Load Balanced FFT Implementations on the Intel iPSC†

Clare Chu
87-824

March 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

†The work that produced this paper was [partially] completed with the assistance of computing facilities of the Advanced Computing Facility at the Cornell Center for Theory and Simulation in Science and Engineering, which is supported by the National Science Foundation and New York State.

Load Balanced FFT Implementations on the Intel iPSC

Clare Chu

Center for Applied Mathematics

Cornell University, Ithaca, NY

Dept. of Computer Science TR 87-824

March, 1987

Abstract: Two implementations of both the Cooley-Tukey and Gentleman-Sande radix-two FFT algorithms are described where the distribution of computational work among the processors is balanced, i.e. every processor does the same number of complex multiplications and additions. One method requires no extra storage space for the buffering of the complex data that is to be exchanged between processors therefore allowing "in-place" computation. The second implementation is useful for doing FFTs of symmetric sequences and an example of its use in the sine transform is presented. We also show how to simulate the second implementation in terms of the first to gain "in-place" computation without the need for buffering and intra-processor swapping.

The work that produced this paper was [partially] completed with the assistance of computing facilities of the Advanced Computing Facility at the Cornell Center for Theory and Simulation in Science and Engineering, which is supported by the National Science Foundation and New York State.

1. Introduction

The boolean n-cube or n-dimensional hypercube is a loosely coupled parallel processor with a versatile interconnection scheme rendering it suitable for many applications. The architecture is best described by numbering, from 0 to $2^n - 1$, the 2^n node processors by binary labels of length n and connecting two nodes only if their binary labels differ at exactly one bit. This gives the nodes a "power of two" adjacency scheme where node i and $i + 2^j$ are adjacent if they are in the same $(j + 1)$-dimensional sub-cube.

We shall see that this adjacency structure is isomorphic to the FFT butterfly data flow graph and that in a sense the hypercube is an ideal topology for the implementation of the FFT. Several authors have considered the implementation of the radix-two complex FFT in the hypercube, including Chamberlain (1986), Chan (1986), Swarztrauber (1986) and Walton(1986).

Effective use of a parallel machine to solve a numerical problem includes the distribution of the subtasks of the problem among the processors so that the work load for each processor is about the same. This is called load balancing. Another cost to consider is that of the communication required. Since the hypercube system does not incorporate shared memory, all communication is done by message passing. A measurement of the amount of computation done versus communication overhead time (which may involve the actual message passing time and also time spent waiting for a message to arrive) is called processor utilization—the percentage of time the processor is actually computing over the total time it takes for the algorithm to run.

In this paper we will examine methods of distributing the data inside the processors so that the amount of computational work for each processor is approximately the same. First we will implement the complex FFT in the conventional manner and compare the load imbalances between the various forms of the algorithm, whether Gentleman-Sande or Cooley-Tukey, multipliers ordered naturally or bit-reversed. Then we present a configuration for mapping data into the cube which is both load balanced and allows in-place computation of the FFT without requiring additional buffering space. Another load-balanced layout which proves useful in the finding of the FFT of symmetric sequences is also implemented and discussed. However we shall see that by redefining this second implementation in

terms of the first, we can do in-place computations without the need for additional buffering at a slight cost in communications time. Finally we shall see that mapping the nodes of the hypercube according to a Binary Reflected Gray Code, which is not isomorphic to the FFT butterfly signal flow graph, is a perfectly viable way to implement the FFT and is also better suited for general numerical computational problems involving FFTs and other computations.

2. The Intel iPSC Hypercube

The Intel iPSC system is a MIMD (multiple instruction, multiple data) parallel processor consisting of a Multibus-based System 286/310 microcomputer (cube manager) and computational units of 32 high-performance microcomputers interconnected in a hypercube topology. The cube manager is connected to each node microcomputer by an Ethernet communication link. Each node processor has its own numeric processing unit and local memory which is completely independent of the others. Communication between nodes is achieved entirely by queued message passing.

Messages can be sent from any node to any other node. The message system automatically routes messages through the system, forwarding them if necessary. One can think of "virtual" communications channels that connect all the processors. However, in the interest of keeping communication overhead costs down, one should always select a distribution of processes on the nodes which minimize the number of messages and the Hamming distances they must travel.

The system we have at Cornell is an Intel iPSC/D4MX (4-dimensional cube) with extended memory on the nodes.

3. The Fast Fourier Transform Algorithm

The discrete Fourier Transform of a complex vector $(x_0, x_1, \ldots, x_{N-1})^T$ is defined by

$$y(k) = \sum_{j=0}^{N-1} \exp(2\pi ijk/N)x(j), \quad k = 0, 1, \ldots, N-1 \tag{3.1}$$

This expression describes the computation of N equations. Defining

$$\omega = \exp(2\pi i/N) \tag{3.2}$$

we can compose the N-by-N matrix W by setting the j, k element of W^T to be $\omega^{jk \bmod N}$. The discrete Fourier transform can then be written in the form of matrix-vector multiplication.

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix} = \begin{pmatrix} \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \cdots & \omega^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{N-1} & \cdots & \omega^1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{pmatrix} \tag{3.3}$$

or more compactly as

$$\mathbf{y} = \mathbf{W^T x} \tag{3.4}$$

Computing the discrete Fourier transform by this method takes $O(N^2)$ operations.

