An Efficient Method For Distributing Data In Hypercube Computers

De-Lei Lee
and
Mokhtar Aboelaze

Department of Computer Science
York University
North York, Ontario
Canada M3J 1P3

Abstract

In this paper, we propose a method for distributing data of an $N \times N$ matrix in an n-hypercube multiprocessor with $N = 2^n$ processors (n is even). This method is capable of distributing data among the processors' local memories so that the processors can have parallel access to rows, columns, contiguous blocks, and distributed blocks of the matrix without any memory contention. In accessing data, each processor can read/write the required element based only on the processor number and the instance of the data structure to be accessed (row number, column number, or block number), thereby requiring no communication among the processors for computing the addresses of the elements. We provide two techniques to permute the accessed data among the processors to accommodate applications with different computational structures. One technique maps the elements of any instance of the data structure into the processors in row-major order. The other technique maps neighboring elements into processors which are hypercube-neighbors. Routing algorithms are presented for each of these techniques, and our results indicate that $O(\log N)$ steps are required for all elements to reach their destinations.

Key Words: Communication overhead, hypercube computers, conflict-free data access, parallel data structures, routing algorithms

1. INTRODUCTION

High-performance computer systems are in great demand in areas such as aerodynamic simulations, quantum chemistry, structural analysis, weather forecasting, among others. Without high-performance computer systems, some of these areas cannot even be approached by engineers. However, with the transistor switching speed, and consequently the processor speed, approaching its physical limits, parallel processing is widely accepted as the only approach for building high-performance computer systems. However, parallel computing is, by no means, a problem free field. In decomposing an application program to run on a multiprocessor system, a communication overhead arises by the need of the different modules of the program to exchange data or to access data that resides in another processor's memory. This usually requires blocking the current execution of the program, preparing a message to send to another processor, sending the message through the interconnection network to the destination processor. Thus, causing a delay that is proportional to the amount of communication between the different processors. In order to fully utilize the power of parallel processing, the communication overhead should be kept minimum. Two factors should be considered when minimizing the communication overhead. The first is the way data is distributed among the processors (i.e. how frequent the processors are interrupted to communicate with each other), and the second is the interconnection network connecting the processors (i.e. how fast the message will be transmitted). In this paper, we consider this problem in the context of hypercube multiprocessors, and we describe an

This work was supported in part by the Natural Science and Engineering Research of Canada under grant NSERC-OGP0009196 and NSERC-OGP0043688.

efficient method for distributing data of matrices in order to minimize the communication overhead. We start our discussion with a very brief introduction to hypercube multiprocessors.

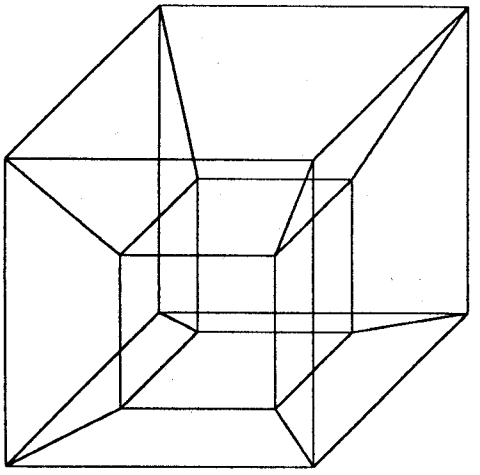

Figure 1

A 3-D view of the 4-hypercube.

An n-dimensional hypercube multiprocessor is a multiprocessor architecture consisting of $N = 2^n$ identical processors. Each processor has its own memory and works independently from the others. Processors exchange data by sending messages to each other. The nodes are numbered from 0 to 2^n-1. Two processors are connected if the binary representation of their numbers (addresses) differ by one and only one bit. This leads to a node connectivity of n, and a diameter of n (the diameter is the largest distance between two nodes). This hypercube network enjoys a very simple routing scheme. In routing, the processor has to compare the destination address, and its own address. Then it sends the message across one of the links that corresponds to a bit that differs in its binary address from the same bit in the destination address. Figure 1 shows a 4-hypercube graph with 16 nodes. The hypercube is a fixed connection design that has the ability to exploit particular topologies of problems in order to minimize communication cost [14]. With its excellent mapping capability, it easily maps common geometries such as grids, linear arrays or meshes in such a way as to preserve the original neighboring relations between processors. This explains in part the growing interest in using hypercube computers. A few machines based on the hypercube topology have been implemented and others are now being built [6], [10], [7]. However, in order to use a hypercube multiprocessor efficiently, we have to minimize the communication overhead, and the synchronization delay involved in the computation.

