Comparison of Two-Dimensional FFT Methods on the Hypercube: The Choice Between Strips and Patches†

Clare Y. Chu

87-850
July 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.

Comparison of Two-Dimensional FFT Methods on the Hypercube: The Choice Between Strips and Patches[1]

Clare Y. Chu
Center for Applied Mathematics
Cornell University, Ithaca, NY

Dept. of Computer Science Tech. Report

July 1987

[1]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.

Abstract

Complex two-dimensional FFT's up to size 256×256 points were implemented on the Intel iPSC/System 286 hypercube with emphasis on comparing the effects of data mapping, data transposition, and distributed FFT's. Comparison is based on partitioning the data into either strips or patches and three distinct methods are discussed. Of the two strips methods, the Transpose-Split implementation involves local independent FFT computations in both directions with an intervening transpose, while the Local-Distributed method performs local FFT's in one direction followed by distributed FFT's in the other. The patch or Block method partitions the matrix in submatrices and maps the (i, j)th block into node $i\hat{\ }j$ ($\hat{\ }$denoting concatenation of binary numbers). Both the row-wise FFT's and the column-wise FFT's require inter-node communication for the distributed butterfly computations. Timing results show that on the Intel iPSC/System 286, there is hardly a difference between the three methods. This is due primarily to the inadequacies of Intel communication handling and its inability to overlap communication and computation as well as a lack of vector boards. A model follows where several important factors such as vectorization and communication complexity are considered. While timing results show that the Transpose-Split method is in general slightly faster, our model shows that on a vectorized hypercube with ideal synchronization of communication and computation the Block method is asymptotically the fastest. The conclusion is that this problem is highly system dependent and therefore no sweeping recommendations can be made.

1 Introduction

John Gustafson recently pointed out [MSI Workshop (1987)] the pitfalls in implementing FFT computations on the hypercube by an algorithm that distributes the radix-2 FFT butterfly computations among the nodes in the natural fashion suggested by the FFT signal flow graph. Instead he advocates a method where all the FFT computations are done locally in each processor and communication is restricted to a transpose operation using all the links of the hypercube to simulate a full cross-bar interconnection. A two-dimensional FFT of a matrix of complex points is implemented by computing FFT's on the rows locally, transposing the whole array, followed by local FFT computations on the columns. Generically, methods that perform computations locally within the processors with an intervening transpose communication are designated Transpose-Split methods by McBryan and Van de Velde (1985). Gustafson has implemented the Transpose-Split FFT on a 1024-node NCUBE machine which has the pleasant property that each node can perform up to 9 simultaneous communications, thereby allowing the use of almost all the links of the hypercube during the transpose stage. He can reduce communication time by a factor of d, the hypercube dimension.

The Floating Point Systems T-Series is a hypercube configuration where computation and communication can proceed in parallel in each node. A two-dimensional complex FFT has been implemented on the T-Series [Miles *et al.*, 1987] by partitioning the data into block matrices and then assigning one block matrix per node. By considering the signal flow graph of the radix-2 FFT algorithm, we see that this implementation requires communication during both the vertical and the horizontal passes. At each step where the butterfly computation is split between two processors, each node exchanges with its neighbor exactly half of its data points. Each processor computes the butterfly updates for the points it possesses after which it contains updates for half of its own points and half of the points belonging to its communicating partner. Another exchange is then necessary to repatriate these updates. Therefore two exchanges are required for one butterfly step. This may seem inefficient unless the communications and computations are overlapped during the distributed butterfly calculations. This is indeed possible on the T-Series since each node possesses a transputer that allows a processor to send data to its neighbor in the next butterfly step even before it has totally completed the present step. Hence, two communication stages can be overlapped in one computational step. This method of implementation will be referred to as the Block method since the matrix is mapped by sub-blocks into the hypercube such that the (i, j)th block is mapped into the node whose label is the binary representation of i concatenated with the binary representation of j. [See Miles *et al.*, 1987]. The powerful vector boards on the T-Series allows this method to be used advantageously.

The Intel iPSC, unlike the NCUBE and T-Series machines, does not allow simultaneous communication. Therefore the full cross-bar interconnection scheme cannot be simulated efficiently. Instead with Intel iPSC/System 286 capabilities, a transpose takes $2d = 2\log_2 p$ steps to perform, as each node can only do one send followed by one receive in one direction at a time. Another drawback is that computation and communication cannot be overlapped and thus distributed FFT's will exhibit blocking during the trans-processor butterfly steps caused by one processor waiting for data from another. Distributed FFT's have been studied by Chamberlain [1986], Walton [1986] and Chu [1987] who have all noted the potential load imbalance among the processors. Therefore we propose a hybrid method to avoid the transpose without requiring communication in both directions. Instead, local FFT's are done in one direction, with distributed FFT's requiring trans-processor butterfly computations done in the second direction. This method requires a block row mapping of the matrix into the hypercube. We shall term this the Local-Distributed Method.

2 One and Two-Dimensional Discrete Fourier Transform

The Discrete Fourier Transform (DFT) of a vector x of length N is defined as

$$X \leftarrow F_N x$$

where F_N is the matrix consisting of powers of the Nth roots of unity $\omega = e^{-2\pi i/N}$.

$$(F_N)_{jk} = \omega^{jk}$$

Suppose $N = N_1N_2$. Another way to compute F_Nx is to view it as a DFT of size $N_1 \times N_2$ [Nussbaumer (1982)], that is, write x as an array $x(m_1, m_2)$ where $m = N_1m_2+m_1$, ($m_1 = 0, \ldots, N_1-1$ and $m_2 = 0, \ldots, N_2-1$). The inputs are in column-wise order. The resulting transform $X(k_1, k_2)$, $k = N_2k_1 + k_2$ ($k_1 = 0, \ldots, N_1 - 1$ and $k_2 = 0, \ldots, N_2 - 1$), has the output answers appearing in row-wise order.