The Fast Fourier Transform developed by Cooley and Tukey in the mid-60s is a method of computing the discrete Fourier transform which reduces the number of operations required to $O(N \log_2 N)$. This improvement comes from a reordering of the data so as to permit the clever pairing of computations to exploit the fact that

$$\omega^p = -\omega^{p+\frac{N}{2}} \tag{3.5}$$

The heart of the FFT algorithm is the judicious pairing of computations brought about by the rearrangement of the vector $\mathbf{x}$ so that only one complex multiplication is required for the following pair of equations:

$$\begin{aligned} x_l(j) &= x_{l-1}(j) + \omega^p x_{l-1}\left(j + \frac{N}{2^l}\right) \\ x_l\left(j + \frac{N}{2^l}\right) &= x_{l-1}(j) - \omega^p x_{l-1}\left(j + \frac{N}{2^l}\right) \end{aligned} \tag{3.7}$$

where l is the step ($l = 1, \ldots, \log_2 N$) and p is the power of ω which is to be determined according to the algorithm. At each step there are $N/2$ independent pairs of equations with each equation pair accessing components of $\mathbf{x}$ which lie exactly a power of 2 apart.

The reordering of the data vector $\mathbf{x}$ in the computations is analogous to the permutation of W by a permutation which sends each index i to i', an integer obtained by making a mirror image of the binary representation of i. This is called bit-reversing. For example, if $i = 13 \quad (01101)$ then $i' = 22 \quad (10110)$. Therefore the output of the FFT algorithm is a vector $\overline{x(j)} = x(j')$, $j = 0, 1, \ldots, N-1$. Another variation is to input $\mathbf{x}$ in bit-reversed order. As it happens, when the input data is in scrambled order, the output data comes out in natural order. This can best be visualized by the signal flow graphs in Figure (1). We shall implement both methods on the hypercube and examine the programming details where they differ.

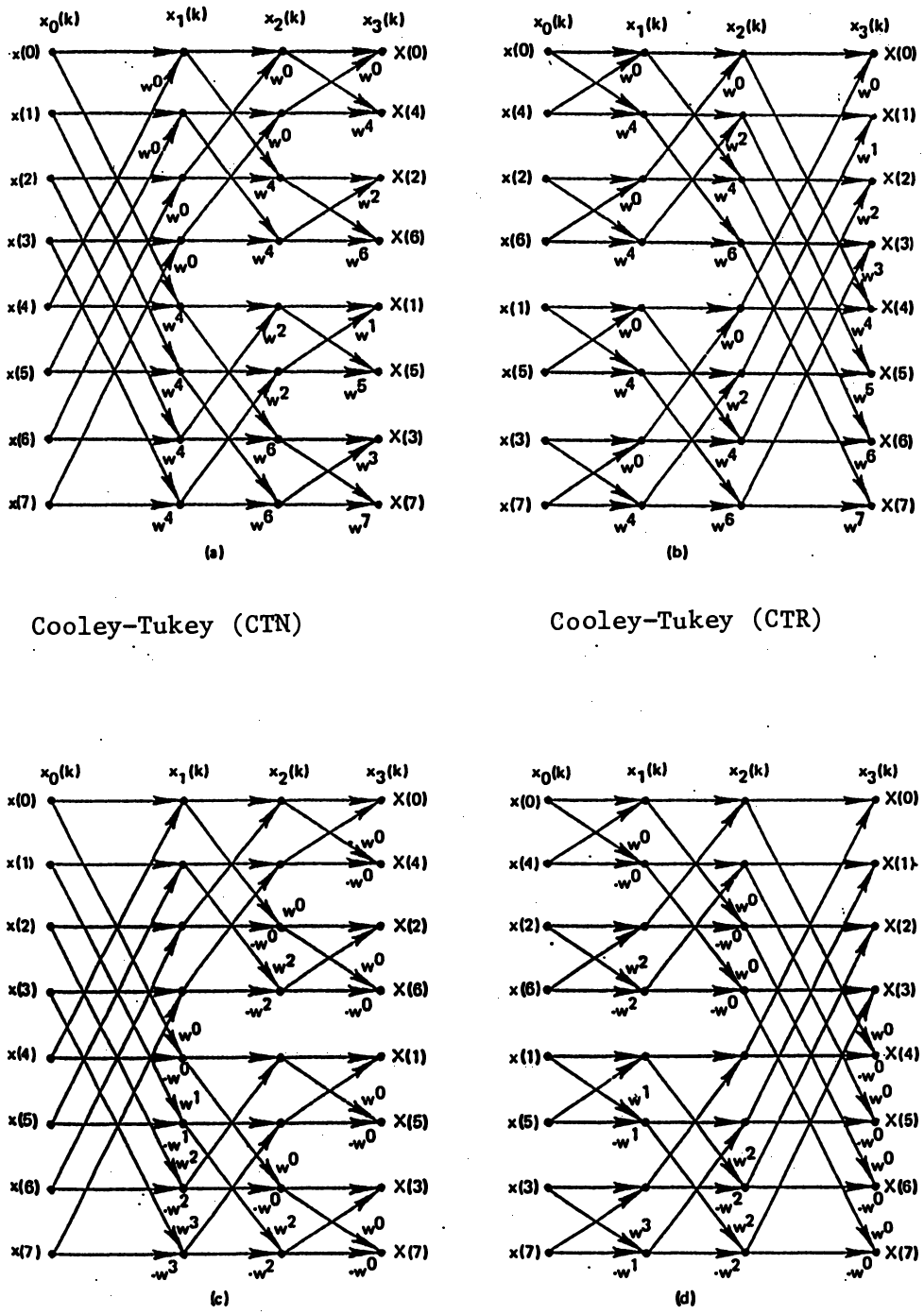
There is also another version of the radix-two FFT which is called Gentleman-Sande or decimation in frequency. The basic idea is the same except that the pair of computations analogous to (3.7) is:

$$\begin{aligned} x_l(j) &= x_{l-1}(j) + x_{l-1}\left(j + \frac{N}{2^l}\right) \\ x_l\left(j + \frac{N}{2^l}\right) &= \omega^p \left[x_{l-1}(j) - x_{l-1}\left(j + \frac{N}{2^l}\right)\right] \end{aligned} \tag{3.8}$$

The variations concern data input order and the order in which p, the power of ω is determined.

- Cooley-Tukey; data input natural order; p determined in bit-reversed order.
- Cooley-Tukey; data input in bit-reversed order; p determined in the natural order.
- Gentleman-Sande; data input natural order; p determined in natural order.
- Gentleman-Sande; data input bit-reversed order; p determined in bit-reversed order.

See Brigham (1974) for an comprehensive introduction to the FFT.



Cooley-Tukey (CTN) Cooley-Tukey (CTR)

Gentleman-Sande (GSN) Gentleman-Sande (GSR)

Figure 11-4. FFT canonic signal flow graphs.

Figure 1. (from E.O. Brigham, The Fast Fourier Transform, Prentice-Hall, 1974.

4. One-Dimensional Complex FFT on the Hypercube

Suppose we wish to find the Discrete Fourier transform of a vector **x** of length N (N a power of two). The most obvious way to distribute the vector **x** into a hypercube of dimension k, (where $P = 2^k$, and $k \leq \log_2 N$) is by partitioning **x** into subvector x_i each of length $n = N/P$,

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{pmatrix} \tag{4.1}$$

and placing x_i in processor i. Here it would be helpful to imagine the hypercube as a column of N nodes. This is the manner in which the problem for input data in natural order is set up.

When the input data is in bit-reversed order, we simply set

$$x' = \begin{pmatrix} x_{0'} \\ x_{1'} \\ \vdots \\ x'_{N-1} \end{pmatrix} \tag{4.2}$$

and place each $x_{i'}$ of length N/P into processor i.

One should note that if the input is in the natural order, the output is scrambled in the bit-reversed order and vice versa. In a single processor, the unscrambling operations are trivial, however in the many processor case such as ours with only local memory, it requires extra communication steps. We shall not concern ourselves with this unscrambling in this paper. It has been dealt with at length by Swarztrauber (1986).

Chamberlain (1986) describes an implementation where the data is input in the natural order and distributed through the processors in the natural order. By using the Gentleman-Sande butterfly the powers of ω can be calculated in the natural order. Intermingling is first, i.e. for a transform of 2^M points on a k-dimensional hypercube, the first k steps involve across processor butterflies and the last $M - k$ steps are independent and occur totally within each processor. (Here we will define *intermingling* as synonymous to doing an across processor butterfly where communication is necessary.) However there is an inefficiency in load balance during the intermingling steps. The processor on top has to work out only the sum

of two complex vectors whereas the lower processor has to calculate the power of ω for each vector component and then multiply it into the difference of the two vectors. This means that the lower processor does two complex multiplies per data point more than the upper one. This creates the imbalance in the amount of work that processors have to do in the across processor steps. We shall see that this imbalance is markedly more in the Gentleman-Sande algorithm than the Cooley-Tukey one because the node on top does not have to wait for the lower node to compute a multiplier. Instead, the exchange is before the butterfly operation, whereas in the Cooley-Tukey algorithm the exchange is done during the butterfly operation.

Walton (1986) describes another implementation based on the FFT algorithm found in Press *et al.* (1986) where the data is first scrambled into bit-reversed order and the Cooley-Tukey algorithm is used. Notice that the powers of ω also occur in natural order. The difference in this method is that the intermingling steps occur during the last k steps of the algorithm. In each intermingling step each processor exchanges $N/2P$ data points with one of its neighbors twice; once to obtain the data for the next step of the transform and a second time to send the partially transformed data "home". Thus he incurs the cost of two exchanges per across-processor step. This, as we shall see is unnecessary as we will not send the data "home" but rather overwrite the array space that it occupies with the update so that our final answer would be in a natural order in two columns instead of in the one column that we are accustomed to. Therefore if we were willing to accept a slightly different layout of our original data, we can circumvent the need for the extra exchange in each step.

4.1 Implementation Results for the One-Dimensional FFT

We have implemented the one dimensional FFT with the data laid out in one column and the hypercube in the natural order for four variations of the FFT algorithm. The results support the theory of load imbalance among the processors during the between processor butterfly steps. Since our hypercube has a maximum dimension of 4, there are at most 4 between processor steps where intermingling occurs.

