Dartmouth College Computer Science Technical Report
PCS-TR96-293
(Revised September 1996)

Early Experiences in Evaluating the Parallel Disk Model with the ViC* Implementation

Thomas H. Cormen*
Melissa Hirschl†

Dartmouth College
Department of Computer Science

{thc, hershey}@cs.dartmouth.edu

Abstract

Although several algorithms have been developed for the Parallel Disk Model (PDM), few have been implemented. Consequently, little has been known about the accuracy of the PDM in measuring I/O time and total running time to perform an out-of-core computation. This paper analyzes timing results on multiple-disk platforms for two PDM algorithms, out-of-core radix sort and BMMC permutations, to determine the strengths and weaknesses of the PDM.

The results indicate the following. First, good PDM algorithms are usually not I/O bound. Second, of the four PDM parameters, one (problem size) is a good indicator of I/O time and running time, one (memory size) is a good indicator of I/O time but not necessarily running time, and the other two (block size and number of disks) do not necessarily indicate either I/O or running time. Third, because PDM algorithms tend not to be I/O bound, using asynchronous I/O can reduce I/O wait times significantly.

The software interface to the PDM is part of the ViC* run-time library. The interface is a set of wrappers that are designed to be both efficient and portable across several underlying file systems and target machines.

1 Introduction

Since its introduction in 1990, the Parallel Disk Model (PDM) of Vitter and Shriver [VS90, VS94] has become the predominant model for developing and analyzing algorithms that operate on data

*Supported in part by funds from Dartmouth College and in part by the National Science Foundation under Grants CCR-9308667 and CCR-9625894. Portions of this work were performed while visiting the Parallel Data Laboratory of the Carnegie Mellon University School of Computer Science.

†Supported in part by a Dartmouth College Fellowship and in part by the National Science Foundation under Grants CCR-9308667 and CCR-9625894.

To appear in *Parallel Computing*.

stored on multiple disks. Although a variety of PDM algorithms have appeared in the literature, there have been few implementations of any of them. (A notable exception is Vengroff's TPIE project [Ven94, Ven97].)

The ViC* project at Dartmouth will implement virtual memory for the data-parallel language C* [TMC93] with the PDM as its underlying abstract disk model. ViC* contains two major components: a compiler that takes a C* program with some variables declared `outofcore` and produces a C program with explicit I/O and library calls, and a run-time library that implements the I/O and library calls.

This paper presents our early experiences in evaluating how accurate a model the PDM is, based on two sophisticated algorithms for the PDM (radix sort and performing BMMC permutations [CSW94]) that we implemented with the ViC* run-time library. We ran these algorithms on two platforms: a uniprocessor with eight disks and a network of eight workstations.

To evaluate any computational model, one needs an implementation of that model and representative programs for the model. The ViC* implementation of the PDM is portable across a wide variety of hardware and software platforms. The two algorithms we have chosen are relatively easy to implement, and they are representative of PDM algorithms in the literature in that they perform multiple discrete passes over the data. Moreover, these algorithms are useful; for example, the BMMC permutation code is at the heart of an efficient out-of-core Fast Fourier Transform implementation [CN96].

The PDM is designed to measure I/O complexity. That is, it measures how many parallel disk accesses an algorithm makes, in terms of four parameters:

- N: the problem size,
- M: the random-access memory size,
- B: the size of each disk block,
- D: the number of disks.

Each parallel disk access reads or writes one block of B records from or to each of the D disks. Because disk-access latencies are so much higher than memory-access times (on the order of 10 milliseconds versus 100 nanoseconds, or a factor of about 100,000), the PDM places a premium on minimizing the number of parallel disk accesses.

On the other hand, the PDM measures *only* I/O complexity. Computation and interprocessor communication incur no cost in this model.

In reality, however, one might expect computation and interprocessor communication to be significant costs. Examination of most algorithms developed for the PDM, including the two in this paper, reveals that they tend to operate in discrete passes over the data by reading in a large amount of data in parallel, processing the data in memory (perhaps entailing interprocessor communication), and writing out the data in parallel. (The reads or writes are sometimes interspersed with the in-memory processing.) The I/O transfer size is typically the same each time for a given run of an algorithm and depends on the values of the four PDM parameters. Each parallel read and write, therefore, induces some computation and communication. The time for such a processing step is usually at least a linear function of the I/O transfer size.

Our implementation of the two algorithms yielded the following findings about the accuracy of the PDM:

- Good PDM algorithms are usually not I/O bound. That is, the sum of the computation and communication time is usually comparable to the I/O time and often exceeds it.

- The PDM parameter for the problem size (N) is a fairly good indicator of I/O time and total running time.

- The PDM parameter for the memory size (M) is a fairly good indicator of I/O time. It is also a good indicator of running time on a uniprocessor, but not as good on a multiprocessor.

- The PDM parameters for the block size (B) and number of disks (D) are poor indicators of both I/O time and total running time. The parameter D can be a good indicator of I/O time on a multiprocessor depending on the system configuration. Optimal values of these parameters depend on the underlying system, which limits the applicability of the PDM when the values of these parameters are varied.

We also found that using asynchronous I/O was an effective means of reducing I/O wait times in compute-bound programs. When I/O time is less than computation time, as is often the case in good PDM algorithms, asynchronous I/O hides most of the I/O latency.

The remainder of this paper is organized as follows. Section 2 presents the Parallel Disk Model and briefly surveys I/O-complexity results for it. Section 3 discusses how we designed the ViC* run-time library to implement the PDM efficiently and portably. Section 4 defines the class of BMMC permutations and outlines the algorithm for performing them on the PDM. Section 5 describes an out-of-core radix sort algorithm for the PDM. Section 6 presents extensive timing results for these two PDM algorithms on a uniprocessor with 8 data disks and for the BMMC algorithm on an 8-node network of workstations. Finally, Section 7 summarizes the timing results to evaluate how well the PDM models real systems.

2 The Parallel Disk Model

This section describes the Parallel Disk Model and gives some I/O complexity results for it.

Figure 1 shows the *Parallel Disk Model*, or *PDM*. In the PDM, N *records* are stored on D *disks* $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$, with N/D records stored on each disk. The records on each disk are partitioned into *blocks* of B records each.[1] When a disk is read from or written to, an entire block of records is transferred. Disk I/O transfers records between the disks and a *random-access memory* (which we shall refer to simply as "memory") capable of holding M records. Any set of M records is a *memoryload*. Each *parallel I/O operation* transfers up to D blocks between the disks and memory, with at most one block transferred per disk, for a total of up to BD records transferred. The most general type of parallel I/O operation is *independent I/O*, in which the blocks accessed in a single parallel I/O may be at any locations on their respective disks. A more restricted operation is *striped I/O*, in which the blocks accessed in a given operation must be at the same location on each disk.

The PDM is also notable for what it does not include. It does not specify how many processors there are, nor how they are connected, and it does not distinguish between shared and distributed

[1]A block might consist of several sectors of a physical device or, in the case of RAID [CGK+88, Gib92, PGK88], sectors from several physical devices.

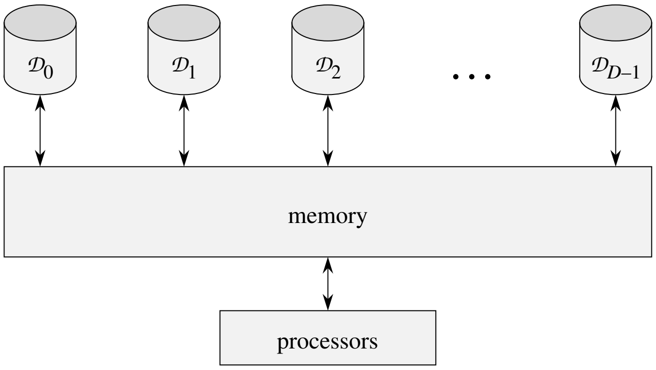


Figure 1: The Parallel Disk Model. Records are stored on disks $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$, with an equal number of records on each disk. The records on each disk are partitioned into blocks of B records each (not shown here). Disk I/O transfers records between disks and memory that can hold M records. Processor and memory organization are unspecified. An algorithm's cost is the number of parallel I/O operations, each of which transfers one block per disk.

	$\mathcal{D}_0$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3$		$\mathcal{D}_4$		$\mathcal{D}_5$		$\mathcal{D}_6$		$\mathcal{D}_7$	
stripe 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
stripe 1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
stripe 2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
stripe 3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Figure 2: The layout of $N = 64$ records in a parallel disk system with $B = 2$ and $D = 8$. Each box represents one block. The number of stripes is $N/BD = 4$. Numbers indicate record indices.

memories. Additionally, as we noted in Section 1, the PDM does not consider the time to process data in memory as part of its cost model.

We measure an algorithm's efficiency by the number of parallel I/O operations it requires. Although this cost model does not account for the variation in disk access times caused by head movement and rotational latency, programmers often have no control over these factors. The number of disk accesses, however, can be minimized by carefully designed algorithms. Optimal algorithms have appeared in the literature for fundamental problems such as sorting [Arg95, BGV96, NV93, NV95, VS94], general permutations [VS94], and structured permutations [Cor92, Cor93, Wis96], as well as higher-level domains such as Fast Fourier transform [CN96, VS94], matrix-matrix multiplication [VS94], LUP decomposition [WGWR93], computational geometry problems [AVV95, GTVV93], and graph algorithms [CGG$^+$95].

We place some restrictions on the PDM parameters. We assume that B, D, M, and N are exact powers of 2. In order for the memory to accomodate the records transferred in a parallel I/O operation to all D disks, we require that $BD \leq M$. Also, we assume that $M < N$, since otherwise we can just perform all operations in memory.

The PDM lays out data on a parallel disk system as shown in Figure 2. A *stripe* consists of the D blocks at the same location on all D disks. We indicate the index of a record as a $(\lg N)$-bit vector x with the least significant bit first: $x = (x_0, x_1, \ldots, x_{\lg N-1})$. Record indices vary most rapidly within a block, then among disks, and finally among stripes.

Since each parallel I/O operation accesses at most BD records, any algorithm that must access all N records requires $\Omega(N/BD)$ parallel I/Os, and so $O(N/BD)$ parallel I/Os is the analogue of linear time in sequential computing. Vitter and Shriver, followed by others, showed an upper bound of $\Theta\left(\frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)$ parallel I/Os for sorting. This bound is asymptotically tight, because it matches the lower bounds proven earlier by Aggarwal and Vitter [AV88] using a model with one disk and D independent read/write heads, which is at least as powerful as the PDM. The quantity $\Theta\left(\frac{\lg(N/B)}{\lg(M/B)}\right)$ represents the number of passes over the data required to sort.

The I/O complexity of $\Theta\left(\frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)$ appears to be the analogue of the $\Theta(N \lg N)$ bound seen for so many sequential algorithms on the standard RAM model. Not only does sorting have this tight I/O bound, but Vitter and Shriver also showed that Fast Fourier transforms do as well.

There are, however, problems that can be solved in linear time on the RAM model but require $\Theta\left(\frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)$ parallel I/Os on the PDM. One such problem is performing general permutations.

An algorithm that is efficient on the PDM has two characteristics. First, each access to a block uses all the records in that block. It is inefficient to incur the cost of a disk access just to access a few records in a block. Second, disk accesses are organized to balance the load evenly across the D disks. That is, the algorithm should perform parallel I/O operations as defined above. Ensuring that each set of D block accesses is for blocks on the D distinct disks is a challenge for the PDM algorithm designer.

3 The ViC* implementation of the PDM

ViC* is designed for efficient data-parallel computation on out-of-core data. It stores each out-of-core parallel variable in a separate file in some underlying file system, which may be a parallel file system. Implementation of out-of-core operations in ViC* is guided by two principles:

1. Out-of-core operations should be efficient in terms of the PDM.
2. The system should be portable to permit implementation across a wide variety of parallel computers and networks of workstations.

This section describes how we designed the ViC* run-time library with these considerations in mind.

