An API for Choreographing Data Accesses*

Elizabeth A. M. Shriver†
Courant Institute of Mathematical Sciences
New York University

Leonard F. Wisniewski‡
Department of Computer Science
Dartmouth College

November 8, 1995

Abstract

Current APIs for multiprocessor multi-disk file systems are not easy to use in developing out-of-core algorithms that choreograph parallel data accesses. Consequently, the efficiency of these algorithms is hard to achieve in practice. We address this deficiency by specifying an API that includes data-access primitives for data choreography. With our API, the programmer can easily access specific blocks from each disk in a single operation, thereby fully utilizing the parallelism of the underlying storage system.

Our API supports the development of libraries of commonly-used higher-level routines such as matrix-matrix addition, matrix-matrix multiplication, and BMMC (bit-matrix-multiply/complement) permutations. We illustrate our API in implementations of these three high-level routines to demonstrate how easy it is to use.

1 Introduction

Since disk and disk-controller speeds have not kept pace with the speeds of processors, the bottleneck for many applications that use large data sets has become the I/O subsystem. A number of approaches can increase I/O throughput, varying from hardware improvements to better algorithms.

*This work was performed in part at Sandia National Laboratories, operated for the U.S. Department of Energy under contract number DE-AC04-94AL85000. Also published as Dartmouth College Technical Report PCS-TR95-267.

†Supported in part by the National Science Foundation under Grant CCS-9204202. Author's e-mail address: shriver@cs.nyu.edu.

‡Supported in part by a Dartmouth Fellowship and in part by the National Science Foundation under Grant CCR-9308667. Author's e-mail address: wisnie@cs.dartmouth.edu.

The hardware solutions include methods to improve the rate of I/O throughput to uniprocessor systems by introducing parallelism into the I/O subsystem. Mechanisms such as disk striping or interleaving [Kim86, SGM86], and RAID [PGK88] have achieved fine-grain parallelism at the physical disk level. At the software level, considerable effort has been made to develop new languages and compiler features that support I/O parallelism and optimizations via data layout conversion [dBC93], compiler hints [PGS93], and preprocessing for out-of-core parallel code [CBH+94, CC94]. Another approach integrates special enhancements for I/O into the file system [CBF93, KS93].

The theory community has developed parameterized computational models, called *memory models* or *input/output (I/O) complexity models*, that aim to represent the key features of computer memory hierarchies and data movement in order to present a suitable model for algorithm design and analysis at a feasible level of abstraction.[1] The interested reader can find full descriptions of these models in [ACFS94, AV88, VS90, VS94].

Using the I/O complexity models, algorithm designers have developed out-of-core algorithms that choreograph data movements to utilize all of the disks in the I/O subsystem concurrently. We feel that this approach is a viable solution because of the asymptotic performance gains achieved by I/O-efficient algorithms. However, in order for this approach to be fully utilized, there must be operating-system support for the data-access primitives used by the algorithms. We suggest that the file systems of multiprocessor multi-disk machines include an application programmer interface (API) that supports the easy implementation and efficient execution of these algorithms. We describe one such API in this paper.

Unlike other APIs, we designed our API for the sole purpose of allowing the application programmer to fully utilize the bandwidth of the parallel storage system. File system developers typically concentrate on issues such as caching, write-behind, and data-layout and, therefore, do not provide routines to directly access specific blocks of out-of-core data on each disk. Our API allows the programmer to access large quantities of data in a single operation even if the desired data resides at different locations on each disk. We hope it will encourage the development of more I/O-efficient algorithms.

Many high-level programmers do not wish to explicitly program I/O. Thus, a portable library of commonly-used functions which require I/O-efficient implementations would abstract the burden of efficiently handling I/O away from the high-level programmer. Our API

[1] These models do not consider issues such as cache consistency, cache coherency, and write-backs from cache; behavior of these issues should be considered in future research on the models.

provides the library writer with routines for scalable disk access to efficiently implement the I/O algorithms.

In this paper, we specify the data-access routines of our API and show that it is easy to use by providing example code for some I/O-efficient algorithms. In particular, we demonstrate the easy use of our API in the implementation of out-of-core algorithms to perform matrix-matrix addition, matrix-matrix multiplication, and BMMC (bit-matrix-multiply/complement) permutations. Our API is easy to implement; it has been implemented as the interface for the Whiptail File System, an experimental file system for I/O-efficient out-of-core problems [SWC+95].

The outline of this paper is as follows. In Section 2, we describe the Parallel Disk Model and the types of data access used by out-of-core algorithms designed on this model. We also discuss the extent to which existing file system interfaces support these out-of-core algorithms. Section 3 details the primary routines in our API. Section 4 discusses the implemention of the I/O-optimal algorithms for performing matrix-matrix addition, matrix-matrix multiplication, and BMMC (bit-matrix-multiply/complement) permutations using our API. Section 5 contains some concluding remarks.

2 Background

Researchers have developed theoretical models to capture important features of data movement between main memory and secondary storage [AV88, Flo72, VS90, VS94]. In particular, Vitter and Shriver's Parallel Disk Model (PDM) [VS94] provides a reasonable model for the design of I/O-efficient algorithms and analysis at a feasible level of abstraction. A number of I/O-efficient algorithms have been designed on PDM to solve problems such as sorting [AP94, Arg95, NV93, VS94], general permuting [VS94], BMMC permutations [Cor92, Cor93, CSW94, Wis95], mesh and torus permutations [Cor92, Wis95], matrix-matrix multiplication [VS94], matrix transpose [Cor92, Cor93, CSW94, VS94], FFT [VS94], LU decomposition [WGWR93], graph algorithms [CGG+95], and geometric algorithms [AVV95, GTVV93].

In this section, we define the parameters and data layout for PDM. We also describe the types of data access needed by PDM algorithms and discuss how well existing file systems support these types of data access.

