Optimizing a Parallel Fast Fourier Transform

by

Richard Hu

submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

June 2004

Author_____
Department of Electrical Engineering and Computer Science
June 21, 2004

Certified by_____
Alan Edelman
Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Graduate Theses

Optimizing a Parallel Fast Fourier Transform

by

Richard Hu

Submitted to the
Department of Electrical Engineering and Computer Science

June 2004

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Parallel computing, especially cluster computing has become more popular and more powerful in recent years. Star-P is a means of harnessing that power by eliminating the difficulties in parallelizing code and by providing the user with a familiar and intuitive interface. This paper presents methods to create a parallel FFT module for Star-P. We find that because calculating a parallel FFT is more communication-intensive than processor-intensive, clever planning and distribution of data is needed to achieve speed-up in a parallel environment.

Thesis Supervisor: Alan Edelman
Title: Professor of Applied Mathematics, Department of Mathematics

Acknowledgements

I would like to thank Professor Alan Edelman for giving me the opportunity to work on such an interesting topic as well as for providing with me much needed support and guidance.

I would also like to thank the members of the Star-P group: Ron Choy, David Cheng, and Viral Shah. I would especially like to thanks Ron Choy would to this day, still knows what I did wrong before I do.

Lastly, I would like to thank the people who have helped keep me sane these last five years: my parents, Freddy Funes, Johnny Yang, and Julia Zhu. Without your ears and your advice, this paper would not have been possible.

Table of Contents

Table of Figures

1. Theory

1.1. Discrete Fourier Transform

1.1.1. One-Dimensional DFT

The one-dimensional Discrete Fourier Transform is a matrix-vector product between a matrix, F_N, the Fourier matrix, and a vector. The DFT can be expressed in the following equations:

$y = F_N x$ **[1],**

where $F_N = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)^2} \end{bmatrix}$ **[2],**

and $\omega = e^{-\frac{2\pi i}{N}}$ **[3].**

We can also express the DFT as a summation:

$y[k] = \sum_{j=0}^{N-1} x[j]\omega_N^{jk}$ where k = 0, 1, …, N-1 **[4].**

1.1.2 Inverse DFT

The inverse Discrete Fourier Transform is a matrix-vector product between the vector and the Fourier matrix.

Expressed as a summation, it is:

$y[k] = \sum_{j=0}^{N-1} x[j]\omega_N^{-jk}$ **[5].**

Often times, the inverse DFT is scaled down by a factor of N. Therefore, we can say that:

$Nx = F^{-1}(Fx)$ **[6].**

1.1.3 Convolution Theorem

The DFT and the inverse DFT can be combined to aid in computing convolutions. Let (f * g) represent the convolution of two discrete signals, f and g. If we let F represent the Fourier Transform operator, then (F f) represents the DFT of the signal f and (F g) represents the DFT of the signal g. The convolution theorem states that:

$$F(f * g) = (Ff) \cdot (Fg) \quad \textbf{[7]},$$

where the dot denotes point-wise multiplication. Another form of the theorem that will prove invaluable later is the following:

$$(f * g) = F^{-1}((Ff) \cdot (Fg)) \quad \textbf{[8]},$$

which states that the convolution of two signals is equal the inverse Fourier transform of the point-wise multiplication of the Fourier transform of each of the two individual signals.

1.1.4 Two-Dimensional DFT

The two-dimensional Discrete Fourier Transform is a matrix-matrix product between the Fourier matrix and the desired matrix in the following form:

$$y = F_N x F_N^T \quad \textbf{[9]},$$

which can also be expressed as:

$$y = (F_N (F_N x)^T)^T \quad \textbf{[10]}.$$

The above equation states that the two-dimensional FFT is a series of smaller steps as follows: a one-dimensional FFT on the matrix, a transpose of the resulting matrix, another one-dimensional FFT on the matrix, and finally another transpose.

1.1.5 Properties of the Discrete Fourier Transform

When the input to the DFT is a matrix that contains only real numbers, then we can apply the following property to the DFT:

$y[k] = y^*[N-k]$ **[11].**

This property will split the calculation of the DFT of a real vector in half **[2].**

Another useful property of the DFT is that for a matrix where the elements are complex, then we can apply the following property of the DFT:

$F(A + Bi) = FA + i * FB$ **[12].**

Therefore, a DFT on a matrix with complex elements can be broken down into two more simple DFTs on matrices with real elements.

1.1.6 Relation between the DFT and the FFT

The Fast Fourier Transform is identical to the Discrete Fourier Transform. The Fast Fourier Transform is a term that was coined to describe the set of algorithms use to increase the speed of calculating the DFT from being $O(N^2)$ as would be expected from a matrix-vector product to being $O(N \log_2 N)$. Another difference between the DFT and the FFT is connotation in the size of the problem investigated. Often times, the DFT is associated with either infinite or an arbitrary size while the FFT is most often associated with sizes that are equal to a power of 2. In fact, most benchmarks for FFTs will only be expressed as the speed of the FFT versus a matrix of size equal to a power of 2. The term FFT and DFT can otherwise be used interchangeably.

1.2 Cooley-Tukey Method

1.2.1 Radix-2 Decimation in Time FFT

Calculating a DFT purely by the mathematical definition given above would take $O(n^2)$ time. However, in 1965, Cooley and Tukey independently came up with a solution that was actually first proposed by Gauss even before Fourier proposed his original paper on the Fourier series. The Cooley-Tukey method is an example of a divide-and-conquer algorithm. A divide-and-conquer algorithm recursively attempts to solve a problem by dividing the original problem into two or more problems of smaller size. The algorithm then attempts to solve each smaller problem by further dividing those problems into even smaller problems until some boundary condition is met. The final solution is obtained by combining the solutions to each of the smaller problems that spawned from the original.

