Don't Be Too Clever:

Routing BMMC Permutations on the MasPar MP-2

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

Kristin Bruhl†
Monitor Company
Cambridge, Massachusetts
kristin@monitor.com

*Supported in part by funds from Dartmouth College and in part by the National Science Foundation under Grant CCR-9308667.

†Work performed while the author was at Dartmouth College [3].

A preliminary version of this work appeared in the 7th Annual ACM Symposium on Parallel Algorithms and Architectures, July 1995.

Send correspondence concerning this paper to Tom Cormen, Department of Computer Science, Dartmouth College, 6211 Sudikoff Laboratory, Hanover, NH 03755-3510. Email address: thc@cs.dartmouth.edu.

Abstract

The authors implemented and measured several methods to perform BMMC permutations on the MasPar MP-2. Each method is coded in C and MPL and uses only the global router to transfer data among processors. Implementation in MPL limits the amount of control the programmer has over the global router.

The methods used were a naive method; the PE method, which adapts the block BMMC algorithm for parallel disk systems to treat processor elements as independent devices; and the cluster method, which adapts the block BMMC algorithm to treat clusters of processor elements as independent devices. The authors also implemented variations of the first two methods to compute virtual-processor numbers in Gray-code order.

Our results indicate that for random BMMC permutations, the best method overall is the naive method with the Gray-code variation. The PE methods are almost as good, especially when each processor element has several items to route. For some permutations, however, the naive method performs very poorly; the best method in these cases is the PE method with the Gray-code variant.

Although one might expect the cluster method to be best, since it minimizes congestion at the global router ports, this method turns out to be the worst overall, and it is always much worse than the PE method. If it had been coded at a lower level than MPL, the cluster method may well have turned out to be the best in many cases.

1 Introduction

Permuting is a basic data-movement operation and one for which routing networks of most parallel computers are designed. This paper describes the results of attempts to optimize a specific class of permutations, BMMC (bit-matrix-multiply/complement) permutations, on the MasPar MP-2.

The authors felt that this investigation was worth undertaking for three reasons:

1. BMMC permutations comprise a small but useful subset of all permutations. This class includes matrix transposition (when all dimensions are powers of 2), bit reversal, vector reversal, matrix reblocking, Gray code, and inverse Gray code permutations.

2. BMMC permutations have compact descriptions. Any BMMC permutation on N elements is fully described by a $\lg N \times \lg N$ matrix of bits along with a vector of $\lg N$ bits. This representation, which requires only $\lg^2 N + \lg N$ bits, is quite compact compared to a vector of N unique target indices, which takes $N \lg N$ bits. A program that needs to perform a BMMC permutation may find it faster and less space consuming to generate the compact representation than to generate the full target-index vector.

3. The BMMC algorithms for parallel disk systems with a block size of 1 adapt quite easily to in-memory problems on the MasPar MP-2. The primary goal of this investigation was to determine whether these adaptations are worthwhile.

This study compares five algorithms. They differ primarily in the order in which they attempt to admit data to the MP-2's global router. Each physical processor element, or PE, acts as a number of virtual processors. The algorithms go through the virtual processors of each PE, and send data into the global router, in different orders. We summarize the five algorithms here:

1. The *naive method* loops sequentially through the virtual processors, sending one element per PE through the global router in each iteration.

2. The *Gray-code naive method* is like the naive method, but it computes virtual-processor numbers according to a Gray code rather than sequentially. This method speeds up target virtual-processor calculation in that each target virtual-processor number is formed from the previous one by a simple calculation.

3. The *PE method* adapts a block BMMC algorithm [5, 6] that was originally designed for parallel disk systems to treat the PEs, rather than disks, as independent devices. It produces less congestion at ports of the global router than the naive method.

4. The *Gray-code PE method* is like the PE method, but it uses the Gray-code technique to compute virtual-processor numbers.

5. The *cluster method* also adapts a block BMMC algorithm, but it treats clusters of PEs (rather than individual PEs) as independent devices. It produces even less congestion at router ports than the PE method.

The machines used in this study were two different models of MasPar MP-2. One (technically a DECmpp 12000/Sx 2000) has 2K PEs, and the other has 16K PEs. Each run of each method measured the time to perform a given BMMC permutation. The input BMMC permutations were generated in two ways. Most of the runs were on randomly generated BMMC permutations. A few runs, however, were on BMMC permutations that were specially designed to demonstrate the limitations of the naive methods and the comparative advantages of the PE and cluster methods.

Briefly, our results are as follows. For random BMMC permutations, the Gray-code naive method was best overall. Certain BMMC permutations—bit-reversal and particular matrix transpositions for example—elicit very poor performance from the naive method, however. The Gray-code PE method turned out to be the best for these permutations. Moreover, on random permutations, the Gray-code PE method was almost as good as the Gray-code naive method, especially when there are several virtual processors per PE. The cluster method, which the authors expected *a priori* to be the fastest, turned out to be by far the slowest; it is perhaps a little too clever.

The results might have been rather different had the authors been able to program the MasPar MP-2 at a different level. All our code is written in either C or MPL [9], a data-parallel derivative of ANSI C used on the MP-2. Ultimately, decisions about the order in which data are actually admitted to the global router are made by low-level system code. Because the PE and cluster methods are designed to reduce congestion at router ports, they might have beaten the naive method more often if they could have been coded at a low level. In fact, the cluster method might have beaten all others.

One interpretation of the results is that machine designers should provide for a finer degree of control over the machine by users willing to exercise that control. Another interpretation is that one can simply outsmart oneself. The authors found that it was worthwhile to be a little clever, but being too clever—introducing too much complexity into the algorithm—was unprofitable. This lesson is one that many system and algorithm designers have learned through experience.

To the best of our knowledge, this is the first study of BMMC permutation methods on a machine such as the MasPar MP-2, and it is the first experimental study of BMMC permutation methods on any platform. Algorithms have appeared for performing BMMC permutations on hypercubes [2, 11], meshes [17], Omega networks[1] [13], and expanded delta networks [19], and on parallel disk systems [8]. Scherson and Subramanian [16] developed a method for performing arbitrary permutations on restricted-access expanded delta networks. An algorithm for the MasPar MP-2 falls out as a special case. The time to compute the communication patterns is so high, however, that their method is practical only as an off-line algorithm. The high cost is due in part to their having to solve a k-edge-coloring problem on a bipartite multigraph. The algorithms in the present paper, by contrast, are designed to be run on-line.

The remainder of this paper is organized as follows. Section 2 defines BMMC permutations, presents several examples, describes their representation, and discusses the advantages of Gray-code ordering. Section 3 describes the relevant points of the architecture of the MasPar MP-2. The authors examined five methods of performing BMMC permutations (the naive, PE, and cluster methods, and Gray-code variants of the first two), and Section 4 describes each of them. Section 5 presents and analyses empirical timing results for the five methods. Finally, Section 6 offers some concluding remarks.

2 BMMC permutations

In this section, we define BMMC permutations and examine some commonly occurring subclasses. We then discuss how to represent a BMMC permutation and the advantages of our representation scheme.

A *permutation* is a one-to-one mapping of a *source vector* to a *target vector*. A *BMMC permutation* on N elements, where N is an integer power of 2, is defined by an affine transformation

[1]The algorithm in [13] is for linear permutations, which are a large subclass of BMMC permutations.

over $GF(2)$.[2] We have a $\lg N \times \lg N$ *characteristic matrix* A, whose entries are 0 or 1, and a *complement vector* c of $\lg N$ bits. The least significant bits appear on the top and left. For convenience, let us denote $\lg N$ by n. If we interpret each index in the set $\{0, 1, \ldots, N-1\}$ as a vector of n bits, we compute the *target index* y that a *source index* x maps to by the equation

$$y = Ax \oplus c \,, \tag{1}$$

where the matrix-vector multiplication Ax is performed over $GF(2)$. (Matrix-vector multiplication over $GF(2)$ is like standard matrix-vector multiplication over the integers but with all arithmetic performed modulo 2. Equivalently, bit multiplication is logical-and, and bit addition is exclusive-or.) As long as the characteristic matrix A is nonsingular, or invertible, over $GF(2)$, equation (1) defines a permutation over the set $\{0, 1, \ldots, N-1\}$.

As an example, consider a 16×4 matrix stored in row-major order. Assuming 0-origin indexing throughout, the (i, j) element appears in linear position $4i + j$. We can form the linear index number by concatenating the $\lg 16 = 4$ bits of the row number as the most significant bits and the $\lg 4 = 2$ bits of the column number as the least significant bits. To transpose this matrix, the (i, j) entry maps to the (j, i) entry of the transpose. The linear position becomes $16j + i$; this position is the concatenation of the 2 bits of the original column number (but now as the most significant bits) and the 4 bits of the original row number (now as the least significant bits). Given a source index $(x_0, x_1, x_2, x_3, x_4, x_5)$ (least significant bits appearing on the left), the corresponding target index is $(x_2, x_3, x_4, x_5, x_0, x_1)$. We represent this permutation by the following version of the equation $y = Ax \oplus c$ for $N = 64$, where least significant bits are at the top and left of a matrix or vector:

$$\underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}}_{x} \oplus \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{c} = \begin{bmatrix} x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_0 \\ x_1 \end{bmatrix} .$$

Here, the characteristic matrix is a permutation matrix (one 1 in each row and in each column) and the complement vector is all 0s. Note that there are two matrices—quite different from each

[2] BMMC permutations are also known as *affine permutations* [11, 17] or *linear-complement permutations* [2].

other—in this example. The matrix of data is 16×4 (and hence it has 64 entries), and the type of each entry is unspecified. The characteristic matrix A is 6×6 (since $\lg 64 = 6$), and its entries are 0 or 1.

We can express many other familiar permutations in the BMMC form:

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

- In a *bit-reversal permutation*, used in the fast Fourier transform, $y_i = x_{n-i-1}$ for $i = 0, 1, \ldots, n-1$. The corresponding characteristic matrix has 1s on the antidiagonal and 0s elsewhere, and the complement vector is all 0s.

- In a *vector-reversal permutation*, the ith 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 $i = 0, 1, \ldots, n-1$. Here, the characteristic matrix is the identity matrix and the complement vector is all 1s.

- In a *Gray-code permutation*, the binary representation of the ith target index differs from the binary representation of the $(i-1)$st target index in only one bit position. In the standard binary reflected Gray code, we have $y_i = x_i \oplus x_{i+1}$ for $i = 0, 1, \ldots, n-1$, letting $x_n = 0$ in the boundary case. The characteristic matrix for $N = 64$ is

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

  and the complement vector is all 0s.