Kuck's study of parallelism suggested that rows, columns, diagonals, and square blocks of matrices are typical parallel data structures frequently encountered in large numerical applications. Budnik and Kuck in [3] proposed a storage scheme that allows a conflict-free access to these data structures by using a prime number of memory modules, which excludes the possibility of using all the nodes of a hypercube multiprocessor, since 2^n is even. Lawrie in [8] proposed a storage scheme that allows parallel, conflict-free access of these data structures by using twice as many memory modules as the number of processors. In the context of using hypercube multiprocessors, this means that in order to achieve conflict-free access in an N processor hypercube computer, only half of the processor can be effectively used. Batcher in [1] proposed a storage scheme that allows conflict-free access to rows, and columns, and the method is directly applicable to hypercube architecture. However, Batcher's method does not allow conflict-free access to contiguous blocks or distributed blocks. Lee in [9] proposed a storage scheme to distribute data in N memory modules, that allows conflict-free access to rows, columns, contiguous blocks, and distributed blocks. However, the work in [9] was for SIMD array processors based on a special purpose multistage interconnection network, which is drastically different from the hypercube interconnection network. For a review article about the limitations on the use of parallel memories the reader is referred to [13]

Although data alignment research was mainly performed for SIMD machines, where a central controller is controlling the whole system, and a single instruction is broadcast to all the processors to be executed simultaneously by all active processors. However, the same principle can be applied to MIMD multiprocessors as well. The goal is not for the data to be at the correct destination at the correct time for the next instruction to be executed, but rather to minimize the synchronization delays among the processors (synchronization delay is the delay when a processor blocks the execution of its module waiting to receive a piece of data from another processor).

In this paper, we present an efficient method for distributing data of $N \times N$ matrices in a n-hypercube computer with $N = 2^n$ nodes (n is even). Our method permits conflict-free access of rows, columns, contiguous blocks, and distributed blocks of the matrix. Each processor can calculate the local memory address it has to access to get the element of this data structure stored in its memory. We also want this calculation to be based

on its own processor number and the instance of the data structure, without a need of any global communication. After accessing the required element, each processor routes its element to the proper destination processor. We considered two cases for the *proper destination* to facilitate applications with different computational structures. The first is to route the elements to processors in a row-major order. i.e. element number i in the data structure is routed to processor number i. The second is to route the neighboring elements in the data structure to processors that are also neighbors in the hypercube. The calculation of the destination processor, is also based on local information without a need for any global communication. Routing algorithms are presented for both cases, our results indicate that $O(\log N)$ steps are required for data routing.

2. PROBLEM OBJECTIVE AND DEFINITIONS

Consider the problem of storing an $N \times N$ matrix A into the N memory modules of the N processors of n-hypercube multiprocessor, where $N = 2^n$. If the matrix is stored in the canonical way i.e. $A(i,j)$ is stored in the memory module of processor j, then it will be possible to fetch all the elements of a row simultaneously, because all the elements of any row lie in distinct memory modules. On the other hand, fetching a column is extremely slow, because all the elements of any column lie in one memory module, and fetching such a column requires N sequential fetching operations. In this paper, we are concerned with a storage scheme that allows conflict-free access to the rows, columns, contiguous blocks, and distributed blocks of a matrix A.

We define the sub-parts of the matrix we want to access as *templates*, in practice, the most frequently encountered templates in numerical computations are rows, columns, contiguous blocks, and distributed blocks. Following, we define these four templates. Let A represents a matrix, where $A(i,j)$ represents the element in row i and column j of A. Throughout the rest of this paper i or j represents a variable, while I or J represents a constant

Definition 1: For $0 \le I < N$, row I of a matrix A is defined as $\{A(I,j) \mid 0 \le j < N\}$

Definition 2: For $0 \le J < N$, column J of a matrix A is defined as $\{A(i,J) \mid 0 \le i < N\}$

Definition 3: For $0 \le I,J < \sqrt{N}$, contiguous block I,J of matrix A, is defined as $\{A(I\sqrt{N} + \alpha, J\sqrt{N} + \beta) \mid 0 \le \alpha,\beta < \sqrt{N}\}$. The contiguous block can be pictured as partitioning the matrix A into N smaller matrices, each is $\sqrt{N} \times \sqrt{N}$, and addressing these matrices by their relative positions in the A matrix.

Definition 4: For $0 \le I,J < \sqrt{N}$, distributed block number I,J of a matrix A, is defined as $\{A(I + \alpha\sqrt{N}, J + \beta\sqrt{N}) \mid 0 \le \alpha,\beta < \sqrt{N}\}$. A distributed block can be envisioned as follows, after partitioning the A matrix into $\sqrt{N} \times \sqrt{N}$ sub-matrices, we take the I^{th}, J^{th} element of each of these sub-matrices, and arrange them in a $\sqrt{N} \times \sqrt{N}$ matrix form.

