Comparison of Two-Dimensional FFT Methods on the Hypercube

Clare Y. Chu*
Northrop Corp., Aircraft Division
One Northrop Avenue, 3812/82
Hawthorne, CA 90250

Abstract

Complex two-dimensional FFTs up to size 256 × 256 points are implemented on the Intel iPSC/System 286 hypercube with emphasis on comparing the effects of data mapping, data transposition or communication needs, and the use of distributed FFTs. Two new implementations of the 2D-FFT include the Local-Distributed method which performs local FFTs in one direction followed by distributed FFTs in the other direction, and a Vector-Radix implementation that is derived from decimating the DFT in two-dimensions instead of one. In addition, the Transpose-Split method involving local FFTs in both directions with an intervening matrix transposition and the Block 2D-FFT involving distributed FFT butterflies in both directions are implemented and compared with the other two methods. Timing results show that on the Intel iPSC/System 286, there is hardly any difference between the methods, with the only differences arising from the efficiency or inefficiency of communication. Since the Intel cannot overlap communication and computation, this forces the user to buffer data. In some of the methods, this causes processor blocking during communication. Issues of vectorization, communication strategies, data storage and buffering requirements are investigated. A model is given that compares vectorization and communication complexity. While timing results show that the Transpose-Split method is in general slightly faster, our model shows that the Block method and Vector-Radix method have the potential to be faster if the communication difficulties were taken care of. Therefore if communication could be "hidden" within computation, the latter two methods can become useful with the Block method vectorizing the best and the Vector-Radix method having 25% fewer multiplications than row-column 2D-FFT methods. Finally the Local-Distributed method is a good hybrid method requiring no transposing and can be useful in certain circumstances. This paper provides some general guidelines in evaluating parallel distributed 2D-FFT implementations and concludes that while different methods may be best suited for different systems, better implementation techniques as well as faster algorithms still perform better when communication become more efficient.

*This work was done by the author at Cornell University, Ithaca NY.

1 Introduction

Multidimensional Fourier transforms, as in the single dimensional case, can also be broken into pieces that can be done in parallel. The possibilities are even richer here. This is because multidimensional transforms can be done either as a sequence of separable one-dimensional transforms, or by directly splitting them into blocks of smaller multidimensional transforms, as in the vector-radix methods [2,9]. We study only the case of the two-dimensional Fourier transform because the discussion and algorithmic methods can be extended directly to computing higher dimensional Fourier transforms.

Work on two-dimensional FFTs on distributed processors has so far been restricted to the row-column approach. The strips method partitions the two-dimensional array, or matrix, into rows, mapping block rows into processors. The transform of each row is then found, and the matrix is transposed before a second row transform pass is done on rows that previously had been columns. We follow [5] in terming this approach the *Transpose-Split* (TS) 2D-FFT.

Another row-column method partitions the matrix into blocks of submatrices, assigning one block per node. The hypercube is then viewed as a two-dimensional cross-product of smaller dimensional hypercubes with distributed FFTs performed along both the rows and the columns. No transposing of data is needed here. We term this method the *Block* (B) 2D-FFT method.

We present two new methods of implementing 2D-FFTs. The first one is a row-column approach that partitions data into strips much like the Transpose-Split method. The difference is that no transpose is done between the horizontal and vertical steps. Instead, the horizontal FFTs are done locally inside each processor and the vertical FFTs are distributed. We call this the *Local-Distributed* (LD) 2D-FFT method.

Finally we discuss the implementation of the Vector-Radix 2D-FFT on the hypercube and show that this method has promise, although the communication of the old iPSC hampered the timings obtained. In order to give a "fair" comparison with the other row-column methods we have chosen to implement a *partial* Vector-Radix (PVR) 2D-FFT on the hypercube. What this means is that the individual 2D-FFTs that are done locally inside the processors are row-column 2D-FFTs, however, the distributed steps use the vector radix update scheme. As noted in [2,9], the serial 2D Vector-Radix method has a 25% reduction in multiplications and fewer butterflies than a serial row-column 2D-FFT.

The Transpose-Split 2D-FFT is favored by some because all FFT computations are performed locally. The only communication that takes place occurs within the transpose step. Gustafson [1] 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. Therefore all that is needed to effectively implement this method is a fast efficient matrix transpose procedure. See [3] for an in-depth analysis of the hypercube matrix transpose problem.

The Block method was implemented in [6] on the Floating Point Sys-

tems T-Series hypercube. By considering the signal flow graph of the radix-2 FFT algorithm, we see that this implementation requires communication during both the vertical and 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 is 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. The powerful vector boards on the T-Series allows this method to be used advantageously.

