Dartmouth College
Dartmouth Digital Commons

Dartmouth College Undergraduate Theses | Theses and Dissertations

6-8-1994

BMMC Permutations on a DECmpp 12000/sx 2000

Kristin Bruhl
Dartmouth College

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

Part of the Computer Sciences Commons

Recommended Citation

Bruhl, Kristin, "BMMC Permutations on a DECmpp 12000/sx 2000" (1994). *Dartmouth College Undergraduate Theses*. 169.
https://digitalcommons.dartmouth.edu/senior_theses/169

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

BMMC Permutations on a DECmpp 12000/Sx 2000

Kristin Bruhl

June 8, 1994

Computer Science Honors Thesis

Contents

1 Introduction **1**

1.1 Contributions of this Thesis 3

1.2 Outline 3

2 Background **5**

2.1 Permutations 5

2.2 Model 6

2.3 Matrix Terminology 9

2.4 BMMC Permutations 11

2.5 BPC Permutations 13

3 The DECmpp Network **16**

3.1 The Machine 16

3.2 Network Communication 17

3.3 Expanded Delta Network Model 19

4 Approach **26**

4.1 Cormen's BMMC Permutation Algorithm 26

4.1.1 Finding a 1-permutable set 28

4.1.2 Creating a schedule, given a 1-permutable set 34

4.2 Using Cormen's Algorithm in the PE Array 35

4.3 Coding Issues 38

4.4 Forming a 1-permutable Set of Clusters 42

5 Evaluation of Results **46**

5.1 Congestion 47

5.2 Overhead 51

5.3 Timing 52

6 Conclusions **59**

Bibliography **62**

I would like to thank Isaac Scherson, Raghu Subramanian, and Brian Alleyne at UCI for letting us use their MasPar to test our algorithms, and for answering our questions about the EDN network model. Also, Len Wisniewski for spending hours working out various properties of BMMC permutations to help explain our results. Finally, I am espcially grateful to my advisor, Tom Cormen, for putting up with me for four terms, offering constant guidance, criticism, and patience, and not getting too mad when I didn't speak to him for three weeks.

Chapter 1

Introduction

Increasingly, modern computing problems, including many scientific and business applications, require huge amounts of data to be examined, modified, and stored. Parallel computers can be used to decrease the time needed to operate on such large data sets, by allowing computations to be performed on many pieces of data at once. For example, on the DECmpp machine used in our research, there are 2048 processors in the parallel processor array. The DECmpp can read data into each of these processors, perform a computation in parallel on all of it, and write the data out again, theoretically decreasing the execution time by a factor of 2048 over the time required by one of its processors.

Often, the computations that occur after the data is in the processors involve rearranging, or *permuting*, the data within the array of parallel processors. Information moves between processors by means of a network connecting them. Com-

munication through the network can be very expensive, especially if there are many *collisions*—simultaneous contentions for the same network resource—between items of data moving from one processor to another. When a program performs hundreds or even thousands of these permutations during its execution, a bottleneck can occur, impeding the overall performance of the program.

Effective algorithms that decrease the time required to permute the data within a parallel computer can yield a significant speed increase in running programs with large data sets. Cormen [Cor92, Cor93] has designed algorithms to improve performance when the data movement is defined by certain classes of permutations. This thesis will examine the performance of one of these classes, the bit-matrix-multiply/complement (BMMC) permutation, when implemented on the DECmpp. Although Cormen's algorithm was designed for parallel disk systems, this thesis adapts it to permutations of data residing in the memory of the parallel processors.

The DECmpp network follows the model of an Extended Delta Network (EDN). One characteristic of an EDN is that it has a set of input and output ports to the network, each of which can carry only one item of data at a time. If more than one item needs to travel over a given port, a collision occurs. The data must access the port serially, which slows down the entire operation. Cormen's algorithm reduces these collisions by computing a schedule for sending the data over the network.

For small data sets, it is not worthwhile to perform the extra operations to generate such a schedule, because the overhead associated with computing the schedule

outweighs the time gained by preventing collisions at the network ports. As the size of the data set increases, eliminating collisions becomes more and more valuable. On the DECmpp, when the data permutation involves more than 128 elements per processor, our algorithm beats the more naive and obvious method for permuting in the parallel processor array.

1.1 Contributions of this Thesis

This thesis provides

- a routing algorithm for BMMC permutations which decreases the number of collisions at the network ports, speeding up the routing of large data sets,
- an understanding of the issues involved in modifying and implementing Cormen's algorithm so that it works on data residing in the memory of the DECmpp parallel processors, and
- a notion of the relative speeds of built-in routing in the network in comparison to our higher level routing implementation.

1.2 Outline

The rest of this thesis is organized as follows. Chapter 2 defines and explains the BMMC class of permutations and gives terminology helpful in understanding the rest

of the thesis. Chapter 3 describes the DECmpp 12000/Sx 2000, including the parallel processor array and the serial processors we also use. Chapter 3 also discusses a model of the network given by Scherson and Subramanian [SS93] and their work in routing permutations using this network model. Chapter 4 presents Cormen's algorithm for routing BMMC permutations. It explains how we modified this algorithm to make it compatible with the DECmpp environment and to maximize the benefit we can derive from running it on a parallel system. Chapter 5 compares network collisions within the array of processors when Cormen's algorithm is used and when it is not, as well as timing results for the two situations. Finally, Chapter 6 contains some concluding remarks.

Chapter 2

Background

In this chapter, we describe our model of a parallel processor array and compare it to the parallel disk model for which Cormen designed his algorithm. We explain some basic matrix terminology, which aids in understanding the material in the following chapters. Finally, we define the class of BMMC permutations and the subclass of BPC permutations, which includes many commonly used permutations.

2.1 Permutations

A *permutation* is a perfect rearrangement of the elements of a set. In the context of an array of parallel processors, we use the term permutation to refer to an interprocessor communication in which each processor sends one piece of data and receives one piece of data [SS93].

Frequently, a program will need to permute a vector of data larger than the number of processors on the machine. In this case, we can imagine a machine with enough processors to successfully complete the permutation. We call this a system of *virtual processors*. These virtual processors can be simulated by storing the contents of several virtual processors on each physical processor, and running the actions of each virtual processor in serial. The ratio of the number of virtual processors to the number of physical processors is called the *virtual processor ratio* (VPR). In cases involving virtual processors, we are less interested in permutations than in *virtual permutations* (where each virtual processor sends and receives exactly one piece of data).

To specify a permutation of N records within an array of processors, we must give the *source address* of each record $x = (x_0, x_1, \ldots, x_{n-1})$, where $n = \lg N$, and its corresponding *target address* $y = (y_0, y_1, \ldots, y_{n-1})$, for each x and y in the range $0, 1, \ldots, N-1$. By definition, the set of target addresses must be a rearrangement of the elements of the set of source addresses.

2.2 Model

The model we use throughout this thesis is as follows. N *records* are stored on D *physical processor elements* (PEs) $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$. The VPR is N/D, equal to the number of records stored in the memory of each PE. Figure 2.1 shows this layout of

$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$	$\mathcal{D}_7$
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 2.1: The layout of $N = 32$ records in a parallel processor array with $D = 8$. The VPR is $(N/D) = 4$. Numbers indicate record indices.

records. Within our model, record indices will vary most rapidly within each PE, and less rapidly between PEs. (DECmpp terminology refers to this layout as "hierarchical virtualization.") Figure 2.2(a) demonstrates how to parse an address using our model. The following notations are used here and throughout this thesis:

$$\begin{aligned} d &= \lg D , \\ v &= \lg(N/D) , \\ n &= \lg N . \end{aligned}$$

The least significant v bits specify the offset of a record within its PE, and the most significant d bits specify on which PE the record resides.

