Structured Permuting in Place on Parallel Disk Systems

Leonard F. Wisniewski*
Department of Computer Science
Dartmouth College

Abstract

The ability to perform permutations of large data sets in place reduces the amount of necessary available disk storage. The simplest way to perform a permutation often is to read the records of a data set from a source portion of data storage, permute them in memory, and write them to a separate target portion of the same size. It can be quite expensive, however, to provide disk storage that is twice the size of very large data sets. Permuting in place reduces the expense by using only a small amount of extra disk storage beyond the size of the data set.

This paper features in-place algorithms for commonly used structured permutations. We have developed an asymptotically optimal algorithm for performing BMMC (bit-matrix-multiply/complement) permutations in place that requires at most $\frac{2N}{BD}\left(2\left\lceil\frac{\operatorname{rank}\gamma}{\lg(M/B)}\right\rceil+\frac{7}{2}\right)$ parallel disk accesses, as long as $M \geq 2BD$, where N is the number of records in the data set, M is the number of records that can fit in memory, D is the number of disks, B is the number of records in a block, and γ is the lower left $\lg(N/B) \times \lg B$ submatrix of the characteristic matrix for the permutation. This algorithm uses $N+M$ records of disk storage and requires only a constant factor more parallel disk accesses and insignificant additional computation than a previously published asymptotically optimal algorithm that uses $2N$ records of disk storage.

We also give algorithms to perform mesh and torus permutations on a d-dimensional mesh. The in-place algorithm for mesh permutations requires at most $3\lceil N/BD\rceil$ parallel I/Os and the in-place algorithm for torus permutations uses at most $4dN/BD$ parallel I/Os. The algorithms for mesh and torus permutations require no extra disk space as long as the memory size M is at least $3BD$. The torus algorithm improves upon the previous best algorithm in terms of both time and space.

*Supported in part by a Dartmouth Fellowship and in part by the National Science Foundation under Grant CCR-9308667.

1 Introduction

Despite the decreasing cost per megabyte of providing high-bandwidth parallel disk systems with large storage capacities, the total cost of very large disk systems remains high. Modern parallel disk systems have gigabyte and terabyte capacities, and in the near future, petabyte- and exabyte-capacity disk storage systems will appear. At \$0.50 per megabyte of disk storage, a 1 terabyte system would cost \$500,000. This cost does not even reflect the higher cost per megabyte of increasing the bandwidth with parallel disk arrays.

Out-of-core permutations are data-movement operations which can use a large amount of costly extra storage if not performed conservatively. Recent theoretical work focuses on the I/O complexity of various out-of-core permutations. Many of the permutation algorithms perform a number of passes over the data. Each pass reads the source data set stored in one portion of disk storage, permutes it in memory, and writes it to a separate target portion of disk storage. Unfortunately, permuting in this manner requires that the capacity of the disk system be twice the size of the data set, or that we limit the data set to be half the available storage capacity. Permuting in place with only a small amount of extra disk storage greatly reduces this additional expense or limitation.

We categorize a permutation algorithm as *in-place* if the algorithm performs the permutation with additional disk storage that is asymptotically proportional to the size of memory. That is, we permute a data set with N *records* in place if we use a total of $N + O(M)$ records of disk storage, where M is the number of records that can fit in main memory. Algorithms that use separate source and target portions use $2N$ records of disk space, whereas the algorithms presented in this paper use no more than $N + M$ records.

In addition to reducing the amount of disk space required to perform the permutation, we would like the algorithm to be efficient in terms of I/O complexity and computation. We prefer that the I/O complexity of an in-place algorithm not exceed that of an algorithm that uses separate source and target portions of disk storage by more than a constant factor. We also do not want the additional time to determine which disk blocks to read and write to be significant.

In this paper, we examine in-place algorithms for several types of commonly used structured permutations that have been shown to require an asymptotically lower number of parallel disk accesses than general permuting. We show how to perform these permutations in place with no more than M records of extra disk space. More specifically, the results of this paper include the following:

1. An asymptotically optimal algorithm to perform an arbitrary BMMC (bit-matrix-multiply/complement) permutation in place which takes at most $\frac{2N}{BD}\left(2\left\lceil\frac{\operatorname{rank}\gamma}{\lg(M/B)}\right\rceil+\frac{7}{2}\right)$ parallel disk accesses and $N+M$ records of disk space as long as $M \geq 2BD$, where D is the number of disks, B is the number of records in a block, and γ is the lower left $\lg(N/B) \times \lg B$ submatrix of the nonsingular characteristic matrix for the permutation.

2. An in-place algorithm to perform mesh permutations which requires at most $3\lceil N/BD\rceil$ parallel disk accesses and no extra disk space.

3. An in-place algorithm to perform d-dimensional torus permutations which uses at most $4dN/BD$ parallel disk accesses, $3BD$ records of memory, and no extra disk space, yielding an asymptotic improvement upon the I/O-complexity bound of the previous best algorithm.

4. In-place algorithms to perform two subclasses of BMMC permutations which take only one pass and require no more than a memoryload of extra disk space.

Outline

The remainder of this paper is organized as follows. Section 2 provides the I/O complexity model and a description of previous work on out-of-core permuting and in-place permuting. In Section 3, we present the BMMC algorithm of [CSW94], adapt it to be performed in place, and present algorithms to perform in place the one-pass permutation subclasses used by the BMMC algorithm. Section 4 contains in-place algorithms to perform mesh permutations in one pass over the data and to perform torus permutations in asymptotically fewer parallel disk accesses than the previous best algorithm. Finally, Section 5 contains some concluding remarks.

The algorithms in Sections 3 and 4 are on-line in the sense that they take little computation time and memory. (They do, however, require permutations to be performed in memory, and various architectures may differ in how efficiently they do so.) The data structures are vectors of length $\lg N$ or matrices of size at most $\lg N \times \lg N$. Even serial algorithms for the harder computations take time polynomial in $\lg N$, in fact $O(\lg^3 N)$. When appropriate, we show that any additional computation for permuting in place is insignificant.

2 Model and previous results

We use the parallel-disk model first proposed by Vitter and Shriver [VS90, VS94], who also gave asymptotically optimal algorithms for several problems including sorting and general permutations. In the Vitter-Shriver model, 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 organized in *blocks* of B records each. 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. 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 parallel-disk model allows *independent I/O*, which accesses blocks that may be at any locations on their respective disks in a single parallel I/O. Independent access is more general than *striped I/O*, which has the restriction that the blocks accessed in a given operation must be at the same location on each disk.

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 such as those in [AP94, Arg95, CGG+95, Cor92, Cor93, CSW94, GTVV93, NV93, VS94] and this paper.

There are two restrictions implied by the Vitter-Shriver model. 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 Vitter-Shriver model lays out data on a parallel disk system as shown in Figure 1. A *stripe* consists of the D blocks at the same location on all D disks. 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. An algorithm makes a *pass* over the data if it reads and writes each record exactly once, for a total of $2N/BD$ parallel I/Os. Vitter and Shriver showed an upper bound of $\Theta\left(\min\left(\frac{N}{D}, \frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)\right)$ parallel I/Os for general permutations, that is, for arbitrary mappings $\pi : \{0, 1, \ldots, N-1\} \xrightarrow{1\text{-}1} \{0, 1, \ldots, N-1\}$. The first term comes into play when the block size B is small, and the second term is the sorting bound $\Theta\left(\frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)$, which was shown by Vitter and Shriver for randomized sorting and by Nodine and Vitter [NV93] and by Arge [Arg95] for deterministic sorting. These bounds are asymptotically tight, for they match 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 Vitter-Shriver model.

