Integrating Theory and Practice in Parallel File Systems

Thomas H. Cormen David Kotz

Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755-3551

Abstract

Several algorithms for parallel disk systems have appeared in the literature recently, and they are asymptotically optimal in terms of the number of disk accesses. Scalable systems with parallel disks must be able to run these algorithms. We present for the first time a list of capabilities that must be provided by the system to support these optimal algorithms: control over declustering, querying about the configuration, independent I/O, and turning off parity, file caching, and prefetching. We summarize recent theoretical and empirical work that justifies the need for these capabilities. In addition, we sketch an organization for a parallel file interface with low-level primitives and higher-level operations.

1 Introduction

To date, the design of parallel disk systems and file systems for parallel computers has not taken into account much of the theoretical work in algorithms for parallel I/O models. Yet, theory has proven to be valuable in the design of other aspects of parallel computers, most notably networks and routing methods. In addition, empirical studies of early parallel file systems have found that optimizing performance requires programs to carefully organize their I/O. This paper describes how the design of parallel I/O software and hardware should be influenced by these theoretical and empirical results.

People use parallel machines for one reason and one reason only: speed. Parallel machines are certainly no easier or cheaper to use than serial machines, but they can be much faster. The design of parallel disk and file systems must be performance-oriented as well. There are several recent algorithms for parallel disk systems that are asymptotically optimal, solve important and interesting problems, and are practical. These algorithms require certain capabilities from the underlying disk and file systems, and these capabilities are not difficult to provide.

Not all parallel systems provide these capabilities, however, and only those that do can be scalable. Here, by *scalable* we mean that disk usage is asymptotically optimal as the problem and machine size increase. Because disk accesses are so time-consuming compared to computation, changing the number of parallel disk accesses by even a constant factor often has a strong impact on overall performance. The impact is even greater as the problem or machine size grows. For applications that use huge amounts of data, it is essential to use the best algorithms to access the

This research was supported in part by funds from Dartmouth College. Authors' names are listed alphabetically. Tom Cormen's Internet address is Thomas.H.Cormen@Dartmouth.edu, and his telephone number is (603) 646-2417. David Kotz's Internet address is David.Kotz@Dartmouth.edu, and his telephone number is (603) 646-1439.

Appeared in DAGS/PC Symposium on Parallel I/O and Databases, pages 64-74.
Available at URL ftp://ftp.cs.dartmouth.edu/pub/CS-papers/Kotz/cormen:integrate.ps.Z

data. The disk and file system capabilities to support these algorithms are then equally essential for scalability.

The capabilities we describe apply to two different uses of parallel I/O. One is the traditional file-access paradigm, in which programs explicitly read input files and write output files. The other is known variously as "out-of-core," "extended memory," "virtual memory," or "external" computing, in which a huge volume of data forces a computation to store most of it on disk. Data is transferred between memory and disk as needed by the program.

This paper sketches an interface that includes primitive operations to provide the capabilities. The interface also includes higher-level operations that use these primitives to implement algorithms whose disk usage is asymptotically optimal.

The remainder of this paper is organized as follows. Section 2 describes the capabilities required for asymptotically optimal parallel I/O performance and surveys some existing systems according to whether they provide these capabilities. Although one may view our list of capabilities as "conventional wisdom," few existing systems, if any, supply them all. Section 3 lists the algorithms that drive these capabilities and presents supporting empirical evidence for why these capabilities are necessary for high performance. Section 4 outlines an organization for a parallel file interface. Maintaining parity for data reliability on parallel disk systems exacts a performance cost, and Section 5 shows that for several parallel I/O-based algorithms, we can dramatically reduce the cost of maintaining parity information. Finally, Section 6 offers some concluding remarks.

2 Necessary capabilities

In this section, we present the capabilities that parallel file systems and disk I/O architectures must have to support the most efficient parallel I/O algorithms. Many of these required capabilities turn out to be at odds with those of some existing parallel systems. We conclude this section with a brief survey of existing parallel file systems in terms of these capabilities.

All disk I/O occurs in *blocks*, which contain the smallest amount of data that can be transferred in a single disk access. Any system may choose to perform its disk I/O in integer multiples of the block size.