In this fashion we can compute the N-point DFT of x by an N_1-point DFT of the rows, a multiplication by *twiddle factors*, and an N_2-point DFT of the columns. Letting T be the matrix of twiddle factors

$$(T)_{jk} = \omega^{jk}, \quad j = 0, \ldots, N_1 - 1; \quad k = 0, \ldots, N_2 - 1.$$

and $*$ denoting point-wise multiplication of two matrices, we have

$$X(k_1, k_2) = F_{N_1}[T * [x(m_1, m_2)F_{N_2}]].$$

The two-dimensional DFT of an N_1-by-N_2 matrix x is computed in a similar manner except that the twiddle factor multiplication is omitted. Thus

$$X(k_1, k_2) = F_{N_1}x(m_1, m_2)F_{N_2}.$$

3 Data Structures and Mapping

The Local-Distributed method as well as the Transpose-Split method are based on the natural mapping of block rows into the hypercube, where block row i is stored in processor i. The particular way we map these block rows into the hypercube is equivalent to McBryan and Van de Velde's *distributed column* mapping [1985]. Figure 1 shows the mapping of a matrix into 4 processors. Local FFT's are done along the rows, with the vectors oriented perpendicular to the direction of the FFT computations.

The Block method involves distributed FFT's along both rows and columns. This is done by viewing a d-dimensional hypercube as a "cross-product" of a k-dimensional by an l-dimensional hypercube where $d = k+l$.

To illustrate what we mean, Figure 2 gives a low dimension example. Suppose we have a 5-cube viewed as a "cross-product" of a 3-cube and a 2-cube. Each node has a binary label A with 5 bits that can be construed

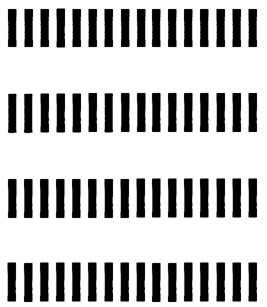

Figure 1: 16 columns mapped into 4 processors.

000^00	000^01	000^10	000^11
001^00	001^01	001^10	001^11
010^00	010^01	010^10	010^11
011^00	011^01	011^10	011^11
100^00	100^01	100^10	100^11
101^00	101^01	101^10	101^11
110^00	110^01	110^10	110^11
111^00	111^01	111^10	111^11

Figure 2: 5-cube = 3-cube $\otimes$ 2-cube.

as the concatenation of a binary label $b_1b_2b_3$ of 3 bits with a binary label c_1c_2 of 2 bits, i.e.

$$A = b_1b_2b_3c_1c_2$$

Now we place each node so that in the x-direction c_1c_2 go from 00 to 11; and in the y-direction $b_1b_2b_3$ go from 000 to 111.

This produces a two-dimensional array of nodes possessing "powers of two" adjacency for both rows *and* columns. Notice that we have exactly the adjacency structure necessary for a two-dimensional radix-2 FFT of a matrix mapped into the processors in a natural fashion.

This scheme extends to arbitrary dimensions. Suppose we have a d-dimensional hypercube where $d = k + l + m$. We can write the binary label

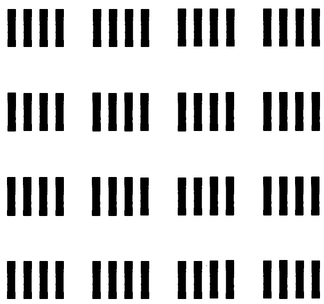

Figure 3: 16 columns mapped into 16 processors.

of each node

$$A = b_1 b_2 \dots b_k c_1 c_2 \dots c_l f_1 f_2 \dots f_m$$

and map our nodes in three dimensions. Denoting concatenation by exponentiation, we order the $f_1 f_2 \dots f_m$ from 0^m to 1^m along the x-direction, the $c_1 c_2 \dots c_l$ from 0^l to 1^l along the y-direction and the $b_1 b_2 \dots b_k$ from 0^k to 1^k along the z-direction. Therefore if $d = d_1 + d_2 + \dots + d_j$ ($j=$ dimension of data structure), a j-dimensional array can be mapped into the d-dimensional hypercube so that in each direction $k = 1, 2, \dots, j$, the interconnections are exactly those of an d_k-dimensional hypercube.

Our matrix is mapped by blocks into the nodes of the hypercube so that block A_{ij} goes into processor $i\hat{\ }j$ (where $\hat{\ }$denotes concatenation). Figure 3 shows the mapping of a matrix into a hypercube of 16 nodes.

The FFT computations must include distributed butterfly computations in both directions. This method can obviously be extended to higher dimensions.

4 Efficient Data Transposition on the Intel iPSC

An extensive study on matrix transposition on hypercube architectures was done by Ching-Tien Ho and S. Lennart Johnsson [1986]. The algorithm used in the implementations comes from a recursive block transpose algorithm described in Johnsson [1985a, 1985b] and McBryan and Van de Velde [1985]. Basically a matrix of dimension N mapped into P processors can be block transposed in time proportional to $N^2 \log_2 P/P$ floating point number transfers. The algorithm is described by the following recursion.

procedure *Transpose(A):*
$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$$
$A_{01} \leftrightarrow A_{10}$
$Transpose(A_{00}) \quad Transpose(A_{01})$
$Transpose(A_{10}) \quad Transpose(A_{11})$