2.1 The Parallel Disk Model (PDM)

In PDM, computer memory consists of two levels: main memory and secondary storage. We partition secondary storage into D disks, which we refer to as the *parallel disk system*. The parameters of PDM are

$$\begin{aligned} N &= \text{number of input records,} \\ M &= \text{number of records that fit in main memory,} \\ B &= \text{number of records that fit in a single disk block,} \\ D &= \text{number of disk blocks that can be transferred concurrently} \\ & \text{(typically, the number of disks),} \\ P &= \text{number of processors,} \end{aligned}$$

with the restrictions that $1 \leq B \leq M/2$, $M < N$, and $BD \leq M$. The first restriction requires that main memory hold at least two blocks to accommodate movement of records between blocks. The second restriction states that the problem size does not fit into main memory, i.e., we must use an out-of-core algorithm to solve the problem. The last restriction mandates that main memory be large enough so that all of the disks can be used to concurrently transfer data. See Figure 1 for an illustration of the model.

The parameters of PDM allow the development of general, portable algorithms by reflecting the physical limitations of a particular architecture. Therefore, algorithms developed on the model use the parameters to determine how much data to process at once, how many times to iterate over the entire data set, etc.

We measure the cost for data transfer between main memory and secondary storage in PDM in terms of *parallel I/Os*; the cost to read or write one block between main memory and each disk (i.e., D blocks) is one parallel I/O. We make two assumptions in defining a parallel I/O:

1. We transfer B contiguous records, starting at a physical block boundary on a disk, in parallel between main memory and a disk in one I/O.

2. We read or write only one block per disk during one parallel I/O.

In PDM, we divide main memory evenly among the processors; each processor can store M/P records. In terms of architecture models, PDM is a *distributed memory* model. Since the use of synchronized collective I/Os (i.e., all processors cooperate to request a large

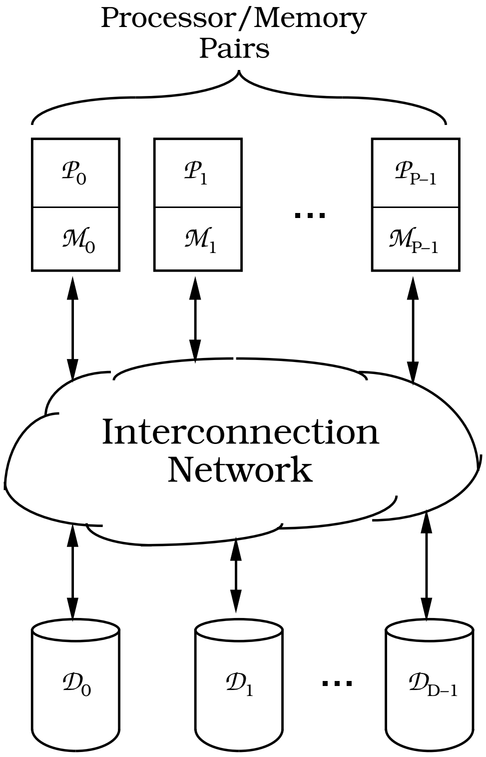


Figure 1: The multiprocessor PDM. The M records of internal memory are distributed over the P processors.

quantity of data at a barrier point) is the natural way to access data in algorithms developed on PDM, we consider PDM either a SIMD (single instruction multiple data) model or a loosely synchronous SPMD (single program multiple data) model. A *loosely synchronous computation* is one in which all the participating processors alternate between phases of computation and I/O [BBS$^+$94].

We assume a striped file across the disks with a striping unit of one block per disk, i.e., the blocks are placed on disk in a round-robin fashion as shown in Figure 2. A *stripe* consists of the D blocks at the same location on all D disks.[2] A *stripeload* is a quantity of BD records (the amount of data that would fit into a stripe); main memory can hold M/BD stripeloads. A stripeload of data may be accessed by a single parallel command, and these records may belong to different stripes. If we consecutively number the stripes of a file starting with stripe 0, we define the *block offset* of a particular block as the stripe number on which it

[2] Modern disks handle bad blocks and sectors below this level, so they are not a concern for us.

	$\mathcal{D}_0$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3$		$\mathcal{D}_4$	
stripe 0	0	1	2	3	4	5	6	7	8	9
stripe 1	10	11	12	13	14	15	16	17	18	19
stripe 2	20	21	22	23	24	25	26	27	28	29
stripe 3	30	31	32	33	34	35	36	37	38	39

Figure 2: The layout of $N = 40$ records in a parallel disk system with $B = 2$ and $D = 5$. Each box represents one block. The number of tracks needed is $N/BD = 4$. Numbers indicate record indices.

resides. One method of processing data is to repeatedly fill up the entire main memory with M records, which we call a *memoryload* (starting at a record index $x \equiv 0 \bmod M$), perform operations on that data, then write all of those M records out to disk. We use this technique in Section 4.

Algorithm designers use standard algorithmic techniques on PDM such as divide-and-conquer, recursion, and iterations over partitioned data. These techniques are well understood for developing algorithms when the entire data set can fit in the main memory of the processors.

Developing an out-of-core algorithm for PDM, however, requires the additional considerations of minimizing disk accesses and of evenly balancing the records to be read or written across the disks. Algorithms developed on PDM use both randomized and deterministic techniques for placing individual records on disk to minimize the total number of parallel I/Os. These techniques use all available disks by requesting only "full" I/Os, that is, parallel disk accesses that read or write the same number of blocks from each disk. Thus, we evenly balance the read or written records across the disks. In particular, these techniques primarily use two types of parallel disk access, independent and consecutive access, as well as striped access, which is a special case of independent access.

In an *independent access*, we read or write one or more blocks between main memory and disk, starting at a possibly different block offset on each disk. That is, the blocks accessed need not be consecutive according to the file layout. Figure 3 illustrates an independent access. With the power of independent access, the algorithm designer can request any "full" I/O. Thus, the algorithm designer has the flexibility to choreograph the data transfer at a block granularity by dictating exactly which blocks to read or write on each parallel disk access.