Before proceeding, we note that the algorithms, and hence the required capabilities, apply to both SIMD and MIMD systems. In SIMD systems, the controller organizes the disk accesses on behalf of the processors. In MIMD systems, the processors organize their own disk accesses. In either case, the algorithms specify the activity of the disks.

The necessary capabilities are control over declustering, querying about the configuration, independent I/O, and turning off parity, file caching, and prefetching. We discuss each in turn.

Control over declustering

Declustering is the method by which data in each file is distributed across multiple disks. A given declustering is defined by a striping unit and a distribution pattern of data across disks. The *striping unit* is the sequence of logically contiguous data that is also physically contiguous within a disk. A common distribution pattern is *striping*, in which striping units are distributed in round-robin order among the disks; a *stripe* consists of the data distributed in one round. Striping unit sizes are often either one bit (as in RAID level three [PGK88]) or equal to the block size (as in RAID levels four and five).

The optimal algorithms assume striping with a block-sized striping unit. The programmer, therefore, should be able to redefine the striping unit size and distribution pattern of individual files.

Querying about the configuration

The optimal algorithms need the ability to query the system about the number of disks, block size, number of processors, amount of available physical memory, and current declustering method. In addition, some algorithms need to know the connection topology among compute processors, I/O processors, and disks.

Independent I/O

The algorithms typically access one block from each disk in an operation known as a *parallel I/O*. Optimality often depends on the ability to access blocks at different locations on the multiple disks in a given parallel I/O. We call such parallel I/O operations *independent*, in contrast to *fully striped* operations, in which all blocks accessed are at the same location on each disk.[1] The block locations we refer to are not absolute disk addresses; rather, they are logical offsets from the beginning of the file on each disk.

In order to perform independent I/O within a SIMD system, the I/O interface must allow specification of one offset into the file for each disk. Contrast this style of access with the standard sequential style, in which all I/O operations specify a single offset into the file. When this single-offset style is extended to parallel file systems, independent I/O is not possible.

Turning off parity

Another necessary capability is that of turning off parity or other redundancy management on a per-file basis. Section 5 examines why turning off parity can help performance and how to do so without compromising data reliability.

Turning off file caching and prefetching

The final capability we require is that of bypassing all file caching and prefetching mechanisms. In Section 3, we show that file caching interferes with many file access patterns and that the optimal algorithms effectively perform their own caching.

Existing systems

Here we survey some existing systems and their support for the above capabilities. Table 1 summarizes these systems.

One of the first commercial multiprocessor file systems is the Concurrent File System (CFS) [Pie89, FPD93, PFDJ89] for the Intel iPSC and Touchstone Delta multiprocessors [Int88]. CFS declusters files across several I/O processors, each with one or more disks. It provides the user with several different access modes, allowing different ways of sharing a common file pointer. Unfortunately, caching and prefetching are completely out of the control of the user, and the pattern for declustering the file across disks is not predictable and mostly out of the user's control.

Its designers claim that the Parallel File System (PFS) for the Intel Paragon supports our list of capabilities [Rul93], but we have not had the opportunity to verify this claim. We note, however, that the Paragon does not maintain parity across I/O nodes. Instead, each I/O node controls a separate RAID-level-three disk array, which maintains its own parity information independent of all other I/O nodes. Whereas a complete Paragon system may have many physical disks, the local

[1] There is potential for confusion here. Fully striped operations are based on the block size, which may or may not correspond to the striping unit size. The term "fully striped," however, is standard in the literature.

System	Control over declustering	Querying configuration	Independent I/O	Turn off caching	Turn off parity
Intel CFS	limited	limited	yes	no	n/a
Paragon PFS	yes	yes	limited	yes	limited
nCUBE (old)	yes	limited	yes	no	n/a
nCUBE (current)	yes	limited	yes	no	n/a
KSR-1	no?	?	limited	no	limited
MasPar	no	yes	no	no	no
TMC DataVault	no	yes	no	no	no
TMC SDA	no	yes	no	no	no
IBM Vesta	yes	yes	yes	no	n/a

Table 1: Some existing systems and whether they support our list of capabilities. We are not sure about support for declustering control and configuration querying in the KSR-1.