From equation **[4]**, we can express one term of the DFT as such:

$$y[k] = \sum_{j=0}^{N-1} x[j]\omega_N^{jk} \quad .$$

First, we shall evaluate the Cooley-Tukey algorithm in its simplest form, the radix-2 FFT. Initially, we re-write the definition of the DFT into the smallest summations of its even terms and its odd terms as follows:

$$y[k] = \sum_{j=0}^{\frac{N}{2}-1} x[2j]\omega_N^{2jk} + \omega_N^k \sum_{j=0}^{\frac{N}{2}-1} x[2j+1]\omega_N^{2jk} \quad \textbf{[13].}$$

Using the identity:

$$\omega_{\frac{N}{2}} = \omega_N^2 \quad \textbf{[14]},$$

we can re-write the above equation as:

$$y[k] = \sum_{j=0}^{\frac{N}{2}-1} x[2j]\omega_{\frac{N}{2}}^{jk} + \omega_N^k \sum_{j=0}^{\frac{N}{2}-1} x[2j+1]\omega_{\frac{N}{2}}^{jk} \quad \textbf{[15]}.$$

As we can see from equation **[15]**, it appears that each summation becomes a smaller DFT of size N/2. The first DFT is over the even-indexed elements and the second DFT is over the odd-indexed elements. Because we are splitting the initial data set, this approach is called decimation in time. In addition, by applying another identity, we can easily find another element of the final DFT.

We can also apply two more identities to easily find another point in the DFT.

$$\omega_N^{\frac{N}{2}+k} = -\omega_N^k \quad \text{and} \quad \omega_{\frac{N}{2}}^{\frac{N}{2}} = 1 \quad \textbf{[16]}$$

help create the next equation:

$$y[k+\frac{N}{2}] = \sum_{j=0}^{\frac{N}{2}-1} x[2j]\omega_{\frac{N}{2}}^{jk} - \omega_N^k \sum_{j=0}^{\frac{N}{2}-1} x[2j+1]\omega_{\frac{N}{2}}^{jk} \quad \textbf{[17]}.$$

Equation **[17]** allows us to calculate two points in the DFT by breaking up the DFT into two smaller DFTs of size N/2. If the smaller DFTs of size N/2 are still divisible by 2, then we can apply the radix-2 algorithm again to create even smaller DFTs of size N/4. Assuming the smaller DFTs are still divisible by 2, then we can keep applying the radix-2 algorithm until either the

DFT becomes so small that we can calculate it directly or until it becomes a one-point DFT, in which case, the result is just that element.

Originally, the DFT takes $O(N^2)$ time to calculate because it is a matrix-vector product. However, with the radix-2 algorithm, the computation is reduced to $O(N \log_2 N)$ because of the divide-and-conquer strategy employed.

1.2.2 Radix-2 Decimation in Frequency FFT

Another radix-2 algorithm to calculate a FFT is to employ decimation in frequency. Instead of dividing the input x into an even-indexed set and an odd-indexed set, the decimation in frequency (DIF) FFT divides the output y, into an even-indexed set y_{2j} and an odd-indexed set y_{2j}. The algorithm is otherwise very similar to the decimation in time (DIT) FFT **[1]**.

1.2.3 Bit-Reversal

Because both the DIT and DIF FFT divide the FFT into an even-indexed set and an odd-indexed set, either the input or the output will be in a scrambled order. For the DIT FFT, it will be output that will be in a scrambled order while for the DIF, it is the input that is in a scrambled order. One method for restoring the natural order of both the input and output is bit-reversal. If we express the index of the element that we wish to de-scramble in binary, then the proper place for the element is the reverse of that index as expressed in binary **[1]**.

1.2.4 Generalized Cooley-Tukey Algorithm

Unfortunately, there are many times where the size of the FFT will not be a power of 2. In fact, there might even be times

when the size of the FFT will not be a multiple of 2. However, if the size N of the FFT is still a composite number, then we can use the generalized Cooley-Tukey method to break the FFT down into smaller sizes.

Again, we start with the definition of the DFT of size N as defined by equation **[4]**

$$y[k] = \sum_{j=0}^{N-1} x[j]\omega_N^{jk} \ .$$

If N is composite, then we can express N as the product of two numbers p and q. We can then express the indices j and k from the definition of a DFT as:

$j = j_1 q + j_2$ **[18], and**

$k = k_1 + k_2 p$ **[19]**

where j_1 and k_2 range from 0 to p-1 and j_2 and k_1 range from 0 to q-1. After substituting these values into the definition of the DFT and then applying simple exponential identities, we achieve:

$$y[k_1 + k_2 p] = \sum_{j_2=0}^{q-1}[(\sum_{j_1=0}^{p-1} x[j_1 q + j_2]w_p^{j_1 k_1})w_N^{j_2 k_1}]w_q^{j_2 k_2} \quad \textbf{[20].}$$

The inner summation, $\sum_{j_1=0}^{p-1} x[j_1 q + j_2]w_p^{j_1 k_1}$, computes q DFTs of size p. It then multiplies it by the result by $w_N^{j_2 k_1}$, which is also known as a twiddle factor and then finally calculates p DFTs of size q. If each of p and q are composite as well, we can apply the Cooley-Tukey algorithm again to achieve smaller DFTs until either p and q are small enough to calculate directly or p and q are prime numbers in which case, we can use either Rader's method or

Bluestein's algorithm to calculate the DFT. Like the radix-2 method, the Cooley-Tukey method enjoys a speed-up in performance from $O(N^2)$ time from the definition of the DFT to $O(N \log_2 N)$ time.

1.3 Rader's Method

Another method for calculating the FFT of size N, where N is a prime number is Rader's method. However, unlike Bluestein's method, Rader's method only works when N is prime.

When N is prime, there exists a number g, which is a generator for the number k, a multiplicative group modulo N that contains the elements from 1 to N-1. Notice that k does not contain the element zero. We can express the above as follows: $k = g^q \bmod n$, where k = 1, 2, …, n-1 and q is unique for each k and is within the set {0, 1, …, n-2}. Similarly, $j = g^{-p} \bmod n$, where j is a non-zero index and p is within the set {0, 1, …, n-2} as well. Therefore, we can re-write the DFT using p and q as follows:

$$y[g^{-p}] = x[0] + \sum_{q=0}^{N-2} x[g^q]\omega^{g^{-(p-q)}}$$ where p = 0, …, n-2. **[21].**

The final summation is a cyclic convolution between two sequences a_q and b_q of length n-1, since q ranges from 0 to n-2, where a_q and b_q are defined by:

$$a[q] = x[g^q]$$ **[22], and**

$$b[q] = \omega^{g^{-q}}$$ **[23]**

We can apply the convolution theorem to the summation to turn it into an inverse FFT of a point-wise multiplication of the two FFTs, one of a and the other of b. The FFTs of a and b will be of size N-1 and since N is prime and N-1 is composite, we can apply the Cooley-Tukey algorithm to compute the FFT of a and b, as well as the inverse FFT. Alternatively, the convolution can be padded with zeroes to a length from 2(n-1) - 1 to a power of 2 and then a radix-2 algorithm can be used.