Figure 3 shows the overall architecture of the ViC* run-time library. In addition to supporting out-of-core computation, the ViC* run-time system will include the entire C* run-time library for in-core parallel computation. The boxes at the top of the figure denote these large sets of functions. Both in-core and out-of-core functions will require interprocessor communication. We have defined a set of macros for interprocessor communication and implemented them for three underlying communication models: MPI [GLS94, SOHL+96], PVM [GBD+94], and a uniprocessor. Switching between models is as simple as recompiling with a different macro set and relinking with a different library.

The ViC* API

A more interesting question is how to implement the PDM abstraction in a portable fashion. We have done so by defining an application programmer interface (API) that we call the *ViC* API*. It

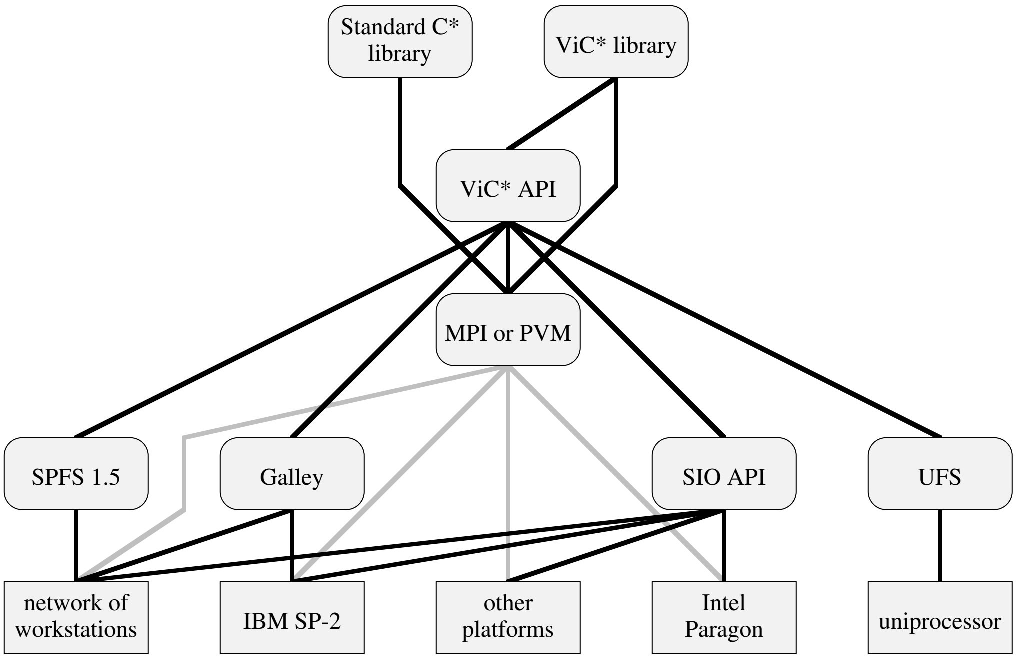


Figure 3: The overall architecture of the ViC* run-time library. The bottom row represents target machines, and all other layers represent software. The standard C* library for in-core computation uses MPI or PVM for interprocessor communication. Lightly shaded lines show some of the platforms that have MPI or PVM implementations. The ViC* library for out-of-core computation contains calls to MPI or PVM and the ViC* API. Wrappers for the ViC* API exist for SPFS 1.5 (which runs on a network of workstations), Galley (running on a network of workstations and the IBM SP-2), and the Unix file system (UFS) on a uniprocessor, and they will be written for the SIO API, which will run on several platforms.

is perhaps a misnomer, in that we do not expect application programmers to actually use it. We believe, however, that it will be useful in writing the ViC* run-time library. The ViC* compiler will produce direct calls to it as well.

The ViC* API is derived from two other interfaces. The primary influence is the Whiptail File System API [SW95], which was also designed to support the PDM. A secondary influence is the low-level file-system API currently being developed by the Scalable I/O (SIO) Initiative Working Group on Operating Systems.

As Figure 3 shows, a particular implementation of the ViC* API will be as a set of wrapper functions on top of an existing file system interface. We have already implemented it on three systems:

- Scotch Parallel File System (SPFS version 1.5) [GSC+95]:
  SPFS uses a client-server model in which computing processes are clients and servers provide access to storage. It provides a Unix-like linear view of each parallel file. SPFS runs on a network of workstations.

- Galley File System [NK96a, NK96b]:
  Like SPFS, Galley is a parallel file system that uses a client-server model. Unlike SPFS, Galley allows applications a more sophisticated view of the parallel file system. In particular,

its interface provides for access to specific disks. Galley was developed to run on a network of IBM RS6000 workstations and on the IBM SP-2, and it has also been ported to a network of DEC Alpha workstations (running Digital Unix) and to a network of PCs (running FreeBSD and Linux).

- Uniprocessor:
  The ViC* API runs on single DEC Alpha workstations, of both the desktop variety and a DEC 2100 server (named "adams") at Dartmouth. The DEC 2100 has two 175-MHz Alpha processors, 320 megabytes of shared memory, and nine 2-gigabyte disk drives of which eight are used for data. The disks are distributed among three SCSI chains on a DEC RAID controller. The operating system (Digital Unix V3.2D-1, a variant of OSF-1) may choose to run a ready thread on either CPU. The ViC* wrappers for adams are the same as for a single desktop workstation; there are no MPI or PVM calls. The underlying file system is UFS. Although the ViC* wrappers make no special use of the second processor on adams, they do spawn a thread for each disk to service I/O requests for that disk for the duration of the program. Consequently, there is a high degree of concurrency on adams as disk-server threads often run on a separate processor from the main computation thread. Each disk contains its own set of UFS files.

As of this writing, we have working implementations of the ViC* API for uniprocessors and for Galley. The Galley implementation is on "Fleet," a network at Dartmouth of eight IBM RS6000 nodes connected by a 100-Mbit/second FDDI network; we use the MPI versions of the communication macros. We have written the wrappers for SPFS 1.5 and successfully run programs with them, but problems with software outside the ViC* system temporarily prevent ViC* from working reliably with SPFS 1.5. Consequently, this paper contains no timing results for ViC* on top of SPFS 1.5.

The SIO interface is still being defined, and so no implementations of it yet exist. We expect the ViC* wrappers for the SIO API to be especially easy to write once the interface is finalized. Our understanding is that implementations for the IBM SP-2 and Intel Paragon are planned; because MPI or PVM already runs on these parallel machines, porting ViC* to them should be a simple task at some future date.

In the remainder of this section, we highlight some features of the ViC* API. We omit discussion of several straightforward file-management functions, including the `ViC_open()` function, which returns a file descriptor.

Configuration

The ViC* API provides to the run-time library an abstraction that each processor owns at least one disk. (Here, a "disk" might be a disk server, as in Galley or SPFS 1.5.) Figure 4 shows an example in which each of four processors owns two disks. ViC* provides this abstraction even when there are more processors than disks; in this case, multiple processors share a given disk. The only restriction is that the number of disks must be an integer multiple of the number of processors or vice versa.

Only underlying C types (`char`, `int`, `long int`, `double`, etc.) map to PDM records in the ViC* run-time library. The ViC* compiler breaks structures and arrays[2] into their underlying

[2] C* uses shapes, which are orthogonal to arrays, to express parallelism.

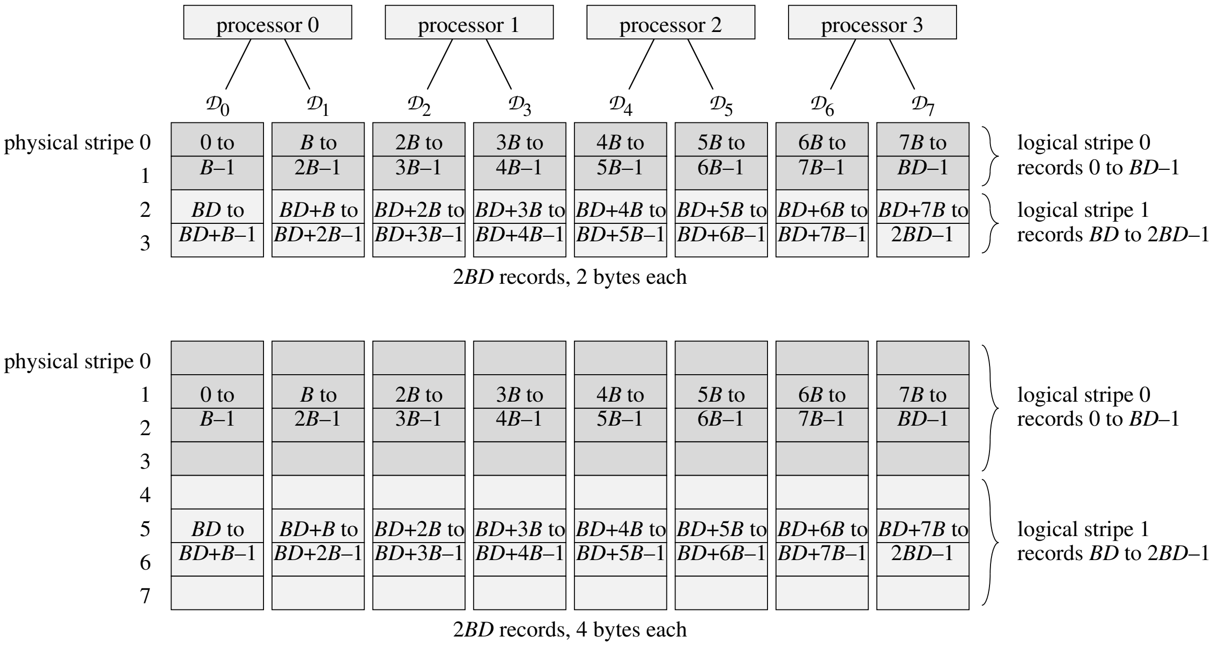


Figure 4: An example with $P = 4$ processors and $D = 8$ disks that shows how records of differing sizes map to disks in ViC*. Shown are two parallel variables. The top one has elements that are 2 bytes each, and the bottom one has 4-byte elements. Each shaded rectangle is a B-byte physical block, and each of the parallel variables occupies 2 logical stripes. Each logical block consists of 2 and 4 physical blocks for the 2-byte and 4-byte element types, respectively. For each of the parallel variables, elements 0 to $B-1$ and BD to $BD+B-1$ are on disk $\mathcal{D}_0$, elements B to $2B-1$ and $BD+B$ to $BD+2B-1$ are on disk $\mathcal{D}_1$, and so on.

components and stores the components separately. Records, therefore, are relatively small; the largest record is the size of a `long double`. Block sizes are typically quite a bit larger, starting at 512 bytes or so and going as high as the SPFS 1.5 striping unit size of 64 KB. Record sizes and block sizes are powers of 2. Consequently, each block contains multiple records, and each record resides on exactly one block.

Because not all record types are the same size and a computation might use records of differing sizes, we needed to devise a way of working with records of various but small sizes. To understand this issue, consider an index i into parallel variables. We want the processor number to which all records at index i map to be the same, regardless of the record size. For example, if we are casting a parallel variable `x` of 2-byte `short int` to a parallel variable `y` of 4-byte `int`, we need the ith elements of `x` and `y` to meet at the same processor. A further consideration is that the data layout across multiple disks should be close to the PDM order shown in Figure 2 so that we can sequentially access the elements of a parallel variable by accessing it stripe by stripe.

Figure 4 shows our solution. We distinguish between physical stripes and logical stripes. A *physical stripe* consists of one *physical block* of B bytes—not necessarily B records—from each disk.[3] A record whose size is k bytes resides in a *logical block* consisting of k consecutive physical blocks on the same disk. A logical block, therefore, contains B records. For a k-byte record size,

[3] Again, a physical block might actually consist of several disk sectors.

a *logical stripe* consists of k consecutive physical stripes or, equivalently, a set of logical blocks at the same k locations on their respective disks. The jth record in a logical block occupies bytes jk through $(j+1)k-1$. With this layout scheme, the processor and disk that a record maps to are a function of only the record index and the parameters P (the number of processors), D, and B. All read and write functions in the ViC* API take as parameters a logical stripe number and a record size so that they can determine physical stripe numbers; all calls to these functions ask for at least one logical block each time.