This model is similar to the Vitter-Shriver scheme [VS90, NV91] for the layout of data on a parallel disk system, used by [Cor92, Cor93, CSW93]. In the Vitter-Shriver model, the independent processors are disks, rather than the PEs that we use. Accordingly, N records are stored on D disks.[1] The layout of records is shown

[1]The Vitter-Shriver model also allows for blocks of records on each disk. Reading and writing

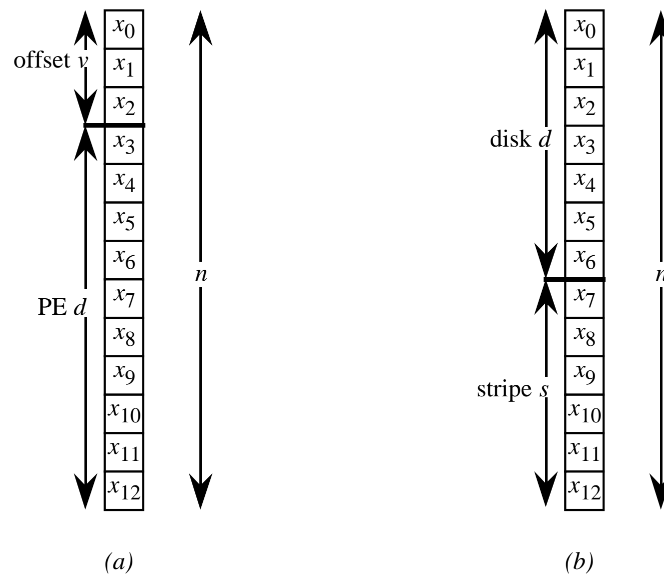

Figure 2.2: Parsing the address $x = (x_0, x_1, \ldots, x_{n-1})$ of a record. (a) On a system of parallel processor elements. Here, $n = 13$, $v = 3$, and $d = 10$. The least significant v bits contain the offset of a record within a PE (the virtual processor within the physical processor) and the most significant d bits contain the number of the PE. (b) On a parallel disk system, using the Vitter-Shriver model. Here, $n = 13$, $d = 7$, and $s = 6$. The least significant d bits contain the number of the disk and the most significant s bits contain the stripe number.

in Figure 2.2(b). The least significant d bits specify the number of a disk, and the most significant $s = n - d$ bits specify on which "stripe" the record resides. The most notable difference between the two models is that in our model the independent devices correspond to the most significant bits of the address, whereas in the Vitter-Shriver model, they correspond to the least significant bits of the address.

2.3 Matrix Terminology

In order to understand the discussion of the following permutation classes, some basic matrix terminology is necessary.[2] A *matrix* is a two dimensional array of numbers. For example,

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

is a 2×3 matrix $A = (a_{ij})$. The element of the matrix in row i and column j is denoted a_{ij}. Uppercase letters denote matrices, and the corresponding lowercase letters with subscripts denote elements of these matrices. We address matrix elements beginning with 0.

---

operations are performed on entire blocks. A block size of 1 allows manipulation of individual records.

[2]The definitions and examples used in this chapter are taken from [CLR90, Section 31.1].

A *vector* is a matrix with just one column. For example,

$$x = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix}$$

is a vector of length 3. We use lowercase letters to denote vectors. The ith element of an n-vector is denoted x_i. We sometimes view an $m \times n$ matrix A as a set of n column vectors $A_0, A_1, \ldots, A_{n-1}$, each of length m.

In the following chapters, we will often be concerned with whether or not a matrix can be inverted. The *inverse* of an $n \times n$ matrix A, written as A^{-1}, has the property that $AA^{-1} = I_n = A^{-1}A$, where I_n is an $n \times n$ matrix with 1s along the main diagonal and 0s in every other position:

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

For example,

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{-1} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}.$$

Not all nonzero matrices have inverses. If a matrix does not have an inverse, it is called *singular*, or *noninvertible*. A matrix which does have an inverse is called *nonsingular*, or *invertible*.

A set of vectors $\{x_1, x_2, \ldots, x_n\}$ is said to be *linearly dependent* if there exist coefficients $c_1, c_2, \ldots, c_n$, at least one of which is non-zero, such that $c_1x_1 + c_2x_2 + \cdots + c_nx_n = 0$. If vectors are not linearly dependent, they are *linearly independent.* The identity matrix is an example of a matrix with linearly independent columns and rows.

A fundamental property of matrices will be very useful in the following chapters:

> If all of the columns and rows of a square matrix are linearly independent, then that matrix is nonsingular.

We shall rely on this property extensively.

2.4 BMMC Permutations

The class of permutations with which we are concerned in this thesis is the class of *bit-matrix-multiply/complement*, or *BMMC* permutations. A BMMC permutation is defined by an $n \times n$ *characteristic matrix* $A = (a_{ij})$ whose entries are drawn from $\{0, 1\}$ and which is nonsingular over $GF(2)$,[3] and a *complement vector* $c = (c_0, c_1, \ldots, c_{n-1})$ of length n. Treating a source address of a record x as an n-bit vector, we perform matrix-vector multiplication over $GF(2)$ and then complement some subset of the resulting bits to form the corresponding target address y. In other words,

[3] 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.

$y = A\,x \oplus c$, or

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

This class includes the permutations for Gray code and inverse Gray code. If $y = \mathrm{Gray}(x)$, then

$$y_i = \begin{cases} x_i \oplus x_{i+1} & \text{if } 0 \le i < \lg N - 1 \ , \\ x_i & \text{if } i = n-1 \ . \end{cases}$$

For example, if $N = 2^6$, the corresponding BMMC permutatation is given by

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

If $y = \mathrm{Gray}^{-1}(x)$, then

$$y_i = \bigoplus_{i \le j \le n-1} x_j \ ,$$

and as a BMMC permutation for $N = 2^6$, we have

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

2.5 BPC Permutations

If we restrict the characteristic matrix of a BMMC permutation to be a permutation matrix—each row and each column contains exactly one 1—we obtain the subclass of *bit-permute/complement* (*BPC*) permutations. We can also think of BPC permutations as being defined by a permutation $\pi : \{0, 1, \ldots, n-1\} \xrightarrow{1-1} \{0, 1, \ldots, n-1\}$ on the address bits, and a complement vector c. The characteristic matrix $A = (a_{ij})$ and the address-bit permutation π have the relationship

$$a_{ij} = \begin{cases} 1 & \text{if } i = \pi(j) \ , \\ 0 & \text{otherwise} \end{cases}$$

for $i, j = 0, 1, \ldots, n-1$.

Many common permutations fall into the BPC class. For example, matrix transposition is a BPC permutation. Matrix transposition consists of mapping the (i, j) entry of an $R \times S$ matrix to the (j, i) position. The n-bit address of each entry is

made up of a $(\lg R)$-bit row number in the most significant bits followed by a $(\lg S)$-bit column number in the least significant bits. Thus, the transposition mapping swaps the upper $\lg R$ bits and the lower $\lg S$ bits of the source address to produce the target address. The permutation of bits, therefore, is a cyclic rotation by either $\lg R$ positions in one direction or $\lg S$ positions in the other (no bits are complemented). The equation for this permutation is

$$\pi(j) = (j + \lg R) \bmod n = (j - \lg S) \bmod n$$

and $c_j = 0$ for $j = 0, 1, \ldots, n-1$.

Bit-reversal permutations are also BPC permutations. A bit-reversal consists of reversing the bits of the source address to form the target address, If the source address of a record is $(x_0, x_1, \ldots, x_{n-1})$, its target address will be $(x_{n-1}, x_{n-2}, \ldots, x_0)$. The permutation equation is

$$\pi(j) = (n-1) - j$$

and $c_j = 0$ for $j = 0, 1, \ldots, n-1$.

A final example of a BPC permutation is matrix reblocking. In a matrix reblocking permutation, the bits of a source address are partitioned into four groups, and the middle two groups are swapped. If a source address is $(\alpha, \beta, \gamma, \delta)$, where α, β, γ and

δ each represent zero or more consecutive bits of the address, the target address will be $(\alpha, \gamma, \beta, \delta)$. The equation for the permutation is

$$\pi(j) = \begin{cases} j + |\gamma| & \text{if } j \in \beta \ , \\ j - |\beta| & \text{if } j \in \gamma \ , \\ j & \text{otherwise} \end{cases}$$

and $c_j = 0$ for $j = 0, 1, \ldots, n-1$.

We have explained the class of permutations on which this thesis focuses. We now proceed to examine the specific machine, the DECmpp 12000/Sx 2000, on which we perform the routing of these permutations.

Chapter 3

The DECmpp Network

In this chapter we describe the DECmpp 12000/Sx 2000, paying special attention to its array of parallel processors [Dig92b]. Then we will discuss a model for this network presented in [SS93], along with their work on routing permutations using this model.

3.1 The Machine

The DECmpp 12000/Sx 2000 is a massively parallel processing system, made up of a *console system* and a *data parallel unit* (DPU). The console is a processor providing standard I/O devices. In our case, the console system is a DECstation 5000/240.

The DPU is made up of an *array control unit* (ACU), an array of PEs (2048 in our case), and a PE communication system. All of the parallel processing takes place within the DPU. The ACU controls the PE array and performs any operations

on *singular* (i.e., non-parallel) data within modules of code written for the parallel processors. When the ACU encounters instructions dealing with parallel data, it sends data and instructions to each PE simultaneously.

Each PE within the PE array has its own processor and data memory. When the ACU sends an instruction to the PEs, each PE carries it out only on variables 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. We refer to any piece of data sent between PEs in this fashion as a *message*. It is the function of the *global message router* in the PE communication system to send these messages between PEs.

The PEs are arranged in a two-dimensional matrix, 32×64 in our case. Each nonoverlapping 4×4 square matrix within the PE array forms a *cluster*, so our machine has a total of 128 clusters, as shown in Figure 3.1.