- In the inverse of the above Gray-code permutation, we have $y_i = x_i \oplus x_{i+1} \oplus \cdots \oplus x_{n-1}$. The characteristic matrix for $N = 64$ is

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

and the complement vector is all 0s. Note that matrices (2) and (3) are inverses of each other over $GF(2)$.

By restricting the characteristic matrix of a BMMC permutation to be a permutation matrix—having exactly one 1 in each row and each column—we obtain the class of *bit-permute/complement*, or *BPC*, permutations.[3] The class of BPC permutations includes the matrix-transposition (with power-of-2 dimensions), shuffle, unshuffle, bit-reversal, and vector-reversal permutations described above. The Gray-code permutation permutation and its inverse are BMMC, but not BPC, permutations.

Representation

It is most convenient to store characteristic matrices in column-major order. We store a bit vector x in a computer word in the obvious way: if x_i is the ith bit of x, then x_0 is stored in the least significant bit, and so on. Viewing a matrix as a set of columns and each column as a bit vector, we represent a matrix in the languages C and MPL as an array of n words, where each word is a column with the least significant bit on top. The advantage of this representation becomes apparent when we examine the C code to perform the computation $y = Ax \oplus c$:

```
for (y = 0, column = A, s = x; s; column++, s >>= 1)
  if (s & 1)
    y ^= *column;

y ^= c;
```

This code takes advantage of the product Ax being equal to $x_0A_0 \oplus x_1A_1 \oplus \cdots \oplus x_{n-1}A_{n-1}$, where A_i is the ith column of A. If we can pack all bits into a word, we can perform matrix-vector multiplication over $GF(2)$ in $O(\lg N)$ word operations.

Gray-code ordering of source indices

The column-major representation has one additional advantage. Suppose we wish to compute the target index corresponding to each source index, but we are free to choose any order for the N source

[3]BPC permutations, described by Nassimi and Sahni [15] are also known as *dimension permutations* [12] or, when the complement vector is 0, *rational permutations* [1].

indices. This situation is in fact similar to what we encounter in performing BMMC permutations on the MasPar MP-2.

We can compute all N target indices with only $O(N)$ word operations rather than the $O(N \lg N)$ we would expect from performing N matrix-vector multiplications on the N source indices. The idea is to vary the source indices by only one bit at a time. In other words, examine the source indices in a Gray-code order. By doing so, we need only $O(1)$ word operations to compute a target index, given the previous one.

In particular, suppose we have already computed $y = Ax \oplus c$, and we wish to compute $y' = Ax' \oplus c$, where x and x' differ only in the ith bit. Then, $x'_i = x_i \oplus 1$ and $x'_j = x_j$ for all $j \neq i$, and so

$$\begin{aligned}
y' &= x'_0 A_0 \oplus x'_1 A_1 \oplus \cdots \oplus x'_i A_i \oplus \cdots \oplus x'_{n-1} A_{n-1} \oplus c \\
&= x_0 A_0 \oplus x_1 A_1 \oplus \cdots \oplus (x_i \oplus 1) A_i \oplus \cdots \oplus x_{n-1} A_{n-1} \oplus c \\
&= x_0 A_0 \oplus x_1 A_1 \oplus \cdots \oplus x_i A_i \oplus A_i \oplus \cdots \oplus x_{n-1} A_{n-1} \oplus c \\
&= Ax \oplus A_i \oplus c \\
&= y \oplus A_i \ .
\end{aligned}$$

Thus, once we know the bit position i for which x and x' differ, we can compute $y' = y \oplus A_i$, using only $O(1)$ word operations.

The above discussion assumes that we know the bit position i at which x and x' differ. It is not too difficult to generate these positions for the standard binary reflected Gray-code ordering. If $y = \text{Gray}(x)$ and $y' = \text{Gray}(x + 1)$, then the lone bit position i at which y and y' differ is the position of the rightmost 1 in the binary representation of $x + 1$. It is easy to write C code to find this bit position:

```
for (s = x+1, i = 0; !(s & 1); s >>= 1, i++)
  ;
```

Over all values of x from 0 to $N - 2$, this code takes a total of only $O(N)$ time, which we can see by a simple aggregate analysis. Of the N integers from 0 to $N - 1$, exactly $N/2^i$ have the rightmost 1 in the ith position, for $i = 0, 1, \ldots, N - 1$. The total time for this code, therefore, is bounded above by

$$\begin{aligned}\sum_{i=0}^{n-1} \frac{N}{2^i} \cdot O(i) &< O\left(N \sum_{i=0}^{\infty} \frac{i}{2^i}\right) \\ &= O(N)\end{aligned}$$

using the identity $\sum_{i=0}^{\infty} ia^i = a/(1-a)^2$ [7, p. 44] with $a = 1/2$. Thus, the amortized time to compute each target index is $O(1)$.

3 Machine architecture

In this section, we describe the MasPar MP-2, paying special attention to its array of parallel processors [10]. For concrete examples, we will use the 2048-processor machine (named *cascade*) installed at Dartmouth. (The machine used at Dartmouth was actually a DECmpp 12000/Sx 2000, which is identical to a MasPar MP-2. We use the MasPar MP-2 name in this paper because of its wider recognition.)

Overall architecture

The MasPar MP-2 is a massively parallel processing system, made up of a *console system* and a *data parallel unit* (DPU). The console is a workstation providing standard I/O devices. For *cascade*, the console system is a DECstation 5000/240.

The DPU is composed of an *array control unit*, or *ACU*, an array of *processor elements*, or *PEs*, and a PE communication system. The number of PEs is a power of 2 in the range 1K to 16K. We denote the number of PEs by P; in *cascade*, $P = 2048$. The PEs are arranged in a two-dimensional grid with an aspect ratio of 1:1 or 1:2, depending on whether P is an even or odd power of 2. The ACU is a serial processor in its own right, and it acts as a controller for the PE array. All parallel processing takes place within the DPU.

Code can execute in any one of three places:

- All code operating on parallel (or, in MPL terminology, *plural*) data executes on the processors of the PE array.
- Code operating on *singular* (i.e., non-parallel) data within modules of parallel code executes on the ACU.

- Modules of singular code may execute on the console.

The console is faster than the ACU, and the ACU is faster than the individual PEs. The MPL programming environment includes library functions to copy data between the console and the ACU and between the console and the PE array; these copying functions incur a slight overhead. In general, therefore, long stretches of singular code are best run on the console, short stretches of singular code within parallel code are best run on the ACU, and inherently parallel code is best run within the PE array.

Each PE has its own processor and data memory; *cascade* has 64K bytes per PE, which was the maximum supported by the MP-2 architecture when the machine was installed. When the ACU sends an instruction to the PEs, each PE carries it out only on data that reside physically in that PE. If a computation requires data from two or more PEs, the PE communication system must send the data to a common PE so that the PE can perform the operation.

The MP-2 includes an I/O subsystem as well, but it is irrelevant to the study in this paper.

Communication

The PEs are arranged in a two-dimensional grid, which is either square or has twice as many columns as rows, depending on the number of PEs. The PEs in *cascade* form a 32×64 grid. Each nonoverlapping 4×4 square subgrid within the PE array forms a *cluster*. We denote the number of clusters by C, so that $C = P/16$; *cascade* has $C = 128$ clusters.

The MP-2 contains two networks for routing data among the PEs:

- The *X-Net* connects each PE to its eight immediate neighbors in the two-dimensional grid.

- The *global router* routes data in arbitrary communication patterns.

The X-Net is faster than the global router. Every PE has a direct connection to its eight immediate neighbors through the X-Net, and all PEs can transmit data via the X-Net simultaneously. The X-Net, therefore, is free of congestion.

The global router, on the other hand, is prone to congestion at its ports and internally. Congestion at its ports is quite common, for the global router has only one input port and one output port per cluster. If more than one PE in a cluster is sending or receiving data through the global router, communications for these PEs are "serialized in a deterministically repeatable but unspecified way"

[9, p. 1-30]. (The MPL reference manual uses this description only for one-to-many or many-to-one connections among PEs, but it is true for communications among clusters, too.) The serialization order is decided by the system; the programmer has no control over it other than to make separate requests to the router.

The MP-2 routes a set of data through the global router by performing a series of *router cycles*. Each router cycle works as follows. From each cluster, the MP-2 selects one PE that still has data to be sent. It then attempts to make a connection from each of the selected sending PEs to its corresponding target PE. Some of the selected PEs will successfully connect to their targets, and some may fail to connect. Those that connect send their data, which completes the router cycle. The next router cycle considers only the remaining sending PEs, that is, those that have yet to successfully connect.

Calls to the router in MPL provide little control over how connections are made. In particular, when there are multiple sending PEs within a cluster, router calls have no way to specify the order in which they attempt to connect to their targets. Moreover, even if only a few PEs attempt to send and they are in different clusters, the corresponding target PEs may include more than one within a given cluster. Because there is only one output port from the router to each cluster, only one sending PE can make a connection into a cluster. The cluster method, which we describe in Section 4, ensures that in each router call, each cluster has exactly one sending PE and one target PE. It does so, however, by making decisions at a high level of code—within MPL—rather than in low-level system code. Consequently, it is relatively slow.

When the routing pattern is a permutation among the PEs—each PE sends exactly one item and receives exactly one item—at least 16 router cycles are required because the 16 sending PEs in each cluster make connections to their target processors in turn. Depending on what occurs in other sending clusters, more than 16 router cycles may be needed when there is congestion at the output ports of the global router.

As mentioned above, congestion also may occur internal to the global router. That is, even if only one PE in each cluster is sending or receiving, more than one router cycle may be necessary. It is possible to eliminate such congestion [19]. In practice, however, congestion within the global router occurs infrequently and has little effect.

PE addressing

There are two ways to address individual PEs. In the first way, which we refer to as *grid-oriented* addressing, PEs are numbered in row-major order within the two-dimensional PE grid. In *cascade*, for example, the PEs form a 32×64 grid. The top row contains PEs 0 through 63, the next row contains PEs 64 through 127, and so on. The self-addressing construct `iproc` of MPL indexes PEs in terms of their grid-oriented addresses. In the second way, *cluster-oriented* addressing, one cluster contains PEs 0 through 15, another cluster contains PEs 16 through 31, and so on. To use the global routing network carefully, one must be aware of these two numbering systems. Conversion between them is a bit permutation on the $\lg P$ bits of PE addresses. The exact bit permutation is a function of the machine size. On *cascade*, for example, with 2048 PEs, the bit permutation π that transforms a grid-oriented address to a cluster-oriented address is

j	0	1	2	3	4	5	6	7	8	9	10
$\pi(j)$	0	1	4	6	8	10	2	3	5	7	9

.