A *striped access* is the special case of independent access which simultaneously accesses

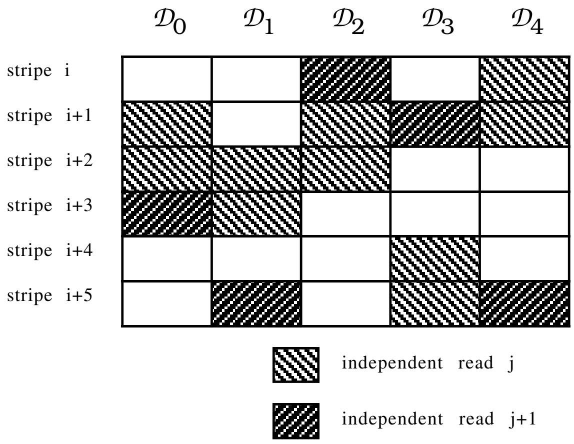


Figure 3: Independent reading of data. All of the reads can occur at a different location on each disk. Note that independent accesses may read more than one consecutive block from each disk, as is the case for independent read j.

the block at the same block offset on each disk. Striped access is useful to access data on disk systems that must operate in lock step (e.g., RAID).

Many algorithms take advantage of the simplicity and seek-time efficiency of accessing the records consecutively. In a *consecutive access*, we read or write one or more stripes beginning with the stripe pointed at by a *consecutive file-access pointer* as pictured in Figure 4. Immediately after the consecutive access occurs, the consecutive file-access pointer points to the stripe after the last stripe just accessed. If the parallel disk system lays out consecutive logical blocks of a file in a physically consecutive fashion, consecutive access can result in small seek times when accessing a file sequentially (e.g., by memoryload access).

2.2 Parallel disk access in file systems and runtime libraries

Cormen and Kotz [CK94] have identified several capabilities that a file system should support to enable high-performance implementations of the I/O-efficient algorithms. They point out that extensions of the traditional single-offset I/O interface for parallel file systems cannot allow independent access across disks. Most current file system APIs do not include multiple-offset (one per disk) independent-access routines which could be used to easily implement the existing I/O-efficient algorithms. Thus, explicitly programming independent access can be difficult to develop and maintain.

The application programmer interfaces of file systems fall into one of two classes: Unix-

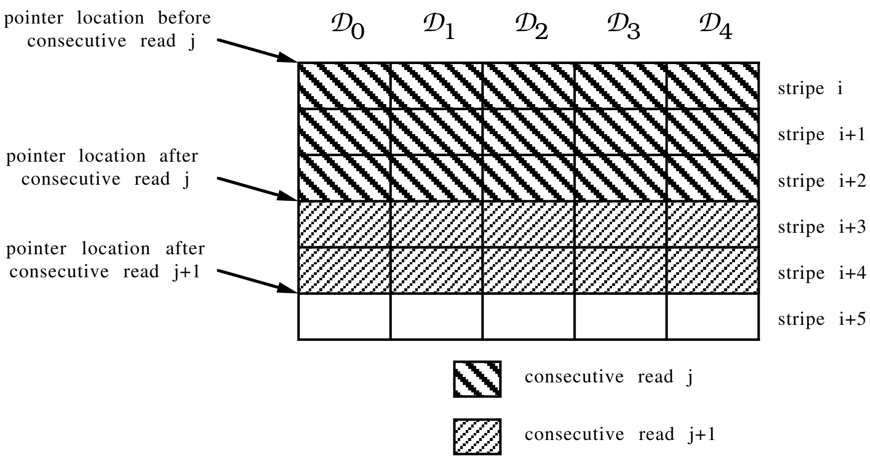


Figure 4: Consecutive reads of data starting at the read file-access pointer. The starting offset and the number of records in a consecutive read are multiples of the stripe size. Here, consecutive read j is of three stripes, and consecutive read $j+1$ is of two stripes.

like or non-Unix-like. Both have advantages and disadvantages. The Unix-like interfaces view a file as an addressable, linear stream of bytes accessible via operations such as `open`, `close`, `read`, and `write`. A Unix-like API allows portability so that program development can occur on a workstation even if the target platform is a scalable system. It also supports code that has already been written. The non-Unix-like APIs utilize the parallel nature of both the processors and the disks. If there are translation routines from the non-Unix-like routines to the Unix-like routines, development of programs that use non-Unix-like APIs can also occur in a workstation environment.

A number of existing file systems provide low-level API support for simultaneous, independent, direct access to the multiple disks provided by modern machine architectures. These APIs include extensions to the conventional Unix interface (e.g., [NK95]), modifications to the conventional Unix interface (e.g., [GS95]), and other interfaces which differ from Unix (e.g., [CFF+95, CFPB93]). An I/O-efficient algorithm would not be easy to program using these low-level file system APIs since the programmer must map the high-level parallel disk accesses to low-level file system operations.

At a higher level of abstraction, run-time libraries achieve I/O-performance improvements on multiple disk systems, but typically lose the direct disk access in the abstraction. The Panda run-time library [SCJ+95, SW94] achieves performance improvements by providing an API and more efficient layout alternatives for multidimensional array data. This approach takes advantage of the spatiotemporal locality of the array. Jovian [BBS+94] and PASSION [CBH+94] also provide support for efficient array data access, but abstract away direct disk access from the programmer. The Transparent Parallel I/O Environment (TPIE) [Ven94] provides a high-level access method interface (AMI) to the I/O paradigms that have already been developed, but does not allow the user to explicitly program disk accesses. Unfortunately, none of these run-time libraries include routines for simultaneous, direct access to the disks.

The API of PPFS provides routines that support independent access at the record level, not at the disk level [EHKM94, HER+95]. This API provides multiple-offset routines that allow the user to access more than one range of data in a single request, but does not provide a disk abstraction that allows the user to request particular blocks from each disk.

Our API can be implemented on top of any file system. It would be easiest to implement and would have the least performance overhead when being implemented on a file system with routines for direct disk or RAID access. Researchers at Sandia National Laboratories have developed the Whiptail File System (WFS) [SWC+95], the first implementation of our API. WFS is a prototype file system built on top of the Parallel File System (PFS) on the Intel Paragon. PFS provides direct access to each of the local RAIDs on the Intel Paragon. WFS uses this direct RAID access to provide the multiple block offset independent access needed by the I/O-efficient algorithms. Preliminary performance measurements show that the implementation of our API adds very little to the file system overhead. We shall be performing more performance tests to verify this claim and to gauge the throughput of the data-access routines.

3 Description of our API

In this section, we describe the C-callable API data-access routines included in our implementation of the Whiptail File System. These routines allow the programmer to access data in a manner that enables easy implementation of the I/O-efficient algorithms designed on PDM. The API routines provide direct access to particular blocks or stripeloads of a file on the parallel disk system. Thus, the programmer can choreograph the data movement by reading and writing specific blocks or stripeloads of a file according to the disk access patterns detailed by the I/O-efficient algorithms.