1.4 Bluestein's Method

Bluestein's method of calculating FFTs can actually be applied to FFTs of any size N. However, it has been found that it is more efficient to use the Cooley-Tukey method to calculate FFTs of composite N's. Therefore, the use of Bluestein has been somewhat relegated to calculating the FFT of prime N's.

First, let us revisit the definition of the DFT from equation **[4]**:

$$y[k] = \sum_{j=0}^{N-1} x[j]\omega_N^{jk} \quad .$$

If we use the simple algebraic equality:

$$jk = \frac{-(j-k)^2 + j^2 + k^2}{2} \quad \textbf{[24].}$$

Then we obtain

$$y[k] = \omega_N^{\frac{k^2}{2}} \sum_{j=0}^{N-1} (x[j]\omega_N^{\frac{j^2}{2}})\omega_N^{\frac{-(j-k)^2}{2}} \quad \textbf{[25].}$$

This summation becomes a linear convolution between two sequences a and b of length N where:

$a[k] = x[k]\omega_N^{\frac{k^2}{2}}$ **[26], and**

$b[k] = \omega_N^{\frac{-k^2}{2}}$ **[27],**

where the output of the convolution is multiplied by a factor of b*[k]. Therefore, the convolution becomes

$y[k] = b^*[k]\sum_{j=0}^{N-1} a[j]b[k-j]$ **[28].**

As stated before, from the convolution theorem, the convolution becomes the inverse FFT of the point-wise product of the FFT of a and b. However, the key is that the length of these new FFTs do not have to be N. Because y has been expressed as a convolution, the input vectors a and b can be zero-padded to any length greater than 2N-1. Therefore, we can zero-pad the convolution to a power of two and then apply the radix-2 algorithm.

There are two other ways of expressing Bluestein's method. The first way is to think of the convolution as first running the input through a chirp transform, which is based on an analog signal filter **[3]**. The other way to think of the convolution is to think of expressing the DFT as a symmetric Toeplitz matrix-vector product where the symmetric Toeplitz matrix is any matrix where its elements can be expressed as follows **[1]**:

$t_{i,j} = h_{i-j}$ and $h_{i-j} = h_{j-i}$ **[29].**

The algebraic equality given in equation **[24]** accomplishes this.

2. Parallel Computing

2.1 Star-P

In recent years, there have been great advances in the field of parallel computing. These advances, in addition to the decrease in the cost of hardware, have caused cluster computing to boom. Many companies and labs are turning to Beowulf clusters instead of traditional supercomputers to handle their computationally intensive needs. Beowulf clusters have the advantage over traditional supercomputers in that they are cheaper to create. This cost-efficiency is balanced by the fact that the clusters, even the ones with high-end interconnects, have much higher communication costs than shared-memory supercomputers. Because of this, parallel computing on Beowulf clusters must take into account the cost of communication between the nodes more keenly than traditional supercomputers.

Another problem with supercomputing is that the cost of creating a parallel program is still extremely high. An algorithm that is optimized for computational cost on a serial platform will not necessarily port well to a distributed computer. Star-P is a solution to that. Star-P provides a Matlab front-end to the user which is simple and intuitive to learn, but also provides a distributed back-end which improves performance over serial computers. Matlab was chosen as a front-end because of its widespread popularity among engineering and scientific communities. By providing a familiar front-end, Star-P reduces the cost of creating parallel programs **[12]**.

The back-end, on the other hand, demonstrates the cost of creating parallel programs. It is written in a mixture of C, Java, and Fortran with the Message Passing Interface (MPI) library as its primary means of communication between nodes. MPI is a popular library that works over TCP/IP connections as well as with shared-memory architectures and consists of six basic functions. One function signifies the start of a parallel program while another signifies the end of the parallel program. Two functions provide information about the parallel environment such as the number of processors and the rank of each processor. The last two functions are a simple send and receive. The send function is a non-blocking send, which is called by the sender and includes as its arguments, the data, the type of the data, and lastly, the recipient. The receive function is a blocking receive, which is called by the target and must include the data, the type of the data and also the sender.

Each function that is normally available in Matlab is re-implemented in Star-P with what is considered the most optimal parallel algorithm. Therefore, each function acts as a black box, receiving the same arguments and outputting the same result as the serial version, which makes implementing new functions through mex files identical to implementing new functions in Matlab.

2.2 Distributions

Star-P uses three different kinds of distributions. These distributions are the ones as specified by the parallel linear

algebra package Scalapack. Each of these three distributions has its strengths and weaknesses depending on the desired function and the user can control which distribution is used. Since each function in Star-P is designed to work with each of the three distributions, the user can chose which distribution is utilized arbitrarily. However, this freedom also places a slight burden on the user in that the user may choose a non-optimal distribution.

The first distribution is row distribution. In this format, the matrix is divided, as evenly as possible, by its rows onto each of the processors. For example, with N rows and P processors, the first processor will receive the first N/P rows, rounded up to the nearest integer. The next processor will receive the next N/P rows and so on. If there are not enough rows such that each processor gets at least one row, then the last few processors will not receive any data and there will be wasted processors.

The second distribution is column distribution. This format is identical to the row distribution except that instead of contiguous rows being on each processor, contiguous columns are on each processor. As in row distribution, the columns will be divided as evenly over the processors as possible.

