel CMMD programs. In both CM-5 programming models, as in CFS, applications perform their I/O via runtime libraries. In this paper, we examine and discuss only the I/O done to and from the Scalable Disk Array.

CMF. As with CFS, we instrumented the run-time CMF I/O libraries to collect traces. While we gathered all our data in a single file on the iPSC, on the CM-5 each application's trace data was written to a separate file. We traced nearly all CMF applications that ran during the 23-day period from June 28, 1994 to July 20, 1994. The instrumentation had a mechanism for users to disable tracing of a particular job by setting an environment variable.

Some users (for example, industrial partners of NCSA) requested this feature and made use of it, thereby not having their applications traced. We had a separate mechanism that allowed us to count the total number of CMF jobs that were run during the tracing period even if they suppressed trace generation. Out of 1943 such jobs in that period, 1760 were traced. Neither figure includes programs that were compiled before the tracing library was installed. The 1760 jobs traced represent 434 distinct applications run by 384 distinct users.

As on the iPSC, we attempted to reduce the effects of our tracing on the user population. We wrote the per-job trace files onto the serial Unix file system to avoid contention with SDA I/O. We buffered the trace records in memory and wrote them to disk in large blocks to minimize tracing overhead. Performance measurements taken during beta-testing indicate that our instrumentation increased total application execution time by less than 5%.

CMMD. While we can classify the CMF workload as a "general" workload, the CMMD workload was self-selecting. We developed the CMMD tracing library at Thinking Machines Corporation on an in-house version of CMMD. Since it was developed off-site, the NCSA systems staff was reluctant to make it the default library, so we relied on users who voluntarily linked their programs to the CMMD tracing library for us to gather traces. We traced for a period of two weeks in the summer of 1994, and obtained traces from 127 jobs representing 29 distinct applications run by 11 distinct users. The volunteers tended to be heavy users of the SDA, and relatively sophisticated programmers, who were interested in parallel-I/O behavior. We can perhaps classify this workload as an I/O-intensive workload compared to the general CMF workload. This difference should be considered when interpreting the CMMD data.

CMMD I/O is implemented as a client/server architecture in which a privileged CM-5 host process is responsible for running a server loop. We monitored CMMD I/O by piggybacking trace records on the client/server protocols. The actual trace records were produced on the CM-5 compute nodes, communicated to the host server, then written to the local Unix file system. Since communication of trace records was embedded into the normal client/server I/O protocols we believe that perturbation was minimal.

4 Results

In this section we compare and contrast the iPSC and CM-5 workloads. We try to identify common trends, and to isolate reasons for differences in behavior. We characterize the workload from the top down. We begin by examining the number of jobs in the machine, then the number and use of files by all jobs, and then examine individual I/O requests. In addition to studying the sizes of I/O requests, we look for sequentiality and regularity among them. We then examine the requests at a higher level and try to identify specific kinds of regular access patterns. Finally we examine file sharing at various granularities.

Summary statistics for the three sets of traces may be seen in Table 1. We classify files by whether they were actually read, written, or read and written within a single open period, rather than by the mode used to open the file. Some files were opened but neither read nor written before being closed.

	Traced	Megabytes		Number of files				
System	Jobs	Read	Written	Opened	Read	Written	Both	Neither
CFS	470	38812.40	44725.29	63779	14540	44500	2259	2480
CMF	1760	35359.27	57631.46	3780	1271	2286	219	4
CMMD	127	30206.51	65693.89	904	257	596	49	2

Table 1: Summary of data collected on both the iPSC and the CM-5.

4.1 Jobs

Fig. 1 shows the amount of time each machine spent running a given number of jobs.[2] Since the CM-5 had a much larger user base, it is not surprising that it spent less time idle than

[2] The data on the overall number of jobs for the CM-5 was collected over 2 weeks in May 1995, not during the tracing period. Since we do not attempt to correlate this information with any other results in the paper, this lack of contemporaneousness should not be viewed as significant.

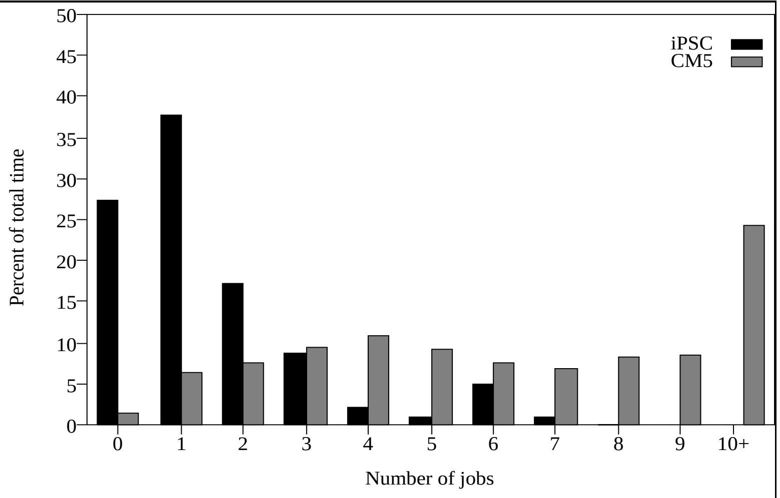


Figure 1: Amount of time the machine spent with the given number of jobs running. This data includes all jobs, even if their file access could not be traced.

did the iPSC. Unlike the iPSC, the CM-5 had timeshared partitions that allowed more jobs to run at the same time. Although the iPSC was idle for nearly 28% of the time we were tracing, the CM-5 was idle for less than 2% of the time. When the machines were actively executing jobs, the iPSC spent 35% of the time running a single job, and the CM-5 spent 6% of the time running a single job. This means the the iPSC was being used to run multiple applications simultaneously 25% of the time, and the CM-5 was executing multiple jobs 92% of the time. Although not all jobs use the file system, a file system clearly must provide high-performance access by many concurrent, presumably unrelated, jobs. While uniprocessor file systems are tuned for this situation, most multiprocessor file-systems research has ignored this issue, focusing on optimizing single-job performance.

Fig. 2 shows the distribution of the number of compute nodes used by each job on each machine. Although single-node jobs appear to dominate the job population on the iPSC, most of those jobs were caused by a daemon that was run periodically to check the status of the machine. The multiple-node jobs were fairly evenly distributed among the remaining sizes, up to 64 nodes. Although the iPSC allowed jobs as small as a single node, the CM-5 had a minimum partition size of 32 nodes. About 60% of the CMF jobs on the CM-5 used