3.2 Network Communication

The global message router can communicate with all PE clusters at the same time. It is limited, however, to communicating with one PE per cluster at a time, because each cluster has only one input port and one output port to the network. Every inter-PE communication must take place over this network. Even if one PE connects to another PE in the same cluster, the message must travel over the global router,

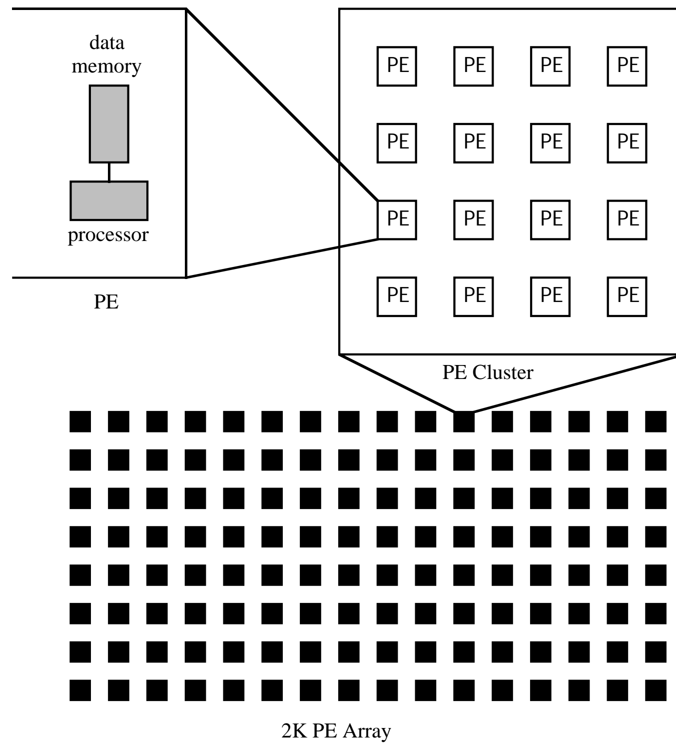

Figure 3.1: The PE Array of the DECmpp 12000/Sx 2000 [Dig92a, page 1-23].

and thus use the input and the output ports for that cluster. To communicate with more than one PE in a cluster, the router must make serial accesses to that cluster.

This network property is significant in our research, since to achieve the fastest results routing our data, we would like to minimize the number of serial accesses to a cluster. We are concerned, then, not only with sending one record to or from each PE at one time, but also with sending only one record from and to each PE cluster. We will discuss this issue in depth in Chapter 4.

3.3 Expanded Delta Network Model

Scherson and Subramanian [SS93] have examined the DECmpp global network[1] with respect to this restriction on accessing multiple PEs per cluster. They classify the network as an *expanded delta network* (EDN) and give an algorithm for routing permutations on the PEs through EDNs. The algorithm is guaranteed to completely route a permutation on the DECmpp in 32 iterations of the global message router. This algorithm is *off-line*, meaning that computing the routing pattern for a given permutation is not included in the running time of the algorithm.

Figure 3.2 shows an example EDN. The thin lines represent *thin wires*, which are capable of carrying only one message at a time. The thin wires on the far left of Figure 3.2 are the input ports to the network, and those to the right are the output

[1]This work was actually done on a MasPar MP-2 network, which is the same as the DECmpp network. For consistency, we will use the name DECmpp throughout this discussion.

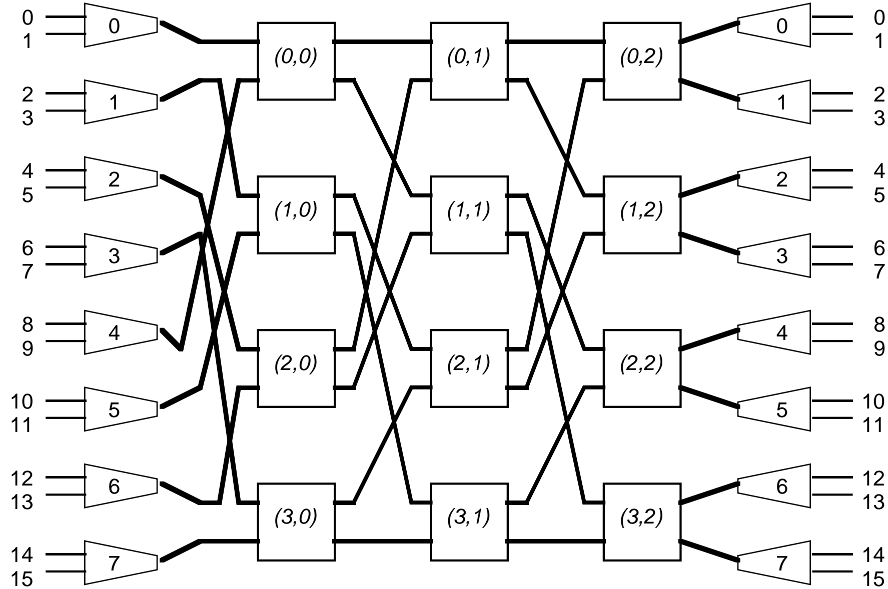

Figure 3.2: An example of an expanded delta network [SS93, page 285]. $T = 16$.

ports. The bold lines represent *thick wires*, and can each carry up to K messages at once. For the DECmpp, $K = 2$. There is a unique path through the network from each input wire to each output wire.

We define T to be the number of thin-wire ports on the network. Let $\mathcal{P} = \{P_0, P_1, \ldots, P_{T-1}\}$ denote a set of processors, each connected to a thin-wire port. A message is specified by an ordered pair of these processors—the first component is the source of the message, and the second component is the destination. A set of messages (a relation over P) is called a *traffic pattern*.

Scherson and Subramanian classify a traffic pattern Γ as *EDN passable* if

- for each thin wire w in the EDN, there is at most one message $\langle P_i, P_j \rangle \in \Gamma$ such that w lies on the unique path from input port i to output port j, and

- for each thick wire w in the EDN, there are at most K messages $\langle P_i, P_j \rangle \in \Gamma$ such that w lies on the unique path from input port i to output port j.

A permutation is a traffic pattern consisting of a set of messages for which each processor is the source of exactly one message and each processor is the destination of exactly one message. Scherson and Subramanian prove that every permuation can be expressed as the composition of three EDN passable permutations, each of which corresponds to one stage in the network. Thus, every permutation of T elements can be routed in exactly three complete passes over the data. First, we route the data through the first stage using the first permutation, which sends the messages to intermediate destination processors. Then, we use the second permutation to send each message from here to a second intermediate destination. Finally, to route the messages through the third network stage, we permute them according to the third permutation, and the messages reach their final destinations. Some EDNs, including the DECmpp, have only two stages. In these cases, every T-element permutation can be routed in only two passes.

When the number of elements N to be permuted is greater than T, we must deal with the issue of restricted access. More than one element must pass through each port to access the network. This idea of a virtual permutation was described in Section 2.1. Let $\mathcal{V} = \{V_0, V_1, \ldots, V_{N-1}\}$ represent a set of virtual processors. There are N/T virtual processors which must share each port to the EDN. A *virtual message* is an ordered pair of virtual processors, similar to the message described above. A

virtual permutation is a traffic pattern in which each virtual processor is the source of exactly one virtual message and the destination of exactly one virtual message. Since each port can only carry one message at a time, the virtual permutation must be sent in N/T serial accesses.

Scherson and Subramanian prove that every virtual permutation can be partitioned into N/T conflict free sets of messages by computing an edge coloring on a bipartite graph, an expensive operation which they perform off-line. Since every permutation of T elements can be routed on an EDN in three passes over the data, it follows that every virtual permutation of N elements can be routed on an EDN in $3N/T$ passes. If the EDN has only two stages, a virtual permutation can be routed in $2N/T$ passes.

As mentioned above, the EDN within our DECmpp has only one network input and output port per PE cluster, for a total of 128 ports. This configuration is analagous to Scherson and Subramanian's model with $T = 128$ processors (the clusters) and $N = 2048$ virtual processors (the PEs). These parameters give us $N/T = 2048/128 = 16$ sets of messages that must be routed serially through the two stage DECmpp network for every PE permutation. Since each set of messages requires 2 router iterations (1 for each pass over the data), all 16 sets require 32 iterations of the router.

In coding this routing algorithm, Subramanian encounters a problem which we also face in dealing with the ordering of PEs within clusters. The DECmpp Sx parallel

0	1	2	3	4	5	6	7	···	63
64	65	66	67	68	69	70	71	···	127
128	129	130	131	132	133	134	135	···	191
192	193	194	195	196	197	198	199	···	255
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮
1984	1985	1986	1987	1988	1989	1990	1991	···	2047

Figure 3.3: The matrix-oriented PE numbering given by the DECmpp programming environment. Bold lines represent cluster boundaries.

programming environment provides a PE numbering system to indicate each PE's relative position within the PE array. The PEs are numbered as one large 32×64 matrix, in the same fashion we described in Section 2.3. We call this numbering system a *matrix-oriented* addressing scheme. Figure 3.3 shows this addressing of the PEs.