Specific classes of structured permutations sometimes require fewer parallel I/Os than general permutations. Vitter and Shriver showed how to transpose an $R \times S$ matrix ($N = RS$) with only $\Theta\left(\frac{N}{BD}\left(1+\frac{\lg\min(B,R,S,N/B)}{\lg(M/B)}\right)\right)$ parallel I/Os. Subsequently, Cormen [Cor93] and Cormen, Sundquist, and Wisniewski [CSW94] also developed algorithms to perform several classes of bit-defined permutations using fewer parallel I/Os than general permutations. The bit-defined classes include matrix transposition with power-of-2 dimensions as a special case. Cormen [Cor93] also shows how to efficiently perform permutations on data with a mesh layout.

Several in-place algorithms have been developed with certain limitations. Fich, Munro and Poblete [FMP95] provide in-place algorithms to perform general permutations in memory. These algorithms, however, do not apply to per-

	$\mathcal{D}_0$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3$		$\mathcal{D}_4$		$\mathcal{D}_5$		$\mathcal{D}_6$		$\mathcal{D}_7$	
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 1: 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.

muting data sets that exceed the size of memory. Vitter and Shriver did not design their out-of-core algorithm for general permuting to be performed in place. Furthermore, the constants implicit in the upper bound for this algorithm are rather high compared to the constants in the worst cases for the structured permutation algorithms mentioned above [VV95].

The next two sections present out-of-core in-place algorithms to perform several commonly used structured permutations. We supply exact constants for the I/O complexity of both the previous best algorithms and the in-place algorithms. With exact constants, we can examine the tradeoffs between time and space when deciding whether to use an in-place algorithm. The in-place algorithms require at most $N + M$ records of disk space, a significant savings over the $2N$ records of disk space required by the previous algorithms when $N >> M$.

3 Performing BMMC permutations in place

In this section, we give a brief overview of the algorithm in [CSW94] to perform BMMC permutations on parallel disk systems. We shall then adapt that algorithm to be performed in place by showing how to perform two subclasses of BMMC permutations in place. The algorithms to perform each of these subclasses use no more than $3N/BD$ parallel I/Os. In fact, in most cases, they use only $2N/BD$ parallel I/Os. In all cases, they use at most an extra memoryload of disk space.

For this section, we use the following notation extensively:

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

We shall assume that b, d, m, and n are nonnegative integers, which implies that B, D, M, and N are exact powers of 2. The two restrictions on the Vitter-Shriver model imply that $b + d \le m < n$. We indicate the *address*, or *index*, of a record as an n-bit vector x with the least significant bit first: $x = (x_0, x_1, \ldots, x_{n-1})$. As Figure 2 shows, the offset within the block is given by the least significant b bits $x_0, x_1, \ldots, x_{b-1}$, the disk number by the next d bits $x_b, x_{b+1}, \ldots, x_{b+d-1}$, and the stripe number by the $s = n - (b + d)$ most significant bits $x_{b+d}, x_{b+d+1}, \ldots, x_{n-1}$. If we number the blocks within each memoryload from 0 to $M/B - 1$, then the $m - b$ bits $x_b, x_{b+1}, \ldots, x_{m-1}$ indicate in which *relative block number* a record resides. We also partition the data into N/M disjoint sets of M consecutive records which we call *memoryloads.* The most significant $n - m$ bits of the source address designate in which *memoryload number* a record resides.

We shall use several other notational conventions in this section. Matrix row and column numbers are indexed from 0 starting from the upper left. Vectors are indexed from 0, too. We index rows and columns by sets to indicate submatrices, using ".." notation to indicate sets of contiguous numbers. We denote an identity matrix by I and a matrix whose entries are all 0s by 0; the dimensions of such matrices will be clear from their contexts. All matrix and vector elements are drawn from $\{0, 1\}$, and all matrix and vector arithmetic is over $GF(2)$[1]. When convenient, we interpret bit vectors as the integers they represent in binary. Vectors are treated as 1-column matrices in context.

BMMC permutations

The most general class of bit-defined permutations is *bit-matrix-multiply/complement*, or *BMMC*, permutations.[2] For a BMMC permutation, we have an $n \times n$ *characteristic matrix* $A = (a_{ij})$ whose entries are drawn from $\{0, 1\}$ and is nonsingular (i.e., invertible) over $GF(2)$, and we have a *complement vector* $c = (c_0, c_1, \ldots, c_{n-1})$ of length n. Treating a *source address* x as an n-bit vector, we perform matrix-vector multiplication over $GF(2)$ and then form the corresponding n-bit *target address* y by complementing some subset of the resulting bits: $y = A\,x \oplus c$.

BMMC permutations include the subclass of BPC (bit-permute/complement) permutations, which have characteristic matrices with exactly one 1 in each row and in each column. BPC permutations include many common permutations such as matrix transposition, bit-reversal permutations (used in performing FFTs), vector-reversal permutations, hypercube permutations, and matrix reblocking. BMMC permutations also include non-BPC permutations such as the standard binary-reflected Gray code and its inverse.

We shall generally focus on the matrix-multiplication portion of BMMC permutations rather than on the complement vector. The permutation π_A characterized by a matrix A is the permutation for which $\pi_A(x) = A\,x$ for all source addresses x.

The following lemma from [CSW94] shows the equivalence of multiplying characteristic matrices and composing permutations when the complement vectors are zero. For permutations π_Y and π_Z, the *composition* $\pi_Z \circ \pi_Y$ is defined by $(\pi_Z \circ \pi_Y)(x) = \pi_Z(\pi_Y(x))$ for all x in the domain of π_Y.

Lemma 1 *Let Z and Y be nonsingular $n \times n$ matrices and let π_Z and π_Y be the permutations characterized by Z and Y, respectively. Then the matrix product $Z\,Y$ characterizes the composition $\pi_Z \circ \pi_Y$.* ■

[1] 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.

[2] Edelman, Heller, and Johnsson [EHJ94] call BMMC permutations *affine transformations* or, if there is no complementing, *linear transformations*.

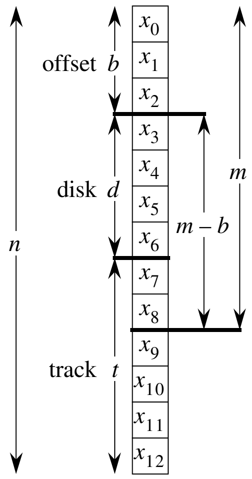


Figure 2: Parsing the address $x = (x_0, x_1, \ldots, x_{n-1})$ of a record on a parallel disk system. Here, $n = 13$, $b = 3$, $d = 4$, $m = 8$, and $s = 6$. The least significant b bits contain the offset of a record within its block, the next d bits contain the disk number, and the most significant s bits contain the stripe number. The most significant $n - m$ bits form the record's memoryload number, and bits $b, b+1, \ldots, m-1$ form the relative block number.