The Local-Distributed method and the Vector-Radix-2 method are implemented on the Intel iPSC. The Intel iPSC, unlike the NCUBE and T-Series machines, does not allow simultaneous communication. This is detrimental for the Vector-Radix-2 method as the distributed stages involve total exchanges between four processors instead of two. A total exchange within a subcube of processors means that each processor in the subcube exchanges data with every other processor in the subcube. This particular property of Intel communication also means that the full cross-bar interconnection scheme cannot be simulated efficiently, and 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 FFTs will exhibit blocking during the distributed butterfly steps caused by one processor waiting for data from another. The Local-Distributed method does not require a transpose and does distributed FFTs along only one direction instead of two (the Block method). Meanwhile the Vector-Radix FFT performs local 2D-FFTs followed by distributed stages requiring the summation and multiplication of the local FFTs by "twiddle" factors. The Vector-Radix FFT has a lot of potential that is not reflected in our implementation on the Intel iPSC precisely because of communication inefficiencies. But we think it is useful to offer it as an alternative to the row-column approach because of its rich parallelism.

2 Two Dimensional FFT Algorithms

2.1 Row-Column

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

$$\mathbf{y} \leftarrow \mathbf{F}_n \mathbf{x}$$

where $\mathbf{F}_n$ is the matrix consisting of powers of the nth root of unity $\omega_n = e^{-2\pi i/n}$.

$$[\mathbf{F}_n]_{jk} = \omega_n^{jk}$$

The two-dimensional (2-D) discrete Fourier transform (DFT) of a two-dimensional array $\mathbf{X} \in \mathcal{C}^{n_1 \times n_2}$ is defined as

$$\mathbf{Y} \leftarrow \mathbf{F}_{n_1} \mathbf{X} \mathbf{F}_{n_2}^T$$

The matrix notation clearly demonstrates the row-column or column-row method of computing the 2-D transform, since matrix-multiplication is associative. If the fast Fourier transform (FFT) is used to evaluate the 1-D FFTs along both the rows and the columns, the number of complex multiplications required is $n^2 \log_2 n$ for $n = n_1 = n_2$. In addition, a matrix transposition algorithm is required.

A one-dimensional FFT of a long vector of length $n = n_1 \cdot n_2$ can be computed in a 2-D "fashion" by viewing it as a DFT of an array of size $n_1 \times n_2$ [7], that is, by writing $\mathbf{x}$ as an array $\mathbf{x}_{n_1 \times n_2}$. We can compute the n-point DFT of $\mathbf{x}$ by an n_1-point FFT of the rows, a point-wise multiplication of $\mathbf{x}$ by the *twiddle factors*, followed by another n_2-point FFT on the columns. The matrix of twiddle factors $\mathbf{T}$ is defined

$$[\mathbf{T}]_{jk} = \omega_n^{jk}, \quad j = 0, \ldots, n_1 - 1; \quad k = 0, \ldots, n_2 - 1.$$

and $*$ denotes the point-wise multiplication of two matrices. Hence the DFT $\mathbf{y}$ of $\mathbf{x}$ is another two-dimensional array given by

$$\mathbf{y}^T_{n_2 \times n_1} = \mathbf{F}_{n_1}[(\mathbf{T}_{n_1 \times n_2}) * [\mathbf{x}_{n_1 \times n_2} \mathbf{F}_{n_2}]]$$

The row-column or column-row method can be used to compute the horizontal and vertical DFTs.

2.2 Vector-Radix

The Vector-Radix FFT is a direct decomposition of the two-dimensional DFT into sums of smaller two-dimensional DFTs multiplied by "twiddle factors", (the diagonal matrix $\mathbf{\Delta}$). Here a 2-D DFT is recursively broken down into four 2-D DFTs until only trivial 2D-DFTs need to be evaluated. The number of complex multiplications is now $\frac{3}{4} n^2 \log_2 n$, 25% lower than the row-column method [2,9]. Moreover, no matrix transpose routine is required.

The recursive block structure of the DFT matrix $\mathbf{F}_n$ is used in two-dimensions to derive the method. The matrix $\mathbf{X} \in \mathcal{C}^{n \times n}$ is segregated into four sets; one over those samples $\mathbf{X}(j,k)$ for which j and k are both even, one for which j is even and k is odd, one for which j is odd and k is even and one for which both j and k are odd.