Unfortunately, this PE numbering system is not easily compatible with the EDN model. The EDN model treats each cluster as a physical processor with a network port. The 16 PEs within that cluster are virtual processors. Ideally, the PEs within a cluster would be numbered sequentially. This way, one cluster would contain $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{15}$, the next cluster would contain $\mathcal{D}_{16}, \mathcal{D}_{17}, \ldots, \mathcal{D}_{31}$, and so on, neatly dividing the PEs into clusters corresponding to the network ports. We call this numbering system a *cluster-oriented* addressing scheme. Figure 3.4 gives a possible cluster-oriented addressing.

0	1	2	3	16	17	18	19	···	1363
4	5	6	7	20	21	22	23	···	1367
8	9	10	11	24	25	26	27	···	1371
12	13	14	15	28	29	30	31	···	1375
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮
684	685	686	687	700	701	702	703	···	2047

(a)

1	2	5	6	17	18	21	22	65	66	69	70	81	82	85	86
3	4	7	8	19	20	23	24	67	68	71	72	83	84	87	88
9	10	13	14	25	26	29	30	73	74	77	78	89	90	93	94
11	12	15	16	27	28	31	32	75	76	79	80	91	92	95	96
33	34	37	38	49	50	53	54	97	98	101	102	113	114	117	118
35	36	39	40	51	52	55	56	99	100	103	104	115	116	119	120
41	42	45	46	57	58	61	62	105	106	109	110	120	121	124	125
43	44	47	48	59	60	63	64	107	108	111	112	122	123	126	127

(b)

Figure 3.4: A cluster-oriented PE numbering, partitioning clusters into groups of sequential PEs. (a) The ordering of PEs within a cluster and (b) the ordering of all the clusters within the PE array.

Subramanian uses two mapping functions to switch back and forth between the two numbering systems in his code. (He refers to the cluster-oriented system as the "real" addressing and the matrix-oriented system as the "fake" addressing.) This mapping is actually a BPC permutation on bits of the PE addresses. For example, the permutation for the transformation from matrix-oriented addresses to cluster-oriented addresses on our $M = 2048$ network ($m = 11$ bits in the address of a PE) is

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

,

and the permutation for the reverse transformation, from cluster-oriented addresses to matrix-oriented addresses is

x	0	1	2	3	4	5	6	7	8	9	10
$\pi^{-1}(x)$	0	1	6	7	2	8	3	9	4	10	5

.

In the next chapter, we discuss the implementation of Cormen's algorithm for routing BMMC permutations on the DECmpp. We deal with several of the same issues in coding (i.e., matrix-oriented versus cluster-oriented PE addressing and contention for the network ports) discussed above. The algorithm differs from that of Scherson and Subramanian in that it is able to compute the routing schedule on-line for the class of BMMC permutations.

Chapter 4

Approach

Now that we have seen the organization of the DECmpp network, we turn to Cormen's algorithm for routing BMMC permutations, which can deal with the issue of restricted access to the network ports. We discuss the algorithm as it applies to Vitter-Shriver parallel disk systems and then present modifications which allow it to work on a parallel processor array. We also mention a few coding issues which arose during implementation of the algorithm.

4.1 Cormen's BMMC Permutation Algorithm

Cormen has developed an algorithm to perform any block BMMC permutation for which the block size is 1 on a Vitter-Shriver parallel disk system in only one pass over the data [Cor93]. To describe this algorithm, he relies on a technique of partitioning

the data into disjoint sets and permuting these sets sequentially. We describe this decomposition, and then show how Cormen's algorithm uses this method.

The method assumes that the records to be permuted reside in the *source portion* of the disks of the disk array, and that each disk has sufficient available room to hold another set of records the same size. Each disk originally holds N/D records, and there must be room for another set of N/D records. This storage space is called the *target portion* of the disks.

The algorithm has two components, which partition the data records into sets of data which we can route serially. First, we find a 1-permutable set of the records. Then, we devise a schedule for the entire permutation, given the first 1-permutable set.

The 1-permutable set is a special case of a k-permutable set, which Cormen defines as follows. For a permutation of source addresses to target addresses and a positive integer k, we define a *k-permutable* set of records as a set of kD source records such that

1. each disk contains exactly k of these source records, and

2. each disk has exactly k target records mapped to it [Cor93].

We sequentially permute sets of kD records at a time until the entire set of source records has been permuted. We call this sequence of sets a *schedule*. As long as we choose a k small enough that the entire set of kD records will fit into memory, we

can perform the permutation looking at each data record only once.

4.1.1 Finding a 1-permutable set

Cormen notes that in finding a 1-permutable set of records, we need only look at the first d rows of the characteristic matrix A. The other $n-d$ rows of A determine the stripe numbers of the target addresses, which do not affect 1-permutability. Likewise, only the first d positions of the complement vector c must be taken into consideration. Using this observation, the algorithm finds a 1-permutable set of blocks in the following four steps:

1. Find a set S of d "basis" columns for the first d rows of A.
2. Given S, define three sets of columns, T, U, and V.
3. Based on T and U, define a permutation R on the set of disks $\{0, 1, \ldots, D-1\}$.
4. From all of the above, define a set of source addresses $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$, which constitutes a 1-permutable set of blocks [Cor93].

We describe each step of the process, using a running example (from [Cor92]) to help in understanding.

Finding a set of basis columns

We first find a set S of d columns such that the submatrix $A_{0..d-1,S}$, is nonsingular. As we stated in Section 2.3, since A is nonsingular, all of its rows are linearly independent.

In particular, the first d rows are linearly independent. Thus, there must exist a subset of column indices $S \subseteq \{0, 1, \ldots, n-1\}$ such that $|S| = d$ and the $d \times d$ submatrix $A_{0..d-1,S}$ is nonsingular. We define

$$Q = A_{0..d-1,S} \ ,$$

so that Q^{-1} exists.

Example: Let $n = 5$ and $d = 3$, and use the characteristic matrix

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

We choose the basis $S = \{1, 3, 4\}$, giving

$$Q = A_{0..d-1,S} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \text{ and } Q^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} .$$

Defining T, U, and V

Given S, we define

$$T \ = \ \{0, 1, \ldots, d-1\} - S \ ,$$

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

The sets S and T form a partition of $\{0, 1, \ldots, d-1\}$ and the sets S, T and V form a partition of $\{0, 1, \ldots, n-1\}$. We can think of T as the subset of the first d columns that are not in the basis S and of U as the set of basis columns that are also in the first d columns. Cormen orders T and U into increasing order so that

$$T = \{t_0, t_1, \ldots, t_{|T|-1}\}, t_j > t_{j-1} \quad \text{for } j = 1, 2, \ldots, |T|-1,$$
$$U = \{u_0, u_1, \ldots, u_{|U|-1}\}, u_j > u_{j-1} \quad \text{for } j = 1, 2, \ldots, |U|-1.$$

Example: With $S = \{1, 3, 4\}$, we have $T = \{0, 2\}$, $U = \{1\}$, and $V = \emptyset$.

Defining the permutation R

In his discussion of this step, Cormen introduces a new notation to allow more explicit differentiations between integers and their binary representations. He denotes the d-bit representation of an integer $i \in \{0, 1, \ldots, D-1\}$ by $\text{bin}(i)$, which is also a vector of length d. The jth bit of this representation, for $j = 0, 1, \ldots, d-1$, is $\text{bin}_j(i)$.

We define the permutation R on $\{0, 1, \ldots, D-1\}$ in the following way. Let $i \in \{0, 1, \ldots, D-1\}$, and let the binary representation of i be $\text{bin}(i) = (i_0, i_1, \ldots, i_{d-1})$. Then

$R(i) = k$, where the binary representation of k is $\text{bin}(k) = (k_0, k_1, \ldots, k_{d-1})$ and

$$\begin{aligned} k_j &= i_{u_j} \quad \text{for } j = 0, 1, \ldots, |U| - 1 \,, \\ k_{|U|+j} &= i_{t_j} \quad \text{for } j = 0, 1, \ldots, |T| - 1 \,. \end{aligned}$$

The permutation R is a BPC permutation on $\text{bin}(i)$, and so it defines a permutation on $\{0, 1, \ldots, D-1\}$.