The API routines assume that the programmer uses the loosely synchronous SPMD programming model. Many of the API data-access routines serve as a barrier, not allowing the application to continue on any processor until all of the processors have completed the routine. The block-access routines are the only routines described below that do not enforce this synchronization.

The API includes primitives for both blocking and non-blocking reads and writes. Each of the blocking routines has an almost identical non-blocking counterpart with an extra flag parameter, which is incremented after the completion of the data access. The non-blocking routines allow for overlap of I/O and computation. In this paper, we present only the blocking routines and the non-blocking routine to read a block (as an example of a non-blocking counterpart).

We stripe a file across the disks in the storage system with a striping unit of B records and a striping factor of D disks. In describing our API, we present the primitives that define the data access interface to files. Most of these primitives are *synchronous* and *collective*, that is, all processors cooperate to request data at a barrier point. We assume the existence of global queries which our API may call to obtain the values of machine parameters, e.g., the PDM parameters B and D.

3.1 Block access

Our API provides the following routines for a processor to directly read or write any one particular block of a file:

```
int read_block (int fd, void *buffer_pointer, int disk_num,
                int block_offset);
int write_block (int fd, void *buffer_pointer, int disk_num,
                 int block_offset);
int iread_block (int fd, volatile int *read_flag, unsigned int *error,
                 void *buffer_pointer, int disk_num, int block_offset);
```

The `read_block()` and `write_block()` routines wait until the read or write of the requested block completes before returning. The programmer specifies the desired block by the `disk_num` and `block_offset` parameters for the file specified by the file descriptor (`fd`).[3]

[3]Our API assumes that straightforward routines exist to open and close a file and that the routine to open the file returns a file descriptor (`fd`).

The `buffer_pointer` identifies the local memory location for the read or written data.

The `iread_block()` routine does not wait for the reading of the block to complete. An `iread_block()` call passes a pointer to an incrementable `read_flag` variable which the file system increments upon completion of the read. The programmer can poll the `read_flag` variable to learn whether the reading of a particular block has completed. Calls to `iread_block()` routine complete in the order that they are invoked. If an error occurs during the execution of the `iread_block()` routine, a specified `error` variable would contain the appropriate error code.

3.2 Independent access

Our API provides independent-access routines which allow each processor to simultaneously access a portion of one or more stripeloads of data. Our API supports the following independent-access routines:

```
int read_independent_stripeloads (int fd, int num_stripeloads,
                                  int *block_offset_array,
                                  void *buffer_pointer, int buffer_size);
int write_independent_stripeloads (int fd, int num_stripeloads,
                                   int *block_offset_array,
                                   void *buffer_pointer, int buffer_size);
```

The programmer must specify the file descriptor (`fd`), the number of blocks to be read per disk (`num_stripeloads`), a block offset for each disk to be accessed (`block_offset_array`), a buffer space in the local memory of the calling processor for the read or written records (`buffer_pointer`), and the size of the buffer space (`buffer_size`). Each calling processor must provide the same values for the `fd`, `num_stripeloads`, and `block_offset_array` parameters, but may provide different values for the `buffer_pointer` and `buffer_size` parameters, which we further describe below.

The independent-access routines are *global-access* routines; that is, all the processors must call the same routine to collectively access a quantity of data. Global-access routines distribute the responsibility for receiving or providing the quantity of data read or written, respectively, over all the processors. Because the I/O-efficient algorithms determine which blocks to access on each parallel disk access and not the distribution of records across the processors, for a particular parallel disk access, each processor passes the same num_stripeloads and block_offset_array parameters to collectively specify the quantity and starting locations of the data on each disk, respectively. The inclusion of the block_offset_array parameter is a unique feature of our API which provides the programmer the ability to specify simultaneous, independent, direct access to the disks in a single operation.

The programmer may specify different values for the buffer_pointer and buffer_size parameters to specify the distribution of records across the processors for a particular parallel disk access, e.g., in the case when the number of processors does not equally divide the number of records requested. During a global read, the parallel disk system distributes the data across the processors in a round-robin fashion. The global read requests an ordered number of blocks, where the first block is the first block read from the first disk, the second block is the first block read from the second disk, etc. Processor 0 receives the first $\texttt{buffer\_size}_0$ bytes from the ordered number of blocks, where $\texttt{buffer\_size}_0$ is the number of bytes requested by processor 0. Processor 1 receives the next $\texttt{buffer\_size}_1$ bytes from the ordered number of blocks, etc. Thus, for $k = 0, 1, \ldots, P-1$, each processor k receives the $\texttt{buffer\_size}_k$ bytes starting at byte $b = \sum_{j=0}^{k-1} \texttt{buffer\_size}_j$. A global write gathers data for writing an ordered number of blocks in a similar fashion. Each $\texttt{buffer\_size}_k$ must be a sufficiently-large integral number of blocks.

3.3 Striped access

Our API provides striped-access routines which allow each processor to access blocks at the same block offset on each disk.[4] Our API supports the following striped-access routines:

```
int read_striped_stripeloads (int fd, int num_stripeloads,
                              int block_offset,
                              void *buffer_pointer, int buffer_size);
int write_striped_stripeloads (int fd, int num_stripeloads,
                               int block_offset,
                               void *buffer_pointer, int buffer_size);
```

The parameters of the striped-access routines are the same as the independent-access routines except that we replace the block_offset_array parameter with a single block_offset

[4] Striped access is a special case of independent access; we include striped access because it is a natural way for programmers to access certain data structures. Striped-access routines may allow more efficient implementations than the more general independent-access routines in certain I/O subsystems (e.g., RAIDs).

which is used for each disk. Again, the `buffer_pointer` and `buffer_size` parameters may differ on each calling processor to specify a distribution of the data across the processors.

