Dartmouth College
Dartmouth Digital Commons

Dartmouth College Undergraduate Theses
Theses and Dissertations

6-7-1995

A Multiple Discrete Pass Algorithm on a DEC Alpha 2100

Scott R. Cushman
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/senior_theses

Part of the Computer Sciences Commons

Recommended Citation
Cushman, Scott R., "A Multiple Discrete Pass Algorithm on a DEC Alpha 2100" (1995). *Dartmouth College Undergraduate Theses*. 172.
https://digitalcommons.dartmouth.edu/senior_theses/172

This Thesis (Undergraduate) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Undergraduate Theses by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.

Dartmouth College Computer Science
Technical Report PCS-TR95-259

A Multiple Discrete Pass Algorithm on a DEC Alpha 2100

Scott R. Cushman

June 7, 1995

Contents

1 Introduction **1**

1.1 Contributions of This Thesis 1

1.2 Organization 2

2 Environment **3**

2.1 The Vitter-Shriver Parallel-Disk Model 3

2.2 The DEC Alpha 2100 5

2.3 Multiple-Pass Programs 7

3 Implementation of an Optimal BMMC Algorithm **9**

3.1 BMMC Permutations 9

3.2 MRC, MLD, and MLDI Permutations 11

3.3 The CSW Algorithm 13

3.4 History of My Implementation 14

3.4.1 Single-Disk Simulation 14

3.4.2 Parallel-Disk Version 17

4 Testing **18**

4.1 Goals and Predictions 18

4.1.1 Organization of Source and Target Portions 18

4.1.2 Optimal Memory and Block Sizes 20

4.2 Results 20

5 Conclusion **26**

Chapter 1

Introduction

Computer programs sometimes must process data sets that are too large to hold in main memory and must therefore be kept on a secondary storage system (generally magnetic disks). When a program modifies a large amount of data stored on a disk, the time required to move information between disk and memory can slow the process enormously. One means of improving performance in these conditions is the parallel-disk system.

1.1 Contributions of This Thesis

This thesis presents the results obtained of a particular I/O-intensive algorithm on one particular platform—a DEC Alpha 2100 with an eight-disk array. Using the Vitter-Shriver parallel-disk model for organizing and accessing data on parallel disks, we address the following questions:

- When moving data from one portion of storage to another, on a system with D disks, is it faster to have both portions spread across all D disks, or to keep one portion on only $D/2$ disks and the second portion on the

other $D/2$?

- What are the optimal sizes for the buffers holding data in memory and the blocks that of data that are moved between main memory and the disks?

Our results indicate that it is faster to spread the data across all available disks. For the algorithm we tested, times were fastest when we kept the memory size large enough and the block size small enough to minimize the number of times data is reordered, i.e., the number of distinct passes over the data. Within this constraint, smaller memory sizes and larger block sizes reduced our program's running time.

As a secondary contribution, this thesis describes an implementation of an asymptotically optimal algorithm for performing BMMC permutations (a specialized form of permutation), which we used in our testing.

1.2 Organization

The remaining chapters of this thesis are organized as follows. Chapter 2 describes the testing environment. It presents the Vitter-Shriver model for organizing and accessing data on parallel disks (for which the BMMC algorithm was developed), along with the DEC Alpha 2100 used in our tests and the general characteristics of the program we examine. Chapter 3 explains BMMC permutations and an algorithm to perform them, gives a brief history of the algorithm's implementation, and describes some of the issues encountered in creating the implementation. Chapter 4 describes the tests that we ran and interprets their results. Finally, Chapter 5 contains some concluding remarks.

Chapter 2

Environment

This chapter describes the testing environment. We discuss the model used for organizing data on a parallel-disk system, the machine used (focusing on its I/O devices), relevant characteristics of the program chosen for running tests, and our options for dividing the available disk space into source and destination portions.

