FAST FOURIER TRANSFORM USING PARALLEL PROCESSING FOR MEDICAL APPLICATIONS

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Vinod Jagtap

May, 2010

`

FAST FOURIER TRANSFORM USING PARALLEL PROCESSING FOR MEDICAL APPLICATIONS

Vinod Jagtap

Thesis

Approved:	Accepted:
Advisor Dr. Dale H. Mugler	Dean of the College Dr. George K. Haritos
Co-Advisor Dr. Wolfgang Pelz	Dean of the Graduate School Dr. George R. Newkome
Department Chair Dr. Daniel Sheffer	Date

`

TABLE OF CONTENTS

`

`

`

LIST OF TABLES

`

LIST OF FIGURES

`

`

CHAPTER I

INTRODUCTION

The Fourier Transform is a mathematical operation used widely in many fields. In medical imaging it is used for many applications such as image filtering, image reconstruction and image analysis. It is an important image processing tool which is used to decompose an image into its sine and cosine components. The output of the transformation represents the image in the frequency domain, while the input image is the spatial domain equivalent. In the frequency domain image, each point represents a particular frequency contained in the spatial domain image [8]. The objective of the research is to develop an algorithm for the Fast Fourier Transform so that it will compute the Fourier Transform much faster for input data with fixed length. The algorithm is developed in the c language and MATLAB.

The goal of this research work basically revolves around the use of the Fourier Transform for reconstruction of an image in MRI and CT scan machines. As we know, when MRI machines take an image of the human body, the output is in the form of raw data. The Fourier Transform is used to reconstruct the image from this raw data. When the raw data size is relatively small, it takes moderate time to reconstruct an image. But, as the raw data size continues increasing, the time for processing the reconstruction increases as well. That triggers the quest for a faster way to compute the Fourier Transform. The Fast Fourier Transform is an efficient algorithm available since 1965 to

`

calculate computationally intensive Fourier Transforms. Our goal in this entire work is to develop a strategy to compute the Fast Fourier Transform more efficiently and to reduce the time it takes for calculation. This mathematical transform makes reconstruction of images with larger data size practical.

`

CHAPTER II

LITERATURE REVIEW

2.1 Fourier Transform

In mathematics, the Fourier Transform is an operation that transforms one function of a real variable into another. The new function, often called the frequency domain representation of the original function, describes which frequencies are present in the original function. This is in a similar spirit to the way that a chord of music that we hear can be described by notes that are being played. In effect, the Fourier Transform decomposes a function into oscillatory functions. The Fourier transform is similar to many other operations in mathematics which make up the subject of Fourier Analysis. In this specific case, both the domains of the original function and its frequency domain representation are continuous and unbounded. The term Fourier Transform can refer to both the frequency domain representation of a function or to the process that transforms one function into the other.

Currently, the FFT is used in many areas, from the identification of characteristic mechanical vibration frequencies to image enhancement. Standard routines are available to perform the FFT by computer in programming languages such as Pascal, Fortran a

`

C, and many spreadsheet and other software packages for the analysis of numerical data allow the FFT of a set of data values to be determined readily.

2.2.1 Discrete Fourier Transform

The Fourier transform operates on continuous functions, i.e., functions which are defined at all values of the time t. Such a function might, for example, represent a continually-varying analog voltage signal produced by a microphone or other type of transducer. Digital signal processing involves discrete signals, which are sampled at regular intervals of time rather than continuous signals. A modified form of the Fourier Transform, known as the Discrete Fourier Transform or DFT, is used in the case of discrete signals.

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N}kn} \qquad\qquad k = 0,......N-1$$

where $e^{\frac{2\pi i}{N}}$ is a primitive N'th root of unity.

The Discrete Fourier Transform is one of the most fundamental operations. In mathematics, the Discrete Fourier Transform is one of the specific forms of the Fourier Series Analysis. It transforms one function into another, which is called the frequency domain representation, or simply the DFT, of the original function which is often a function in the time domain. But the DFT requires an input function that is discrete and

`

whose non-zero values have a limited or finite duration. Such inputs are often created by sampling a continuous function. Unlike the discrete-time Fourier Transform, it only evaluates enough frequency components to reconstruct the finite segment that was analyzed. Its inverse transform cannot reproduce the entire time domain, unless the input happens to be periodic. Therefore it is often said that the DFT is a transform for Fourier analysis of finite-domain discrete-time functions. The sinusoidal basis functions of the decomposition have the same properties.