The small record size is necessary for this scheme to work in practice. The size of each logical block is the record size times the physical block size. If record sizes could be large, logical block sizes could be very large, and we might not be able to allocate space for many logical stripes before running out of memory on a typical workstation.

Striped access

The functions `ViC_read_stripes()` and `ViC_write_stripes()` provide synchronous access (they do not return until the I/O operation completes) to a set of consecutive logical stripes. Each takes the following parameters: a file descriptor of an open file, the number of the first logical stripe to access, the number of consecutive logical stripes to access, the record size in bytes, and the address of a buffer in memory to read into or write from.

When a processor calls `ViC_read_stripes()` or `ViC_write_stripes()`, it reads or writes its portion of the stripe on the disk(s) it owns. No interprocessor synchronization (i.e., barrier) is needed. Each processor runs at its own speed.

The functions `ViC_async_read_stripes()` and `ViC_async_write_stripes()` are asynchronous versions of the above functions. They take an additional parameter, which is a pointer to a "handle" that they give back to the caller. This handle is passed later on in a call to `ViC_async_status()` to wait for the operation to complete or to poll its status.

Independent access

The functions `ViC_read_indep()` and `ViC_write_indep()` provide synchronous, independent I/O. Like the striped-access functions, no interprocessor synchronization is needed. Each takes the following parameters: a file descriptor of an open file, a count of how many read or write requests are being made, the record size in bytes, an array of buffer addresses in memory, an array of disk numbers for this processor, and an array of logical stripe numbers. Each of the latter three arrays contains one entry for each I/O request, and each I/O request is for one logical block. There is no prohibition against repeating a disk number in the array of disk numbers, and so an I/O request might not be truly independent; we view this situation as a performance issue rather than one of correctness.

The functions `ViC_async_read_indep()` and `ViC_async_write_indep()` are the asynchronous independent-access functions. Like their synchronous counterparts, they take a pointer to a handle that is passed to `ViC_async_status()` later on.

4 The BMMC permutation algorithm for the PDM

In order to exercise the ViC* API's implementation of the PDM, we needed to implement a sophisticated PDM algorithm. Moreover, we wanted to use one that requires independent I/O for optimal performance. (If all parallel I/O operations are striped so that none are independent, then a RAID level 3 disk organization is efficient.) We chose the BMMC permutation algorithm of [CSW94]. This section defines the class of BMMC permutations, summarizes the BMMC permutation algorithm, and describes some issues in the implementation of the algorithm.

BMMC permutations

Any permutation is defined by a bijection of a set $\{0, 1, \ldots, N-1\}$ onto itself. We say that each *source index* in $\{0, 1, \ldots, N-1\}$ maps to a distinct *target index* in $\{0, 1, \ldots, N-1\}$. The mapping for a *BMMC* (bit-matrix-multiply/complement) permutation on N elements is specified by a $(\lg N) \times (\lg N)$ *characteristic matrix* $A = (a_{ij})$ whose entries are drawn from $\{0, 1\}$ and is nonsingular (i.e., invertible) over $GF(2)$.[4] The specification also includes a *complement vector* $c = (c_0, c_1, \ldots, c_{\lg N-1})$. Treating each source index x as a $(\lg N)$-bit vector, we perform matrix-vector multiplication over $GF(2)$ and then form the corresponding $(\lg N)$-bit target index y by complementing some subset of the resulting bits: $y = A\,x \oplus c$, or

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{\lg N-1} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0,\lg N-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1,\lg N-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2,\lg N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{\lg N-1,0} & a_{\lg N-1,1} & a_{\lg N-1,2} & \cdots & a_{\lg N-1,\lg N-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{\lg N-1} \end{bmatrix} \oplus \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{\lg N-1} \end{bmatrix} .$$

As long as the characteristic matrix A is nonsingular, the mapping of source indices to target indices is one-to-one.

Although not all permutations are BMMC, many permutations often encountered in practice are. The class of BPC (bit-permute/complement) permutations is the subclass of BMMC permutations in which the characteristic matrix is a permutation matrix, containing one 1 in each row and in each column. Matrix transpose (with power-of-2 dimensions), bit reversal (used in performing FFTs), vector reversal, and matrix reblocking are all BPC, and hence BMMC, permutations. Gray-code permutations, inverse Gray-code permutations, and permutations used by fast cosine transforms [MW95] are BMMC (but not BPC). BMMC permutations are closed under composition so that, for example, the composition of bit-reversal and Gray-code permutations is BMMC.

We generally focus on the matrix-multiplication portion of BMMC permutations rather than on the complement vector. A key technique used to perform BMMC permutations is factoring a characteristic matrix into multiple matrix factors, each of which is nonsingular and of a desired form. If we factor a characteristic matrix A as $A = A^{(k)}\,A^{(k-1)}\,A^{(k-2)} \cdots A^{(1)}$, then we can perform the BMMC permutation characterized by A by performing, in order, the BMMC permutations characterized by $A^{(1)}, A^{(2)}, \ldots, A^{(k)}$. That is, we perform the permutations characterized by the factors of a matrix from right to left.

Summary of the BMMC algorithm for the PDM

Each factor produced by the BMMC algorithm of [CSW94] characterizes a restricted form of BMMC permutation that can be performed in one pass over the data. (We refer the reader to [CSW94]

[4] Matrix multiplication over $GF(2)$ is like standard matrix multiplication over the reals but with all arithmetic performed modulo 2. Equivalently, multiplication is replaced by logical-and, and addition is replaced by exclusive-or.

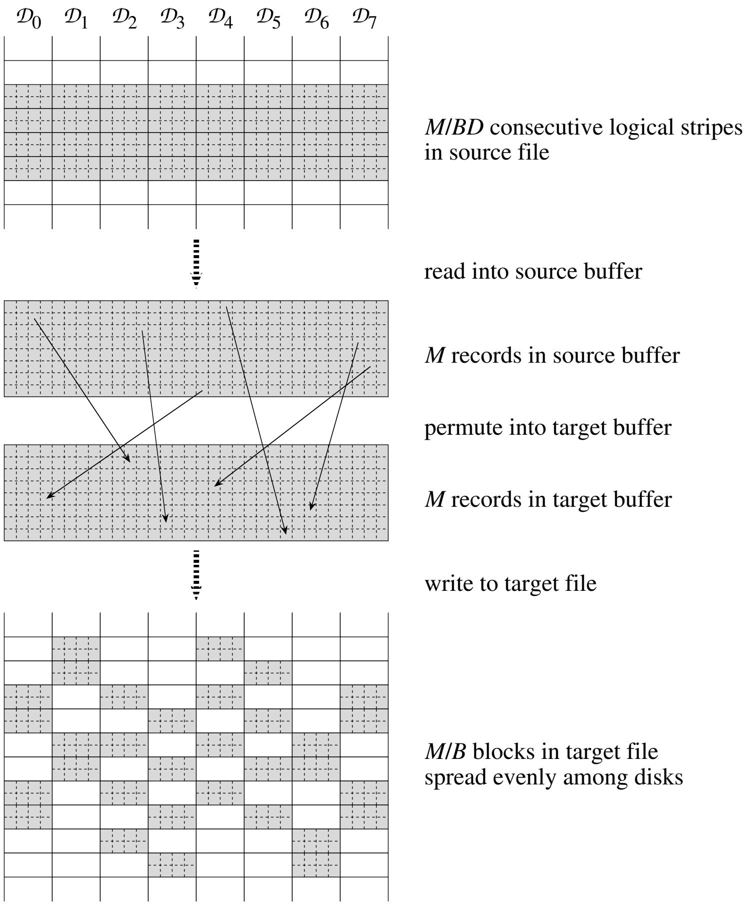


Figure 5: Processing a memoryload in an MLD permutation. Each small square represents a record. The M records are read from M/BD consecutive logical stripes of the source file into a source buffer in memory, permuted into a target buffer (to avoid clutter, we have drawn only some of the arrows showing this in-memory permutation), and written out to M/B blocks of the target file spread evenly among the disks.

for details about the forms of the matrix factors.) Each pass copies records from a *source file* to a *target file* stored across the D disks. Regardless of the number of matrix factors, only two files are needed over the course of the algorithm, since we can repeatedly swap the roles of the source and target files from pass to pass.

The restricted form of BMMC permutation used in the algorithm is known as *MLD* (memory-load-dispersal) because it can be performed as Figure 5 shows. Given an effective memory size of M records, process each of N/M memoryloads by reading in M records, permute them in memory, and write them out. In an MLD permutation, the M records read in reside on M/BD consecutive logical stripes of the source file, and so we can call `ViC_read_stripes()` or `ViC_async_read_stripes()` to read them. On the other hand, when they are written out to the target file, they are not necessarily in consecutive logical stripes. An MLD permutation has the desirable property, however, that the records written are clustered into M/B blocks that are evenly distributed among the disks, with

M/BD blocks on each disk. Hence we can call `ViC_write_indep()` or `ViC_async_write_indep()` to write them out.

By "effective" memory size, we mean that the size of each memoryload—and hence the value of the PDM parameter M for the algorithm—is smaller than the physical memory size. The reason is that once the records are read into memory, we need additional memory to process them. Records are read into a source buffer and permuted into a separate target buffer within each processor, from where they are written out. That reduces the effective memory size to half the physical memory size. If we are also using asynchronous I/O, we need to allocate two more memoryload-sized buffers: one to hold the memoryload that we are reading ahead, and one to hold the memoryload that we are writing behind. That further reduces the effective memory size to a quarter of the physical memory size.

The number of MLD factors is at most $\left\lceil \frac{\operatorname{rank}\gamma}{\lg(M/B)} \right\rceil + 2$, where γ is the lower left $\lg(N/B) \times \lg B$ submatrix of the characteristic matrix, and the rank is computed over $GF(2)$. (Note that because of the dimensions of γ, its rank is at most $\lg \min(N/B, B)$.) This number of factors is asymptotically optimal and is very close to the best known exact lower bound.

The last (i.e., leftmost) MLD factor has an even more restricted form. Not only does it read memoryloads by reading M/BD consecutive logical stripes, but it also writes them by writing M/BD consecutive logical stripes. Hence we can write them by calling `ViC_write_stripes()` or `ViC_async_write_stripes()`. We call such permutations MRC (memory-rearrangement/complement).

Implementation notes

We conclude this section by noting a couple of further implementation details. First, we have designed the interprocessor communication portion of each MLD factor to be efficient. All processors store a copy of the characteristic matrix for the MLD permutation, which means that by agreeing upon the order in which records are sent, only the records—and not their source or target indices—need be sent. This optimization saves on network bandwidth. Using factoring techniques like those in [CSW94], we order the interprocessor communication into rounds in which each sending processor sends to a unique receiving processor. We also designed a technique whereby the two processors agree on the order of the source indices of the records in the transmitted buffer.

A second optimization applies to both uniprocessor and multiprocessor systems. To move each record to its place in the target buffer, a processor must compute the record's target index from its source index. Done naively, this computation would require a matrix-vector multiplication of a $(\lg N) \times (\lg N)$ matrix by a $(\lg N)$-vector. This matrix-vector multiplication requires $\Theta(\lg^2 N)$ time, and so we would spend $\Theta(N \lg^2 N)$ time in each pass computing target indices. We can reduce this time to $\Theta(N)$ per pass, which is optimal. Observe that in any practical situation, $\lg N \leq 64$, and so we can store each matrix column in one or two `long ints`. We can exclusive-or a column into a $(\lg N)$-bit vector, also packed into one or two words, using $\Theta(1)$ bitwise operations. Next, note that we may choose source indices in any order. By choosing a Gray-code order, in which each source index differs from the previous one in only one bit position, each target index differs from the previous one by the exclusive-or of one matrix column. As we have just noted, that exclusive-or operation takes $\Theta(1)$ time. Moreover, we can compute which column to use in only $\Theta(1)$ amortized time; see [CB95] for details. Thus, we spend only $\Theta(1)$ amortized time per record in each pass.

5 Radix sort for the PDM

The other PDM algorithm that we have implemented and measured is an out-of-core, bucketized radix sort. This section describes the out-of-core radix sort algorithm and our implementation of it. This description is oriented toward a uniprocessor with multiple disks.