2.1 The Vitter-Shriver Parallel-Disk Model

We organize our information according to the Vitter-Shriver [VS94] model for organizing and accessing information on a parallel-disk system. In this model, data consists of N *records*, evenly distributed over D *disks*. The disks are denoted $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$. Each disk is organized into *blocks* of B records, with any disk read or write performed on an entire block. As Figure 2.1 shows, disk I/O takes place between the disks and a random-access *memory* with the capacity to hold up to M records. The records are spread across disks in *stripes*, where the D blocks at the same location on each of the disks make up one stripe,

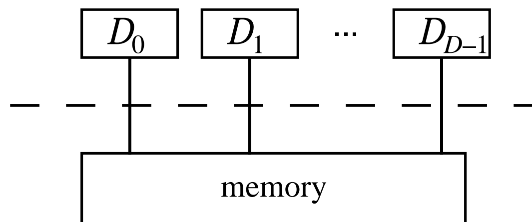


Figure 2.1: A parallel-disk system. In a parallel I/O operation, D blocks are transferred across the dashed line, with one block transferred between memory and each of the D disks.

and we have $S = N/BD$ stripes.

Disk accesses are described in terms of *parallel read operations* and *parallel write operations*, in which at most D blocks are transferred between the disks and memory, with at most one block transferred per disk. Throughout this thesis, all parallel I/O operations transfer exactly D blocks, one for each disk. There are two basic forms of parallel I/O operations: *striped I/O*, in which the blocks transferred in an operation are all members of the same stripe, and *independent I/O*, in which the blocks may be located anywhere on their respective disks.

A parallel-disk algorithm's cost is how many parallel I/O operations it requires. An optimal algorithm minimizes the number of times blocks cross the dashed line separating memory and disks in Figure 2.1.

The model places some restrictions on the values allowed for N, M, B, and D. Each parameter must be an exact power of 2. We require $M \geq BD$ so that all the records transferred in one parallel I/O operation may be held in memory. To force our tests to use extensive I/O, we require that $M < N$. Following the

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

Figure 2.2: The layout of a file containing $N = 32$ records on a parallel-disk system with $D = 4$ and $B = 2$. Numbers indicate record indices.

example introduced in [Cor93], we use the following notation:

$$n = \lg N, \quad m = \lg M, \quad b = \lg B, \quad d = \lg D, \quad s = \lg S.$$

Figure 2.2 shows the organization of files striped according to the Vitter-Shriver model. Each record has an *index* indicating its location in the file relative to the other records. As Figure 2.3 shows, an index's least significant b bits indicate its offset within a block, followed by d bits indicating which disk it is stored on, and finally by s bits indicating which stripe it is in. We indicate a record's index as an n-bit vector x with the least significant bits first: $x = (x_0, x_1, \ldots, x_{n-1})$.

2.2 The DEC Alpha 2100

Adams, the DEC Alpha 2100 with which we work, runs the DEC OSF/1 V3.2 operating system. The machine has two 175-MHz DEC Alpha CPUs, 320 MB of RAM, and nine 2-gigabyte SCSI-2 disks. One of these disks stores the system software, and we use the other eight as raw data devices. We use raw I/O (discussed further in Section 3.4.2) because it does not cache data in memory and so the I/O times we record correspond to the actual time required to transfer information between disk and memory. In our tests, we spread files over one,

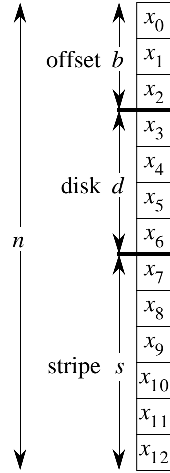


Figure 2.3: Parsing the index $x = (x_0, x_1, \ldots, x_{n-1})$ of a record on a parallel-disk system. Here, $n = 13, b = 3, d = 4$, and $s = 6$. The least significant b bits contain the offset of a record within its block, the next d bits contain the disk number, and the most significant s bits contain the stripe number.

two, four, or all eight of the raw disks.

When we let R equal the size in bytes of one record and $r = \lg R$, we are able to quantify a number of additional restrictions that adams imposes on our test parameters:

- $D \leq 8$.
- $NR/D \leq 2^{31}$, or $n \leq 31 + d - r$, since we have NR bytes of data and each disk stores up to 2^{31} bytes.
- $MR \leq 2^{26}$, or $m \leq 26 - r$. Our implementation of the BMMC algorithm divides memory into four buffers, where M indicates the number of records each buffer is capable of holding. Because M must be a power of 2 and adams has 320 MB of memory, each buffer is at most 64 MB, or 2^{26} bytes.

- $2^9 \leq BR \leq 2^{16}$, or $9 \leq (b+r) \leq 16$. In order to use the raw I/O routines provided by Digital, each access to each disk (BR bytes) must be between 512 and 64K bytes.