After the factorization of a characteristic matrix A into the product of several nonsingular matrices, each factor characterizes a BMMC permutation. The following corollary from [CSW94] describes the order in which we perform these permutations to effect the permutation characterized by A.

Corollary 2 *Let the $n \times n$ characteristic matrix A be factored as $A = A^{(k)} A^{(k-1)} A^{(k-2)} \cdots A^{(1)}$, where each factor $A^{(i)}$ is a nonsingular $n \times n$ matrix. 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.* ■

BMMC algorithm

The BMMC algorithm presented in [CSW94] uses a matrix decomposition/composition method which factors the characteristic matrix A of the BMMC permutation into at most $\left\lceil \frac{\operatorname{rank} \gamma}{\lg(M/B)} \right\rceil + 2$ matrix factors, where γ is the submatrix $A_{b..n-1,0..b-1}$, i.e., the lower left $(n-b) \times b$ submatrix of A. Each factor characterizes a permutation that can be performed in one pass over the data. The algorithm first uses linear-algebraic techniques to decompose the characteristic matrix into at most $2\left\lceil \frac{\operatorname{rank} \gamma}{\lg(M/B)} \right\rceil + 5$ factors, each of which characterizes a one-pass permutation. After the factorization, the algorithm uses composition properties among the classes of one-pass permutations to combine pairs of one-pass permutations into single one-pass permutations. The compositions reduce the number of one-pass factors to at most $\left\lceil \frac{\operatorname{rank} \gamma}{\lg(M/B)} \right\rceil + 2$. Since one pass over the data requires $\frac{2N}{BD}$ parallel I/Os (i.e., each record is read and written once), the I/O complexity of the BMMC algorithm is $\frac{2N}{BD}\left(\left\lceil \frac{\operatorname{rank} \gamma}{\lg(M/B)} \right\rceil + 2\right)$ parallel I/Os. This upper bound on the number of parallel I/Os asymptotically matches the lower bound also shown in [CSW94] and is, in fact, almost equal to the best known exact lower bound.

All the one-pass permutations in the factorization of the matrix A have characteristic matrix forms included in the the BMMC subclasses of MRC (memory-rearrangement/complement) and MLD (memoryload-dispersal) permutations. The algorithms to perform MRC and MLD permutations in [Cor93] and [CSW94], respectively, read the data from a source portion of N records of disk storage, permute the data in memory, and write the data to a separate target portion of N records of disk storage.

MRC permutations are BMMC permutations with the additional restrictions that both the leading $m \times m$ and trailing $(n-m) \times (n-m)$ submatrices of the characteristic matrix are nonsingular, the upper right $m \times (n-m)$ submatrix can contain any 0-1 values at all, and the lower left $(n-m) \times m$ submatrix is all 0:

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

Cormen [Cor93] shows that any MRC permutation requires only one pass of N/BD parallel reads and N/BD parallel writes. We partition the N records into N/M disjoint memoryloads of M consecutive records each. Each memoryload consists of M/BD consecutive stripes in which all addresses have the same value in the most significant $n-m$ bits. Any MRC permutation can be performed by reading a memoryload, permuting its records in memory, and writing them out to a (possibly different) memoryload number.

MLD permutations are BMMC permutations with a characteristic matrix that is nonsingular and of the form

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

subject to the *kernel condition.* We define the kernel of a matrix ψ (also known as the nullspace) as $\ker \psi = \{x : \psi x = 0\}$.

The kernel condition requires that $\ker \lambda \subseteq \ker \mu$ or, equivalently, $\lambda x = 0$ implies $\mu x = 0$. We can perform any MLD permutation in one pass by reading each source memoryload, permuting its records in memory, and writing these records out to M/BD target blocks on each disk. Although the source blocks read from each memoryload must come from M/BD consecutive stripes, the target blocks written may go to any locations at all, as long as M/BD full target blocks are written to each disk. That is, MLD permutations use striped reads and independent writes.

To perform a BMMC permutation in place, we would just need to develop algorithms to perform MRC and MLD permutations in place. Unfortunately, we do not know how to perform an arbitrary MLD permutation in $O(N/BD)$ parallel I/Os using only $O(M)$ records of extra disk space. The dispersal of blocks from each source memoryload in the MLD algorithm to as many as M/B different target memoryloads does not allow us to easily reuse the vacated disk space of the source memoryload. The MLD permutations produced by the decomposition phase of the factoring method, however, belong to the even more restricted subclass of *erasure permutations* which we shall define later. Fortunately, we can easily perform erasure permutations with no extra disk space.

The rest of this section shows how to perform MRC permutations and erasure permutations in place using $O(N/BD)$ parallel I/Os. After the matrix-decomposition phase of the BMMC algorithm of [CSW94], the factorization is

$$A = F\, E_g^{-1}\, S_g^{-1}\, E_{g-1}^{-1}\, S_{g-1}^{-1} \cdots E_1^{-1}\, S_1^{-1} R^{-1} T^{-1} \;, \qquad (1)$$

where the factors F, S_i^{-1}, R^{-1} and T^{-1} characterize MRC permutations, the factors E_i^{-1} characterize erasure permutations, and $g \leq \left\lceil \frac{\operatorname{rank}\gamma}{\lg(M/B)} \right\rceil + 1$. Thus, if we show how to perform MRC and erasure permutations in place using $O(N/BD)$ parallel I/Os, we would have an asymptotically optimal algorithm that uses $O\left(\frac{N}{BD}\left(\left\lceil \frac{\operatorname{rank}\gamma}{\lg(M/B)} \right\rceil + 1\right)\right)$ parallel I/Os. Each of the characteristic matrices T^{-1}, R^{-1}, S_i^{-1}, and E_i^{-1} have special matrix forms which we shall discuss later in more detail.

Performing MRC permutations in place

Here we show how to perform any MRC permutation in place using at most $3N/BD$ parallel I/Os and only M records of extra disk space. Our in-place MRC permutation algorithm only requires $2N/BD$ parallel I/Os when we use half of memory in place of the extra disk space or when the characteristic matrix is of a particular form, which we shall see later.

We examine more closely the restrictions on the characteristic matrix form to develop an algorithm to perform MRC permutations in place. The following equation calculates a target address y from an MRC characteristic matrix and a source address x:

$$\left[\begin{array}{c} y_{0..m-1} \\ \hline y_{m..n-1} \end{array}\right] = \begin{array}{c} \begin{array}{cc} m & n-m \end{array} \\ \left[\begin{array}{c|c} \alpha & \beta \\ \hline 0 & \delta \end{array}\right] \end{array} \left[\begin{array}{c} x_{0..m-1} \\ \hline x_{m..n-1} \end{array}\right] \begin{array}{l} m \\ n-m \end{array} .$$

Since the lower left $(n-m)\times m$ submatrix of the characteristic matrix is 0, the target memoryload number is $y_{m..n-1} = \delta x_{m..n-1}$. Moreover, every record in a given source memoryload number moves to the same target memoryload. Thus, the submatrix δ defines a permutation on the memoryload numbers. Because any permutation is the union of disjoint cycles, we can represent the memoryload numbers as a set of s disjoint cycles $a = (a^{(0)}, a^{(1)}, \ldots, a^{(s-1)})$, for some s in the range $1 \leq s \leq N/M$.