Theorem 2.1 *Let $\mathbf{X} \in \mathcal{C}^{n \times n}$ with $n = 2^t$, then the 2D vector-radix splitting of the 2D-DFT of $\mathbf{X}$ is*

$$\begin{aligned} \mathbf{F}_n \mathbf{X} \mathbf{F}_n^T &= (\mathbf{F}_n \mathbf{\Pi}_n)(\mathbf{M}_n \mathbf{X} \mathbf{M}_n^T)(\mathbf{\Pi}_n^T \mathbf{F}_n^T) \\ &= \begin{bmatrix} \mathbf{I}_{n/2} & \mathbf{\Delta}_{n/2} \\ \mathbf{I}_{n/2} & -\mathbf{\Delta}_{n/2} \end{bmatrix} \begin{bmatrix} \widehat{\mathbf{X}}_{11} & \widehat{\mathbf{X}}_{12} \\ \widehat{\mathbf{X}}_{21} & \widehat{\mathbf{X}}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{n/2} & \mathbf{I}_{n/2} \\ \mathbf{\Delta}_{n/2} & -\mathbf{\Delta}_{n/2} \end{bmatrix} \end{aligned}$$

where $\mathbf{\Delta}_{n/2} = diag(1, \omega_n, \ldots, \omega_n^{n/2-1})$ and

$$\begin{aligned} \widehat{\mathbf{X}}_{11} &= \mathbf{F}_{n/2} \mathbf{X}(0:2:n-2, 0:2:n-2) \mathbf{F}_{n/2}^T \\ \widehat{\mathbf{X}}_{12} &= \mathbf{F}_{n/2} \mathbf{X}(0:2:n-2, 1:2:n-1) \mathbf{F}_{n/2}^T \\ \widehat{\mathbf{X}}_{21} &= \mathbf{F}_{n/2} \mathbf{X}(1:2:n-1, 0:2:n-2) \mathbf{F}_{n/2}^T \\ \widehat{\mathbf{X}}_{22} &= \mathbf{F}_{n/2} \mathbf{X}(1:2:n-1, 1:2:n-1) \mathbf{F}_{n/2}^T \end{aligned}$$

The $\mathbf{\Pi}_n$, $\mathbf{M}_n$ are permutation matrices for the perfect shuffle and inverse perfect shuffle operators, respectively.

Proof From [11], we have

$$\mathbf{F}_n \mathbf{\Pi}_n = \begin{bmatrix} \mathbf{I} & \mathbf{\Delta} \\ \mathbf{I} & -\mathbf{\Delta} \end{bmatrix} \begin{bmatrix} \mathbf{F}_{n/2} & 0 \\ 0 & \mathbf{F}_{n/2} \end{bmatrix}$$

Applying this to both sides of $\mathbf{M}_n \mathbf{X} \mathbf{M}_n^T$ gives the required decomposition.

———*———

Theorem 2.1 is the basic two-dimensional Cooley-Tukey (CT2) splitting of the Vector-Radix method for computing the 2-D FFT. The complete algorithm is obtained by recursive application of this basic decomposition.

Algorithm 2.1 Vector-Radix 2-D FFT

$n = 2^t$
$\mathbf{X} \leftarrow \mathbf{P}_n \mathbf{X} \mathbf{P}_n^T$
for $q = 1:t$
  $L \leftarrow 2^q$
  $\mathbf{\Delta}_{L/2} \leftarrow diag(1, \omega_L, \ldots, \omega_L^{L/2-1})$

$$\mathbf{X} \leftarrow \left[\mathbf{I}_{n/L} \otimes \begin{bmatrix} \mathbf{I}_{L/2} & \mathbf{\Delta}_{L/2} \\ \mathbf{I}_{L/2} & -\mathbf{\Delta}_{L/2} \end{bmatrix}\right] \mathbf{X} \left[\mathbf{I}_{n/L} \otimes \begin{bmatrix} \mathbf{I}_{L/2} & \mathbf{I}_{L/2} \\ \mathbf{\Delta}_{L/2} & -\mathbf{\Delta}_{L/2} \end{bmatrix}\right]$$

end

Here $\mathbf{P}_n$ is the bit-reversal permutation matrix.

Table 1: Multiple Transforms and Vector Length.

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

†# of FFTs = vector length

‡length of portion in each processor

3 Comments on Data Ordering and Vectorization

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 FFTs are performed locally so that this permutation is also done independently and locally. We can also avoid bit-reversing by using Stockham FFTs locally, albeit this necessitates an extra array of workspace. In any case, no distributed bit-reversing operation is necessary. The Local-Distributed methods, require up to an extra d communications [10], on a cube which can communicate simultaneously in all directions, to perform distributed bit-reversal. The Block and the Vector-Radix FFT results in a need for distributed bit-reversal along both the horizontal and vertical directions, or an extra $2\sqrt{d}$ communications if Swarztrauber's [10] method is utilized. On the Intel cube, autosort FFTs 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 effectively vectorize the butterfly computations. 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 [4,10].

The basic operation here is a vector SAXPY where

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

$\mathbf{v}$, $\mathbf{x}$, 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 FFTs of length N simultaneously and thus has an effective vector length of N/p. The Block method has each processor responsible for $N/\sqrt{p}$ FFTs of which 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.