2.3 Multiple-Pass Programs

For our parallel-disk tests, we ran a program with certain generalizable characteristics. Since we are interested in the performance of the parallel-disk system, we wanted to observe high levels of I/O activity.

More specifically, we chose to run a program that accesses data in multiple discrete passes. Given an N-record file, each of the records is read from disk and written back to disk exactly once during a pass, and the ith pass must finish before the $(i+1)$st pass begins. Many of the optimal algorithms in the literature for the Vitter-Shriver model operate with multiple discrete passes.

The algorithm described in this thesis divides the available disk space in half, creating a *source portion* and a *target portion* of storage. Each pass of the algorithm reads the records from the source portion, reorders them, and writes them to the target portion of the disks. Then the target portion and source portion exchange roles for the next pass. Indices into both portions start at 0 and go to $N-1$.

Because several multiple-discrete pass algorithms can use this approach, it is important to determine how to most efficiently split the disks. We examine two choices—given D disks, either use $D/2$ of the disks as the source portion and the other $D/2$ as the target portion, or divide each of the D disks into a source

	$\mathcal{D}_0$		$\mathcal{D}_1$			$\mathcal{D}_2$		$\mathcal{D}_3$	
	Source-portion disks					Target-portion disks			
stripe 0	0	1	2	3		0	1	2	3
stripe 1	4	5	6	7		4	5	6	7
stripe 2	8	9	10	11		8	9	10	11
stripe 3	12	13	14	15		12	13	14	15

(a)

Source-portion stripes

	$\mathcal{D}_0$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3$	
stripe 0	0	1	2	3	4	5	6	7
stripe 1	8	9	10	11	12	13	14	15

Target-portion stripes

stripe 2	0	1	2	3	4	5	6	7
stripe 3	8	9	10	11	12	13	14	15

(b)

Figure 2.4: Dividing storage space into source and target portions, where $N = 16, D = 4$, and $B = 2$. Numbers indicate record indices. We either use (a) $D/2$ complete disks for each portion or (b) half of each disk's space for each portion.

half and a target half. Figure 2.4 illustrates these approaches. In Section 4.1.1, we discuss the potential advantages and disadvantages of both methods.

Chapter 3

Implementation of an Optimal BMMC Algorithm

This chapter discusses BMMC permutations and an implementation of the BMMC-permutation algorithm of Cormen, Sundquist, and Wisniewski.

3.1 BMMC Permutations

A *bit-matrix-multiply/complement* (*BMMC*) permutation is a reordering of elements based on element indices. When indices (into an N-record file in our case) are represented as n-bit vectors, we describe BMMC permutations in terms of an $n \times n$ *characteristic matrix* and an n-bit *complement vector*. Both the characteristic matrix and the complement vector consist of entries drawn from $\{0, 1\}$, and the characteristic matrix is nonsingular, or invertible, over $GF(2)$.[1]

Given a record's source index, we determine its target index by multiplying the characteristic matrix by the source index and adding the product of that operation to the complement vector. Alternatively, we can view the complement

[1] According to [Cor93, CSW94], "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."

vector as specifying a subset of bits of the product to complement. In general, if x is an n-bit source address, A is an $n \times n$ nonsingular characteristic matrix, c is an n-bit complement vector, and y is an n-bit target address, we define the BMMC mapping by

$$y = Ax \oplus c ,$$

where $\oplus$ is the symbol for the logical exclusive-or operation. Consider the following example, with $n = 4$. If

$$x = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \text{and} \quad c = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix},$$

then

$$\begin{aligned} y &= \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}. \end{aligned}$$

BMMC permutations have the advantage of concise representation. A reordering of N records may be completely represented in only $\lg^2 N + \lg N$, or $n^2 + n$, bits.

Bit-permute/complement, or *BPC*, permutations are an important subclass of BMMC permutations. These permutations have the additional restriction that the characteristic matrix must be a permutation matrix—each row and each column of the matrix must contain exactly one 1. We can view BPC

permutations as applying a fixed permutation to the address bits and then complementing some subset of the result. This class contains many common permutations, including matrix transposition, bit-reversal permutations, vector-reversal permutations, and matrix reblocking.

Other common permutations, such as Gray-code and inverse Gray-code are also BMMC. The matrix A used in the example above is the characteristic matrix for a Gray-code permutation.