Figure 2 shows 4×4 matrix. The matrix is divided into 4 contiguous blocks by the vertical and horizontal dashed lines. The elements of the distributed block 0,0 are shown inside circles while the elements of the distributed block 0,1 are shown inside squares.

$$A = \begin{bmatrix} (a_{0,0}) & [a_{0,1}] & (a_{0,2}) & [a_{0,3}] \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ (a_{2,0}) & [a_{2,1}] & (a_{2,2}) & [a_{2,3}] \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

Figure 2

4×4 matrix with distributed block 0,0 elements inside circles, and distributed block 0,1 inside squares

The need for parallel fetching of rows, columns, contiguous blocks, and distributed blocks is well-known for various common matrix operations and image processing operations [3] [5] [11]. We assume that the hypercube has N nodes, such that $N = 2^n$, and $n = 2k$. Following are two more definitions we use throughout the rest of this paper.

Definition 5: $I:J$ is the number formed by the concatenation of the binary representation of I followed by the binary representation of J. i.e. if $I = 0010$, and $J = 1100$ then $I:J = 00101100$

Definition 6: Let the binary representation of I be n bits long. I_H is the high order $\frac{n}{2}$ bits of I, and I_L is the low order $\frac{n}{2}$ bits of I

In this paper, we are interested in evenly distributing the N^2 entries of A into the N processors of the n-hypercube computer in such a way that, (a) each processor stores N entries, (b) the processors have a conflict-free access to rows, columns, contiguous blocks, and distributed blocks of the matrix, and (c) the accessed data can be routed to their destinations in parallel.

3. DATA ACCESS METHOD

To achieve conflict-free memory access to data, we assign element $A(i,j)$ to location i of the local memory of processor $m = \xi(i) \oplus j$, where $\xi(i) = i_L : i_H$, which interchanges the lower order $n/2$ bits with the higher order $n/2$ bits of i. For properties regarding conflict-free access to the various data of interest, proofs are contained in Appendix I.

The method of accessing data is as follows. When the system accesses a specific data structure (row I, column J, contiguous block I,J, or distributed block I,J), each processor receives (or internally generate) the index of this data structure (I,or J for rows or columns, or I,J for contiguous or distributed blocks). This index, together with the processor number, each processor can calculate the memory location it has to access to get the element of this data structure stored in its memory.

After accessing the data, each data element must be routed to the proper destination. In this paper, we consider two different *proper destinations* for the elements of the template.

The first is to route element i in any template to processor i, i.e., in a row-major order.

The second is to let the 2^n nodes of the hypercube to reconfigure in a linear array fashion for rows and columns, and to reconfigure in a $N^{1/2} \times N^{1/2}$ mesh for the contiguous, and distributed block, then the purpose is to map the element in the template to its corresponding node in the array or mesh. Using this method, the neighboring relation between the elements of a template is preserved (if two elements are neighbors in the template, they are mapped into two neighboring processors of the hypercube). In section 4 we will explain these two methods.

In order to read/write the required element, each processor must be capable of calculating the address of the data element in its own memory, given the index of the template to be accessed. The only information required is the index of the template (row I, column J, or contiguous or distributed block I,J) and the processor own number (address).

Lemma: In order for any processor m to access a specific element of any template, the address generation is as follows:

(1) For row I, each processor accesses memory location I

(2) For column J, each processor accesses memory location $\xi(m \oplus J)$.

(3) For contiguous block I,J, each processor accesses memory locations $I : m_H \oplus J$.

(4) For distributed block I,J, each processor accesses memory locations $m_L \oplus J : I$.

Where, m_H is the higher order $n/2$ bits of m, and m_L is the lower order $n/2$ bits of m.

Proof:
Note that $A(i,j)$ is stored in location i of the local memory of processor $\xi(i) \oplus j$.

(1). When accessing the I^{th} row, i.e., elements $\{A(I,j) \mid 0 \le j < N\}$ each processor knows that if it has $A(I,j)$ then it is stored in location I

(2). When accessing the J^{th} column i.e., $\{A(i,J) \mid 0 \le i < N\}$ each processor knows that element $A(i,J)$ is stored in location i (if it has it in its own memory). However each processor knows only the column number J, however, since $m = \xi(i) \oplus J$, by solving this equation for i, we get $i = \xi(m \oplus J)$, which is the memory location each processor should access to get an element of the J^{th} column.