The last distribution is the block-cyclic distribution. This format is the trickiest because it is also the most flexible. When Scalapack is installed, it allows the user to either chose the size of each block or to allow Scalapack to

dynamically chose the size of the blocks. The processors are broken up into two groups. The first group contains the first P/2 processors, rounded up. The second group contains the next P/2 processors rounded down. The data is then broken into blocks, using the sizes either specified by the user or the Scalapack library. In a row major fashion, the blocks are distributed cyclically across the first group of processors until there are no blocks that can be distributed. The next set of rows is distributed cyclically across the second group of processors. The third set of rows is distributed across the first group of processors and each successive set of rows is distributed across a group of processors. For example, if the size of the block ix MX by MY then the first MX rows are cyclically column distributed across the first P/2 processors. The next MX rows are cyclically column distributed across the second P/2 processors and so on. This format is demonstrated by the following diagram of a matrix distributed over 4 processors **[11]**.

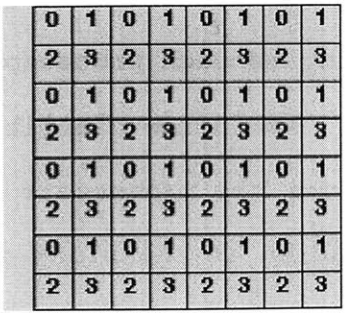

FIGURE 1
Block-Cyclic Distribution Over Four Processors

2.3 FFTW

Fastest Fourier Transform in the West, FFTW, is a C sub-routine for computing Discrete Fourier Transforms in one or more dimensions. The program was developed at MIT by Matteo Frigo and Steven Johnson and is extremely competitive in terms of speed with even manufacturer's hand-coded FFT functions. In addition, FFTW is very portable from one architecture to another. Initially FFTW was coded as a serial program. However, a parallel version was added in FFTW 2.1.5, which is the version that Star-P utilizes. The parallel version attempts to make the calculations for FFTW as embarrassingly parallel as possible by fitting smaller FFTs on each processor and then combining them as necessary.

The key to the speed behind the FFTW package is the FFTW planner **[2]**. Instead of trying to figure out the best method for calculating a FFT, FFTW allows the computer to try out different methods in order to come up with an optimal solution for a FFT of a given size. The final and quickest method is stored in a plan that can be reused for any FFT of that size. Since scientists and engineers often calculate many FFTs of the same size repeatedly, re-using the plan leads to a vast savings in time that eventually outweighs the cost of calculating the plan.

Star-P as well as Matlab use FFTW as its means of calculating serial FFTs. However, in both programs, each FFT call is made individually so it is not efficient to call the planner each time and there is no good method for caching the

plan between calls. Therefore, FFTW also includes another option where the processor guesses the best method for calculating FFT based solely on the size of the problem. While this option does not always provide the most optimal plan, the plan is created almost instantaneously.

In this project, FFTW was utilized as the primary method of calculating FFTs because of its speed and portability, in addition to the fact that FFTW already has quite a bit of code that takes advantage of different hardware architectures, a feat that would take quite a while to replicate.

3. Methodology

In this section, we will investigate the methodology behind evaluating a parallel transform. In particular, we will build from the easiest situation to parallelize to the most difficult in its application with Star-P. As will be seen from this section, each situation builds on the one from before.

3.1 Column Distributed One-Dimensional Fast Fourier Transform

We shall first begin with the most basic and trivial case, a column distributed one-dimensional matrix. As discussed in a previous section, the column-distributed matrix has a set of contiguous columns on each processor. Since the one-dimensional FFT is an independent operation on each column, each processor can perform the FFT over its set of data without any inter-processor communication. This solution would be described as 'embarrassingly parallel.'

If the data set is small enough to fit in the resident memory of each individual processor and each processor has approximately the same number of elements, then we should experience almost linear speed-up as the number of processors increase. In the case that the data set is too large for the resident memory of each processor and has to go into an alternate means of storing the data set, then the distributed version should experience super-linear speed-up. However, if the cluster is not homogeneous or each processor does not contain

approximately the same amount of data, then the FFT will only speed-up relative to the speed of the slowest processor.

In application, Star-P utilizes Scalapack, which distributes the columns as evenly as possible so at least linear speed-up should be seen. However, Star-P also anticipates that the data is in column-major format as opposed to FFTW, which expects the data to be in row-major format. This obstacle is overcome without performance degradation because FFTW has been optimized for strided data. We merely set the stride to the entire length of the DFT to be computed.

3.2 Column Distributed Two-Dimensional Fast Fourier Transform

As seen before, we can express the two-dimensional FFT as a matrix-matrix multiplication, as opposed to the matrix-vector multiplication that categorizes the one-dimensional FFT. Parallel to the one-dimensional FFT, this matrix-matrix multiplication can be expressed as a summation as follows:

As can be seen from the equation above, the two-dimensional FFT is nothing more than a set of nested one-dimensional FFTs. In fact, the exact order of operations for a two-dimensional FFT is as follows: one-dimensional FFT along the columns, a matrix transpose, then another one-dimensional FFT along the columns of the transpose matrix, and finally another transpose to return the matrix to its original form. This technique lends itself well to being parallelized. In order to calculate the two-dimensional FFT on a column-distributed matrix, we shall compute a one-dimensional FFT along each column, take the transpose of the

result, computer the one-dimensional FFT along each column again, and then take the transpose again.

From the section on the one-dimensional column distributed FFT, each of the one-dimensional FFTs will be embarrassingly parallel and therefore have at least linear speed-up. However, there is inter-processor communication and it comes in the form of the matrix transpose. In essence, a matrix transpose on a distributed machine is like converting a column-distributed matrix into a row-distributed matrix and switching the matrix from row-major to column-major. From before, switching the matrix from row-major to column-major the serial FFTW program is not a significant obstacle. On the other hand, converting the column-distributed matrix into row-distributed matrix requires a considerable amount of communication between processors.

In a best-case scenario in a N x N matrix with P processors, each processor will have to send 1/P of its data to each of the other P-1 processors when converting from column-distributed to row-distributed and back. In a shared-memory architecture, this communication is not significant as since no two processors will be attempting to access the same data location at the same time. However, in a distributed memory architecture, the communication must occur under several constraints in order to achieve optimal performance. First, there must be some sort of load-balancing for communication between processors to ensure that no processor is sitting idling. Second, in order to minimize the latency of communication, the

data should be sent in blocks that are as large as possible. Third, we must assume that each processor can only communicate with one other processor in a send and receive pair and each communication between two processors constitute a round. Last, we want as few rounds as possible. It turns out that there is a theoretical solution to this problem in another form. The other problem is called the "soccer match" problem and has a very well known solution.