3.2 MRC, MLD, and MLDI Permutations

This section defines three subclasses of BMMC permutations. The first two classes, MRC and MLD permutations, are used by the BMMC algorithm described in [CSW94]. The final BMMC subclass presented, MLDI permutations, replaces the MLDs in our implementation of this algorithm.

Memory-rearrange/complement, or *MRC*, permutations are BMMC permutations with additional restrictions placed on the characteristic matrix. The leading (upper left) $m \times m$ and trailing (lower right) $(n-m) \times (n-m)$ submatrices are both nonsingular, and the lower left $(n-m) \times m$ submatrix contains all 0s. The values in the upper right $m \times (n-m)$ submatrix are not restricted beyond the $\{0, 1\}$ requirement, and so an MRC permutation's characteristic matrix is of the form

$$\begin{array}{c} \begin{array}{cc} \quad m \quad & \quad n-m \quad \end{array} \\ \left[\begin{array}{c|c} \text{nonsingular} & \text{arbitrary} \\ \hline 0 & \text{nonsingular} \end{array}\right] \end{array} \begin{array}{l} \\ m \\ n-m \end{array} .$$

Gray-code and inverse Gray-code permutations are examples of the MRC subclass.

Cormen [Cor93] shows that any MRC permutation may be performed in one pass by reading, in turn, each *memoryload* (i.e., M records) of M/BD consecutive stripes from the source portion, permuting them in memory, and writing them out to a (possibly different) set of M/BD consecutive stripes in the target portion.

A *memoryload-dispersal*, or *MLD*, permutation [CSW94] has a characteristic matrix that is nonsingular and of the form

$$\begin{array}{c} \\ \\ \\ \\ \end{array}\begin{array}{cc} m & n-m \end{array}$$
$$\left[\begin{array}{c|c} \text{arbitrary} & \\ \hline \lambda & \text{arbitrary} \\ \hline \mu & \end{array}\right] \begin{array}{l} b \\ m-b \\ n-m \end{array} ,$$

subject to the *kernel condition*

$$\ker \lambda \subseteq \ker \mu \ ,$$

or $\lambda x = 0$ implies $\mu x = 0$. MLD permutations can also be performed in one pass, using striped reads and independent writes.

An *inverse memoryload-dispersal*, or *MLDI*, permutation is one whose characteristic matrix is the inverse of the characteristic matrix for some MLD permutation. Since all BMMC permutations are nonsingular, every MLD permutation has a corresponding MLDI permutation. The product y of an invertible matrix A and some bit vector x, when multiplied by the inverse A^{-1} of that matrix, produces the original bit vector x:

$$Ax = y \leftrightarrow A^{-1}y = x.$$

In other words, an MLDI permutation reverses its corresponding MLD permutation. Therefore, we can perform any MLDI permutation by reading records from

the target addresses of some MLD permutation, and writing them to that permutation's source addresses. MLDI permutations may therefore be performed in one pass using independent reads and striped writes.

3.3 The CSW Algorithm

In [CSW94], Cormen, Sundquist, and Wisniewski present a lower bound on the number of parallel I/O operations for an arbitrary BMMC permutation and an asymptotically optimal algorithm to perform BMMC permutations.

Cormen, Sundquist, and Wisniewski prove a lower bound for parallel I/Os for performing a BMMC permutation of

$$\Omega\left(\frac{N}{BD}\left(1+\frac{\operatorname{rank}\gamma}{\lg(M/B)}\right)\right),$$

where γ is the lower left $\lg(N/B) \times \lg B$ submatrix of the characteristic matrix. Their algorithm uses at most

$$\frac{2N}{BD}\left(\left\lceil\frac{\operatorname{rank}\gamma}{\lg(M/B)}\right\rceil+2\right)$$

parallel I/Os. Each one-pass factor permutation requires $2N/BD$ parallel I/Os to read and write each of the N/B blocks of data once, and $\left\lceil\frac{\operatorname{rank}\gamma}{\lg(M/B)}\right\rceil+2$ is the number of factor matrices produced, or the number of passes that will be made.