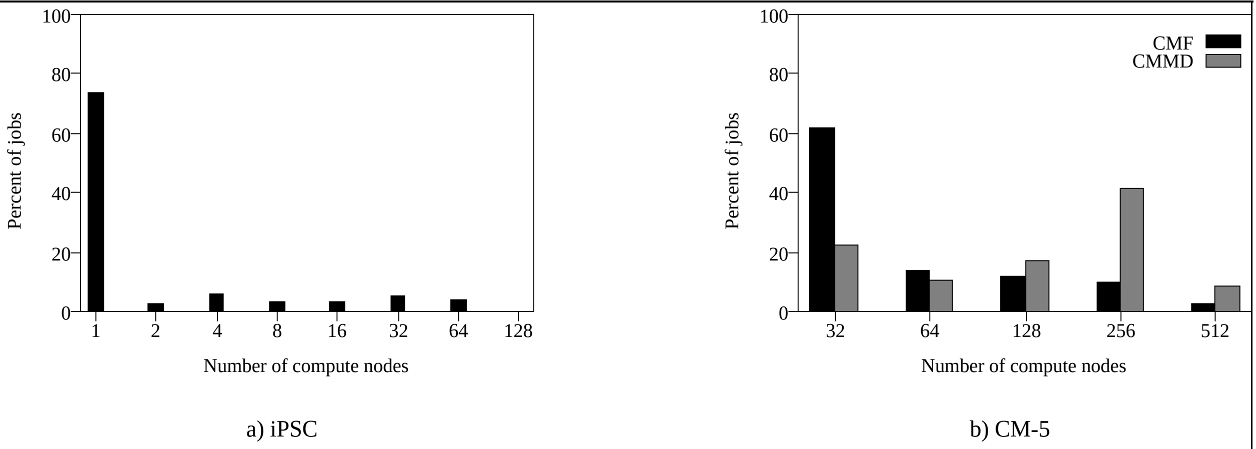


Figure 2: Distribution of the number of compute nodes used by jobs in our workload. Both machines limit the choice to powers of 2 and the CM-5 has a minimum partition size of 32 nodes.

this smallest partition size. On the other hand, since the CMMD workload was self-selecting and included fairly large and I/O-intensive applications, we observe a bias toward a large number of nodes. Over 50% of traced CMMD jobs used 256 nodes or more. Clearly, for a file system to be successful, it must allow efficient access from both small, sequential jobs and large, highly parallel jobs under a variety of conditions and system loads.

4.2 Files

In the two systems studied, there are two different manners in which a file may be opened: *locally* or *globally*. A file is said to be locally opened if each node that accesses the file issues an independent request to open the file. When a file is locally opened, each node that opens the file has a private view of that file, and operations on that file are not directly affected by other nodes using that file. In contrast, a file is said to be opened globally when all the nodes in an application collectively issue a request to open that file. When a file is globally opened, the nodes have a shared view of the file.

The CFS I/O model does not support the notion of a global open, so each file in CFS must be opened locally. As is discussed in Section 2.3.1 CFS provides several access modes that allow files to be treated globally once they have been opened. When discussing per-job file statistics, we coalesce the local opens issued by CFS into a single global open. That is, if multiple nodes in a CFS application each issue a local open for the same file, we count those local opens as a single global open. Since CMF is a data-parallel language, and provides only a single thread of control, every file operation in CMF is collective, and all file opens in CMF are global. CMMD allows programmers to open files either locally or globally. Since CMMD applications that wish to open a file globally may do so explicitly, and since few CMMD files were opened locally, we do not attempt to coalesce these local opens into global opens as we do with CFS.

In Table 1 above, note that many more files were written than were read (indeed, 2 to 3 times as many). We speculate that the programmers of traced CFS applications often found it easier to open a separate output file for each compute node, rather than coordinating writes to a common output file. This hypothesis is supported by the substantially smaller average number of bytes written per file (1.2 MB) than average bytes read per file (3.3 MB) on the iPSC. This difference in the average number of bytes accessed does not appear in the CM-5 workload. CMF jobs read an average of 27.8 MB/file and wrote an average of 25.2 MB/file, while CMMD applications read 117.5 MB/file and wrote 110.2 MB/file. The domination of write-only files on the CM-5 appears to come partly from checkpointing activity and partly from output files that were written to the SDA for later visualization. While the number of bytes read or written per file with CMF was substantially smaller than with CMMD, the amount of data transferred per file is still an order of magnitude larger than we observed with CFS. The users seem to have made use of the higher disk capacity and bandwidth that the CM-5 offers. Another common trend across all three platforms was that there were very few files that were both read and written (3.5% in CFS, 5.4% in CMMD, 5.8% in CMF). This behavior is also common in Unix file systems [3] and may be accentuated here by the difficulty in coordinating concurrent reads and writes to the same file.

Table 2 shows that most jobs opened only a few files over the course of their execution,

Number of Files	Number of Jobs CFS	CMF	CMMD
1	71	813	8
2	15	205	6
3	24	63	10
4	120	31	14
5	2	19	11
6	10	31	12
7	1	16	33
8+	24	151	33

Table 2: Among traced jobs, the number of files opened by jobs was often small (1–4).

although a few opened many files (one CFS job opened 2217 files). Although CMF required that files be opened on all nodes, under CFS some of the jobs that opened a large number of files were opening one file per node. Although it is not shown in the table above, nearly 25% of the jobs that used CMF did not use any files on the SDA. These applications were probably compute intensive and did their I/O via NFS. The number of files opened per job was higher in CMMD than CMF, again perhaps due to the self-selected nature of the users.

Despite the differences in the absolute numbers of files opened, it appears clear that the use of multiple files per job is common. Therefore, although not all files were open concurrently, file-system designers must optimize access to several files within the same job.

We found that only 0.61% of all opens in the CFS workload were to "temporary" files (defined as a file deleted by the same job that created it). The rarity of temporary files and of files that were both read and written indicates that few applications chose to use files as an extension of memory for "out of core" solutions. Many of the CFS applications were computational fluid dynamics codes, for which they have found that out-of-core methods are in general too slow. The workload on the CM-5 exhibited a larger number of temporary files (3.8% of CMF jobs and 4.9% of CMMD jobs). This difference may indicate that out-of-core methods were more common on the CM-5, or it may have been caused by deletion of checkpoint files by jobs that ran to completion.

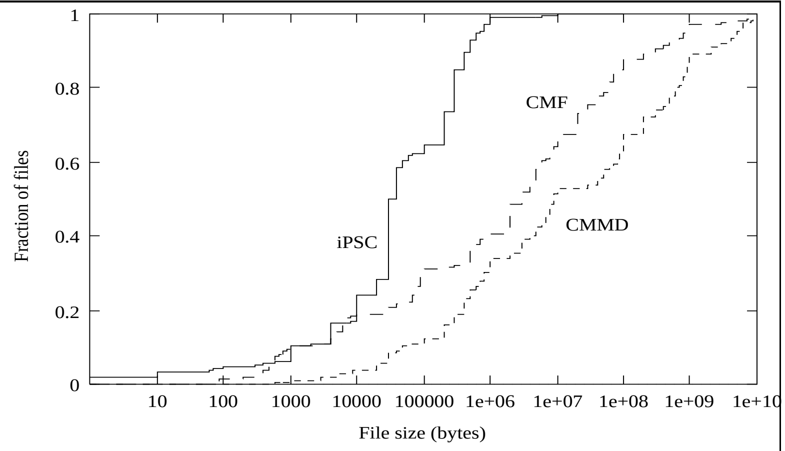


Figure 3: Cumulative distribution function (CDF) of the number of files of each size at close. For a file size x, CDF(x) represents the fraction of all files that had x or fewer bytes.

Fig. 3 shows a wide range in the size of files from system to system.[3] Most of the files accessed with CFS were between 10 KB and 1 MB. Although these files were larger than those in a general-purpose file system [6], they were smaller than we would expect to see in a scientific supercomputing environment [10]. Files on the CM-5 were significantly larger than on the iPSC, and the sizes were much more evenly distributed. One likely reason that files on the CM-5 were larger is the availability of 20 times more disk space.