Several other sorting algorithms for the PDM have appeared in the literature [Arg95, BGV96, NV93, NV95, VS94]. These use independent I/O and are asymptotically optimal. By contrast, our radix sort algorithm is suboptimal but it uses only striped I/O. Vengroff [Ven97] has observed that for realistic ranges of PDM parameters, asymptotically optimal PDM algorithms often require more parallel I/Os than do suboptimal algorithms. This is because the asymptotically optimal sorting algorithms tend to have high constant factors. The constant factors for radix sort, on the other hand, are relatively low. Radix sort has the further advantage that it is considerably simpler to implement than any of the optimal sorting algorithms.

Summary of the bucketized radix sort algorithm for the PDM

Our out-of-core, bucketized radix sort algorithm bears some similarity to the usual in-core radix sort method (see [CLR90, pp. 178–179] for a related algorithm). We make a number of passes over the data, where each pass examines K sort-key bits. We shall show in a moment how to choose the value of K. The first pass examines the least significant K bits, the second pass examines the next group of K bits, and so on. If there are L sort-key bits altogether, then $\lceil L/K \rceil$ passes are required. The jth pass performs a stable sort based on the jth group of K bits. In particular, we place each record into one of 2^K *buckets* based on the value of the K bits. Once all records have been placed into buckets, we concatenate the buckets in order of the K-bit values $0, 1, \ldots, 2^K - 1$ to produce a sequence of items sorted according to the least significant jK key bits. The correctness of this method relies on stability: records placed into the same bucket in some order in one pass must be processed in the same order in the next pass. The time per pass is $O(N + 2^K)$ for N records, and so the total time is $O(\lceil L/K \rceil (N + 2^K))$. We choose K to minimize this time.

Like the BMMC algorithm, out-of-core radix sort moves the data from a source file to a target file. It makes a number of passes, each of which rearranges the data. We swap the roles of the source and target files from pass to pass.

For out-of-core radix sort, it is important to minimize the number of passes, since each pass requires each record to be read and written once. Therefore, we wish to maximize K. We carve memory into $2^K = M/4BD$ buckets, so that $K = \lg(M/4BD)$.

Bucketized radix sort has two phases: census and distribution. In the *census phase*, we calculate how many records will fall in each bucket on each pass. We do so by reading the data into memory, one stripe at a time, and surveying each record to see which bucket it will fall into on each pass. The census information is later used to see how many stripes worth of data will be written out for each bucket.

The *distribution phase* performs the $\lceil L/K \rceil$ passes over the data. We read one stripe at a time into an input buffer and copy each record from the input buffer to its appropriate bucket. If the addition of a record to a bucket completes a stripe within that bucket, the stripe is then written out. The census information tells us where in the target file to write the stripe. At the end of each pass of the distribution phase, we write out any stripes that have not already been written.

Each pass performs $\lceil N/BD \rceil$ striped reads and $\lceil N/BD \rceil$ striped writes. The distribution phase consists of $\lceil L/\lg(M/4BD) \rceil$ passes, and the census phase requires $\lceil N/BD \rceil$ striped reads. The total number of parallel I/Os is thus $\lceil \frac{N}{BD} \rceil \left(2 \left\lceil \frac{L}{\lg(M/4BD)} \right\rceil + 1\right)$.

Implementation notes

From the above description, it would seem that we should be able to use as many as M/BD buckets, rather than only $M/4BD$. Like the BMMC algorithm, however, the use of asynchronous I/O changes the effective memory size for our radix sort algorithm. When we use asynchronous I/O, we allocate buffers for three stripes worth of data per bucket. The first two stripe buffers are a double buffer for writing behind.

The third stripe buffer of each bucket is only written once in each pass. Its purpose bears more explanation. We have no guarantee that any bucket starts or ends at a stripe boundary. In fact, most buckets start and end in the middle of stripes. When a bucket ends in the middle of a stripe, it shares that stripe with the beginning of the next nonempty bucket. The census information tells us where these bucket boundaries are. The third stripe buffer contains the last partial stripe of data from that bucket and the first partial stripe of data from the next nonempty bucket. By careful bookkeeping, we limit the number of partially filled stripes that are written to one per pass rather than one per bucket per pass.

To maintain that the number of buckets is a power of 2, we use $M/4BD$ rather than $M/3BD$ as the above description would suggest. Our out-of-core radix sort implementation with synchronous I/O also uses $M/4BD$ buckets; unlike our BMMC code, it is written without sensitivity to whether I/O is synchronous or asynchronous when making decisions based on memory size.

The census pass yields another useful optimization. If a pass places all records into the same bucket, then we skip that pass. In the common situation in which all N keys have equal high-order bits, this optimization can save several passes. The experiments we report in Section 6 use keys in which all bits are randomly chosen, and so this optimization does not affect the timings in this paper.

6 Timing results

This section presents timing results for the radix sort and BMMC permutation algorithms. We ran the BMMC algorithm on two platforms: adams with the uniprocessor ViC* wrappers and Fleet with the Galley implementation of the ViC* API. The in-core portion of out-of-core radix sort does not parallelize well, and so we ran radix sort only on adams. For radix sort, all runs were on 4-byte records, and for BMMC permutations, all runs were on 8-byte records. All characteristic matrices and complement vectors were randomly generated by repeated calls to the Unix `random()` function until the matrix so produced was nonsingular. For radix sort, keys are 4-byte integers generated by `random()`, and each record consists only of its key. All code was written in C and compiled with gcc. On adams, we used optimization level -O2, and on Fleet we used optimization level -O1.

We varied the PDM parameters differently on adams and Fleet. On adams, we varied the problem sizes from 2^{21} up to 2^{29} records. Memory sizes varied from 2^{24} bytes to 2^{27} bytes. Most runs on adams were with all 8 disks, but we also measured the effect of using 1, 2, and 4 disks. Physical block sizes varied from 2^8 bytes to 2^{14} bytes.

On Fleet, we could choose anywhere from 1 to 8 *compute processors* (*CPs*) and from 1 to 8 *I/O processors* (*IOPs*). Each IOP serves a disk (so that D is the number of IOPs), and it may be coresident with a CP. In order to allow each disk its maximum I/O bandwidth, we always kept the number of CPs and IOPs equal, i.e., $P = D$. The system runs fastest when CPs and IOPs reside on distinct processors, however. With 8 nodes, we performed most runs with 4 nodes as CPs and the other 4 nodes as IOPs. We did perform runs with 1, 2, and 8 IOPs as well. (With 8 IOPs, each IOP must be coresident with a CP.) The maximum number of records and maximum memory size for the entire system vary with the number of CPs. Our runs varied the problem sizes from 2^{19} to 2^{24} records per CP, and memory sizes varied from 2^{22} bytes to 2^{25} bytes per CP. As in our adams runs, physical block sizes varied from 2^8 bytes to 2^{14} bytes.

All I/O timings start upon calling ViC* synchronous read or write functions and end when they return. I/O timings, therefore, include software overhead from the ViC* wrappers, file system, and operating system. They also include any beneficial file-cache effects. Because even synchronous write calls return before actually writing the data to disk, write times are lower than read times once the problem size becomes large enough to negate file caching for reads. Nevertheless, the times we report here are what the programs observe.

It was impractical for us to produce timings for all combinations of PDM parameters in the ranges that make sense for adams and Fleet. On adams, for example, with 9 different values of N, 4 values of M, 4 values of D, and runs for synchronous vs. asynchronous I/O, we would have to run a program 288 times to try each combination once. Multiply that by the range of block sizes we might consider, and realize that some of these runs take over an hour, and it becomes apparent why we must choose our timing runs judiciously. Consequently, we varied block sizes by holding all other parameters except for the problem size fixed, and we did the same when we varied the number of disks.

BMMC permutations

Figure 6 shows running times for the BMMC permutation algorithm on adams. One plot shows problem sizes of 2^{21} to 2^{25} records, or 2^{24} to 2^{28} bytes. The other plot shows problem sizes of 2^{26} to 2^{29} records, or 2^{29} to 2^{32} bytes. We have separated these plots in order to maintain resolution in the faster runs for smaller problem sizes. All runs in Figure 6 use a physical block size of $B = 2^{10}$ and $D = 8$ disks.

For each problem size, there is a pair of bars. For now, we focus on the left bars of each pair, which breaks down the running time with synchronous I/O. The total height gives the total running time with synchronous I/O and a memory size of 2^{24} bytes. The left bars are comprised of four stacked rectangles. From bottom to top, the heights of these rectangles give the time spent reading memoryloads, the time spent writing memoryloads, the time spent permuting in memory, and all other time spent. The sum of the read and write times is, with a few exceptions, just over half of the total time. I/O and computation are approximately balanced in the BMMC algorithm.

Figure 6 also shows the effect and limits of file caching in the underlying Unix file system on adams. (Recall that the ViC* wrappers make calls to UFS.) For problem sizes up to 2^{23} records, read times are very small. They begin to jump at 2^{24} records, and they are relatively high from 2^{25} records and up. Why? The physical memory size of adams is 320 megabytes, and a problem size of 2^{25} records is 256 megabytes in each of the source and target files. At this problem size, for most records, between the time that the record is written and the time it is next read, more bytes than the size of the physical memory have passed through the file cache, and so the record is not present in the file cache.

Write times are lower than read times starting at 2^{25} records. Again, this behavior is due to

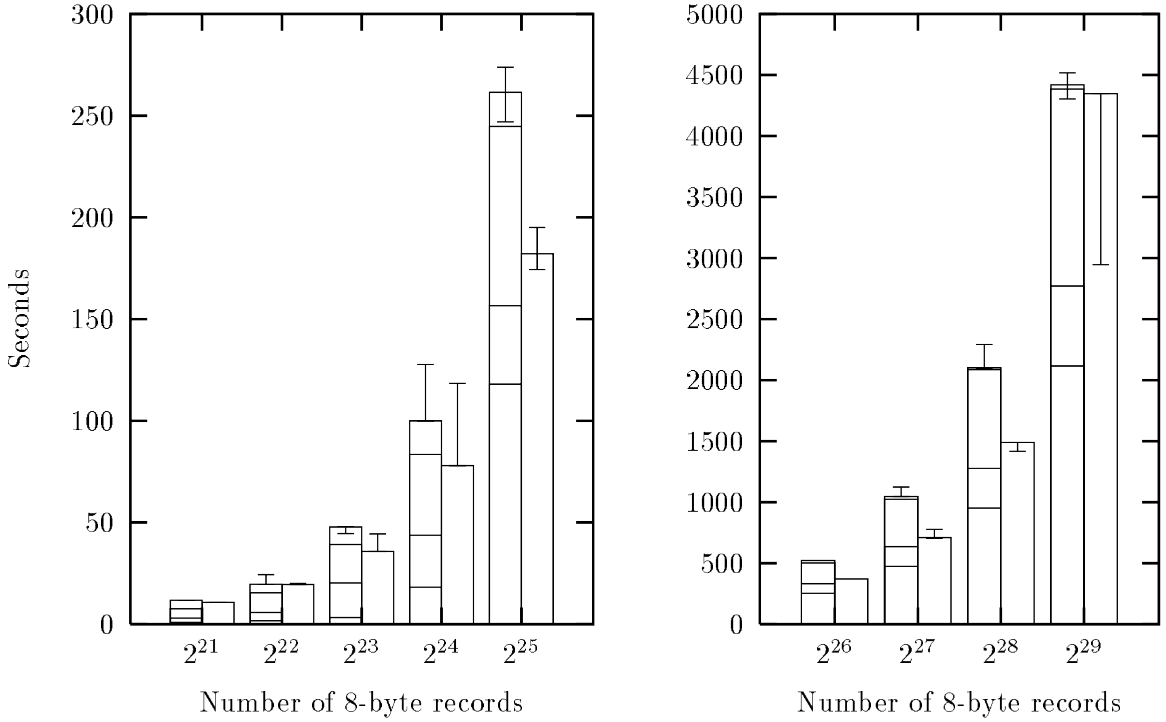


Figure 6: Breakdown of time spent on adams in performing BMMC permutations with synchronous I/O, and the total time with asynchronous I/O. The timings are shown in two parts in order to maintain resolution in the faster runs for smaller problem sizes. The horizontal axis is the problem size, as a number of 8-byte records, and the vertical axis is time in seconds. For each run shown here, the memory size is 2^{24} bytes, the physical block size is 2^{10} bytes, and 8 disks are used. For each problem size, the left side shows time using synchronous I/O, and the right side shows the total time using asynchronous I/O. The left sides are comprised of four rectangles stacked on top of each other; the height of all four together gives the total running time. The bottom rectangle represents the total time spent reading memoryloads, the next rectangle shows the time spent writing memoryloads, the next rectangle represents time spent permuting in memory, and the top rectangle (usually quite small) encompasses all other time spent. "Error bars" represent variations observed in the total time for memory sizes ranging from 2^{24} through 2^{27} bytes (but not exceeding the problem size in bytes).

file caching. UFS write calls do not guarantee that the data has actually gone out to disk by the time they return.