Since the input function is a finite sequence of real or complex numbers, the DFT is ideal for processing information stored in computers. In particular, the DFT is widely employed in signal processing and related fields to analyze the frequencies contained in a sampled signal, to solve partial differential equations, and to perform other operations such as convolutions. The DFT can be computed efficiently in practice using a Fast Fourier Transform algorithm.

When the DFT is applied to a discrete signal, the result is a set of sine and cosine coefficients. When sine and cosine waves of appropriate frequencies are multiplied by these coefficients and then added together, the original signal waveform is exactly reconstructed. The sine and cosine waves are the frequency components of the original signal, in the sense that the signal can be built up from these components. The coefficients determined by the DFT represent the amplitudes of each of these components.

The procedure by which the sine and cosine coefficients are calculated is straightforward in principle, although in practice it requires a great deal of computation. To determine each individual coefficient, every one of the sampled values of the signal

`

must be multiplied by the corresponding sampled value of a sine or cosine wave of the appropriate frequency. These products must then be added together, and the result then divided by the number of samples involved, to give the value of the coefficient.

If the signal consists of a number of samples N, the DFT requires the calculation of N sine and N cosine coefficients. For each coefficient to be determined, N products of samples of the signal and the appropriate sine or cosine wave must be evaluated and summed. The total number of steps in the computation of the DFT is thus N^2, each step requiring the evaluation of a sine or cosine function together with a multiplication (and this does not include the calculation of the N products in order to find each coefficient). To compute the DFT of a signal comprising 1000 samples, say, would entail on the order of one million calculations. The DFT is therefore an extremely numerically intensive procedure.

2.2.2 Definition

The sequence of N complex numbers $x_0, \ldots, x_{N-1}$ is transformed into the sequence of N complex numbers $X_0, \ldots, X_{N-1}$ by the DFT according to the formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N}kn} \qquad k = 0, \ldots\ldots N-1$$

`

where $e^{\frac{2\pi i}{N}}$ is a primitive N'th root of unity. The transform is sometimes denoted by the symbol f, as in $X = f\{x\}$ or $f(x)$ or fx. The inverse discrete Fourier transform (IDFT) is given by

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1}\frac{1}{N}\left(X_k e^{\frac{2\pi i}{N}kn}\right) \qquad n = 0,......N-1$$

A simple description of these equations is that the complex numbers X_k represent the amplitude and phase of the different sinusoidal components of the input "signal" x_n. The DFT computes the X_k from the x_n, while the IDFT shows how to compute the x_n as a sum of sinusoidal components $\frac{1}{N}\left(X_k e^{\frac{2\pi i}{N}kn}\right)$ with frequency k / N cycles per sample. By writing the equations in this form, we are making extensive use of Euler's formula to express sinusoids in terms of complex exponentials, which are much easier to manipulate. Note that the normalization factor multiplying the DFT and IDFT (here 1 and 1/N) and the signs of the exponents are merely conventions, and differ in some treatments. The only requirements of these conventions are that the DFT and IDFT have opposite-sign exponents and that the product of their normalization factors be 1/N. A normalization of $\frac{1}{\sqrt{N}}$ for both the DFT and IDFT makes the transforms unitary, which has some theoretical advantages, but it is often more practical in numerical computation to perform the scaling all at once as above (and a unit scaling can be convenient in other ways).

`

(The convention of a negative sign in the exponent is often convenient because it means that X_k is the amplitude of a "positive frequency" 2πk / N. Equivalently, the DFT is often thought of as a matched filter: when looking for a frequency of +1, one correlates the incoming signal with a frequency of −1.)

2.3 Cooley-Tukey FFT Algorithm

The Cooley-Tukey FFT algorithm is the most common algorithm for developing FFT. This algorithm published in 1965, uses a recursive way of solving discrete Fourier transforms of any arbitrary size N. The technique is to divide the larger DFT into smaller DFT problems, which subsequently reduces the complexity of their algorithm. If the size of the DFT is N, then this algorithm makes N=N1*N2 where N1 and N2 are smaller sized DFT's. The complexity then becomes O (N log N) [12].