4.3 I/O Request Sizes

Figures 4 and 5 show that on both the iPSC and the CM-5, the vast majority of accesses were small, but that most bytes were transferred through large accesses.

Indeed, 96% of all reads under CFS requested fewer than 100 bytes, but those reads transferred only 2% of all data read. Similarly, 90% of all writes under CFS were for fewer than 100 bytes, but those writes transferred only 3% of all data written. The number of small requests is surprising due to their poor performance in CFS [36]. CMMD's interface is

[3]As there were many small files as well as several distinct peaks across the whole range of sizes, there was no constant granularity that captured the detail we felt was important in a histogram. We chose to plot the file sizes on a logarithmic scale with pseudo-logarithmic bucket sizes; the bucket size between 10 and 100 bytes is 10 bytes, the bucket size between 100 and 1000 is 100 bytes, and so on.

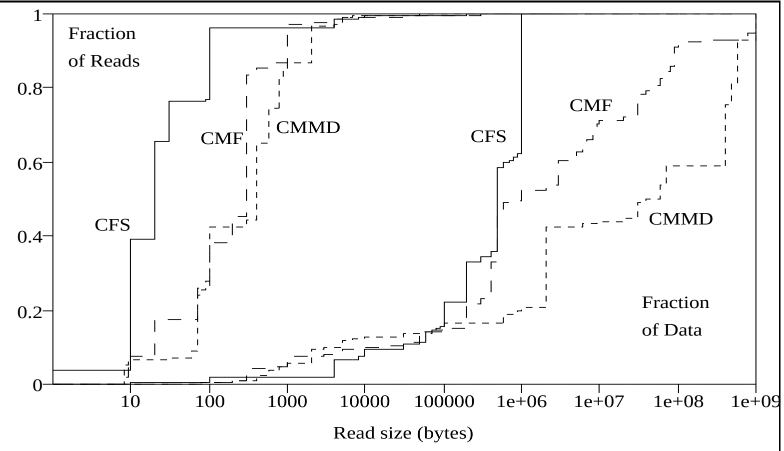


Figure 4: CDF of the number of reads by request size and of the amount of data read by request size.

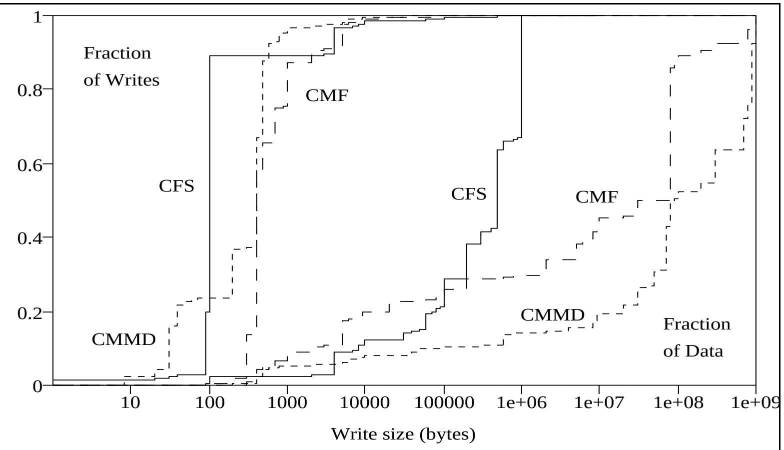


Figure 5: CDF of the number of writes by request size and of the amount of data written by request size.

similar to that of CFS, in that each compute node issues requests for data independently of all other compute nodes. Requests in CMMD were somewhat larger than CFS, with 87% of the reads and 96% of the writes under 1000 bytes. CMF provides a collective model for I/O, in which requests are issued for all compute nodes at once. Accordingly, we would expect to see much larger requests under CMF than either CMMD or CFS. We found, however, that even under CMF, 97% of the reads and 87% of the writes were under 1000 bytes. As with the iPSC, small requests on the CM-5 are known to perform poorly.

Although accesses on the CM-5 were larger than those observed on the iPSC, they were still significantly smaller than the tens or hundreds of kilobytes used in typical performance analyses of these systems [36, 43]. Studies have shown that large I/O requests are common in scientific applications running on supercomputers, but we have now seen that *small* requests are common in scientific applications running on parallel computers. Indeed, this trend holds across two different parallel machines, using three parallel file-system interfaces and two parallel programming models. Therefore, we believe that this preponderance of small request sizes in the observed scientific workloads is a natural result of parallelization and is fundamental to a large class of parallel applications. We conclude that future parallel file systems must focus on providing low latency for small requests as well as high bandwidth for large requests.

4.4 Sequentiality

One common characteristic of previous file system workload studies, particularly of scientific workloads, is that files are typically accessed sequentially [5, 6, 10]. We define a *sequential* request to be one that begins at a higher file offset than the point where the previous request from that compute node ended. This is a looser definition of sequential than is used in the studies referred to above. What previous studies have called sequential, we call *consecutive*. A consecutive request is a sequential request that begins precisely where the previous request ended. Figures 6 and 7 show the amount of sequential and consecutive access to files in the observed workloads. In these figures, we look at per-node access patterns for CFS and CMMD, and at per-job access patterns for CMF.

With all three interfaces, nearly all of the accesses to write-only files were 100% sequential. While access to read-only files was also predominantly sequential, both CMF and CMMD had several files that were read non-sequentially. There were several applications on the CM-5 that wrote data to files in forward order and then read it back in reverse order. This behavior accounts for at least some of the non-sequential accesses on that machine. Unsurprisingly, most read-write files were accessed non-sequentially.

