3-31-1993

Integrating Theory and Practice in Parallel File Systems

Thomas H. Cormen
Dartmouth College

David Kotz
Dartmouth College

Integrating Theory and Practice in Parallel File Systems

Thomas H. Cormen David Kotz

Department of Computer Science
Dartmouth College
Hanover, NH 03755

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 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, turning off file caching and prefetching, and bypassing parity. We summarize recent theoretical and empirical work that justifies the need for these capabilities.

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

Tom Cormen (thc@cs.dartmouth.edu) is supported in part by funds from Dartmouth College and in part by the National Science Foundation under Grant CCR-9308667. David Kotz (dfk@cs.dartmouth.edu) is supported in part by funds from Dartmouth College and in part by the NASA Ames Research Center under Agreement Number NCC 2-849.

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.

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 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. Maintaining parity for data reliability on parallel disk systems exacts a performance cost, and Section 4 shows that for several parallel-I/O-based algorithms, we can dramatically reduce the cost of maintaining parity information. Finally, Section 5 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, turning off file caching and prefetching, and bypassing parity. 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 3 [PGK88]) or equal to the block size (as in RAID levels 4 and 5). Some systems, such as Vesta [CFPB93], allow the user to define the striping unit size.

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.

Bypassing parity

Another necessary capability is that of bypassing parity or other redundancy management for large temporary files. Section 4 examines why bypassing 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 turning off 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.

The Parallel File System (PFS) for the Intel Paragon [Roy93] supports our list of capabilities [Rul93], with a few restrictions. While it appears that caching can be disabled on the compute node, I/O-node caching is always active. Declustering parameters are determined on a per-filesystem basis. Finally, the Paragon does not maintain parity across I/O nodes. Instead, each I/O node controls a separate RAID 3 disk array, which maintains its own parity information independent

[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	Bypass 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	yes	limited	no	limited
TMC DataVault	no	yes	no	no	no
TMC SDA	no	yes	no	no	no
MasPar	no	yes	no	no	no
IBM Vesta	yes	yes	yes	no	n/a
Meiko CS-2	yes	yes	yes	yes	yes
Hector HFS	yes	yes	yes	yes	yes

Table 1: Some existing systems and whether they support our list of capabilities.

of all other I/O nodes. Whereas a complete Paragon system may have many physical disks, the local RAID 3 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 [Dd93, 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. Like the Intel Paragon, each disk array is a RAID 3 [For94]. The KSR-1, however, has an unalterable striping unit size. 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. To bypass parity within a RAID 3, one can deliberately "fail" the RAID parity disk, but then parity is lost for all files, not just large temporary files.

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, BGST93] nor the file system for the MasPar MP-1 and MP-2 [Mas92] support independent I/O as we have defined it.[2]

IBM's Vesta file system [CFPB93] for the SP-1 and SP-2 multiprocessors 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. A program may query to find out a file's declustering information [CF94]. All I/O is independent, and there is no support for parity (they depend on checkpoints for reliability).

The Parallel File System (PFS) for the Meiko CS-2 [Mei93, Mei94] apparently supports all of our required capabilities [Bar94]. In Meiko's PFS, separate file systems run under UFS (the Unix file system) on multiple server nodes. One server node stores directory information, and all

[2]These systems use RAID 3, which serializes what look to the programmer like independent writes.

others store data. The directory information includes the declustering method, for which there is a default that the programmer may override. The programmer may query about the configuration. The file-access interface allows the programmer to access individual blocks, given knowledge of the declustering method, and so it supports independent I/O. There is no hardware parity maintained across the separate file systems, although one is free to configure the system with a RAID at each node (like the Paragon). Meiko's PFS was designed to support Parallel Oracle, which has its own cache management mechanism; therefore, PFS allows client nodes to turn off file caching. Server nodes, however, must use the file caching of UFS.

The Hurricane File System (HFS) [Kri94] for the Hector multiprocessor at the University of Toronto supplies all of our requirements. HFS uses an object-oriented building-block approach to provide flexible, scalable high performance. Indeed, HFS appears to be one of the most flexible parallel file systems available, allowing users to independently control (or redefine) policies for prefetching, caching, redundancy and fault tolerance, and declustering.

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.

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 [VS94] give a randomized sorting algorithm, and Nodine and Vitter [NV91, NV92] present two deterministic sorting algorithms.

General permutations: Vitter and Shriver [VS94] use their sorting algorithm to perform general permutations by sorting on target addresses.

Mesh and torus permutations: Cormen [Cor92] presents algorithms for mesh and torus permutations, in which each element moves a fixed amount in each dimension of a multidimensional grid. In fact, these permutations are just special cases of monotonic and k-monotonic routes. In a *monotonic route*, all elements stay in the same relative order. A *k-monotonic route* is the superposition of k monotonic routes. All of the above permutations can be performed with fewer parallel I/Os than general permutations.

