Dartmouth College Computer Science
Technical Report PCS-TR97-317

# Performing BMMC Permutations Efficiently on Distributed-Memory Multiprocessors with MPI

Thomas H. Cormen*
Dartmouth College
Department of Computer Science
thc@cs.dartmouth.edu

**Abstract**

This paper presents an architecture-independent method for performing BMMC permutations on multiprocessors with distributed memory. All interprocessor communication uses the MPI function `MPI_Sendrecv_replace()`. The number of elements and number of processors must be powers of 2, with at least one element per processor, and there is no inherent upper bound on the ratio of elements per processor.

Our method transmits only data without transmitting any source or target indices, which conserves network bandwidth. When data is transmitted, the source and target processors implicitly agree on each other's identity and the indices of the elements being transmitted.

A C-callable implementation of our method is available from Netlib. The implementation allows preprocessing (which incurs a modest cost) to be factored out for multiple runs of the same permutation, even if on different data. Data may be laid out in any one of several ways: processor-major, processor-minor, or anything in between.

## 1 Introduction

Suppose you were writing a multiprocessor application using MPI [GLS94, SOHL+96], and you needed to transpose a fairly large matrix distributed over several processors. You could write a specialized matrix-transpose function. You might parameterize it enough to work on non-square matrices. If you planned the data movement carefully, you could arrange the MPI communication calls so that each processor knew implicitly exactly which entries every other processor was sending it, in order to avoid sending row and column numbers along with the data.

Now suppose you were implementing a Fast Fourier Transform (FFT) using MPI, and you needed to perform the bit-reversal permutation used in some FFT methods. You could write a specialized bit-reversal permutation function and, like for matrix transpose, you could plan the data movement carefully in order to avoid sending indices along with the data.

*Supported in part by the National Science Foundation under grant CCR-9625894.

Suppose further that you had implemented one or both of the above functions under the assumption that data was stored in *processor-major* order (the most significant bits of an element's index denote the number of the processor that it resides on) but that you decided to store the data instead in *processor-minor* order (processor numbers are in the least significant index bits). You would have to rewrite your functions, for the data-movement patterns would change.

All the above cases, and many more, are specific instances of a more general operation that this paper shows how to perform. In particular, matrix transpose (when all dimensions are powers of 2) and the bit-reversal permutation are BMMC (bit-matrix-multiply/complement) permutations. In this paper, we show how to perform BMMC permutations on distributed-memory multiprocessors using only the MPI function `MPI_Sendrecv_replace()` for communication, subject to certain technical conditions. Our method assumes that there is at least one element per processor, and there is no inherent upper bound on the ratio of elements per processor. One useful property of BMMC permutations makes it easy to adjust the actual permutation performed when the data layout is not processor-major.

Our BMMC-permutation method is as fast as one could hope for in an MPI environment. It transmits only data without transmitting any source or target indices, thus conserving network bandwidth. We avoid transmitting indices by ensuring that during communication, source and target processors implicitly agree on each other's identity and the indices of the elements being transmitted. Whenever a processor has data to send to another processor, it sends it in one message, which reduces message overhead.

The basic idea of our method is as follows. We decompose the BMMC permutation into two BMMC permutations that, when performed in sequence, give the original one. The first permutation rearranges data within each processor's memory to get elements destined for the same target processor into contiguous memory locations. The second permutation transmits data among processors and places it into its correct location in the target processor's memory. By default, the method assumes that data is laid out in processor-major order, but we shall see that we can compensate for other orders by performing a modified BMMC permutation.

A C-callable implementation of our method is available from Netlib [Cor97]. The implementation allows the computation of the decomposition (which incurs a modest cost) to be factored out for multiple runs of the same permutation, even if on different data.

To our knowledge, this paper represents the first BMMC-permutation algorithm that is independent of the network architecture. Other authors have shown how to perform BMMC permutations (which are also known as affine transformations) on specific networks such as hypercubes [BR90, EHJ94], meshes [Sib92], Omega networks[1] [KS88], and expanded delta networks [WCS96]. Many of the techniques used in the present paper are adapted from earlier work in performing BMMC permutations on parallel disk systems [CSW94].

The remainder of this paper is organized as follows. Section 2 defines the class of BMMC permutations, shows how we represent them, and gives the technical conditions required to use our method. Section 3 previews the matrix forms used in Section 4 to decompose a BMMC permutation as described above. Section 5 shows how to actually perform the two permutations produced by the decomposition method. Section 6 presents how to compensate for non-processor-major data layouts. Section 7 contains some concluding remarks, focusing on the implementation in Netlib. There are three appendices containing background material. Appendix A shows how to compute a

[1]The algorithm in [KS88] is for BPC (bit-permute/complement) permutations, which are a large subclass of BMMC permutations.

column basis for a matrix, which we need to do in Section 4. Appendix B presents the Gray-code technique of calculating source and target indices, which we use in Section 5. Appendix C lists a few examples of BMMC permutations.

## 2 BMMC permutations

In this section, we define the class of BMMC permutations and give some examples of commonly used ones. We then show how to represent them and interpret data indices in a multiprocessor context.

### Definition of BMMC permutations

A *permutation* on $N$ elements is defined by a one-to-one mapping of *source indices* to *target indices*. The class of permutations we consider in this paper is *BMMC*, or *bit-matrix-multiply/complement*, permutations. BMMC permutations are defined only when the number $N$ of elements is an integer power of 2, so that $n = \lg N$ is an integer. A BMMC permutation is specified by an $n \times n$ *characteristic matrix* $A = (a_{ij})$ whose entries are drawn from $\{0, 1\}$ and is nonsingular (i.e., invertible) over $GF(2)$.[2] The specification also includes a *complement vector* $c = (c_{n-1}, c_{n-2}, \ldots, c_0)$ of length $n$. Treating each source index $x = (x_{n-1}, x_{n-2}, \ldots, x_0)$ as an $n$-bit vector, we perform matrix-vector multiplication over $GF(2)$ and then form the corresponding $n$-bit target index $y = (y_{n-1}, y_{n-2}, \ldots, y_0)$ by complementing some subset of the resulting bits: $y = A\,x \oplus c$, or

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

Note that we place the least significant bit at the top and left. Because we require the characteristic matrix $A$ to be nonsingular, the mapping of source addresses to target addresses is one-to-one.