4 Implementation and Timings

The Transpose-Split, Local-Distributed, Block and partial Vector-Radix methods for the two dimensional FFT are 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 are *not* available so that all computations within a node are done serially. The Transpose-Split method uses local FFTs 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.

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 Vector-Radix method is implemented only partially. In other words, the local independent 2D-DFTs are done by a conventional row-column 2D-FFT subroutine found in [8], and only the distributed steps involve updating by the Vector-Radix method. Since our hypercube has only 16 nodes, only two such distributed steps are done. In order to perform in-place computation and do away with the need for extra buffers, the Vector-Radix updating was done so that each processor in the corner of its two-dimensional subcube was responsible for all the updating of that particular corner of the data for its three partners as well as for itself. For example, the processor in the south-west corner of the two-dimensional subcube for that iteration will send its north-west corner to the processor above it in exchange for the south-west corner of that processor. It will also send its north-east corner to the processor diagonally across from it in exchange for the south-west corner of that processor. And it will send its south-east corner to the processor to its right in exchange for the south-east corner of that processor. This processor will update all the submatrices it receives and then do a reverse exchange where all its partners get their own updated corners back.

Computation time as well as communication time is displayed for all four methods. A portion of the communication time is reflected in a processor blocking while awaiting data that it is to receive. The total execution time is shown in Table 2. Table 3 shows the computation time required by each method while Table 4 displays the communication time whereas Table 5 displays the amount of time a processor spends blocked. Times are given in milliseconds and range from the fastest processor to the slowest processor.

The results show timings that are roughly within 10% of each other for the four different methods. However if we look at the break-down of communication versus computational time, some interesting differences surface.

The computational times of the Transpose-Split method and the Vector-Radix method are the most load balanced. The Transpose-Split computations exhibit the most parallelism as all of the actual FFT steps are independent and done in parallel. For the Vector-Radix method the individual 2D-FFTs done locally within each processor are also done entirely independently and in parallel. As expected, the computational times in the distributed row-column methods show a small amount of imbalance, with greater gaps between the faster and slowest processor than shown by the Transpose-Split method and the Vector-Radix 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 the three row-column 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 FFTs, 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. The computational time for the Vector-Radix method is on the average faster than any of the row-column methods, especially on large problems.

Table 2: Total Execution Time.

size	Total Execution Time			
dim	TS	LD	B	PVR
16 × 16				
1	155	145–160	–	–
2	90	80–95	90–110	160
3	75–125	55–70	65–85	–
4	140–145	105–150	90–110	125–190
32 × 32				
1	710	665–715	–	–
2	385–390	340–395	365–430	465–470
3	230–275	180–230	235–290	–
4	170–220	105–160	110–205	215–330
64 × 64				
1	3280	3110–3305	–	–
2	1700	1550–1750	1675–1895	1850–1895
3	910–945	785–935	1045–1230	–
4	525–570	415–585	425–555	685–840
128 × 128				
1	15130	15025–15230	–	–
2	7825–7840	7165–7930	7820–8630	7735–7820
3	3965	3575–4145	4835–5480	–
4	2155	1800–2225	1895–2350	2345–2490
256 × 256				
3	17650–17665	16870–18605	22220–24660	–
4	9040–9215	8135–9615	8695–10370	9040–9215

Keep in mind that we have not taken advantage of the 25% reduction of multiplications since our Vector-Radix implementation does not recurse all the way down to the 2×2 trivial 2-D transform. Instead, only the distributed steps are done via the Vector-Radix splitting and the processor local 2D-FFTs are done by the regular row-column approach. Hence even at this limited level, we see that the Vector-Radix shows potential in speeding up computation.

The analysis of the communication times show a different story. Here the three row-column methods exhibit roughly the same range of times for communication. As already mentioned, our implementation of the Vector-Radix 2-cube total exchange is very primitive since the Intel iPSC cannot communicate across more than one link at the same time. Hence a huge amount of blocking is seen to be responsible for the slowness of the Vector-Radix communication. The blocking time for the three row-column methods is about the same. This demonstrates that the load imbalance of the distributed methods is not really much of a problem as far as blocking between send and recv are concerned. One interesting point is that the Transpose-Split communication 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.

In the next section we 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.

5 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.