Radix-2 decimation-in-time is the most common form of the Cooley-Tukey algorithm, for any arbitrary size N; Radix-2 DIT divides the size N DFT into two interleaved DFT's of size N/2, the DFT as defined earlier,

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N}kn} \qquad k = 0,......N-1$$

Radix-2 divides the DFT into two equal parts, the first part calculates the Fourier Transform of the even index numbers and other part calculates the Fourier Transform of

`

the odd index numbers and then finally merges them to get the Fourier Transform for the whole sequence. This will reduce the overall time to O(N log N). In figure 2.1, a Cooley-Tukey based decimation in frequency for 8-point FFT algorithm is shown:

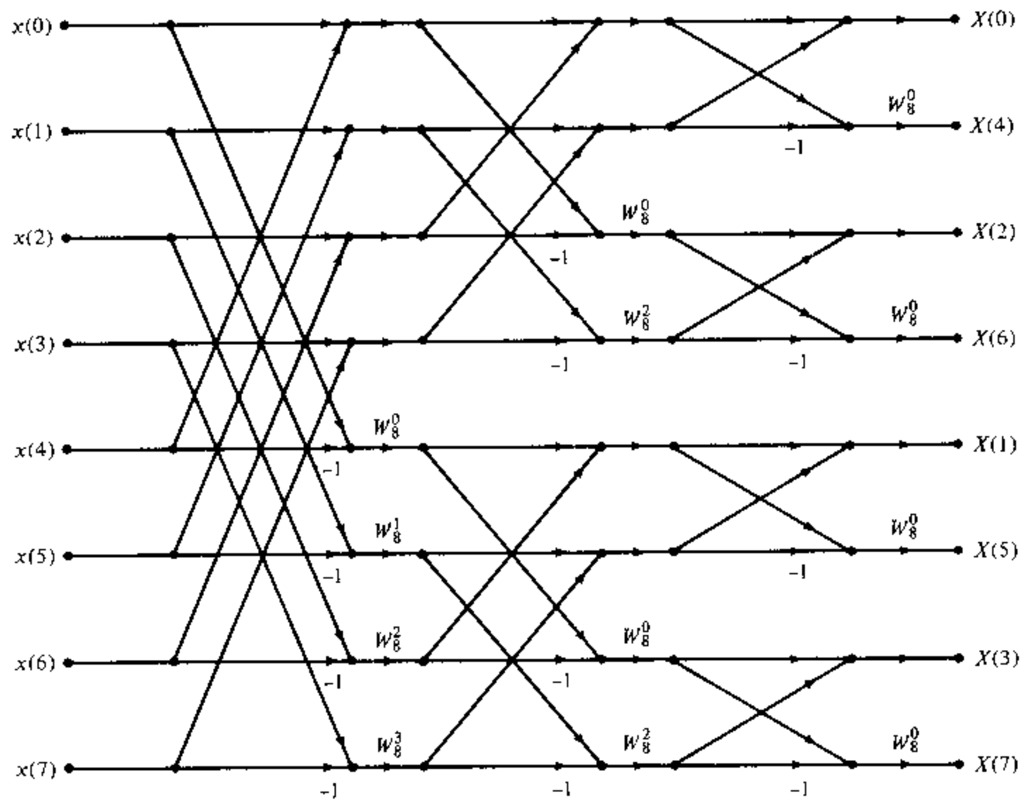


Figure 1: N = 8-point decimation-in-frequency FFT algorithm [12].

The structure describes that given an input N, the algorithm divides it into equal pairs, and further divides it in a recursive way until the data points become 1. Once all the data points are formed, the algorithm then merges them to get the Fourier transform for the whole sequence.

`

Table1: Operation count in DFT and FFT

N	DFT(counts)	FFT(counts)
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16384	896
256	65536	2048
512	262144	4608
1024	1048576	10240

`

CHAPTER III

THE ROLE OF FFT IN THE MEDICAL FIELD

The Fourier transform is a tool for solving physical problems. The Fast Fourier Transform is used widely in the medical field for numerous applications. In a number of medical imaging modalities, the Fast Fourier Transform (FFT) is being used for reconstruction of images from acquired raw data; it is used in medical image de-noising. The Fast Fourier Transform is a powerful technique which facilitates the analysis of signals in the frequency domain. The Fast Fourier Transform (FFT) is at the heart of Digital Signal Processing. It plays an important role in applications involving signal processing such as compression, filtering and de-noising.

3.1 Medical Image Reconstruction in MRI

The signal that we measure in MRI is a combination of signals from the object being imaged. It so happens that any signal can be considered to be a series of sine waves, each with an individual frequency and amplitude. The Fourier Transform allows us to work out what those frequencies and amplitudes are. That is to say, it converts the signal from the time domain into the frequency domain. Since we encode the signal with

`

magnetic field gradients which make frequency and phase okay to position, if we can separate out the frequencies we can determine where we should plot the amplitudes on the image. For any image, use of the Fourier Transform allows us to manipulate the data in the frequency domain (*k*-space), which can be easier, and makes things easier to understand, as in the example of a high pass filter. The high pass filter is nothing but a filter that allows high frequency to pass through it and does not allow low frequency.