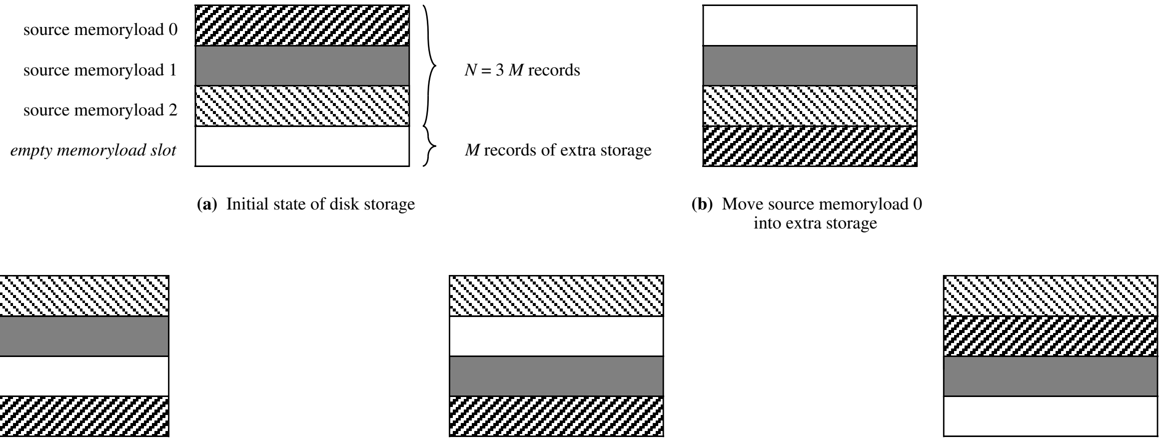
We perform MRC permutations in place by rotating each of the disjoint cycles once as Figure 3 shows. We rotate each disjoint cycle $a^{(i)} = (a_0^{(i)}, a_1^{(i)}, \ldots, a_{k-1}^{(i)})$ of k memoryload numbers by first moving memoryload $a_0^{(i)}$ into M records of extra disk space, thereby creating an empty memoryload slot where memoryload $a_0^{(i)}$ initially resided on disk. We then fill the empty memoryload slot by reading memoryload $a_{k-1}^{(i)}$ into memory, permuting within the memoryload according to the upper m rows of the characteristic matrix, and writing it to the empty memoryload slot. At this point, there is a new empty memoryload slot where memoryload $a_{k-1}^{(i)}$ originally resided. By repeatedly reading, permuting, and writing to the empty memoryload slot, we rotate each memoryload in the cycle until we have written all but one memoryload to its correct target location. Finally, we read memoryload $a_0^{(i)}$ back into memory from the extra disk space, permute it, and write it to the remaining empty memoryload slot.

The only unresolved problem is to find the disjoint cycles without rotating any disjoint cycle more than once. Fich et al. [FMP95] provide several algorithms to perform an arbitrary permutation in place when the entire data set fits into memory. The following theorem from [FMP95] reflects the tradeoff between time and additional space when permuting an array of length p in memory when an extra q bits of storage are available.

Theorem 3 *In the worst case, permuting an array of length p, given the permutation, can be done in $O(p^2/q)$ time and $q + O(\log p)$ bits of auxiliary space (consisting of a bit vector of length q plus a constant number of pointers) for $q \leq p$.* ■

In this context, we show how to perform a permutation on the p memoryload numbers, where $p = N/M$. We can adapt the simplest algorithm from [FMP95], which uses $q = N/M$ bits of extra space to rotate each memoryload exactly once. With current disk prices approximately 100 times less per megabyte than memory prices, current parallel disk systems do not typically have a capacity of more than 1000 memoryloads of data storage. It is reasonable to assume, therefore, that we can fit the extra q bits in memory. By Theorem 3, the additional computation time is $O(N/M)$, which is a constant number of operations per memoryload. Since we can easily generate the inverse permutation from the characteristic matrix, we can also adapt the more complex algorithms in [FMP95] to rotate the memoryloads when $N >> M$, using $O(\log(N/M))$ extra bits of storage and $O\left(\frac{N}{M}\log(N/M)\right)$ time. For simplicity of presentation, we adapt the algorithm that uses N/M extra bits of storage.

Our algorithm maintains a *rotate bit* for each memoryload number to indicate whether that memoryload has moved to its target location. In consecutive order, we examine the rotate bit for each memoryload number $l \in$



(a) Initial state of disk storage

(b) Move source memoryload 0 into extra storage

(c) Rotate source memoryload 2

(d) Rotate source memoryload 1

(e) Final state of disk storage

Figure 3: Rotating the 3-cycle of memoryload numbers 0, 1, and 2, where $N = 3M$ and the cycle is $(0, 1, 2)$. We move each memoryload to an empty slot by reading it into memory, permuting it, then writing it to the empty slot. We create the initial empty slot by reading source memoryload number 0 and writing it into the extra M records of disk space.

$\{0, 1, \ldots, N/M - 1\}$. If the rotate bit is set, the disjoint cycle with memoryload number l has already been rotated. Since we examine the rotate bits of the memoryload numbers from lowest to highest, for each disjoint cycle, we always examine the rotate bit of the lowest memoryload number in the cycle first. That is, if the rotate bit is not set, the cycle containing memoryload number l has not been rotated. We proceed to rotate the memoryload numbers in this cycle by treating memoryload number l as memoryload number $a_0^{(i)}$ and setting the rotate bit of each memoryload number in the cycle. Thus, the constant amount of extra computation time per memoryload is to check and set the rotate bits.

For each disjoint cycle $a^{(i)}$, we read and write each memoryload once, except for memoryload $a_0^{(i)}$, which we read and write an extra time to empty a memoryload slot. Thus, if $s < N/M$, the MRC permutation requires $2(N + sM)/BD$ parallel I/Os to perform in place, i.e., a little more than one pass over the data. In the worst case, an MRC permutation can have $N/2M$ 2-cycles of memoryload numbers, resulting in a total of $3N/BD$ parallel I/Os.

If $s = N/M$, then the trailing $(n-m)\times(n-m)$ submatrix is the identity matrix, i.e., a source memoryload number maps to the same target memoryload number. In this case, we read each memoryload, permute it in memory, then write it back to its original location on disk. No extra parallel I/Os are necessary to create an empty memoryload slot. This special case requires only $2N/BD$ parallel I/Os.

If $BD \leq M/2$, we have the option of reducing the amount of available memory to $M' = M/2$ records and storing memoryload $a_0^{(i)}$ in half of memory to avoid the expense of reading and writing it an extra time. Consequently, we must reduce the size of memory to M' for the entire BMMC factoring. Thus, the $\lg(M/B)$ denominator in the upper bound may decrease by one, i.e., the number of passes may increase.