RAID level three organization limits the disk array at each I/O node to only fully striped I/O. The apparent number of independent disks, therefore, is only the number of I/O nodes, rather than the larger number of physical disks.

The first file system for the nCUBE multiprocessor [PFDJ89] gives plenty of control to the user. In fact, the operating system treats each disk as a separate file system and does not decluster individual files across disks. Thus, the nCUBE provides the low-level access one needs, but no higher-level access. The current nCUBE file system [dBC93] supports declustering and does allow applications to manipulate the striping unit size and distribution pattern.

The file system for the Kendall Square Research KSR-1 [KSR92] shared-memory multiprocessor declusters file data across disk arrays attached to different processors. The memory-mapped interface uses virtual memory techniques to page data to and from the file, which does not provide sufficient control to an application trying to optimize disk I/O.

Reads and writes in the Thinking Machines Corporation's DataVault [TMC91] are controlled directly by the user. Writes must be fully striped, however, thus limiting some algorithms. Neither the file system for the newer Scalable Disk Array [TMC92, LIN+93] nor the file system for the MasPar MP-1 and MP-2 [Mas91, Mas92] support independent I/O as we have defined it.[2]

IBM's Vesta file system [CBF93] for its Vulcan prototype multiprocessor supports many of the capabilities we require. Users can control the declustering of a file when it is created, specifying the number of disks, record size, and stripe-unit size. It is not clear whether a program may query to find out the available memory or a file's declustering information. All I/O is independent, and there is no support for parity (they depend on checkpoints for reliability).

3 Justification

In this section, we justify the capabilities of parallel file systems and disk I/O architectures that we claimed to be necessary in Section 2. Our justification is based on both theoretical and empirical grounds.

[2]These systems use RAID level three, which serializes what look to the programmer like independent writes.

Theoretical grounds

Several algorithms for parallel disk systems have been developed recently. These algorithms, which are oriented toward out-of-core situations, are asymptotically optimal in terms of the number of parallel disk accesses. They solve the following problems:

Sorting: Vitter and Shriver [VS90, VS92] give a randomized sorting algorithm, and Nodine and Vitter [NV91, NV92] present a deterministic sorting algorithm.

General permutations: Vitter and Shriver [VS90, VS92] use their sorting algorithm to perform general permutations by sorting on target addresses.

Bit-defined permutations: Cormen [Cor92, Cor93] presents algorithms to perform bit-defined permutations often with fewer parallel I/O operations than general permutations. This class of permutations includes *BPC* (bit-permute/complement) permutations, in which each target address is formed by applying a fixed permutation to the bits of a source address and then complementing a fixed subset of the resulting bits. Among the useful BPC permutations are matrix transpose[3] with dimensions that are powers of 2, bit-reversal permutations, vector-reversal permutations, hypercube permutations, and matrix reblocking. Cormen and Wisniewski [CW93] present an asymptotically optimal algorithm for *BMMC* (bit-matrix-multiply/complement) permutations, in which each target address is formed by multiplying a source address by a matrix that is nonsingular over $GF(2)$ and then complementing a fixed subset of the resulting bits. This class includes all BPC permutations, Gray code permutations, and inverse Gray code permutations.

General matrix transpose: Cormen [Cor92] gives an asymptotically optimal algorithm for matrix transpose with arbitrary dimensions, not just those that are powers of 2.

Fast Fourier Transform: Vitter and Shriver [VS90, VS92] give an asymptotically optimal algorithm to compute an FFT.

Matrix multiplication: Vitter and Shriver [VS90, VS92] cover matrix multiplication as well.

LU decomposition: Womble et al. [WGWR93] sketch an LU-decomposition algorithm.

These algorithms have the following characteristics:

- They solve important and interesting problems.
- They are designed for a parallel disk model based on control over declustering, knowledge of the configuration, independent I/O, and no parity, file caching, or prefetching.
- They are asymptotically optimal in this model. That is, their parallel I/O counts match known lower bounds for the problems they solve to within a constant factor.
- Several of them are practical in that the constant factors in their parallel I/O counts are small integers.
- Although the algorithms, as described in the literature, appear to directly access disk blocks, it is straightforward to modify them to access blocks within files instead.