3.1.1 Basics of Magnetic Resonance Imaging

Magnetic Resonance Imaging is based on the principles of computational tomography, [1]-[3] and it is used in medical imaging for diagnosis in disease of human body. So far, MRI provides more useful diagnostic information than other competitive techniques. The following is an equation that describes the MRI Imaging in two dimensions,

$$s(k_x, k_y) = f[I(x, y)]$$

where f represents a spatial information encoding scheme [1] and if f is invertible, a data consistent *I* can be obtained from an inverse transform such that,

$$I(x, y) = f^{-1}[s(k_x, k_y)]$$

`

3.1.2 Associated Terminology

Spin: Spin is an important property of electrical charge or mass and usually it is multiples of ½ [4]. Spin can be either positive or negative and is present in protons, neutrons, and electrons.

Spin resonance equation:-

When spins are placed in a magnetic field of strength (B), they exhibit resonance at a well-defined frequency, called the Larmor frequency (*υ)* [1], [4], [5]. The equation governing this is,

$$\upsilon = \gamma B$$

where γ is the gyro-magnetic ratio of the particle.

Spin Energy

The spin vector of a particle aligns itself with the external magnetic field to create two distinct energy states namely, the low energy level and the high energy level [1], [2], [4]. A particle in the low energy state can absorb photons and ends up in the high energy state.

Net Magnetization

The net magnetization can be described as the vector sum of all tiny magnetic fields of each proton pointing in the same direction as the system's magnetic field. The net magnetization and the external field *Bo* points in the z-direction and is called the longitudinal magnetization *Mz*.

`

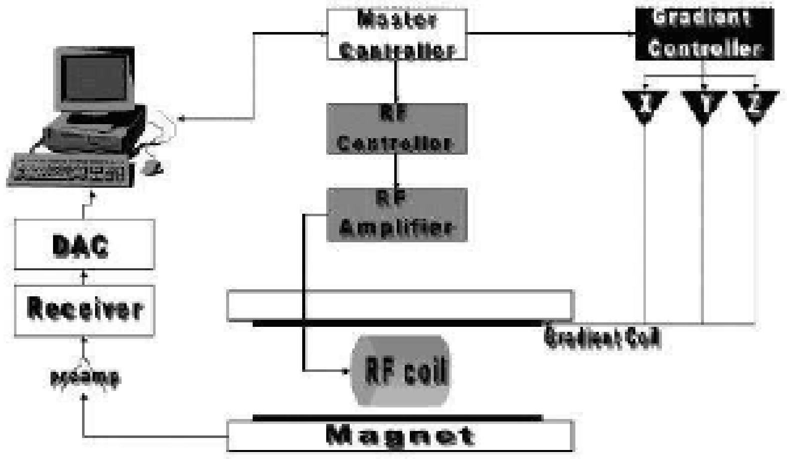


Figure 2: MRI System from Instrumentation point of view [1]-[3]

Spin Echo Relaxation *T*1 Processes

The direction of *Mo* can be changed by exposing the nuclear spin system to energy of a frequency equal to the energy difference between the spin states. If enough energy is put into the system, *Mz* can be made to lie in the x-y plane by the application of an RF field *B*1 [1], [4]. This external force, excites these spins out of equilibrium and tips *Mo* away from the z-axis (the direction of *Bo*), creating a measurable (nonzero) transverse component *Mxy*.

Spin Echo Relaxation *T*2 Processes

If the net magnetization is placed in the x-y plane, it will rotate about the Z axis at a frequency equal to the Larmor frequency. Apart from rotation, the *Mz* (transverse magnetization) starts to de-phase because each of the spin packets is experiencing a different magnetic field which is due to the variation in the *Bo* field [1],[4]. The time

`

constant which describes the return to equilibrium of the transverse magnetization, *Mxy*, is called the spin-spin relaxation time, *T*2.

3.1.3 MRI System

In MRI, the object is placed in the magnetic field in order to take its images, which align and de-align atoms in the object. In this process, RF energy is emitted and absorbed. Typically MRI contains four hardware components,

1) Permanent magnet
2) Magnetic Field Gradient System
3) Radio-Frequency (RF) System
4) Computer/Reconstruction System

3.1.4 MRI Data Acquisition

MRI data acquisition involves three major steps namely

1) Gz, Slice selection by the use of Gz gradient:

2) Gy, Phase encoding using the Gy gradient:

3) Gx, Frequency encoding using the Gx gradient:

The k-space is filled with one line at a time, so that the whole process of slice encoding, phase encoding and frequency encoding has to be repeated many times. It uses either linear or non-linear method to fill the data in the k-space matrix. The usual linear method is recti-linear while centric spiral and reversed centric are two non-linear ones.