Bit-defined permutations: Cormen, Sundquist, and Wisniewski [Cor92, Cor93, CSW94] present algorithms to perform BMMC permutations often with fewer parallel I/O operations than general permutations. In a *BMMC* (bit-matrix-multiply/complement) permutation, 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 Gray code permutations and inverse Gray code permutations, as well as the useful class of *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

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

dimensions that are powers of 2, bit-reversal permutations, vector-reversal permutations, hypercube permutations, and matrix reblocking.

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 [VS94] give an asymptotically optimal algorithm to compute an FFT.

Matrix multiplication: Vitter and Shriver [VS94] cover matrix multiplication as well.

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

Computational geometry: Goodrich et al. [GTVV93] present algorithms for several computational-geometry problems, including convex hull in 2 and 3 dimensions, planar point location, all nearest neighbors in the plane, rectangle intersection and union, and line segment visibility from a point.

Graph algorithms: Chiang et al. [CGG+94] present algorithms for several graph problems.

These algorithms have the following characteristics:

- They solve problems that arise in real applications.
- 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.

The parallel disk model used by these algorithms was originally proposed in 1990 by Vitter and Shriver [VS90]. (See [VS94] for a more recent version.) 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 unless there are very few disks [VS94]. 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 (the literature sometimes employs the more colorful term "choreograph") their own I/O patterns so as to minimize traffic between the parallel disk system and the memory. File caching is 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, BCR93, 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. Both problems may be solved with a technique known as disk-directed I/O, in which the high-level I/O request is passed through to the I/O nodes, which then arrange for blocks of data to be transferred between disks and memory in a way that is efficient in terms of the number and order of disk accesses [Kot94].

Each of these examples highlights the need for programs to organize their I/O carefully. To do so, we must have file-system primitives to discover and control the I/O system configuration. The ELFS file system is based on this principle [GP91, GL91]. ELFS is an extensible file system, building object-oriented, application-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.

4 Parity

We claimed in Section 2 that parallel file systems should be able to bypass parity or other redundancy information for large temporary files. 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 bypass consistent parity maintenance 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 4 and 5 support independent I/Os. Both use check disks to store parity information.

In RAID 4, the parity information is stored on a single dedicated check disk. If all parallel writes are fully striped and the RAID controller knows so, then parity maintenance need not entail 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. If the RAID

controller knows that it is writing an entire stripe, it can write each check-disk block concurrently with the rest of its stripe. When parallel writes are independent, however, maintaining parity information in RAID 4 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 $2k$ serial accesses to the check disk (k to read old parity, and k to write new parity).

In RAID 5, also known as "rotated parity," the data and parity information are distributed across all disks. The cost of independent writes is lower than for RAID 4, since the check disk is no longer as severe a bottleneck. RAID 5 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.[4]

The third problem with RAID 5 is that the physical and logical locations for a data block may differ. In a left-symmetric RAID 5 organization [HG92], for example, there are D logical disks to hold data but $D + 1$ physical disks, with parity blocks spread among them. In this organization, the mapping from where one expects a block to be to where it actually goes appears to be relatively straightforward. We expect the ith logical block to reside on disk i, modulo the number of disks, but possibly at a different location on the disk [Gib93]. The phrase "modulo the number of disks" is open to interpretation, however, since there is one more physical disk than logical disks. The left-symmetric RAID 5 interpretation uses the number of physical disks.

There are two sources of trouble here. One is that some of the optimal algorithms (e.g., those in [Cor93, CSW94]) compute disk numbers and relative locations on disks mathematically, based in part on the number of logical—not physical—disks. The algorithms use these disk numbers to generate an independent I/O that accesses exactly one block on each logical disk. Blocks believed by the algorithm to be on different disks, however, may reside on the same physical disk, resulting in unexpectedly long access times. The other source of trouble is that the mapping for a specific block depends on its exact location on the disk, rather than its relative location within a file. Even if an algorithm were to try to compensate for this mapping, it would need to know the block's exact location. This information might not be available to the algorithm when file system block-allocation policies hide physical locations from the application.

One might be tempted to use parity logging [SGH93] to alleviate RAID 5 parity-mapping problems. Parity logging, however, further complicates the mapping between logical and physical locations. Moreover, it requires us to dedicate some physical memory to hold several track images of parity log information prior to writing it out to disks. We prefer to use physical memory to hold data rather than large amounts of parity information.

[4] A standard probability argument shows that this situation is likely. If we randomly write a single data block to a RAID 5 with D logical disks, and hence $D + 1$ physical disks, each of the D disks not holding the data block has a $1/D$ probability of holding the parity block associated with this data block. If we randomly write D blocks to D logical disks, one can show that although on average each disk contains at most one parity block that must be updated, with high probability there exists a disk containing $\Theta(\log D / \log \log D)$ parity blocks to update. The cost of updating the parity blocks in a single independent write in RAID 5, therefore, is not constant but in fact increases with the number of disks. If, however, we randomly write $D \log D$ blocks to D logical disks, the average disk contains at most $\log D$ parity blocks to update. Moreover, with high probability each disk contains only $O(\log D)$ parity blocks to update. If independent writes are batched into groups of $\log D$, therefore, the cost of updating parity blocks in RAID 5 is "only" a constant factor.