(3). When accessing contiguous block I,J, notice that the elements of this contiguous block are $\{A(I\sqrt{N} + \alpha, J\sqrt{N} + \beta) = A(I:\alpha, J:\beta) \mid 0 \le \alpha,\beta < \sqrt{N}\}$. Notice that since both I, and α are $n/2$ bits numbers, $I\sqrt{N}$ is the same as I shifted $n/2$ bits to the left. Then, $I\sqrt{N} + \alpha = I : \alpha$. Each processor knows that its element of this contiguous block is stored in location $I:\alpha$, to get the value of α, each processor should solve the equation $m = \xi(I:\alpha) \oplus (J:\beta)$ yielding $\alpha = m_H \oplus J$ and the memory location to be accessed is $I:\alpha = I : m_H \oplus J$

(4). When accessing distributed block I,J, notice that the elements of this contiguous block are $\{A(I + \alpha\sqrt{N}, J + \beta\sqrt{N}) = A(\alpha:I, \beta:J) \mid 0 \le \alpha,\beta \le \sqrt{N}\}$. Notice that since both I, and α are $n/2$ bits numbers, $\alpha\sqrt{N}$ is the same as α shifted $n/2$ bits to the left. Then, $\alpha\sqrt{N} + I = \alpha : I$. Each element of this block is stored in location $\alpha:I$, to get α each processor has to solve the equation $m = \xi(\alpha:I) \oplus (\beta:J)$ yielding $\alpha = m_L \oplus J$, and the memory location to be fetched is $m_L \oplus J : I$

4. ROUTING METHODS

In this section, we discuss the routing of the data to its destination. After each processor fetches its element of the required templates, it should send it to the proper destination. we consider two different alternatives for destination processor.

4.1. Method 1

What we mean by the proper destination in this method is as follows. For rows and columns, the i^{th} element of the template (either a row or a column) is routed to processor number i, for example in 4×4 matrix, and fetching the 2^{nd} row, ($A(2,0)$,$A(2,1)$,$A(2,2)$,$A(2,3)$), element $A(2,0)$ is routed to processor 0, $A(2,1)$ is routed to processor 1, $A(2,2)$ is routed to processor 2, and finally $A(2,3)$ is routed to processor 3.

For the contiguous and distributed blocks, the templates are arranged as a $\sqrt{N} \times \sqrt{N}$ mesh, and the elements are mapped to the processors according to their position in the mesh, in a row-major fashion. For example, if we considered a 4×4 matrix, and we considered contiguous block 0,1. The elements of this block are $A(0,2)$,$A(0,3)$,$A(1,2)$,$A(1,3)$. Then element $A(0,2)$ is routed to processor 0, element $A(0,3)$ is routed to processor 1, element $A(1,2)$ is routed to processor 2, and element $A(1,3)$ to processor 3. If we consider the distributed block number 1,0, whose elements are ($A(0,1)$,$A(0,3)$,$A(2,1)$,$A(2,3)$). Then, $A(0,1)$ is routed to processor 0, $A(0,3)$ is routed to processor 1, $A(2,1)$ is routed to processor 2, and $A(2,3)$ is routed to processor 3.

For calculating the forwarding address for the different templates, for the rows, in fetching row I, element $A(I,j)$ is routed to processor j, which can be calculated by solving the equation $m = \xi(I) \oplus j$. yielding $j = m \oplus \xi(I)$. For columns, in fetching column J element $A(i,J)$ is routed to processor i, which can be calculated by solving the equation $m = \xi(i) \oplus J$, yielding $\xi(m \oplus J)$. For a contiguous block number I,J, $A(I{:}\alpha,J{:}\beta)$, $0 \le \alpha,\beta < \sqrt{N}$, or a distributed block number I,J, $A(\alpha{:}I,\beta{:}J)$, $0 \le \alpha,\beta < \sqrt{N}$. Since the element position in the template is simply α,β, then the target processor for this element should be $\alpha{:}\beta$. Both α and β can be calculated from the equation $m = \xi(I{:}\alpha) \oplus (J{:}\beta)$ for contiguous blocks, and $m = \xi(\alpha{:}I) \oplus (\beta{:}J)$ for distributed blocks. Table 1 shows the destination addresses for the previously mentioned templates. Notice that in calculating the destination processor for each data element, each processor just need to know its own number and the template number (I, J, or I,J)