4 BMMC permutation algorithms for the MP-2

In this section, we describe the five algorithms the authors implemented for performing BMMC permutations on the MasPar MP-2, and we briefly justify why each one was considered. We include a description of the block BMMC algorithm for independent devices, upon which several of our candidate methods are based.

None of the methods we describe are in-place. That is, each method permutes from a source vector into a separate target vector.

Before describing the algorithms, we first discuss data-layout strategies and give an overview of the types of time costs common to all five methods.

Data layout

When $N > P$, i.e., there are more elements than PEs, each PE acts as a number of *virtual processors.* We can think of each element as having its own virtual processor. If the elements are spread as evenly as possible across the PEs, the PE with the most elements contains $\lceil N/P \rceil$, which we refer to as the *virtual processor ratio*, or *VPR*.

Cut-and-stack

PE 0	PE 1	PE 2	PE 3	PE 4	PE 5	PE 6	PE 7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

Hierarchical

PE 0	PE 1	PE 2	PE 3	PE 4	PE 5	PE 6	PE 7
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

Figure 1: Vector layout of 32 elements on 8 PEs for cut-and-stack and hierarchical virtualization. In both cases, the *VPR* is 4.

Two ways are commonly used to organize data under virtual processing. In either case, as Figure 1 shows, we can view the data as a two-dimensional array with the column corresponding to the PE number and the row as the offset within the PE. To use MasPar terminology, in *cut-and-stack* virtualization, the ith element (indexing from 0) resides in PE $i \bmod P$ and has offset $\lfloor i/P \rfloor$. In *hierarchical* virtualization, the ith element has offset $i \bmod VPR$ and resides in PE $\lfloor i/VPR \rfloor$. Cut-and-stack corresponds to row-major layout, and hierarchical corresponds to column-major. For BMMC permutations, N is a power of 2, and therefore all floors and ceilings in the above expressions disappear.

Our algorithms assume hierarchical virtualization, although they are all easily convertible to cut-and-stack. We will refer to the ith *row* as the set of P elements at offset i across all P PEs. In hierarchical virtualization, an element's $\lg N$-bit index consists of its PE number in the most significant $\lg P$ bits and its row number in the least significant $\lg VPR$ bits.

Costs

We give four time measurements in Section 5 for each method:

1. *Setup time* is the time to precompute any information needed before computing any target indices.
2. *Calculation time* is the time to calculate source and target indices.
3. *Routing time* is the time to route data through the global router.
4. *Total time* is the total time from start to finish.

	Setup time	Calculation time	Router calls
Naive	0	$\Theta\left(\frac{N}{P}\lg N\right)$	N/P
Gray-code naive	$\Theta(\lg P)$	$\Theta(N/P)$	N/P
PE	$\Theta(\lg N \lg P)$	$\Theta\left(\frac{N}{P}\lg N\right)$	N/P
Gray-code PE	$\Theta(\lg N \lg P)$	$\Theta(N/P)$	N/P
Cluster	$\Theta(\lg N \lg P)$	$\Theta\left(\frac{N}{P}\lg N\right)$	$16N/P$

Table 1: Analyzed setup and calculation times and number of router calls for the five algorithms, in terms of N and P. These times assume that matrix columns and vectors are stored in one word and hence word operations are used whenever possible.

We can analyze setup and calculation times as functions of N and P, and we can count the number of calls to the router routines. As discussed in Section 3, we cannot determine the number of router cycles, and hence routing times, based solely on the number of calls to the router routines. Table 1 summarizes what we can analyze about the five algorithms.

Naive method

In the *naive method*, we simply permute row-by-row. That is, we send the rows in the order $0, 1, \ldots, VPR - 1$. To send the ith row, each PE computes the target index corresponding to the source index of its element. The following MPL code, which each PE executes in parallel, gives the idea:

```
plural int source_index, target_index, target_PE, target_row;
int        n, source_row, VPR, v, mask, *A, c, element_size;

for (source_row = 0; source_row < VPR; source_row++)
  {
    source_index = (iproc << v) + source_row;
    target_index = compute_target(source_index, A, c, n);
    target_PE = target_index >> v;
    target_row = target_index & mask;
    pp_rsend(target_PE, &source[source_row], &target[target_row], element_size);
  }
```

Variables declared as `plural` (`source_index`, `target_index`, `target_PE`, and `target_row`) exist in each PE. Other variables are singular and reside in the ACU. The function `compute_target()` performs the matrix-vector multiplication/complement code in Section 2, but with the matrix and vector sizes `n` (which equals $\lg N$) as an additional parameter. The variable `v` equals $\lg VPR = \lg(N/P)$, and `mask` has 1s only in the least significant $\lg VPR$ bits in order to isolate the row bits of the target index. The function `pp_rsend()` performs the actual routing; the rightmost column in Table 1 counts the number of calls to `pp_rsend()`. Its arguments are the PE to route to, the address of the source element within the sending PE, the address of where the element is to end up in the target PE, and the number of bytes to send.

We analyze the naive method as follows. There is no setup time. Target indices are calculated once per row, or $VPR = N/P$ times, and each calculation is a matrix-vector multiplication/complement that takes $\Theta(\lg N)$ word operations, for a calculation time of $\Theta\left(\frac{N}{P}\lg N\right)$ in all. There is one router call per row, or N/P in all. Note that the only property of BMMC permutations that the naive method uses is computing each target index based on the matrix-vector multiplication/complement code.

Gray-code naive method

The *Gray-code naive method* permutes row-by-row but with the target-index calculation improved.

Observe that it doesn't matter in which order the rows are sent; they need not progress in the order $0, 1, \ldots, VPR - 1$. In the Gray-code naive method, we send the rows in Gray-code order. That is, the binary representation of each row number sent differs from the previous row number sent in only one bit. This approach gives the advantage discussed in Section 2: given the previous target index, it takes only $O(1)$ word operations (amortized) to compute the next target index. The Gray-code naive method, therefore, has a calculation time of only $\Theta(N/P)$.

Although it takes $O(\lg N)$ word operations to compute the first target index, we can get the setup time down to only $\Theta(\lg P)$, as follows. If we start with row 0, then only $\lg P$ bits of the initial source indices may be nonzero. The initial target index calculation, therefore, takes only $\Theta(\lg P)$ word operations.

The Gray-code naive method code is the following. The variable `lnproc` is builtin to MPL, and it is equal to $\lg P$.

```
plural int target_index, target_PE, target_row;
int        n, source_row, k, s, j, gray_bit, VPR, v, mask, *A, c, element_size;

target_index = compute_target(iproc, A+v, c, lnproc);

for (source_row = 0, k = 0; k < VPR; k++)
  {
    target_PE = target_index >> v;
    target_row = target_index & mask;
    pp_rsend(target_PE, &source[source_row], &target[target_row], element_size);

    if (k < VPR-1)
      {
        /* Determine the column j which differs from the current value of
           source_row in the Gray-code order. */
        for (s = k+1, j = 0, gray_bit = 1;
             (s & 1) == 0;
             s >>= 1, j++, gray_bit <<= 1)
          ;

        source_row ^= gray_bit;
        target_index ^= A[j];
      }
  }
```

The Gray-code ordering does not alter the number of calls to `pp_rsend()`, and so this method makes $\Theta(N/P)$ calls.

Block BMMC algorithm for independent devices

The remaining methods adapt the block BMMC algorithm [5, 6] for parallel disk systems. This algorithm performs BMMC permutations on data that reside on D independent devices. By *independent*, we mean that we can access data from a different location within each device simultaneously. The block BMMC algorithm also requires that the "block size" be 1. That is, elements can be accessed one at a time from a given device rather than being grouped into blocks of elements, as occurs for example on disks.

Although this algorithm was developed for parallel disk systems, we can view the PEs or the clusters of the MasPar MP-2 as independent devices just as easily. The number D of devices would then equal P or $P/16$, respectively.

The block BMMC algorithm relies on the notion of a *1-permutable set*, which is a set of D source elements such that

1. each device contains exactly one of these source elements, and
2. each device has exactly one element mapped to it.

We repeatedly permute disjoint 1-permutable sets until the entire source vector has been permuted. We call this sequence of 1-permutable sets a *schedule*.

The algorithm has two components. It first finds some 1-permutable set. It then generates a schedule for the entire permutation, based on having any 1-permutable set to start. In the following description of these components, we denote the jth bit of an n-bit index x by x_j, and we denote a range of bits using the notation "..". Recall that $n = \lg N$; we also adopt the notation that $d = \lg D$. Because we assume hierarchical virtualization, bits $x_{n-d..n-1}$ represent the independent device number and bits $x_{0..n-d-1}$ represent an index within the independent device for a given index x. For example, if the PEs are the independent devices, then if we let $v = \lg \mathit{VPR} = \lg(N/P)$, so that $d = \lg P$ and $v = n - d$, bits $x_{v..n-1}$ represent the PE number and bits $x_{0..v-1}$ represent the row number. We won't prove the correctness of this method here; see [5, 6] for a proof.

We find a 1-permutable set in four steps:

1. Find a set S of "basis" columns for the last d rows of A. That is, find a set $S \subseteq \{n-d, n-d+1, \ldots, n-1\}$ such that the submatrix $A_{n-d..n-1,S}$ is nonsingular. (This submatrix of A consists of the intersection of rows $n-d, n-d+1, \ldots, n-1$ and columns in S.) Define $Q = A_{n-d..n-1,S}$ so that the matrix Q^{-1} exists.
2. Given the set S, define three sets of columns:

$$T = \{n-d, n-d+1, \ldots, n-1\} - S \ ,$$

$$
\begin{aligned}
U &= S \cap \{n-d, n-d+1, \ldots, n-1\} \ , \\
V &= \{0, 1, \ldots, n-1\} - (S \cup T) \ .
\end{aligned}
$$

3. Given T and U, define a permutation R on $\{0, 1, \ldots, D-1\}$. See [5, 6] for details.

4. Finally, we compute a 1-permutable set $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$ of source indices as follows:

$$
\begin{aligned}
x_S^{(i)} &= Q^{-1} A_{n-d..n-1,T} \ \mathrm{bin}_T(i) \oplus \mathrm{bin}(R(i)) \oplus c_{n-d..n-1} \ , \\
x_T^{(i)} &= \mathrm{bin}_T(i) \ , \\
x_V^{(i)} &= 0
\end{aligned}
$$

for $i = 0, 1, \ldots, D-1$. Here, $x_S^{(i)}$ denotes the bit positions indexed by the set S in $x^{(i)}$ (similarly for subsets T and V), $\mathrm{bin}(i)$ is the d-vector given by the binary representation of i, and $\mathrm{bin}_T(i)$ is a vector of length $|T|$ given by the bits of $\mathrm{bin}(i)$ indexed by T.