The following lemma gives two useful properties of BMMC permutations.

**Lemma 1** *The class of BMMC permutations is closed under composition and inverse.*

*Proof:* We first show that BMMC permutations are closed under composition. Consider a BMMC permutation with characteristic matrix $A$ and complement vector $c$, and another BMMC permutation with characteristic matrix $A'$ and complement vector $c'$. We will show that their composition is a BMMC permutation with some characteristic matrix $A''$ and complement vector $c''$. The composition of the two BMMC permutations first maps a source index $x$ to $x' = A\,x \oplus c$ and then maps $x'$ to

$$\begin{aligned} y &= A'x' \oplus c' \\ &= A'(A\,x \oplus c) \oplus c' \\ &= A'\,A\,x \oplus A'\,c \oplus c' \\ &= (A'\,A)x \oplus (A'\,c \oplus c') , \end{aligned}$$

[2] 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.

and so $A'' = A' A$ and $c'' = A' c \oplus c'$.

Now we show that BMMC permutations are closed under inverse. If $y = A x \oplus c$, then we add $c$ to both sides and multiply by $A^{-1}$, giving

$$\begin{aligned} x &= A^{-1}(y \oplus c) \\ &= A^{-1} y \oplus (A^{-1} c) , \end{aligned}$$

and so the inverse permutation has characteristic matrix $A^{-1}$ and complement vector $A^{-1} c$. ∎

### Representation

Under the reasonable assumption that the word size of the underlying machine is at least $n$ bits, we represent the characteristic matrix as an array of $n$ columns, where each column is packed into a word with the top bit of the column as the least significant bit of the word. We denote the $j$th column of matrix $A$ by $A_j$. Source and target indices are stored in the obvious way, and the complement vector is stored in the same way. With this representation, it takes only $n+1 = \lg N + 1$ words to represent a BMMC permutation. Moreover, we can compute $y = A x \oplus c$ in $O(n)$ word operations:

```
y ← c
for j ← 0 to n − 1
     do if x_j = 1
           then y ← y ⊕ A_j
```

### Data layouts

In the remainder of this paper, we let $P$ denote the number of processors, and we assume that $P$ is a power of 2. Let $p = \lg P$. We refer to the number of elements per processor $N/P$ as the *virtual processor ratio*, or *VPR*.

Let us examine how to interpret the $n$ bits of an index. A particular set of $p$ bits determines which processor the element resides on, and the remaining $n - p = \lg(N/P)$ bits determine the offset of that element within its processor. Although it is possible to have a layout in which the $p$ processor bits are not consecutive within an index, we do not consider such layouts in this paper. We let the parameter $f$ denote the position of the least significant processor bit, so that $0 \le f \le n - p$.

Figure 1 shows how layouts differ depending on the value of $f$. In *processor-major* layout, the most significant bits contain the processor number, so that $f = n - p$. In *processor-minor* layout, the least significant bits contain the processor number, so that $f = 0$. In general, we can view a layout as consisting of "bands" striped across the processors. Each band consists of $2^f P$ elements, and there are $N/(2^f P)$ bands altogether.

Until Section 6, we assume that the data layout is processor-major. In Section 6, we shall see how to express the conversion between layouts with $f < n - p$ and $f = n - p$ (processor-major) as a BMMC permutation. We will adjust the permutation actually performed according to this conversion permutation.

| | processor-major $f=3$ | | | | $f=2$ | | | | $f=1$ | | | | processor-minor $f=0$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\mathcal{P}_0$ | $\mathcal{P}_1$ | $\mathcal{P}_2$ | $\mathcal{P}_3$ | $\mathcal{P}_0$ | $\mathcal{P}_1$ | $\mathcal{P}_2$ | $\mathcal{P}_3$ | $\mathcal{P}_0$ | $\mathcal{P}_1$ | $\mathcal{P}_2$ | $\mathcal{P}_3$ | $\mathcal{P}_0$ | $\mathcal{P}_1$ | $\mathcal{P}_2$ | $\mathcal{P}_3$ |
| offset 0 | 0 | 8 | 16 | 24 | 0 | 4 | 8 | 12 | 0 | 2 | 4 | 6 | 0 | 1 | 2 | 3 |
| offset 1 | 1 | 9 | 17 | 25 | 1 | 5 | 9 | 13 | 1 | 3 | 5 | 7 | 4 | 5 | 6 | 7 |
| offset 2 | 2 | 10 | 18 | 26 | 2 | 6 | 10 | 14 | 8 | 10 | 12 | 14 | 8 | 9 | 10 | 11 |
| offset 3 | 3 | 11 | 19 | 27 | 3 | 7 | 11 | 15 | 9 | 11 | 13 | 15 | 12 | 13 | 14 | 15 |
| offset 4 | 4 | 12 | 20 | 28 | 16 | 20 | 24 | 28 | 16 | 18 | 20 | 22 | 16 | 17 | 18 | 19 |
| offset 5 | 5 | 13 | 21 | 29 | 17 | 21 | 25 | 29 | 17 | 19 | 21 | 23 | 20 | 21 | 22 | 23 |
| offset 6 | 6 | 14 | 22 | 30 | 18 | 22 | 26 | 30 | 24 | 26 | 28 | 30 | 24 | 25 | 26 | 27 |
| offset 7 | 7 | 15 | 23 | 31 | 19 | 23 | 27 | 31 | 25 | 27 | 29 | 31 | 28 | 29 | 30 | 31 |

**Figure 1:** How data layouts differ depending on where the $p$ processor bits start (the parameter $f$). Each layout shows which processor and offset each element index maps to. In this example, $N = 32$, $P = 4$, and the $i$th processor is $\mathcal{P}_i$. Each layout can be viewed as consisting of bands of $2^f P$ elements.

# 3 Matrix forms

When factoring a characteristic matrix in Section 4, we will want certain submatrices to have particular forms. This section reviews a technique used in [CSW94] to produce such desired forms.

