U.C.L.A.
COMPUTATIONAL AND APPLIED MATHEMATICS

On Gray Code Mappings for Mesh-FFTs on Binary N-Cubes

Tony F. Chan

April, 1987

CAM Report 87-02

Department of Mathematics
University of California, Los Angeles
Los Angeles, CA. 90024

1. Introduction

A d-dimensional binary n-cube, sometimes called the hypercube, is a graph of 2^d vertices, each numbered with a distinct positive integer less than 2^d-1, with an edge between two vertices if and only if the Hamming distance (the number of bits that differ) between the binary representation of their vertex numbers is equal to one. With the recent advent in parallel computing, the hypercube has become a very popular topology for building multiprocessor parallel computers, with the vertices correspond to processors and the edges correspond to communication channels. Part of the reason often quoted for this choice of topology is the fact that many other topologies, such as meshes, trees, pyramids and butterflies, are embeddable in the binary n-cube [7]. By this we mean the graphs representing these other topologies can be mapped onto the binary n-cube graph with small dilation (the relative increase in the distance between vertices) and expansion (the relative increase in the total number of vertices) [6]. In practice, this implies that computations involving data flow graphs taking the form of the previously mentioned topologies can be carried out with minimum communication overheads on binary n-cube computers.

Unfortunately, what is often not appreciated is the fact that different graphs require different mappings. Moreover, in applications where the same data set must be operated on by more than one algorithm, it is possible that the optimal mappings for the individual algorithms are not compatible with one another. What is ideal for one could be disasterous for the others.

The main goal of this article is to point out one such example involving the FFT butterfly and the nearest neighbor mesh. The binary n-cube is an ideal FFT machine. However, the usual mapping for FFT is quite bad for performing nearest neighbor computations on a mesh, as we shall show later. This poses a dilemma in applications in which both FFT and NN computations are required on the same data, such as in algorithms for solving the partial differential equations of fluid dynamics. We shall show that a particular mapping which is ideal for the NN mesh, namely one employing the binary reflected Gray code (BRGC), is also suitable for FFT computations. If the data is mapped onto the cube using this mapping, then FFTs can be performed on the same data with data transfers limited to at most two vertices apart. That is, the BRGC mapping maps the FFT butterfly onto the cube with dilation bounded above by 2. Moreover, this property is independent of the size of the data array. We also derive a model for the communication cost which can be used to decide which mapping minimizes the communication cost in a particular application and whether a conversion between the two mappings is cost effective.

The BRGC possesses a special property, not shared by other Gray codes, which is very useful in many algorithms for solving partial differential equations. The idea of exploiting this property for FFTs on hypercubes seems to have occurred independently to several other researchers. For example, this possibility was mentioned by Johnsson in [11], although the context did not arise within NN-mesh computations.

Chamberlain [10] also pointed out the advantage of using the BRGC mapping for mesh-FFTs and performed some numerical experiments on the Intel iPSC. He did not derive a timing model nor did he consider the conversion between the two mappings. We will compare our model to his timing results in this paper.

In section 2, we review the FFT and its optimal mapping onto the binary n-cube. The mapping for the NN mesh is discussed next in section 3. The incompatibility of these two mappings is explained in section 4. The use of the BRGC for FFT is discussed in section 5. In section 6, we present the conversion algorithm and in section 7 we briefly discuss extensions to larger problems. We derive the communication cost model in section 8 and compare this to the timing results of Chamberlain. We close with some remarks in section 9. We shall restrict our discussions to the radix-2 FFT algorithm.

2. The FFT Mapping

The data flow graph for the FFT is usually referred to as the "butterfly", an example of which is shown in Fig. 1 for an array of size 8. Both the decimation in time and decimination in frequency form of the FFT algorithm has the same data flow graph. More generally, the data flow for an array of size 2^d can be described through the binary representation of the indices of the array elements. Let the array elements be x_j, where the index j ranges from 0 to 2^d-1. These array elements are updated at each of d stages of the computation. At the i-th stage of the computation, the array element with index j must communicate with another element with index k, whose binary representation differs from that of j in the i-th most significant position. On a hypercube, the natural mapping is to map x_j to node number j of the cube (see for example [2]). To be mathematically precise, let $M:I \rightarrow N$ be the class of mapping functions, where I denote the set of indices of the array elements and N the set of node numbers of the hypercube. Then the FFT mapping $f \in M$ is given by

$$f(j) = j. \tag{1}$$

With this mapping, it is easy to see that at every stage of the computation, the necessary communication will be between neighboring nodes. Moreover, all such communications at a given stage can be carried out in parallel. Each stage can be viewed as "collapsing" the hypercube in one of its coordinates. In fact, the hypercube is *isomorphic* to the butterfly of the same dimension. An illustration of this mapping on the cube is illustrated in Fig. 2 for the case $d = 3$.