Table 3: Computational Time of 2D-FFT.

size	Computation Time			
dim	TS†	LD	B	PVR
16 × 16				
1	150/10	140–155	–	–
2	90/10	70–85	80–100	90–100
3	55/5	40–65	50–70	–
4	40/5	15–40	25–40	30–45
32 × 32				
1	705/25	645–695	–	–
2	375/15	320–380	345–410	380–385
3	210/10	155–215	210–270	–
4	125/10	85–120	80–135	115–130
64 × 64				
1	3250/80	3055–3250	–	–
2	1670/50	1490–1690	1615–1835	1625–1630
3	880/30	735–885	995–1180	–
4	480/20	380–470	385–515	445–460
128 × 128				
1	15015/285	14805–15010	–	–
2	7575/155	6935–7705	7565–8405	7235–7250
3	3870/90	3400–3975	4660–5315	–
4	2015/60	1680–2075	1775–2225	1895–1915
256 × 256				
3	17325/315	16150–17875	21515–23975	–
4	8815/185	7650–9145	8205–9910	8300–8320

†total computation time and time for internal transpose

Table 4: Communication Time Including Blocking Overhead.

size	Communication Time/Blocked Time			
dim	TS	LD	B	PVR
16 × 16				
1	5	5	–	–
2	5–10	5–10	5–10	60–65
3	15–70	60–65	5–10	–
4	95–110	65–120	55–60	80–155
32 × 32				
1	5	15	–	–
2	10–15	15–20	15–20	85–90
3	25–65	15–20	15	–
4	35–95	5–20	15–65	90–205
64 × 64				
1	30	55–60	–	–
2	30–35	60	50–55	220–265
3	25–70	45–85	40–45	–
4	40–90	35–40	30–40	230–385
128 × 128				
1	110	220	–	–
2	245–265	220	220–255	495–570
3	80–95	165–170	160–165	–
4	115–145	105–120	105–120	445–585
256 × 256				
3	310–335	650–710	660–695	–
4	230–400	445–510	435–495	860–1355

Table 5: Time Spent Blocked during Communication

size	Blocked Time			
dim	TS	LD	B	PVR
16 × 16				
1	0	0	–	–
2	0–5	0–5	0–5	10–45
3	5–60	0–60	0–5	–
4	15–85	0–110	5–50	15–90
32 × 32				
1	0	0	–	–
2	0–5	0–5	5	30–65
3	0–50	0–10	0–5	–
4	5–65	0–15	0–60	30–175
64 × 64				
1	5	10	–	–
2	5	0	10	10–175
3	5–20	5–55	0–15	–
4	5–60	0–15	5–10	115–355
128 × 128				
1	15–40	35	–	–
2	20–155	30–35	35–60	60–340
3	10–25	20–30	20–25	–
4	20–85	10–20	15–20	125–405
256 × 256				
3	40–60	90–150	95–130	–
4	45–220	65–170	60–135	95–715

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 row-column FFT algorithms have d steps involving trans-processor butterflies. Each step requires the exchange of n^2/P points. One can see here that twice as much data points are exchanged at each step. However since these points are all contiguous there is no overhead of sending multiple messages nor the 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. Three extra buffers would be needed for the Vector-Radix method *if* we did not use this trick of exchanging twice per distributed step. Therefore in our implementation, we incur the cost of the extra exchange and hence no extra buffers are necessary.

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 we look at unbuffered transpose communication. Ho and Johnsson [3] show that the complexity for unbuffered communication is

$$T^{TS}_{comm} = d\frac{n^2}{P}t_c + \left(P + \left\lceil \frac{n^2}{2B_mP} \right\rceil \min(d, \log_2 \left\lceil \frac{n^2}{B_mP} \right\rceil) - \frac{n^2}{B_mP}\right) 2\tau$$

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

$$T^{TS}_{comm} = d\frac{n^2}{P}t_c + (\sum_{i=0}^{d-1} 2^i)2\tau.$$

When $\frac{n^2}{P} > B_m$, we must take into account these extra start-ups and the number of start-ups is $O(P + \log_2 P \lceil n^2/2PB_m \rceil)$.

Buffered communication makes sense only when 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

$$T^{LD}_{comm} = 2d\frac{n^2}{P}t_c + 2d\tau\max(1, \left\lceil \frac{n^2}{B_mP} \right\rceil).$$

as does the Block method.

$$T^{B}_{comm} = T^{LD}_{comm}$$

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 $\lceil n^2/B_mP \rceil$ times the start-up costs. Thus the complexity of the distributed methods is $O(\log_2 P)$ start-ups when $n^2/P < B_m$ and $O(\log_2 P \lceil n^2/B_mP \rceil)$ when $n^2/P > B_m$.