Performing erasure permutations in place

We now show how to perform the MLD subclass of erasure permutations in place. Erasure permutations have nonsingular characteristic matrices of the form

$$E^{-1} = \begin{array}{c} \begin{array}{ccc} b & m-b & n-m \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & I & 0 \\ \hline 0 & \phi & I \end{array}\right] \end{array} \begin{array}{l} \\ b \\ m-b \\ n-m \end{array} ,$$

where ϕ is arbitrary. The key parts of the characteristic matrix to observe are the lower left $(n-b)\times b$ submatrix (which is all 0) and the trailing $(n-b)\times(n-b)$ submatrix. The most significant $n - b$ bits of a source address, $x_{b..n-1}$, represent the *block number* for each source block. Since the lower left $(n-b)\times b$ submatrix is zero, the trailing $(n-b)\times(n-b)$ submatrix must be nonsingular. The least significant b bits of a source address do not come into play when determining the target block number $y_{b..n-1}$ to which a record maps. Thus, all the records in a source block map to the same target block. Therefore, the trailing $(n-b)\times(n-b)$ submatrix induces a cycle of blocks just as the trailing $(n-m)\times(n-m)$ submatrix of the characteristic matrix for MRC permutations defined a cycle of memoryloads.

We cannot rotate blocks using the method of reading full source memoryloads and writing them to full target memoryloads as in the in-place algorithm for MRC permutations. Since memoryloads are evenly distributed across the disks, that method guaranteed full utilization of available disk bandwidth when performing MRC permutations. For a permutation of block numbers, there exist nonsingular trailing $(n-b)\times(n-b)$ submatrices for which all the blocks of a full source memoryload do not map evenly across the disks.

The trailing $(n-b)\times(n-b)$ submatrix of the matrix E^{-1} has two properties that allow us to use a simple algorithm for performing erasure permutations in place:

1. The middle $m - b$ rows of the submatrix define an identity mapping on the relative block numbers.

2. The trailing $(n-b)\times(n-b)$ submatrix is self-invertible. Moreover, $E^{-1} = E$, i.e., the entire erasure matrix is self-invertible.

Property 1 restricts the mapping of block numbers such that each source block moves to a target block with the same relative block number, possibly in a different memoryload. Property 2 implies that there are $N/2B$ disjoint 2-cycles of blocks. That is, the mapping swaps blocks.

We find the appropriate blocks to swap using the following sequential pseudocode (which is easily parallelized):

```
1  for k ← 0 to N/M − 1 do
2      for j ← 0 to M/B − 1 do
3          k' ← φj ⊕ k
4          if k' > k then
5              swap blocks with relative block number j
               in memoryloads k and k'
```

Lines 4 compares the memoryload number of each block to the memoryload number of its swap partner and line 5 performs a swap if the current block has the lower memoryload number in its pair.

To guarantee the utilization of all the disks on each parallel I/O, we perform $N/2M$ stages of M/B swaps. During each stage, we swap the next pair of blocks for each relative block number. By reading and writing two blocks with each relative block number, we evenly distribute the I/Os across the disks.

We swap each block exactly once, reading and writing each record one time. Thus, we perform any erasure permutation in place using exactly $2N/BD$ parallel I/Os. As long as $2BD \leq M$, our algorithm for erasure permutations does not require any extra disk storage for temporarily storing data.

The algorithm only requires an additive constant amount of extra computation time per block. For each block, we test whether it has already been swapped. Since we perform the mapping of each source block number in line 3 anyway, the only additional computation time is to make the comparison in line 4.

Determining constants for the in-place BMMC algorithm

We now use the bounds on the in-place MRC and erasure permutation algorithms to determine exact constants for the in-place BMMC algorithm. Each of the matrices in equation (1) belong to subclasses of BMMC permutations with special matrix forms. The matrices T^{-1}, R^{-1}, and S_i^{-1} have the trailer, reducer, and swapper matrix forms, respectively, defined in [CSW94]. Figure 4 shows these matrix forms. Each of these matrix forms has an inverse of the same form.

Since the factors S_1^{-1}, R^{-1}, and T^{-1} from equation (1) all characterize MRC permutations, by the composition property shown in [CSW94], their product $S_1^{-1}R^{-1}T^{-1}$ also characterizes an MRC permutation. Furthermore, since the matrices S_1^{-1}, R^{-1}, and T^{-1} have trailing $(n-m)\times(n-m)$ identity submatrices and a lower left $(n-m)\times m$ submatrix that is all 0, so does their product. Thus, the product $S_1^{-1}R^{-1}T^{-1}$ requires only $2N/BD$ parallel I/Os to perform.

Our algorithm performs any BMMC permutation in at most $\frac{2N}{BD}\left(2\left\lceil\frac{\operatorname{rank}\gamma}{\lg(M/B)}\right\rceil+\frac{7}{2}\right)$ parallel I/Os, which we see as follows. In equation (1), the factor F takes at most $3N/BD$ parallel I/Os to perform and each of the factors S_i^{-1}, E_i^{-1}, and $S_1^{-1}R^{-1}T^{-1}$ takes $2N/BD$ parallel I/Os. Since $g \leq \left\lceil\frac{\operatorname{rank}\gamma}{\lg(M/B)}\right\rceil+1$, the algorithm takes at most

$$\frac{3N}{BD}+\frac{2N}{BD}\left(2\left\lceil\frac{\operatorname{rank}\gamma}{\lg(M/B)}\right\rceil+2\right)=\frac{2N}{BD}\left(2\left\lceil\frac{\operatorname{rank}\gamma}{\lg(M/B)}\right\rceil+\frac{7}{2}\right)$$

parallel I/Os.

4 Performing mesh and torus permutations in place

In this section, we give in-place algorithms for performing mesh and torus permutations. The algorithm for mesh permutations requires at most $3\lceil N/BD\rceil$ parallel I/Os. The in-place algorithm for torus permutations is asymptotically faster in terms of parallel I/Os than the previous best algorithm, which uses $2N$ records of disk space. The algorithms in this section do not require that N, M, B, and D be powers of 2.

Some applications perform operations on data laid out in a d-dimensional mesh. Figure 5(a) shows the record layout on a 3×5 mesh. Let the dimensions of the mesh be $m=(m_0,m_1,\ldots,m_{d-1})$, with positions in dimension i indexed from 0 to m_i-1. We associate with each record a unique vector $p=(p_0,p_1,\ldots,p_{d-1})$ for which p_i indicates its location along dimension i.

Several useful permutations move the data along one or more dimensions of the mesh. The two types of permutations examined in this section handle the boundary conditions of the mesh layout differently. A *mesh permutation* is a partial permutation that shifts records by a constant number of locations in a specified direction along each dimension of the mesh, but does not move records that are shifted past any of the borders of the mesh. In a *torus permutation*, records wrap around dimension boundaries as necessary. Thus, any torus permutation is a full permutation which moves each record to a unique location in the mesh.

More formally, we can permute the data by adding an offset vector $o=(o_0,o_1,\ldots,o_{d-1})$ to the location of each record, where o_i is the offset in dimension i. We restrict the offset in each dimension such that $-m_i<o_i<m_i$ for mesh permutations and $0\leq o_i<m_i$ for torus permutations.

Mesh permutations

First we examine mesh permutations, in which a record in position p moves to position $\operatorname{mesh}(p,o)=(p_0+o_0,p_1+o_1,\ldots,p_{d-1}+o_{d-1})$. We store the records in row-major order, with 0-origin indexing in each dimension, and we define an index for each record of the mesh. The indexing function ι maps each grid position $p=(p_0,p_1,\ldots,p_{d-1})$ to a unique index in row-major order:

$$\iota(p,m)=\sum_{i=0}^{d-1}\left[\left(\prod_{j=i+1}^{d-1}m_j\right)p_i\right].$$

Figure 5(b) shows the one-dimensional row-major ordering of the records of the 3×5 mesh.