`

Recti-linear method is widely used for filling the k-space matrix because of availability of the Fourier transform for reconstruction of k-space data.

3.1.5 k-Space Data

A free induction decay (FID) curve is generated as excited nuclei relax. The FID signal is sampled to get the discrete signal which contains all the necessary information to reconstruct an image, called the k-space signal. The signal sampled in Fourier space contains all the low frequency data at the center which gives the information about the contrast change in gray level with highest amplitude values. On the other hand, the high frequency component gives information about the spatial resolution of the object [13].

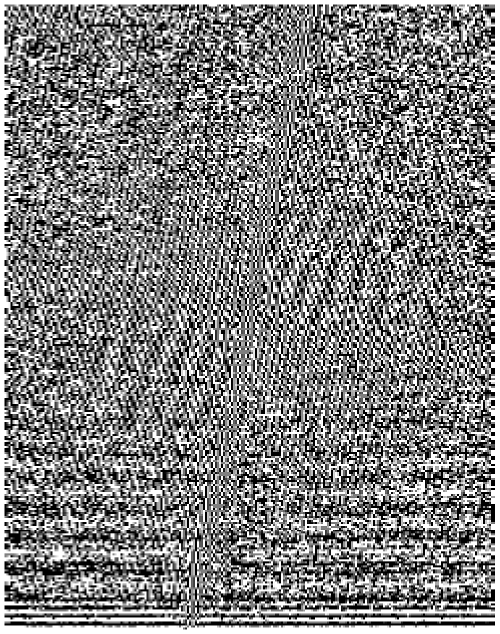


Figure 3: Uniformly sampled K-SPACE data

`

There are various types of methods used to reconstruct MRI image from the sampled k-space data. These include: the use of Discrete Fourier Transform, Radon Transform, Parametric technique, and artificial neural network based reconstruction technique, etc.

3.1.6 Application of Inverse FFT to K-Space Data

To construct an image from k-space data samples using FFT/IFFT, it takes the 1-D IFFT of all the rows. Then, in second step it takes the 1-D IFFT of all columns. Altering the processing sequence of rows and columns doesn’t change the result because of the separability property of the Fourier Transform. In the practical reconstruction of an image from k-space, there are additional steps. It involves the following:

• 1: Read data header information: Load RAW-nmn contains information about the MRI data file. It is a text file containing information about Offset, DATA size, Kx Co-ordinate, Ky-Co-ordinate etc.

• 2: Read in the K-space information.

• 3: IFFT in Kx Direction,

• 4: IFFT in Ky Direction

• 5: IFFT shift

• 6: Image Display

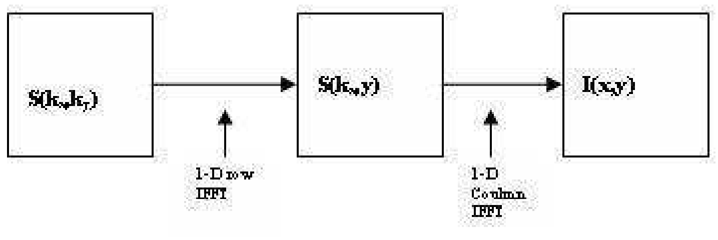


Figure 4: 2D IFFT as cascade of 2 by 1D-IFFT

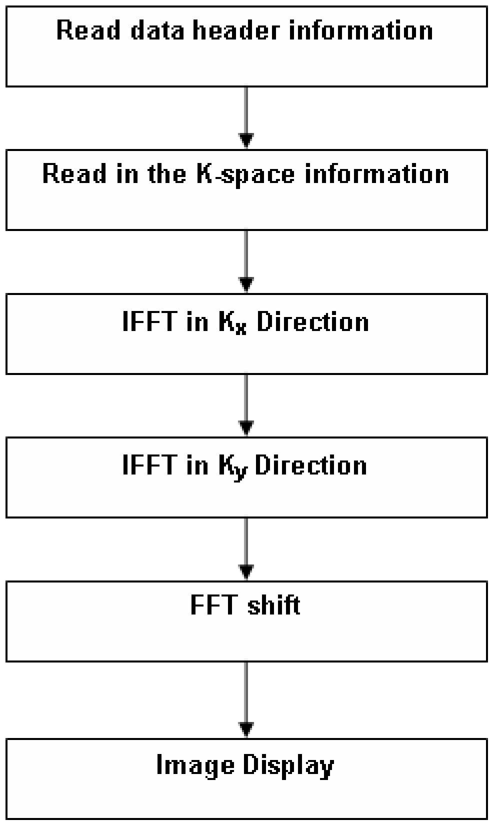


Figure: 5 Flow Diagram of IFFT method of Reconstruction

`

To decompose a 2-D image, we need to perform the 2-D Fourier Transform. The first step consists in performing a 1-D Fourier Transform in one direction (for example in the row direction Ox). In the following example, we can see:

- the original image that will be decomposed row by row
- the gray level intensities of the chosen line
- the spectrum obtained after 1-D Fourier transform

In this, low spatial frequencies are prevailing. Low spatial frequencies have the greatest change in intensity. On the contrary, high spatial frequencies have lower amplitudes. The general shape of the image is described by low spatial frequencies: this is also true with MRI images. The second step of the 2-D Fourier transform is a second 1-D Fourier transform in the orthogonal direction (column by column, Oy), performed on the result of the first one. The final result is a Fourier plane that can be represented by an image.

3.1.7 Challenges of DFT based MRI Image Reconstruction

A few problems are associated with the DFT technique of MRI image reconstruction. These are as follows:

Gibb's effect

Using DFT for reconstruction creates some spurious ringing around some sharp edges [6], [7], [9], [15]. This ringing sequence is due to finite data sequence of MRI. To remove this, a large data set of MRI can be used during reconstruction. But due to practical limitations, other solutions are used such as extrapolating missing data points by using parametric modeling and artificial neural network.

`

Truncation Artifacts

The Fourier Transform of a rectangular window is the sinc function and the sinc function sidelobe has appreciable amplitude which introduces uncertainty in the discrimination of anatomical detail in the reconstructed images as truncation artifacts.

This truncation effect can be reduced by using window functions which have lower sidelobe amplitudes. Experimental results show that the Hamming Window outperforms other windows in this concern.

Decrease in Spatial resolution

The use of the FFT for MRI reconstruction leads to a decrease in the spatial resolution [1], [5], [6]. Using a window function to reduce the Gibbs effect and truncation artifacts leads to blurring of the reconstructed image which gives poor spatial resolution.

`

CHAPTER IV

IMAGE PROCESSING USING FFT IN 2-D

When concerned with digital images, we will consider the Discrete Fourier Transform (DFT). The DFT is the sampled Fourier Transform and therefore does not contain all frequencies forming an image, but only a set of samples which is large enough to fully describe the spatial domain image. The number of frequencies corresponds to the number of pixels in the spatial domain image, *i.e.* the image in the spatial and Fourier domain is of the same size. For a square image of size N×N, the two-dimensional DFT is given by:

$$F(k,l) = \frac{1}{N^2}\sum_{a=0}^{N-1}\sum_{b=0}^{N-1} f(a,b)e^{-j2\pi(\frac{ka}{N}+\frac{lb}{N})}$$

where *f(a,b)* is the image in the spatial domain and the exponential term is the basis function corresponding to each point *F(k,l)* in the Fourier space. The equation can be interpreted as: the value of each point *(k,l)* is obtained by multiplying the spatial image with the corresponding base function and summing the result. The basic functions are sine and cosine waves with increasing frequencies, *i.e.* *F(0,0)* represents the DC-component of the image which corresponds to the average brightness and *F(N-1,N-1)* represents the highest frequency. In a similar way, the Fourier image can be re-transformed to the spatial domain. The inverse Fourier transform is given by:

`

$$f(a,b) = \frac{1}{N^2}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1} F(k,l)(e^{i2\pi(\frac{ka}{N}+\frac{lb}{N})})$$

To obtain the result for the above equations, a double sum has to be calculated for each image point. However, because the Fourier Transform is *separable*, it can be written as

$$F(k,l) = \frac{1}{N}\sum_{b=0}^{N-1} P(k,b)e^{-\ell 2\pi\frac{lb}{N}}$$

Where

$$P(k,b) = \frac{1}{N}\sum_{a=0}^{N-1} f(a,b)e^{-\ell 2\pi\frac{ka}{N}}$$

Using these two formulas, the spatial domain image is first transformed into an intermediate image using N one-dimensional Fourier Transforms. This intermediate image is then transformed into the final image, again using N one-dimensional Fourier Transforms. Expressing the two-dimensional Fourier Transform in terms of a series of *2N* one-dimensional transforms decreases the number of required computations.