3.4 Consecutive access

Our API provides consecutive-access routines which allow each processor to simultaneously access a portion of one or more stripeloads of data. Our API supports the following consecutive-access routines:

```
int read_consecutive_stripeloads (int fd, int num_stripes,
                                  void *buffer_pointer, int buffer_size);
int write_consecutive_stripeloads (int fd, int num_stripes,
                                   void *buffer_pointer, int buffer_size);
```

The programmer needs to specify the file descriptor (`fd`), the number of stripes requested (`num_stripes`), the buffer space in the local memory for the read or written records (`buffer_pointer`), and the size of the buffer (`buffer_size`). The consecutive-access routines are also global-access routines; thus, the `buffer_pointer` and `buffer_size` parameters serve to specify the data distribution across the processors as in the independent- and striped-access routines.

The consecutive access routines use two separate consecutive file-access pointers per file, one for reading and one for writing. Upon opening a file, the file system must initialize these consecutive file-access pointers to point to the first stripe. After the user performs a consecutive read or write on the file, the file system increments the appropriate consecutive file-access pointer. Our API includes the following routines to reset the consecutive file-access pointers:

```
seek_read (int fd, int stripe_number);
seek_write (int fd, int stripe_number);
```

Because the independent- and striped-access routines explicitly specify which blocks to access, they do not use the consecutive file-access pointers.

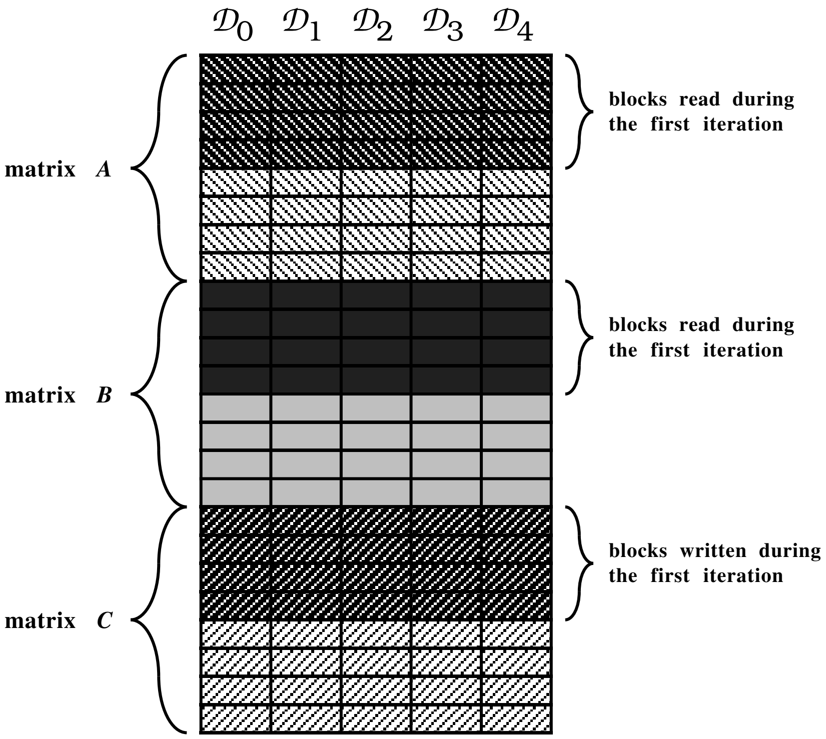


Figure 5: Data layout for out-of-core matrix-matrix addition with $M = 60B$. During the first iteration of the loop in Figure 6, we read the darker-shaded halves of matrices A and B and write the darker-shaded half of matrix C.

4 Library routines

In this section, we provide three sample applications that highlight the use of our API: matrix-matrix addition, matrix-matrix multiplication, and BMMC permutations. The implementations of these applications assume a data layout as defined in Section 2.

To shorten the presentation of our code, we use only local variables and omit the derivation of their values from global structures that contain the values of the machine parameters. We use indentation to show the nesting structure of our code, omitting curly braces.

4.1 Matrix-matrix addition

Matrix-matrix addition is an easy problem for which to design an out-of-core algorithm.[5]

[5]The files contain every element of the input and output matrices; dense matrices require this method of storage. We store the matrix in row-major order into full blocks.

```
void matrix_matrix_add (int fdA, int record_offsetA, int fdB, int record_offsetB,
                        int fdC, int record_offsetC, int num_rows, int num_columns)

   record_type *A, *B, *C; /* pointers in main memory to matrices */
   int memload_num, record_num; /* for-loop indexes */
   int local_num_records, num_memloads; /* for-loop bounds */
   int num_stripes, buffer_size;

   compute local_num_records, num_memloads, num_stripes, and buffer_size
   if record_offsetA, record_offsetB, or record_offsetC is not equal to 0
      call wfs_seek_read() and wfs_seek_write() to set file-access pointer
         to starting location in each matrix
   malloc local_num_records of record_type into A, B, and C

   for (memload_num = 0; memload_num < num_memloads; memload_num++)

      if last iteration /* if not processing a full memoryload */
         update local_num_records, num_stripes, and buffer_size

      /* perform matrix addition on a memoryload of records */

      /* read in a portion of matrix A and the corresponding portion in matrix B */
      read_consecutive_stripeloads (fdA, num_stripes, A, buffer_size);
      read_consecutive_stripeloads (fdB, num_stripes, B, buffer_size);

      /* compute corresponding portion of matrix C */
      for (record_num = 0; record_num < local_num_records; record_num++)
         C[record_num] = A[record_num] + B[record_num];

      /* write out the portion of matrix C */
      write_consecutive_stripeloads (fdC, num_stripes, C, buffer_size);
```

Figure 6: Out-of-core matrix-matrix addition using the consecutive-access routines of our API.

In our algorithm, each processor reads the same portion of the two operand matrices, computes their sum without a need for any interprocessor communication, and writes the corresponding portion of the output matrix to disk. Figure 5 shows the data layout on disk for a matrix-matrix addition.

Figure 6 shows the simple invocation of the consecutive-access routines in the code for matrix-matrix addition. The `read_consecutive_stripeloads()` and `write_consecutive_stripeloads()` calls access the matrices by reading a contiguous part of the input matrices and writing the corresponding part of the output matrix.

The input parameters to the `matrix_matrix_add()` function are file descriptors for the files containing the input matrices `A` (`fdA`) and `B` (`fdB`) and the output matrix `C` (`fdC`), the dimensions of the matrices, and, for each file, a record offset that specifies the first record of the matrix. These last parameters are necessary for the routine to be called with submatrices as input.