We construct a schedule given any 1-permutable set $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$ (this set need not be the one produced by the above method) as follows. Define the set of N/D n-vectors $\{m^{(0)}, m^{(1)}, \ldots, m^{(N/D-1)}\}$ such that $m_{n-d..n-1}^{(i)} = 0$ and $m_{0..n-d-1}^{(i)}$ is the binary representation of i, for $i = 0, 1, \ldots, D-1$. Then the schedule consists of the N/D 1-permutable sets

$$
\begin{array}{c}
\{x^{(0)} \oplus m^{(0)}, x^{(1)} \oplus m^{(0)}, \ldots, x^{(D-1)} \oplus m^{(0)}\} \ , \\
\{x^{(0)} \oplus m^{(1)}, x^{(1)} \oplus m^{(1)}, \ldots, x^{(D-1)} \oplus m^{(1)}\} \ , \\
\vdots \\
\{x^{(0)} \oplus m^{(N/D-1)}, x^{(1)} \oplus m^{(N/D-1)}, \ldots, x^{(D-1)} \oplus m^{(N/D-1)}\} \ .
\end{array}
\tag{4}
$$

We may permute these sets in any order.

PE method

In the *PE method*, we adapt the block BMMC algorithm to treat the P PEs as the independent devices. That is, $D = P$ in the above block BMMC algorithm. There are N/P 1-permutable sets, and they are routed in the order given by (4). Hence the number of router calls is N/P.

Although we omit most of the programming details in this paper, there are a couple of significant points. First, note that the first two steps of finding a 1-permutable set deal with relatively small data—a $\lg D \times \lg N$ bit matrix and sets of cardinality at most $\lg N$—and so they were implemented as sequential routines on the console. Because $\lg D = \lg P$, the setup time in our implementation is $\Theta(\lg N \lg P)$.

The second programming detail is an optimization in computing target indices. Observe that because the jth source index computed on the ith PE is $x^{(i)} \oplus m^{(j)}$, we can write the corresponding target index as $A(x^{(i)} \oplus m^{(j)}) = Ax^{(i)} \oplus Am^{(j)}$. The first term, $Ax^{(i)}$, is the same for all rows j on a given PE, and so we need compute it only once per PE. The second term, $Am^{(j)}$, is the same for all PEs for a given row j, and so we compute it on the ACU, which is faster than the PEs. As we shall see in Section 5, this optimization improves our target-index calculations by a constant factor, but the time remains the same as for the naive method, $\Theta\left(\frac{N}{P} \lg N\right)$.

To justify why the authors considered the PE method, we first examine why the naive method may be troublesome. Consider what happens when we send one whole row. Although each PE sends exactly one element, there may be congestion at target PEs. That is, there may be some PE that receives multiple elements. If k elements are destined for a target PE, then at least k routing cycles will be required.

In fact, we can determine the amount of congestion at target PEs based on the characteristic matrix of the BMMC permutation. We start by defining the *range* of a $p \times q$ matrix A with 0-1 entries by

$$\mathcal{R}(A) = \{y : y = Ax \text{ for some } x \in \{0, 1, \ldots, 2^q - 1\}\} \ .$$

That is, $\mathcal{R}(A)$ is the set of p-vectors that can be produced by multiplying all q-vectors with 0-1 entries (interpreted as integers in $\{0, 1, \ldots, 2^q - 1\}$) by A over $GF(2)$. We also adopt the notation

$$\mathcal{R}(A) \oplus c = \{z : z = y \oplus c \text{ for some } y \in \mathcal{R}(A)\} \ .$$

That is, $R(A) \oplus c$ is the exclusive-or of the range of A and a fixed vector c.

From [8], we have the following lemma.

Lemma 1 *Let A be any $p \times q$ matrix whose entries are drawn from $\{0, 1\}$, let c be any p-vector whose entries are drawn from $\{0, 1\}$, and let $r = \operatorname{rank} A$. Then $|\mathcal{R}(A) \oplus c| = 2^r$.* ■

Using this lemma, we can compute the amount of congestion at receiving PEs with the naive methods.

Theorem 2 *Let r be the rank of the lower right $\lg P \times \lg P$ submatrix of the characteristic matrix of a BMMC permutation. When the naive method or Gray-code naive method sends a row, the number of distinct PEs receiving elements from that row is 2^r.*

Proof: With hierarchical virtualization, bits $\lg VPR, \ldots, \lg N-1$ give the PE number of an index. Let us write the characteristic matrix A as

$$A = \begin{array}{cc} \begin{array}{cc} \lg VPR & \lg P \end{array} & \\ \left[\begin{array}{c|c} \alpha & \beta \\ \hline \gamma & \delta \end{array}\right] & \begin{array}{c} \lg VPR \\ \lg P \end{array} \end{array} ,$$

so that, for example, δ denotes the lower right $\lg P \times \lg P$ submatrix. Given a source index x, the PE number of the corresponding target index y is given by

$$y_{\lg VPR..\lg N-1} = \gamma\, x_{0..\lg VPR-1} \oplus \delta\, x_{\lg VPR..\lg N-1} \oplus c_{\lg VPR..\lg N-1} ,$$

where c is the complement vector. A given row of the source vector has source indices x such that $x_{0..\lg VPR-1}$ is fixed and $x_{\lg VPR..\lg N-1}$ takes on all P possible bit values. The vector given by the expression $\gamma\, x_{0..\lg VPR-1} \oplus c_{\lg VPR..\lg N-1}$ is fixed for a given source row, and so it acts as the complement vector for the row. We apply Lemma 1 to conclude that the number of distinct values of $y_{\lg VPR..\lg N-1}$ is equal to 2^r, where $r = \operatorname{rank} \delta$. ∎

By Theorem 2, the higher the rank of the lower right $\lg P \times \lg P$ submatrix, the more spread out the target PEs are for a given source row. Conversely, if the rank of this submatrix is low, the naive methods generate congestion at the target PEs.

The PE method, by contrast, decomposes the BMMC permutation into a sequence of 1-permutable sets. In so doing, it guarantees that in each send, each PE sends exactly one element and receives exactly one element. Therefore, there can be no congestion at the PEs.

Gray-code PE method

The *Gray-code PE method* is the same as the PE method but with a different ordering of the 1-permutable sets. In particular, rather than XORing the $x^{(i)}$ with $m^{(0)}, m^{(1)}, \ldots, m^{(VPR-1)}$ in order, we vary the superscripts of the $m^{(j)}$ by Gray-code order.

This approach speeds up the calculation of the target indices in the same way as the Gray-code naive method calculates target indices faster than the naive method. The setup time and number of routing calls are the same as for the PE method, and the calculation time is asymptotically the same as for the Gray-code naive method.

Cluster method

In the *cluster method*, we adapt the block BMMC algorithm to treat the clusters as the independent devices. That is, $D = C = P/16$. Again, we omit the programming details, except to note that we must account for the difference between grid-oriented and cluster-oriented PE addressing.

What potential advantage might the cluster method hold over the PE method? Let us examine an individual send in the PE method. For each sending cluster, in each router cycle, the global router selects one PE from the cluster to actually attempt to send from; it selects C sending PEs in this way. The target PEs for these C elements may not be in C distinct target clusters, however. That is, there may be several selected sending PEs that all send to the same cluster. As we noted in Section 3, within each routing cycle, only one of these sending PEs actually establishes a connection to its target PE. The result is further serialization.

In an individual send in the cluster method, each cluster contains exactly one PE that sends an element, and each cluster contains exactly one PE that receives an element. There can be no congestion at the clusters or PEs. Although it is relatively rare, there may be congestion within the global router itself. (See [18] for the probability distribution of router congestion with the cluster method.)

The cluster method has setup and calculation times that are asymptotically the same as the PE method, since the number of independent devices differs by a constant factor (16). The number of router calls is $16N/P$, since in each router call only one PE per cluster attempts to send.

Although the cluster method cleverly limits congestion at router ports, Section 5 shows that the cluster method is by far the slowest of the five methods implemented by the authors. We believe that this poor performance is caused by low utilization of the PEs. Because the cluster method has 16 times fewer independent devices than the PE method, it routes 16 times as many 1-permutable sets. Each time it routes a 1-permutable set, 15 of the 16 PEs in a cluster are idle. Compared to the PE method, in which serialization is performed by the low-level code that controls the global router, the cluster method serializes at a high level. If the cluster method had been coded at the same low level as the global router congestion-resolution code, it might have been considerably faster.

Because the cluster method was so slow relative to the other methods, the authors chose not to implement a Gray-code version of it. Target-index calculation does not appear to be the bottleneck.

5 Timing results

This section presents timing results for the five algorithms described in Section 4. The authors ran these algorithms on two machines: *cascade*, described in Section 3, and *nibbles*, a MasPar MP-2 with 16K PEs at the NASA Goddard Flight Space Center. We first describe our average-case results, which favor the naive methods for most VPR values. Then we examine two subclasses for which the naive methods perform extremely poorly but the PE-based methods perform well.

Average-case results

To generate average-case timing results, for each machine, we tested VPRs that were powers of 2 ranging from 2^0 to 2^{10}, which was as high as one can go. For each VPR, we randomly generated 100 nonsingular matrices and complement vectors and timed how long each method took on each one. The times clocked for each run were setup time, calculation time, routing time, and the total time.

Some methodological points are in order. First, how were the matrices randomly generated? Each nonsingular matrix was generated by calling the Unix `random()` function $\lg^2 N$ times to fill in a matrix and another $\lg N$ times to fill in a complement vector and then testing the matrix for singularity, repeating until the matrix so generated was nonsingular. Because the Unix `random()` function is considered by some to be a poor pseudorandom number generator, the authors ran the same tests using the Unix `lrand48()` generator. Although we do not present the results of these tests here, they were essentially the same as with `random()`. That is, on average they were always within a few percent (within 2% in most cases) of the `random()`-generated timings, and the relative performance of the five methods was the same.