Figure 4 describes the recursive block transpose for an 8-by-8 block matrix mapped into a hypercube of 8 nodes. Each node receives one block row of the matrix. And the arrows denote the scope of the blocks that are transposed at each step. Figure 5 concentrates on processor 2 and highlights in black the portions of the matrix that are to be exchanged at each step. Since the matrix is mapped in the distributed columns formation, each dark block consists of contiguous elements and can be sent as a single message. The initial step involves an exchange between processors whose node numbers differ at the highest order bit, or bit number $d-1$. The messages are of length $L/2$, where L is the length of the portion of our matrix resident in each processor. In our case, L is the number of data-points in a block row multiplied by the number of bytes per data-point. In the next step, each processor now has 2 messages of length $L/4$ and these are exchanged between processors whose node numbers differ at the 2nd-highest order bit, (bit number $d-2$. Counting the steps from $k=0$ to $k=d-1$, we have the property that at the kth step, there are 2^k messages of length $L/2^{k+1}$ exchanged between processors whose node numbers differ at the $d-k-1$th bit.

A simple procedure describes where each chunk goes at each step.

Procedure Recursive Block Transpose
/* mynode = $(b_{d-1} \dots b_0)_2$ is the node number */
/* L = length of block row resident in processor */
  for $k = 0 : d-1$

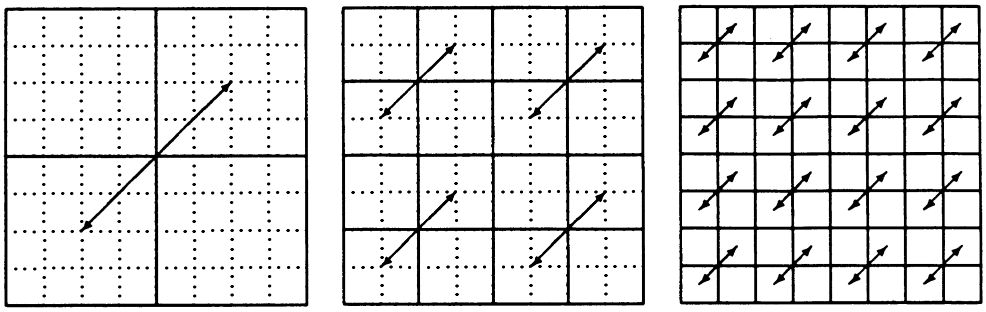

Figure 4: Block Recursive Transpose.

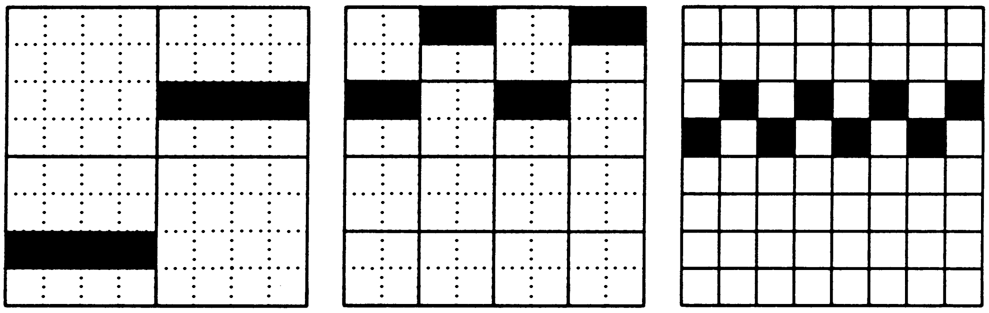

Figure 5: Blocks Exchanged by Processor # 2.

```
/* partition block row into 2^{k+1} submatrices of length L/2^{k+1} */
/* with the submatrices numbered from 0 to 2^{k+1} - 1 */
/* send 2^k submatrices to neighbor in (d - k)th direction */
/* receive 2^k submatrices from neighbor in (d - k)th direction */
    if mynode has a 0 at bit (d - k - 1) send odd submatrices
    if mynode has a 1 at bit (d - k - 1) send even submatrices
    if mynode has a 0 at bit (d - k - 1) receive even submatrices
    if mynode has a 1 at bit (d - k - 1) receive odd submatrices
  end
```

Notice that as the algorithm progresses, there are more and more messages of smaller and smaller sizes to exchange. This can cause a tremendous amount of overhead in terms of message start-ups. Since the start-up time is, in general, several orders of magnitude slower than the data transfer rate, we would prefer to buffer the smaller chunks into the largest possible array and transmit a fewer messages. Of course there are constraints to this approach also, taking into consideration the largest message size B_m and also the "copy" rate of the processors.

Ho and Johnsson [1986] present an optimized version of this algorithm for the Intel iPSC taking into account the communication startup time, the copying time for buffering, and other peculiarities of the Intel cube such as its allowing communication over only one port at time. They have discovered that the time it takes to copy $4K$ bytes is about 37 milliseconds and that the time for copying 256 bytes is approximated the same as one communication start-up. Hence their algorithm sends blocks of length at least 64 floating-point numbers without buffering. We have chosen to implement their algorithm for complex numbers. One should also note that if $P = 2^d$, then after d steps in the block recursive transpose algorithm, the remaining transpose operations are local to each processor.

5 Issues on Data Ordering, Vectorization, Mixed Radix FFT's, and Storage

One dimensional transforms of long vectors can also be implemented in procedures similar to the methods described here by utilizing the *twiddle* *factor* algorithm where the sequence is initially mapped in column order by distributed columns, i.e. block rows of columns. DFT's are taken along the rows, followed by a point-wise multiplication of twiddle factors. Then the whole array is transposed and the "column" DFT's are again done on the rows.

$$
\begin{aligned}
X &\leftarrow XF_{N_2} \\
X &\leftarrow T * X \\
X &\leftarrow X^T \\
X &\leftarrow XF_{N_1}
\end{aligned}
$$

Recall that $*$ denotes the point-wise multiplication of two matrices. Since the results $X(k_1, k_2)$ are in transposed order, the resulting array X, which actually contains X^T is in the correct order. No extra transpose is needed to bring the data back to its original order. A two-dimensional DFT proceeds in exactly the same manner except for the omission of the twiddle factor multiplication. However, the resulting array is in transposed order.