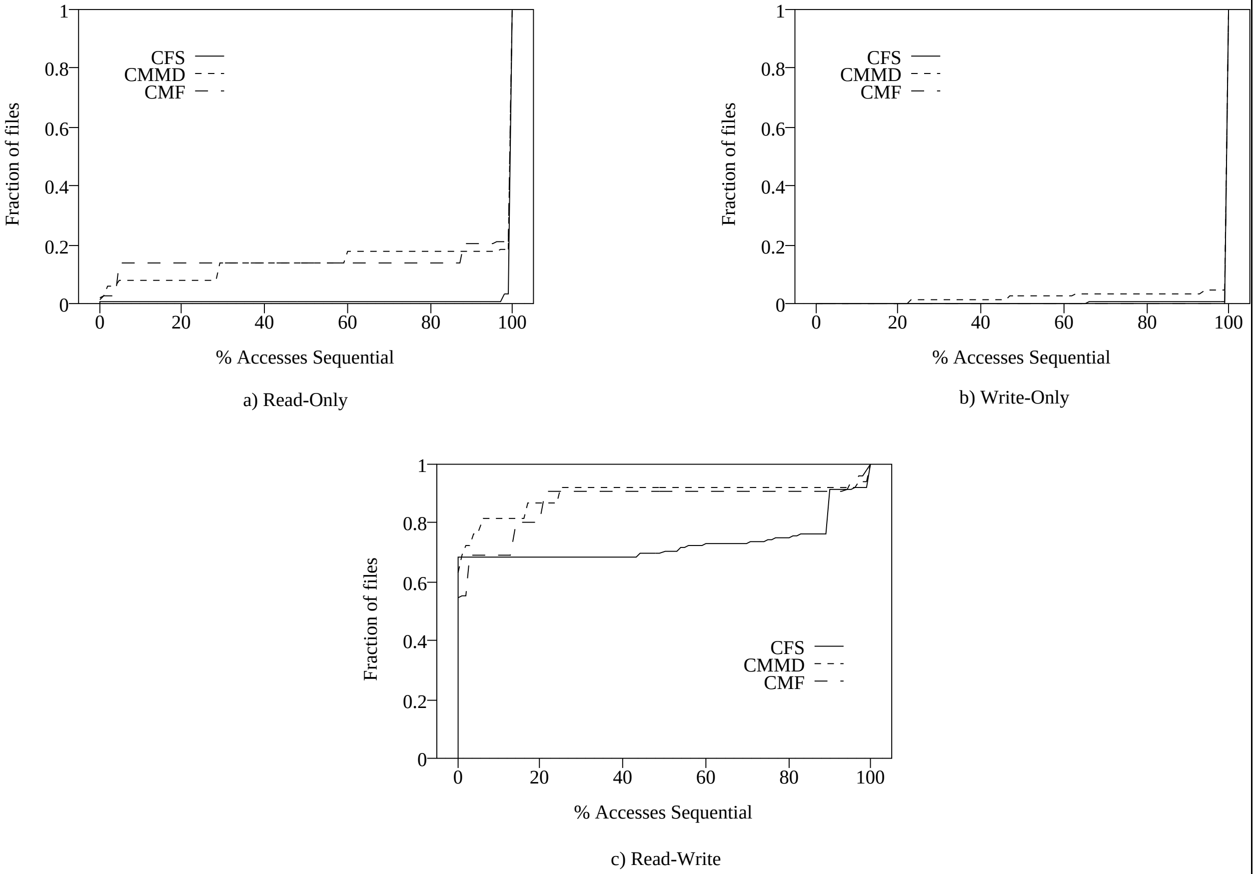


Figure 6: CDF of sequential access to files. Each point on a line indicates the fraction of files in the workload that contain no more than the indicated percentage of sequential accesses.

Looking at the graphs of consecutive access in Fig. 7, we find that the behavior varies between systems and interfaces. With both CFS and CMF, nearly 90% of the write-only files were accessed 100% consecutively. With CMMD, on the other hand, only 60% of the write-only files were accessed completely consecutively. With all three interfaces, read-only files were much less likely to be accessed consecutively than write-only files. The least consecutive access was found in CFS, in which over 65% of the read-only files had no consecutive accesses at all. In all cases, access to read-write files was primarily non-consecutive.

One significant reason for the relatively high percentage of consecutive access in write-only files on the iPSC was a tendency for applications to assign a different file to each node that was writing data. When only a single node accesses a file, there is frequently

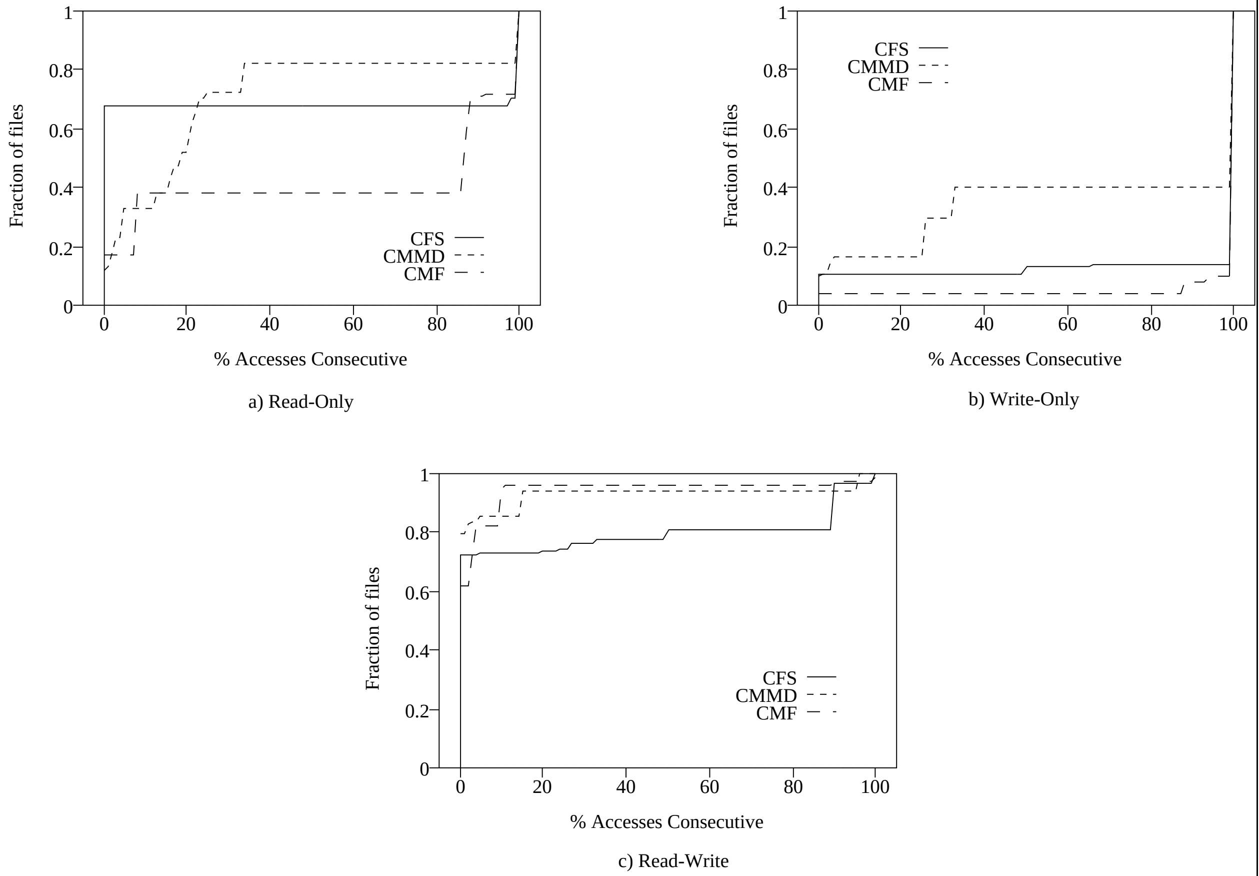


Figure 7: CDF of consecutive access to files. Each point on a line indicates the fraction of files in the workload that contain no more than the indicated percentage of consecutive accesses.

no reason for that node to access the file non-consecutively. When multiple nodes access a file, as happened frequently with read-only files in CFS and with files in CMMD, the large number of sequential, but non-consecutive, accesses was often the result of interleaved access. Interleaved access arises when successive records of a file are accessed by different nodes, so from the perspective of an individual node, some bytes must be skipped between one request and the next. The high percentage of consecutive access to files from CMF programs is expected because we are looking at collective, job-level patterns, rather than individual, node-level patterns. Since the I/O requests in CMF applications are not issued by the individual nodes, this sort of interleaving is unlikely to appear.