The second methodological point is how the timings were produced. Each input was run once for each method to determine just an end-to-end total time, and then it was run another time for each method to determine the component times: setup, calculation, and routing. Each run to determine component times includes several calls to the MasPar MP-2 timer function, and each timer call induces some overhead. Test runs determined the timer overheads, which were then subtracted out from the component times.

The third and final methodological point is that, unfortunately, the timer on the MasPar MP-2 sometimes gives erroneous results. Luckily, such results are so blatantly incorrect that they easy to filter out. Our filter removed runs for which any of the following held:

- A measured time was negative. (Several runs had negative component times!)
- A measured time exceeded 100 seconds. As Figures 2–5 show, such times are clear outliers.
- Any of the component times—setup, calculation, or routing—exceeded the total time. One might question whether our filter removed good data in which additive rounding errors caused component times to exceed the total time. Only four runs out of the 500 on *cascade* had a component time between the total time and twice the total time, and these four runs would not have changed the results to any significant degree.

Hardly any end-to-end timings were filtered out: two out of 5500 runs on *cascade* and one out of 5500 on *nibbles*. The component timings produced many more results that failed the filter tests. The timings that passed the filter appeared to be consistent, and there were enough good timings to yield meaningful results.

Figures 2–5 show the average total times for the five methods on both linear and log-log scales. These figures do not show variations in runs for a given VPR, but Figure 6 is a histogram of the standard deviations of the total times on *cascade* as a percentage of the total time for the corresponding problem size. For the random BMMC permutations generated, the Gray-code naive method yields the best times overall. Observe, however, that the Gray-code PE method becomes quite competitive at larger VPRs; its performance is quite close to that of the Gray-code naive method at VPRs 16 and greater. If one needed to select just one of the five methods to run for *random* BMMC permutations on *all* VPRs, the Gray-code naive method would be the method of choice. Observe, however, that for VPRs of at least 8, the average time for the Gray-code PE method is within a factor of 2 of the average time for the Gray-code naive method; for large VPRs,

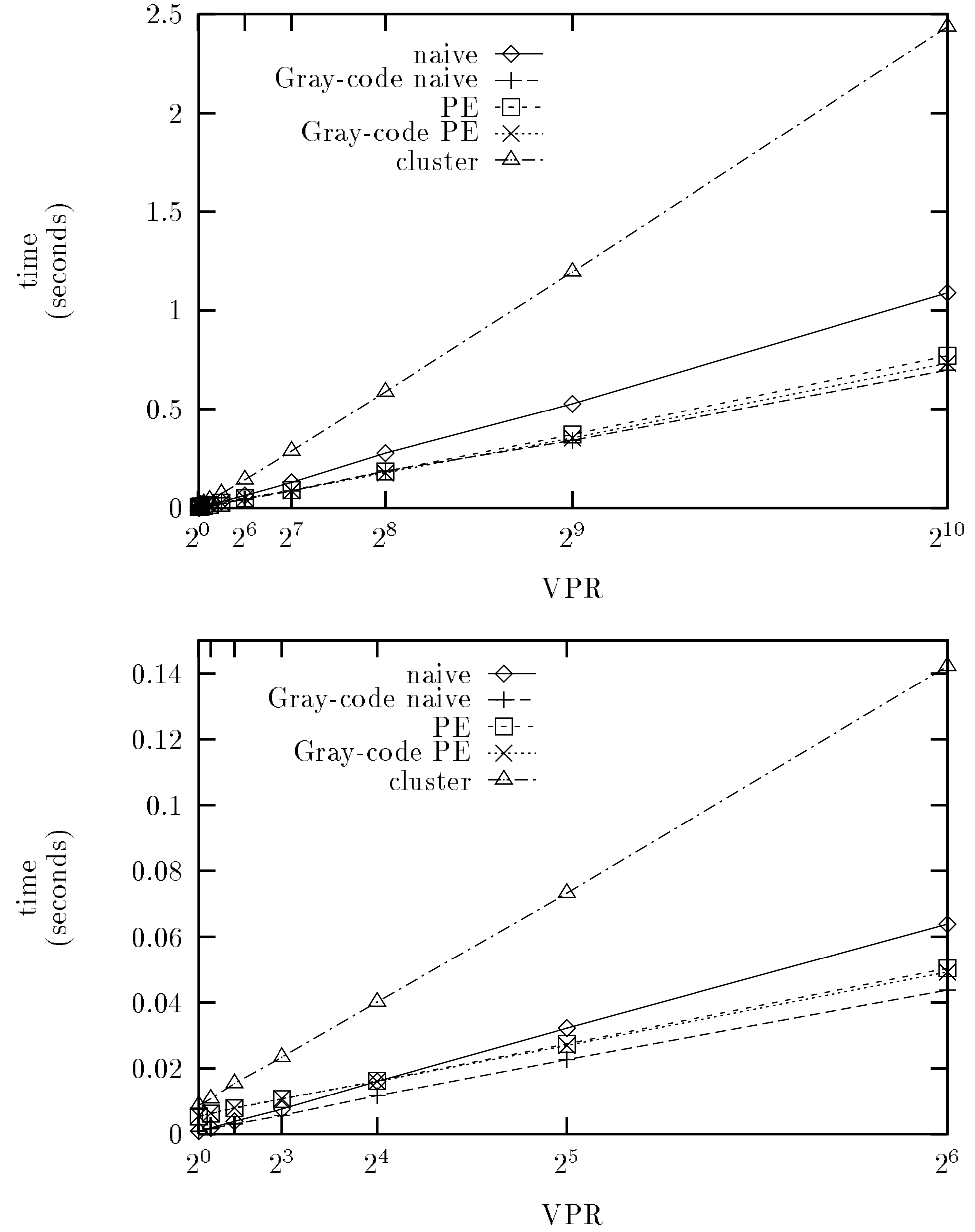


Figure 2: Average total times in seconds on *cascade* for the five methods on a linear scale. The top figure shows VPRs of 2^0 to 2^{10}, and the bottom figure zooms in on VPRs 2^0 to 2^6.

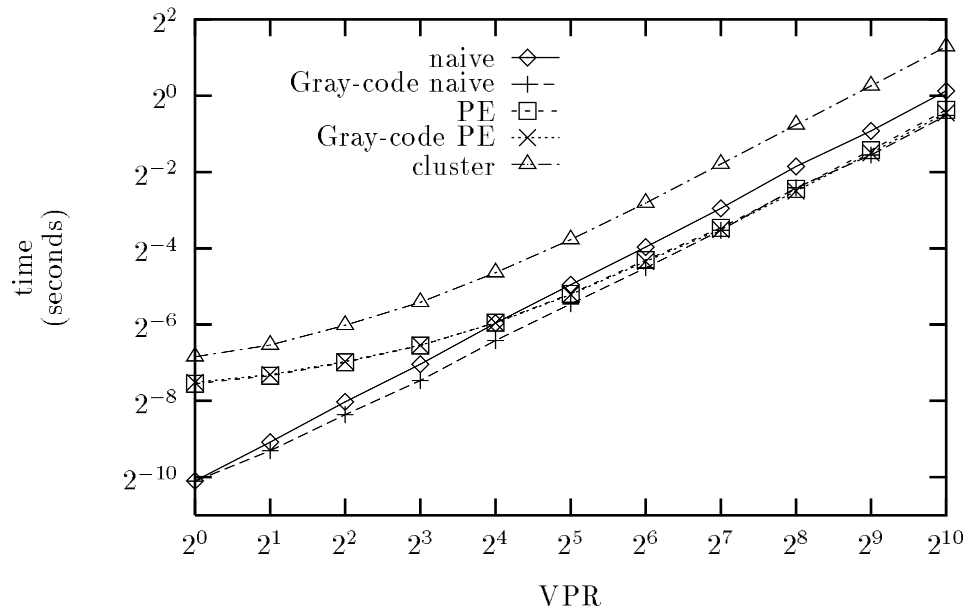


Figure 3: Average total times in seconds on *cascade* for the five methods on a log-log scale.

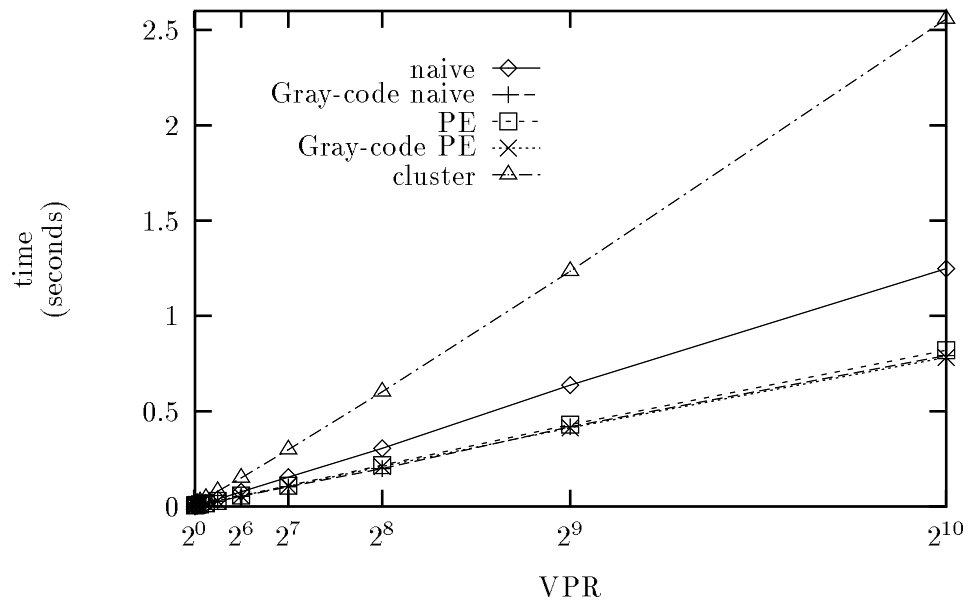
the two methods perform about the same. As we shall see, certain input permutations elicit very poor performance from the naive methods, whereas the PE-based methods are more robust.

As Figures 2–5 show, the timing results on *nibbles* differed very little from those on *cascade* when viewed as a function of the VPR. In the remainder of this section, therefore, we shall examine only data from *cascade*.