The usual radix-two in-place FFT algorithms require a data permutation either at the start of the procedure or at the end. This permutation is the well-known bit-reversing permutation. In the Transpose-Split method, the FFT's are performed locally so that this permutation is also done independently and locally. In fact Stockham's autosort FFT can be used, hence no distributed bit-reversing operation is necessary. The other two distributed methods, require up to an extra d communications [Swarztrauber (1986)], on a cube which can communicate simultaneously in all directions, to perform distributed bit-reversal. This is also discussed by Van Loan (1987). On the Intel cube, autosort FFT's incur too much communication overhead to be efficient. However if the data is to be transformed and then inverse transformed, there would be no need to unscramble in the transform domain since there are algorithms which take bit-reversed data on input and return the inverse transform in natural order.

If the data array is mapped within each processor so that its vector orientation is perpendicular to the direction of the FFT, one can most effectively vectorize the butterfly computations. What we means is that FFT's are done on matrix rows with the matrix mapped in column-major order. The vector length is equal to the number of multiple transforms and hence one would prefer to do the FFT of m transforms in parallel rather than do one transform after the other. See [Hockney and Jesshope (1981) and Swarztrauber (1986)]. The basic operation here is a vector SAXPY where

$$\mathbf{x} \leftarrow \alpha \mathbf{x} + \mathbf{y}$$

$\mathbf{v}$, $\mathbf{x}$, and $\mathbf{y}$ are vectors with α a scalar. Since the direction of the second FFT pass is perpendicular to that of the first pass, a transpose is needed between the two stages. For the Transpose-Split method this means that *after* the block transpose, one must complete the transpose by performing transposes on all the submatrices locally. For the two distributed methods, a local transpose of the array resident in each node is required to retain the correct orientation of the vectors. The Floating Point Systems implementation of the Block method consists of mapping the array X in sub-blocks into the nodes of the hypercube. Since the T-Series consists of vector boards, local transposes are done on each submatrix to keep the correct orientation for vectorizing.

Vector length is also an important issue when doing multiple transforms on vector processors. Suppose we have an N-by-N array and p processors. The length of the vectors in the direction perpendicular to the FFT computations should be as long as possible (up to the length of the vector register) so as to take full advantage of vector operations. The Transpose-Split method requires each processor to do N/p FFT's of length N simultaneously and thus has an effective vector length of N/p. The Block method has each processor responsible for $N/\sqrt{p}$ FFT's of that only $N/\sqrt{p}$ elements of each FFT are processor-local. The vector length here is $N/\sqrt{p}$ and is $\sqrt{p}$ times longer than the Transpose-Split method. The Local-Distributed method has the same characteristics of the Transpose-Split method during its local phase, vectors of length N/p; however in its distributed phase, the effective vector lengths are N. These observations are summarized in Table 1

On the Floating Point Systems T-Series hypercube, the vector registers are of length 128, and thus the Block method is favored here since its average vector length is the longest. For example, if $p = 64$, and $N = 1024$, the Block method would give vectors of length 128, whereas the Transpose-

	horizontal			vertical		
	TS	LD	B	TS	LD	B
$m^\dagger$	$\frac{N}{p}$	$\frac{N}{p}$	$\frac{N}{\sqrt{p}}$	$\frac{N}{p}$	N	$\frac{N}{\sqrt{p}}$
$n^\ddagger$	N	N	$\frac{N}{\sqrt{p}}$	N	$\frac{N}{p}$	$\frac{N}{\sqrt{p}}$

†# of FFT's = vector length

‡length of portion in each processor

Table 1: Multiple Transforms and Vector Length.

Split produces vectors of length 16. Since the vector boards are so fast (50–60 Mflops for the 2D-FFT) [Miles *et al.*, 1987], one would like to "fill up" the registers to minimize the loading and storage of data. In fact, this was a compelling reason for chosing the Block method for the FPS T-Series implementation.

A further consideration is whether the problem at hand requires vectors whose lengths are not a power of two. As the powers of two get large, they are spaced further and further apart. A partial differential equation problem may take place on a grid that is 100-by-100. Most of the distributed methods in the literature presume the use of a radix-two FFT on vector length $= 2^k$ (k some integer $\geq d$, the hypercube dimension). Therefore McBryan points out that that the Transpose-Split method permits one to use any FFT subroutine that accepts general n. Given that there is a good transpose algorithm, such a method is more portable and simple for the end-user to implement. However, the Local-Distributed Method which we have proposed here can be used to handle transforms of length $n = 2^k m$. This is because the twiddle factor algorithm can be adapted so that the local in-house FFT's are of length m and the distributed FFT's running the other direction are of length 2^k, where a radix-two algorithm is used. This method which we call the Mixed-Radix-2 will be discussed in a forthcoming report.

Finally, storage requirements differ for these methods. In general, the distributed methods require an extra N^2/p points of storage per processor if one wants to do only one exchange per distributed butterfly step. This is because all information must be sent to the other processor before the updating is done. If one is allowed to incur two exchanges per distributed butterfly step, as in the FPS implementation, there is no extra buffer space necessary for storage. The Transpose-Split method can be done totally in place, if unbuffered transposing is used. However, as Ho and Johnsson [1986] have noticed, this can cause the transpose time to grow exponentially in the number of dimensions. Therefore, one extra buffer is needed to collect the noncontigous blocks and ship them all out at once. For the Intel iPSC, this buffer is not too large (only 64 floating point numbers).