The next step is to route each element to its destination. The way routing is done in a hypercube multiprocessor is by each processor examining the binary representation of its address, and the binary representation of the destination address, and forwarding the message to a link that corresponds to a bit that is different in these two binary representations. If we consider a completely synchronized system (SIMD), where it takes at least n routing steps to transfer any data such that in the first cycle each processor examine bit 0, and decide whether the message will be forwarded across link 0 or not, then in the second cycle, each processor examine bit 1 and so on. In this case the hypercube emulates a multistage banyan network, and it takes $2n$ to perform the previous routing. However, this may not necessarily be the case for MIMD systems, even for SIMD system there is no need for this restrictions. For example in the first cycle, each processor searches for the first bit that differ in its own address from the same bit in the destination address (which may be the first, second or the i^{th} bit) and forward the message across the link corresponding to this bit. In this case, it takes, in the average, less than $2n$ cycles. Figure 3 shows the simulation results for the relation between the cube size and the number of hops (which can be considered a crude measure of delay) for forwarding data for the four above mentioned templates

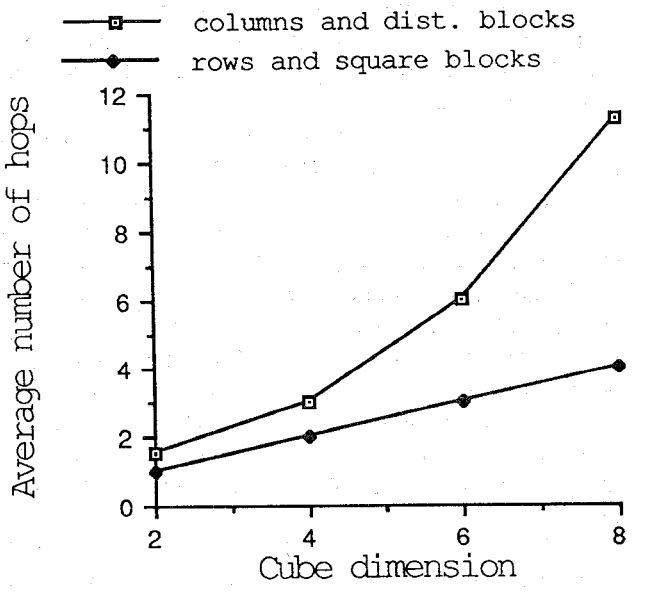

Figure 3

Number of hops vs the cube size for Method 1

Notice that in Figure 3, routing of rows and contiguous blocks requires the same number of hops, while columns and distributed blocks require substantially

more time.

4.2. Method 2

In this method we investigate another approach for data routing. One of the main advantages of the hypercube multiprocessors is its ability of mapping other graphs into the hypercube network. What we mean by mapping other graphs into hypercube is to find a one-to-one mapping between the nodes of the graph and the nodes of the hypercube in such a way to preserve the original neighboring relation in the graph (if two nodes are neighbors in the original graph, then, they are mapped into two neighboring nodes in the hypercube) [12], [2], [15].

The advantages of such mapping are two folds. First, it facilitate the design of algorithms for the hypercube, because if some algorithm has been developed to run on a different architecture, and this architecture can be mapped into hypercube. Then, with some minor modifications, this algorithm can be adopted to run on a hypercube multiprocessor. Second, some problems have a well-defined communication pattern that is different from the hypercube. Mapping this pattern into hypercube, results in substantial saving in communication overhead. For example consider solving elliptic differential equations [4] using iterative methods, all the communications are local between the grid points. If the grid can be mapped into hypercube, then all the communications are between neighboring nodes, which means less communication overhead. Another example is the 2-Dimensional convolution, where a 2-dimensional grid of the window size scans the image from left to right performing the required convolution. If the processors of a hypercube can be configured in a mesh of the same size as the window, this will facilitate loading the image to the different processors.

Since we are interested in this paper in fetching rows, columns, contiguous blocks, and distributed blocks. We investigate how to route these templates to processors in order to maintain the original neighboring relation. In other words, if two elements are neighbors in the row, column, contiguous block, or distributed block accessed, then, they should be forwarded into two neighboring processors in the hypercube. Essentially, we are interested in mapping linear arrays (for rows and columns) and meshes (contiguous and distributed blocks) into the hypercube, also we want the calculation of the destination address to be simple and does not require any global information.