Example: Since $d = 3$, we have $D = 8$. We can see how to form the permutation R by writing out 0 through $D-1$ in binary and labeling each of the d columns of bits as to whether it belongs in T or U. Then we can rearrange the columns so that all the columns of U precede the columns of T:

	T	U	T		U	T	T	
i	0	1	2		1	0	2	$R(i)$
0	0	0	0		0	0	0	0
1	1	0	0		0	1	0	2
2	0	1	0		1	0	0	1
3	1	1	0	$\Longrightarrow$	1	1	0	3
4	0	0	1		0	0	1	4
5	1	0	1		0	1	1	6
6	0	1	1		1	0	1	5
7	1	1	1		1	1	1	7

Defining the set of source addresses

Finally, we define the set $(x^{(0)}, x^{(1)}, \ldots, x^{(D-1)})$ the following way:

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

for $i = 0, 1, \ldots, D-1$.

The source addresses specified by the above method yield a 1-permutable set, as Cormen proves [Cor93].

Example: We let the complement vector c be all 0s to keep the example simple. Here we compute $x^{(6)}$. We start by computing the bits in positions 1, 3, and 4, since $S = \{1, 3, 4\}$. Writing lower-order bits on the left, and indexing from zero, we have that $\mathrm{bin}(6) = 011$ and, taking bits 0 and 2 because $T = \{0, 2\}$, we have $\mathrm{bin}_T(6) = 01$. We know $\mathrm{bin}(R(6)) = 101$ (from above), so

$$\begin{aligned} x_{\{1,3,4\}}^{(6)} &= Q^{-1} A_{0..2,T} \ \mathrm{bin}_T(6) \oplus \mathrm{bin}(R(6)) \\ &= \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \end{aligned}$$

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

$$= \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

Filling these values into the positions indexed by set S in the source address, we get $x^{(6)}$ =?0?01, where the positions marked by question marks correspond to set T. For these remaining positions, we use the bits of $\mathrm{bin}_T(6) = 01$, and so $x^{(6)} = 00101$. We also have that $y^{(6)} = Ax^{(6)} = 11110$.

If we do this calculation for $i = 0, 1, \ldots, 7$, we compute the following source and target addresses:

	0	1	2	3	4
$x^{(0)}$	0	0	0	0	0
$x^{(1)}$	1	0	0	1	0
$x^{(2)}$	0	1	0	0	0
$x^{(3)}$	1	1	0	1	0
$x^{(4)}$	0	1	1	0	1
$x^{(5)}$	1	1	1	1	1
$x^{(6)}$	0	0	1	0	1
$x^{(7)}$	1	0	1	1	1

	0	1	2	3	4
$y^{(0)}$	0	0	0	0	0
$y^{(1)}$	1	1	0	1	0
$y^{(2)}$	1	0	1	1	0
$y^{(3)}$	0	1	1	0	0
$y^{(4)}$	0	1	0	0	0
$y^{(5)}$	1	0	0	1	0
$y^{(6)}$	1	1	1	1	0
$y^{(7)}$	0	0	1	0	0

Note that for both the set of source addresses and the set of target addresses, the bits that represent the disk numbers (in the three leftmost columns), are permutations of $\{0, 1, \ldots, 7\}$.

4.1.2 Creating a schedule, given a 1-permutable set

Once we have a 1-permutable set, we can partition the BMMC permutation into 1-permutable sets of records. This sequence of sets forms a schedule.

To create the schedule, Cormen defines a set of $N/D - 1$ n-vectors $\{p^{(0)}, p^{(1)}, \ldots, p^{(N/D-1)}\}$ such that $p^{(j)}_{0..d-1} = 0$ and $p^{(j)}_{d..n-1}$ is the binary representation of j for $j = 0, 1, \ldots, N/D - 1$. Given a 1-permutable set of blocks $\{x^{(0)}, x^{(1)}, \ldots, x_{(D-1)}\}$, we

form the following N/D sets:

$$\{x^{(0)} \oplus p^{(0)}, x^{(1)} \oplus p^{(0)}, \ldots, x^{(D-1)} \oplus p^{(0)}\} \ ,$$

$$\{x^{(0)} \oplus p^{(1)}, x^{(1)} \oplus p^{(1)}, \ldots, x^{(D-1)} \oplus p^{(1)}\} \ ,$$

$$\{x^{(0)} \oplus p^{(2)}, x^{(1)} \oplus p^{(2)}, \ldots, x^{(D-1)} \oplus p^{(2)}\} \ ,$$

$$\vdots$$

$$\{x^{(0)} \oplus p^{(N/D-1)}, x^{(1)} \oplus p^{(N/D-1)}, \ldots, x^{(D-1)} \oplus p^{(N/D-1)}\} \ .$$

Through this procedure, we XOR the most significant s bits by $0, 1, \ldots, N/D-1$ in turn, to form the entire schedule. Cormen proves that this method does in fact generate a schedule given any 1-permutable set [Cor93].

4.2 Using Cormen's Algorithm in the PE Array

Cormen's algorithm for BMMC permutations was designed specifically for permuting data residing on a parallel disk array. However, we can use the same algorithm, with very little modification, to perform BMMC permutations on data residing on PEs within a PE array. We saw in Section 2.2 the differences in parsing an address from the Vitter-Shriver parallel disk model and our PE array. In the Vitter-Shriver model, the least significant d bits of an address specify the disk number, whereas in our case, it is the most significant d bits which define the PE number. Thus, we must form a 1-permutable set using the upper d bits, rather than the lower d bits that Cormen

uses. Here, we briefly summarize the changes made to the algorithm for this reason.

- In computing the set of basis columns, we define

$$Q = A_{n-d..n-1,S} \ .$$

- In defining T, U and V, we use

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

- In defining the source addresses, we use

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

- In creating the schedule of 1-permutable sets, we define the set of $N/D-1$ n-vectors $\{p^{(0)}, p^{(1)}, \ldots, p^{(N/D-1)}\}$ such that $p^{(j)}_{n-d..n-1} = 0$ and $p^{(j)}_{0..n-d-1}$ is the binary representation of j for $j = 0, 1, \ldots, N/D-1$.

In this way, we can create a schedule of 1-permutable sets in the PE array.

Running this modified version of Cormen's algorithm on a SIMD machine allows us improve the performance of the algorithm by parallelizing some of its steps. When we perform a computation for each element in a large set, we benefit from doing that computation in parallel within the PE array.

The first two steps in creating a 1-permutable set—finding the set of basis columns S and, from that, creating the sets T, U and V—do not require multiple calculations which depend on the values of the PE numbers. Hence, these steps will not take advantage of the capabilities of the DPU and are better run on the console, a much faster scalar processor. However, the third step—defining a permutation R on the set of PEs—does need to perform operations with different data for each PE. Specifically, this step must rearrange the bits of each PE number. If we had to perform this calculation serially for 2048 processors, we would spend a significant amount of time on this one operation. By running this step in the PEs, we perform the entire computation in parallel; each PE refers to its own number and permutes its own address. Similarly, the fourth step in creating the 1-permutable set—defining the source addresses—lends itself easily to running in parallel. The PEs can refer to the relevant bits of their own addresses, and the source addresses are all computed in parallel.

Once we have computed the 1-permutable set, we can also create the schedule in parallel. We now have the advantage that every PE already contains a element of the 1-permutable set. All we must do is XOR the least significant v bits of that element by $0, 1, \ldots, N/D - 1$ in turn to determine which record within each PE is part of

the current 1-permutable set. (Remember from Chapter 2 that v represents the bits giving the offset of a record within a PE.)

We see more thoroughly in the next section how Cormen's algorithm[1] takes advantage of properties of the parallel network.

4.3 Coding Issues

To test the effectiveness of Cormen's algorithm at speeding up BMMC permutations in the PE array, we use two methods to route the permutations through the network: a straightforward "naive" algorithm and Cormen's algorithm.

Implementing the naive algorithm

The simplest way to route a BMMC permutation in the PE array is to cycle through the records in each PE, sending them in order regardless of their target addresses. Each PE contains N/D records. Thus, we can keep a counter i, which increments from 0 to $v-1$, and with each iteration send the ith record in each PE to its target address storage location. This can be thought of as a row-by-row routing. Referring back to Figure 2.1, we see that sending the first element from each processor is comparable to sending the first "row" of data in the PE array.

[1] Throughout the rest of this thesis, we will use the version of Cormen's algorithm modified to run on the PE array rather than a parallel disk system. The changes we had to make to the original algorithm are not significant enough to warrant continued differentiation between the two versions; hence we will refer to the modified version simply as Cormen's algorithm.

This naive method computes a schedule of N/D sets of records to route through the network. This is the same number of sets that Cormen's algorithm generates. With this row-by-row method, however, we have no guarantees about the permutability of each set of records we send. We know that each PE contains exactly one source record per set, but we know nothing about how many target records are mapped to each PE. This mapping may be significant when we examine this routing in light of the expanded delta network model we presented in Section 3.3.