Note that after the completion of the forward transform, the data elements are not arranged in the natural order. Rather, they are in what is known as bit-reversed order. This is unimportant if the inverse transform of the data array is to be computed next, perhaps after some computations on the transformed array itself, which is typical in many applications. If needed, the array elements can be permuted into natural order in d steps with only nearest neighbor communication by reversing the steps of the butterfly [8].

3. The NN Mesh Mapping

Another very common data flow graph is the nearest neighbor (NN) mesh, which occurs naturally in the solution of partial differential equations. We shall limit our discussion here to one dimensional meshes, since higher dimensional meshes can be easily built from tensor products of one dimensional ones[7]. If the array elements are denoted by x_j, then the NN-mesh graph with the x_j's as vertices contains all edges connecting a given vertex x_j with its nearest neighbors on the mesh. In one dimension, these are the two vertices x_{j-1} and x_{j+1}. (Throughout this paper, all indices are to be taken modulo 2^d, where d is the dimension of the hypercube.)

It is well-known that the NN-mesh graph with 2^d vertices can be mapped into a d-dimensional hypercube with no dilation or expansion via Gray codes. We shall review this construction briefly.

A d-dimensional Gray code is a sequence of 2^d distinct d-bit numbers $G_d = [g_0, g_1, \ldots, g_{2^d-1}]$, with the property that the Hamming distance between any two consecutive (cyclically) members of the sequence is equal to one. For example,

$$G_3 = [\ 000,\ 001,\ 011,\ 010,\ 110,\ 111,\ 101,\ 100] \tag{2}$$

is a 3-dimensional Gray code. There are many efficient algorithms for generating Gray codes [5]. On the other hand, Gray codes are not unique. For example, the following is a Gray code different from G_3:

$$H_3 = [\ 000,\ 001,\ 011,\ 111,\ 101,\ 100,\ 110,\ 010]. \tag{3}$$

The natural mapping $m \in M$ for the NN-mesh to a d-dimensional hypercube is

$$m(j) = g_j, \tag{4}$$

where g_j is the j-th member of a d-dimensional Gray code. Due to the defining property of Gray codes, with this mapping all nearest neighbors on the mesh are mapped onto nodes on the hypercubes at distance one apart. It is important to note that any d-dimensional Gray code will work. Figure 3 shows the mapping m for a 3-dimensional hypercube using the Gray code G_3. The arrows indicate the Hamiltonian circuit through the hypercube formed by the mesh.

4. The Incompatability Of the FFT and the NN-Mesh Mappings

It appears that the mapping f is ideal for mapping data onto the hypercube for fast Fourier transforms - the data flow graph of the FFT is mapped directly into the hypercube with no dilation or expansion. Similarly, the mapping m is ideal for mapping data onto the hypercube for NN-mesh computations. Unfortunately, the mappings f and m required for each kind of computation are different! This creats a potential problem in applications where the same data must be operated on by both the FFT and the NN-mesh algorithms. In fact, the most natural mapping for the FFT algorithm,

namely f, turns out to be inefficient for NN-mesh computations. This is directly related to the following property of the hypercube.

Theorem 1. In a d-dimensional hypercube, the maximum Hamming distance between nodes with consecutive node numbers is equal to d.

Proof. It can be verified that $dist(2^{d-1}-1, 2^{d-1})=d$.

The practical significance of the above theorem is that if the data array x_j is mapped onto the hypercube using the FFT mapping f, then a nearest neighbor computation on the same data array requires communication over a distance d for some nodes. For higher dimensional hypercubes, this may cause a significant reduction in the efficiency of algorithms using the nearest neighbor data flow graph.

Note that the Hamming distance between the two specific neighbors on the NN-mesh, namely node $2^{d-1}-1$ and node 2^{d-1}, is equal to d, the *diameter* of the hypercube, which is the *maximum* distance between *any* two nodes on the hypercube. Thus, as far as the NN-mesh computation is concerned, the FFT mapping f is the worst possible mapping.

Applications employing both the FFT and the NN-mesh data flow graphs are very common, especially if they involve the solution of partial differential equations. Computational fluid dynamics is one such example. A finite difference discretization of the Navier Stokes equation often produces a nearest neighbor stencil. Thus the NN-mesh mapping is natural for computation of residuals. On the other hand, if a spectral method or a fast Poisson solver is used to solve the difference equations, then the FFT mapping is called for. The most natural mapping for one operation is inefficient for the other.

A natural solution to this dilemma is to rearrange the data before the execution of each algorithm according to the most natural mapping for that algorithm. This may be cost effective if the algorithm is to be executed many times over between data rearrangements. However, there is still a cost for such rearrangements and it would be preferable to employ a mapping that is almost optimal for both algorithms, thus avoiding any unnecessary data movements. We shall show in the next section that this is possible.