Saad and Schultz in [12], proposed a mapping of the grid and ring (linear array is a ring without the connection between the first and last processor) into the hypercube. For mapping rings into hypercube, their idea depends on Gray coding. Basically, they considered a gray code of length 1 to be the two numbers 0 and 1, then to get Gray code of length 2, take the sequence (0,1) and insert 0 in front of each member, then, take the same sequence in reverse order and insert 1 in front of each element yielding (00 , 01 , 11 , 10). In general to construct Gray code of n bits, take a Gray code sequence of $n-1$ bits, insert 0 in front of each element, then take the same sequence in reverse order and insert 1 in front of each element. For Gray code of length 3, which can be done by taking Gray code sequence of length 2 (00 , 01 , 11 , 10) and insert 0 in front of each element, then take it in a reverse order and insert 1 in front of each element, yielding

000	001	011	010	110	111	101	100
↑	↑	↑	↑	↑	↑	↑	↑
0	1	2	3	4	5	6	7

Figure 4

Gray code of $0 \rightarrow 7$

which means that Gray code for 0 is 000, Gray code for 1 is 001, Gray code for 2 is 011 . . . etc. Notice that the hamming distance between any two numbers is 1, which means that using the mapping in Figure 4, the hyper cube is reconfigured into a linear array. In Figure 5 we present an algorithm to calculate a Gray code for for any number i of length n bits. The idea of the algorithm is better explained by an example using the above 3 bits Gray code. Assume that the number (i) is represented as $b_2 b_1 b_0$. First, we divide the 8 numbers into two groups, the first 4 numbers, and the last 4 numbers. Then if the number i is in the first half, then the M.S.B of its Gray code (b_2) is 0, else it is 1. Assume that we chose the number 4, since 4 is in the second half, then set $b_2 = 1$, then we consider the numbers in the second half (4,5,6,7), we divide the four numbers into two groups, the first two numbers, and the last two numbers. Since these four numbers have already been reversed during constructing the code, if the required number is in the first half, then the 2^{nd} M.S.B. of its Gray code is 1, else it is 0. Since 4 is in the first half of (4,5,6,7), we set $(b_1 = 1)$. Then we consider the lower half of these four numbers (4,5), we divide them into two groups, set $b_0 = 0$ for the lower half, and set $b_0 = 1$ for the upper half, i.e.we set $b_0 = 0$ yielding 110 as the gray code for 4. Figure 5 shows algorithm to calculate the Gray code of any length. Notice that this algorithm can be easily mapped into hardware, the flag variable can be represented by one bit register that is toppled from 0 to 1

or vice versa when incrementing the flag, and checking for even flag is simply checking for zero. Checking if i is in the lower half or upper half of the 2^j numbers can be simply achieved by checking bit number j of the binary representation of i

Algorithm Gray(i,n)
/* This algorithm calculates the gray code of a number i, with n bits binary representation */

```
begin
Flag = 0
for (j=n-1  0  step -1)
   begin
      if (flag = even) then
            begin
            if (i is in the lower half of the 2^(j+1) numbers)
            then
                  put b_j = 0
                  else
                        begin
                        put b_j = 1
                        increment flag
                        end
            end
            else
            begin
            if (i is in the upper half of the 2^(j+1) numbers)
            then
                  put b_j = 0
                  else
                        begin
                        put b_j = 1
                        increment flag
                        end
            end
      return (b)
end
```

Figure 5

Gray code of i of length n

For mapping a $N^{1/2} \times N^{1/2}$ square mesh into hypercube with N nodes, assume that each mesh point is labelled (x,y), where $0 \le x, y < N^{1/2}$. Notice that both x and y can be represented as $n/2$ bits binary number. Since any n-hypercube can be considered as the cross product of two $n/2$-hypercubes, in this case the mapping can simply be achieved by finding the $n/2$ bits Gray code of x ($Gray(x,n/2)$) and the $n/2$ bits Gray code of y ($Gray(y,n/2)$). Then (x,y) are mapped to the node number $Gray(x,n/2):Gray(y,n/2)$, where : means concatenation. Table 1. shows the destination processors for the different templates for Method 2. Figure 6 shows the simulation results for the average number of hops as a function of the cube size for routing the different templates in Method 2. Notice that the it required more time than Method 1 to perform the permutations.