4.2 Matrix-matrix multiplication

There are several approaches to performing out-of-core matrix-matrix multiplication. These approaches include the standard recursive divide-and-conquer method presented in [VS94] and a method similar to the LU factorization algorithm presented in [WGWR93], which divides one of the matrices into groups of columns.

We implement a variation of Vitter and Shriver's recursive out-of-core algorithm [VS94]. This algorithm is as follows:

1. If $k \leq \sqrt{M}$, multiply the matrices internally. Otherwise do the following steps:

2. Subdivide A and B into eight $k/2 \times k/2$ submatrices: $A_{1,1}$–$A_{2,2}$ and $B_{1,1}$–$B_{2,2}$.

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}; \qquad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}.$$

   Reposition the records of the eight submatrices so that each submatrix is stored in row-major order.

3. Recursively compute the following:

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

4. Reposition $C_{1,1}$–$C_{2,2}$ so that C is stored in row-major order.

This method divides each of the input matrices into approximately equal-sized submatrices until the multiplication can be handled in memory (Steps 1 and 2). The algorithm explicitly moves the data of the submatrices on each level of recursion (Step 2). This data movement is not needed; Figure 7 shows an implementation of this method that performs the recursion implicitly.

Before performing any multiplications, we physically rearrange the records on disk such that the records of each submatrix reside contiguously in full blocks. The `reposition_matrix_on_disk()` routine in Figure 8 performs this repositioning on an input matrix. (The code in Figure 7 calls `reposition_matrix_on_disk()` with both input

```
void matrix_matrix_multiply (int fdA, int fdB, int fdC, int num_rowsA, int num_columnsA,
                             int num_rowsB, int num_columnsB)

   int rec_level, row, column; /* for-loop indexes */
   int num_recursion_groups = number of submatrices to be created/4; /* for-loop bound */
   int num_rowsC = num_rowsA, num_columnsC = num_columnsB;
   int tempC1, tempC2; /* pointers in main memory to temporary matrices */

   /* record offsets into the files for the starting location of each submatrix */
   int submatrixA[2][2][num_recursion_groups], submatrixB[2][2][num_recursion_groups],
      submatrixC[2][2][num_recursion_groups];

   if the number of records in matrices A, B, and C is less than local_num_records
      a designated processor performs the matrix-matrix multiply

   else /* subdivide the problem */

      /* subdivide input matrices and reposition them on disk */
      subdivide matrices A and B into recursion groups where each recursion group is 4 somewhat equal-sized submatrices
         that represent the last level of recursion
      create submatrix pointers into matrices A, B, and C so that the submatrices can be written
         in the same space on disk, returning submatrixA[][][], submatrixB[][][], and submatrixC[][][]
      reposition_matrix_on_disk (fdA, local_num_records, num_rowsA, num_columnsA, num_recursion_groups);
      reposition_matrix_on_disk (fdB, local_num_records, num_rowsB, num_columnsB, num_recursion_groups);

      /* compute submatrices C by working with each lowest-level recursion group independently */
      for (rec_level = 0; rec_level < num_recursion_groups; rec_level++)
         for (row = 0; row < 2; row++)
            for (column = 0; column < 2; column++)
               in_core_matrix_matrix_multiply (fdA, submatrixA[row][0][rec_level], fdB,
                  submatrixB[0][column][rec_level], tempC1, 0, parameters associated with size of submatrices);
               in_core_matrix_matrix_multiply (fdA, submatrixA[row][1][[rec_level], fdB,
                  submatrixB[row][1][rec_level], tempC2, 0, parameters associated with size of submatrices);
               in_core_matrix_matrix_add (tempC1, 0, tempC2, 0, C, submatrixC[row][column][rec_level],
                  num_rowsC, num_columnsC);

      position submatrices of matrix C on disk so that matrix C is stored in row-major order
```

Figure 7: Out-of-core matrix-matrix multiply using recursive divide-and-conquer algorithm.

matrices.) Figure 9 illustrates a repositioning of a matrix on disk using consecutive reads and independent writes.

Once the repositioning of the input matrices (A and B) has been completed, each block on disk contains records from only one submatrix. This repositioning may result in some partially-filled blocks. The submatrices are multiplied as in Vitter and Shriver's algorithm. After the submatrices of the output matrix have been computed, we reposition the output records such that the records of the output matrix are contiguous in row-major order. Effectively, this final repositioning performs the inverse of `reposition_matrix_on_disk()`.

```
void reposition_matrix_on_disk (int fd_matrix, int local_num_records, int num_rows,
                                int num_columns, num_recursion_groups)

   int memload_num, num_memloads;
   int read_num_stripes, write_num_stripes, read_buffer_size, write_buffer_size;
   int *block_offset_array;
   record_type *matrix, *matrix_offset;

   malloc local_num_records of record_type into matrix
   compute num_memloads, read_num_stripes, and read_buffer_size

   for (memload_num = 0; memload_num < num_memloads; memload_num++)

      /* read portion of fd_matrix */
      read_consecutive_stripeloads (fd_matrix, read_num_stripes, matrix, read_buffer_size);

      reposition the records in matrix into their appropriate submatrices, packing them into as many full blocks as
         possible (may require interprocessor communication)

      /* write portion of fd_matrix */
      while blocks need to be written
         compute write_num_stripes, write_buffer_size, block_offset_array, and matrix_offset
         write_independent_stripeloads (fd_matrix, write_num_stripes, block_offset_array,
            matrix_offset, write_buffer_size);

   free (matrix);
```

Figure 8: Subroutine to reposition a matrix using the consecutive-access and independent-access routines of our API.

4.3 BMMC permutations

The BMMC (bit-matrix-multiply/complement) permutations are a class of permutations that include such useful permutations as matrix transpose, binary-reflected Gray code, and the bit-reversal permutation (used by the FFT). We define a BMMC permutation of N records, where N is a power of 2, as a mapping of each $\lg N$-bit source address x to a unique $\lg N$-bit target address y by the transformation $y = A\,x \oplus c$, using matrix arithmetic over $GF(2)$,[6] where a nonsingular $\lg N \times \lg N$ bit matrix A and $\lg N$-bit complement vector c characterize the permutation.