### Column-addition matrices

A *column-addition matrix* is a matrix $M$ such that the product $A' = A\,M$ is a modified form of $A$ in which specified columns of $A$ have been added (elementwise) into others. We define the matrix $M = (m_{ij})$ by

$$m_{ij} = \begin{cases} 1 & \text{if } i = j \;, \\ 1 & \text{if column } A_i \text{ is added into column } A_j \;, \\ 0 & \text{otherwise} \;. \end{cases}$$

For example,

$$\underbrace{\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}}_{M} = \left[\begin{array}{c|c|c|c} A_0 & A_0 \oplus A_1 \oplus A_3 & A_0 \oplus A_2 & A_3 \end{array}\right] = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{A'} .$$

Column-addition matrices must also obey a *dependency restriction* that if column $i$ is added into column $j$, then column $j$ must not be added into any column. In other words, if $m_{ij} = 1$, then $m_{jk} = 0$ for all $k \neq j$. With the dependency restriction, any column-addition matrix is nonsingular [CSW94].

In the remainder of this section, we present three matrix forms that we will use for factoring in Section 4. The matrices that we will use are of block forms. In particular, we partition a matrix according to the leftmost $n - p$ columns (the *left section*), the rightmost $p$ columns (the *right section*), the top $n-p$ rows (the *top section*), and the bottom $p$ rows (the *bottom section*). Certain submatrices will be either all 0 or an identity submatrix. We label such submatrices accordingly and place an asterisk (*) in all other submatrices.

**Trailer matrix form**

In Section 4, we shall transform a nonsingular matrix into one that has a nonsingular trailing $p \times p$ submatrix. With the resulting matrix, each processor will be able to determine implicitly which processors it communicates with. To perform this transformation, we add columns from the left section into columns in the right section. A *trailer matrix* $T$ is a column-addition matrix that does so. It has the following form:

$$T = \begin{array}{cc} \begin{array}{cc} n-p & p \end{array} & \\ \left[\begin{array}{c|c} I & * \\ \hline 0 & I \end{array}\right] & \begin{array}{l} n-p \\ p \end{array} \end{array} .$$

Observe that because the bottom section of a trailer matrix is the bottom $p$ rows of an identity matrix and because we assume processor-major layout, if a trailer matrix is taken as a characteristic matrix, the BMMC permutation it induces leaves every element in the processor it started in. That is, only offsets within processors change, rather than processor numbers. We call such a permutation an *intraprocessor permutation.*

**Reducer matrix form**

We will be putting matrices into "reduced form," which we will precisely define in Section 4. Reduced form will make it so that we can order communication into rounds. We will put a matrix into reduced form by adding columns from the left section into other columns from the left section. Therefore, a *reducer matrix* $R$ is a column-addition matrix with the following form:

$$R = \begin{array}{cc} \begin{array}{cc} n-p & p \end{array} & \\ \left[\begin{array}{c|c} * & 0 \\ \hline 0 & I \end{array}\right] & \begin{array}{l} n-p \\ p \end{array} \end{array} .$$

Since $R$ obeys the dependency restriction, it is nonsingular, even though we do not know the exact form of the leading $(n-p) \times (n-p)$ submatrix. Like a trailer matrix, the BMMC permutation induced by a reducer matrix is intraprocessor.

**Swapper matrix form**

The final matrix form we consider swaps pairs of columns from among the left section. This operation will help gather elements destined for the same target processor into consecutive memory locations. A *swapper matrix* $\Pi$ has the form

$$\Pi = \begin{array}{cc} \begin{array}{cc} n-p & p \end{array} & \\ \left[\begin{array}{c|c} * & 0 \\ \hline 0 & I \end{array}\right] & \begin{array}{l} n-p \\ p \end{array} \end{array} ,$$

where the leading $(n-p) \times (n-p)$ submatrix is a *permutation matrix*, i.e., it contains exactly one 1 in each row and in each column. (A swapper matrix is not a column-addition matrix.) Once again, the BMMC permutation induced is intraprocessor.

### Composition

In fact, we will compose the BMMC permutations induced by matrices of the three forms above. The composition will have a characteristic matrix of the form

$$T\,R\,\Pi \;=\; \begin{array}{cc} \begin{array}{cc} n-p & p \end{array} & \\ \left[\begin{array}{c|c} * & * \\ \hline 0 & I \end{array}\right] & \begin{array}{l} n-p \\ p \end{array} \end{array} \quad .$$

Observe that both this composition and its inverse are intraprocessor permutations.

## 4 Factoring a BMMC permutation

With the matrix forms in hand, we are now ready to factor a BMMC permutation. We will factor the characteristic matrix $A$ into two factors:

$$A = V\,W \;.$$

The factor $W$ will be the inverse of the composition described in Section 3, and so the BMMC permutation it characterizes is an intraprocessor permutation. This permutation gathers elements destined for the same target processor into contiguous memory locations so that they can be sent in one message. The BMMC permutation induced by the factor $V$, on the other hand, is an *interprocessor permutation*, i.e., elements may move among processors. Given these factors, Section 5 will show how to first perform the permutation with the mapping

$$x' = W\,x \tag{1}$$

followed by the permutation with the mapping

$$y = V\,x' \oplus c = A\,x \oplus c \;. \tag{2}$$

In this section, we first present the transformations that produce the factoring. Then we examine desirable properties of the resulting factoring.

### Creating a nonsingular trailing submatrix

We start by transforming the characteristic matrix $A$ into a nonsingular matrix $A'$ that has a nonsingular trailing $p \times p$ submatrix. One useful property is given by the following well-known theorem from linear algebra. The *column (row) rank* of a matrix is the size of a maximal set of linearly independent columns (rows), with linear independence here being over $GF(2)$.

**Theorem 2** *For any matrix, the row rank equals the column rank.* ■

Thus, we can refer to either the column rank or the row rank as simply the *rank*, and we denote the rank of a matrix $A$ by $\operatorname{rank} A$. We call a maximal set of linearly independent columns (rows) a *column (row) basis.*

If we represent the matrix $A$ as

$$A = \begin{array}{c} \\ \\ \end{array}\begin{array}{cc} \begin{array}{cc} n-p & p \end{array} & \\ \left[\begin{array}{c|c} \alpha & \beta \\ \hline \gamma & \delta \end{array}\right] & \begin{array}{l} n-p \\ p \end{array} \end{array} ,$$

then we will add columns in $\gamma$ into those in $\delta$. Consider $\delta$ as a set of $p$ columns and $\gamma$ as a set of $n-p$ columns. Because $A$ is nonsingular, all rows of the bottom section of $A$, consisting of $\gamma$ and $\delta$, are linearly independent. Hence, the bottom section has row rank $p$, and by Theorem 2, there exists a column basis of $p$ columns within the bottom section. One such basis contains a set $Y$ of $\operatorname{rank}\delta$ columns of $\delta$ and a set $Z$ of $p - \operatorname{rank}\delta$ columns of $\gamma$. (Appendix A shows how to find this column basis.) Let $\overline{Y}$ be the set of $p - \operatorname{rank}\delta$ columns of $\delta$ not in $Y$.