The communication complexity of our in-place Vector-Radix FFT necessitates two exchanges per distributed step. Recall that this scheme is similar to that of the Floating Point Systems implementation [6] and also Walton's [12] one-dimensional FFT implementation where portions of the matrix or vector are exchanged and then sent "home". Here each portion is of size $\frac{3}{4}n^2/P$ and each processor communicates with three processors. Hence the communication complexity is

$$T^{PVR}_{comm} = 4\log_2\sqrt{P}(\frac{3}{4}\frac{n^2}{P}t_c + 3\tau)$$

If the three exchanges per distributed step can be done simultaneously, the startup term becomes just

$$4\log_2\sqrt{P}\tau$$

In summary,

$$\begin{aligned}
T^{TS}_{comm} &\asymp 2(P-1)\tau + (\log_2 P)\frac{n^2}{P}t_c \\
T^{LD}_{comm} &\asymp (2\log_2 P)\left(\frac{n^2}{P}t_c + \tau\right) \\
T^{B}_{comm} &\asymp (2\log_2 P)\left(\frac{n^2}{P}t_c + \tau\right) \\
T^{PVR}_{comm} &\asymp (2\log_2 P)\left(\frac{3}{4}\frac{n^2}{P}t_c + 3\tau\right)
\end{aligned}$$

Table 6 illustrates data transfer time (in milliseconds) without any computation. Two different transposes are compared for the Transpose-Split method, the unbuffered and buffered methods described in [3]. 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. 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 large enough so that the maximum packet size B_m is reached. Here we see that for the 128 × 128 problem, communication for the distributed row-column 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. Finally as expected the Vector-Radix communication was the slowest because of the inefficient implementation on the Intel iPSC. It is expected, however that given an efficient "total exchange" capability where processors can communicate simultaneously, that the communication times should speed up making the Vector-Radix method viable and feasible.

Table 6: Communication Times without Intervening Computation.

Analysis of Communication w/o Computation					
size	TS		LD	B	PVR
dim	unbuff.	buff.			
16 × 16					
1	5	5	5	–	–
2	10	10	10	10–15	50–90
3	20–60	65–70	10–45	15–20	–
4	105–110	75–140	70–110	65–70	10–115
32 × 32					
1	10	10	20	–	–
2	15	10	20	20–25	45–80
3	20–65	20	15–20	25–60	–
4	45–140	20–25	20–25	25–75	15–190
64 × 64					
1	30	25–30	55	–	–
2	35–70	30–65	60	60–65	55–175
3	35–75	30–75	50–85	50–90	–
4	105–110	35–90	35–85	45–105	85–285
128 × 128					
1	115	110	220–225	–	–
2	110–115	110–115	220–225	225–230	110–235
3	140	90–95	170	175–180	–
4	80–125	80–125	120–150	175–180	95–385
256 × 256					
3	335–340	330–335	660–690	670–760	–
4	235–240	235–345	445–450	450–455	295–740

Next we model computational time for the three row-column methods on vector nodes. Hockney and Jesshope [4] 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 $N_{1/2}$ be the vector length required to achieve half of the asymptotic performance. Then

$$T_{multft} = 5\alpha N(N_{1/2} + M)\log_2 N$$

Using this model we see that for the Transpose-Split method each processor does n/P transforms of length n twice, hence

$$T^{TS}_{comp} = 2\alpha(5n(N_{1/2} + \frac{n}{P})\log_2 n)$$

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

$$T^{B}_{comp} = 2\alpha(5\frac{n}{\sqrt{P}}(N_{1/2} + \frac{n}{\sqrt{P}})\log_2 n)$$

Finally for the Local-Distributed, each processor does n/P transforms of length n during the horizontal phase, and $(1/P)$th of the work for n transforms of length n during the vertical stage.

$$T^{LD}_{comp} = 5\alpha n(N_{1/2} + \frac{n}{P})\log_2 n + 5\alpha\frac{n}{P}(N_{1/2} + n)\log_2 n$$

The difference in computation time among the three methods comes from the $N_{1/2}$ term, with $T^{B}_{v} = (n/\sqrt{P})N_{1/2}$, $T^{TS}_{v} = nN_{1/2}$ and $T^{LD}_{v} = 1/2(n + n/P)N_{1/2}$. 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 $N_{1/2}$.

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 n^2/P points the overhead here is $n^2/P \cdot t_{copy}$.

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

$$T^{TS} = 2\alpha(5n(N_{1/2} + \frac{n}{P})\log_2 n) + 2(P-1)\tau + (\log_2 P)\frac{n^2}{P}t_c$$

$$T^{B} = 2\alpha(5\frac{n}{\sqrt{P}}(N_{1/2} + \frac{n}{\sqrt{P}})\log_2 \frac{n}{\sqrt{P}}) + (2\log_2 P)(\frac{n^2}{P}t_c + \tau)$$

$$\begin{aligned} T^{LD} &= 5\alpha n(N_{1/2} + \frac{n}{P})\log_2 n + 5\alpha\frac{n}{P}(N_{1/2} + n)\log_2 \frac{n}{P} \\ &+ (2\log_2 P)(\frac{n^2}{P}t_c + \tau) \end{aligned}$$

$$T^{VR} = \frac{3}{4}\frac{n^2}{P}\log_2 n + (2\log_2 P)(\frac{3}{4}\frac{n^2}{P}t_c + 3\tau)$$

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.