Cormen, Sundquist, and Wisniewski [CSW94] show how to perform any BMMC permutation in an asymptotically optimal number of parallel I/Os. Their algorithm decomposes the permutation into a series of permutations, each of which can be performed in one pass over the data. To perform each one-pass permutation, we read each source memoryload of records from disk into main memory, permute that memoryload into full target blocks, then

[6]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 replace by exclusive-or.

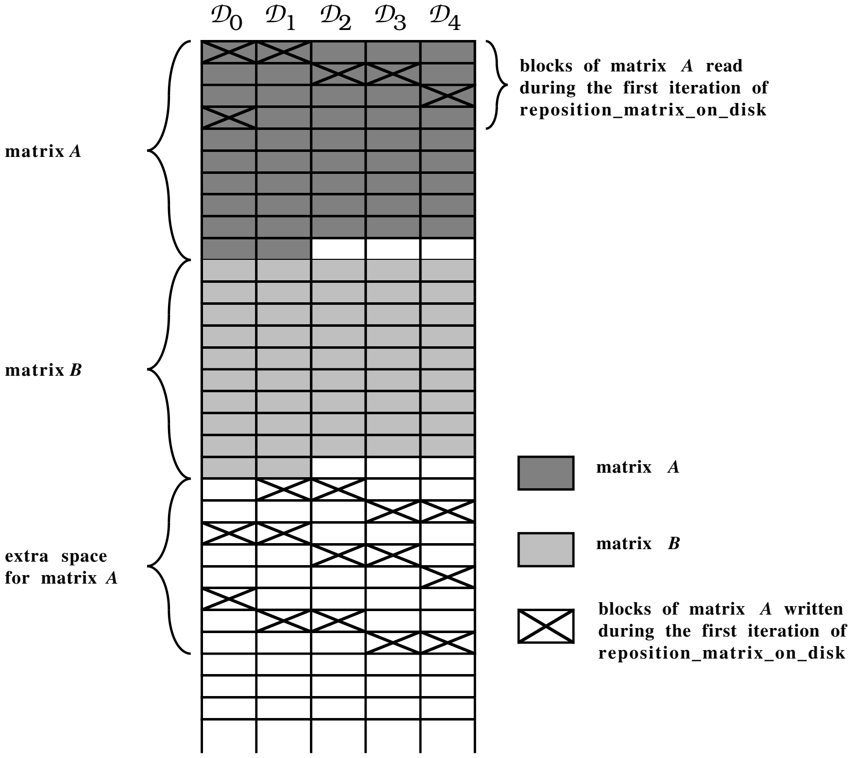


Figure 9: Repositioning the data on disk for a matrix-matrix multiply where $M = 20B$. We consecutively read 20 blocks of the first memoryload of the matrix A and independently write the blocks in the form of submatrix blocks to disk in the original location of matrix A and the extra space.

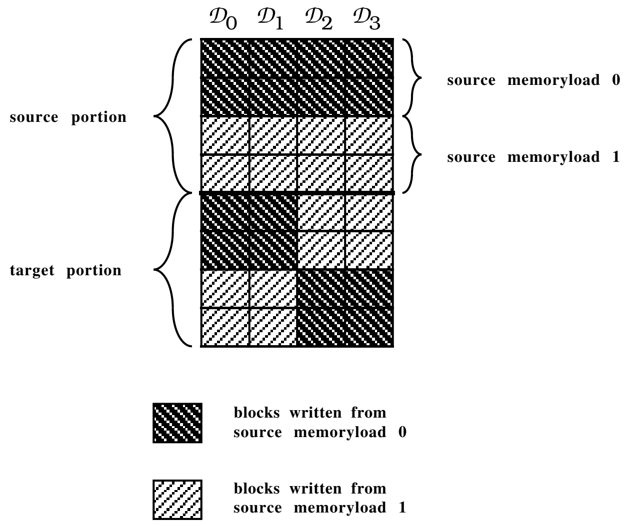
write those blocks to disk, using independent access. Figure 10 shows a BMMC permutation that we can perform in one pass over the data using striped reads and independent writes.

We only demonstrate the use of our API routines for BMMC permutations and thereby omit the code that factors the matrix A into a series of factors each of which characterize one-pass permutations. For each factor, all processors call the `one_pass_permute()` routine, shown in Figure 11, passing a bit matrix `factor` that characterizes a one-pass permutation.

The `one_pass_permute()` routine repeatedly reads the next M consecutive source records into memory, permutes those records in memory into full target blocks, and writes the target blocks to disk. The `one_pass_permute()` routine allocates two buffers, `read_buffer` and `write_buffer`, to hold the source blocks and target blocks, respectively. The `read_consecutive_stripeloads` routine reads the M/BD consecutive stripes of the next source memoryload. After the source memoryload resides in memory,

$$A = \begin{bmatrix} 1&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&1&0 \\ 0&0&1&0&1 \end{bmatrix} \qquad c = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

(a)



(b)

Figure 10: (a) The characteristic matrix A and complement vector c for a BMMC permutation, where $B = 2$, $D = 4$, $M = 16$, and $N = 32$. (b) Data layout after performing the BMMC permutation characterized by the matrix A. The data initially reside in the source portion. After performing the permutation, the data reside in the target portion. Since the records of each source memoryload move to different halves of the target memoryload, we perform this permutation with consecutive reads and independent writes.

the `permute_in_memory()` routine permutes the records of that memoryload from `read_buffer` to `write_buffer` according to the bit matrix `factor`, which may involve transmitting records among processors. Before proceeding to read the next memoryload, each processor writes M/BD full target blocks to each of the D disks using the `write_independent_stripeloads()` routine. For each stripeload of target blocks, the bit matrix `factor` determines which blocks to write to each disk, i.e., the appropriate entries for `block_offset_array`.