Because a serial two-dimensional FFT is normally calculated by taking the one-dimensional FFT across all the columns, taking the transpose, taking the one-dimensional FFT across all the columns again, and then transposing the matrix back to its original form, the parallel version is extremely similar. The parallel version will experience speed-up in a linear or super-linear fashion across each of the one-dimensional FFTs that are calculated. This speed-up is however offset by the amount of communication that must be performed in each of the two transposes. In a serial FFT, no transpose is necessary and because FFTW can handle strided data without significant performance problems, the transposes can essentially be ignored. On the other hand, in the parallel FFT, depending on the interconnect between the processors and the size of the problem, the time spent in the transpose could outweigh the time saved by having multiple processors.

3.3 Row Distributed Two-Dimensional Fast Fourier Transform

The two-dimensional row-distributed FFT is extremely similar to the two-dimensional column-distributed FFT that was discussed in the previous section. However, the primary difference is that since the data is row-distributed, in order to compute the first one-dimensional FFT across the columns in the initial step, we must first transpose the matrix to become column-distributed.

As states in the previous section, the two-dimensional column distributed FFT requires significant communication in performing each of the matrix transposes. The row-distributed two-dimensional FFT requires even more time because it actually requires three matrix transposes. It requires a transpose at the beginning of the FFT to start the first one-dimensional FFT along the columns.

3.4 Row Distributed One-Dimensional Fast Fourier Transform

There are two ways to calculate the one-dimensional row-distributed FFT. One method is change the row distribution into a column distribution using an algorithm similar to that of a matrix transpose and then using the one-dimensional column-distributed FFT to calculate the FFT in an embarrassingly parallel fashion before changing the matrix back to row distribution. This method requires two transpose-like functions and is very similar to the two-dimensional column distributed FFT.

The second method is to take a single column and turn it into a two-dimensional matrix and then by computing the two-

dimensional FFT of that matrix in a manner described above and then converting the matrix back into a one-dimensional column. In order to do this, the length N of the column is first factored into the product of two composites, p and q. We can then apply the Cooley-Tukey algorithm to make the FFT of the column equal to the nested FFTs of size p and q with a twiddle factor included. On a parallel machine, we create a two-dimensional matrix of size p x q that is column distributed across the processors and then we use the two-dimensional column distributed FFT to calculate the FFT of the p x q matrix. The second method involves three matrix transposes, therefore making it similar to the row-distributed two-dimensional FFT.

3.5 Block-Cyclic Fast Fourier Transform

Both the one-dimensional and two-dimensional block-cyclic FFTs are very similar. FFTW currently does not allow for parallel strided FFTs and demands that each processor have a contiguous set of rows. Therefore, in order to compute a FFT on a block-cyclic matrix, we must either convert the matrix into column-distributed or into row-distributed. Because block-cyclic distribution is a row permutation followed by a column permutation, converting to either column or row distribution will require approximately the same amount of time. However, calculating a FFT in column distribution is embarrassingly parallel in one-dimension and requires one less matrix transpose than row distribution in two-dimensions so therefore, the obvious choice would be to convert the FFT to column distribution.

4. Results

The parallel FFTs were all tested on a Beowulf cluster which utilized a Fast Ethernet (100 MB) interconnect. In addition, each node was a dual-processor node with 1 Gigabyte of shared memory. Each test was run multiple times with the results taken being the average. Occasionally, the results of the test would prove to be unstable in either the node would crash occasionally while running the test or the results from the test were too spread apart. These results will not be displayed. In addition, it seemed that maximum size that could be run stably on Beowulf was a problem of size 4096 x 4096. This matrix would consume 134 MBs of space which is well within the boundaries of the hardware involved. A 8192 x 8192 matrix, which usually did not successfully complete would only take four times that amount of space which is still well within the bounds of the Beowulf cluster. This instability was somewhat puzzling.