Having defined these column sets, we create a nonsingular trailing submatrix by pairing up columns of $Z$ with columns of $\overline{Y}$ and adding each column in $Z$ into its counterpart in $\overline{Y}$. Because $|Y| = \operatorname{rank}\delta$, $Y$ is a column basis for $\delta$, and so the columns of $\overline{Y}$ depend only on columns of $Y$ and not on columns of $Z$. Adding a column of $Z$ into a column of $\overline{Y}$ must produce a column that is linearly independent of those in $Y$. Because each column of $\overline{Y}$ has a different column of $Z$ added in, the resulting columns are linearly independent of each other, too.

Let us express this operation as a column-addition matrix. Although we focused on adding columns of $\gamma$ to columns of $\delta$, we are in fact adding entire columns, and so we are also adding some columns of $\alpha$ into columns of $\beta$. Because we are adding columns of the left section into columns of the right section, we use a trailer matrix $T$ of the form described in Section 3, giving us the matrix product

$$A' = AT = \begin{array}{cc} \begin{array}{cc} n-p & p \end{array} & \\ \left[\begin{array}{c|c} \alpha & \widehat{\beta} \\ \hline \gamma & \widehat{\delta} \end{array}\right] & \begin{array}{l} n-p \\ p \end{array} \end{array} ,$$

where $\widehat{\delta}$ is nonsingular. Since the matrices $A$ and $T$ are nonsingular, the matrix $A'$ is nonsingular.

### Transforming the matrix into reduced form

The next step is to transform the submatrix $\gamma$ into reduced form. For our purposes, a matrix is in *reduced form* when all non-basis columns are 0.

To perform this transformation, we start by finding a column basis $S$ for $\gamma$. For each column $\gamma_j$ not in $S$, we find a set of columns of $\gamma$ that add up to $\gamma_j$. (Appendix A shows how to compute these column dependencies.) Because we are working over $GF(2)$, adding these columns into $\gamma_j$ zeroes it out.

Again, we are actually working with entire columns. We add basis columns from the left section into non-basis columns in the left section, and so as a column-addition matrix, the operation respects the dependency restriction. The column-addition matrix is a reducer matrix $R$, and we now have the product

$$A'' = A'R = \begin{array}{cc} \begin{array}{cc} n-p & p \end{array} & \\ \left[\begin{array}{c|c} \widehat{\alpha} & \widehat{\beta} \\ \hline \widehat{\gamma} & \widehat{\delta} \end{array}\right] & \begin{array}{l} n-p \\ p \end{array} \end{array} ,$$

where $\widehat{\delta}$ is nonsingular and $\widehat{\gamma}$ is in reduced form. Because the basis columns of $\gamma$ do not change, $\operatorname{rank}\widehat{\gamma} = \operatorname{rank}\gamma$. Because $A'$ and $R$ are nonsingular, so is $A''$.

### Gathering basis columns

Our final transformation is to gather the columns of the basis $S$ into consecutive positions on the right end of $\widehat{\gamma}$. That is, we permute columns in the left section so that the leftmost $n - p - \operatorname{rank}\gamma$ columns of $\widehat{\gamma}$ are 0 and the rightmost $\operatorname{rank}\gamma$ columns of $\widehat{\gamma}$ form the column basis $S$. The matrix $\Pi$ that gathers the basis columns exchanges pairs of columns in the leftmost section, and so it is a swapper matrix.

The resulting product will be the factor $V$, and it has the form

$$V \;=\; A''\,\Pi \;=\; \begin{array}{c} \begin{array}{ccc} \scriptstyle n-p-\operatorname{rank}\gamma & \scriptstyle \operatorname{rank}\gamma & \scriptstyle p \end{array} \\ \left[\begin{array}{c|c|c} \widehat{\alpha}' & \widehat{\alpha}'' & \widehat{\beta} \\ \hline 0 & \widehat{\gamma}'' & \widehat{\delta} \end{array}\right] \end{array} \begin{array}{l} \scriptstyle n-p \\ \scriptstyle p \end{array} \quad , \tag{3}$$

where $\widehat{\gamma}''$ consists of the basis columns of $\widehat{\gamma}$ (and hence of $\gamma$ as well). Both $\Pi$ and $V$ are nonsingular.

### Factorization

Since we want the factorization $A = V\,W$, we expand equation (3) to produce

$$\begin{aligned} V &= A''\,\Pi \\ &= A'\,R\,\Pi \\ &= A\,T\,R\,\Pi \ . \end{aligned}$$

If we let $W = (T\,R\,\Pi)^{-1}$, we have

$$\begin{aligned} A &= V\,(T\,R\,\Pi)^{-1} \\ &= V\,W \ , \end{aligned}$$

giving the desired factorization to use in performing the permutations given by equations (1) and (2).

### Properties of the factors

Why did we go to so much trouble to factor $A$ in the above manner? The answer lies in the properties of the factors $V$ and $W$. As we noted in Section 3, the factor $W$ characterizes an intraprocessor permutation. We shall see in Section 5 that intraprocessor BMMC permutations are easy and fast.

The key to the factorization lies in the properties of the interprocessor permutation characterized by the factor $V$. More specifically, they lie in how elements map between processors. Consider a given processor $k$, where $0 \le k \le P - 1$. How many different processors do the elements that start on $k$ map to when performing the permutation characterized by the original matrix $A$? The answer is given by the following lemma, whose proof appears in [CSW94].[3]

**Lemma 3** *Let $\gamma$ be the lower left $p \times (n-p)$ submatrix of $A$, and consider any processor $k$. There are exactly $2^{\operatorname{rank}\gamma}$ target processors that the elements of $k$ map to, and for each such target processor, exactly $N/(2^{\operatorname{rank}\gamma}\,P)$ elements from processor $k$ map to it.* ■

[3] The lemma in [CSW94] is couched in terms of blocks on a parallel disk system, but it translates easily to the context of the present paper.