6 Implementation and Timings

The Transpose-Split, Local-Distributed, and Block methods for the two dimensional FFT were implemented on the Intel iPSC/D4MX hypercube running XENIX R3.4, iPSC Release 3.1 with Exelan R3.3 networking software. The code was written using Ryan-McFarland FORTRAN. Vector boards were *not* available so that all computations within a node were done serially. The Transpose-Split method uses local FFT's in both the vertical and horizontal stages and a recursive block transpose routine. The transpose code used was a modified version of Ching-Tien Ho's with the only change being the removal of the aforementioned buffering. The method of implementation for distributed FFT's follows that of Chu (1987), where the sequences are mapped in natural order into the hypercube. The isomorphism of the radix-2 FFT signal flow graph to the hypercube interconnection scheme is used to determine the inter-node communication pattern. The FFT subroutine implemented was the Gentleman-Sande version from Chu (1987).

Computation time as well as communication time was measured for all three methods. A portion of the communication time is reflected in a processor blocking while awaiting data that it is to receive. Table 2 shows the computation time required by each method while Table 3 displays the communication time as well as the time a processor spends blocked. The total execution time is shown in Table 4. Times are given in milliseconds in a range spanning from the fastest processor to the slowest processor.

We also mention that our implementation of the Block method differs from Floating Point System's in that only one exchange is incurred during each distributed butterfly step. Since the Intel cannot overlap communication and computation, the added cost of performing two exchanges would degrade the performance of the distributed methods without giving any basis for comparison.

The results show a three-way tie in terms of total execution time. All the timings were roughly within 10% of each other. But if we look at the break-down of communication versus computational time, some interesting differences surface.

As expected, the computational times in the distributed methods show a small amount of imbalance, with greater gaps between the faster and slowest processor than shown by the Transpose-Split method. While the Local-Distributed method stays about even with the Transpose-Split method as far as computation is concerned, the Block method on the average takes a bit longer. Since the same FFT algorithm was used for all three methods, we conjecture that this might be due to the fact that processors in the Block method get interrupted during computation at two stages, during both the vertical and horizontal FFT's, whereas the Local-Distributed method only gets interrupted during its vertical FFT. The Transpose-Split method is only interrupted once, between the horizontal and vertical stages. Of course the effects of these interruptions can be minimized if the processors are able to simultaneously communicate and compute.

Next notice that the Transpose-Split results actually show an *increase* in time for the $N = 16 \times 16$ problem with increasing number of processors. This is due primarily to the added complexity of having to send smaller and smaller messages or extra buffering costs. Even though we have implemented the unbuffered transpose, the results from the use of buffered transposing still show this increase. See Section 7.

Finally notice that the blocking time is roughly equal among the three methods. This demonstrates that the load imbalance of the distributed methods is not really much of a problem as far as blocking between **sends** and **receives** are concerned.

In the next section we shall see that while the distributed methods require $O(\log_2 p)$ start-ups for communication, the transpose method could possibly require up to $O(p)$ startups if not done carefully. We also consider the effective vector length of the different methods and hypothesize what would happen if the node processors have vector boards.

size	Computation Time		
dim	Transpose-Split	Local-Distributed	Block
16×16			
1	145	140–155	–
2	80	70–85	80–100
3	50–55	40–65	50–70
4	30–40	15–40	25–40
32×32			
1	680	645–695	–
2	355	320–380	345–410
3	195–200	155–215	210–270
4	110–120	85–120	80–135
64×64			
1	3170	3055–3250	–
2	1620	1490–1690	1615–1835
3	845–850	735–885	995–1180
4	450-460	380–470	385–515
128×128			
1	14720–14730	14805–15010	–
2	7415–7420	6935–7705	7565–8405
3	3770–3780	3400–3975	4660–5315
4	1950–1955	1680–2075	1775–2225
256×256			
3	16995–17010	16150–17875	21515–23975
4	8620–8630	7650–9145	8205–9910

Table 2: Computational Time of 2D-FFT.

size	Communication Time/Blocked Time		
dim	Transpose-Split	Local-Distributed	Block
16 × 16			
1	5/5	5/0	–
2	10–45/5–35	5–10/0–5	5–10/0–5
3	75–115/0–85	60–65/0–60	5–10/0–5
4	110–155/75–130	65–120/0–110	55-60/5–50
32 × 32			
1	5–10/0	15/0	–
2	10–15/0–5	15–20/0–5	15–20/5
3	25–70/0–60	15–20/0–10	15/0–5
4	90–105/10–75	5–20/0–15	15–65/0–60
64 × 64			
1	30/5	55–60/10	–
2	30–35/5	60/10	50–55/10
3	30–35/5–20	45–85/5–55	40–45/0–15
4	35–90/5–55	35–40/0–15	30–40/5–10
128 × 128			
1	120–135/15–40	220/35	–
2	110/20	220/30–35	220–255/35–60
3	80–95/10–25	165–170/20–30	160–165/20–25
4	70–115/10–60	105–120/10–20	105–120/15–20
256 × 256			
3	335–460/55–205	650–710/90–150	660–695/95–130
4	380–400/45–220	445–510/65–170	435–495/60–135

Table 3: Communication Time Including Blocking Overhead.

size	Total Execution Time		
dim	Transpose-Split	Local-Distributed	Block
16×16			
1	150	145–160	–
2	90	80–95	90–110
3	75–120	55–70	65–85
4	140–145	105–150	90–110
32×32			
1	690	665–715	–
2	370	340–395	365–430
3	220–260	180–230	235–290
4	160–270	105–160	110–205
64×64			
1	3200–3205	3110–3305	–
2	1650	1550–1750	1675–1895
3	880–915	785–935	1045–1230
4	560–615	415–585	425–555
128×128			
1	14845–14860	15025–15230	–
2	7530	7165–7930	7820–8630
3	3865–3870	3575–4145	4835–5480
4	2030–2070	1800–2225	1895–2350
256×256			
3	17335–17345	16870–18605	22220–24660
4	8860–8900	8135–9615	8695–10370

Table 4: Total Execution Time.