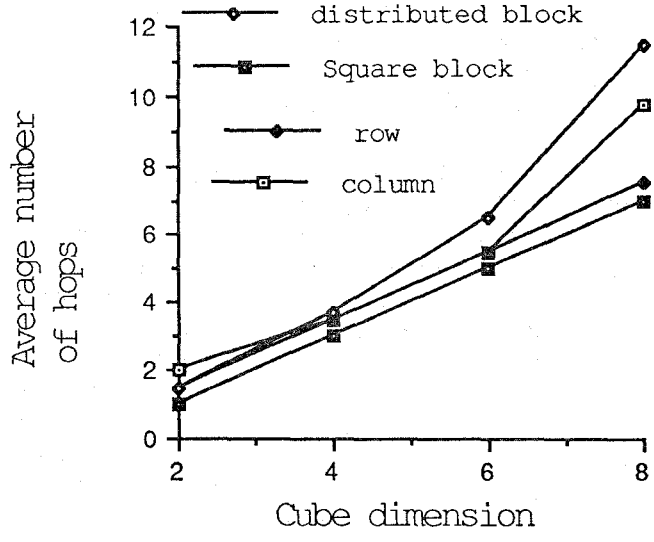

Figure 6

Number of hops vs cube size for Method 2

5. CONCLUSION

In this paper, we presented a method for distributing matrices into the memory modules of n-hypercube multiprocessors, that allows conflict-free access to rows, columns, contiguous blocks, and distributed blocks. Generation of memory addresses to access the elements of any instance of the data structure is simple and does not require any global communication among processors. We also investigated the delay (measured in number of hops) associated with sending the accessed elements to their proper destinations. Specifically, we have considered two different *proper destinations*. The first uses a row-major ordering based on the processors' numbers. (element number i in the accessed data structure is routed to processor i). The second involves ordering of the processors to preserve the neighboring relation between the elements of the data structure. (if two elements are neighbors in the data structure, then they are

Template	Location in local memory	Destination for Method 1
Row I	I	$m \oplus \xi(I)$
Column J	$\xi(m \oplus J)$	$\xi(m \oplus J)$
S-Block I,J	$I : m_H \oplus J$	$J \oplus m_H : I \oplus m_L$
D-Block I,J	$m_L \oplus J : I$	$J \oplus m_L : I \oplus m_H$

Template	Destination for Method 2
Row I	$Gray(m \oplus \xi(I), n)$
Column J	$Gray(\xi(m \oplus J), n)$
S-Block I,J	$Gray((J \oplus m_H), n/2) : Gray((I \oplus m_L), n/2)$
D-Block I,J	$Gray((J \oplus m_L), n/2) : Gray((I \oplus m_H), n/2)$

Table 1. The local address for accessing the different data structures (m is the processor number) and calculating the destination processor for Method 1 and 2. (Gray(i,n) is the function defined in Figure 5).

routed to two neighboring processors). A simulation results for the number of hops required to perform these two routings is presented.

6. REFERENCES

[1] Kenneth E. Batcher, "The Multidimensional Access Memory in STARAN," *IEEE Transactions on Computers*, February 1977, pp. 174-177.

[2] S. N. Bhatt and I. C. F. Ispen, *How to Embed Trees in Hypercubes*, Res. Rep. 443, Dep. Computer Science, Yale University, Yale University, 1985.

[3] Paul Budnik and David J. Kuck, "The organization and Use of Parallel Memories," *IEEE Transactions on Computers*, December 1971, pp. 1566-1569.

[4] Tony F. Chan, Youcef Saad, and Martin H. Schultz, "Solving Elliptic Partial Differential Equations on Hypercubes," in *Hypercube Multiprocessors 1986*, Michael T. Heath, ed., SIAM, 1986.

[5] P. E. Danielsson and S. Levialdi, "Computer Architecture for Pictorial Information Systems," *Computer*, Vol. 14, Nov. 1981, pp. 53-67.

[6] G. Fox, *The Performance of the Caltech Hypercube in Scientific Calculations*, Caltech Report CALT-68-1298, Caltech, 1985.

[7] INMOS Corp., *Transputer Reference Manual*, INMOS Corp., Colorado Springs, 1985.

[8] D. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers*, Vol. C-24, December 1975, pp. 1145-1155.

[9] D-L. Lee, "Scrambled Storage for Parallel Memory Systems," *Proc. of the 15th International Symposium on Computer Architecture*, May 1988, pp. 232-239.

[10] NCUBE Corp., *NCUBE Handbook, version 1.0*, NCUBE Corp., Beaverton, Oregon, 1986.

[11] James M. Ortega and William G. Poole, Jr., *Numerical Methods for Differential Equations*, Pitman Publishing Inc., 1981.

[12] Youcef Saad and Martin H. Schultz, "Topological Properties of Hypercubes," *IEEE Transactions on Computers*, Vol. 37, July 1988, pp. 867-872.

[13] H. D. Shapiro, "Theoretical Limitations on the Efficient Use of Parallel Memories," *IEEE Transactions on Computers*, Vol. C-27, May 1978, pp. 421-428.

[14] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, Lexington Books, Lexington, MA, 1985.

[15] A. Y. Wu, "Embedding of Tree Networks into Hypercubes," *J. Parallel and Distributed Computing*, Vol. 2, 1985, pp. 238-249.

APPENDIX I

Property 1 : No two elements of the same column are in the local memory of the same processor.