Let us assume for the moment that we are not dealing with a restricted access EDN—that is, we have one input and one output port to the network for each PE. When we send one source record from each PE, then, we should have no problem getting the data out into the network. It is likely, however, that the target addresses for two or more of these records are contained on the same PE.[2] Those records need to share the output port from the network to that destination PE. Since only one record can use the port at once, we have contention for the network ports, causing these records to be routed serially to their destinations. This is the condition we are able to avoid by using Cormen's schedule of 1-permutable sets.

[2]In fact, one can use a standard high-probability argument for random permutations to show that with probability $1-\Theta(1/D)$, there exists a PE that is the target of $\Theta(\lg D/\lg\lg D)$ source PEs. For BMMC permutations, the number of source PEs that route to one target PE is an increasing function of the rank of a particular submatrix of the characteristic matrix.

Implementing Cormen's algorithm

Using Cormen's algorithm, we guarantee that each PE will only send and receive one record for each set of records in the schedule. Therefore, we can avoid the port contention that our control algorithm suffers within the non-restricted EDN. However, Cormen's algorithm involves several computations to create the 1-permutable set before we can send any data. To maximize our benefit from the congestion-free routing, then, we must minimize the amount of time needed to compute the first 1-permutable set. We used several optimizations in coding both Cormen's algorithm and the naive algorithm, in an attempt to speed up the first set generation and also to ensure that we were not using poor methods for either algorithm. Using the same optimizations in both implementations makes our comparison of the algorithms more fair. We list some of these "coding tricks" here.

- We stored our matrices in column-major order, with each column packed into a word.[3] This organization simplifies matrix-vector multiplication (such as we perform every time we compute a record's target address from its source address) by allowing columns of the matrix to be XORed in one step.
- We stored our sets S, T, and U packed into single words as well. This approach decreases the amount of storage space required. It also allows us to determine easily what elements are in a given set by shifting the bits of the word—much

[3] We assumed that all addresses in our system fit in one word.

faster than accessing elements of an array as we would have done had we not packed the sets.

- We used a method developed by Len Wisniewski [Wis94] for computing the set of basis columns, S. This method works as follows:

  1. Create a submatrix A_{bottom} of A, containing the lower d rows of A.
  2. Find the leftmost column, j, of A_{bottom} that contains a 1 and is not already in S.
  3. Choose a row i containing a 1 in column j.
  4. Add column j into every column of A_{bottom} to the right which has a 1 in row i.
  5. Add column j to S.
  6. Repeat steps 2 through 5 until the basis contains d columns.

- We ran as many routines as possible on the console. A large part of the generataion of the first 1-permutable set—finding the basis colummS S and defining the sets T, U, and V— do not require parallel computations, and so we copy these variables out to the console, compute their values there, and then copy them back to the DPU.

Until now, we have been assuming that the EDN within the PE array is non-restricted. We saw in Section 3.3 that this is not really the case. The router network

within the DECmpp has only one network port per cluster rather than one per PE. Because Cormen's algorithm guarantees that each PE will only send and receive one record at a time, each cluster receives 16 records from each set of the schedule. Those 16 records have to be routed serially to their destination PEs. This method produces less contention than we expect with our naive method since, as we have seen, a row-by-row send may route more than 16 records to one cluster. We would like a way, however, to eliminate the port contention inherent in in the 1-permutable set of PEs.

4.4 Forming a 1-permutable Set of Clusters

To alleviate the problem of contention for network ports, we devise a method to route a 1-permutable set containing only one record per network port at a time. We can use Cormen's algorithm to do so, by thinking of the clusters rather than the PEs as the independent devices involved. We compute a 1-permutable set of the clusters and then create a schedule to send all of the records within each cluster in sequential sets. Figure 4.1 shows how we parse a cluster-oriented address using Cormen's algorithm on clusters.

When we try to access records within a cluster, we encounter the same problem described in Section 3.3. We would like a cluster-oriented number scheme for addressing the PEs, rather than the matrix-oriented scheme inherent in the DECmpp's native language, MPL. We use the BPC transformation between the two addressing systems

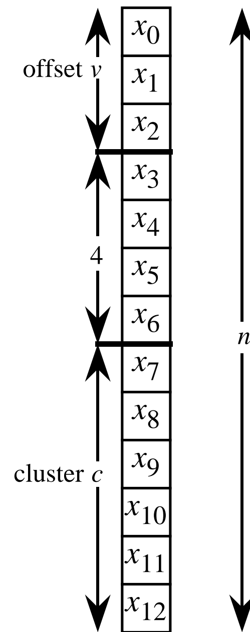

Figure 4.1: Parsing the address $x = (x_0, x_1, \ldots, x_{n-1})$ of a record using the cluster rather than the PE as the independent device. This address layout uses cluster-oriented addressing. Here we have $n = 13$, $v = 2$ and $c = 7$. The cluster size is 16, giving 4 bits for the offset of a PE within the cluster.

discussed in that section to permute the address bits of the PEs, thus renumbering them according to the addressing scheme we want.

Since our characteristic matrix A and complement vector c give us a BMMC permutation on the actual, matrix-oriented layout of PEs, we must adjust this permutation to yield the same results on our new layout. To do so, we create two characteristic matrices—one for the transformation from matrix-oriented addressing to cluster-oriented addressing and one for the inverse transformation, cluster-oriented to matrix-oriented.

From the original BMMC permutation, which A and c give us, we create a new characteristic matrix $A_{cluster}$ and complement vector $c_{cluster}$, which simulate the same BMMC permutation on a cluster-oriented layout of PEs. Let us call the characteristic matrix of the permutation of the PE addresses from matrix-oriented to cluster-oriented G, and the inverse permutation from cluster-oriented to matrix-oriented G^{-1}. Through multiplication of A, c, G, and G^{-1}, we create $A_{cluster}$ and $c_{cluster}$ using the equations

$$\begin{aligned} A_{cluster} &= GAG^{-1} \ , \\ c_{cluster} &= Gc \ . \end{aligned}$$

This is the BMMC permutation we perform on the cluster-oriented PEs.

By using a 1-permutable set of clusters, we can remove the port contention in

routing the permutation. We are, however, introducing more overhead into the computations. We must calculate the transformations back and forth between matrix-oriented addressing and cluster-oriented addressing. Also, we have reduced the number of records in each 1-permutable set from one per PE, or D records, to one per cluster, or C records, where $C = D/16$. Using D records per set, we can route all N records in N/D sets. Using C records per set, we need $16N/D$ sets to route all N records. We have increased the number of sets in our schedule by a factor of 16. Thus, the loop overhead associated with sending each set also increases.

In the next chapter, we examine the results we obtained from all three methods for routing BMMC permutations: the 1-permutable set of PEs, the 1-permutable set of clusters and our naive row-by-row scheduling. We shall look at both the intra-network congestion each of the methods produced and the actual time in which each method could perform the entire permutations.

Chapter 5

Evaluation of Results

In Chapter 4, we described our algorithm for routing BMMC permutations. We would like to perform these permutations more quickly by using this algorithm than we can by using the naive row-by-row routing of the records. We hope that by reducing the number of collisions that occur at the network ports through the calculation of a schedule of 1-permutable sets, we can lower the total running time (including the schedule calculation) to route the permutation. In this chapter, we examine the empirical data gathered for routing BMMC permutations using 1-permutable sets of PEs, 1-permutable sets of clusters, and the naive algorithm.

There are two main factors that figure into the total time each algorithm requires: congestion at the network ports, and the overhead time to set up the routing schedule before we can send any records. The 1-permutable-set algorithm reduces the number of intra-network collisions from the number we observe under the naive method. There

is, however, virtually no setup time associated with the naive method, whereas the 1-permutable set algorithm must perform several computations to create a schedule. We analyze each of these factors separately and then report overall running times for each of the three methods.

5.1 Congestion

We can measure the amount of contention at the network ports by counting the number of iterations that the global message router uses in sending a given set of records from their source locations to their target locations. Since it can only send one message through a network port at a time, the number of iterations should equal the number of messages trying to enter each cluster. We call this number of iterations the *port serialization.* For example, if we send a record from one PE within each cluster, and each record's target address guarantees that only one PE per cluster will receive a record, the port serialization is 1. No serial accesses of network ports need to be made. If, on the other hand, we send one record from a particular PE to every PE in the PE array, the port serialization is D, because all D messages must queue up at the sending PE's input port to the network.

In this section, we predict the port serialization we expect to see for our method using PEs as the independent devices, our method using clusters as the independent devices, and the naive method based on our discussion of the algorithms in Chap-

ter 4. Then, we compare these predictions to empirical data observed with the three methods.

If we use PEs as the independent devices, our algorithm generates a schedule of 1-permutable sets of PEs. By definition, then, when we route each set, each PE sends and receives exactly one record. Each cluster (and each network port) must send and receive exactly 16 records. These 16 records are accessed serially at the network ports, and so we expect the port serialization for this method to be 16.