The total time follows the prediction of the PDM quite well. Each run with synchronous I/O in Figure 6 uses 2 passes. With the number of passes held fixed, the PDM predicts that the total time is linear in the number of records. Except for the jump at 2^{25} records from file caching no longer yielding a benefit, we see that the total time approximately doubles each time the number of records doubles.

Figure 7 shows analogous timings on Fleet with 4 CPs and 4 IOPs, so that $P = D = 4$. The memory size is 2^{24} bytes per CP, or 2^{26} bytes altogether. Physical block sizes are 2^{12} bytes. From bottom to top, the stacked rectangles represent read time, write time, communication time outside the I/O calls, compute time, and all other time. The sum of read and write times is about half of the total time on Fleet, so that I/O and computation/communication are balanced. There appears

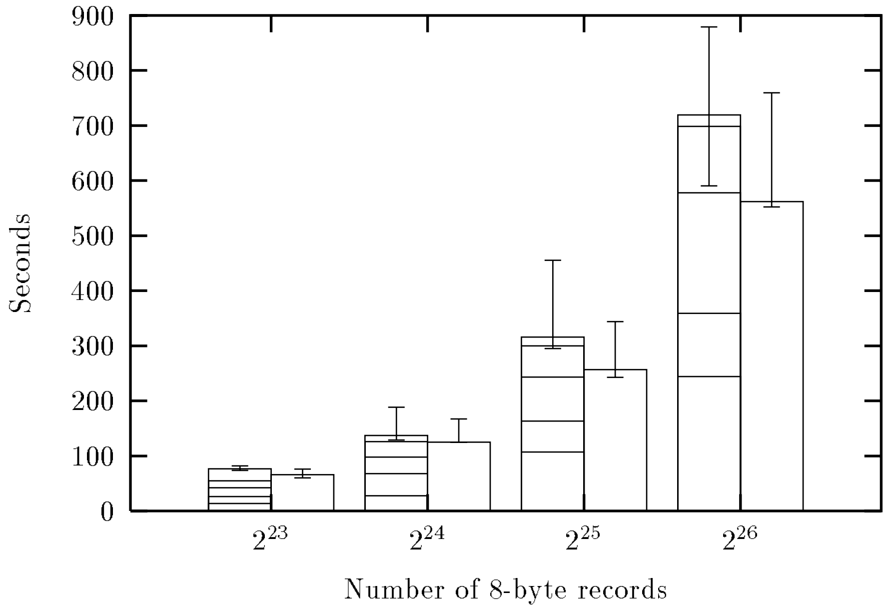


Figure 7: The analogue of Figure 6 for Fleet. For each run shown here, the physical block size is 2^{12} bytes and there are 4 CPs, 4 IOPs, and 2^{24} bytes of memory per CP, or 2^{26} bytes of memory altogether. The left sides are comprised of five rectangles stacked on top of each other; from bottom to top, they represent read time, write time, communication time outside the I/O calls, compute time, and all other time. "Error bars" represent variations observed in the total time for memory sizes ranging from 2^{22} through 2^{25} bytes per CP, or 2^{24} to 2^{27} bytes in total.

to be some file caching, as evidenced by the jump in read times from 2^{24} records to 2^{25} records. As was the case with adams, each run took 2 passes and the total time increases approximately linearly with the number of records, as the PDM predicts.

The right bar of each pair in Figures 6 and 7 shows the total time for the BMMC permutation with asynchronous I/O. Figures 8 and 9 show the percentage of total time saved by using asynchronous I/O at various problem and memory sizes. For large enough problem sizes, the asynchronous time is quite a bit lower than the synchronous time. An apparent exception is the case for 2^{29} records on adams, in which the asynchronous time for $M = 2^{24}$ bytes is only marginally lower than the synchronous time. This phenomenon is an outcome of our discussion in Section 4 on the effective memory size. Recall that the number of passes is at most $\left\lceil \frac{\operatorname{rank} \gamma}{\lg(M/B)} \right\rceil + 2$, where γ is the lower left $\lg(N/B) \times \lg B$ submatrix of the characteristic matrix. For $N = 2^{29}$ and $B = 2^{10}$, the submatrix γ has rank $\lg \min(N/B, B) = 10$. When the memory size is 2^{24} bytes, we use $M = 2^{20}$ with synchronous I/O; we lose a factor of 8 because of the conversion from bytes to records, and we lose another factor of 2 because we need to allocate both source and target buffers in memory. In this case, $\left\lceil \frac{\operatorname{rank} \gamma}{\lg(M/B)} \right\rceil = \lceil 10/10 \rceil = 1$. With asynchronous I/O, however, we lose another factor of 2 from the effective memory size because we must allocate buffers to hold the memoryloads that we are reading ahead and writing behind. Hence, we reduce the effective memory size to $M = 2^{19}$. Now $\left\lceil \frac{\operatorname{rank} \gamma}{\lg(M/B)} \right\rceil = \lceil 10/9 \rceil = 2$. With this particular combination of PDM parameters, therefore, asynchronous I/O incurs the expense of an additional pass. It is testimony to the benefit of over-

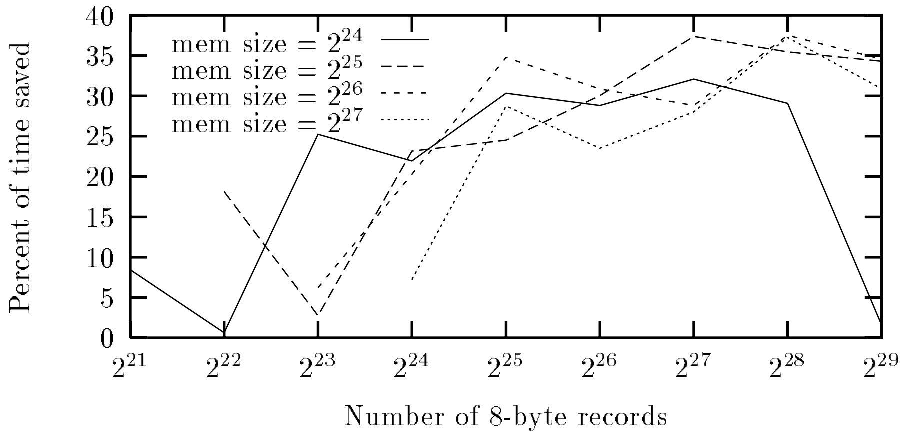


Figure 8: Percentage of total time for BMMC permutations saved with asynchronous I/O on adams at various problem and memory sizes with 8 disks and a physical block size of 2^{10} bytes.

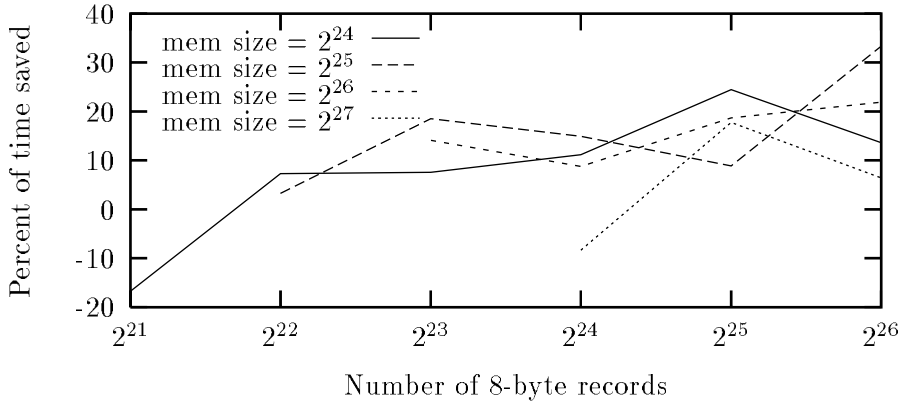


Figure 9: The analogue of Figure 8 for Fleet with 4 CPs, 4 IOPs, and a physical block size of 2^{12} bytes.

lapping I/O and computation that the algorithm manages to run faster with asynchronous I/O in this case.

Because the BMMC algorithm spends much of its time performing I/O, the benefits of asynchronous I/O are limited, especially at large problem sizes. Figures 10 and 11 show the percentage of time spent doing I/O and the percentage of time saved by using asynchronous I/O, at a memory size of 2^{25} bytes on adams (so that the number of passes would not change between synchronous and asynchronous I/O) and 2^{26} bytes on Fleet. If all I/O latency was hidden by asynchronous I/O, then the two curves would coincide. When I/O time dominates, asynchronous I/O cannot hide all of it. Because I/O implies communication in Fleet (CPs and IOPs communicate via the network), we cannot fully overlap I/O with communication that occurs outside the I/O. Figure 11 also plots the percentage of time spent in computation. Observe that the percentage of time saved by using asynchronous I/O is close to the computation percentage, which implies that we are able to overlap

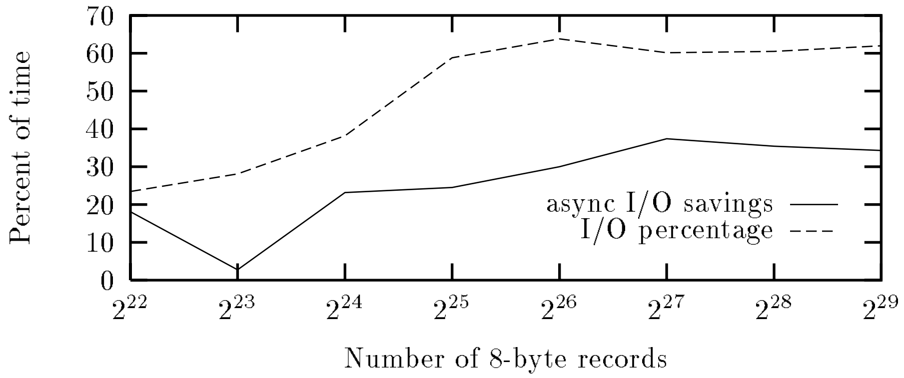


Figure 10: Percentage of time spent in the BMMC algorithm doing I/O and percentage of time saved by using asynchronous I/O on adams at various problem sizes. The memory size was 2^{25} bytes, the physical block size was 2^{10} bytes, and there were 8 disks.

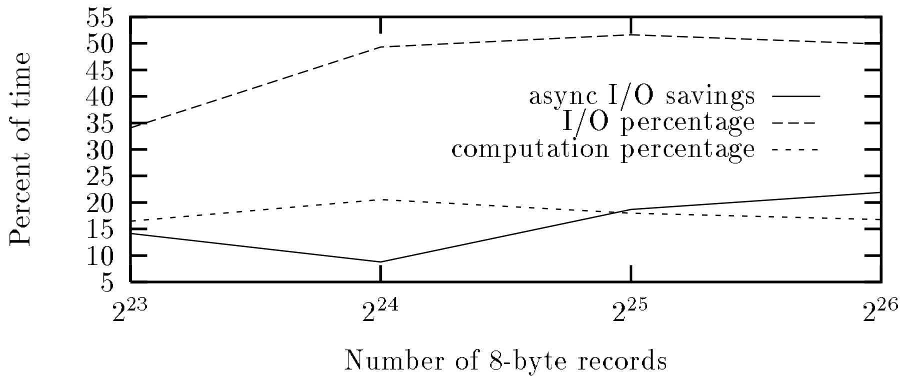


Figure 11: The analogue of Figure 10 for Fleet, with a 4 CPs, 4 IOPs, a physical block size of 2^{12} bytes, and a memory size of 2^{26} bytes. Percentage of time spent in computation is also included here.

as much I/O with pure computation as possible.