Given a characteristic matrix and complement vector for a BMMC permutation, the CSW algorithm decomposes the matrix into a series of factor matrices, each of which is the characteristic matrix for a single-pass permutation. The algorithm then performs each permutation in turn starting with the permuta-

tion represented by the rightmost matrix, with the original complement vector included in the final permutation. For example, given the characteristic matrix

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

when $n = 6, m = 4$, and $b = 3$, our implementation of the CSW algorithm factors A to produce

$$A = \underbrace{\begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}}_{\text{MLDI}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}}_{\text{MLDI}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{MRC}}.$$

The algorithm would then perform the MRC permutation characterized by the rightmost factor matrix, followed by the MLDI permutation characterized by the middle matrix, and finally the MLDI permutation characterized by the leftmost matrix.

3.4 History of My Implementation

My implementation of the CSW algorithm was developed in two distinct stages: first on a single-disk platform, and then on parallel disks.

3.4.1 Single-Disk Simulation

The original implementation was developed on and intended for a single-disk DEC Alpha 3000. I used Digital's asynchronous I/O library, which provides aio_read() and aio_write() functions (with an interface similar to standard UNIX read() and write()), along with a few supplemental routines.

In the single-disk environment, we simulated a parallel-disk system by letting a record's address in the file represent which simulated disk it was stored on.

I started by writing a quick prototype to make sure I understood the concept of permuting a file of records. After the prototype, I developed a program to perform MRC permutations when given a characteristic matrix in the correct form. In writing the MRC code, I learned how to permute a file's records and how to simulate striped I/O operations using the aio library routines.

Next I wrote a similar program for MLD permutations. Independent parallel I/O operations presented an unexpected problem—the aio library limits the number of read or write calls that can be queued at the same time. After unsuccessfully trying to solve the problem using signal handlers, I redesigned both the MRC and the MLD programs to use multithreading. Digital provides a pthread library, with routines and structures to create and synchronize threads.

My modified programs create three threads, which read records, permute them in memory, and write them back to disk, respectively. We divide memory into four buffers: for reading records into, permuting from, permuting to, and writing from. By allowing these operations to proceed concurrently, we hope to minimize the time lost to disk-access latency. While the permute thread permutes the ith memoryload from the "permute from" buffer to the "permute to" buffer, the read thread is free to read the $(i+1)$st memoryload into the "read to" buffer, and the write thread can write the $(i-1)$st memoryload back to disk from the "write from" buffer.

We can view the algorithm's synchronization as a pair of double-buffered

producer/consumer relations. The read thread produces records in the "read to" buffer while the permute thread consumes records from the "permute from" buffer. When the read thread has filled its buffer and the permute thread has emptied its buffer, they swap and start again. At the same time, the permute thread produces records in the "permute to" buffer, and the write thread consumes from the "write from" buffer. When these threads have finished, they swap buffers and both start on their next memoryloads.

Once I finished the MLD program, I developed a modified version to perform MLDI permutations. This program required another routine, to determine which source blocks should be read in order to obtain a set of records that, when permuted, will map to M consecutive source addresses. Before starting each new memoryload, the read thread uses this routine to determine which blocks to read.

The last step in implementing the single-disk BMMC algorithm was to write a factoring routine, which follows the steps described in [CSW94] for factoring the BMMC permutation's characteristic matrix into a series of MRC and MLD matrices. I changed the order in which it combines matrices so that it produces one MRC permutation followed by a series of MLDIs rather than the series of MLDs followed by one MRC described in [CSW94]. As noted in Section 3.2, MLDI permutations use striped writes and independent reads, whereas MLD permutations use striped reads and independent reads. MLDI permutations are preferable to MLD permutations because striped writes permit parity information to be more easily maintained.

The completed BMMC program takes a file of records, a BMMC matrix, and a complement vector. It factors the matrix into component matrices and then uses the MRC and MLDI routines to perform each of these factor permutations on the records in turn.

3.4.2 Parallel-Disk Version

A few months after I completed the first version of my BMMC implementation, Dartmouth's Computer Science Department purchased adams, and we began converting the program for testing on our new environment.

Converting my program to use the parallel-disk system was straightforward once a Digital representative told us how to do *raw I/O* (bypassing the OSF file system and buffering) on our machine. There is a *character-special* file associated with each disk, and raw I/O calls are made to these files, with arguments that include the number of bytes requested and the offset from the beginning of the disk.

I developed a new set of procedures to convert parallel I/O requests to raw asynchronous read and write calls to character-special files. After I modified the MRC and MLDI procedures to make these requests in place of calls to the aio file routines and made a few changes to the interface, permute.c correctly performed BMMC permutations on our eight-disk system.