5. The Binary Reflected Gray Code Mapping

We have shown in the last section that the natural FFT mapping f is not suited for NN-mesh computations. A natural question at this point is whether the natural NN-mesh mapping m is suited for FFT computations. We shall show in this section that the answer to this question is affirmative, provided we choose the mapping m in a special way.

Recall that the mapping m is defined in terms of Gray codes. Since Gray codes are nonunique, so is the mapping m. For NN-mesh computations, it does not matter which of the many possible mappings for m we use. It turns out, however, that many of these allowable mappings for m are unsuitable for FFT computations. Fortunately, and somewhat surprisingly, one can always find a mapping m that is also suitable for FFTs. This is precisely the mapping m defined by a special Gray code, namely the Binary Reflected Gray Code (BRGC), as we shall show.

The BRGC can be recursively defined as follows:

$$B_1 = [\,0,\,1] \tag{5}$$

$$B_{i+1} = [\,0B_i,\,1B_i^R], \tag{6}$$

where $0B_i$ denotes the sequence obtained by prefixing each member of B_i by 0 and B_i^R denote the sequence obtained by reversing the order of B_i. The first few BRGC's are shown below:

$$B_1 = [\,0,\,1] \tag{7}$$

$$B_2 = [\,00,\,01,\,11,\,10] \tag{8}$$

$$B_3 = [\,000,\,001,\,011,\,010,\,110,\,111,\,101,\,100] \tag{9}$$

$$B_4 = [0000,0001,0011,0010,0110,0111,0101,0100,$$
$$1100,1101,1111,1110,1010,1011,1001,1000]. \tag{10}$$

It is easy to prove by induction that B_i as defined above are Gray codes. One simply observes that in G_{i+1}, if a neighboring pair of members lie completely in the first or the second half, then the Hamming distance between them is the same as that between the corresponding members of G_i, which by induction is equal to one; and if the pair "straddles" the two halves, then they also differ by one bit because they are obtained from the last member of G_i by prefixing a 0 and a 1.

What distinguishes the BRGC from other Gray codes is the following property.

Theorem 2. Let B_d=[b_0, b_1,..., b_{2^d}−1] be a d-dimensional BRGC, then for $0 \le i < 2^d$,

$$dist(b_i, b_{i+2^j}) = \begin{cases} 1 & \text{if } j=0 \\ 2 & \text{if } j>0 \end{cases} \tag{11}$$

where all subscripts are to be taken modulo 2^d.

Proof. A proof can be found in [4].

The above theorem states that if we take an arbitrary member b_i of the BRGC B_d, then it is at a distance one away from its neighbor b_{i+1} (the case $j = 0$, which follows

directly from that fact that B_d is a Gray code), and, more importantly, that if one takes strides of increasing powers of 2 from b_i (the case $j > 0$), then the distance remains exactly 2, no matter how large the stride or the dimension of the BRGC is. This important property of the BRGC can be viewed as a property of a regular global connection between the members of the BRGC. Many algorithms can take advantage of this global connectivity, such as cyclic reduction algorithms for solving tridiagonal linear systems [3] and multigrid algorithms for solving partial differential equations [1].

We emphasize that not every Gray code possess this very useful property. For example, the following 4-dimensional Gray codes does not:

$$G_4 = [0000,0001,0011,0010,0110,0111,0101,1101,$$

$$1111,1110,1010,1011,1001,1000,1100,0100], \qquad (12)$$

because $dist(0000,1111) = 4$.

It is interesting to note that the BRGC happens to be the most commonly used Gray code in the literature for mapping NN-meshes onto hypercubes, although the global connectivity property stated in Theorem 2, which is relatively little known, is seldom the reason for this choice.

To see that this property is also useful for FFT's, we note that the FFT butterfly involves pairs of array elements separated by strides of diminishing powers of 2 and thus if the array elements are mapped to the hypercube using a BRGC, then the communications occur between nodes at most a distance two apart. This leads us to the following theorem.

Theorem 3. The FFT butterfly graph with 2^d vertices is mapped into the d-dimensional hypercube with no expansion and with dilation bounded above by 2 under the mapping $r \in F$ defined by $r(j) = b_j$, where b_j is the j-th member of the d-dimensional BRGC B_d.

Proof. Recall that at each stage of the FFT computation, the necessary communications are between pairs of array elements whose indices differ in exactly one bit in their binary representations. It is easy to verify that if it is the i-th least significant bit that differs, then the array elements are at a stride of 2^{i-1} apart. Under the mapping r, the nodes where they reside are at a distance at most 2 apart, due to the above mentioned property of the BRGC.