Figures 6 and 7 show "error bars" for the synchronous and asynchronous total times, representing variations among memory sizes ranging from 2^{24} bytes to 2^{27} bytes on both adams and Fleet. Except for cases in which asynchronous I/O induced an additional pass, variations due to differing memory sizes on adams were small. We observe higher variations on Fleet, even though all runs shown in Figure 7 use exactly 2 passes. Although none of our figures show why, these variations are due mainly to differences in communication time (outside the I/O calls). Changing the memory size produces different interprocessor communication patterns with varying performance.

Like the synchronous running times, asynchronous running times are approximately linear in the number of records. Exceptions occur at problem sizes for which file caching stops yielding a benefit and for which asynchronous I/O incurs an additional pass. Note, however, that the bottom error bar for asynchronous I/O on adams with 2^{29} records, which occurs with a memory size of 2^{26}

bytes, is about double the asynchronous time for 2^{28} records. In this case, the algorithm avoids the additional pass once the memory size reaches 2^{27} bytes.

Next we consider the physical block size. According to the PDM, increasing the block size makes each pass use fewer parallel I/O operations, since it takes N/BD of them to read or write each record once. This predicted effect is balanced by the block size's effect on the number of passes, however. Recalling that the number of passes has a denominator of $\lg(M/B)$, we see that if the block size gets large enough, it will induce more passes.

Figures 12 and 13 show how changing the physical block size changes the running time. Although relatively small block sizes—2^9 bytes or smaller—may produce fewer passes, the time per pass increases markedly. The PDM predicts this behavior, although the effect of the increased I/O time is ameliorated by the computing time. Once the physical block size reaches 2^{10} bytes, however, the time per pass does not change significantly. On adams, therefore, 2^{10} bytes, corresponding to a logical block size of 2^{13} bytes with 8-byte records, appears to be the block size of choice. This size—2^{13} bytes—happens to be the system page size. On Fleet, the best physical block size is 2^{12} bytes, corresponding to a logical block size of 2^{15} bytes, which is in fact the Galley block size.

It appears that the PDM is fairly accurate in predicting the effect of the block size up to a fundamental block size for the system, but it is inaccurate beyond that.

The PDM predictions about the number of disks are far less accurate on adams. Figure 14 shows total read and write times for 1, 2, 4, and 8 disks. Problem sizes range from 2^{21} records to 2^{26} records, which is the largest problem size that fits on 1 disk. The PDM predicts that for a given problem size, I/O times would be inversely proportional to the number of disks. As Figure 14 shows, this is not the case on adams. Figure 16 shows read and write bandwidths. The PDM predicts that for a given problem size, I/O bandwidth would increase linearly with the number of disks. Again, the figure shows this not be the case on adams.

On Fleet, the PDM predictions about the number of disks are somewhat better. Figure 15 shows total read and write times for 1, 2, 4, and 8 CPs and IOPs. Problem sizes range from 2^{22} to 2^{24} records per CP. Because problem sizes are expressed per CP, perfect I/O scalability, as predicted by the PDM, would mean equal I/O times for the various numbers of IOPs. The times for 1, 2, and 4 IOPs are fairly close (except for 1 IOP with 2^{23} records), but read times for 8 IOPs are significantly higher. Because CPs and IOPs are coresident at this size, they contend with each other for local resources, the CPU in particular. Figure 17 shows read and write bandwidths for Fleet. They increase linearly with the number of IOPs, except for the read bandwidth at 8 IOPs, which is about the same as the read bandwidth for 4 IOPs.

Radix sort

We found that radix sort's behavior on adams fit the PDM somewhat better than the BMMC algorithm did.

Figure 18 shows the breakdown of running time into read time, write time, compute time, and other time for radix sort with synchronous I/O, and it also shows the total time with asynchronous I/O. Here, the memory size is 2^{27} bytes; with 4-byte records, it is not until we reach 2^{26} records that the problem size exceeds the memory size. All measured times increase close to linearly with the problem size. The only real difference is a jump in the read time between 2^{24} and 2^{25} records. Note that this size, in bytes, is the same point at which file caching began to yield no benefit in the BMMC algorithm.

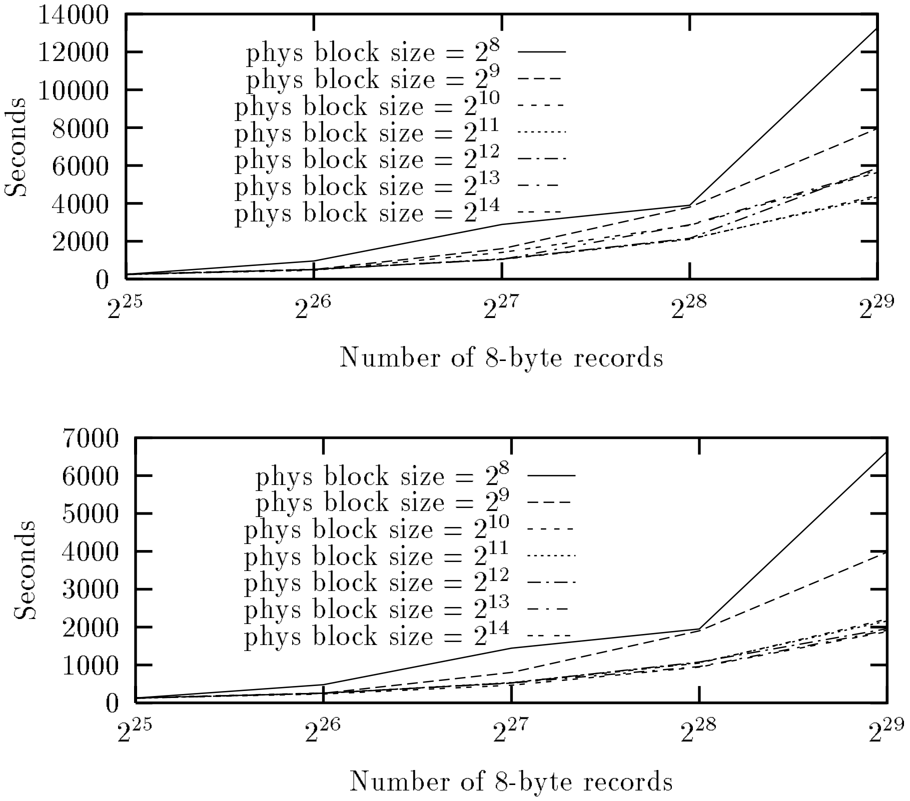


Figure 12: Total time (top) and time per pass (bottom) for BMMC permutations on adams with varying numbers of records and physical block sizes and synchronous I/O. All runs shown are for 8 disks and a memory size of 2^{24} bytes.

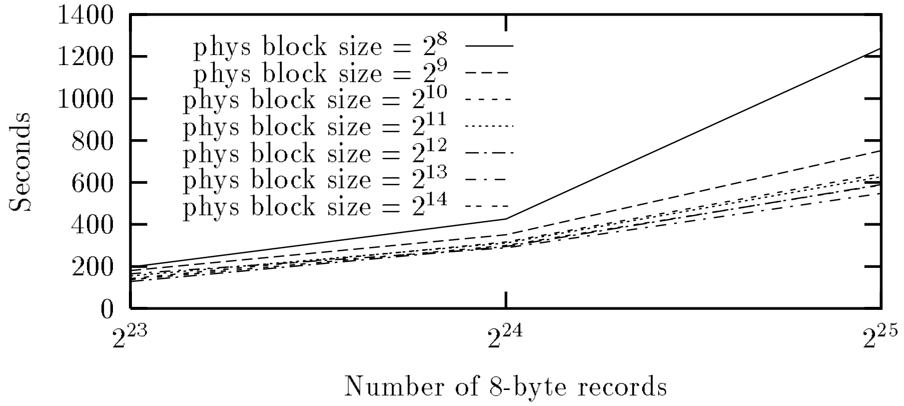


Figure 13: Total time on Fleet for BMMC permutations with varying numbers of records and physical block sizes and synchronous I/O. All runs shown are for 4 CPs, 4 IOPs, and a total memory size of 2^{27} bytes. Each run shown here takes 2 passes, so that there is no need to plot time per pass.

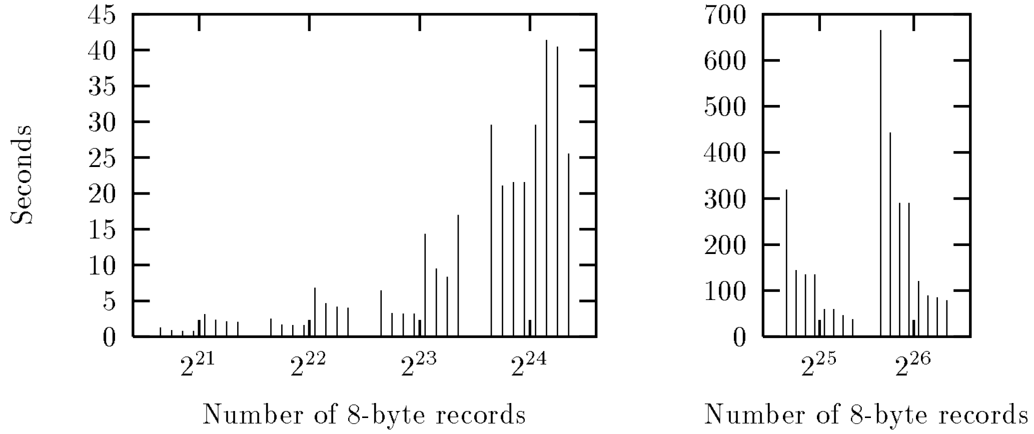


Figure 14: Read and write times on adams for BMMC permutations on 8-byte records with 1, 2, 4, and 8 disks. Read times for 1, 2, 4, and 8 disks, respectively, are on the left side of each set, and write times for the same ordering are on the right side of each set. Problem sizes vary as shown, the memory size is fixed at 2^{24} bytes, and the physical block size is 2^{10} bytes.

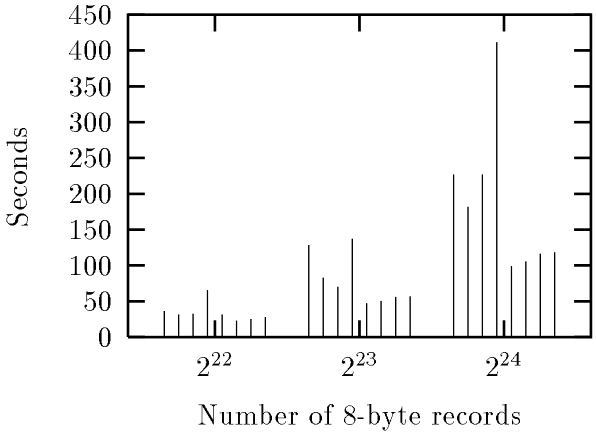


Figure 15: The analogue of Figure 14 for Fleet. In each case, the number of CPs and IOPs is kept equal, and they vary among 1, 2, 4, and 8. Problem sizes vary as shown but are per CP, the total memory size is 2^{25} bytes per CP, and the physical block size is 2^{12} bytes. Unlike Figure 14, here scalability is represented by equal times for a given problem size per CP.