The following lemma from [Cor92] states that the difference between the source and target index of each record in row-major order is the same for every grid location mapped by a mesh permutation.

Lemma 4 *Let $p=(p_0,p_1,\ldots,p_{d-1})$ be any grid location mapped by a mesh permutation with offset $o=(o_0,o_1,\ldots,o_{d-1})$ on a d-dimensional grid with dimensions $m=(m_0,m_1,\ldots,m_{d-1})$. Then $\iota(\operatorname{mesh}(p,o),m)-\iota(p,m)=\iota(o,m)$.* ∎

$$T^{-1} = \begin{array}{cc} & \begin{array}{ccc} b & m-b & n-m \end{array} \\ & \left[\begin{array}{c|c|c} I & 0 & * \\ \hline 0 & I & * \\ \hline 0 & 0 & I \end{array}\right] \begin{array}{l} b \\ m-b \\ n-m \end{array} \end{array}$$

(**a**) Trailer matrix form

$$R^{-1} = \begin{array}{cc} & \begin{array}{ccc} b & m-b & n-m \end{array} \\ & \left[\begin{array}{c|c|c} * & * & 0 \\ \hline * & * & 0 \\ \hline 0 & 0 & I \end{array}\right] \begin{array}{l} b \\ m-b \\ n-m \end{array} \end{array}$$

(**b**) Reducer matrix form

$$S^{-1} = \begin{array}{cc} & \begin{array}{cc} m & n-m \end{array} \\ & \left[\begin{array}{c|c} \text{permutation} & 0 \\ \hline 0 & I \end{array}\right] \begin{array}{l} m \\ n-m \end{array} \end{array}$$

(**c**) Swapper matrix form

$$E^{-1} = \begin{array}{cc} & \begin{array}{ccc} b & m-b & n-m \end{array} \\ & \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & I & 0 \\ \hline 0 & * & I \end{array}\right] \begin{array}{l} b \\ m-b \\ n-m \end{array} \end{array}$$

(**d**) Erasure matrix form

Figure 4: Nonsingular column-addition matrix forms for special BMMC subclasses defined in [CSW94]. The factors T^{-1}, R^{-1}, S_i^{-1}, and E_i^{-1} in equation (1) belong to the trailer, reducer, swapper, and erasure subclasses, respectively. An asterisk (*) indicates that the submatrix may be nonzero. The reducer matrix form has a 1 in every location along the diagonal.

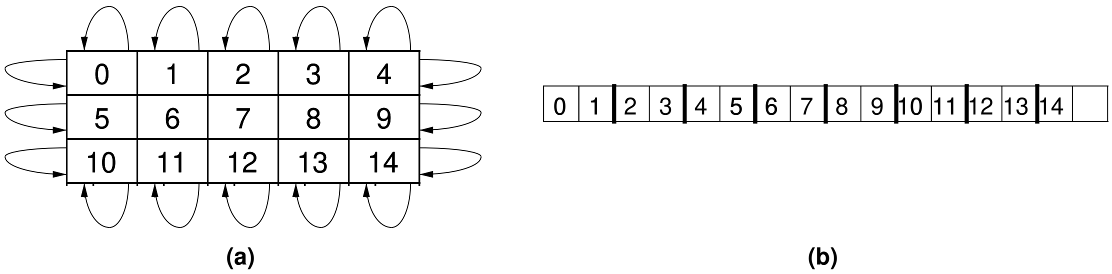


Figure 5: (**a**) Layout of data in a 3×5 mesh. The wraparound lines show the axes of rotation along the rows and columns for a torus permutation. (**b**) Row-major ordering of the records. We store as stripes the consecutive BD records starting at each record indexed by $x \equiv 0 \bmod BD$. In this example, $BD = 2$ and we denote stripe boundaries by thick lines.

This lemma implies that mesh permutations are *monotonic routes*, i.e., for any two grid locations p and q that can be mapped and $\iota(p, m) < \iota(q, m)$, then $\iota(\text{mesh}(p, o), m) < \iota(\text{mesh}(q, o), m)$. Figure 6 shows two mesh permutations, one with a positive offset and one with a negative offset.

Cormen uses an algorithm for performing monotonic routes to perform mesh permutations by reading source and target stripes in order, moving the source records to their correct target locations, and writing the target stripes. This algorithm requires at most $3 \lceil N/BD \rceil$ parallel I/Os and $2N$ records of disk space, where $N = m_0 m_1 \cdots m_{d-1}$ is the number of elements in the grid.

We derive a similar algorithm for performing mesh permutations in place. Since all the grid locations to be mapped have the same offset, we write the target stripes in order in an appropriate direction. We must be careful not to write into a location before moving the source record in that location. Thus, based on the sign of the offset, we process the target stripes in the appropriate direction. If $\iota(o, m)$ is positive as in Figure 6(a), we process the target stripes from highest (on the right) to lowest (on the left); otherwise, $\iota(o, m)$ is negative as in Figure 6(b), and so we process the target stripes from lowest to highest. This in-place algorithm also reads each stripe at most twice (if a target stripe holds data not being overwritten, we have to read it before permuting into it) and writes each stripe at most once for a maximum total of $3 \lceil N/BD \rceil$ parallel I/Os, requiring only N records of disk space and two stripes of memory ($2BD \leq M$).

Torus permutations

Our algorithm for torus permutations improves on the algorithm of [Cor92], which performs a torus permutation as a 2^d-monotonic route, i.e., a superposition of 2^d disjoint monotonic routes. Cormen's algorithm uses at most $(2^{d+1} + 1) \lceil N/BD \rceil$ parallel I/Os and requires $2N$ records of disk space and $2^d + 1$ stripes of memory. Our in-place algorithm for torus permutations rotates the data along one dimension of the mesh at a time, at most reading twice and writing twice each stripe of data per dimension. Thus, our algorithm uses at most $4dN/BD$ parallel I/Os. It requires only N records of disk space and 3 stripes of memory. Our in-place algorithm improves upon the previous best algorithm in terms of time, disk space, and memory space.

We now define a torus permutation more formally. A torus permutation is a rotation of the records along each dimension. The direction of movement in each dimension depends on both the offset vector o and each source position p. That is, for each dimension i, we redefine the offset vector o as an offset vector $o'(p, i) = (o'_0, o'_1, \ldots, o'_{d-1})$ with

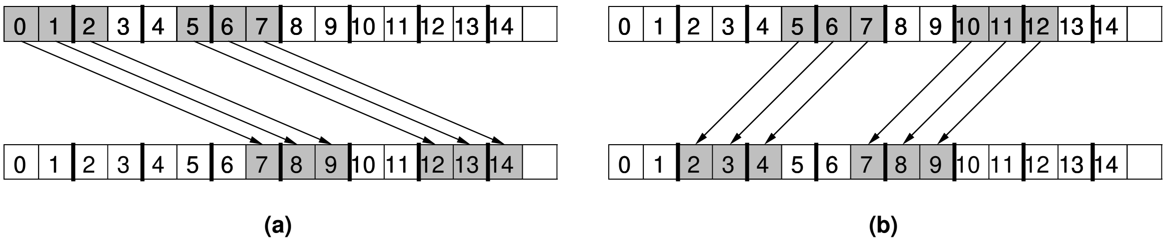


Figure 6: Mesh permutations with positive and negative offsets. On our 3×5 mesh, the indexing function applied to the offset for a mesh permutation may have either a positive or a negative sign. **(a)** A mesh permutation with only positive offsets ($o = (1, 2)$). **(b)** A mesh permutation with only negative offsets ($o = (-1, 2)$).