It is easy to verify the above result for the case $d = 3$ as illustrated in Figure 3. (Note that G_3 is a BRGC.) For example, the element x_0 communicates with x_4 at the first stage of the FFT and with x_2 at the second stage, both at distance 2 away. At the final stage, it communicates with x_1 which is distance 1 away.

6. Conversion Between the BRGC and FFT Mappings

As mentioned earlier, one can consider rearranging the data before the execution of each algorithm according to the most natural mapping for that algorithm, especially if the algorithm is to be executed many times. In this section, we shall present an algorithm, adapted from one given by Johnsson [11], for accomplishing such a conversion between the FFT and the NN-mesh mappings.

We shall consider converting from the BRGC mapping to the FFT mapping. Conversion in the other direction can be performed by reversing the steps.

Lemma. A BRGC mapping can be converted to a FFT mapping in $d-1$ edge disjoint exchanges in a d-dimensional hypercube.

Proof [11]. The proof is by induction and is constructive. The lemma can be verified to be true for the 1-cube where no exchanges are necessary. Assume it is true for a k-cube. For a $(k+1)$-cube, the BRGC maps the data x_j's to the processors

$$[0b_0,0b_1,...,0b_{2^k-1},1b_{2^k-1},\cdots 1b_1,1b_0],$$

where the b_i's are the k-dimensional BRGCs. The first half of the data is in the k-dimensional subcube with leading bit 0, mapped according to the k-dimensional BRGC. By the induction hypothesis, they can be converted to the FFT mapping by $k-1$ edge disjoint exchanges. The second half of the data in the subcube with leading bit 1 are mapped according to the *reversed* k-dimensional BRGC. Performing 2^{k-1} pairwise edge disjoint exchanges between the pairs $(1b_{2^k-1},1b_0),(1b_{2^k-2},1b_1)$ etc., which corresponds to an exchange in the leading bit of the subcube, the data can be brought into a configuration corresponding to a mapping by the k-dimensional BRGC. By the induction hypothesis, a further $k-1$ edge disjoint exchanges completes the conversion, making a total of k such exchanges.

The algorithm for the conversion can be summarized as follows. Let the binary representation of the node number of a particular node be $i = i_0 i_1 \cdots i_{d-1}$. The algorithm consists of $d-1$ exchanges. At the j-th stage, if $i_j=1$ then the data in that node is exchanged with the data in the neighboring node in the $(j+1)$-th dimension.

7. Larger Problems

All our mappings and algorithms extend straighforwardly to situations where there are more data elements than the number of nodes. Suppose that we have 2^n data and a 2^d cube. We divide the data array into 2^d equal parts, with consecutive 2^{n-d} elements grouped together into blocks. Then each blcok of elements can be mapped onto the cube using any one of the mappings presented earlier. For the first d stages of the FFT, each block of elements must be communicated to a neighboring node. For the subsequent $n-d$ stages, no internode communications are needed. For NN-mesh

computations, only a boundary data element needs to be communicated.

Finally, higher dimensional meshes can be mapped using tensor products of one dimensional mappings [1,4,7,11]. The mapping for a point in the mesh is obtained by concatenating the binary representation of the one dimensional mappings of each of its coordinates. Since the FFT and the NN-mesh data flow graph in each coordinate direction is mapped into subcubes, all properties of the one dimensional mappings are preserved. The conversion algorithm can also be extended by applying the one dimensional version in each coordinate in succession.

8. Timing Analysis

In this section, we shall give a performance analysis for the various algorithms that we have given earlier. We shall consider a one dimensional data array with 2^n elements on a d-cube, mapped in groups of $q = 2^{n-d}$ consecutive elements as described in the last section. We shall let s, c denote respectively the startup time for internode communication and the communication rate per data element. We shall also assume bidirectional communication over the same channel and therefore an exchange actually takes the same time as performing just one send.

8.1. Timing for FFT

We first consider the FFT computation. Let a denote the computational time at each node at each stage of the FFT computation, Then the time for an FFT on the array is given by

$$T_{\text{fft}} = qn\ a+(d-1)(s+qc)\delta+(s+qc)$$

where $\delta = 1$ under the FFT mapping f and $\delta = 2$ under the BRGC mapping r. The last term on the right reflects the fact that the last stage of the FFT involves only nearest neighbor communication under either mapping. For the FFT mapping, models similar to ours have been used by Swarztrauber [9] and Walton [12].