Abbreviations

`gsn` Gentleman-Sande with data input in natural order; intermingling first.

`gsr` Gentleman-Sande with data input in bit-reversed order; intermingling last.

`ctn` Cooley-Tukey with data input in natural order; intermingling first.

`ctr` Cooley-Tukey with data input in bit-reversed order; intermingling last.

$N = 4096$

dim	gsn	gsr	ctn	ctr
1	4490–5175	5310–5555	6040–6045	4920–4930
2	2210–2910	2630–2880	3105–3110	2930–2940
3	1100–1635	1310–1505	1600–1610	1685–1695
4	550–915	655–795	830–875	950–960

(The times are given ranging from the fastest processor to the slowest processor and are given in milliseconds.)

As noted above, the Gentleman-Sande algorithm exhibits more load imbalance than the Cooley-Tukey algorithm because of the different butterfly computations. For the Gentleman Sande butterfly, the top processor does not have to wait for the lower processor to compute its multipliers before it can go ahead and do its update. Keep in mind that the amount of computations are the same for the Gentleman-Sande and the Cooley-Tukey algorithms. The major difference lies in the fact that the Cooley-Tukey algorithms require the top processor to block and wait for the lower one to compute a scaled version of its vector before the exchange even takes place. Pseudo-code for the two butterflies are given below.

```
Gentleman-Sande Butterfly (gsr)
  begin
/* exchange vectors */
  sendw z( ) to neighbor
  recvw cbuf( ) from neighbor
```

```
  if (top) then
    z( ) ← z( ) + cbuf( )
  else
/* bottom processor has to figure out multipliers */
/* theta is the argument angle, p is w for this step */
    wr = cos(p*theta); wi = sin(p*theta); w = cmplx(wr,wi)
    z( ) ← cbuf( ) - z( )
    z( ) ← w*z( )
  end if
  end
```

Cooley-Tukey Butterfly (ctn)

```
  begin
  if (.not.top) then
/* bottom processor has to figure out multipliers */
/* theta is the argument angle, p is w for this step */
    wr = cos(p*theta); wi = sin(p*theta); w = cmplx(wr,wi)
/* scale z( ) */
    z() ← w*z( )
  end if
/* exchange vectors */
  sendw z to neighbor
  recvw cbuf from neighbor
/* do Cooley-Tukey butterfly */
  if (top) then
    z( ) ← z( ) + cbuf( )
  else
    z( ) ← z( ) - cbuf( )
  end if
  end
```

Another detail is that in the algorithms where intermingling takes place first (`gsr/ctn`), there is more of an imbalance because the top half at each step gets smaller and smaller. For example, processor 0 wins at every step, but its partners after the first step are also winners in their butterfly. In a three dimensional cube, processor 0 wins with processor 4 in step 1, wins with processor 2 in step 2 (but processor 2 is already a winner in step 1), and wins with processor 1 in step 3 (processor 1 having won in both steps 1 and 2). Thus this load imbalance builds up with growing dimension. After the intermingling steps, the fastest processor is "set-free" from waiting for other processors and proceeds to do its own computations. For the algorithms where intermingling is last (`gsn/ctr`) the imbalance is there, but it is not as noticeable since the time differences only appear near the end of the computations.

There is another crucial difference between the algorithms `gsr/ctn` and `gsn/ctr` in the computation of multipliers In `gsr/ctn`, a single multiplier is used for the whole task: `z( )` ← `w*(cbuf( ) - z( ))` or `z( )` ← `w*z( )` where `z( )` and `cbuf( )` are vectors of length `N/P`. This is essentially a `LINPACK SSCAL` operation and easily vectorized. However in `gsn/ctr`, each component of `z(j)` needs a different multiplier. Thus we have to determine the power of ω to use *within* the inner loop. The updating task becomes

$$z_j \leftarrow w(p_j) * (cbuf_j - z_j), \quad j = 1, \ldots, N/P$$

or

$$z_j \leftarrow w(p_j) * z_j, \quad j = 1, \ldots, N/P.$$

However the subsequent multipliers are calculated in the natural order and can be done by a trigonometric recurrence involving four real multiplies or one complex multiply. In the `gsr/ctn` case, the power `p` is calculated by a bit-reversal operation and the multipliers calculated by one call to sine and one call to cosine. Therefore the `gsr/ctn` case saves `N/P * 4` multiplications per step.

5. The Binary Reflected Gray Code

While the topology of the hypercube is, in a sense, optimal for the radix-two FFT algorithm, most such computations do not take place by themselves. Usually the FFT is only a portion of a series of computations. These other computations may require an entirely different topology for the minimization of communication steps. For example if the FFT is involved in the solution of partial differential equations, it is natural to use a finite difference discretization where a Nearest-Neighbor mesh type topology would be best suited. It would be inefficient to arrange the data in a nearest neighbor configuration for the finite difference steps and then rearranging it for the FFT steps. Therefore Chamberlain (1986) and Chan (1986) have proposed a compromise by mapping the data into the nodes which are ordered by a Binary Reflected Gray Code. Several properties of this special gray code make it useful for the purposes at hand.