Even with these computational savings, the ordinary one-dimensional DFT has N^2 complexity. This can be reduced to $N\log_2 N$ if we employ the Fast Fourier Transform (FFT) to compute the one-dimensional DFTs. This is a significant improvement for large images. There are various forms of the FFT and most of them restrict the size of the input image that may be transformed, often to $N = 2^n$ where *n* is an integer. The Fourier Transform produces a complex number-valued output image

`

which can be displayed with two images, either with the *real* and *imaginary* part or with *magnitude* and *phase*. In image processing, often only the magnitude of the Fourier Transform is displayed, as it contains most of the information of the geometric structure of the spatial domain image. However, if we want to re-transform the Fourier Image into the correct spatial domain after some processing in the frequency domain, we must make sure to preserve both magnitude and phase of the Fourier Image. The Fourier domain image has a much greater range than the image in the spatial domain. Hence, to be sufficiently accurate, its values are usually calculated and stored in float values.

4.1.1 Early Work

In this area of developing the Fast Fourier Transform using a parallel computation, two major computer codes have been developed so far by previous UA students. Both of these were implemented using the MPI-message passing interface technique to make it parallel and the code is written in computer C language. The reason for using C language is MPI friendly and easy to implement. Both the previous codes used different techniques to implement the Fourier Transform and the timings were analyzed for a different number of input data points and for a different number of processors on which they ran. For calculating the Fast Fourier transform, both of them followed the newly developed interleaved Fourier Transform and its variations (such as Centered Fourier Transform) by Dr. Mugler.

`

4.1.2 Misal’s Code

The code written by Misal [10] using MPI in C language has been a very efficient code so far developed under guidance of Dr. Mugler. Misal’s idea was to parallelize the Fast Fourier computation using the Message Passing Interface. He divided the computation into parts, assigned those parts to different processors, and the parts were processed in parallel. This code is a long code comparatively and has subroutines for calculating Fourier transform individually for different values of N, i.e. as you run the code, at the start it will check for the value of N. Accordingly it will select the subroutine to carry out the calculation. In this code, subroutines are written for different values of N such as N=8, 16, 32…….1024…etc. This is reason it is a long code. Perhaps that is the reason the code is so efficient. Meaning, depending upon the value of N, it is just executing corresponding subroutines each of which is hardly couple of hundreds of lines long. Each time you run the code, it is not executing the entire several thousand line code, but just several related subroutines.

4.1.3 Mirza’s Code

On the other hand, Mirza’s [19] code has been altogether different than Misal’s code. This code also has several subroutines but its compact code is hardly several hundred lines long. Comparatively, the previous code is 10 times longer than Mirza’s’s code. The reason behind that is that it is a generalized code; it doesn’t have individual subroutines for different values of N, but is generalized for all values of N. It does have

`

subroutines but those are for calculating other required mathematical computations such as sum-difference and bit reversal. This is the reason behind being so compact. Also this code implements a few different techniques for making it faster such as lifting function and gg90 functions. These functions are implemented to reduce the sine and cosine calculations (additions and multiplications) of the DCT function in the Fourier Transform done by the individual processors. At the end of development, it has been the conclusion that these functions hardly helped in making the code run more efficiently. Surprisingly while testing the later code; it had the advantage of being tested on much more efficient hardware, faster processors along with much more memory. Still the development in the later code has hardly made it more efficient; and the older code being tested on less advanced hardware, was more efficient and stands as benchmark in the development of the Fast Fourier Transform using parallel computation.

4.2 New Approach

In this thesis, we decided to try a new approach to reduce the processor burden of calculating the additions and multiplications for sine and cosine functions. In this approach we determine a strategy which will calculate the necessary sine and cosine values of angles before the main program starts. In this way, when actual computation of the Fourier Transform is going on, it will just take the calculated values of sines and cosines which will make it more efficient.

Another new method is set of functions called Cordic functions, which we decided to implement. The good thing about the cordic function is that it allows you to

`

calculate the same sine and cosine of angles without multiplicative calculations and that is the reason we believe it will make the algorithm more efficient. In this algorithm, instead of multiplications it uses the technique of bit shift to get the desired results.

Cordic stands for COordinate Rotation DIgital Computer and is a simple and efficient algorithm to calculate hyperbolic and trigonometric functions. The only operations it requires are addition, subtraction, bit shift and table lookup. Cordic gets its name because it uses properties of rotation on a plane to calculate sines and cosines.