Note that the computational time decreases with d while the communication time may increase with d. Thus an optimal value of d can be found which minimizes the parallel execution time. Swarztrauber [9] has carried out a similar analysis for the usual FFT mapping f and he found that, with the optimal number of processors, the communication time actually dominates the computational time. It is therefore important to consider communication efficient mappings of the data. With the BRGC mapping, the optimal value of d is lower than with the FFT mapping because the communication is more costly. In Figure 4, we plotted the times for a 1024-point FFT (n=10) using a set of machine parameters considered by Swarztrauber [9]. It is seen that with the optimal values of d, communication times dominate the computation times. However, for a large range of values of d less than the optimal values, the communication times are significantly lower than the computational times. In other words, for large

values of n/d, relatively high efficiencies can be achieved, a fact also observed by Walton [12]. It must be emphasized that perfect efficiency cannot be achieved however large n/d is because the ratio of computation time to communication time does not tend to zero. Finally, it can be seen from the plots that for large values of n/d, the times for the BRGC mapping is only slightly higher than that for the FFT mapping. The conclusion is that for an efficient FFT computation, the overhead of the BRGC mapping over the FFT mapping is not significant.

Next, we compare our model to the timing results obtained by Chamberlain [10] on the Intel iPSC for a 4096-point FFT using both the FFT mapping and the BRGC mapping. These results are tabulated in Table 1.

d	FFT Mapping	BRGC Mapping
0	28240	28240
1	13456-15120	13456-15152
2	6336-8048	6544-8240
3	2992-4272	3040-4384
4	1424-2256	1520-2352
5	672-1232	720-1296
6	352-720	368-800

Table 1. Timings (in milliseconds) for a 4096-point FFT (Chamberlain's results)

Due to imperfect load balancing, some nodes finishes before others and the two values given in the table are for the fastest processor and the slowest processor. To compare with our model, we have to use a reasonable set of machine parameters. For the arithmetic time a, we took the single processor time and divided that by qn, resulting in a value of a=0.575 milliseconds. This accounts for the actual arithmetic speed achieved with the compiled code (versus the peak flops rating of the arithmetic processor.) For the communication parameters s and c, we used the timing results obtained at Yale by Saied [13]: a startup cost of 6 milliseconds and a transfer rate of 1 microsecond per byte and unidirectional channels. (These figures are for the first iPSC systems delivered in late 1985. Intel has since improved the communication timing.) This translates to the values s=12 milliseconds and c=16 microseconds for 32-bit numbers. Using this set of machine parameters, the predicted times for the model are then computed for various values of d and compared with the *averaged* values in Table 1. (This is a fair comparison because our model does not take into account the load imbalance which caused the spread in the timings in Table 1.) The speedup factors are tabulated in Table 2 and plotted in Figure 5. The model agrees rather well with the timing results.

	FFT Mapping		BRGC Mapping	
d	Model	Experiment	Model	Experiment
1	1.99	1.98	1.99	1.98
2	3.97	3.93	3.95	3.82
3	7.86	7.78	7.78	7.61
4	15.44	15.36	15.04	14.60
5	29.66	29.69	28.01	28.04
6	54.35	52.73	48.31	48.39

Table 2. Speed up comparison between model and Chamberlain experiment.

8.2. Timing for the Conversion Algorithm

Next we consider the cost of the conversion algorithm between the BRGC mapping and the FFT mapping. The algorithm in section 6 can be extended in two ways to the case of q data elements per node. The first method is simply to send the q elements at each step instead of one. The time for this is

$$T_c = (d-1)(s+cq).$$

The second method is to pipeline the data elements in each send [4,11]. In othe words, each send consists of only one element but each node will send q times. The motivation is that the processors will be kept busy more often because some sends can be overlapped. Since a total of $q+d-2$ exchanges are needed, the time for this method is

$$T_{cp} = (q+d-2)(s+c).$$

By comparing the two, we find that $T_c < T_{cp}$ if $s/c > d(1-1/q)-1$. If $q \gg 1$ then this condition reduces to $s/c > d-1$. For most commercially available hypercubes, $s/c \gg d-1$ and therefore the non-pipelining method is faster. The main reason is that the pipelining method has more startups which are costly on these machines. From now on we shall only use T_c.

8.3. Comparison of the Mappings

We now consider the question of which mapping is better for a given application and whether the conversion to the optimal mapping is worthwhile before we execute an algorithm.

Consider the situation in which we have an application which consists of k FFTs then followed by l NN-mesh computations. We shall consider the communication time of the various possibilities. We assume that one NN-mesh communication consists of sending one boundary data to a neighbor.

If we use the BRGC mapping for both computations with no conversion, then the communication time is given by

$$T_b = l(s+c)+k(2d-1)(s+qc).$$

The factor $(2d-1)$ in the second term on the right corresponds to a dilation of 2 for the first $d-1$ steps of the butterfly and a dilation of 1 for the final step.

If we use the FFT mapping with no conversion, the time is

$$T_f = ld(s+c)+kd(s+qc).$$

The extra factor of d in the first term corresponds to the dilation of mapping the NN-mesh graph to the hypercube using the FFT mapping.