5 Conclusions

This paper constitutes a demonstration that it is feasible and indeed reasonable to produce programs by implementing I/O-optimal algorithms designed with the structure provided by the Parallel Disk Model. In particular, we implemented our API as the interface for the Whiptail File System on the Paragon and implemented out-of-core algorithms for matrix-matrix addition, matrix-matrix multiplication, and BMMC permutations. Our API played

```
void one_pass_permute (bit_matrix factor, int input_fd, int output_fd)

   int num_local_records, stripes_per_memoryload;
   record_type *read_buffer, *write_buffer;
   int read_buffer_size, write_buffer_size;

   compute num_local_records, read_buffer_size, and write_buffer_size
   malloc num_local_records records of record_type into read_buffer
   malloc num_local_records records of record_type into write_buffer

   for each source memoryload
      /* reading, processing, and writing memoryloads */

      /* read memoryload */
      read_consecutive_stripeloads (input_fd, stripes_per_memoryload, read_buffer, read_buffer_size);

      /* permute the memoryload residing in memory */
      permute_in_memory (factor, read_buffer, write_buffer);

      /* write memoryload */
      for each target stripe
         use factor to compute block_offset_array
         write_independent_stripeloads (output_fd, 1, block_offset_array, write_buffer, write_buffer_size);

   free (read_buffer);
   free (write_buffer);
```

Figure 11: Out-of-core one-pass permutations using consecutive reads and independent writes.

an important role in the smooth transition from algorithms to programs by providing easy-to-use, simple-to-understand data-access routines for programming algorithms designed on the Parallel Disk Model.

Several potentially useful extensions of our API exist. For example, we can add parameters to the data-access primitives to specify the ordering of the disks and the processors. These new parameters would have the effect of changing the layout of the accessed data across the distributed memory of the processors. This capability could eliminate unnecessary interprocessor communication immediately after completing the data access. Another useful extension would be the inclusion of a *broadcast-read* primitive so that all processors could read the exact same data.

One of our research goals is to determine if our API facilitates application programming. While ease of use and clarity of code are important, resulting performance often becomes the determining factor for the feasibility/reasonability question in high-performance computing. We need performance measurements to determine if our API adds significant overhead to the underlying file-system data-access routines.

Another interesting extension of our research would be to analyze the benefit of overlapping disk I/O with computation and/or interprocessor communication. How can we best use the non-blocking versions of the data-access routines presented in this paper to more easily achieve the greatest possible overlap? Future experiments will also examine the tradeoffs of partitioning main memory into multiple buffers for reading, writing, and computing.

Acknowledgments

We thank David Greenberg and David Womble for putting together the WFS team and providing us with the use of the Paragon at Sandia. The WFS team, which includes Bruce Calder, David Greenberg, Ryan Moore, and David Womble, co-developed WFS with us during the summer of 1994. Tom Cormen, Alan Siegel, John Wilkes, Bruce Shriver, Dave Kotz, Sumit Chawla, and Melissa Hirschl provided us with many helpful comments on the various drafts of this paper.

References

[ACFS94] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2/3):72–109, August and September 1994.

[AP94] Alok Aggarwal and C. Greg Plaxton. Optimal parallel sorting in multi-level storage. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 659–668, Arlington, VA, January 1994.

[Arg95] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *4th International Workshop on Algorithms and Data Structures (Proceedings)*, Lecture Notes in Computer Science, number 955, pages 334–345, Kingston, Canada, August 1995. Springer-Verlag.

[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] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. To appear, *Third European Symposium on Algorithms*, 1995.

[BBS+94] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*. IEEE Computer Society Press, October 1994.

[CBF93] Peter F. Corbett, Sandra Johnson Baylor, and Dror G. Feitelson. Overview of the Vesta parallel file system. In *Proceedings of IPPS '93 Workshop on I/O in Parallel Computer Systems*, pages 1–16, April 1993. Reprinted in Computer Architecture News, December 1993.

[CBH+94] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[CC94] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS–TR94–243, Dartmouth College Department of Computer Science, November 1994.

[CFF+95] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, April 1995.

[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, November 1993.

[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, San Francisco, CA, January 1995.

[CK94] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. Technical Report PCS–TR93–188, Dartmouth College Department of Computer Science, September 1994. Earlier version appeared in *Proceedings of the 1993 DAGS/PC Symposium*, Hanover, NH, pages 64–74, June 1993.

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

[Cor93] Thomas H. Cormen. Fast permuting on 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. Extended abstract appeared in *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, Velen, Germany, June 1993.

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

[EHKM94] Christopher L. Elford, Jay Huber, Chris Kuszmaul, and Tara Madhyastha. Portable parallel file system detailed design. Technical report, Department of Computer Science, University of Illinois, September 1994.

[Flo72] R. W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Calculations*, pages 105–109. Plenum Press, New York, 1972.

[GS95] Garth Gibson and Daniel Stodolsky. Issues arising in the SIO-OS low-level PFS API. Presentation at the Scalable Input/Output for High Performance Computers Workshop at the Fifth Symposium on the Frontiers of Massively Parallel Computation, February 1995.

[GTVV93] M. H. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometery. In *Proceedings of the 34th Annual Symposium on*

Foundations of Computer Science, pages 714–723, Palo Alto, CA, November 1993.

[HER+95] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995.

[Kim86] Michelle Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[KS93] Orran Krieger and Michael Stumm. HFS: A flexible file system for large-scale multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 6–14, Hanover, NH, June 1993.

[NK95] Nils Nieuwejaar and David Kotz. A multiprocessor extension to the conventional file system interface. Technical Report PCS-TR95-253, Dartmouth College, 1995.

[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, Velen, Germany, June 1993.

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

[PGS93] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. Informed prefetching: Converting high throughput to low latency. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993.

[SCJ+95] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995. To appear.

[SGM86] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.

[SW94] K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In *Proceedings of Supercomputing '94*, pages 650–659, November 1994.

[SWC+95] Elizabeth A. M. Shriver, Leonard F. Wisniewski, Bruce G. Calder, David Greenberg, Ryan Moore, and David Womble. Parallel disk access using the Whiptail File System: Design and implementation. Manuscript, 1995.

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

[VS90] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Optimal disk I/O with parallel block transfer. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 159–169, Baltimore, MD, 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 *Proceedings of the 1993 DAGS/PC Symposium*, pages 56–63, Hanover, NH, June 1993.

[Wis95] Leonard F. Wisniewski. Structured permuting in place on parallel disk systems. Technical Report PCS–TR95–265, Dartmouth College Department of Computer Science, September 1995.