The new interface allows the user at run time to specify desired values for N, M, B, and D, and (when $D < 8$) to choose whether to stripe files across all or half of the disks. The final version also reports timing statistics for permutations, and can log the I/O activity.

Chapter 4

Testing

4.1 Goals and Predictions

In designing our test cases, we were interested in characterizing the performance of the DEC Alpha 2100's parallel-disk system. We varied the number of bytes transferred in each parallel read or write, the number of disks used, and the way we divided available disks into target and source portions. We also varied the total number of bytes stored on disk and the number of bytes allowed in memory.

4.1.1 Organization of Source and Target Portions

Our primary interest was with the layout of data on disk. As described in Section 2.3, we can split our disk space into either two portions of $D/2$ disks each or two portions of $S/2$ stripes each.

By storing the source and target portions on separate groups of disks, we hope to reduce disk-head movement. Multiple-pass algorithms alternate between source reads and target writes, so dividing each disk into source and target halves may involve a large amount of disk-head movement from one por-

tion to the other. Conversely, separate source and target disks should greatly reduce disk head movement. Since our MRC algorithm reads the N records in order (by their indices) and our MLDI algorithm writes the records in order, we expect that half of the I/O operations will be to adjacent blocks when the source and target portions are not on the same disks.

When the source and target portions reside on separate disks, we allow simultaneous disk reads and writes, since the parallel read operations and parallel write operations are not competing for access to the same disks. Ideally, the read, write, and permute threads would be completely overlapped, and none of the three would need to waste time waiting for the others. In practice, however, the write operations were generally somewhat slower than the reads.

On the other hand, spreading both the source and target portions over all D disks reduces the number of parallel I/O operations and their associated OS call overhead. Section 3.3 explains that a single-pass permutation requires at least $2N/BD$ parallel I/O operations, since N/B blocks are each being read once and written once. If we let D' indicate the *effective* number of disks, or the number of disks each parallel I/O operation accesses, we can say that each MRC or MLDI permutation actually requires $2N/BD'$ parallel I/O operations. When the source and target portions are located on separate disks, $D' = D/2$ and each MRC or MLDI permutation requires $4N/BD$ parallel I/O operations, as opposed to the $2N/BD$ operations required when the source and target portions are striped over all D disks.

The number of factor permutations produced by the BMMC algorithm,

$\left\lceil \frac{\text{rank } \gamma}{\lg(M/B)} \right\rceil + 2$, is not related to the number of disks used, and so separating the source and target portions by disk requires exactly twice as many parallel I/O operations as separating them by stripe.

4.1.2 Optimal Memory and Block Sizes

We were also interested in characterizing optimal sizes for memory buffers (M) and data blocks (B).

The value of B and the number of parallel I/O operations per pass are inversely related—doubling B halves the number of parallel I/Os. Therefore we might expect larger values for B to yield lower execution times than smaller values. On the other hand, because larger values for $\lg(M/B)$ tend to reduce the number of passes, larger block sizes might increase the execution times.

Since increasing M tends to reduce the number of passes and has no effect on the number of parallel I/O operations per pass, we expect larger values of M to yield faster times than smaller values. However, not all computation and I/O can be overlapped. The first memoryload must be read before the permute thread can start, and the last memoryload cannot be written until the permute thread has finished. The time spent on these operations is directly related to the memory size, so increasing M increases the costs associated with the first read and the last write.

4.2 Results

We take our results from tests we ran the BMMC permutation program described in Section 3.4.2. Times were calculated from standard UNIX getclock()