If we use the optimal mapping for each computation and convert between them, the time is

$$T_c = l(s+c)+2(d-1)(s+qc)+kd(s+qc).$$

The second term in the right hand side corresponds to the conversion time.

For given values of k and l, the above timing formulas can be used to determine which mapping/conversion combination has the least communication cost. We have the following general results.

Theorem 4

$$T_c < T_b \text{ if } k > 2 \text{ for any } l,$$

$$T_c < T_f \text{ if } l > 2(s+qc)/(s+c) \text{ for any } k,$$

$$T_b < T_f \text{ if } l/k > (s+qc)/(s+c).$$

We shall consider two extreme situations here. The first case is where l is much larger than k, in other words a primarily NN-mesh computation. In particular, if $l/k > (s+qc)/(s+c)$, then $T_b < T_f$ and so the BRGC mapping should be used for the NN-mesh computations. If in addition $k>2$, then $T_c < T_b$. In other words, we should convert the data to the FFT mapping if we are going to perform more than two FFTs. In many applications, an FFT is immediately folllowed by an inverse FFT and our result shows that in these cases it is not necessary to convert the data to the FFT mapping.

The second situation is where k is much larger than l which means that we have a primarily FFT computation. In particular, if $l/k < (s+qc)/(s+c)$, then the FFT mapping should be used for the FFT computations. Moreover, if $l > 2(s+qc)/(s+c)$, then $T_c < T_f$, and we should convert to the BRGC mapping for the NN-mesh computations.

9. Concluding Remarks

A typical parallel program contains several algorithms working on the same data. One of the main purposes of this article is to emphasize the need to go beyond studying the optimal implementation of the individual algorithms but to look at the whole collection as a whole. The reason is simply that what is best for one algorithm may not be optimal for the others. We have shown in this article one such example involving the implementation of the FFT algorithm and the nearest neighbor mesh algorithm on hypercubes. We demonstrated that the natural FFT mapping of data onto the hypercube is not perfectly suited for NN-mesh computations. On the other hand, the natural NN-mesh mapping using the special BRGC is quite suitable for FFT computations.

The optimal choice of a mapping depends on many factors, such as the frequency the algorithms are to be executed and the ratio of the arithmetic speed to the communication speed. Moreover, it may be cost effective to convert the data into the optimal configuration for a certain algorithm before executing it.

Since the BRGC mapping is also ideal for other algorithm for computations on meshes, such as cyclic reduction and multigrid algorithms, we feel that perhaps it should be considered in addition to the natural FFT mapping in applications involving FFT computations.

References

[1] T.F. Chan, Y. Saad, *Multigrid Algorithms on the Hypercube Multiprocessor*, Research Report YALEU/DCS/RR-368, February, 1985. IEEE Trans. on Comp., Vol. C-35, No.11, November 1986, pp. 969-977.

[2] G. Fox, S. Otto, *Decomposition of Scientific Problems for Concurrent Processors*, Physics Today, May, 1984.

[3] L.S. Johnsson, *Odd-Even Cyclic Reduction on Ensemble Architectures*, Research Report YALEU/DCS/RR-339, Oct. 1984.

[4] L.S. Johnsson, *Communication Efficient Basic Linear Algebra Computations on Hypercube Architecture*, Research Report YALEU/DCS/RR-361, Sept. 1985. To appear in J. Parallel and Distributed Computing.

[5] E.M. Reingold, J. Nievergelt, N. Deo, *Combinatorial Algorithms*, Prentice Hall, New York, 1977.

[6] A. Rosenberg, *Data Encoding and Their Costs*, Acta Inform. 9 (1978), pp.273-292.

[7] Y. Saad, M.H. Schultz, *Some Topological Properties of the Hypercube Multiprocessor*, Research Report YALEU/DCS/RR-428, October, 1985.

[8] J. Salmon, *private communication.* See also Chapter 8 of the book "*Solving Problems on Concurrent Processors*" by G. Fox et al, to be published.

[9] P. N. Swarztrauber, *Multiprocessor FFTs,* manuscript, June, 1986. Paper given at the International Conference on Vector and Parallel Computing, June 2-6, Loen, Norway.

[10] R.M. Chamberlain, *Gray Codes, Fast Fourier Transforms and Hypercubes*, Report CCS 86/1, Chr. Michelsen Institute, Bergen, Norway, May, 1986.

[11] L.S. Johnsson, *Data Permutation and Basic Linear Algebra Computations on Ensemble Architectures*, Research Report YALEU/DCS/RR-367, Feb. 1985.

[12] S. R. Walton, *Fast Fourier Transforms on the Hypercube*, paper delivered at the Second Hypercube Conference, Knoxville, Tennessee, September, 1986.