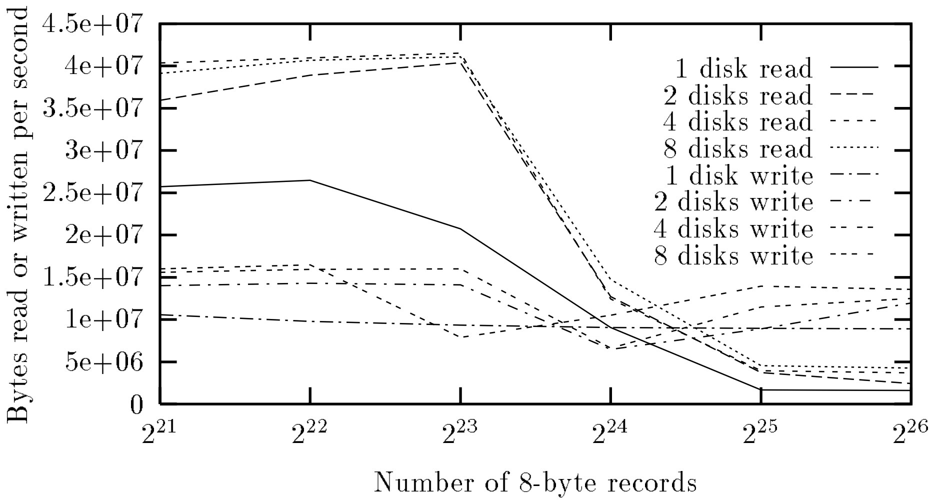


Figure 16: Read and write bandwidths on adams for BMMC permutations with varying numbers of disks and problem sizes. The memory size for each run is 2^{24} bytes, and the physical block size is 2^{10} bytes.

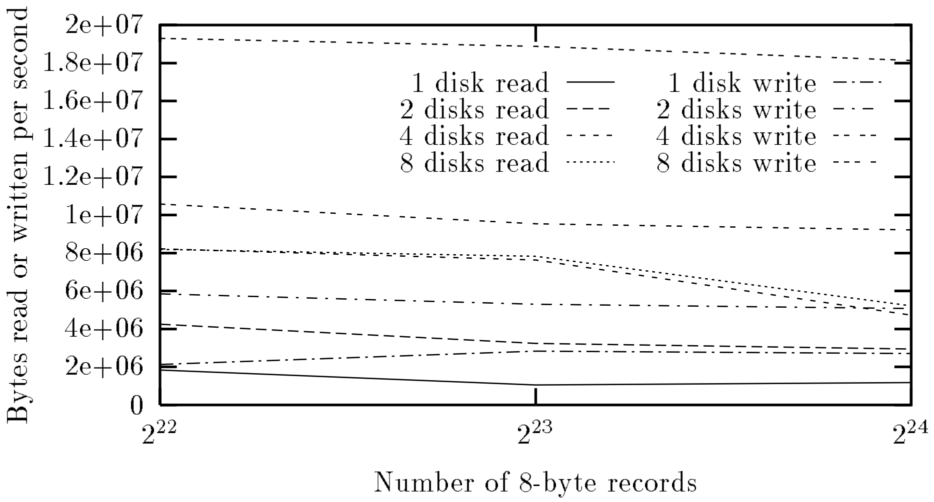


Figure 17: The analogue of Figure 16 for Fleet. The memory size for each run is 2^{25} bytes per CP, and the physical block size is 2^{12} bytes.

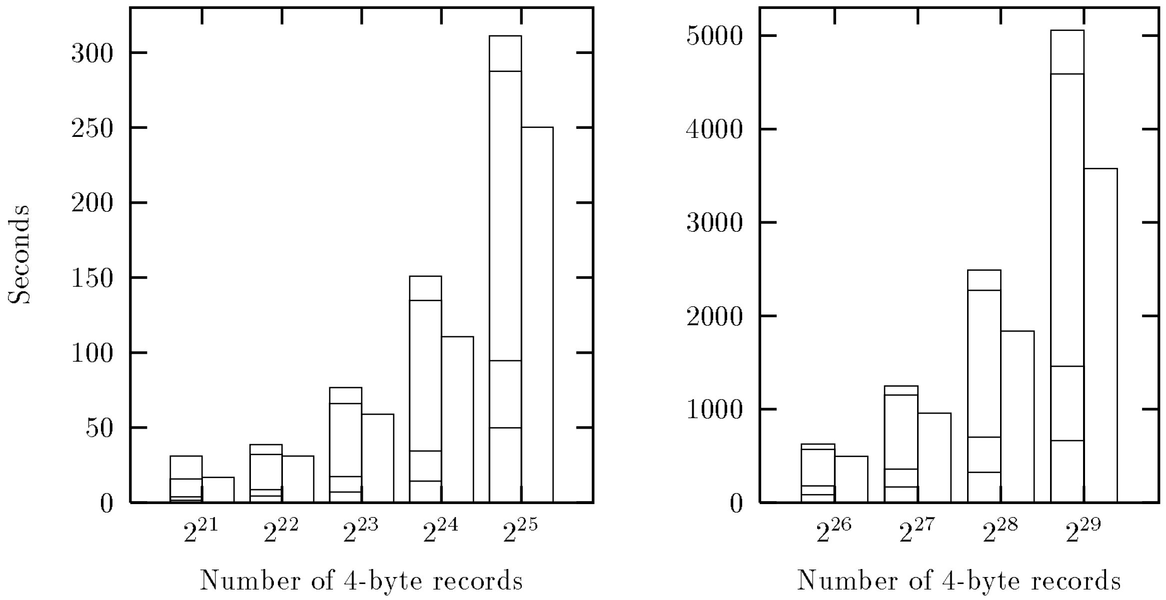


Figure 18: Breakdown of time spent on adams in performing out-of-core radix sort with synchronous I/O, and the total time with asynchronous I/O. This plot is organized in the same manner as Figure 6 but without showing variations among memory sizes. There are 8 disks, the memory size is fixed at 2^{27} bytes, and the physical block size is 2^{11} bytes, which corresponds to a logical block size of 2^{13} bytes.

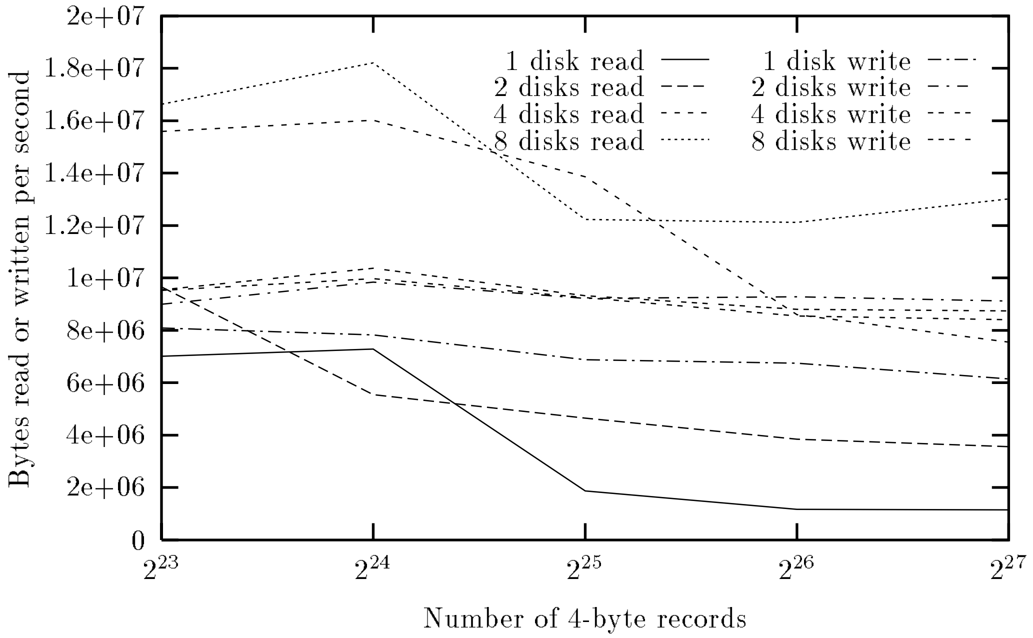


Figure 19: Read and write bandwidths on adams for radix sort with varying numbers of disks and problem sizes. The memory size for each run is 2^{27} bytes, and the physical block size is 2^{11} bytes.

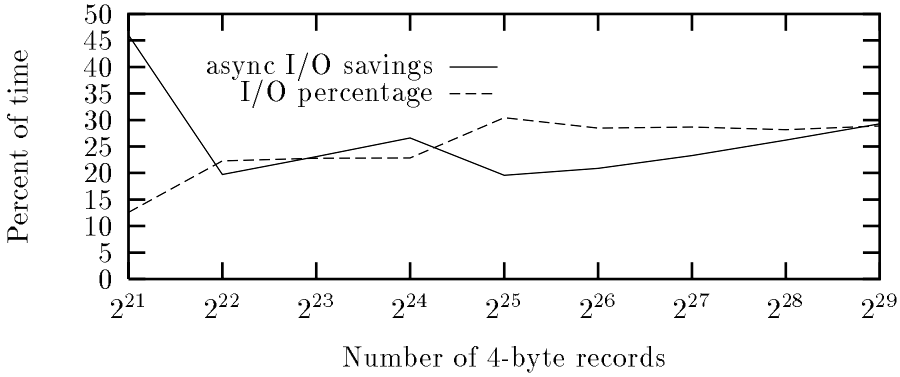


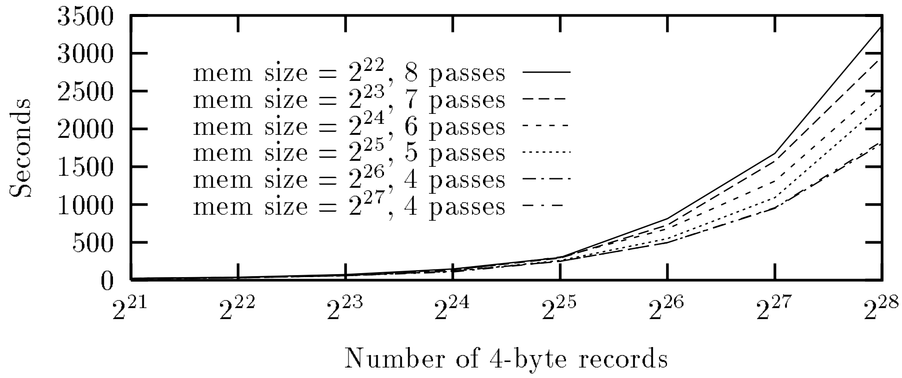
Figure 20: Percentage of time spent in radix sort doing I/O and percentage of time saved by using asynchronous I/O on adams at various problem sizes. The memory size is 2^{27} bytes, the physical block size is 2^{11} bytes, and there are 8 disks.

Because we used a memory size of 2^{27} bytes in our radix sort runs, file caching has a lesser effect than we saw in the BMMC algorithm. Figure 19 shows read and write bandwidths for varying numbers of disks at this memory size. File caching for reads ends when the problem size reaches the memory size of 2^{27} bytes. Figure 19 also shows some scalability in read bandwidth as the number of disks increases, but no scalability in write bandwidth.

Radix sort makes effective use of asynchronous I/O because it is not as I/O bound as the BMMC algorithm. Figure 20 shows the percentage of time spent doing I/O and the percentage of time saved by using asynchronous I/O at the 2^{27}-byte memory size. The percentages are quite close, indicating that I/O costs are successfully hidden by the asynchronous version.

Memory size affects radix sort's running time pretty much as predicted by the PDM. Figure 21 shows total time and time per pass with differing memory sizes. The time per pass does not change much with memory size. The number of passes depends heavily on the memory size, as is apparent from the out-of-core radix sort description in Section 5. Simply put, the more memory, the better, as the PDM predicts.

Finally, we consider the physical block size. As with the BMMC algorithm, the PDM predicts that a larger block size yields a tradeoff between faster passes but more of them. Figure 22 shows the reality on adams, which is overall the same as for BMMC permutations: use the physical block size for which the logical block size equals the page size. Note the sudden jump in time per pass and total time for radix sort at a physical block size of 2^9 bytes. We believe that this jump is analogous to the jump in Figure 12 with a physical block size of 2^8 bytes. Because the record size we used for radix sort is half that of the BMMC algorithm, the logical block sizes are then the same. The jump is more pronounced for radix sort because all its accesses are for a stripe at a time but some of the BMMC code's accesses are for memoryloads of consecutive logical stripes. These are combined by the ViC* wrappers into larger accesses. The access size, therefore, is large enough in these cases that the block size does not matter.