Looking at Figure 2, the total times for the five methods appear to be linear functions of the number of the VPRs. A linear regression using the `|STAT` package bears out this hypothesis. The `|STAT` package performs two tests. The F-test determines how likely it is that the observed times are random with respect to the number of elements; a low value means that there is a correlation, with 0 being the highest correlation. The R^2 test gives the quality of a linear fit as a fraction between 0 and 1; the ideal value is 1. In each case, the F-test for each method was 0, and the R^2 values ranged from 0.9994 to 0.9998.

Figure 7 plots the breakdown of the average times used on *cascade* by each of the five methods into the setup times ("setup"), calculation times ("calc"), and the routing times ("route"), along with the total time ("total").

These plots show several characteristics of the methods:



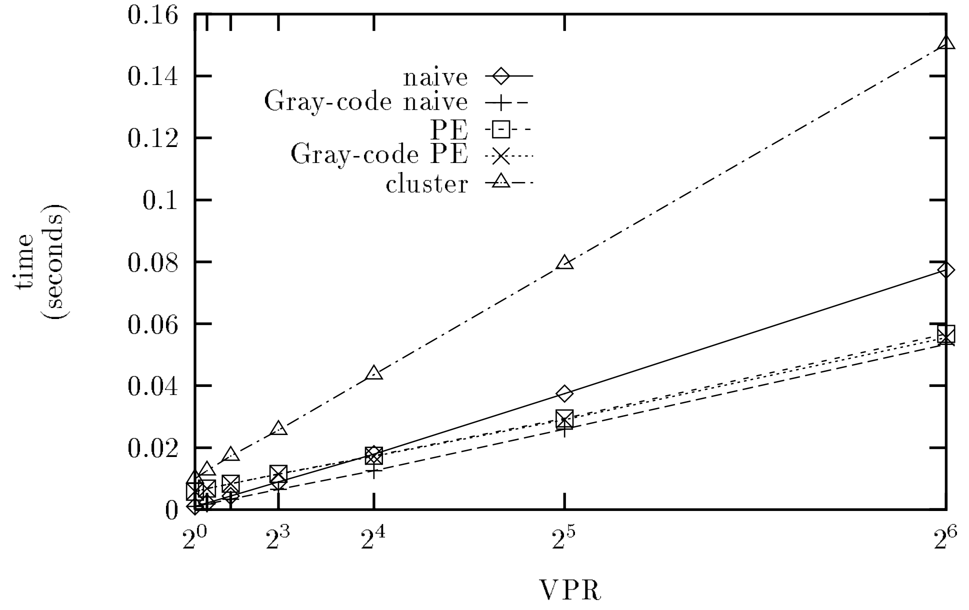


Figure 4: Average total times in seconds on *nibbles* for the five methods on a linear scale. The top figure shows VPRs of 2^0 to 2^{10}, and the bottom figure zooms in on VPRs 2^0 to 2^6.

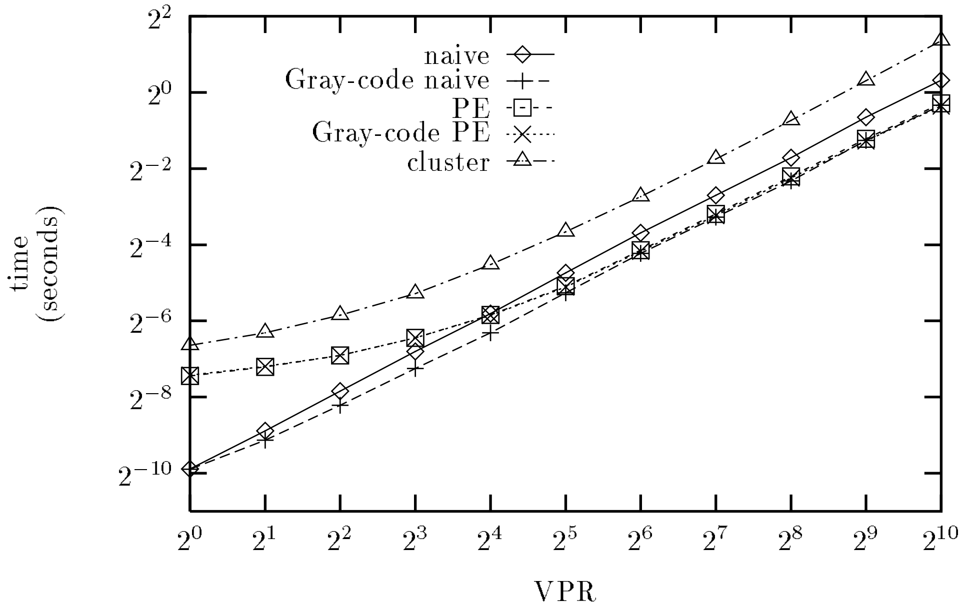


Figure 5: Average total times in seconds on *nibbles* for the five methods on a log-log scale.

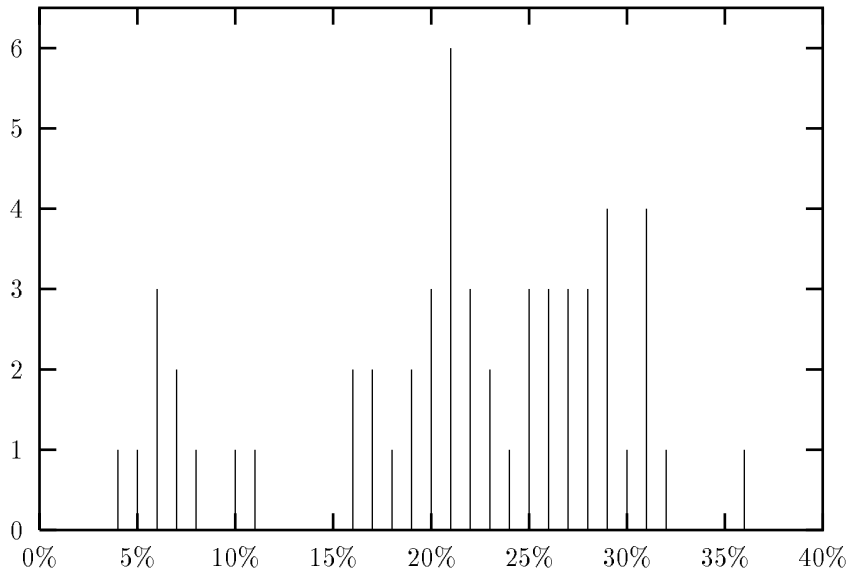


Figure 6: A histogram of standard deviations in total time, expressed as a percentage of the total time. For the purpose of this histogram, all percentages are rounded to the nearest integer.

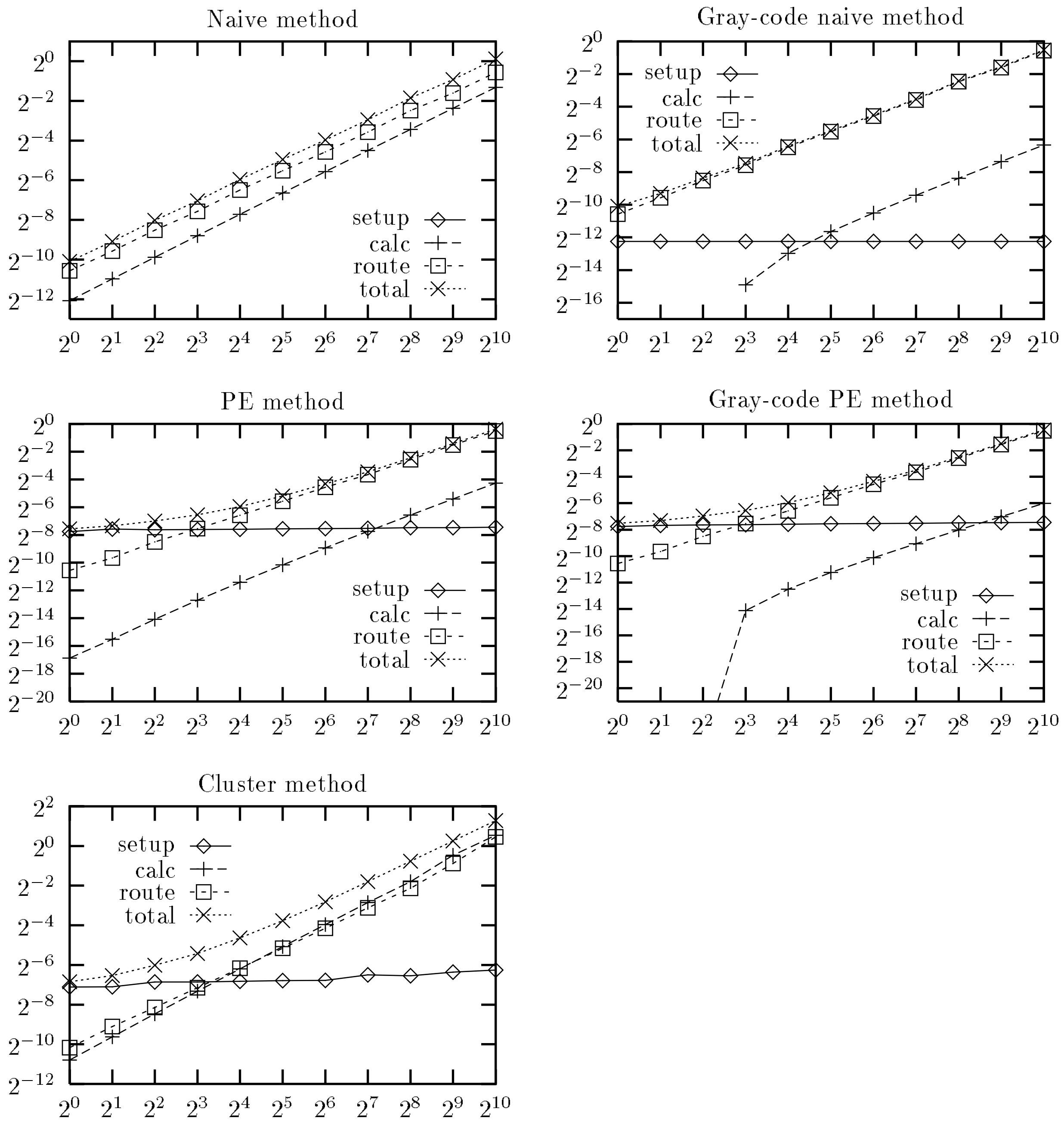


Figure 7: The breakdown of average times for the five methods on *cascade*. Plots are on a log-log scale, with the VPR on the horizontal axis and the time in seconds on the vertical axis. The naive method has no setup time. Calculation times for small VPRs with the Gray-code methods are too low to show.