Proof : By contradiction. Consider two elements in the J^{th} column $A(i_1, J)$ and $A(i_2, J)$ according to the above scheme, these two elements are mapped into memories of processors m_1 and m_2 respectively, Assume that $m_1 = m_2$ i.e. $\xi(i_1) \oplus J = \xi(i_2) \oplus J$ therefore $\xi(i_1) = \xi(i_2)$ hence $i_1 = i_2$. □

Property 2 : No two elements of the same row are in the local memory of the same processor.

Proof : By contradiction. Consider two elements of the same row $A(I, j_1)$, and $A(I, j_2)$. According to the mapping, these two elements are mapped to processor $m_1 = \xi(I) \oplus j_1$, and $m_2 = \xi(I) \oplus j_2$ respectively, if $m_1 = m_2$ then $\xi(I) \oplus j_1 = \xi(I) \oplus j_2$, therefore $j_1 = j_2$ □

Property 3 : No two elements of the same contiguous block are in the local memory of the same processor.

Proof : By contradiction. From Definition 3, contiguous block $(I,J)\ 0 \le I,J < \sqrt{N}$ of matrix A is defined as $\{A(I\sqrt{N}+\alpha, J\sqrt{N}+\beta),\ 0 \le \alpha,\beta < \sqrt{N}\}$ where $0 < I,J,\alpha,\beta < \sqrt{N}$ which means that both I,J,α,β are $n/2$ bits long. Hence, $I\sqrt{N}$ is the same as I shifted $n/2$ bits to the left (multiplied by $2^{n/2} = \sqrt{N}$) i.e. $I\sqrt{N}+\alpha$ can be rewritten as $I:\alpha$. Similarly, $J\sqrt{N}+\beta$ can be rewritten as $J:\beta$.
Assume that two elements of contiguous block I,J $A(I\sqrt{N}+\alpha_1, J\sqrt{N}+\beta_1)$ and $A(I\sqrt{N}+\alpha_2, J\sqrt{N}+\beta_2)$ are mapped to the same memory module. i.e.

$$\xi(I\sqrt{N}+\alpha_1) \oplus (J\sqrt{N}+\beta_1) = \xi(I\sqrt{N}+\alpha_2) \oplus (J\sqrt{N}+\beta_2)$$

$$\xi(I:\alpha_1) \oplus (J:\beta_1) = \xi(I:\alpha_2) \oplus (J:\beta_2)$$

Since $\xi(I)$ is to switch the low order $n/2$ bits with the high order $n/2$ bits. The above equation can be rewritten as

$$(\alpha_1:I) \oplus (J:\beta_1) = (\alpha_2:I) \oplus (J:\beta_2)$$

$$(\alpha_1 \oplus J):(I \oplus \beta_1) = (\alpha_2 \oplus J):(I \oplus \beta_2)$$

For two binary numbers to be equal, the low order half of the first number should equal the low order half of the second number, and similarly for the high order half. Then

$$\alpha_1 \oplus J = \alpha_2 \oplus J \quad \text{and} \quad I \oplus \beta_1 = I \oplus \beta_2$$

$$\alpha_1 = \alpha_2 \quad \text{and} \quad \beta_1 = \beta_2 \qquad \square$$

Property 4 : No two elements of the same distributed block are in the local memory of the same processor.

Proof : By contradiction. From Definition 4, the elements of a distributed block $(I,J)\ 0 \le I,J \le \sqrt{N}$ of matrix A are $\{A(I+\alpha\sqrt{N}, J+\beta\sqrt{N}),\ 0 \le \alpha,\beta < \sqrt{N}\}$. Using the same logic as in property 2, this can be rewritten as $A(\alpha:I,\beta:J)$, assume that two elements of the same distributed block, $A(\alpha_1:I,\beta_1:J)$ and $A(\alpha_2:I,\beta_2:J)$, are mapped into the same processor memory, i.e.

$$\xi(\alpha_1:I) \oplus (\beta_1:J) = \xi(\alpha_2:I) \oplus (\beta_2:J)$$

$$(I:\alpha_1) \oplus (\beta_1:J) = (I:\alpha_2) \oplus (\beta_2:J)$$

$$(I \oplus \beta_1):(\alpha_1 \oplus J) = (I \oplus \beta_2):(\alpha_2 \oplus J)$$

$$I \oplus \beta_1 = I \oplus \beta_2 \quad \text{and} \quad \alpha_1 \oplus J = \alpha_2 \oplus J$$

$$\beta_1 = \beta_2 \quad \text{and} \quad \alpha_1 = \alpha_2 \qquad \square$$