7 Discussion and Model

Models of computational and communicational complexity are often useful in giving general guidelines to the benefits of various methods of implementation. Since FFT implementations are usually communication bound, we first consider the analysis of simply transferring data among the processors as specified by the recursive block transform procedure and the distributed methods. The vectorization of multiple transforms are dealt with next. Finally we give an estimate of the total time required.

Suppose we have an N-by-N array and $p = 2^d$ processors. Throughout this discussion we shall assume that p is an even power of two. Assuming that p divides N, each processor would have N^2/p points. The recursive block transpose algorithm requires d steps where $N^2/2p$ points are exchanged per step. Meanwhile both distributed FFT algorithms have d steps involving trans-processor butterflies. Each step requires the exchange of N^2/p points. One can see immediately that twice as much data points are exchanged at each step by the distributed methods. Notice however, that these points are all contiguous, so that there is no overhead of sending multiple messages or need to copy into a buffer. However, due to the algebraic structure of the butterflies, an extra buffer array of N^2/p points is needed for each processor since it cannot overwrite its array until after the butterfly computation. The extra buffer is not required in the Floating Point System implementation, however an extra exchange per trans-processor butterfly step is needed.

Let τ be the data startup time or communication latency time, B_m the maximum packet size that can be transferred at a time, and t_c the per element transfer time. Denote time by

$$T^{method}_{operation},$$

the time required for a certain operation by a certain method.

The total communication overhead is measured by

$$t_c \text{ (number of elements to be sent)} + \tau \text{ (number of start-ups)}$$

First let's look at unbuffered transpose communication. Ho and Johnsson

[1986] show that the complexity for unbuffered communication is

$$T_{comm}^{TS} = d\frac{N^2}{p}t_c + \left(p + \left\lceil \frac{N^2}{2B_m p} \right\rceil \min(d, \log_2 \left\lceil \frac{N^2}{B_m p} \right\rceil) - \frac{N^2}{B_m p}\right) 2\tau$$

The complexity for startups is $O(p)$ and grows exponentially with the dimension $d = \log_2 p$ of the cube. This can be seen easily where, ignoring B_m, the complexity becomes

$$T_{comm}^{TS} = d\frac{N^2}{p}t_c + (\sum_{i=0}^{d-1} 2^i)2\tau.$$

When $\frac{N^2}{p} > B_m$, we must take into account these extra start-ups and the number of start-ups is $O(p + \log_2 p \left\lceil \frac{N^2}{2pB_m} \right\rceil)$.

Buffered communication makes sense only when $\frac{N^2}{p}$ remains small and the complexity is approximately $O(\log_2 p)$ start-ups growing linearly with cube dimension. Here one must also take into account the extra time required for buffering as well as the fact that the effective buffer is small, so that on large problems the transpose is essentially unbuffered.

The Local-Distributed method has communication complexity

$$T_{comm}^{LD} = 2d\frac{N^2}{p}t_c + 2d\tau\max(1, \left\lceil \frac{N^2}{B_m p} \right\rceil).$$

as does the Block method.

$$T_{comm}^{B} = T_{comm}^{LD}$$

Here the complexity only grows linearly with the number of cube dimensions, however, there is twice as much data to transfer and when the problem gets large, we get a measurement proportional to $\left\lceil \frac{N^2}{B_m p} \right\rceil$ times the start-up costs. Thus the complexity of the distributed methods is $O(\log_2 p)$ start-ups when $\frac{N^2}{p} < B_m$ and $O(\log_2 p \left\lceil \frac{N^2}{B_m p} \right\rceil)$ when $\frac{N^2}{p} > B_m$.

In summary,

$$\begin{aligned}
T_{comm}^{TS} &\asymp 2(p-1)\tau + (\log_2 p)\frac{N^2}{p}t_c \\
T_{comm}^{LD} &\asymp (2\log_2 p)\left(\frac{N^2}{p}t_c + \tau\right) \\
T_{comm}^{B} &\asymp (2\log_2 p)\left(\frac{N^2}{p}t_c + \tau\right)
\end{aligned}$$

Table 5 illustrates data transfer time (in milliseconds) without any computation. Several different Transpose methods are also given, unbuffered, buffered and the direct transpose method. One can see that for small problems, the buffered recursive transpose and the distributed methods take about the same time communicating, even though the distributed methods send twice as much data. This is because the data lie in contiguous locations and thus require only one message. Direct transpose, where each processor sends block i to processor i and receives a block from each processor, is always worse on the Intel since each processor cannot send and receive more than one message per step. In direct transpose, each processor sends and receives exactly $p-1$ messages. This is the same number of messages as unbuffered transpose. Therefore even though our measurements of direct transposing are slower than recursive block transpose, on systems which support multiple message trafficking, this can actually be a simple and efficient transpose method. Finally by comparing buffered and unbuffered transposing we can find the cut-off point beyond which it makes no difference whether to buffer or not to buffer. Here we see that buffering just does not matter when the size is larger than 64×64. The communication times of the distributed methods are comparable to buffered transpose times until the problem size gets larger enough so that the maximum packet size B_m is reached. Here we see that for the 128×128 problem, communication for the distributed methods is roughly twice the time for transposing. Since the number of data points moved is also twice that of transposing, this is consistent. For $d = 4$, the timings are about equal since the message packets are small enough so that they can be all transferred in one step.