The Binary Reflected Gray Code (BRGC) can be defined recursively as follows:

$$G_k = [0, 1]$$
$$G_{k+1} = [0G_k, 1G_k^R]$$

where $0G_k$ denotes the sequence obtained by prefixing each member of G_k by 0 and G_k^R is the sequence obtained by reversing the order of G_k.

This particular Gray code has several important properties which we shall summarize below.

Property 1. The Hamming distance between any two consecutive (cyclically) members of the sequence is equal to one.

Property 2. The Hamming distance between elements $G_k(i)$ and $G_k(2^j - 1 - i)$ $(1 \leq j \leq k)$, where all the subscripts are taken modulo $2^{(j+1)}$, is equal to one.

Proof. By induction. $G_1 = [0,1]$ where $G_1(0) = 0$ and $G_1(2^1 - 1 - 0) = 1$ so it is trivially true for $k = 1$. Suppose that the sequence G_k also has this property. Then by induction the Hamming distance between any pair $G_{k+1}(i)$ and $G_{k+1}(2^j - 1 - i)$ $(1 \leq j \leq k)$ is equal to one because both of them are in the same half of the sequence since the indices are taken modulo $2^{(j+1)}$. For $j = k + 1$ we have $G_{k+1}(i)$ and $G_{k+1}(2^{k+1} - 1 - i)$ which differ at exactly one bit, the $(k+1)$st bit by construction.

$$
\begin{aligned}
G_{k+1}(i) &= 0G_k(i) \\
G_{k+1}(2^{k+1} - 1 - i) &= 1G_k(i) \quad \text{for } i = 0, \ldots, 2^k - 1 \\
G_{k+1}(i) &= 1G_k(2^{k+1} - 1 - i) \\
G_{k+1}(2^{k+1} - 1 - i) &= 0G_k(2^{k+1} - 1 - i) \quad \text{for } i = 2^k, \ldots, 2^{k+1} - 1
\end{aligned}
$$

Property 3. The Hamming distance between elements $G_k(i)$ and $G_k(i+2^j)$ $(j > 0)$ is precisely 2. The indices are taken modulo 2^k.

Proof. [See Johnsson, 1985]

Property 1 shows that the BRGC is indeed a Gray code as claimed. Each element differs from its neighbors at exactly one bit. Property 2 is the reflecting property of the BRGC where elements which differ only at the jth bit are reflections of each other around the jth dimension. Property 3 is a global connectivity property since elements which differ by strides 2^j $(j > 0)$, no matter how large, remain at a constant Hamming distance of 2. This distance also does not increase with the dimension of the Gray code. We shall term these three properties the Nearest Neighbor Property, the Reflection Property, and the Global Connectivity Property in what follows.

Let B_k be the binary encoding of the numbers $0, 1, \ldots, 2^k - 1$. We would like to compare the three properties of the BRGC with that of the binary encoding. First

we see an easy way to convert a BRGC encoding into a binary encoding where a routing step is defined as one bit-flip.

Conversion. A Gray code encoding G_k can be rearranged to a binary encoding B_k in $k-1$ routing steps.

Proof. By induction. $G_1 = [0,1] = B_1$ so it is trivially true for $k = 1$. Assume true for $G_k \rightarrow B_k$.

$$G_{k+1} = [0G_k(0), 0G_k(1), \ldots, 0G_k(2^k - 2), 0G_k(2^k - 1),$$
$$1Gk_k(2^k - 1), 1G_k(2^k - 2), \ldots, 1G_k(1), 1G_k(0)]$$

The highest order bit in G_{k+1} and B_{k+1} already coincide. And the first half of G_{k+1} is just $0G_k$ which can be converted to B_k in $k-1$ steps, by the induction hypothesis. The trick next is to reverse the order of the elements in the second half. This amounts to a direct exchange of elements in dimension $k-1$, i.e. the second highest order bit. This takes one routing step by Property 2. The second half becomes $1G_k$ and can be converted to $1B_k$ in $k-1$ routing steps. Thus the total number of routing steps to convert G_{k+1} to B_{k+1} is k.

Claim 1. The binary encoding B_k does not possess the nearest neighbor property. This is easy to see, simply consider the element $01\ldots1$ and its successor $10\ldots0$ differ at all of their bits so are at a Hamming distance of k for a k-bit binary number.

Claim 2. The binary encoding B_k has Hamming distance j for elements who are reflections around the jth dimension. An example is the element $0\ldots0$ and $1\ldots1$ who are reflections of each other around the kth dimension.

Claim 3. The binary encoding B_k possesses the property such that elements that differ by 2^j when i and $i + 2^j$ are in the same $(j+1)$-dimensional sub-cube have Hamming distance exactly one. This can be seen from the fact that flipping the jth bit results in a binary number which differs from the original one by a stride which is a power of 2.

Therefore one can conclude that while the binary encoding is superior in terms of the particular powers-of-two connections needed for the FFT butterfly, it falls short with regard to nearest neighbor connections and reflecting connections, with a maximum Hamming distance which increases with the dimension of the code. The BRGC, however incorporates the best compromise between these three properties.