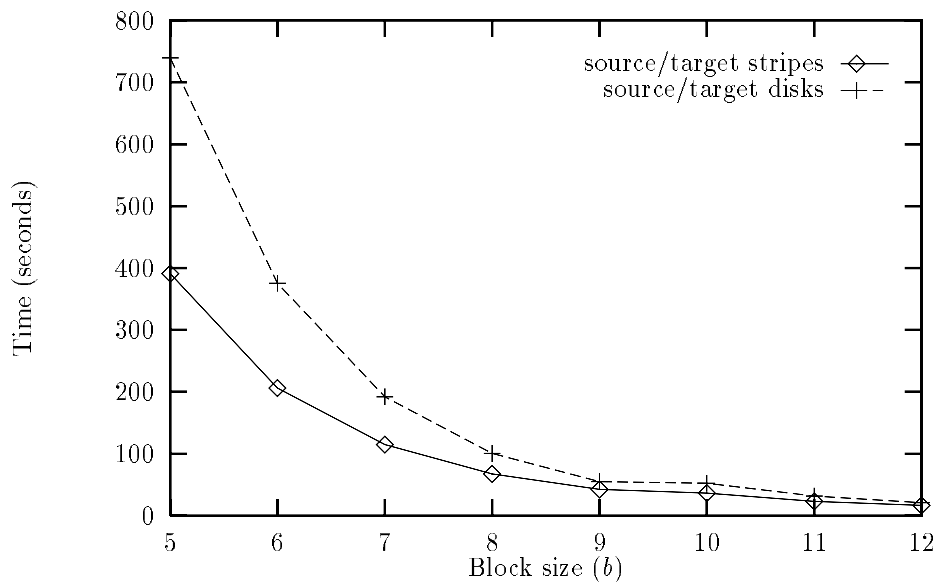


Figure 4.1: Effects of the organization of data on disks on permutation times. Times represent one pass in the permutation of a randomly generated BMMC matrix for $n = 20$, $m = 16$, and $D = 2$, and using 16-byte records striped across all available disks. Each of the test runs reported here made exactly two passes. The source and target portions are either both spread over both disks or each given their own disk.

calls, which we used to measure the elapsed time between the start of a randomly generated BMMC permutation's MRC factor permutation and the finish of its final MLDI factor permutation. The following graphs report these times, except where we specifically note that the times are normalized to the number of passes each BMMC permutation required.

For every combination tested, spreading the source and target portions over all the available disks was faster than separating them by disk. Figure 4.1 shows one such comparison. Our results clearly demonstrate that splitting the disks to create separate source and target portions is not an effective strategy.

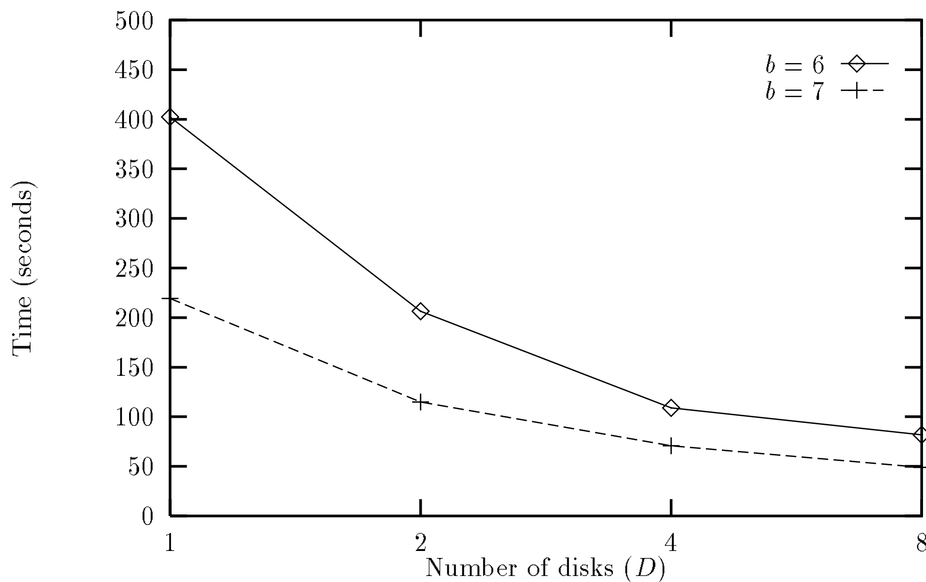


Figure 4.2: The I/O bottleneck. Times are for $n = 20$, $m = 16$, and either $b = 6$ or $b = 7$, with 16-byte records striped across all available disks.