Running our algorithm with clusters as the independent devices yields a schedule of 1-permutable sets of the clusters. For each set routed, each cluster sends and receives only one record. This situation is analagous to our best case example described above, and thus the port serialization should be 1.

We cannot predict an exact number of collisions that we expect when using the naive row-by-row method. The number of records sent to each cluster during a given routing step varies depending on the permutation we perform. If, for a given row, the target addresses of the records in that row all lie in different PEs, the port serialization will be 16, just as in the PE permutable set method. The identity permutation, for example, where the source address and target address for each record are the same, yields a port serialization of 16. If all the target addresses of the records in a row lie within the same cluster, we should observe a port serialization of D, since all the records must serially enter that cluster's network port. An example of this case is a matrix transpose permutation.

We gathered data on the port serialization using the system defined `__routerCount` variable in MPL, which counts the number of iterations of the global message router. These iterations are caused by two types of occurences in the network. One is contention for network ports. The other is congestion in the switches of the network, shown in Figure 3.2 as the lines between the three interior stages of the EDN [Sub93]. When we gathered data on port serialization under the three algorithms, we noted that in a small percentage of the runs, `__routerCount` exceeds what we expect the port serialization to be under the PE-permutable set method and the cluster-permutable-set methods. The values that we observed, both on our 2048 PE machine and a 4096 PE MasPar at University of California Irvine,[1] are given in Figure 5.1. This discrepancy arises because not all BMMC permutations are EDN-passable. When we route a permutation that is not EDN-passable, `__routerCount` measures congestion within the switches as well as port serialization. The port serialization we observed under the naive method is, as expected, significantly higher than under the permutable set methods. Figure 5.2 shows this data.

The method of Subramanian and Scherson (given in [SS93]) which we discussed in Section 3.3 guarantees that every permutation can be routed in two passes through the DECmpp network. We implemented a specialized routing algorithm for BMMC permutations using their EDN model of the network. The algorithm computed a

[1]We thank Isaac Scherson, whose work is supported by NSF/MASPAR grant number MIP-9205737, for access to the MasPar at UCI.

__routerCount	2048 PE machine	4096 PE machine
1	88.1%	82.7%
2	11.9%	17.0%
4	0.0%	0.3%

Figure 5.1: Frequencies of router counts observed under the cluster-permutable-set method. The 4096 PE machine is `lilliput.ics.uci.edu`

VPR	Min. Port Serialization	Max. Port Serialization	Avg. Port Serialization
1	16	16	16.0
2	16	32	18.0
4	16	64	19.3
8	16	128	19.7
16	16	256	20.1
32	16	512	20.1
64	16	1024	21.6
128	16	2048	22.4
256	16	4096	24.6
512	16	8192	29.5
1024	16	16384	34.1

Figure 5.2: Port serialization under the naive routing algorithm for various sizes of source vectors.

schedule of sets using Cormen's method, and routed each element of the set to an intermediate destination and then its final destination, according to the two stages of the EDN. We found that, as Subramanian and Scherson state, we could route all the permutations through the network in two passes with this algorithm. The setup time was significant, however, especially when we found that when we omitted routing to the intermediate destinations, instead routing directly to the final destinations, we observed the results in Figure 5.1.

5.2 Overhead

We have seen that using either the 1-permutable-sets-of-clusters or the 1-permutable-sets-of-PEs methods results in significantly fewer collisions at the network ports. This property is clearly a benefit to routing BMMC permutations using one of these algorithms rather than sending row-by-row. Both these algorithms, however, require us to calculate the first 1-permutable set. Even when we run the scalar portions of this code on the console, as we discussed in Section 4.3, the time taken to execute this portion of the algorithm is significant. Certainly, for very small VPRs, the overhead associated with the calculation of the first 1-permutable set will be higher than the time to resolve the intra-network collisions of the naive method. We hope that, as the VPR increases (and the number of sends needed to route all of the data also increases), the one-time cost of generating the first 1-permutable set will be compensated by the

time saved from low port serialization.

There is also a measurable overhead inherent in creating each 1-permutable set in the schedule, even once we have calculated the first set. We must XOR each source address with the current value of the mask to determine whether a given record is part of the current set. Even the simple loop statement we use to increment the set number within the schedule eats up a little bit of time. When we send only one record per cluster rather than one record per PE in each set, we must create 16 times more sets in the schedule to route every record. By doing so, we assure that the port serialization will usually be 1 rather than the 16 we obtain by sending one record from each PE. Is the overhead inherent in running 16 times as many iterations of our main loop offset by the time needed to perform 16 times as many serial accesses within the network?

In the next section, we report the actual timing figures obtained by routing BMMC permutations using each of the three algorithms. We shall analyze these figures in light of the tradeoffs discussed above to determine which of the methods yields the most favorable results.

5.3 Timing

Two essential measures in determining the value of an algorithm are the overall time and space it requires. We know that running either of the permutable-set algorithms

requires extra space to hold infomation for creating the schedule of 1-permutable sets. The naive method does not require this extra space. A partially redeeming factor of the permutable-set algorithms is that they require only $\Theta(D)$ space for computation, not $\Theta(N)$. Thus, as the VPR increases, for a given size PE array the space needed remains constant.

We now examine the time taken to actually run BMMC permutations on the DECmpp using each method and show that the permutable-set algorithms require less time than the naive method for sufficiently large VPRs. We tested each algorithm on characteristic matrices which produce various amounts of congestion within the network, to determine the threshold VPR for which the cost of computing the first 1-permutable set outweighs the cost of resolving intra-network collisions.

To test the case in which there is as little network congestion as possible, we ran the three algorithms on the identity matrix. As expected, our naive algorithm performed much better than either the PE-permutable-set or the cluster-permutable-set methods, as shown in Figure 5.3. Since there is no contention in the network to start with, computing contention-free sets will not help routing time and will increase overhead computation time.

At the other end of the spectrum, we ran the three algorithms on a transpose matrix to maximize the amount of congestion in the network. Referring to Figure 2.1 and viewing the PE array as a matrix, we can intuitively understand how this permutation creates maximum congestion. A matrix transpose swaps the rows and the

VPR	Cluster	PE	Naive
1	0.011	0.007	0.001
2	0.016	0.038	0.022
4	0.025	0.069	0.063
8	0.043	0.012	0.005
16	0.08	0.017	0.011
32	0.155	0.028	0.021
64	0.308	0.049	0.043
128	0.619	0.092	0.065
256	1.251	0.178	0.121
512	2.53	0.352	0.295
1024	5.128	0.702	0.654

Figure 5.3: Routing times (in seconds) for the identity permutation, using each of three algorithms. We give times for permuting data sets with VPRs of 1 to 1024.

columns of a matrix. In the PE array, this pattern means sending the first record in each processor to the first PE. The 1-permutable-set algorithms will compute a schedule to guarantee that only one record is sent to each PE in each pass, but the naive algorithm will route the entire first row at once. Since all the first row records map to the first processor, they must all be routed serially. This situation is the worst case for the naive method or equivalently, the case in which our 1-permutable set algorithm can beat the naive method by the greatest margin. The running times of the three algorithms on a matrix transpose permutation confirm this intuition, as shown in Figure 5.4.

Perhaps more interesting than either of these extreme situations is timing the "average case" matrix with each of the algorithms. Any nonsingular matrix over $GF(2)$ characterizes a BMMC permutation and can be routed using these algorithms.

VPR	Cluster	PE	Naive
1	0.011	0.008	0.001
2	0.017	0.011	0.005
4	0.027	0.013	0.010
8	0.043	0.013	0.029
16	0.080	0.023	0.075
32	0.169	0.044	0.125
64	0.336	0.108	0.457
128	0.620	0.194	1.779
256	1.251	0.301	7.022
512	2.532	0.680	27.936
1024	5.130	1.136	55.876

Figure 5.4: Routing times (in seconds) for the matrix transpose permutation, using each of the three algorithms.

Almost every nonsingular matrix falls somewhere between the two cases described above with respect to port contention in the network and consequently the overall running time. To test the average-case performance of the algorithms, we ran them on randomly generated nonsingular matrices.

It is much less obvious which algorithm will win out on these matrices and, as we see in Figures 5.5 and 5.6, the size of the source vector determines which method is faster. On VPRs less than 2^7, the naive method beats the 1-permutable set of PEs. Once we reach a VPR of 2^7, or 128, records per PE, the 1-permutable-set-of-PEs method becomes faster than routing the records row-by-row.