Since $W$ characterizes an intraprocessor permutation, any movement among processors must occur during the permutation characterized by $V$. In other words, the $2^{\operatorname{rank}\gamma}$ processors that processor $k$ sends elements to when performing the permutation characterized by $A$ are the same as the target processors for the permutation characterized by $V$.

To determine exactly which $2^{\operatorname{rank}\gamma}$ processors $k$ sends to, we write the matrix equation given by the bottom section of equation (3) in block form, where the notation ". ." indicates a range of bit indices. We also include the complement vector:

$$y_{n-p..n-1} = \widehat{\gamma}'' \, x_{(n-p-\operatorname{rank}\gamma)..(n-p-1)} \oplus \widehat{\delta} \, x_{n-p..n-1} \oplus c_{n-p..n-1} \ . \qquad (4)$$

Here, $x_{n-p..n-1}$ is the binary representation of $k$. As we vary the bits of $x_{(n-p-\operatorname{rank}\gamma)..(n-p-1)}$ (we call these the *basis bits*) among all $2^{\operatorname{rank}\gamma}$ combinations, $y_{n-p..n-1}$ takes on all $2^{\operatorname{rank}\gamma}$ target processor numbers. Including the reducer matrix as a factor made it so that we only need to vary $2^{\operatorname{rank}\gamma}$ basis bits, and including the swapper matrix made it so that the basis bits are in predetermined locations.

Another effect of the swapper matrix is that all elements going from processor $k$ to a given target processor reside in consecutive locations in processor $k$'s memory. Consider a target processor $l$ that processor $k$ sends elements to. By Lemma 3, processor $k$ sends exactly $N/(2^{\operatorname{rank}\gamma} P)$ elements to processor $l$. Observe that in the permutation characterized by $V$, if elements with source indices $x$ and $x'$ both move from processor $k$ to processor $l$, then $x_{(n-p-\operatorname{rank}\gamma)..(n-1)} = x'_{(n-p-\operatorname{rank}\gamma)..(n-1)}$, i.e., they must have the same bits in positions $n-p-\operatorname{rank}\gamma$ to $n-1$ of their source indices. Because these are the most significant bit positions, these elements reside in $N/(2^{\operatorname{rank}\gamma} P)$ consecutive locations in processor $k$'s memory prior to being sent.

Finally we come to the effect of the trailer matrix, which makes it so that we can easily determine which processor is sending to processor $k$. That is, we can determine which processor is the source processor when processor $k$ is the target. Let us rewrite equation (4) to compute the source processor $x_{n-p..n-1}$ corresponding to a given target processor $y_{n-p..n-1}$ and the basis bits $x_{(n-p-\operatorname{rank}\gamma)..(n-p-1)}$:

$$x_{n-p..n-1} = (\widehat{\delta})^{-1} \left( y_{n-p..n-1} \oplus \widehat{\gamma}'' \, x_{(n-p-\operatorname{rank}\gamma)..(n-p-1)} \oplus c_{n-p..n-1} \right) . \qquad (5)$$

Substituting $k$ for $y_{n-p..n-1}$, equation (5) tells us which processor is sending to processor $k$ for a given set of basis bits.

## 5 Performing the factor permutations

In this section, we see how to perform the BMMC permutations characterized by equations (1) and (2). Neither of these permutations is performed in-place. That is, both copy the data from one buffer into another. When we are done, however, each processor's original data is overwritten with the permuted data. We first see how to perform the intraprocessor permutation given by equation (1), and then how to perform the interprocessor permutation given by equation (2). Recall that elements are stored in processor-major order, so that the most significant $p$ bits of an index give an element's processor number and the least significant $n-p$ bits give the offset within the processor.

### Performing the intraprocessor permutation

To perform the intraprocessor permutation given by equation (1), we assume that each processor starts with $N/P$ elements stored in an $N/P$-element array *data*. The $N/P$ elements are copied, in a permuted order, into the $N/P$-element array *temp*.

We could perform the intraprocessor permutation in a straightforward manner:

1 **for** each processor $k$, in parallel<br>
2   **do** $x_{n-p..n-1} \leftarrow k$<br>
3    **for** $x_{0..n-p-1} \leftarrow 0$ **to** $N/P - 1$<br>
4     **do** $x' \leftarrow W\,x$<br>
5      $temp[x'_{0..n-p-1}] \leftarrow data[x_{0..n-p-1}]$

Note that we are careful to index offsets using only the least significant $n - p$ bits.

There is a more efficient way to perform this permutation, however. Observe that computing $x'$ in line 4 takes $O(n)$ time using the matrix-vector multiplication method in Section 2. Over all $N/P$ elements in each processor, we use $O((N/P) \lg N)$ word operations. We can reduce this count to $O((N/P) \lg(N/P))$ by realizing that the most significant $p$ bits of $x$ are fixed at $k$:

1 **for** each processor $k$, in parallel<br>
2   **do** $x_{n-p..n-1} \leftarrow k$<br>
3    let $W'$ be the left section (leftmost $n - p$ columns) of $W$<br>
4    let $W''$ be the right section (rightmost $p$ columns) of $W$<br>
5    $z \leftarrow W''\,x_{n-p..n-1}$<br>
6    **for** $x_{0..n-p-1} \leftarrow 0$ **to** $N/P - 1$<br>
7     **do** $x' \leftarrow W'\,x_{0..n-p-1} \oplus z$<br>
8      $temp[x'_{0..n-p-1}] \leftarrow data[x_{0..n-p-1}]$

Now each matrix-vector multiplication in line 7 uses only $O(n - p)$ word operations.

We can do even better: $\Theta(N/P)$ word operations over all $N/P$ elements. The idea is to choose elements to move not in linear order but in Gray-code order. After all, we can choose any order we want for moving the elements, as long as we move them all. In a Gray code, each index differs from the index before it in only one bit position. We use this property to compute each target index $x'$ in $O(1)$ word operations on average. Appendix B presents the details of the Gray-code method.

### Performing the interprocessor permutation

To perform the interprocessor permutation given by equation (2), we assume that each processor starts with its $N/P$ elements placed into the array *temp* by the intraprocessor permutation. When the permutation concludes, the elements are placed into the appropriate positions of the array *data* on the correct processors.