[13] F. Saied, *Private Communication.*

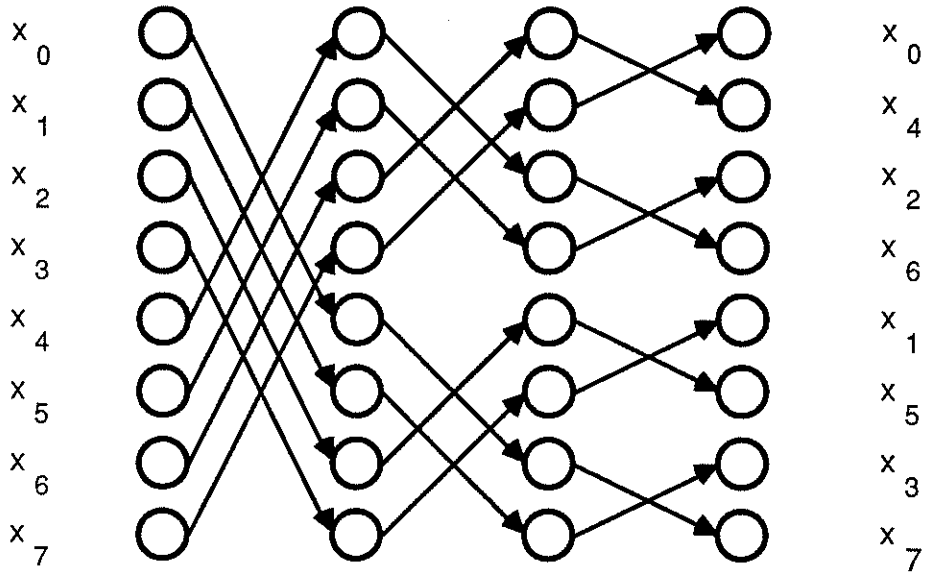

Fig. 1 The FFT "Butterfly"

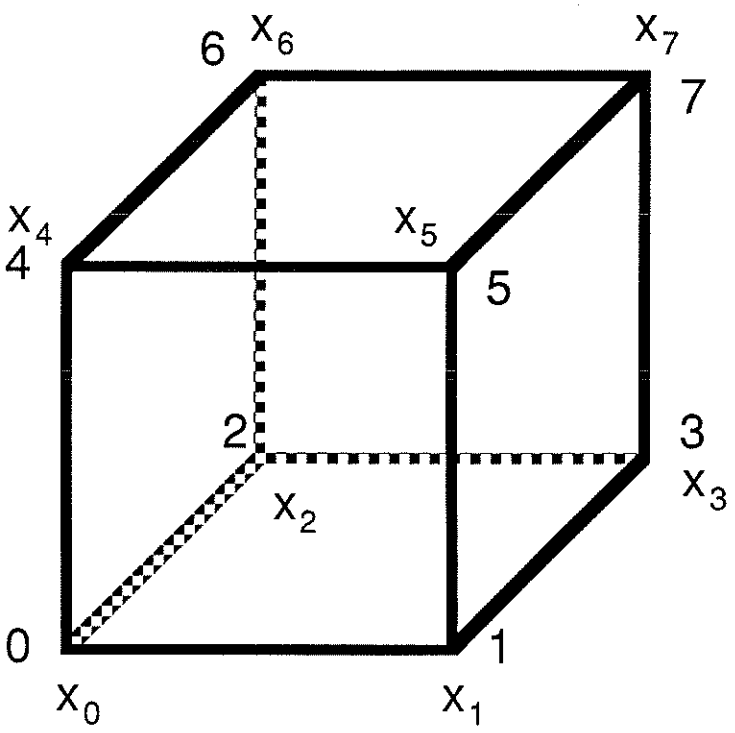

Fig. 2 The Natural FFT Mapping for the Hypercube

The x_i's denote array elements, the integers node numbers.