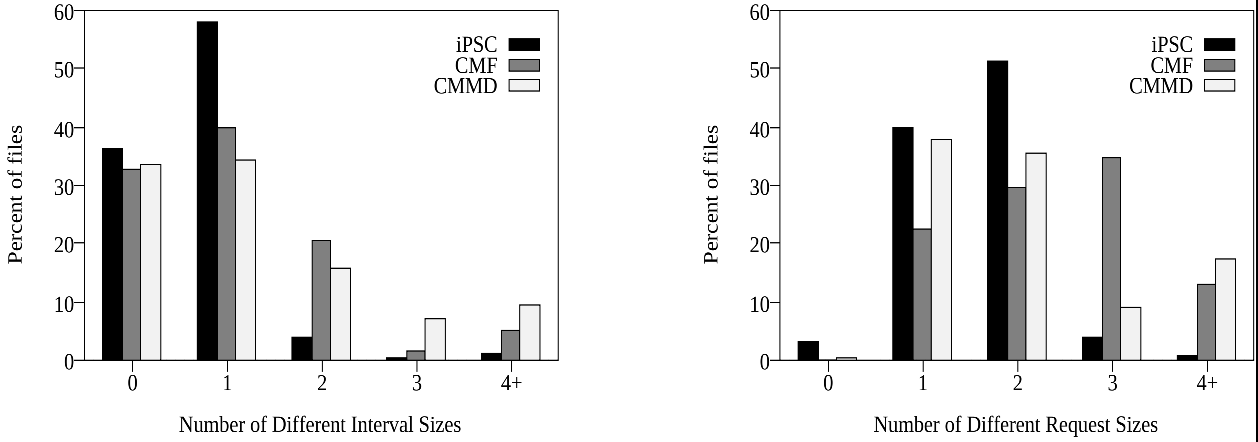


Figure 8: The number of different interval and request sizes used in each file across all participating nodes. Files with zero interval sizes had at most one access by each node. Files with zero request sizes were opened and closed without being accessed.

4.5 Regularity

These workloads, with many small, non-consecutive requests, are different from previously observed workloads on traditional uniprocessors and supercomputers. In an attempt to gain a better understanding of the observed workloads, we tried to identify points of regularity.

Intervals. We first looked at the *interval* between requests, or the number of bytes between the end of one request and the beginning of the next. Consecutive accesses have an interval of size 0. The number of interval sizes used in each file, across all nodes that access that file, is shown in Fig. 8. A surprising number of files (around 1/3 in all cases) were read or written in one request per node (i.e., there were no intervals). Most of the files (99% in CFS, 79% in CMF, and 51% in CMMD) that were accessed with a single interval size were accessed consecutively (i.e., the one interval size was 0). The remainder of 1-interval-size files, along with the 2-interval-size files, represent most of the remaining files, which suggests that there exists another form of highly regular access pattern. Only a few files had 3 or more different interval sizes, and their regularity (if any) was more complex.

Requests. To get a better feel for this regularity, Fig. 8 also shows the number of different *request sizes* used in each file. CFS exhibited the highest degree of regularity, with over 90% of the files being accessed with only one or two request sizes. CMMD was next with about 75% of the files being accessed with only one or two different request sizes. CMF was the least regular with just over half of the files being accessed with two or fewer request sizes. This may indicate that CMF users used the same file to store different data structures (e.g., different matrices). Even in CMF, over 80% of the files were accessed with three or fewer request sizes. Combining the regularity of request sizes with the regularity of interval sizes, many applications clearly used regular, structured access patterns, possibly because much of the data was in matrix form.

4.6 Strided Access

To better understand the structure and causes of the regular but non-consecutive access patterns, we examined the trace files for evidence of *strided* access patterns [44].

4.6.1 Simple-Strided

We refer to a series of I/O requests as a *simple-strided* access pattern if each request is for the same number of bytes, and if the file pointer is incremented by the same amount between each request. This pattern would occur, for example, if each process in a parallel application read a column of data from a matrix stored in row-major order. It could also correspond to the pattern generated by an application that distributed the columns of a matrix across its processors in a cyclic pattern, if the columns could be distributed evenly and if the matrix was stored in row-major order. Since a strided pattern was less likely to occur in single-node files, and since it could not occur in files that had only one or two accesses, we looked only at those files that had three or more requests by multiple nodes.

Fig. 9 shows that many of the accesses to the selected subset of CFS and CMMD files appeared to be part of a simple-strided access pattern. Since consecutive access could be

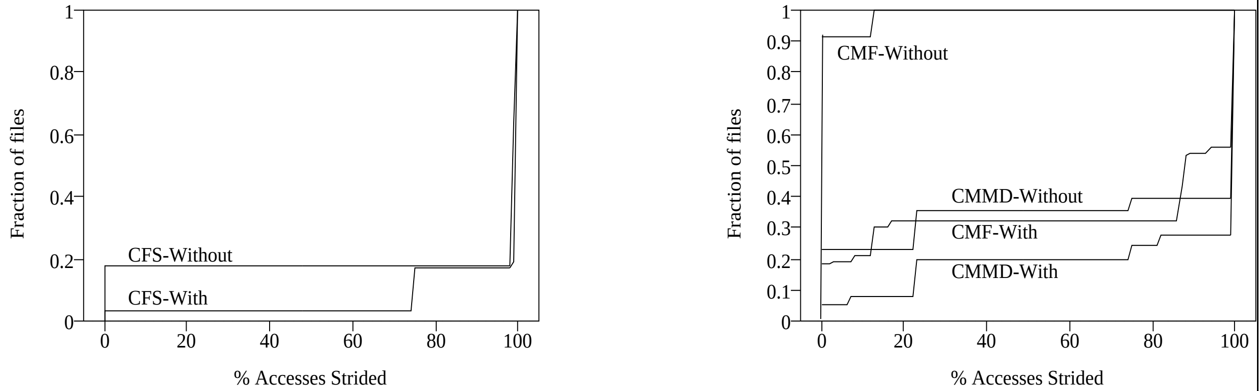
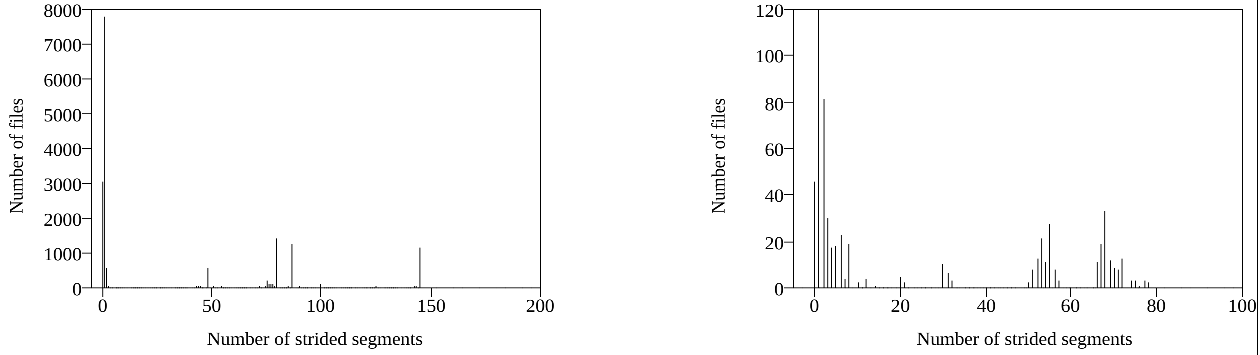


Figure 9: Cumulative distribution of files according to the fraction of accesses that were involved in a simple-strided pattern. These plots show the frequency of strided access both with consecutive accesses counted as strided, and without.

considered a trivial form of strided access (with an interval of 0), Fig. 9 shows the frequency of strided accesses with and without consecutive accesses included. In either case, over 80% of all the files we examined in CFS were apparently accessed entirely with a strided pattern. Strided access was also common in CMMD, with over 60% of the files being accessed entirely in a strided, non-consecutive pattern. If we exclude consecutive access, there appeared to be almost no strided access in CMF, with no more than 20% of the requests to any file taking part in a strided pattern. This lack of strided access in CMF is not surprising, since strided access is typically caused by the explicit expression of data distribution in a control-parallel program. Accordingly, the remainder of our discussion will focus on CFS and CMMD.