[3] Vitter and Shriver earlier gave an algorithm for matrix transpose.

The parallel disk model used by these algorithms was originally proposed by Vitter and Shriver [VS90, VS92]. The cost measure is the number of parallel I/O operations performed over the course of a computation. The model does not specify the memory's organization, connection to the disks, or relation to the processors, and so it is independent of any particular machine architecture. Moving or manipulating records solely within the physical memory is free. The cost measure focuses on the amount of traffic between the memory and the parallel disk system, which is the dominant cost.

Note that these algorithms are asymptotically optimal over all SIMD or MIMD algorithms. The lower-bound proofs make no distinction between SIMD and MIMD; they simply count the number of times that any algorithm to solve a problem must access the parallel disk system.

Asymptotically optimal algorithms require independent parallel I/O. Restricting the I/O operations to be fully striped is equivalent to using just one disk whose block size is multiplied by the number of disks. It turns out that the constraint of fully striped I/O increases the number of disk accesses by more than a constant factor compared to independent I/O [VS90, VS92]. Disk accesses are expensive enough; to increase their number by more than a constant factor for large amounts of data can be prohibitively expensive.

The algorithms treat all physical memory uniformly; there is no distinct file cache. They carefully plan[4] their own I/O patterns so as to minimize traffic between the parallel disk system and the memory. File caching, and hence cache-consistency mechanisms, are unnecessary because the algorithms are already making optimal use of the available memory. In effect, the algorithms perform their own caching.

Empirical grounds

Several empirical studies of multiprocessor file system performance have found that common file access patterns do not always fit well with the underlying file system's expectations, leading to disappointing performance. Therefore, the basic file system interface should include primitives to control file declustering, caching, and prefetching.

The performance of Intel's CFS when reading or writing a two-dimensional matrix, for example, depends heavily on the layout of the matrix across disks and across memories of the multiprocessor, and also on the order of requests [dBC93, BCR92, Nit92, GP91, GL91]. del Rosario et al. [dBC93] find that the nCUBE exhibits similar inefficiencies: when reading columns from a two-dimensional matrix stored in row-major order, read times increase by factors of 30–50. One solution is to transfer data from disk into memory and then permute it within memory to its final destination [dBC93]. Nitzberg [Nit92] shows that some layouts experience poor performance on CFS because of thrashing in the file system cache. His solution to this problem carefully schedules the processors' accesses to the disks by reducing concurrency [Nit92]. Each of these examples highlights the need for programs to organize their I/O carefully. To do so, we must be able to discover and control the I/O system configuration.

Grimshaw et al. make many of the same arguments for their ELFS file system [GP91, GL91]. ELFS is an extensible file system, building object-oriented, operation-specific classes on top of a simple set of file access primitives. ELFS leaves decisions about declustering, caching, and prefetching to the higher-level functions, which have a broader understanding of the operation. Asynchronous I/O primitives are necessary for these libraries to perform prefetching and parallel I/O operations.

[4] The literature sometimes employs the more colorful term "choreograph."

4 Interface

In Sections 2 and 3, we argued that a multiprocessor file system must provide sufficient control to allow user-level applications to control file declustering, caching, prefetching, and parity, because a higher-level understanding of the application I/O patterns can lead to significant, even asymptotic, performance gains. Without detailing a specific file system interface (although some of our ideas are given in [Kot93]), we propose an interface with two personalities.

Low-level primitive operations

The primitive operations provide the "traditional" file system interface, such as basic read, write, and seek operations. The file system provides default declustering, caching, prefetching, and parity, making this interface sufficient for many simple applications. In addition, the interface includes primitives implementing all the capabilities listed in Section 2. Most current systems lack this degree of control.