Next we model computational time on vector nodes. Hockney and Jesshope (1981) give a model for performing m independent transforms of length n by using the best serial algorithm and vectorizing the arithmetic. Let α^{-1} be the per flop computation time, and $n_{\frac{1}{2}}$ be the vector length required to achieve half of the asymptotic performance. Then

$$T_{multft} = 5\alpha n(n_{\frac{1}{2}} + m)\log_2 n$$

Using this model we see that for the Transpose-Split method each processor

Analysis of Communication w/o Computation					
size	Transpose-Split			Local-Distr.	Block
dim	unbuff.	buff.	direct	data transfer	data transfer
16 × 16					
1	5	5	5	5	–
2	10	10	40–50	10	10–15
3	20–60	65–70	20–120	10–45	15–20
4	105–110	75–140	50–140	70–110	65–70
32 × 32					
1	10	10	10	20	–
2	15	10	45–90	20	20–25
3	20–65	20	60–110	15–20	25–60
4	45–140	20–25	85–225	20–25	25–75
64 × 64					
1	30	25–30	30	55	–
2	35–70	30–65	85–100	60	60–65
3	35–75	30–75	90–200	50–85	50–90
4	105–110	35–90	175–250	35–85	45–105
128 × 128					
1	115	110	110	220–225	–
2	110–115	110–115	210–320	220–225	225–230
3	140	90–95	155–345	170	175–180
4	80–125	80–125	280–460	120–150	175–180
256 × 256					
3	335–340	330–335	690–1150	660–690	670–760
4	235–240	235–345	500–1125	445–450	450–455

Table 5: Communication Times without Intervening Computation.

does $\frac{N}{p}$ transforms of length N twice, hence

$$T^{TS}_{comp} = 2\alpha(5N(n_{\frac{1}{2}} + \frac{N}{p}) \log_2 N)$$

For the Block method, each processor is responsible for $\frac{1}{\sqrt{p}}$th of the work for $\frac{N}{\sqrt{p}}$ transforms of length N and this occurs twice, so

$$T^{B}_{comp} = 2\alpha(5\frac{N}{\sqrt{p}}(n_{\frac{1}{2}} + \frac{N}{\sqrt{p}}) \log_2 N)$$

Finally for the Local-Distributed, each processor does $\frac{N}{p}$ transforms of length N during the horizontal phase, and $\frac{1}{p}$th of the work for N transforms of length N during the vertical stage.

$$T^{LD}_{comp} = 5\alpha N(n_{\frac{1}{2}} + \frac{N}{p}) \log_2 N + 5\alpha\frac{N}{p}(n_{\frac{1}{2}} + N) \log_2 N$$

The difference in computation time comes from the $n_{\frac{1}{2}}$ term, with $T^{B}_{v} = \frac{N}{\sqrt{p}}n_{\frac{1}{2}}$, $T^{TS}_{v} = Nn_{\frac{1}{2}}$ and $T^{LD}_{v} = \frac{1}{2}(N + \frac{N}{p})n_{\frac{1}{2}}$. Comparing coefficients, we see that

$$T^{B}_{comp} < T^{LD}_{comp} < T^{TS}_{comp} \quad \text{when } p > 1$$

Hence in terms of vectorization, the Block method is at an advantage which increases with increasing parallelism as measured by $n_{\frac{1}{2}}$.

All of the vectorized FFT implementations require an internal transform of the data in order to set up the vectors in the correct orientation. Since each processor has $\frac{N^2}{p}$ points the overhead here is $t_{copy}\frac{N^2}{p}$.

Our final model combines the communication and computational portions in a straight-forward manner.

$$T^{TS} = 2\alpha(5N(n_{\frac{1}{2}} + \frac{N}{p}) \log_2 N) + 2(p-1)\tau + (\log_2 p)\frac{N^2}{p}t_c$$

$$T^{B} = 2\alpha(5\frac{N}{\sqrt{p}}(n_{\frac{1}{2}} + \frac{N}{\sqrt{p}}) \log_2 \frac{N}{\sqrt{p}}) + (2\log_2 p)(\frac{N^2}{p}t_c + \tau)$$

$$T^{LD} = 5\alpha N(n_{\frac{1}{2}} + \frac{N}{p}) \log_2 N + 5\alpha\frac{N}{p}(n_{\frac{1}{2}} + N) \log_2 \frac{N}{p} + (2\log_2 p)(\frac{N^2}{p}t_c + \tau)$$

An analysis of the communication time shows that the coefficient of the t_c term is of the same order, but that of the τ term is clearly against the Transpose-Split method. Since the τ term represents the latency time or startup time for each communication and is several orders of magnitude larger than t_c, it is obvious that as p increases, the overhead for transpose communication will become significant. Of course buffering can reduce this overhead. However the minimum number of communications needed for recursive block transpose is still of order $\log_2 p$. Therefore, the communication needs of the distributed methods present a lower bound for the transpose communication.

8 Conclusion

After consideration of the model and timing results we can draw the following conclusions.

- The distributed methods are hurt by the interruption of computation during the trans-processor butterfly stages, hence their performance should be enhanced on systems which interleave computation and communication.
- The complexity of communication for distributed methods is $\log_2 p$, therefore they could show promise on systems where p is large.
- In the presence of vector processor nodes, the Block method exhibits more parallelism than any of the other methods since its working vector length is approximate $\sqrt{p}$ times longer.
- The Transpose-Split method is superior on systems which support an efficient transpose algorithm since its major deficiency is the asymptotic complexity of recursive block transpose communication growing exponentially to the number of dimensions. Presently on the Intel iPSC/System 286 where each node is connected by an Ethernet communication channel, the penalties for traversing several nodes to get a message across increases linearly with the distance. Therefore on the current system, Transpose-Split is likely to be superior when the number of processors is small.