elements

$$o'_i = \begin{cases} o_i & \text{if } (p_i + o_i) \bmod m_i \geq p_i \ , \\ o_i - m_i & \text{if } (p_i + o_i) \bmod m_i < p_i \ . \end{cases}$$

We perform a torus permutation as a sequence of d permutations, each of which rotates along a single dimension. That is, in the ith permutation, for $i = 0, 1, \ldots, d-1$, we move a record at position $p = (p_0, p_1, \ldots, p_{d-1})$ to position $(p_0, p_1, \ldots, p_i + o'_i, \ldots, p_{d-1})$. If the offset is zero, we do not need to permute; otherwise the offset o'_i has two different values. If we write the target stripes consecutively in one direction, as in the mesh permutation algorithm, we may write over unmoved source records. Thus, we need a more clever algorithm to perform torus permutations in place.

Since there are two different offsets, we view the permutation along each dimension as a 2-monotonic route. To permute along dimension i, we partition the source records into two sets: one set of source records with offset o_i and another set of source records with offset $o_i - m_i$. The *majority set* is the set with greater cardinality and the other set is the *minority set*. (If the sets each have cardinality $N/2$, arbitrarily choose one as the majority set and the other as the minority set.) Since we partition the source records into two sets, each of which is the input to a monotonic route, a torus permutation along a single dimension is a 2-monotonic route. Unfortunately, we do not know how to perform in place an arbitrary k-monotonic route, for $k \geq 1$, and so we make some observations about this particular 2-monotonic route to perform it in place.

For each dimension, there may be multiple repetitions of a pattern of alternately α records of the majority set and β records of the minority set. In Figure 7, for example, there are three copies of the alternating majority/minority pattern with $\alpha = 3$ and $\beta = 2$. Without loss of generality, we limit ourselves to a single copy of this image, which has $N = \alpha + \beta$ records. In our analysis, we shall account for the stripes that contain records in more than one copy of the pattern.

Three facts provide the structure necessary for our in-place algorithm to perform a toroidal route along a single dimension using at most $4N/BD$ parallel I/Os and three stripes of memory (i.e., $3BD \leq M$). The first two facts are specific to this particular 2-monotonic route.

1. The records of the minority set initially reside in the β lowest or highest consecutively-numbered locations. The records of the majority set reside in the other $\alpha = N - \beta$ locations.

2. The records of the minority set move from the β highest (lowest) numbered locations to the β lowest (highest) numbered locations. Similarly, the records of the majority set move from the α lowest (highest) numbered locations to the α highest (lowest) numbered locations.

3. Any consecutive BD records span no more than two stripes.

Figure 8 illustrates our algorithm to perform a toroidal route along a single dimension. We repeatedly slide the β records of the minority set toward their final destination by swapping them with the next β consecutive records of the majority set along its path until only $\alpha \bmod \beta < \beta$ records of the majority set have not been swapped.[3] At that point, there are not enough records in the majority set left to swap. All the swapped records in the majority set have been shifted β locations to their final locations (by Fact 2). If the β records of the minority set and the $\alpha \bmod \beta$ remaining records of the majority set can fit into memory, then we complete the toroidal route along a single dimension by reading these records into memory, permuting them into their correct order, and writing them to their final locations. Otherwise, we recursively reassign the β records of the minority set as the new majority set and the remaining $\alpha \bmod \beta$ records of the majority set as the new minority set. We continue swapping and reassigning majority and minority sets until $\beta + (\alpha \bmod \beta) \leq M$.

Our algorithm has three cases, of which the first two are easy to analyze. In the first case, when $\alpha + \beta \leq M$, we read the majority and minority sets into memory, permute them into the correct order, and write them to their final locations. Thus, we perform this case using $2\lceil(\alpha+\beta)/BD\rceil$ parallel I/Os, making no restriction on the size of memory as long as $\alpha + \beta \leq M$.

In the second case, $\alpha + \beta > M$ and $\beta \leq BD$, and we require that $3BD \leq M$. To perform a swap, we begin by reading the first two stripes that contain records from the minority set. These stripes may contain records from the majority set as well. Because $\beta \leq BD$, however, Fact 3 implies that the entire minority set resides in memory at this point. For the remainder of the algorithm, we keep the records of the minority set in memory and only read the stripes of the majority set as needed. Thus, we read and write every stripe exactly once for a total of $2\lceil(\alpha+\beta)/BD\rceil$ parallel I/Os. This case requires at most $3BD$ records of

[3] The sliding of the β records of the minority set toward the other end of the data set is similar to the process by which a snake digests a mouse. Thus, one could say that this is a *peristaltic* algorithm.

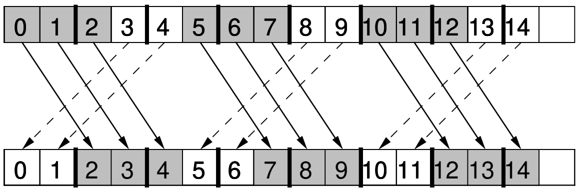


Figure 7: Mapping a torus permutation along dimension 1. On our 3×5 mesh, we map a torus permutation with positive index offset ($o'(p,1) = (0,2)$) and negative index offset ($o'(p,1) = (0,-3)$). In this example, the plain and dashed arrows indicate the majority and minority sets, respectively.

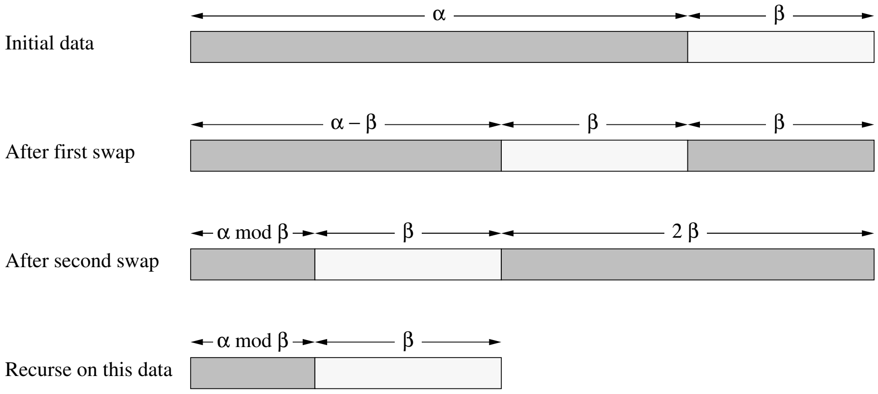


Figure 8: Sliding the minority set toward the other end of the data set. We iteratively swap the β records of the (light-shaded) minority set with the next β records of the (dark-shaded) majority set. After the second swap, $\alpha \bmod \beta < \beta$, preventing us from swapping one-to-one the records in the majority and minority sets. If $\beta + (\alpha \bmod \beta) > M$, we recurse on these records, reassigning the roles of the majority and minority sets.

memory, which include at most BD records to hold the β records of the minority set and at most $2BD$ records to hold the two stripes containing the records of the majority set that are involved in the current swap or the $\alpha \bmod \beta$ records of the majority set for the final permutation.