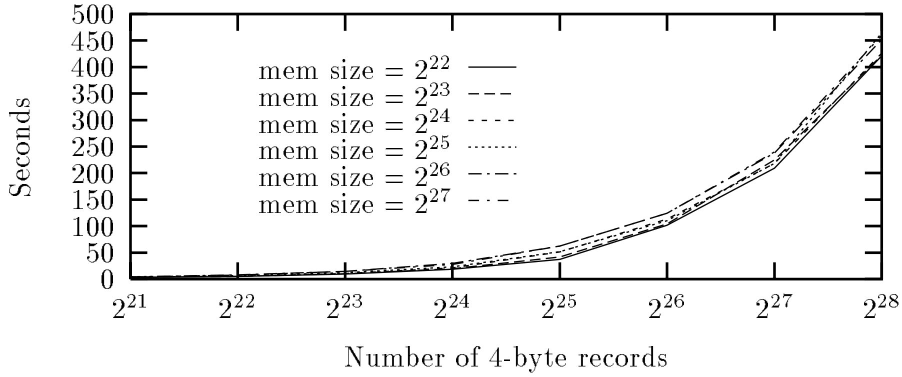


Figure 21: Total time (top) and time per pass (bottom) for radix sort on adams with varying numbers of records and memory sizes and asynchronous I/O. All runs shown are for 8 disks and a physical block size of 2^{11} bytes.

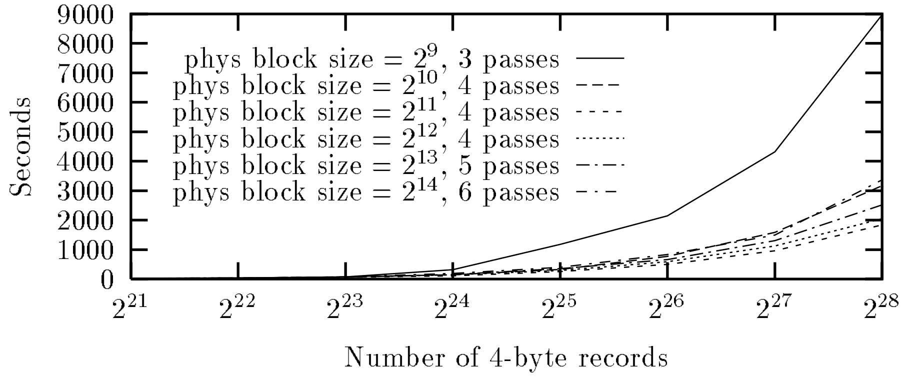
7 Conclusion

The primary goal of the ViC* project is to provide an system for out-of-core data-parallel computing in a transparent, efficient, and portable manner. That it enables us to evaluate the PDM is a side benefit.

As side benefits go, however, it is a valuable one. The experiences we have reported in this paper have led us to new ways of looking at the PDM. The most surprising result to us is that the algorithms we implemented were not as heavily I/O bound as we had expected. With 8 disks and the best block size on a fast uniprocessor, the BMMC permutation algorithm with synchronous I/O spent no more than 64% of its time waiting for I/O. Considering how much higher disk-access times are than memory-access times, our intuition was that we would have seen figures upward of 80%. The out-of-core radix sort algorithm, which performs a more complex computation, is even less I/O bound. With 8 disks and the best block size on a fast uniprocessor, it spent no more than 31% of its time waiting for I/O.

We view our results as containing both good news and bad news about the PDM. First, the good news:

- Well designed algorithms for the PDM are likely to not be I/O bound. The purpose of the



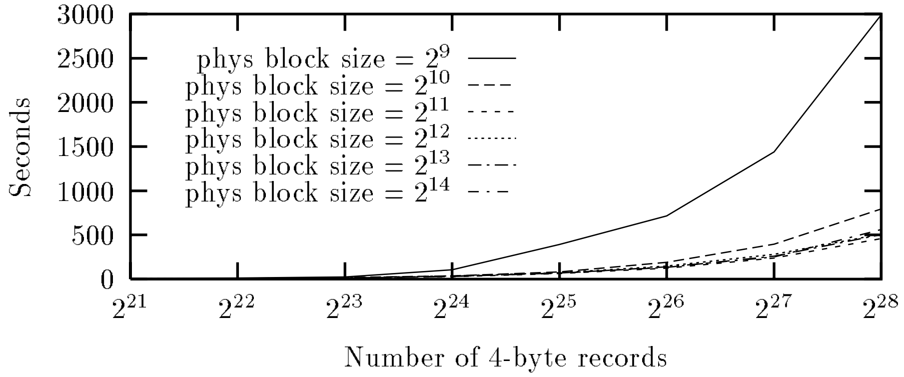


Figure 22: Total time (top) and time per pass (bottom) for radix sort on adams with varying numbers of records and physical block sizes and asynchronous I/O. All runs shown are for 8 disks and a memory size of 2^{27} bytes.

PDM is to alleviate the I/O bottleneck at the level of algorithm design and implementation, and it appears likely to achieve this goal.

- PDM algorithms that are not I/O bound have the potential to have their I/O wait times reduced by using asynchronous I/O. There are two disclaimers to this benefit. First, setting aside additional buffers for asynchronous I/O reduces the effective memory size at the disposal of a PDM algorithm, which may in turn lead to the algorithm requiring extra passes over the data. Second, in order for an algorithm to hide read latencies, the identities of the blocks to be read must be known well enough in advance. In particular, if an algorithm cannot determine which blocks to read next until it has processed a memoryload, then it will not hide read latencies well.

- I/O times seem to follow the PDM predictions based on the parameters for problem size and memory size.

- Total run times seem to follow the PDM predictions based on the parameter for problem size.

- As the ViC* API demonstrates, we can implement a PDM interface in a portable and efficient fashion.

The bad news is that the PDM's predictions do not follow its other two parameters, block size and number of disks, all that well. Too small a block size can increase run times more than the PDM predicts, and too large a block size fails to yield improved times per pass. The block size is a fundamental parameter of the underlying system, and the algorithm designer has little leeway in picking its value.

As we found on the uniprocessor, adding disks does not necessarily increase I/O bandwidth. A moment's reflection reveals why. To travel between the disks and the memory, bits go over a set of wires. These wires may be a network, they may be SCSI cables, or they may just be ports into the memory. Whatever they are, they can carry only so many bits per second. Because the PDM does not include a notion of limited network bandwidth, it considers each additional disk as delivering more I/O bandwidth. Once enough disks are added, they provide only the benefit of more capacity and not more performance. Unlike memory, adding more disks does not necessary help. We observed this behavior even on Fleet once IOPs became coresident with CPs. One should be aware of the I/O system's bandwidth limits before adding more disks.

We also found that the memory size can affect communication time, and hence overall running time, in a way that the PDM cannot account for. Of course, it is not really reasonable to expect the PDM to account for changes in communication time since it is designed to model only I/O times.

Future work

These results suggest two research directions.

First, can we develop a model that more accurately models I/O, computation, and communication, yet does not overwhelm the algorithm designer with parameters? Perhaps a hybrid of the PDM and Bulk Synchronous Processing (BSP) models [Val90] would be suitable. Or perhaps any accurate model would be too complex to design algorithms on.

Second, although we have measured two algorithms developed for the PDM, we have not determined whether they run any faster than what would be the simplest out-of-core implementation of all: running the standard in-core algorithm in the presence of traditional demand-paged virtual memory. A BMMC algorithm would then be quite easy to code up, and sorting would be even easier—just use the Unix `qsort()` function. Indeed, we attempted to run Unix `qsort()` on 512 megabytes of data on adams. The program crashed due to a lack of swap space. One of our future experiments will be to reconfigure the disks on adams to use the 8 data disks as a RAID for virtual-memory swap space and run simple in-core algorithms. We expect these runs to be far slower than our PDM algorithms. After all, in-core algorithms are not designed to make efficient use of multiple disks or of disk blocks. Moreover, RAIDs improve bandwidth but not latency. In-core algorithms on native virtual memory have one advantage, however: reduced file-system overhead. Based on out-of-core FFT experiments in [CN96], we doubt that this advantage is enough to overcome the efficiency of PDM algorithms.

Acknowledgments

We thank Garth Gibson and Jim Zelenka for their help in implementing ViC* on top of SPFS; they made many changes to SPFS at our request. Erik Riedel and Adam Beguelin helped us get started with PVM, and Nils Nieuwejaar got us started with MPI. Hugo Patterson provided valuable ideas on data presentation. Nils Nieuwejaar and David Kotz graciously responded to our numerous questions about the Galley File System. Len Wisniewski provided insights that led to the two optimizations for the BMMC algorithm described in the implementation notes. Len, Alex Colvin, and Anna Poplawski provided comments throughout the ViC* API design. Discussions with Mike Goodrich led to the question about whether we can design a more accurate but still useful model for I/O, computation, and communication. David Kotz and anonymous referees provided many valuable comments on our presentation. The purchase of adams was made possible in part by an equipment grant from Digital Equipment Corporation.

References

[Arg95] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *4th International Workshop on Algorithms and Data Structures (WADS)*, pages 334–345, August 1995.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[AVV95] Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory algorithms for processing line segments in geographic information systems. In Paul Spirakis, editor, *Proceedings of the Third Annual European Symposium on Algorithms (ESA '95)*, volume 979 of *Lecture Notes in Computer Science*, pages 295–310. Springer-Verlag, September 1995.

[BGV96] Rakesh D. Barve, Edward F. Grove, and Jeffrey Scott Vitter. Simple randomized mergesort for parallel disks. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 109–118, June 1996.

[CB95] Thomas H. Cormen and Kristin Bruhl. Don't be too clever: Routing BMMC permutations on the MasPar MP-2. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 288–297, July 1995.

[CGG+95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, January 1995.

[CGK+88] Peter Chen, Garth Gibson, Randy H. Katz, David A. Patterson, and Martin Schulze. Two papers on RAIDs. Technical Report UCB/CSD 88/479, Computer Science Division (EECS), University of California, Berkeley, December 1988.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[CN96] Thomas H. Cormen and David M. Nicol. Performing out-of-core FFTs on parallel disk systems. Technical Report PCS-TR96-294, Dartmouth College Department of Computer Science, August 1996.

[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. Preliminary version appeared in *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures.* Revised version to appear in *SIAM Journal on Computing.*

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weichang Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[Gib92] Garth A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage.* The MIT Press, Cambridge, Massachusetts, 1992. Also available as Technical Report UCB/CSD 91/613, Computer Science Division (EECS), University of California, Berkeley, May 1991.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, 1994.

[GSC+95] Garth A. Gibson, Daniel Stodolsky, Fay W. Chang, William V. Courtright II, Chris G. Demetriou, Eka Ginting, Mark Holland, Qingming Ma, LeAnn Neal, R. Hugo Patterson, Jiawen Su, Rachad Youssef, and Jim Zelenka. The Scotch parallel storage systems. In *Proceedings of the IEEE CompCon Conference*, pages 403–410, March 1995.

[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.

[MW95] Sean S. B. Moore and Leonard F. Wisniewski. Complexity analysis of two permutations used by fast cosine transforms. Technical Report PCS-TR95-266, Dartmouth College Department of Computer Science, October 1995.

[NK96a] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, May 1996.

[NK96b] Nils Nieuwejaar and David Kotz. Performance of the Galley parallel file system. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 83–94, May 1996.

[NV93] Mark H. Nodine and Jeffrey Scott Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, June 1993.

[NV95] Mark H. Nodine and Jeffrey Scott Vitter. Greed sort: Optimal deterministic sorting on parallel disks. *Journal of the ACM*, 42(4):919–933, July 1995.

[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.

[SOHL+96] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[SW95] Elizabeth A. M. Shriver and Leonard F. Wisniewski. An API for choreographing data accesses. Technical Report PCS-TR95-267, Dartmouth College Department of Computer Science, November 1995.

[TMC93] Thinking Machines Corporation. *C* Programming Guide*, May 1993.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

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

[Ven97] Darren Erik Vengroff. *The Theory and Practice of I/O-Efficient Computation*. PhD thesis, Brown University Department of Computer Science, 1997. To appear.

[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.

[Wis96] Leonard F. Wisniewski. Structured permuting in place on parallel disk systems. In *Proceedings of the Fourth Annual Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, pages 128–139, May 1996.