High-level operations

Operations such as sorting, FFT, file copy, matrix transpose, and matrix transfer between distributed disks and distributed memories are programmed using the appropriate algorithms (Section 3), tuned for the particular architecture and combined into an I/O library. The library can be invoked either directly by the user or by a smart compiler, much like the LINPACK suite of numerical algorithms [DBMS79]. This library depends on the existence of the above primitive operations for detailed control of I/O.

5 Parity

We claimed in Section 2 that parallel file systems should be able to turn off parity or other redundancy information on a per-file basis. This section shows why we want to do so. Because we maintain parity to improve data reliability, this section also describes typical situations in which we can turn off parity without compromising data reliability.

The cost of maintaining parity

Patterson, Gibson, and Katz [PGK88] outline various RAID (Redundant Arrays of Inexpensive Disks) organizations. RAID levels four and five support independent I/Os. Both use check disks to store parity information.

In level four, the parity information is stored on a single dedicated check disk. If all parallel writes are fully striped, parity maintenance entails no additional disk accesses. Why? First, all the information needed to compute parity is drawn from the data to be written, and so no further information needs to be read to compute the parity. Second, each block written on the check disk is considered to be part of a stripe, and so each check-disk block is written concurrently with the rest of its stripe. When parallel writes are independent, however, maintaining parity information in RAID level four often entails extra disk accesses. The blocks are still striped across the disks. When writing some, but not all, the blocks in a stripe, we incur the additional expense of reading the old values in these blocks and the old parity values in order to compute the new parity values. Moreover, the check disk becomes a bottleneck. For each block written, the check disk in its stripe must be written as well. In a write to blocks in k different stripes, parity maintenance causes k serial accesses to the check disk.

In RAID level five, also known as "rotated parity," the data and parity information are distributed across all the disks. The cost of independent writes is lower than for level four, since the check disk is no longer as severe a bottleneck. Level five still suffers from three performance disadvantages for independent writes, however. First, the additional read of the old data block and old parity block is still necessary to compute the new parity block. Second, any individual disk can still be a bottleneck in a write if it happens to store parity blocks corresponding to more than one of the data blocks being written. Third, the block addresses are moved to accommodate the rotated parity information. The logical location of a block within a stripe might not match its physical location, especially when file system block allocation policies hide physical stripe locations from the application. This mismatch can complicate the algorithms of Section 3, which carefully plan so that when several blocks are accessed at once, they are on distinct disks.

Turning off parity safely

Systems maintain parity to enhance data reliability. When parity is maintained correctly, if a disk fails, its contents can be reconstructed from the remaining disks.

Although reliability is important for permanent data files, it is much less important for temporary data files. By *temporary*, we mean that the lifetime of the file is solely within the course of the application execution. For example, several of the algorithms listed in Section 3 perform multiple passes over the data. Each pass copies the data from one file to another, reordering or modifying the data. With the possible exceptions of the input file for the first pass and the output file for the last pass, all other files are temporary from the point of view of these algorithms.

What is the cost of a disk failure during a computation that uses only temporary files? The computation needs to be restarted from the last point at which parity information was maintained. We call this time a *paritypoint*, by analogy to the term "checkpoint." Disks definitely do fail, but only rarely. Therefore, it pays to avoid the cost of maintaining parity all the time for the rarely incurred cost of restarting the computation from the last paritypoint. Note that once any file has been written to disk, we can choose to paritypoint it at the cost of just one pass.

Furthermore, if a temporary file is written solely in full stripes, paritypointing is free for that file. This observation is significant because some of the algorithms listed in Section 3 perform some of their passes with fully striped writes. For example, the BPC algorithm mentioned in Section 3 alternates passes that use independent I/O with passes that use fully striped I/O. Every other pass, therefore, can paritypoint its output file as it is produced.

Turning off parity alleviates the problems of RAID level four and the first two problems of level five but not the third level-five problem: the alteration of block addresses due to rotated parity. Consequently, for the out-of-core algorithms, we prefer RAID level four with the capability to turn off parity. We note, however, that turning off parity in a RAID system is generally more than just a software issue—parity maintenance and error recovery are usually performed by the RAID controller. To turn off parity in a RAID level four disk array, the controller would need to keep track of which stripes are within temporary files so that it does not try to maintain their parity or to reconstruct their contents from garbage on the parity disk in case of a disk failure.