Bypassing 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 relatively 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 BMMC algorithm mentioned in Section 3 can be modified to use fully striped writes, and so each pass can paritypoint its output file as it is produced.

Bypassing parity alleviates the problems of RAID 4 and the first two problems of RAID 5 but not the third RAID 5 problem: the alteration of block addresses due to rotated parity. Perhaps the best solution, therefore, is to store temporary files in a separate RAID 0 partition, i.e., a partition of the storage system that has been configured without parity [Gib93]. It would need $D+1$ physical disks if we wish to use D logical disks with occasional paritypointing.

Another solution is to use RAID 4 with the capability to turn off parity on a per-file basis. This capability is not difficult to include in a software implementation of RAID. It is more difficult to turn off parity on a per-file basis in a hardware implementation of RAID, where parity maintenance and error recovery are performed by a controller. One solution is for the controller to support selectable parity on a per-stripe, per-track, or per-cylinder basis, and for the file system to allocate files on these boundaries, choosing the appropriate parity level for each allocation unit.

5 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.

In this paper, we have argued that for a parallel file system interface to be successful, its primitives must include querying about the configuration, control over declustering, independent I/O, and turning off file caching and prefetching, and bypassing parity. The file system may provide

default strategies, but the programmer must be able to override them when higher-level knowledge so dictates.

This paper has not proposed any specific file-system interface for out-of-core algorithms using parallel I/O. There are several proposed interfaces that support the specific operations of reading and writing matrices [BdC93, GGL93, Mas92].

The authors are aware of two projects that define more general interfaces providing the required capabilities listed in this paper. Researchers at Sandia National Laboratories [Shr94] are developing a library of C-callable, low-level, synchronous, parallel-I/O functions to run on the Intel Paragon under the SUNMOS operating system [MMRW94] on compute nodes and under OSF on I/O nodes. These functions perform synchronous reads and writes on blocks, full stripes, or entire memory loads, with the declustering method specified in the call. The other project is Vengroff's TPIE (Transparent Parallel I/O Environment) [Ven94], which is a C++ interface for synchronous parallel I/O. TPIE includes low-level functions to access disk blocks and manage memory, and it also includes higher-level functions that operate on streams of records by filtering, distributing, merging, permuting, and sorting.

Acknowledgments

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

References

[Bar94] Eric Barton. Private communication, August 1994.

[BCR93] Rajesh Bordawekar, Alok Choudhary, and Juan Miguel Del Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *International Conference on Supercomputing*, pages 367–376, 1993.

[BdC93] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993.

[BGST93] 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*, pages 489–495, 1993.

[CF94] Peter F. Corbett and Dror G. Feitelson. *Vesta File System Programmer's Reference*. IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, February 15 1994. Version 0.93.

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

[CGG+94] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms (extended abstract). Submitted to SODA '95, July 1994.

[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.

[CSW94] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dartmouth College Department of Computer Science, July 1994. Submitted to *IEEE Transactions on Parallel and Distributed Systems*. Preliminary version appeared in Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures.

[dBC93] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[Dd93] Erik P. DeBenedictis and Juan Miguel del Rosario. Modular scalable I/O. *Journal of Parallel and Distributed Computing*, 17(1–2):122–128, January and February 1993.

[For94] Rich Fortier, September 1994. Private communication.

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

[GGL93] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, 1993.

[Gib93] Garth A. Gibson. Private communication, June 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.

[GTVV93] Michael T. Goodrich, Jyh-Jong Tsay, Darren E. Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 714–723, November 1993.

[HG92] Mark Holland and Garth A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 23–35, October 1992.

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

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, November 1994. To appear.

[Kri94] Orran Krieger. *HFS: A flexible file system for shared-memory multiprocessors.* PhD thesis, University of Toronto, October 1994.

[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*, pages 291–305, 1993.

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

[Mei93] Meiko. *Elan Widget Library*, 1993.

[Mei94] Meiko. *CS-2 System Administration Guide*, 1994.

[MMRW94] Arthur B. Maccabe, Kevin S. McCurley, Rolf Riesen, and Stephen R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Proceedings of the Intel Supercomputer Users Group Conference*, pages 245–251, June 1994.

[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.

[Roy93] Paul J. Roy. Unix file access and caching in a multicomputer environment. In *Proceedings of the Usenix Mach III Symposium*, pages 21–37, 1993.

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

[SGH93] Daniel Stodolsky, Garth Gibson, and Mark Holland. Parity logging: Overcoming the small write problem in redundant disk arrays. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 64–75, May 1993.

[Shr94] Elizabeth A. M. Shriver. Private communication, July 1994. Paper in progress.

[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.

[Ven94] Darren Erik Vengroff. A transparent parallel I/O environment. In *Proceedings of the DAGS '94 Symposium*, pages 117–134, July 1994.

[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.

[VS94] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.

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