The computation proceeds in $2^{\text{rank}\,\gamma}$ rounds, where each round uses a different value $b$ for the rank $\gamma$ basis bits. Data moves in each round according to some permutation of the processors; that is, in each round, each processor sends to a unique target processor and therefore receives from a unique source processor. Using the form of the matrix $V$ given in equation (3), the pseudocode is as follows:

1 **for** each processor $k$, in parallel
2 **do for** $b \leftarrow 0$ **to** $2^{\text{rank}\,\gamma} - 1$
3 **do** $\mathit{target\_proc} \leftarrow \widehat{\gamma}''\, b \oplus \widehat{\delta}\, k \oplus c_{n-p..n-1}$
4 $\mathit{source\_proc} \leftarrow (\widehat{\delta})^{-1}\, (k \oplus \widehat{\gamma}''\, b \oplus c_{n-p..n-1})$
5 simultanenously send the $N/(2^{\text{rank}\,\gamma}P)$ elements starting at $\mathit{temp}[b \cdot N/(2^{\text{rank}\,\gamma}P)]$ to processor *target_proc* and receive $N/(2^{\text{rank}\,\gamma}P)$ elements from processor *source_proc*, overwriting the buffer starting at $\mathit{temp}[b \cdot N/(2^{\text{rank}\,\gamma}P)]$ with the received elements
6 **for** $j \leftarrow 0$ **to** $N/(2^{\text{rank}\,\gamma}P) - 1$
7 **do** $t \leftarrow \widehat{\alpha}'\, j \oplus \widehat{\alpha}''\, b \oplus \widehat{\beta} \cdot \mathit{source\_proc} \oplus c_{0..n-p-1}$
8 $\mathit{data}[t] \leftarrow \mathit{temp}[j]$

This code works as follows. The value of $b$ used in line 2 determines which round the processor is on; because all processors choose values of $b$ in the same order, they all agree on the round. Line 3 computes which processor will receive data from processor $k$ in the current round, according to equation (4). Similarly, line 4 computes which processor will be sending data to processor $k$ in the current round, according to equation (5). As we have seen, for a given value of the basis bits, each processor sends data to a unique target processor, so that the simultaneous send/receive of line 5 is well defined. Moreover, once a processor sends data, it no longer needs it, so that the data it receives can overwrite the send buffer. Our implementation calls the `MPI_Sendrecv_replace()` function, which is a perfect fit for this style of communication. The for-loop of lines 6–8 iterates through the $N/(2^{\text{rank}\,\gamma}P)$ non-basis bits within the processor, copying the elements into their final locations. The computation of the target offset $t$ in line 7 is based on equation (3). It depends on the value $j$ of the non-basis bits, the value $b$ of the basis bits, the value *source_proc* of the processor that just sent to processor $k$, and the least significant $n-p$ bits of the complement vector.

Just as our implementation of the intraprocessor permutation copies data in Gray-code order, so does our implementation of the for-loop of lines 6–8. Each processor uses a total of $\Theta(N/P)$ word operations in copying received data from *temp* to *data*. Although one could process the basis bits in the outer for-loop in Gray-code order, our implementation does not. We expect $\text{rank}\,\gamma$ usually to be much smaller than $\lg(N/P)$ so that the savings from computing *target_proc* and *source_proc* in lines 3 and 4 with the Gray-code technique would be negligible. Our implementation does factor out common subexpressions where possible.

We point out once more that the interprocessor communication in line 5 transmits only data. Because the receiving processor implicitly knows all the bits of each element's source index, it has enough information to compute the corresponding target index.

## 6 Adjusting for non-processor-major data layouts

The method for performing BMMC permutations given in Sections 4 and 5 assumes that data is laid out in processor-major order. In this section, we see how to adjust for other data layouts. We adjust by altering the characteristic matrix and complement vector of the given permutation. The methods for factoring and performing the permutations do not change at all once the characteristic matrix and complement vector have been altered. The only assumption we need to make is that the $p$ bits indicating which processor an element resides on are the consecutive bits in positions

$f, f+1, \ldots, f+p-1$, where $0 \le f \le n-p$.

The key idea is that we can convert between the data layout actually used (which we will call the *actual layout*) and processor-major layout via a BMMC permutation. To convert from processor-major layout to the actual layout, we use the characteristic matrix

$$
Q \;=\; \begin{array}{c} \begin{array}{ccc} f & p & n-p-f \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} \\ f \\ n-p-f \\ p \end{array}
$$

and a complement vector of 0. The inverse permutation, which converts from the actual layout to processor-major, uses the characteristic matrix

$$
Q^{-1} \;=\; \begin{array}{c} \begin{array}{ccc} f & n-p-f & p \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} \\ f \\ p \\ n-p-f \end{array}
$$

and again a complement vector of 0. Note that $Q = Q^{-1} = I$ if $f = n-p$, i.e., if the actual layout is processor-major.

Given the characteristic matrices $Q$ and $Q^{-1}$ and the original characteristic matrix $A$ and complement vector $c$, we can express the BMMC permutation as the composition of three BMMC permutations. Because our method assumes processor-major layout, we first convert the actual layout to processor-major by performing the permutation $x' = Q^{-1}\,x$. We next perform the original BMMC permutation, but on the processor-major layout: $x'' = A\,x' \oplus c = A\,(Q^{-1}\,x) \oplus c$. Finally, we convert back to the actual layout by performing $y = Q\,x'' = Q\,(A\,(Q^{-1}\,x) \oplus c) = (Q\,A\,Q^{-1})\,x \oplus (Q\,c)$.

Applying Lemma 1, instead of performing these three permutations consecutively, we perform their composition. That is, we perform the BMMC permutation with characteristic matrix $Q\,A\,Q^{-1}$ and complement vector $Q\,c$.

# 7 Conclusion

This paper has detailed the algorithm behind the software package `libbmmc_mpi.a` [Cor97], which is available from Netlib. Although the factoring techniques in Section 4 are borrowed from the out-of-core BMMC permutation algorithm [CSW94], the application of these techniques in Section 5 is new.