6 Conclusion

Since many high-performance parallel applications depend heavily on I/O, whether for out-of-core operations on large data sets, loading input data, or writing output data, multiprocessors must have high-performance file systems. Obtaining maximum performance, however, requires a careful interaction between the application, which has an understanding of the high-level operations, and

the I/O subsystem, which has an understanding of the architecture's capabilities. Many high-level operations can gain significant, even asymptotic, performance gains through careful choreography of I/O operations. We know of algorithms for many complex high-level operations, such as sorting, FFT, and matrix transpose, but also for simpler operations such as reading an input matrix into distributed memories.

We argue that the file system of a high-performance multiprocessor should include both the typical primitive operations such as read and write, as well as a library of high-level operations that optimize I/O. For these operations to be successful, the primitives must include querying about the configuration, control over declustering, independent I/O, and turning off parity, file caching, and prefetching. In short, the file system may provide default strategies, but the programmer must be able to override them when higher-level knowledge so dictates.

Acknowledgments

Thanks to Michael Best, Peter Corbett, Mike del Rosario, and Ernie Rael for their help in clarifying the capabilities of existing file systems.

References

[BCR92] Rajesh Bordawekar, Alok Choudhary, and Juan Miguel Del Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. Technical Report SCCS-420, NPAC, Syracuse University, 1992. To appear, 1993 International Conference on Supercomputing.

[CBF93] Peter F. Corbett, Sandra Johnson Baylor, and Dror G. Feitelson. Overview of the Vesta parallel file system. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 1–16, 1993.

[Cor92] Thomas H. Cormen. *Virtual Memory for Data-Parallel Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992. Available as Technical Report MIT/LCS/TR-559.

[Cor93] Thomas H. Cormen. Fast permuting in disk arrays. *Journal of Parallel and Distributed Computing*, 17(1–2):41–57, January and February 1993.

[CW93] Thomas H. Cormen and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. To appear in SPAA '93, January 1993.

[dBC93] Juan Miguel del Rosario, Rajesh Borawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993.

[DBMS79] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart. *LINPACK User's Guide.* SIAM, Philadelphia, PA, 1979.

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

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

[GP91] Andrew S. Grimshaw and Jeff Prem. High performance parallel file objects. In *Sixth Annual Distributed-Memory Computer Conference*, pages 720–723, 1991.

[Int88] iPSC/2 I/O facilities. Intel Corporation, 1988. Order number 280120-001.

[Kot93] David Kotz. Multiprocessor file system interfaces. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 194–201, 1993.

[KSR92] KSR1 technology background. Kendall Square Research, January 1992.

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

[Mas91] MP-1 family: Massively parallel computers. MasPar Computer Corporation brochure number PL014.0691, 1990,1991.

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

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

[NV91] Mark H. Nodine and Jeffrey Scott Vitter. Large-scale sorting in parallel memories. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 29–39, July 1991.

[NV92] Mark H. Nodine and Jeffrey Scott Vitter. Optimal deterministic sorting on parallel disks. Technical Report CS-92-08, Department of Computer Science, Brown University, 1992.

[PFDJ89] Terrence W. Pratt, James C. French, Phillip M. Dickens, and Stanley A. Janet, Jr. A comparison of the architecture and performance of two parallel file systems. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 161–166, 1989.

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

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

[Rul93] Brad Rullman, April 1993. Private communication.

[TMC91] Thinking Machines Corporation, Cambridge, Massachusetts. *Connection Machine I/O System Programming Guide*, October 1991.

[TMC92] CM-5 scalable disk array. Thinking Machines Corporation glossy, November 1992.

[VS90] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Optimal disk I/O with parallel block transfer. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 159–169, May 1990.

[VS92] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. Technical Report CS-92-04, Department of Computer Science, Brown University, August 1992. Revised version of Technical Report CS-90-21.

[WGWR93] David Womble, David Greenberg, Stephen Wheat, and Rolf Riesen. Beyond core: Making parallel computer I/O practical. In *DAGS '93*, June 1993.