A certain degree of optimization is built into the PE communication system. Given that these optimizations are implemented at a low level, whereas we must code in higher level MPL, it is impressive that our algorithm is able to outperform the naive

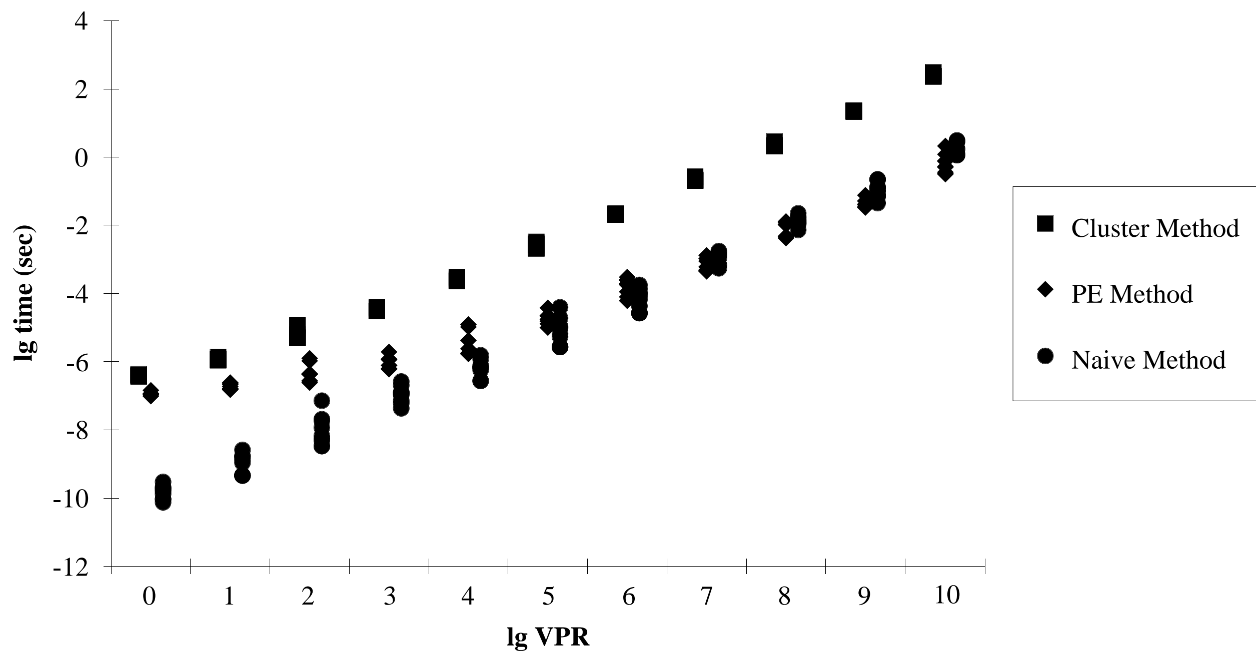

Figure 5.5: Scatter plot of running times for three methods, routing randomly generated nonsingular matrices.

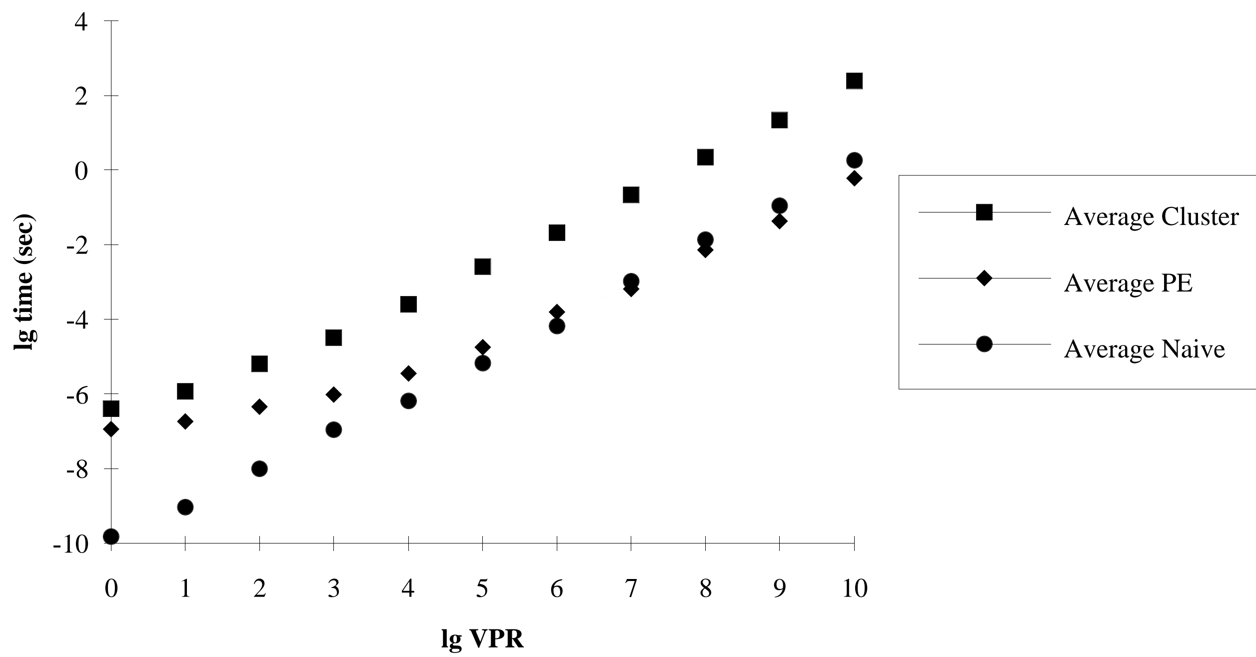

Figure 5.6: The average running times for the three methods, routing randomly generate nonsingular matrices.

method at all. If we could implement our permutable-set algorithm at a lower level, the crossover point would likely be at much lower VPRs.

Interestingly, we can observe that the algorithm computing the 1-permutable set of clusters is not competitive with the other two methods. Although this algorithm assures that that there is no congestion within the network, the duty cycle is too low for this method to compete. The overhead caused by the higher number of sets in the schedule more than makes up for the total lack of collisions.

Chapter 6

Conclusions

We have adapted Cormen's algorithm for performing BMMC permutations on parallel disk systems to work in the PE array of a DECmpp. This algorithm computes a schedule of 1-permutable sets of records, so that exactly one record is sent from and received into each independent device for each set of data records. In the PE array, each PE sends and receives one record for each set of the schedule. No more than 16 (the number of PEs per cluster) records can ever be routed to the same cluster or equivalently, the same network port, at once. When permuting data using a naive row-by-row routing method, the number of records through a single network port can be much higher. Thus, by limiting the contention for network ports, our algorithm decreases the total routing time for BMMC permutations. We can reduce the port contention even further by creating the 1-permutable set with only one record per cluster. This guarantees that there will be no contention for network ports during

routing. However, the overhead inherent in routing 16 times as many sets overshadows the time we gain from reduced contention.

Our algorithm must compute the first 1-permutable set of the schedule before we can do any routing. Since the naive method does not need to peform these calculations, on small source vectors the overall running time of this method is faster than when using our algorithm, despite the increased contention for network ports. For VPRs at least 2^7, or 128 records per PE, our algorithm beats the naive method in overall running time. Our algorithm requires extra space to hold the information on the schedule of 1-permutable sets; however, this space depends only the number of PEs in the array and is independent of the number of records in the source vector.

For routing BMMC permutations, our algorithm compares favorably with that given by Scherson and Subramanian. They must compute a bipartite edge coloring to determine the routing schedule before the beginning of their routing algorithm, an expensive calculation that they perform off-line. We are able to perform our entire algorithm on-line.

This thesis has provided

- a routing algorithm for BMMC permutations that beats the naive method on large VPRs,
- an understanding of the coding issues involved in implementing Cormen's algorithm, and

- an understanding of the relative speeds of built-in routing in comparison to MPL-coded routing.

We have seen the power of efficient algorithms. For sufficiently large VPRs, our adaption of Cormen's algorithm for BMMC permutations, coded in MPL, can outperform the naive method, even though the naive method's optimizations are implemented at a much lower level.

Bibliography

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

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

[Cor93] Thomas H. Cormen. Fast permuting in disk arrays. *Journal of Parallel and Distributed Computing*, 17(1–2):41–57, January and February 1993.

[CSW93] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Submitted to *IEEE Transactions on Parallel and Distributed Systems.* Preliminary version appeared in Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, 1993.

[Dig92a] Digital Equipment Corporation, Maynard, Massachusetts. *DECmpp Programming Language (ANSI) Reference Manual*, Version 3.1 Field Test edition, December 1992.

[Dig92b] Digital Equipment Corporation, Maynard, Massachusetts. *DECmpp Programming Language (ANSI) User's Guide*, Version 3.1 Field Test edition, December 1992.

[NV91] Mark H. Nodine and Jeffrey Scott Vitter. Large-scale sorting in parallel memories. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 29–39, July 1991.

[SS93] Isaac D. Scherson and Raghu 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, April 1993.

[Sub93] Raghu Subramanian. Private Communication, December 1993.

[VS90] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Optimal disk I/O with parallel block transfer. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 159–169, May 1990.

[Wis94] Leneord Wisniewski. Private Communication, April 1994.