- Using the Gray-code idea to calculate target PEs and rows reduces the calculation time for the naive method by a factor of approximately 32 for all but the smallest VPRs. The Gray-code technique does induce a slight setup cost. Both methods have the same routing time since they compute the same sets of source-target pairs. Because the naive method is bound by routing time, as is the PE method for large enough VPRs, the advantage of the Gray-code method is slight.

- In the three methods based on the block BMMC algorithm (PE, Gray-code PE, and cluster), the setup time is the dominant cost for small VPRs. In each case, the routing time dominates once the VPR reaches 16.

- The Gray-code idea reduces target-calculation times in the PE method by a factor in the range 2–3.3 for all but the smallest VPRs. The Gray-code technique does not yield as much benefit in the PE method as in the naive method. As explained in Section 4, the PE method already includes an optimization in how it computes target PEs and rows. The Gray-code improvement, therefore, has less effect than in the naive method, in which targets are calculated by a full matrix-vector multiplication in the PEs for each row. As in the naive method, however, both PE-based methods have the same routing time since they compute the same sets of source-target pairs. Again, the routing time dominates, and so the advantage of the Gray-code method is slight.

- The cluster method has high costs in all areas. The setup time is about twice as high in the cluster method because it is more complex than the PE method. Also higher in the cluster method are the calculation (approximately 30 times higher) and routing times (approximately 1.5 times higher) because it routes 16 times as many 1-permutable sets as the PE method. Although each 1-permutable set in the cluster method has no contention for router ports, the cost of achieving this property is just too high.

The actual calculation times followed the asymptotic times from Table 1 quite closely. Figure 8 plots the actual calcuation times against the number of word operations. Note that of the non-Gray-code methods, the PE method has the lowest slope, indicating that the optimizations discussed in Section 4 really do help.

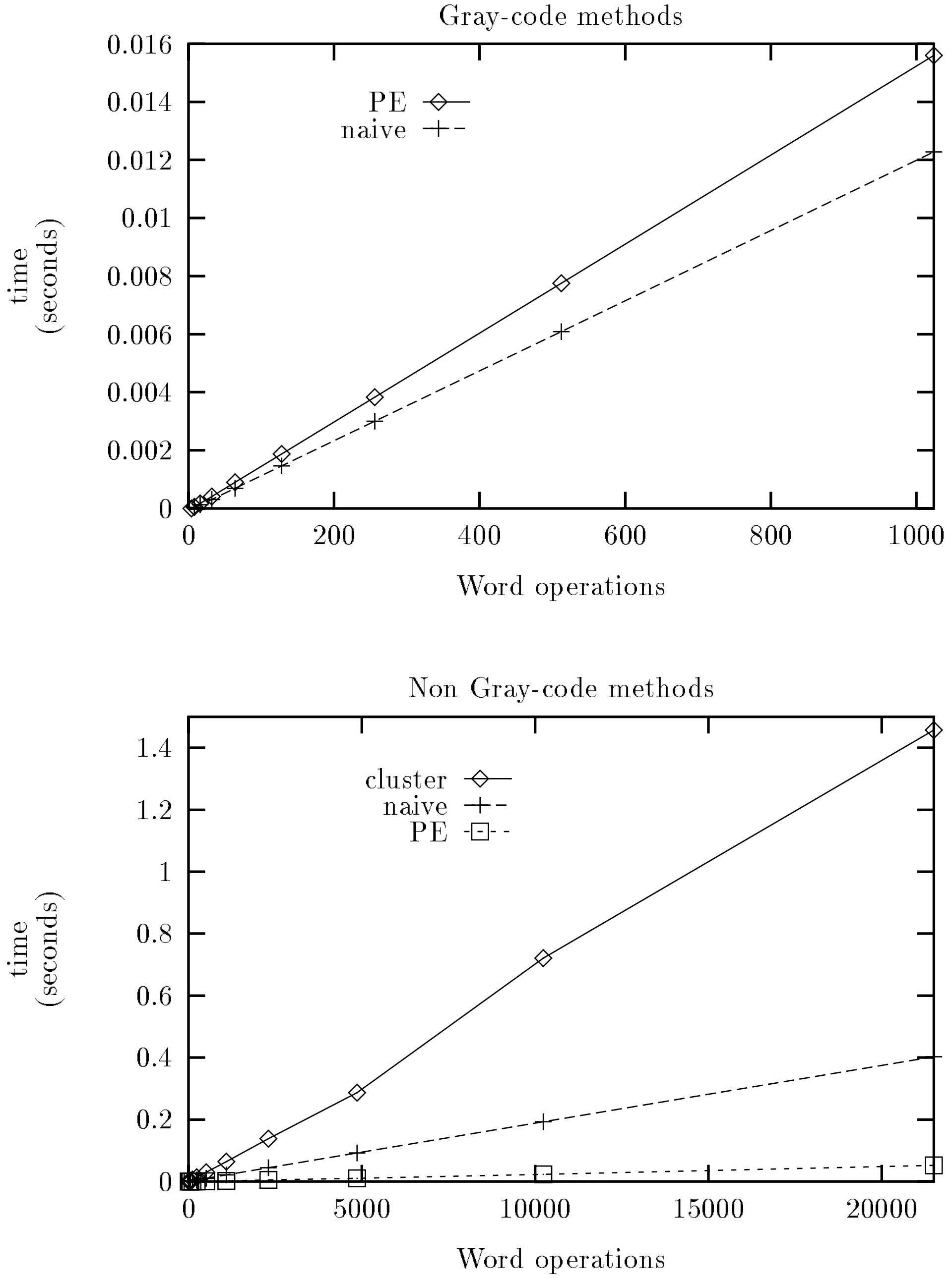


Figure 8: Plots of actual calculation times against the number of word operations for the five methods.

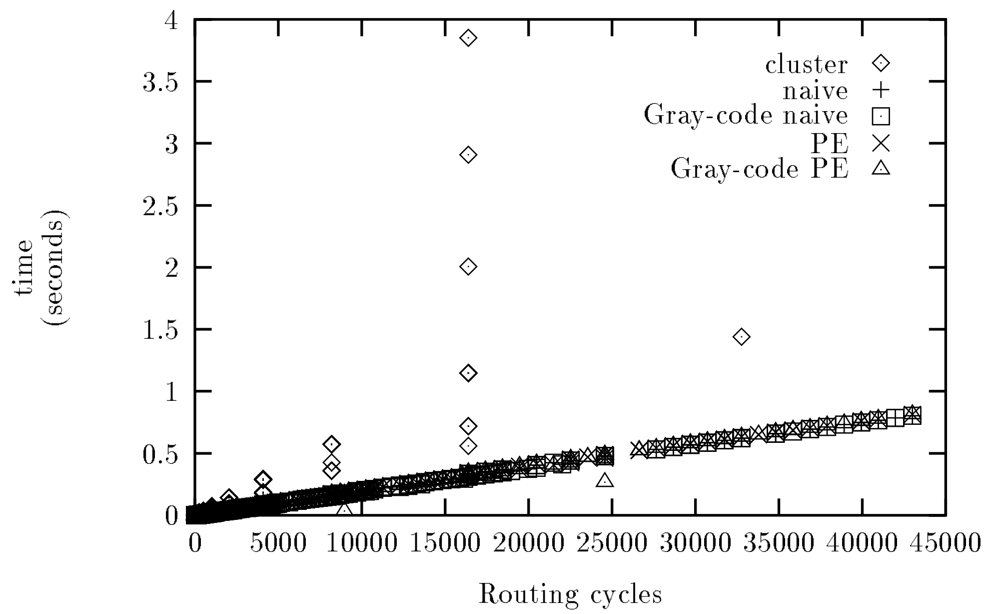


Figure 9: A scatter plot of routing time versus the number of routing cycles for the five methods.

The programs were also instrumented to count routing cycles in addition to measuring the routing time. There is no way to count routing cycles when using the `pp_rsend()` function, however. Using the same data, the authors instead invoked the global router using the `router` construct of MPL; this is the only way known that permits us to count routing cycles. For our situation, in which the sending PE determines the location in the receiving PE of where the data is to go, the `router` construct is unwieldy, and calls to `pp_rsend()` are much cleaner. The authors believe that the number of routing cycles measured using the `router` construct are the same as calls to `pp_rsend()` would use, but we have no way to verify this hypothesis. Figure 9 shows a scatter plot of routing time versus the number of routing cycles for the five methods.[4] For all but the cluster method, the routing time clearly increases linearly with the number of routing cycles. Because the details of the global router are difficult to come by, the authors do not understand why the cluster method exhibits wider variations in routing time for a given number of router cycles.

[4]The code that yielded the times in Figure 9 differs from the code used for all other timings, since it uses the `router` construct rather than calls to `pp_rsend()`. Consequently, routing times in this figure differ from those in Figure 7.

Bad permutations for the naive methods

Although the naive and Gray-code naive methods are good on average, they are susceptible to very poor worst-case behavior. For example, the naive methods are very slow on two subclasses of BMMC permutations—bit-reversal and certain matrix transpositions—but the PE-based methods are much faster.

Recall that the characteristic matrix for a bit-reversal permutation has 1s on the antidiagonal and 0s elsewhere. Figure 10 shows the total time to perform bit-reversal permutations on *cascade*. Although the naive methods are fast at VPRs of 1 and 2, the PE-based methods outperform it for VPRs of 8 and greater. At a VPR of 1024, the Gray-code naive method takes 55.66 seconds whereas the Gray-code PE method takes only 0.928 seconds.