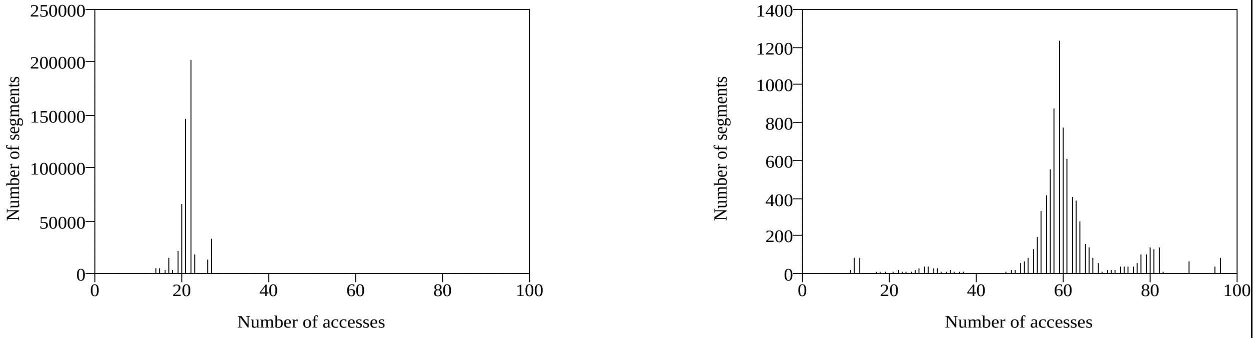
We define a *strided segment* to be a group of requests that appear to be part of a single simple-strided pattern. While Fig. 9 shows the percentage of requests that were involved in some strided segment, it does not tell us whether each file was accessed with a single, file-long strided segment, or with many shorter segments. Fig. 10 shows that while most files had only a few strided segments, there were some files that were accessed with many strided segments. Since we were only interested in those cases where a file was clearly being accessed in a strided pattern, this figure does not include consecutive accesses or segments with fewer than 10 requests. The number of requests in a segment varied between the machines. Fig. 11



a) CFS

b) CMMD

Figure 10: The number of different strided segments in each file. We ignore segments with fewer than 10 requests. Note that the two plots use different scales.



a) CFS

b) CMMD

Figure 11: Head of the segment-length distribution. These plots show number of segments of a given length (including 'short' segments of 10 or fewer accesses).

shows that while most segments in CFS fell into the range of 20 to 30 requests, most of the segments in CMMD had 55 to 65 requests. Fig. 12 shows that there were some files that were accessed with much longer segments on both machines.

While the existence of these simple-strided patterns is interesting and potentially useful, the fact that many files were accessed in multiple short segments suggests that there was a level of structure beyond that described by a simple-strided pattern.

4.6.2 Nested Patterns

A *nested-strided* access pattern is similar to a simple-strided access pattern but rather than being composed of simple requests separated by regular strides in the file, it is composed of

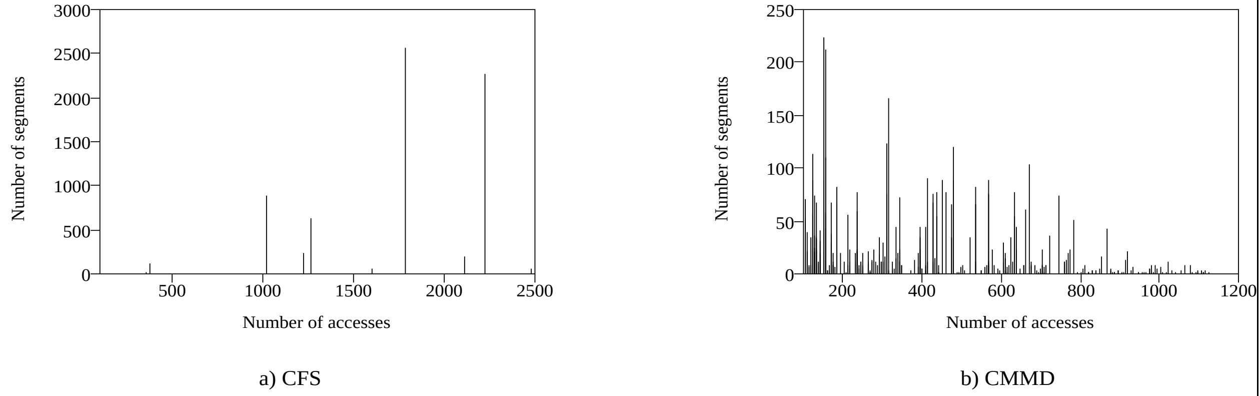


Figure 12: Tail of the segment-length distribution.

strided segments separated by regular strides in the file. The simple-strided patterns examined in the last section could be called singly-nested patterns. A doubly-nested pattern could correspond to the pattern generated by an application that distributed the columns of a matrix stored in row-major order across its processors in a cyclic pattern, if the columns could not be distributed evenly across the processors. The simple-strided sub-pattern corresponds to the requests generated within each row of the matrix, while the top-level pattern corresponds to the distance between one row and the next. This access pattern could also be generated by an application that was reading a single column of data from a three-dimensional matrix. Higher levels of nesting could occur if an application mapped a multidimensional matrix onto a set of processors.

Maximum Level of Nesting	Number of CFS files	Number of CMMD files
0	469	38
1	10945	311
2	747	102
3	5151	148
4+	0	3

Table 3: The number of files that use a given maximum level of nesting.

Table 3 shows how frequently nested patterns occurred in CFS and CMMD. A file that had no apparent strided accesses had zero levels of nesting. Files that were accessed with only simple-strided patterns had a single level of nesting. Interestingly, on both machines it was far more common for files to exhibit three levels of nesting than two. This tendency suggests that the use of multidimensional matrices was common on both systems.

4.7 Synchronized Access Modes

Although the notion of synchronized access to files is built into the semantics of the data-parallel CMF, such is not the case with CFS and CMMD. To provide synchronized access to files, both CFS and CMMD provide the user with the option of using a file pointer that is shared among all the nodes. They also provide several *modes*, each of which provides the user with different semantics governing how the file pointer is shared.

Given the regularity of request and interval sizes on the iPSC, CFS's modes (see Section 2.3.1) would seem to be helpful. Our traces show, however, that over 99% of the files used mode 0, which provides each node with an independent file pointer. Fig. 8 gives one hint as to why: although there were few different request sizes and interval sizes, there were often more than one, something not easily supported by the shared-pointer modes. Mode 0 was also known to be the fastest of the four modes offered by CFS.