There are several ways to invoke the C-callable implementation in `libbmmc_mpi.a`. The basic call is of `int BMMC_MPI(bit_matrix A, matrix_column c, int n, int p, int f, int rank, MPI_Comm comm, int size, void *data, void *temp)`, where a header file provides the types `bit_matrix` and `matrix_column`, the parameter `rank` is the calling processor's number (between 0 and $P-1$), `comm` is an MPI communicator, and `size` is the size in bytes of each element to permute. The return value is an error code.

When performing the same BMMC permutation multiple times, the factoring procedure of Section 4 can be "factored out" with the results placed into an opaque type `BMMC_MPI_factor_info`, which is defined in the header file. The factoring is performed by calling `int factor_BMMC_MPI(bit_matrix A, matrix_column c, int n, int p, int f, BMMC_MPI_factor_info *info)`. Given the factoring, one may pass the opaque type into calls to `int perform_BMMC_MPI(BMMC_MPI_factor_info *info, int n, int p, int rank, MPI_Comm comm, int size, void *data, void *temp)`. Again, the return values are error codes. Because the factoring does not depend on the data or even the element sizes, a single call of `factor_BMMC_MPI()` may be used for several calls to `perform_BMMC_MPI()` with differing values of `rank`, `comm`, `size`, `data`, and `temp`.

There are also modified versions of `BMMC_MPI()` and `factor_BMMC_MPI()` that are specialized for processor-major and processor-minor data layouts. These functions are simply wrappers that call `BMMC_MPI()` and `factor_BMMC_MPI()` with the parameter `f` set to `n-p` for processor-major and to `0` for processor-minor.

This paper does not include experimental results. MPI is an interface standard. There are many conforming implementations. Performance depends on both the MPI implementation and the underlying hardware and software. The purpose of this paper has been to present the technique in the context of a portable implementation on top of MPI.

Although our implementation performs all communication via the `MPI_Sendrecv_replace()` function, a slight modification allows the use of either `MPI_Sendrecv()` or `MPI_Alltoallv()`. Limited experiments using two networks of workstations and the MPICH[4] implementation of MPI have shown no significant difference among implementations using `MPI_Sendrecv_replace()`, `MPI_Sendrecv()`, or `MPI_Alltoallv()`. On a different platform with a different implementation of MPI, it may be the case that the all-to-all communication of `MPI_Alltoallv()` proves superior.

**Acknowledgments**

Thanks to Len Wisniewski for many helpful discussions on the factoring method. James Clippinger and Anna Poplawski provided valuable comments on an earlier draft of this paper, and James also checked over the software. Steve Heller pointed out the Gray-code technique for target-index calculation. Lars Paul Huse gave useful feedback on the software in Netlib.

## References

[BR90] Rajendra Boppana and C. S. Raghavendra. Optimal self routing of linear-complement permutations in hypercubes. In *Proceedings of the 5th Distributed Memory Conference*, pages 800–808, April 1990.

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

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

[4] Available at http://www.mcs.anl.gov/mpi/mpich/.

[Cor97] Thomas H. Cormen. `libbmmc_mpi.a`: a library to perform fast BMMC permutations for any multiprocessor system that supports MPI. Available from Netlib at http://www.netlib.org/mpi/contrib/bmmc.tar.gz, 1997.

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

[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.

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

[KS88] John Keohane and Richard E. Stearns. Routing linear permutations though the Omega network in two passes. In *Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 479–482, October 1988.

[Sib92] Jop Frederik Sibeyn. *Algorithms for Routing on Meshes*. PhD thesis, Utrecht University, December 1992.

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

[WCS96] Leonard F. Wisniewski, Thomas H. Cormen, and Thomas Sundquist. Performing BMMC permutations in two passes through the expanded delta network and MasPar MP-2. In *Proceedings of the Sixth Symposium on The Frontiers of Massively Parallel Computation*, pages 282–289, October 1996.

# A Finding basis columns and dependencies

Although one can look up in a linear algebra text how to find a column basis for a matrix and determine the column dependencies, we include a method here for completeness. The input is an $n$-column matrix $A = (a_{ij})$, stored as an array of columns so that each column $A_j$ is packed into a word. There are two outputs. The set $S \subseteq \{0, 1, \ldots, n-1\}$, where $|S| \leq n$, indexes the columns of $A$ in the column basis. The matrix $D = (d_{ij})$ gives the column dependencies: $d_{ij} = 0$ if $i \neq j$ and column $A_j$ depends on column $A_i$. The method given here destroys the matrix $A$ in the process; if we need to keep the matrix $A$ intact, we assume that we are working with a copy of it. Here is pseudocode to produce $S$ and $D$ given $A$:

```
1   S ← ∅
2   for j ← 0 to n − 1
3        do D_j ← 0
4   for j ← n − 1 downto 0
5        do if A_j ≠ 0
6              then S ← S ∪ {j}
7                   let i be the minimum row number such that a_ij = 1
8                   for k ← 0 to j − 1
9                       do if a_ik = 1
10                            then A_k ← A_k ⊕ A_j
11                                 D_k ← D_k ⊕ D_j
12                                 d_jk ← ¬d_jk
13                  D_j ← 0
```

This code works as follows. Lines 1–3 initialize the basis $S$ to be empty and the dependency matrix $D$ to 0. The outermost for-loop of lines 4–13 goes in decreasing order so that we add columns to the basis from right to left. This order helps when we are trying to find the column basis needed in Section 4 to make the trailing $p \times p$ submatrix be nonsingular. All non-basis columns are eventually zeroed out. If we find a column $A_j$ in line 5 that has not been zeroed out, we add $j$ to the basis in line 6. We find the topmost 1 in column $A_j$ in line 7, letting it be in row $i$. Then we check all columns $A_k$ to the left of $A_j$ in lines 8–12. Any such column with a 1 in row $i$ depends on $A_j$, and so we add $A_j$ into it in line 10. Because column $A_k$ depends on any columns that $A_j$ depends on, we add $A_j$'s dependencies into $A_k$'s dependencies in line 11. We know that $A_j$ does not depend on itself, however, and so we complement $d_{jk}$ in line 12 to record $A_k$'s dependence on $A_j$. Once we are done adding in $A_j$'s dependencies, we zero them out in line 13.

With each column packed into a word, this code uses $O(n^2)$ word operations.