Figure 10 also shows running times for five methods for the transposition of a $VPR \times P$ matrix on *cascade*. The characteristic matrix for such a permutation has the form

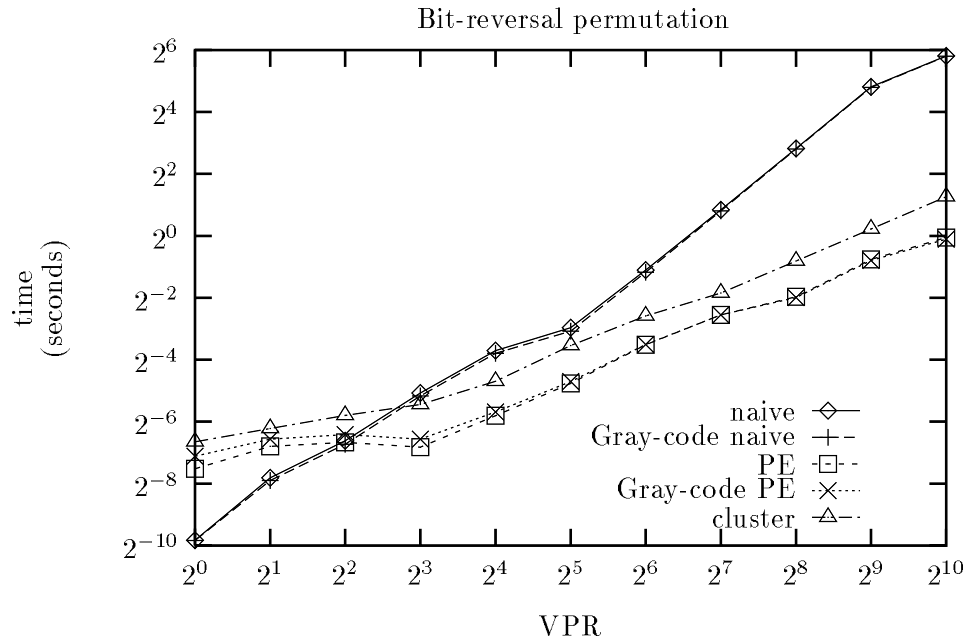
$$\begin{array}{cc} \begin{array}{cc} \lg VPR & \lg P \end{array} & \\ \left[\begin{array}{c|c} 0 & I \\ \hline I & 0 \end{array}\right] & \begin{array}{l} \lg P \\ \lg VPR \end{array} \end{array} ,$$

where I denotes a square identity submatrix. These running times are similar to those for the bit-reversal permutation. In particular, for high VPRs, the PE-based methods are much faster than the naive methods.

Theorem 2 explains why the naive algorithms do so poorly on these permutations. Both permutations have low rank in the lower right $\lg P \times \lg P$ submatrices of their characteristic matrices: $\lg P - \lg VPR$. For $P = 2048$ and $VPR = 1024$, this rank is only 1. Consequently, they may have congestion at the target PEs upon each row send. The PE-based and cluster methods are good for these situations, since they evenly distribute the target PEs by forming 1-permutable sets.

6 Conclusion

We have examined five methods for performing BMMC permutations on the MasPar MP-2. The Gray-code modifications to the naive and PE methods yield slight improvements. For random permutations, the Gray-code naive method is generally the fastest. For certain permutations, however, the Gray-code naive method is very slow and the Gray-code PE method is far better.



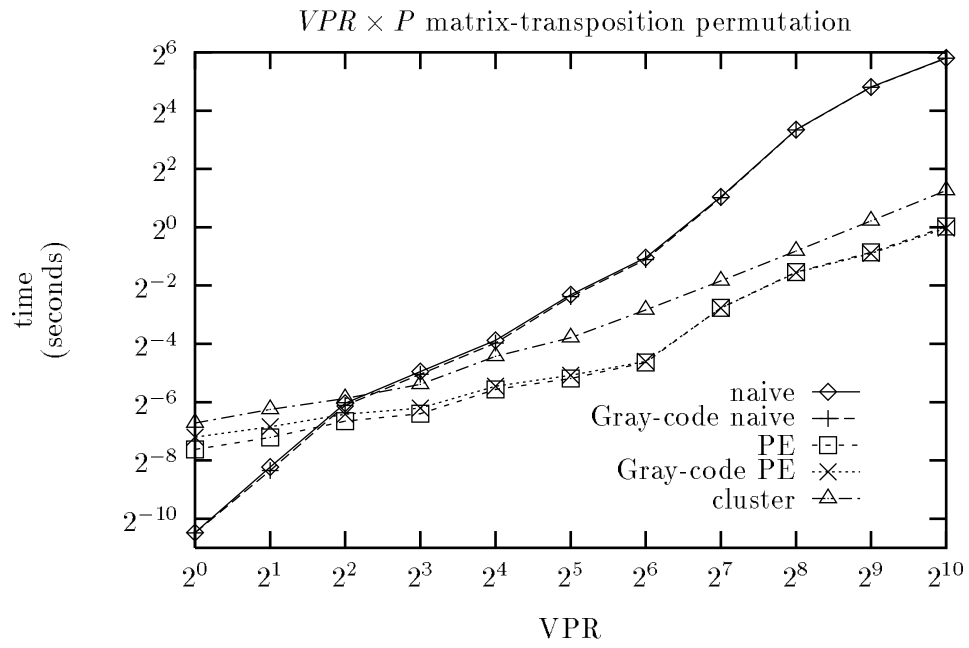


Figure 10: Total times to perform the bit-reversal permutation and the $VPR \times P$ matrix-transposition permutation on *cascade*. Plots are on a log-log scale.

The cluster method, which has the least congestion at router ports, may be clever, but it incurs too much overhead to be competitive.

Which method we recommend depends on the expected forms of characteristic matrices. For a truly random distribution, we recommend the Gray-code naive method. The Gray-code PE method is almost as good, especially at high VPRs. Some forms occur more often than others, however, and many of these forms elicit poor performance from the naive algorithms because their characteristic matrices have low rank in the lower right $\lg P \times \lg P$ submatrix. As we have seen, bit-reversal and certain matrix-transposition permutations have these forms. The PE-based and cluster methods are relatively insensitive to the form of the characteristic matrix.

The bottom line is this. When implementing at the level of MPL source code, being somewhat clever—using the PE method and using Gray-code orderings to compute source indices—yields faster routing than the naive method. But being *too* clever—using the cluster method—slows down the routing.

The running time of each of the five methods is dominated by the routing time (although for some, the setup time dominates at low VPRs). It is natural to ask whether the routing time can be improved. The authors know of no way to do so from the level of MPL code, but it may be possible by dropping down into lower-level code (perhaps at the level of the implementation of `pp_rsend()` function) or by using the X-Net instead of the global router. Scherson and Subramanian [16] have investigated how to perform general permutations quickly on the MasPar MP-2 global routing network, and the present authors, along with Len Wisniewski and Tom Sundquist at Dartmouth, have fast methods for performing BMMC permutations on the global router. Both these strategies apply only to the cluster method, however, since they assume only one element per network port. Because of the cluster method's high loop overhead, these strategies do not pay off. They may be worthwhile if they are implemented at the level of `pp_rsend()`. Munthe-Kaas [14] has investigated using the X-Net for routing BMMC permutations. This approach has the advantage of bypassing the cluster bottleneck, and so it may yet be possible to improve the routing time from the level of MPL.

This paper has not addressed the issue of determining whether a given permutation is BMMC. Given a vector of target indices on the MasPar MP-2, it is generally faster to just permute the data, routing row by row, than to determine whether the index vector describes a BMMC permutation and then use a special BMMC permuting algorithm. Consequently, the BMMC algorithms in this paper are best invoked either explicitly by the programmer or within library functions (e.g., power-of-2 matrix transpose) called by the programmer.

Finally, we are pleased to note that the PE method has been adapted to speed up a real-time magnetic resonance imaging (MRI) program. Chawla and Healy [4] perform a fixed BMMC permutation within their MRI code, and a simple variation of the PE method sped up their code.

Acknowledgments

The authors are indebted to Raghu Subramanian and Brian Alleyne for their help in understanding the MasPar MP-2 global router. Jacqueline LeMoigne and Lisa Hamet Bernard of the NASA Goddard Space Flight Center provided access to *nibbles*. Isaac Scherson, whose work is supported by NSF/MasPar Grant MIP-9205737, graciously allowed us to use a MasPar MP-2 with 4K PEs in an earlier version of this paper. Russ Tuck of MasPar answered our detailed questions about the MP-2. Steve Heller suggested the Gray-code technique for target-index calculation. Thanks also to Len Wisniewski and Tom Sundquist for many helpful discussions and to David Kotz for his help with the gnuplot software. Cindy Phillips, Catherine McGeoch, David Nicol, and anonymous referees provided valuable comments on earlier drafts of this paper. The purchase of *cascade* was made possible in part by National Science Foundation Grant CDA-9222806.

References

[1] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 204–216, Oct. 1987.

[2] R. 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, Apr. 1990.

[3] K. Bruhl. BMMC permutations on a DECmpp 12000/Sx 2000. Technical Report PCS-TR94-224, Dartmouth College Department of Computer Science, July 1994. Senior thesis.

[4] S. Chawla and D. Healy. The wavelet packet best basis algorithm: A fast parallel implementation on the MP-2 for real-time MRI. Technical Report PCS-TR96-271, Dartmouth College Department of Computer Science, Oct. 1996.

[5] T. 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.

[6] T. H. Cormen. Fast permuting in disk arrays. *Journal of Parallel and Distributed Computing*, 17(1–2):41–57, Jan. and Feb. 1993.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[8] T. H. Cormen, T. Sundquist, and L. 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.*

[9] Digital Equipment Corporation, Maynard, Massachusetts. *DECmpp Programming Language (ANSI) Reference Manual*, Dec. 1992.

[10] Digital Equipment Corporation, Maynard, Massachusetts. *DECmpp Programming Language (ANSI) User's Guide*, Dec. 1992.

[11] A. Edelman, S. Heller, and S. L. Johnsson. Index transformation algorithms in a linear algebra framework. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1302–1309, Dec. 1994.

[12] S. L. Johnsson and C.-T. Ho. Generalized shuffle permutations on boolean cubes. Technical Report TR-04-91, Harvard University Center for Research in Computing Technology, Feb. 1991.

[13] J. Keohane and R. 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, Oct. 1988.

[14] H. Munthe-Kaas. An algebraic theory for regular data mappings on parallel computers. Unpublished manuscript, Oct. 1994.

[15] D. Nassimi and S. Sahni. An optimal routing algorithm for mesh-connected parallel computers. *J. ACM*, 27(1):6–29, Jan. 1980.

[16] I. D. Scherson and R. Subramanian. Efficient off-line routing of permutations on restricted access expanded delta networks. In *Proceedings of the 7th International Parallel Processing Symposium*, pages 284–290, Apr. 1993.

[17] J. F. Sibeyn. *Algorithms for Routing on Meshes*. PhD thesis, Utrecht University, Dec. 1992.

[18] L. F. Wisniewski. *Efficient Design and Implementation of Permutation Algorithms on the Memory Hierarchy*. PhD thesis, Dartmouth College Department of Computer Science, Mar. 1996.

[19] L. F. Wisniewski, T. H. Cormen, and T. 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, Oct. 1996.