6. The Two-Track FFT Implementation

Suppose we would like to find the FFT of a vector $\mathbf{x}$ of length N where N is a power of 2. The two-track layout breaks this vector into two pieces, the first half which we shall denote $\mathbf{x^{(1)}}$ and the second $\mathbf{x^{(2)}}$. Suppose we have P processors. Then we partition $\mathbf{x^{(1)}}$ and $\mathbf{x^{(2)}}$ into subvectors $x_i^{(1)}$ and $x_i^{(2)}$ respectively. These subvectors are of length $N/2P$ and are placed in the processors by mapping x_i in processor i.

Here we have assumed a Binary Reflected Gray Code mapping of the processors. If this is the case, then we see that $x_{N/2-1}$ and $x_{N/2}$ are still in neighboring processors, whereas if the processors were mapped in the natural order, these two consecutive elements in the original vector would be separated by a Hamming distance equal to the dimension of the hypercube. We define the left track to consist of the subvectors of $\mathbf{x^{(1)}}$ and the right track the subvectors of $\mathbf{x^{(2)}}$.

We shall describe our method in the terms of a Gentleman Sande butterfly, but the results hold equally with the Cooley-Tukey version as far as load balancing and data transfer is concerned.

The Gentleman Sande butterfly at step L combines elements x_j and $x_{j+N/2^L}$. Notice that for the first step $L = 1$ these elements are already in the same processor. Furthermore, each processor has the same amount of complex multiplications and additions to do since each processor possesses a piece of the first half of $\mathbf{x}$ and the second half of $\mathbf{x}$.

At each step which involves across processor butterflies, each processor exchanges its portion of $\mathbf{x^{(2)}}$ for the portion of $\mathbf{x^{(1)}}$ that its exchange partner for that step has. Possession of that portion of the array now passes to said processor, meaning that there is no need to do a second exchange after the butterfly to send the results "home". The important thing to note is that in this scheme, the complete butterfly for each position takes place totally *within* an individual processor, rather than in the previous section where the top half of the butterfly computation took place in one processor and the bottom half in another. Therefore not only do we eliminate an extra exchange step, we also eliminate the need for buffering and have achieved load balancing.

During the independent steps which take place inside each processor, processor i mimics the exchange by swapping portions of $x_i^{(2)}$ and $x_i^{(1)}$ so that the final answer will come out bit-reversed with the ordering one uses to read a book. This swapping

will be referred to as *intra-processor swapping.*

Let's illustrate with a simple example. $(N = 16), (P = 4)$, with elements for the next exchange at each step highlighted in bold.

	Step 1		Step 2		Step 3		Step 4	
p(00)	0	**8**	0	**4**	0	**2**	0	1
	1	**9**	1	**5**	**0**	3	2	3
p(01)	2	**10**	**2**	6	4	**6**	4	5
	3	**11**	**3**	7	**5**	7	6	7
p(11)	**4**	12	8	**12**	8	**10**	8	9
	5	13	9	**13**	**9**	11	10	11
p(10)	**6**	14	**10**	14	12	**14**	12	13
	7	15	**11**	15	**13**	15	14	15

An inverse transform can now be done analogously where the complete butterfly computation takes place within the same processor and sub-vectors are swapped in the reverse order.

	Step 1		Step 2		Step 3		Step 4	
p(00)	0	**1**	0	**2**	0	**4**	0	8
	2	3	1	**3**	1	**5**	1	9
p(01)	4	**5**	**4**	6	2	**6**	2	10
	6	7	**5**	7	3	**7**	3	11
p(11)	8	**9**	8	**10**	**8**	12	4	12
	10	11	9	**11**	**9**	13	5	13
p(10)	12	**13**	**12**	14	**10**	14	6	14
	14	15	**13**	15	**11**	15	7	15

The timing results obtained next were done with the processors mapped in natural order so that a fair comparison can be done to that of the implementations in Section 4.

	$N = 4096$			
dim	gsn	speedup	ctn	speedup
0	8845	1.0	9125	1.00
1	4700–4730	1.87	4825–4825	1.89
2	2500–2525	3.50	2555–2555	3.57
3	1335–1385	6.39	1355–1360	6.71
4	720–725	12.20	725–730	12.50

As one can see, these times are balanced and moreover faster than the slowest processor in Section 4 in each case. This is because a lot of the waiting inefficiencies have been eliminated. Since this scheme only makes sense for implementation on the hypercube with the BRGC mapping, we include implementation results below.

	$N = 4096$			
dim	gsn	speedup	ctr	speedup
0	8845	1.0	9135	1.00
1	4700–4730	1.87	4830–4830	1.89
2	2650–2720	3.25	2610-2665	3.43
3	1445-1490	5.37	1435–1490	6.13
4	840–900	9.83	785–910	10.04

Hence even taking into account the inefficiencies of a Hamming distance of two for some of the exchange steps, the two-track method still beats out the usual one column method in Section 4.

To summarize:

- Load balancing achieved by placing sub-vectors so that complete butterflies are done inside a processor.
- Only $N/2P$ elements are exchanged at each step rather than N/P elements.
- Truly in-place computation since no extra buffering is needed.
- Only one exchange per across processor step since update data is not sent "home".
- Timing results faster than for one column method, since each processor only has to calculate $N/2P$ multipliers rather than N/P multipliers (for the two algorithms with natural ordering of multipliers).