In contrast to CFS, CMMD's local-independent mode was known to be slow and was only used for 0.88% of total I/O. Instead, CMMD applications used synchronous-sequential mode for most (78%) of their I/O. Synchronous-broadcast mode was used for only 8.7% of total I/O. Global-independent mode was used for 11.9% of total I/O, and we speculate that it was used when just one node or a subset of nodes were using a file among themselves. From the above data, one may be inclined to conclude that CM-5 applications only needed fast synchronous I/O. Anecdotal evidence suggests, however, that users frequently wanted independent I/O but were not willing to pay the performance penalty. That CMMD and CFS users adopt very different I/O strategies to achieve the same end result (high performance), illustrates how the capabilities of an existing machine may influence user behavior.

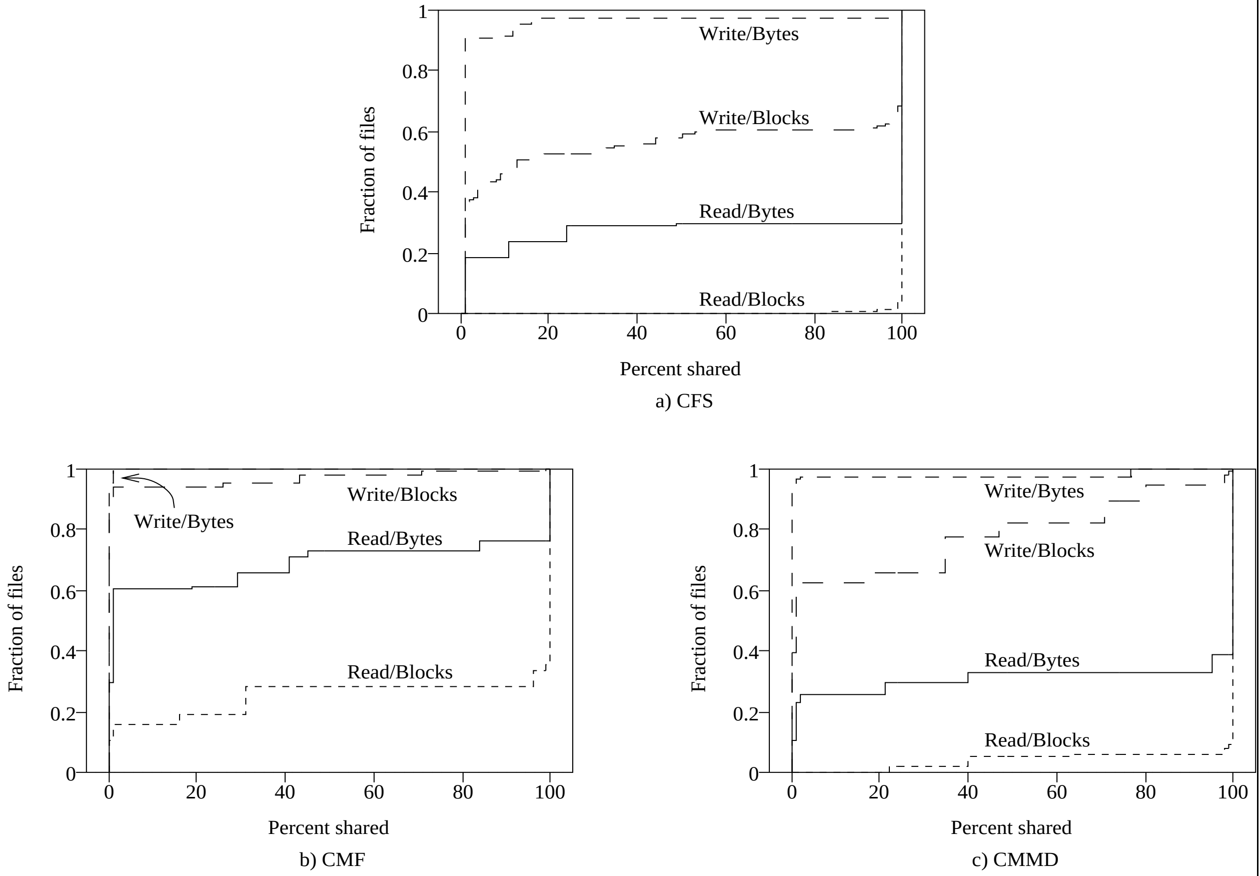


Figure 13: CDF of file sharing between nodes in Read-Only and Write-Only files at byte and block granularity. The block size on the iPSC was 4 KBytes and the block size on the CM-5 was 29.5 KBytes.

4.8 File Sharing Between and Within Jobs

A file is *concurrently shared* if two or more processes have it open at the same time. In uniprocessor and distributed-system workloads, concurrent sharing is known to be uncommon, and writing to concurrently shared files is almost unheard of [6]. In a parallel file system, of course, concurrent file sharing among processes within a job is presumably the norm, while concurrent file sharing between jobs is likely to be rare. Indeed, in our traces we saw a great deal of file sharing within jobs, and *no* concurrent file sharing between jobs. The interesting question is *how* the individual bytes and blocks of the files were shared. Fig. 13 shows the frequency of both byte- and block-sharing in each system.

In all three cases, there was more sharing of read-only files than of write-only files, which is not surprising given the complexity of coordinating write sharing. Indeed, in CFS 70% of read-only files had all of their bytes shared by multiple compute nodes, while 90% of write-only files had no bytes shared at all. We found similar results with CMMD, in which 61% of the read-only files had all their bytes shared, and 93% of the write-only files had none of their bytes shared. CMF had the least sharing of the three systems, with 95% of the write-only files having no bytes shared and 60% of the read-only files having 1% or fewer bytes shared by multiple compute nodes. This lack of sharing is likely an artifact of CMF's data-parallel programming model, where processors are statically assigned non-overlapping portions of a matrix. Even when there was not a lot of byte-sharing, there was usually a large amount of block-sharing. Overall, the amount of block sharing implies strong *interprocess* spatial locality, and suggests that caching at the I/O nodes may improve system performance [1].

5 Conclusions and Recommendations

Across the two machines and two programming models covered in this paper, we found important similarities and differences. Compared to uniprocessor workloads, all three parallel workloads used much larger files, and were dominated by writes. Although there were variations in magnitude, we found small request sizes to be common in all three parallel workloads, just as they are in uniprocessor workloads. Compared to vector-supercomputer workloads, we observed much smaller requests and a tendency toward non-consecutive, but sequential file access. Finally, parallelism leads to new, interleaved access patterns with high interprocess spatial locality at the I/O node. While some of the details of our results may be specific to the two systems we studied, or to the workloads at the two sites, we believe that the general conclusions above are widely applicable to scientific workloads running on loosely-coupled MIMD multiprocessors. This category includes many current multiprocessors.

Ultimately, we believe that the file-system interface must change. The current interface forces the programmer to break down large parallel I/O activities into small, non-consecutive requests. We believe that a control-parallel model should support strided I/O requests from the programmer's interface to the compute node, and from the compute node to the I/O node [24, 44]. A strided request can effectively increase the request size, which lowers overhead and introduces opportunities for low-level optimization [45].

Future Work

While we believe that low-level workload analyses such as we have conducted are an important first step towards developing parallel file systems that can meet the needs of parallel scientific applications, there is still a great deal of work to be done.