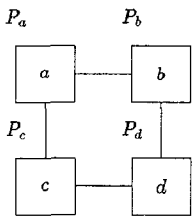
6 Improvements to the Vector-Radix Distributed Step

Consider the 2×2 Vector-Radix butterfly

$$\begin{bmatrix} a' & b' \\ c' & d' \end{bmatrix} \leftarrow \begin{bmatrix} 1 & \omega_1 \\ 1 & -\omega_1 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 1 & 1 \\ \omega_2 & -\omega_2 \end{bmatrix}$$

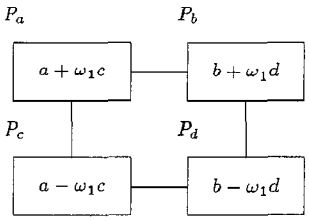
$$= \begin{bmatrix} a + \omega_1 c + \omega_2 b + (\omega_2\omega_1)d & a + \omega_1 c - \omega_2 b - (\omega_2\omega_1)d \\ a - \omega_1 c + \omega_2 b - (\omega_2\omega_1)d & a - \omega_1 c - \omega_2 b + (\omega_2\omega_1)d \end{bmatrix}$$

Suppose that the the elements of this 2×2 matrix are distributed among four processors:



Now if we implement this operation by a total exchange, each processor would have to compute 3 multiplications and 3 additions. The multipliers ω_1, ω_2 and $(\omega_1\omega_2)$ may also be computed redundantly. Other alternatives include precomputing $\omega_1 c$, $\omega_2 b$ and $(\omega_2\omega_1)d$ before the total exchange. This cuts down on the number of multiplications as each processor now does at most one multiplication. We write $(\omega_1\omega_2)$ together because if $\omega_1 = \omega_n^k$ and $\omega_2 = \omega_n^l$, $(\omega_1\omega_2) = \omega_n^{k+l}$ and no multiplication is needed here. The load balancing here is not good because the processor P_a does not have to do any multiplication. However each processor still does three additions.

Another approach is to perform the 2×2 Vector-Radix butterfly in two stages. In the first stage, processors P_a and P_c work together to compute $a + \omega_1 c$ and $a - \omega_1 c$ whereas processors P_b and P_d do $b + \omega_1 d$ and $b - \omega_1 d$. After this stage we have:



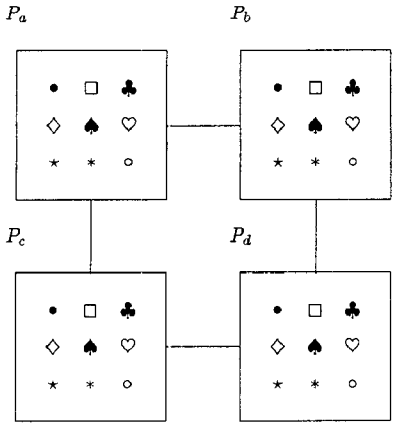


Figure 1: Vector-Radix 2 × 2 Butterflies

Communication is between neighboring processors. One multiplication and one addition is done. Next processors P_a and P_b cooperate to compute $(a+\omega_1 c)+\omega_2(b+\omega_1 d)$ and $(a+\omega_1 c)-\omega_2(b+\omega_1 d)$. Correspondingly processor P_c and P_d compute $(a-\omega_1 c)+\omega_2(b-\omega_1 d)$ and $(a-\omega_1 c)-\omega_2(b-\omega_1 d)$ respectively. In this stage only one addition and one multiplication is done. Again communication is between neighboring processors. And the total number of multiplications and additions is at most two per processor.

In general each processor contains a matrix of points and each 2 × 2 butterfly takes place independently from the others. Figure 1 shows the situation schematically. If communication can proceed along both edges of each node simultaneously we might be able to overlap the split stages of the 2 × 2 butterfly, i.e. while P_a and P_c are computing $a^{\diamondsuit}+\omega_1^{\diamondsuit}c^{\diamondsuit}$ and $a^{\diamondsuit}-\omega_1^{\diamondsuit}c^{\diamondsuit}$, P_a and P_b are computing $(a^{\bullet}-\omega_1^{\bullet}c^{\bullet})+\omega_2^{\bullet}(b^{\bullet}-\omega_1^{\bullet}d^{\bullet})$ and $(a^{\bullet}-\omega_1^{\bullet}c^{\bullet})-\omega_2^{\bullet}(b^{\bullet}-\omega_1^{\bullet}d^{\bullet})$ from the previous butterfly.

Another variation includes P_a and P_b computing $a+\omega_2 b$ and $a-\omega_2 b$ for one set of points and P_a and P_c computing $a+\omega_1 c$ and $a-\omega_1 c$ for another set of points, simultaneously communicating in two directions.