The pseudo-code for the Gentleman-Sande algorithm is given below. The Cooley-Tukey method differs only in the butterfly operation, but is otherwise similar. Details for multiplier generation and ordering of between processor steps will differ, of course.

```
Node process (gsn)
  begin
/* the first dim steps involve other processors */
  for L = 1, dim
    begin
/* Gentleman Sande Butterfly */
    theta = (2**L)*π/n
    ω0 = exp(i * theta)
    p = mod(mynode/(N/P/2),N/2**L)
    wr = cos(p*theta); wi = sin(p*theta); w = cmplx(wr,wi)
    for  j = 1, N/P
    begin
      temp ← z1(j) - z2(j)
      z1(j) ← z1(j) + z2(j)
      z2(j) ← w*temp
      w ← ω0*w
    end
/* determine which neighbor to exchange with */
    position = dim - L + 1
    neighbor = flip(mynode,position)
/* exchange vectors */
    sendw z2( ) to neighbor
    recvw z1( ) from neighbor
    end
/* do an fft of length N/P within processor */
  end
```

7. The U-Turn FFT Implementation.

Another method to achieve a load balanced implementation is to map the data out in a U-Turn. We break our vector $\mathbf{x}$ into two parts $\mathbf{x^{(1)}}$ and $\mathbf{x^{(2)}}$ as before. We partition $\mathbf{x^{(1)}}$ as before and map it into column one, but for column two, we map $\mathbf{x^{(2)}}$ out in reversed order. Now at every step, after the first one, each processor would be the top processor for the subvector in the first column and the lower processor for the subvector in the second column. Thus each processor will have to calculate $N/2P$ multipliers and works on half a butterfly operation for the first column and half a butterfly operation for the second column. This method does require an extra buffer array for each column so its storage requirements are the same as that for the one column method. Indeed its only use is for the computation of the complex FFT in the intermediate steps for some of the symmetric transforms. For example, suppose we would like to find the sine transform of a real sequence $Y(j)$, $j = 1, \ldots, N-1$. One way to do this is to extend $Y(j)$ into a $2N$-point odd sequence and then split that sequence into an N-point odd sequence and an N-point quarter-odd sequence. Then one finds the complex FFT of a N-point complex sequence $X(j)$ where $\mathrm{Re}X(j)$ consists of the jth point in the odd sequence and $\mathrm{Im}X(j)$, the jth point in the quarter-odd sequence. After the complex FFT, the sine transform can be recovered by some post-processing. [See Chu (1987) for details].

In any case, one would not really want to extend $Y(j)$ to $2N$ points, since one knows what those values would be. Thus suppose $Y(j)$ is distributed in the hypercube in the natural way and pre-processed according to the following formula.

$$\begin{aligned} X(0) &= iY(1) \\ X(j) &= Y(2j) + iY(2j+1) \quad j = 1, \ldots, N/2-1 \\ X(N-j) &= -Y(2j) - iY(2j-1) \quad j = 1, \ldots, N/2-1 \\ X(N/2) &= -iY(N-1) \end{aligned}$$

If communication costs were to be minimized, it would seem natural to place $X(j)$ and $X(N-j-1)$ in the same processor as $Y(2j)$ and $Y(2j+1)$. Therefore, if one were to map out where all the $X(j)$'s end up, we would find them mapped in two columns in a U-Turn.

If the processors were mapped out in the natural order, the first step requires that each processor j exchange with processor $P-j-1$. Claim 2 in Section 5 shows

this to be inefficient. However, since the BRGC possesses the Reflection Property, the Step 1 exchanges can be done at a Hamming distance of one if we were to map our processors accordingly.

The results given below are for the algorithms with multipliers figured out in the natural order.

$N = 4096$

dim	gsn	speedup	ctr	speedup
0	8545	1.0	8430	1.00
1	4515–4560	1.87	4565–4570	1.84
2	2705–2755	3.10	2720–2730	3.09
3	1525–1640	5.21	1570–1610	5.24
4	955–1050	8.14	965–1070	7.88

These timings are roughly comparable with the maximum times in the one column method, but are better balanced. The speedups are not quite as good since we are implementing it on a BRGC mapped hypercube which does exchanges necessary for the FFT at Hamming distance two. But since this method is necessary for use in the implementation of symmetric FFTs on the hypercube, it is important to present it here.

```
Node process (gsn)
  begin
/* the first dim + 1 steps involve other processors */
  np2 = N/P/2
  L = 1
/* exchange with your reflection */
  sendw z1( ) to neighbor
  recvw z2b( ) from neighbor
  sendw z2( ) to neighbor
  recvw z1b( ) from neighbor
  theta = (2**L)π/N
  ω0 = exp(i * theta)
  p = mod(bindex(neighbor,dim)*(N/P/2),N/2**L)
/* Gentleman Sande Butterfly */
  for j = 1, np2
  begin
    z1(j) ← z1(j) + z1b(np2-j+1)
    z2(np2-j+1) = w*(z2b(j) - z2(np2-j+1))
    w = ω0*w
  end
  for L = 2, dim + 1
    begin
```

```
/* determine which neighbor to exchange with */
    position = dim - L + 2
    neighbor = flip(mbnode,position)
/* exchange vectors */
    sendw z1( ) to neighbor
    recvw z2b( ) from neighbor
    sendw z2( ) to neighbor
    recvw z1b( ) from neighbor
/* figure out multiplier */
    theta = (2**L)*π/N
    ω0 = exp(i * theta)
    if (top) then
      bitpos = dim - L + 1
      nnode = flip(myynode,bitpos)
      p = mod(bindex(nnode,dim)*(np2),N/2**L)
      wr = cos(p*theta); wi = sin(p*theta); w = cmplx(wr,wi)
/* Gentleman Sande Butterfly */
      for j = 1, np2
      begin
        z1(j) ← z1(j) + z1b(j)
        z2(np2-j+1) = w*(z2b(np2-j+1) - z2(np2-j+1))
        w = ω0*w
      end
    else
      p = mod(bindex(neighbor,dim)*np2,N/2**L)
      wr = cos(p*theta); wi = sin(p*theta); w = cmplx(wr,wi)
      for j = 1, np2
      begin
        z1(j) ← z1(j) + z1b(j)
        z2(j) = w*(z2b(j) - z2(j))
        w = ω0*w
      end
    end if
    end
/* do an fft of length N/P within processor */
  end
```