In the third case, which is when $\alpha + \beta > M$ and $\beta > BD$, our algorithm again requires only three stripes of memory. To begin any swap, we must have in memory the two stripes containing the first records to swap from the majority and minority sets. Before or after a swap, we label a stripe containing records from both the majority and minority sets as a *boundary stripe*. If the stripe with the first records of the majority set is a boundary stripe, then we pin that stripe in memory for the duration of the swap. Since the pinned stripe will already be in memory with the first records of the minority set for the next swap or level of recursion, over the course of the entire algorithm, we read and write each boundary stripe only once. We use the other two stripes of memory to hold the stripes with the current records to swap from the majority and minority sets when the records are not in the currently pinned stripe.

If we further define α_j and β_j as the number of records in the majority and minority sets for recursion level j, for $j \geq 0$, we can show that our algorithm uses at most $4(\alpha_j - \beta_{j+1})/BD$ parallel I/Os to perform level j in the recursion. (Note that $\alpha_{j+1} = \beta_j$ and $\beta_{j+1} = \alpha_j \bmod \beta_j$) Figure 9 shows the layout of records on the jth level of recursion. During a recursion level, we may read a stripe containing just records from the majority set, write it back to disk with just records of the minority set, read it back into memory again, and finally write it back to disk containing just records of the majority set. Thus, some stripes could account for a total of 4 parallel I/Os. After completing the last swap of β records between the majority and minority sets on a given recursion level, two boundary stripes remain in memory which contain the first records to swap in the next recursion level. Since we have read these two stripes only once up to this point, we shall defer counting the parallel I/Os for these two stripes until the next level of recursion. Thus, during the jth level of recursion, we access at most $(\alpha_j - \beta_{j+1} + \beta_j)/BD$ stripes, requiring at most a total of $4(\alpha_j - \beta_{j+1} + \beta_j)/BD$ parallel I/Os. (We avoid using the ceiling function because we do not count the two stripes that remain in memory.) Since we only read once and write once the first β_j and last β_j of these records, we reduce our parallel I/O count for the jth level of recursion to

$$\frac{4(\alpha_j - \beta_{j+1} + \beta_j)}{BD} - \frac{2(2\beta_j)}{BD} = \frac{4(\alpha_j - \beta_{j+1})}{BD} .$$

From the above descriptions, we define the following equations for the three cases of our in-place algorithm. Each bound represents a limit for the number $T(\alpha_j, \beta_j)$ of paral-

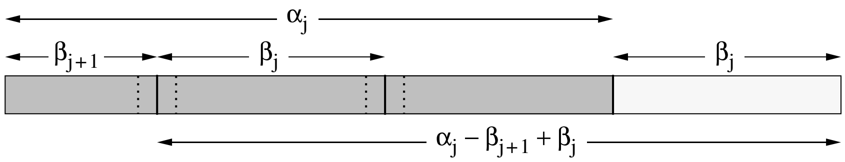


Figure 9: Layout of data on jth level of recursion. This level of recursion accesses the upper $\alpha_j - \beta_{j+1} + \beta_j$ records. We do not assess any parallel I/Os for the stripes within the dashed lines. These are the first stripes accessed during the next level of recursion, so we do not have to write them during this level of recursion. The two sections of β_j records are read and written at most once on this level of recursion.

lel I/Os performed within the jth level of recursion:

$$T(\alpha_j, \beta_j) \le \begin{cases} 2\lceil(\alpha_j+\beta_j)/BD\rceil & \text{if } \alpha_j+\beta_j \le M \ , \\ 2\lceil(\alpha_j+\beta_j)/BD\rceil & \text{if } \alpha_j+\beta_j > M \text{ and } \beta_j \le BD \ , \\ T(\alpha_{j+1},\beta_{j+1}) + 4(\alpha_j - \beta_{j+1})/BD & \text{if } \alpha_j+\beta_j > M \text{ and } \beta_j > BD \ . \end{cases}$$

If we let $N = \alpha_0 + \beta_0$, the first two cases require $2\lceil N/BD\rceil$ parallel I/Os. For the third case, we use the substitution method to show that $T(\alpha_j, \beta_j) \le 4(\alpha_j+\beta_j)/BD$:

$$\begin{aligned} T(\alpha_j, \beta_j) &= T(\alpha_{j+1},\beta_{j+1}) + 4(\alpha_j - \beta_{j+1})/BD \\ &= T(\beta_j,\beta_{j+1}) + 4(\alpha_j - \beta_{j+1})/BD \\ &\le 4(\beta_j+\beta_{j+1})/BD + 4(\alpha_j - \beta_{j+1})/BD \\ &= 4(\alpha_j+\beta_j)/BD \ . \end{aligned}$$

We can extend our analysis to include multiple copies of the alternating pattern of majority and minority sets by crediting the lowest level of the recursion (which uses only $\lceil 2(\alpha_j+\beta_j)/BD\rceil$ parallel I/Os) with the two parallel I/Os to access the uppermost stripe of a copy. Thus, our algorithm for the 2-monotonic toroidal route along a single dimension uses at most $4N/BD$ parallel I/Os. Since we perform at most d such permutations, we have an upper bound of $4dN/BD$ parallel I/Os.

5 Conclusions

We have shown how to perform a number of commonly used structured permutations in place with at most a constant factor more parallel I/Os and insignificant extra computation than the best known algorithms that use $2N$ records of disk space. These in-place algorithms use at most an extra memoryload of disk space, thereby requiring a little more than half of the disk space required by previous algorithms.

We further ask the following questions:

- Are there other interesting out-of-core computations for which algorithms exist to more efficiently use the available disk space?
- Is it possible to reduce the constants even further for the problems addressed in this paper, in particular, performing BMMC permutations in place?
- What are the exact constant factors for the general permuting algorithm of Vitter and Shriver? How much extra disk space does their algorithm require? Can we develop general algorithms that permute in place? Are there algorithms for general permuting that use $2N$ records of disk space and result in lower constants than the known algorithms?
- How do out-of-core permutation algorithms that are in-place compare in practice with algorithms that use $2N$ records of disk storage?

Acknowledgments

Thanks to Tom Cormen and Tom Sundquist for many helpful discussions on in-place permutations. Special thanks to Tom Cormen for challenging me to develop in-place algorithms for torus permutations, and for believing that the solution would be simple and elegant.

References

[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, 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. Springer-Verlag, 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.

[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.

[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.*

[EHJ94] Alan Edelman, Steve Heller, and S. Lennart Johnsson. Index transformation algorithms in a linear algebra framework. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1302–1309, December 1994.

[FMP95] Faith E. Fich, J. Ian Munro, and Patricio V. Poblete. Permuting in place. *SIAM Journal on Computing*, 24(2):266–278, April 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.

[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.

[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.

[VV95] Darren Erik Vengroff and Jeffrey Scott Vitter. I/O-efficient scientific computation using TPIE. In *Seventh IEEE Symposium on Parallel and Distributed Processing*, October 1995. To appear.