- Future generations of hypercubes will likely support more efficient transpose algorithms and routing hardware, making the Transpose-Split methodology more efficient. We have already seen the difference between transpose methods in Table 5.

- The Local-Distributed method is a compromise between the first two methods. In our implementation it is competitive with the Transpose-Split method and can be used when one wishes to avoid the transpose.

Our implementation results show that on the Intel iPSC/System 286 without vector boards on the nodes, there is essentially no difference between the three methods. The final analysis is that this problem is highly system dependent, and one should be aware of the advantages and disadvantages of these different methods in order to best utilize the parameters of a particular system.

9 Acknowledgements

I appreciate the helpfulness of Ching-Tien Ho, Yale University for sending me his Transpose subroutine without which I could not have done this comparison. I am also grateful to Charles Van Loan, my thesis advisor, who provided many suggestions concerning this study. Finally I would like to thank Oliver McBryan for sharing some of his insights with me and Rodney Fazzari for clarifying the Floating Point System T-Series implementation.

10 Bibliography

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

Chu, Clare. 1987. "Load Balanced FFT Implementations on the Intel iPSC," Dept. of Computer Science Tech. Report. (TR 87-824) Cornell University, Ithaca, New York.

Gustafson, John. 1987. "Ensemble FFT's as a Function of Compute/Communication Ratios," in Charles Van Loan ed., *Fast Fourier Transforms for Vector and Parallel Computers Workshop*, Sponsored by The Mathematical Sciences Institute, Cornell University, Ithaca, New York. March 22–25, 1987.

Ho, Ching-Tien, and Johnsson, S. Lennart. Sept. 1986. "Matrix Transposition on Boolean n-cube Configured Ensemble Architectures." Technical Report YALEU/DCS/TR-494, Yale University.

Hockney, R., and Jesshope, C. 1981. *Parallel Computers: Architecture, Programming and Algorithms.* Adam Hilger, Bristol.

Johnsson, S. Lennart. January 1985a. "Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures," Technical Report YALEU/CDS/RR-361, Yale University.

———. February 1985b. "Data Permutations and Basic Linear Algebra Computations on Ensemble Architectures," Technical Report YALEU/-CSD/RR-367, Yale University.

McBryan, Oliver A. and Van de Velde, Eric F. 1985. "Hypercube Algorithms and Implementations," Presented to the 2nd SIAM Conference on Parallel Computing, Norfolk, Nov. 18–21, 1985 also in *SIAM Journal on Scientific and Statistical Computing,* vol. 8, no. 2, pp. s227–s287, March 1987.

Miles, Douglas, Kinney, Paul, Groshong, Judson, and Fazzari, Rodney. 1987. "Specification and Performance Analysis of Six Benchmark Programs for the FPS T Series," Floating Point Systems, Inc, P.O. Box 23489, Portland, OR 97223.

Nussbaumer, Henri J. 1982. *Fast Fourier Transform and Convolution Algorithms.* Springer-Verlag, Berlin Heidelberg, 1982.

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

Van Loan, Charles F. 1987. "FFT's From the Matrix Point of View," manuscript.

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

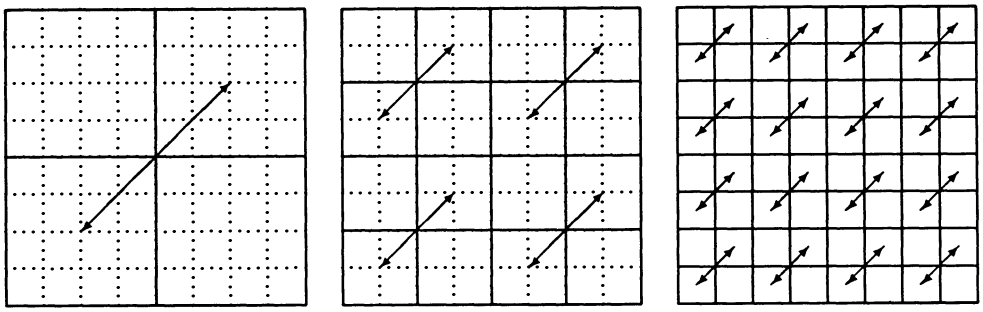

Figure 4: Block Recursive Transpose.

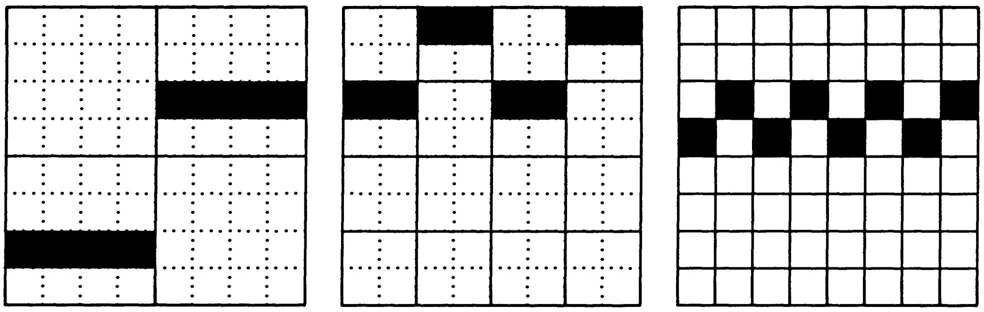

Figure 5: Blocks Exchanged by Processor # 2.