Clearly many more variations can be presented from computing the basic 2 × 2 Vector-Radix butterfly to organizing the order or concurrency of the independent 2 × 2 Vector-Radix butterflies. Each would be best suited for a certain communication scheme, whether fine-grained or coarse-grained (as in the Intel iPSC). Also, buffering needs differ for the various schemes. Therefore the implementation of vector-radix methods on concurrent processors need to be tailored to the particular architecture available and offers potential for both high parallelism and reduced operations counts.

7 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 that interleave computation and communication.
- The complexity of communication for distributed methods is $\log_2 P$, therefore they are promising on systems where P is large.
- In the presence of vector processor nodes, the Block method exhibits better vectorization 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.
- The Vector-Radix method is highly parallel and also requires 25% fewer complex multiplications than the row-column approaches. Since each distributed butterfly depends on data in a d-dimensional sub-cube for a d-dimensional Vector-Radix FFT, a good implementation of the total exchange communication is needed.
- Future generations of hypercubes will likely support more efficient transpose algorithms and routing hardware, making the Transpose-Split methodology more efficient.
- 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 row-column methods. Efficient implementation of the Vector-Radix method depends on efficient total exchange communication and is therefore this method is promising given its faster computational potential. 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.

Our discussion of higher-dimensional transform methods naturally leads to insights on further areas of research. A higher-dimensional DFT can be approached from any combination of the following: row-column, transposing along any dimension, or distributed block manipulations. For example, one way of looking at a 3-dimensional transform is as a 2-dimensional transform and a 1-dimensional transform mapped in planes within nodes of a linear array of processors. Another way of viewing it is to map the data into solid chunks within a three-dimensional array of processors. Finally, the use of the vector-radix approach involves no transposing and exhibits maximal parallelism during the independent stages. Transposing distributed data in three dimensions is extremely awkward, hence we are especially looking at vector-radix approaches to 3-dimensional FFTs as an avenue for further work.

8 Acknowledgements

The author appreciates the helpfulness of Ching-Tien Ho, Yale University for sending his Transpose subroutine. The work that produced this paper was 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. Part of the typesetting and graphics was completed with the facilities of Northrop Research and Technology Center, Palos Verdes Peninsula, California.

9 References

[1] **J. Gustafson** (Mar. 22–25, 1987), "Ensemble FFT's as a Function of Compute/Communication Ratios," in *Fast Fourier Transforms for Vector and Parallel Computers Workshop*, Charles F. Van Loan, ed., The Mathematical Sciences Institute, Cornell University, Ithaca, NY.

[2] **D. B. Harris, J. H. McClellan, D. S. K. Chan & H. W. Schuessler** (1977), "Vector Radix Fast Fourier Transform," *Rec. 1977 IEEE Internat. Conf. Acoust., Speech, Signal Proc.*.

[3] **C-T. Ho & S. L. Johnsson** (1986), "Matrix Transposition on Boolean n-cube Configured Ensemble Architectures," Yale University, Department of Computer Sciences, YALEU/DCS/TR-494, New Haven, CT.

[4] **R. Hockney & C. Jesshope** (1981), in *Parallel Computers: Architecture, Programming and Algorithms,* Adam Hilger, Bristol.

[5] **O. A. McBryan & E. F. Van de Velde** (1987), "Hypercube Algorithms and Implementations," *SIAM J. Sci. Statist. Comput.* v. 8, s277–s287.

[6] **D. Miles, P. Kinney, J. Groshong & R. Fazzari** (1987), " Specification and Performance Analysis of Six Benchmark Programs for the FPS T-Series," Floating Point Systems, Inc., P.O. Box 23489, Portland, OR.

[7] **H. J. Nussbaumer** (1982), *Fast Fourier Transform and Convolution Algorithms,* Springer-Verlag, Berlin, Heidelberg.

[8] **W. H. Press, B. P. Flannery, S. A. Teukolsky & W. T. Vetterling** (1986), *Numerical Recipes: The Art of Scientific Computeing,* Cambridge University Press, Cambridge.

[9] **G. K. Rivard** (1977), "Direct Fast Fourier Transform of Bivariate Functions," *IEEE Trans. Acoust. Speech Signal Process.* ASSP-25, 250–52.

[10] **P. N. Swarztrauber** (1986), "Multiprocessor FFTs," National Center for Atmospheric Research, Boulder, CO, (to appear in *Parallel Computing*).

[11] **C. F. Van Loan** (1987), *FFTs From the Matrix Point of View,* (manuscript).

[12] **S. R. Walton** (1986), "Fast Fourier Transforms on the Hypercube," Ametek Computer Research Division, Arcadia, CA.