- Trace more platforms to reduce the likelihood that results are specific to an architecture or environment.
- Study specific applications in greater detail. Our workload studies describe *how* parallel file systems are used, but studying individual applications will allow us to understand *why* they are being used in that fashion, and to better understand the application programmer's fundamental needs.
- Design and implement new interfaces and file systems based on these workload analyses.

Acknowledgments

Many thanks to the NAS division at NASA Ames, especially Jeff Becker, Russell Carter, Sam Fineberg, Art Lazanoff, Bill Nitzberg, and Leigh Ann Tanner. Many thanks also to Orran Krieger, Bernard Traversat, and the rest of the CHARISMA group.

We thank Michael Welge and Curtis Canada of NCSA. Many thanks to the NCSA users including Greg Bryan, Diane Cook, Tom Cortese, Kathryn Johnston, Chris Kuszmaul, Fady Najjar, and Robert Sugar. We also thank Kapil Mathur and David Phillimore at Thinking Machines Corporation and Doreen Revis at Duke.

Finally, we thank the many users who agreed to have their applications traced.

References

[1] David Kotz and Nils Nieuwejaar, "File-system workload on a scientific multiprocessor", *IEEE Parallel and Distributed Technology*, pp. 51–60, Spring 1995.

[2] Apratim Purakayastha, Carla Schlatter Ellis, David Kotz, Nils Nieuwejaar, and Michael Best, "Characterizing parallel file-access patterns on a large-scale multiprocessor", in *Proceedings of the Ninth International Parallel Processing Symposium*, Apr. 1995, pp. 165–172.

[3] Rick Floyd, "Short-term file reference patterns in a UNIX environment", Tech. Rep. 177, Dept. of Computer Science, Univ. of Rochester, Mar. 1986.

[4] Richard Allen Floyd and Carla Schlatter Ellis, "Directory reference patterns in hierarchical file systems", *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 2, pp. 238–247, June 1989.

[5] John Ousterhout, Hervé Da Costa, David Harrison, John Kunze, Mike Kupfer, and James Thompson, "A trace driven analysis of the UNIX 4.2 BSD file system", in *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, Dec. 1985, pp. 15–24.

[6] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, "Measurements of a distributed file system", in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 1991, pp. 198–212.

[7] K. K. Ramakrishnan, P. Biswas, and Ramakrishna Karedla, "Analysis of file I/O traces in commercial computing environments", in *Proceedings of ACM SIGMETRICS and PERFORMANCE '92*, 1992, pp. 78–90.

[8] Juan Miguel del Rosario and Alok Choudhary, "High performance I/O for parallel computers: Problems and prospects", *IEEE Computer*, vol. 27, no. 3, pp. 59–68, Mar. 1994.

[9] Michael L. Powell, "The DEMOS File System", in *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, Nov. 1977, pp. 33–42.

[10] Ethan L. Miller and Randy H. Katz, "Input/output behavior of supercomputer applications", in *Proceedings of Supercomputing '91*, Nov. 1991, pp. 567–576.

[11] Ethan L. Miller and Randy H. Katz, "An analysis of file migration in a UNIX supercomputing environment", in *Proceedings of the 1993 Winter USENIX Conference*, Jan. 1993, pp. 421–434.

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

[13] Barbara K. Pasquale and George C. Polyzos, “A case study of a scientific application I/O behavior”, in *Proceedings of the International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 1994, pp. 101–106.

[14] Thomas W. Crockett, “File concepts for parallel I/O”, in *Proceedings of Supercomputing '89*, 1989, pp. 574–579.

[15] David Kotz and Carla Schlatter Ellis, “Prefetching in file systems for MIMD multiprocessors”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 218–230, Apr. 1990.

[16] A. L. Narasimha Reddy and Prithviraj Banerjee, “A study of I/O behavior of Perfect benchmarks on a multiprocessor”, in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 312–321.

[17] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina, “Architectural requirements of parallel scientific applications with explicit communication”, in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993, pp. 2–13.

[18] N. Galbreath, W. Gropp, and D. Levine, “Applications-driven parallel I/O”, in *Proceedings of Supercomputing '93*, 1993, pp. 462–471.

[19] Rajive Bagrodia, Andrew Chien, Yarson Hsu, and Daniel Reed, “Input/output: Instrumentation, characterization, modeling and management policy”, Tech. Rep. CCSF-41, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[20] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed, “Input/output characteristics of scalable parallel applications”, in *Proceedings of Supercomputing '95*, Dec. 1995, To appear.

[21] Paul Pierce, “A concurrent file system for a highly parallel mass storage system”, in *Fourth Conference on Hypercube Concurrent Computers and Applications*, 1989, pp. 155–160.

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

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

[24] David Kotz, "Multiprocessor file system interfaces", in *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, 1993, pp. 194–201.

[25] S. R. Chapple and S. M. Trewin, *PUL-GF Prototype User Guide*, Feb. 1993, EPCC-KTP-PUL-GF-UG 0.1.

[26] "KSR1 technology background", Kendall Square Research, Jan. 1992.

[27] Orran Krieger and Michael Stumm, "HFS: a flexible file system for large-scale multiprocessors", in *Proceedings of the 1993 DAGS/PC Symposium*, Hanover, NH, June 1993, Dartmouth Institute for Advanced Graduate Studies, pp. 6–14.

[28] 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*, 1993, pp. 472–481.

[29] Erik DeBenedictis and Juan Miguel del Rosario, "nCUBE parallel I/O software", in *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC)*, Apr. 1992, pp. 0117–0124.

[30] "Connection Machine model CM-2 technical summary", Tech. Rep. HA87-4, Thinking Machines, Apr. 1987.

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

[32] Peter Corbett, Dror Feitelson, Yarson Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong, "MPI-IO: a parallel file I/O interface for MPI", Tech. Rep. NAS-95-002, NASA Ames Research Center, Jan. 1995, Version 0.3.

[33] Intel Corporation, *iPSC/2 and iPSC/860 User's Guide*, Apr. 1991.

[34] NASA Ames Research Center, Moffet Field, CA, *NAS User Guide*, 6.1 edition, Mar. 1993.

[35] 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*, vol. 17, no. 1–2, pp. 115–121, January and February 1993.

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

[37] Thinking Machines Corporation, *CM5 Technical Summary*, November 1993.

[38] Thinking Machines Corporation, *CM-5 I/O System Programming Guide Version 7.2*, September 1993.

[39] Thinking Machines Corporation, *CMMD Reference Manual Version 3.0*, May 1993.

[40] NCSA Consulting Staff and NCSA CM-5 Systems Staff, "Personal communication", June 1994.

[41] Russell Carter, Bob Ciotti, Sam Fineberg, and Bill Nitzberg, "NHT-1 I/O benchmarks", Tech. Rep. RND-92-016, NAS Systems Division, NASA Ames, Nov. 1992.

[42] James C. French, "A global time reference for hypercube multiprocessors", in *Fourth Conference on Hypercube Concurrent Computers and Applications*, 1989, pp. 217–220.

[43] Thomas T. Kwan and Daniel A. Reed, "Performance of the CM-5 scalable file system", in *Proceedings of the 8th ACM International Conference on Supercomputing*, July 1994, pp. 156–165.

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

[45] 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.