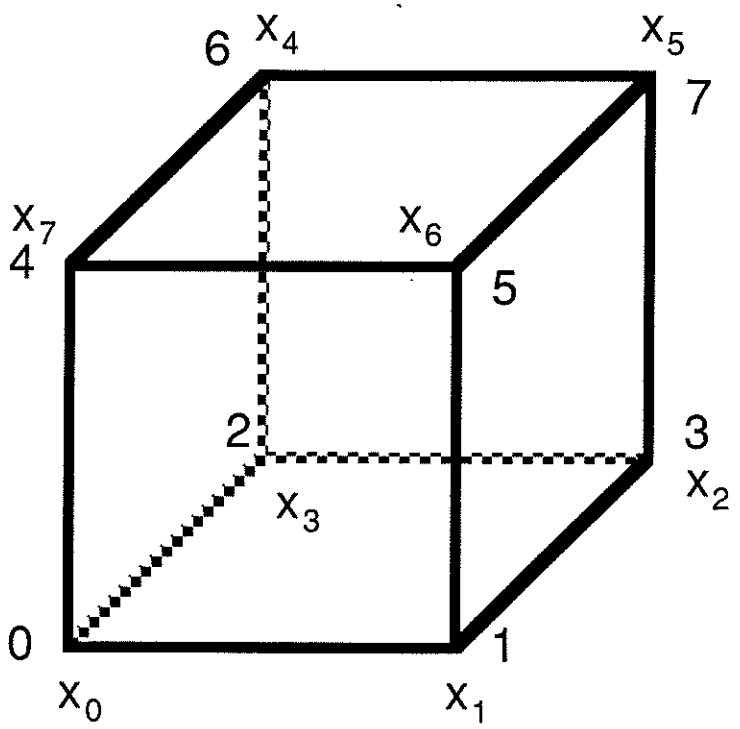

Fig. 3 The Gray Code NN-Mesh Mapping for the Hypercube

The x_i's denote array elements, the integers node numbers.

Fig. 4. Model Times for 1024-point FFT

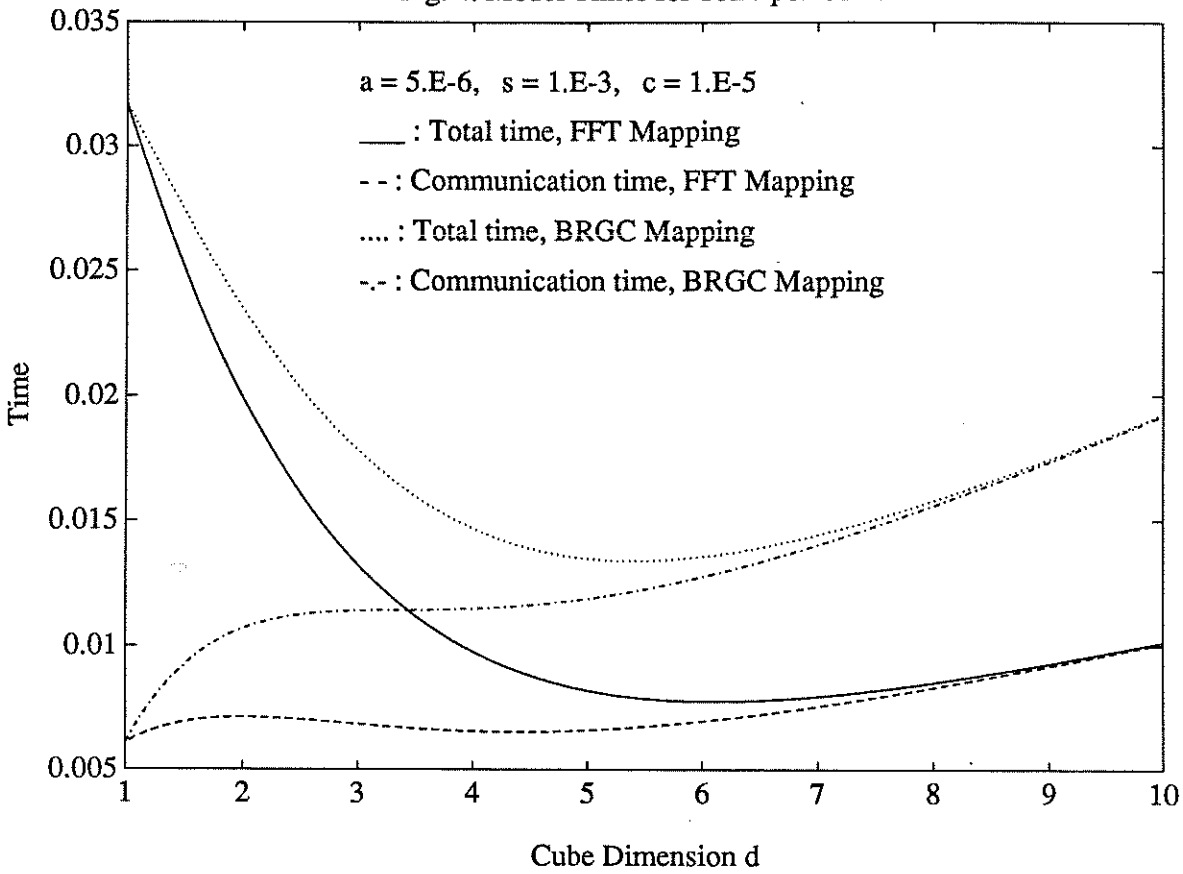

Fig. 5. Speed Up Comparison Between Model and Chamberlain Data