## B Calculating indices in Gray-code order

This appendix shows how to calculate source and target indices in Gray-code order. As noted in Section 5, the advantage of doing so is that each index calculation takes only $O(1)$ word operations on average. This technique was previously reported in [CB95].

To simplify the discussion, let us consider the general case in which we are performing the permutation given by $y = A\,x \oplus c$. Let $A$ have $\lg N$ columns, and assume that we wish to generate source indices varying from 0 to $N-1$ in Gray-code order along with the corresponding target index for each source index. The idea will translate to any specific permutation in a straightforward manner.

We start with source index $x = 0$, with the corresponding target index $y = c$. Now suppose that we have already computed $y = A\,x \oplus c$, and we wish to compute $y' = A\,x' \oplus c$, where $x$ and $x'$ differ only in the $i$th bit. Then $x'_i = x_i \oplus 1$ and $x'_j = x_j$ for all $j \neq i$. We have

$$\begin{aligned} y' &= x'_0 A_0 \oplus \cdots \oplus x'_i A_i \oplus \cdots \oplus x'_{n-1} A_{n-1} \oplus c \\ &= x_0 A_0 \oplus \cdots \oplus (x_i \oplus 1) A_i \oplus \cdots \oplus x_{n-1} A_{n-1} \oplus c \\ &= x_0 A_0 \oplus \cdots \oplus x_i \oplus A_i \oplus A_i \oplus \cdots \oplus x_{n-1} A_{n-1} \oplus c \end{aligned}$$

$$
\begin{aligned}
&= A\,x \oplus c \oplus A_i \\
&= y \oplus A_i \ .
\end{aligned}
$$

Once we know the bit position $i$ in which $x$ and $x'$ differ, we can compute $y' = y \oplus A_i$ in only $\Theta(1)$ word operations.

How do we determine the bit position $i$? For the standard binary reflected Gray code, the $j$th value and the $(j+1)$st value differ in the position of the rightmost 1 in the binary representation of $j+1$. This value is easy to find by simply examining bits from right to left. We can show that on average, we examine fewer than 2 bits to find the rightmost 1. Of the $N$ integers from 1 to $N$, exactly $N/2^i$ have the rightmost 1 in the $i$th bit examined. The total number of bits examined, therefore, is at most

$$
\begin{aligned}
\sum_{i=0}^{\lg N - 1} i \cdot \frac{N}{2^i} &< \sum_{i=0}^{\infty} i \cdot \frac{N}{2^i} \\
&= 2N
\end{aligned}
$$

using the identity $\sum_{i=0}^{\infty} i a^i = a/(1-a)^2$ [CLR90] with $a = 1/2$. Thus, the total number of word operations over all $N$ elements is $\Theta(N)$. In the application of this technique in Section 5, there are $N/P$ index pairs to calculate, and so it takes $\Theta(N/P)$ word operations in total.

In practice, one can optimize the process of finding the bit position $i$ to reduce the constant factors. In our implementation, for example, we maintain a static array *flips* with entries indexed 1 to 15, where *flips*$[j]$ holds the position of the rightmost 1 in the binary representation of $j$. We find the position $i$ of the rightmost 1 in $j+1$ with the following pseudocode:

```
q ← j + 1
i ← 0
nibble ← least significant 4 bits of q
while nibble = 0
    do q ← q/16          (shift q right by 4 bits)
       i ← i + 4
       nibble ← least significant 4 bits of q
i ← i + flips[nibble]
```

Here we examine 4 bits (a nibble) at a time rather than just one bit. Only one time in 16 does the while-loop body execute.

# C Examples of BMMC permutations

Although not all permutations on $N$ elements are BMMC, several commonly performed permutations are. We list some of them in this appendix.

It is easy to see that only a small fraction of all permutations are BMMC. There are $N!$ permutations on $N$ elements. Because a characteristic matrix has $\lg^2 N$ entries and a complement vector has $\lg N$ entries, there are $(2^{\lg^2 N})(2^{\lg N}) = N^{\lg N + 1} \ll N!$ possible combinations of characteristic matrix and complement vector. Moreover, not all of the possible characteristic matrices are nonsingular.

Even though relatively few permutations are BMMC, they occur frequently. Here are some common ones:

- Matrix transpose when all dimensions are powers of 2. If a matrix is $q \times r$, where $q$ and $r$ are powers of 2, the transpose permutation maps the $(i, j)$ entry to the $(j, i)$ entry. Assuming that the matrix is stored in row-major order, the transpose permutation interchanges the most significant $\lg q$ bits (initially containing the row number) and the least significant $\lg r$ bits (initially containing the column number). The characteristic matrix has the form

$$\begin{array}{cc} \begin{array}{cc} \lg q & \lg r \end{array} & \\ \left[\begin{array}{c|c} 0 & I \\ \hline I & 0 \end{array}\right] & \begin{array}{l} \lg r \\ \lg q \end{array} \end{array} ,$$

  where $I$ denotes identity submatrices. The complement vector is 0.

  Note that there are two matrices here. The data to be transposed forms a $q \times r$ matrix, where $qr = N$, and the type of each entry is unspecified. The characteristic matrix is $\lg N \times \lg N$, and each entry is 0 or 1.

- Shuffle and unshuffle permutations. These are matrix transpose permutations where the matrix to transpose is $N/2 \times 2$ or $2 \times N/2$.

- Bit-reversal permutations. Here, $y_i = x_{\lg N - i - 1}$ for $i = 0, 1, \ldots, \lg N - 1$. The corresponding characteristic matrix has 1s on the antidiagonal and 0s elsewhere, and the complement vector is 0.

- Vector-reversal permutations. Here, the $i$th input element maps to the $(N - i - 1)$st output element. This mapping corresponds to simply complementing all bits of the index: $y_i = \overline{x}_i$ for $i = 0, 1, \ldots, \lg N - 1$. Here, the characteristic matrix is the identity matrix and the complement vector is all 1s.

- Gray-code permutations. In the standard binary reflected Gray code, we have $y_i = x_i \oplus x_{i+1}$ for $i = 0, 1, \ldots, \lg N - 1$, letting $x_{\lg N} = 0$ in the boundary case. The characteristic matrix for $N = 64$ is

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

  and the complement vector is 0.

By Lemma 1, all compositions and inverses of the above BMMC permutations are also BMMC.