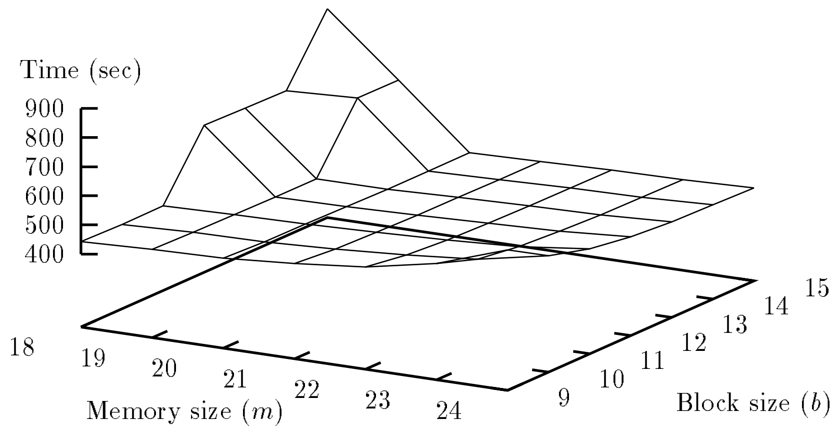
With values of D greater than 2, we encountered a bottleneck, shown in Figure 4.2, which we believe to be caused by the I/O system. Since our 8 data disks are chained on 3 SCSI buses, we believe that when we read or write to two disks on the same bus, the bus saturates and we cannot achieve the full I/O bandwidth of each disk.

We were surprised by the program's behavior when we ran it on larger numbers of small records. Under these conditions, the running time appeared to be determined by the computation time, rather than by the I/O time as we expected. The block size didn't affect the running time as directly as it had for smaller numbers of larger records (compare the effects of b shown in Figure 4.3 with those in Figure 4.1), and the running times actually tended to increase as

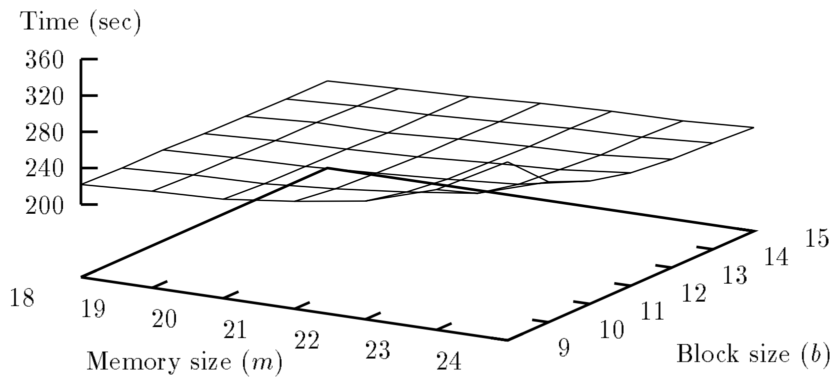
we increased memory. Figure 4.3 reports these results.

We can explain this behavior by assuming that our program's running time depends on the computation time when it is run on large numbers of small records. Increasing the number of passes increases the number of times target addresses must be calculated, which increases the running time. Decreasing the number of parallel I/O operations (by increasing the block size) has little affect on the running time. Except in the cases where the number of passes changes (compare Figures 4.3 and 4.4), increasing memory doesn't affect either the number of parallel I/O operations or the speed of computation. However, one full memoryload must be read before computations can begin, and one full memoryload must be written after computations have finished. The time required to transfer these memoryloads increases with the size of memory.

According to our results, the bottom line is that the fastest running times occured when the smallest number of passes were required, and m was as small as possible and b was as large as possible for that minimum number of passes.



Permutation times



Times normalized to the number of passes

Figure 4.3: Permuting a large number of small records. Times are for $n = 25$ and $D = 8$, with 1-byte records spread across all available disks. (We did not run these tests on data sizes larger than $n = 25$ due to time constraints.)

Block size (b) \ Memory size (m)	18	19	20	21	22	23	24
9	1	1	1	1	1	1	1
10	1	1	1	1	1	1	1
11	1	1	1	1	1	1	1
12	2	1	1	1	1	1	1
13	2	1	1	1	1	1	1
14	2	2	1	1	1	1	1
15	3	2	1	1	1	1	1

Figure 4.4: Number of passes required for a randomly generated BMMC matrix with $n = 25$.

Chapter 5

Conclusion

In this thesis, we presented the results obtained from running a multiple-pass algorithm on a parallel-disk system, the DEC Alpha 2100. We determined that striping information across all available disks was faster than splitting the disks into a source half and a destination half in every test case run, and we saw the fastest running times for our program when the memory size and block size were just large enough and just small enough, respectively, to minimize the number of passes.

We also developed a successful implementation of the CSW algorithm for BMMC permutations.

Bibliography

[Cor93] Thomas H. Cormen. Fast permuting on disk arrays. *Journal of Parallel and Distributed Computing*, 17:41–57, 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*.

[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.