4.1 Column Distributed One-Dimensional FFT

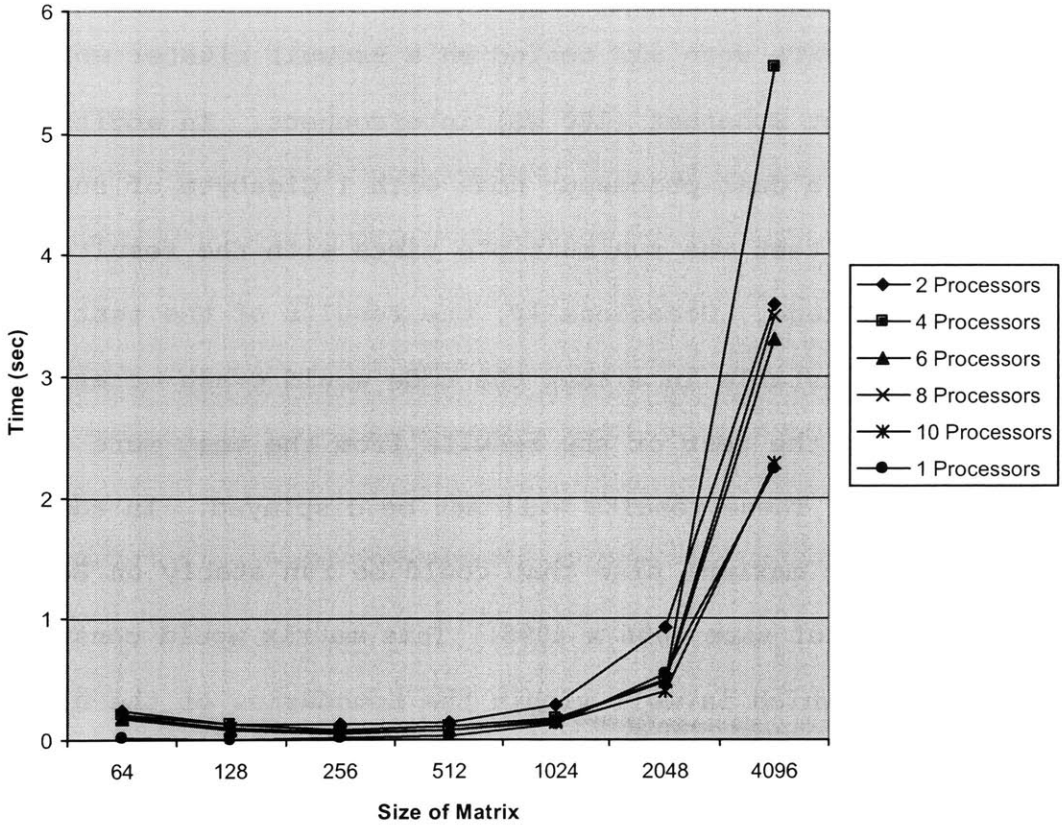

Figure 2
Speed of One-Dimensional Column Distributed FFT

Unfortunately, there was no speed-up observed. In fact, only at ten processors did the parallel version match the speed of the serial version. This behavior would seem to contradict what would be expected to happen, linear to super-linear speed-up. There is no communication between processors so all the loss of speed is coming from some unknown source. There are two possible explanations for this. Because the speed of the serial FFT is already so quick, the gaps in performance could be due to the overhead involved in accessing a parallel matrix. Star-P has its own overhead in dealing with parallel code and parallel matrices.

The second explanation could be the overhead in dealing with the Star-P log. Because Star-P is still in a development stage, the log contains a very detailed list of parallel functions that are called. Serial FFT does not trigger these events in the log.

4.2 Row Distributed One-Dimensional FFT

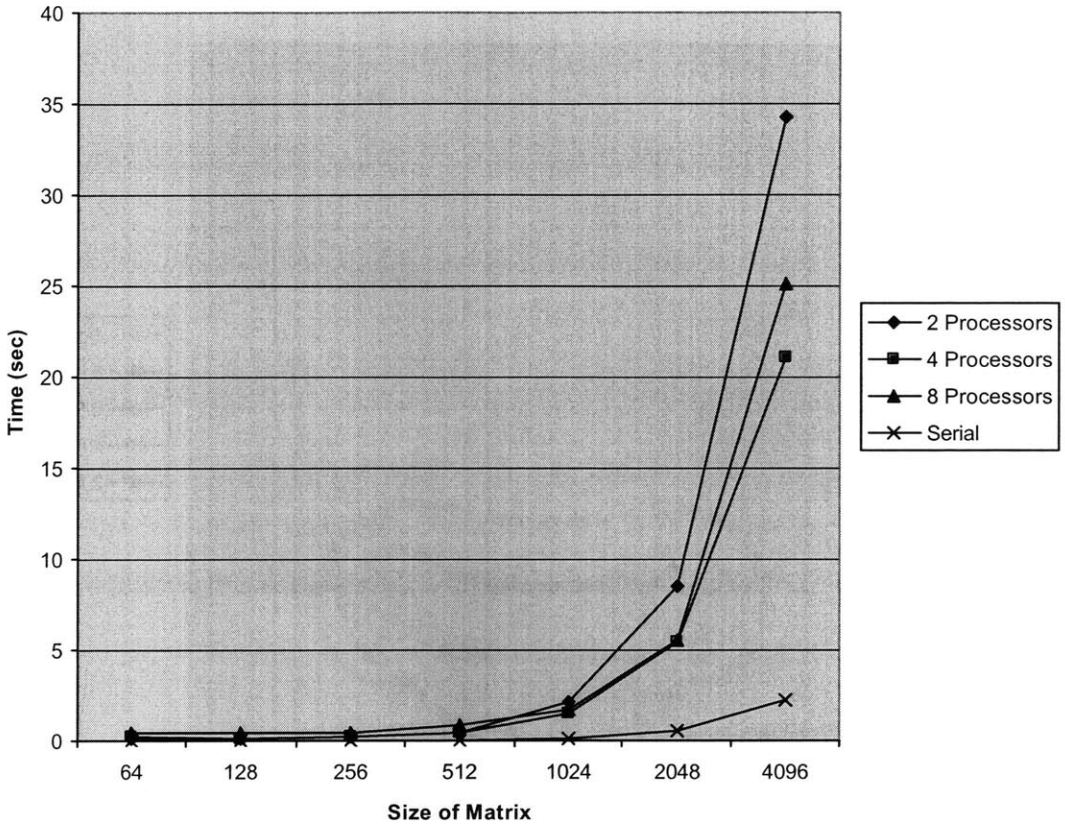

Figure 3
Speed of One-Dimensional Row-Distributed FFT

Again, serial beats parallel and in fact parallel is doing even worse, which was to be expected. Another disturbing trend is that the speed of the problem is not necessarily related to the number of processors in the parallel version. We would expect that eight processors would be faster than four, especially when

the problem size grows as the case was with the column distributed. However, it is clear that the four processor version is much faster than the eight processor version for a 4096 x 4096 matrix.