The main drawback of this method is the need for buffering of the data to be exchanged. Therefore, in the interest of saving this storage space, we may wish to implement the U-Turn method in terms of the Two-Track method. This is easily done by performing two extra reorderings of the second column, once before the FFT and once after. Basically, the column is turned upside down and the two-track method is used. Upon completion, it is once again flipped so that the data is back in a U-Turn in bit-reversed order. The two-track method must now be modified so that the data is "sent home", which gives us Walton's (1986) method. We shall see that these extra exchanges cost only a small amount, especially if we

modify Walton's method so that the superfluous intra-processor swaps between the first and second column are not done in the $\log_2 n - d$ steps which occur totally within each processor. If we compare the timings for $N = 4096$, we see that while our new modification is slightly slower than the U-Turn method, due to the extra communications, it is faster than the method which uses two columns and continues swapping the data between the columns in the intra-processor stages.

	$N = 4096$			
dim	U-Turn	Walton*	no swaps	speedup
0	8545	10440	8910	1.00
1	4515–4560	5600–5605	4845–4850	1.84
2	2705–2755	3175–3190	2800–2815	3.17
3	1525–1640	1820–1860	1625–1660	5.37
4	955–1050	1165–1205	1060–1110	8.03

* my implementation of Walton's method.

(where the speedup shown is for the "no-swaps" method).

Since we should be more concerned with storage space, especially when doing two-dimensional transforms, we recommend the modification described above for the computation of the FFT of data which is mapped in a U-Turn. We shall do this for our implementation of the two dimensional sine transform.

8. Conclusion.

We have presented two load balanced algorithms for the implementation of the complex FFT on the hypercube. Both come from the rearrangement of the input data and show that often in the field of parallel computing, the way one maps the data and organizes the computation can make a difference for algorithms which have essentially the same operations count. If one were willing to accept the mapping of data in two tracks for any particular application, then one could achieve a load balanced algorithm which is faster than the traditional one and also uses "in-place" computations and minimizes storage. We have also adapted the ability to do "in-place" computations to the problem of finding the FFT mapped in a U-Turn, a configuration which occurs naturally during the pre-processing of symmetric FFT algorithms.

9. Appendix

Functions used in the Pseudo-Code:

`integer function flip(value,position)`
purpose: toggle the indicated bit `position` of the binary representation of `value`.

example: `flip(19,4) = 27`;
since `19 = 10011` and `27 = 11011`.

`integer function ibitr(j,ln)`
purpose: bit reverse the binary representation of `j` as a binary number of `ln` bits.

example: `ibitr(5,6) = 40`;
since `5 = 000101` and `40 = 101000`.

`integer function bindex(num,dim)`
purpose: computes the binary index of a number which is part of a binary reflected gray code of dimension dim.

10. Acknowledgements.

I would like to thank Professor Charles Van Loan for his helpful comments and conversation during the preparation of this material, and also Ashley Doerschug of the Cornell Theory Center for her help as system administrator of the Intel iPSC hypercube.

11. Bibliography

E. Oran Brigham. 1974. *The Fast Fourier Transform,* Prentice-Hall, Inc. Englewood Cliffs, New Jersey.

Chamberlain, R. M. 1986. "Gray Codes, Fast Fourier Transforms and Hypercubes," Tech. Report 864502-1, Chr. Michelsen Institute. Bergen, Norway.

Chan, Tony F. 1986. "On Gray Code Mappings for Mesh-FFTs on Binary N-Cubes," Tech. Report 86.17, RIACS, NASA Ames Research Center.

Chu, Clare. 1987. "A Sine Transform Algorithm for the Hypercube," Dept. of Computer Science Tech. Report. (TR 87-825) Cornell University, Ithaca, New York.

Johnsson, S. Lennart. 1985. "Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures," Research Report YALEU/DCS/RR-361.

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. 1986. *Numerical Recipes,* Cambridge: Cambridge University Press.

Swarztrauber, Paul N. 1986. "Multiprocessor FFTs," preprint from National Center for Atmospheric Research. Boulder, Colorado. (to appear in *Parallel Computing*)

Walton, Stephen R. 1986. "Fast Fourier Transforms on the Hypercube," preprint from Ametek Computer Research Division. Arcadia, California.