4.3 Block-Cyclic One-Dimensional FFT

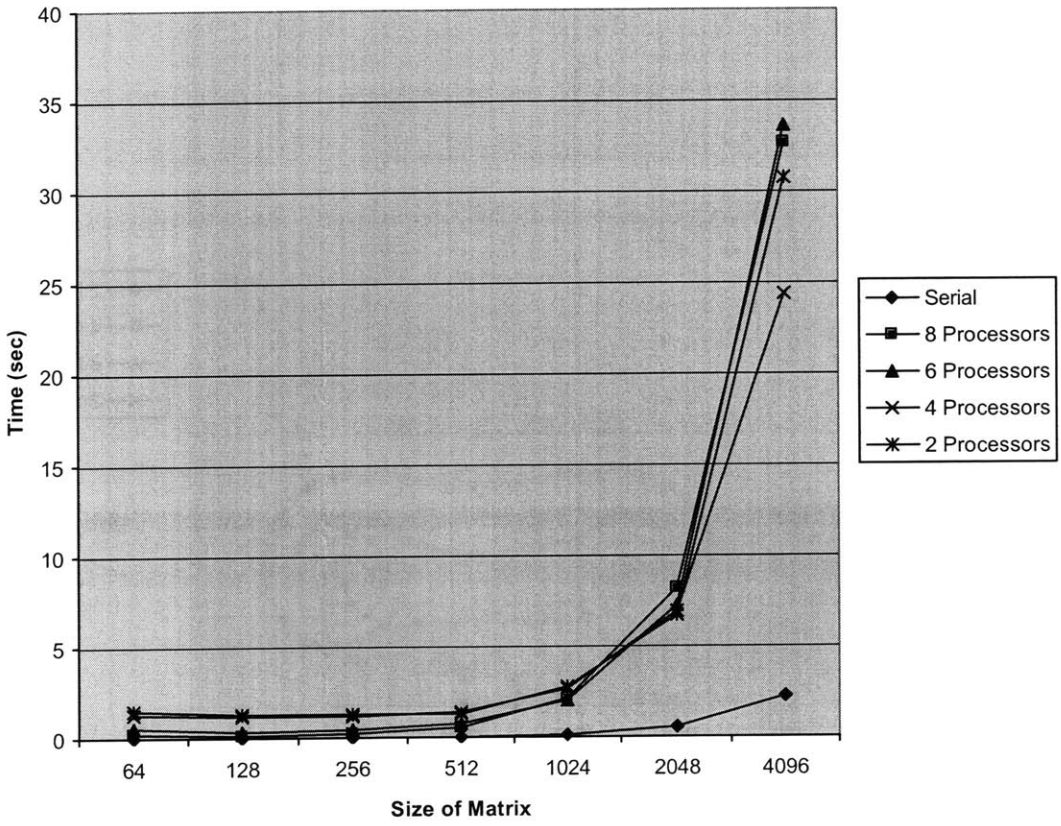

Figure 4
Speed of One-Dimensional Block-Cyclic FFT

The blocks in these examples were of size 64 x 64. Block-Cyclic FFT is only slightly slower than row distribution. It gains the advantage of more embarrassingly steps when actually calculating the FFT but loses a significant amount when dealing with the communication of changing distributions.

4.4 Two-Dimensional FFT

Instead of showing graphs of how poorly a parallel 2D FFT compares to the serial version, I decided to investigate the speed of changing distributions because that seems to be one of the primary slow-downs in the non-column distributed FFT.

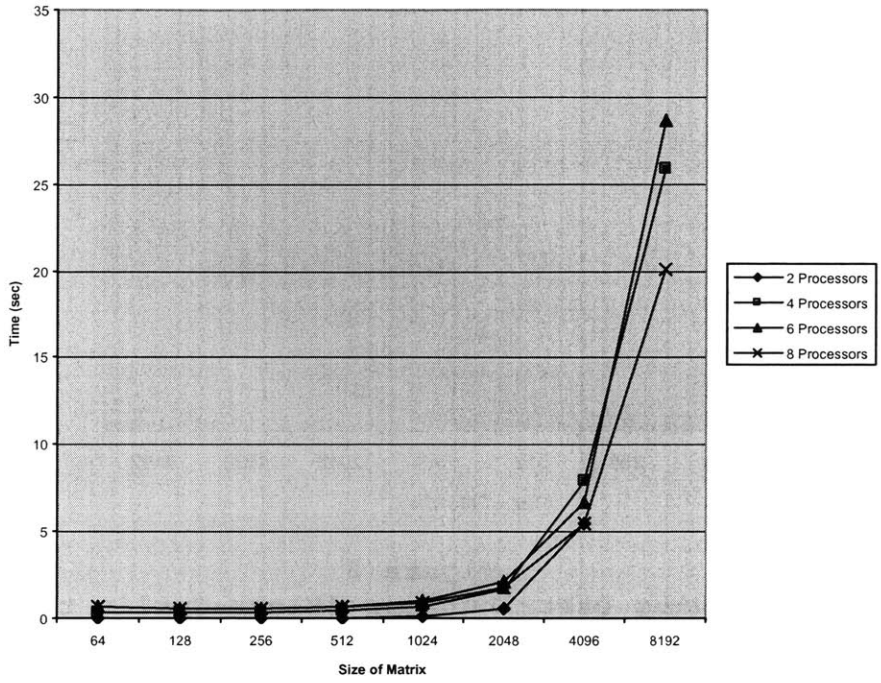

Figure 4
Speed of Changing Distributions (Row to Column)

From the look of the chart, changing distributions exhibits the same exponential behavior demonstrated by the parallel FFT program. This unfortunately means that despite all the best efforts of the parallelization and the efficiency in changing distributions, the limiting factor is the matrix transposes involved. The same behavior is exhibited in the block-cyclic to column distribution.

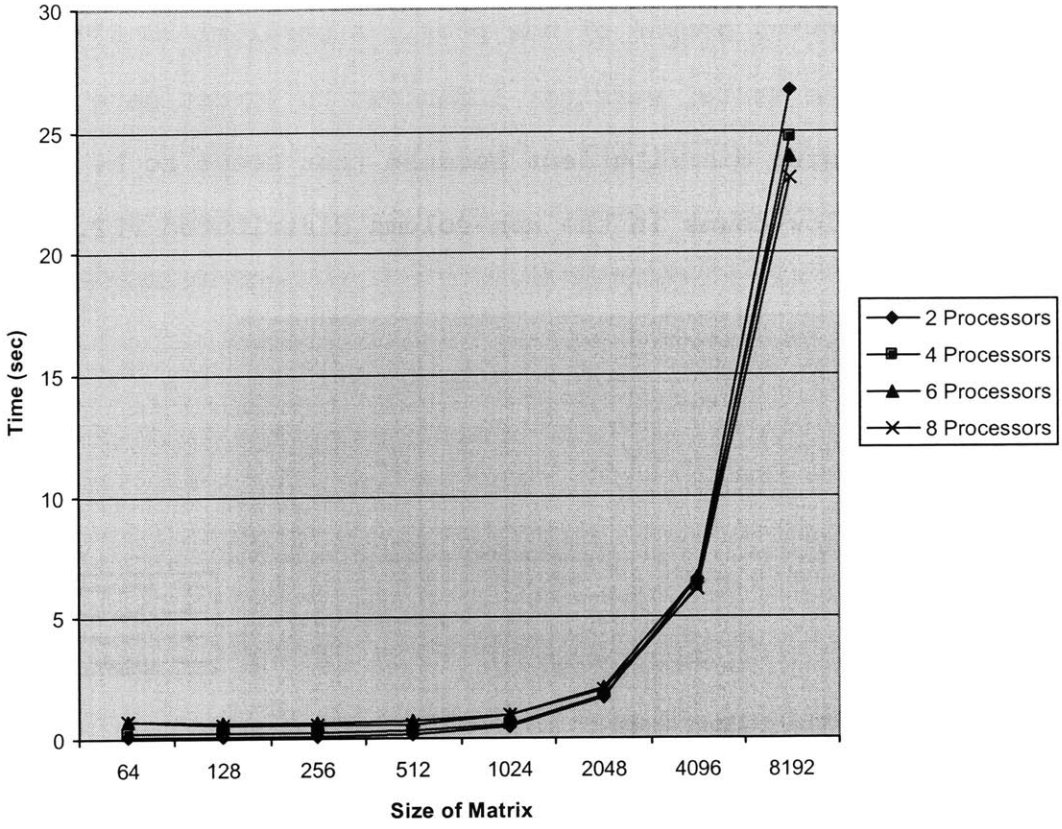

Figure 5
Speed of Changing Distributions (Block-Cyclic to Column)

Again, the same exponential behavior is exhibited.

5. Future Considerations

From the results, it seems that a parallel FFT runs slower than a serial FFT. However, the tests were run on only one iteration of the FFT at a time because of the way that Star-P works and the overhead to the Star-P calls seems to out-weigh any speed-advantage granted from using multiple processors. In real-life applications, many FFTs are often strung together in a sequence. A parallel FFT can take advantage of this property by taking the time to change the distribution into the most optimal format initially, and change the final answer back to the original distribution. This simple shortcut would reduce the time of each FFT and also reduce the overhead of the Matlab function calls.

Ideally, a parallel FFT would adopt an approach more similar to that of the serial FFTW with its planner. The true power of FFTW is the planner because it actually runs part of the FFT in order to determine a re-useable plan for a FFT of that size. Therefore, when a FFT of size N is run multiple times, the plan, which has some fixed cost to calculate, is re-used and the cost of running a FFT with this optimized plan is much less than the cost of running a FFT with the estimate plan. Because of the large overhead involved in communication in a parallel FFT, the ability to perform that communication once, and optimally would greatly reduce the cost of the communication.

For example, if a FFT were being run multiple times on a distributed matrix, it would be better to have the output from one FFT remain bit reversed to be the input into the next FFT.

The program would alternate between using DIF and DIT algorithms to ensure this. This way, there would be no communication necessary to move the elements into the proper places between FFTs which would greatly reduce communication cost.

Another future consideration would be a more flexible FFTW program in MPI. Currently the FFTW program as implemented in MPI dictates to the user which rows have to be on which processors. Furthermore, FFTW assumes that every single processor is being used and each processor holds a contiguous set of data. Lastly, the program does not allow for strided FFTs like it does in the serial version. None of these should be strict constraints. In a two-dimensional FFT, it does not matter which rows are on which processors because the first step is to transpose the matrix.

Most FFTs are tested for their speed on matrices with dimensions that are equal to a power of two, or in the case of FFTW, dimensions that are equal to the product of very small primes. In practical applications, FFTs are used on matrices of any size. However, it is possible for both Rader's algorithm and Bluestein's algorithm for calculating FFTs to be parallelized rather elegantly for different values of N. This parallelization could be the most beneficial speed-up seen in a practical FFT.

For example, in the case that N is a Mersenne prime, then parallelizing Rader's algorithm becomes a simple matter. From Scalapack, each of the processors holds the same amount of data with the first processor holding any extra elements. After applying Rader's algorithm, we then have to compute a series of

FFTs on the a matrix of length N-1, which is also 2^S, where S is an integer. Suddenly, we have a row-distributed matrix of size 2^S, which is a simple matter to compute using a radix-2 algorithm.

Bluestein's algorithm is even more easily parallelized, albeit at the cost of space. Since the point of Bluestein's method is to grow the matrix to a more easily computable size, usually 2^S but with FFTW that is not necessary, a parallel Bluestein can grow the matrix to a size that is optimal and relative to the number of processors with only minimal communication. However, since the speed benefit from Bluestein's algorithm is already balanced by its cost in space, this problem should be minimal and in fact, should play better to the strengths that parallel computing has to offer.

Lastly, in hindsight, attempting to apply a serial program, FFTW that contains minimal parallel support, at least for distributed memory systems, to a distributed system, Star-P, that attempts to abstract away the parallelism, will not yield optimal results. There is no simple way of parallelizing the FFT especially since FFTW achieves its results by combining many different algorithms. Therefore, in order to optimized a parallel FFT, we would have to optimized each of these different algorithms, if not create new ones. As it is, the most efficient way of calculating the FFTs would be to transpose them into column distribution and then calculate the FFTs along the columns in an embarrassingly parallel fashion.

Bibliography

1. Chu, Eleanor and Alan George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms.* USA: CRC Press, 2000.

2. Frigo, Matteo and Steven Johnson. *The Design and Implementation of FFTW.* 2003.

3. Gold, Bernard and Lawrence Rabiner. *Theory and Application of Digital Signal Processing.* New Jersey: Prentice Hall, 1975.

4. Frigo, Matteo and Steven Johnson. "The FFTW Web Page," http://www.fftw.org. 2003.

5. Briggs, William L and Van Emden Henson. *The DFT: An Owner's Manual for the Discrete Fourier Transform. Philadelphia: SIAM, 1995.*

6. Bluestein, L. "A Linear Filtering Approach to the Computation of Discrete Fourier Transform." *IEEE Transactions on Audio and Electroacoustics*, AU-18:451-455, 1970.

7. C. M. Rader, "Discrete Fourier transforms When the Number of Data Samples Is Prime," *Proc. IEEE* 56, 1107-1108, 1968.

8. Cooley, James W. and John W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comput.* 19, 297-301, 1965.

9. Press, William et al. *Numerical Recipes in C: The Art of Scientific Computing.* 2nd Ed. USA: Cambridge University Press, 1988.

10. Van Loan, Charles. Computation Frameworks for the Fast Fourier Transform. Philadelphia: SIAM, 1992.

11. "The Scalapack Web Page," www.netlib.org, 2003.

12. Choy, Ron. "The Star-P Website," http://theory.lcs.mit.edu/~cly/matlabp.html, 2003.