`

CHAPTER V

PARALLEL STRUCTURE OF INTERLEAVED FFT

5.1 General structure of FFTP

The FFTP computation of the Fourier transform consists of three main parts. These are as given below:

1. Summation: The summation part of the computation splits the data into odd *xo* and even *xe* vectors. For example, as shown in the figure below for the data of length 64, odd and even vectors are formed. Odd and even vector does not refer to the length of the vector but to the symmetry of the vector. So an odd vector can be of even length but have odd symmetry. Whereas an even vector has even symmetry.
2. DCT-IV/DST-IV computation: The cosine and sine transforms of the odd and even vectors is done. Direct factorizations of the DCT-IV matrix can be done to obtain the fast cosine transform. But currently it is being computed indirectly by DCT-II transform followed by pre and post processing. The sine transform can be obtained by trivially modifying the odd vector. This is simply done by changing the sign of the alternating data points of the odd vector and taking its DCT transform.

`

3. Sorting: For a real valued input vector the output of the DCT-IV and DST-IV are the real and imaginary parts of the final output respectively. The outputs of these transforms need to be interleaved or sorted. Interleaving consists of placing the output values next to each other.

`

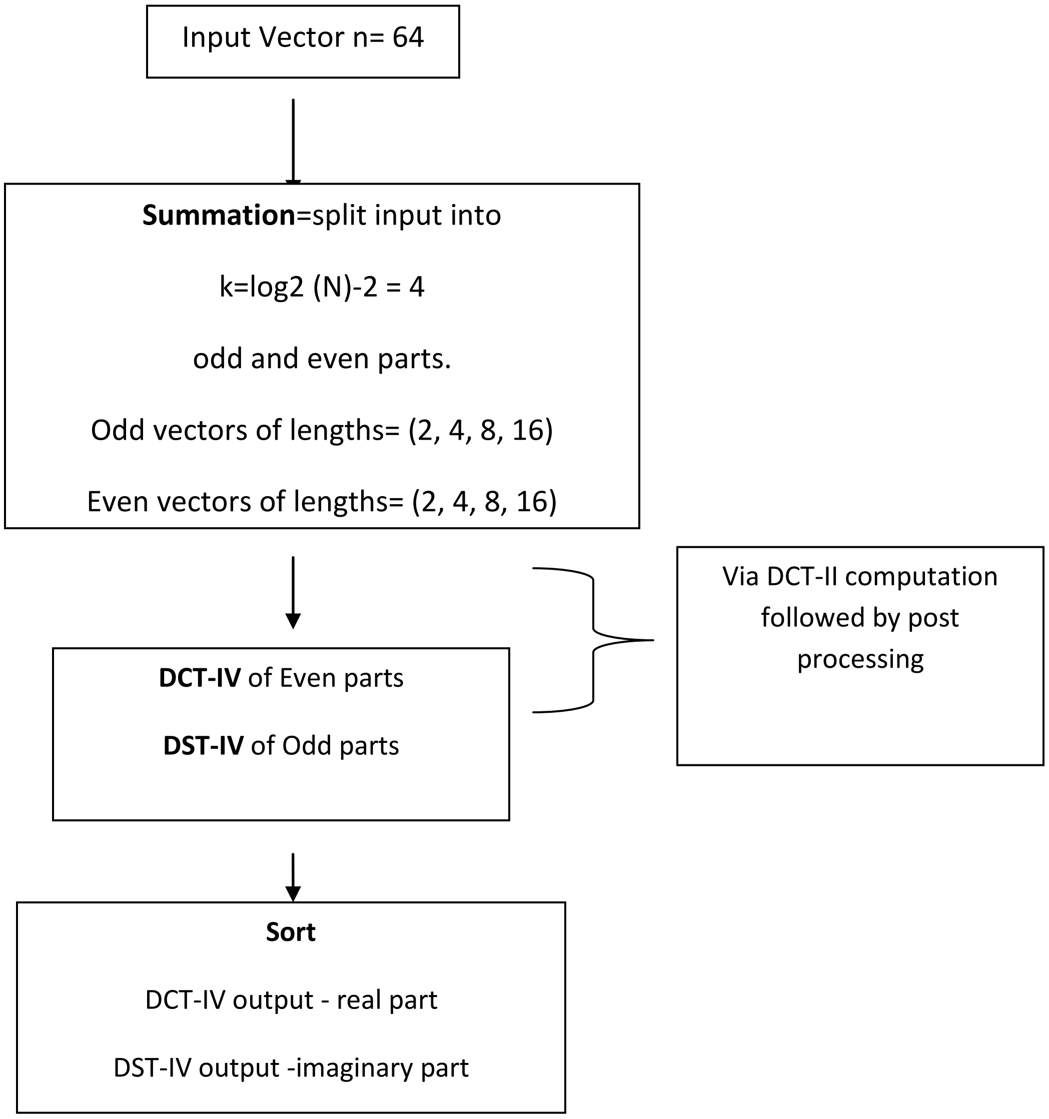

Input Vector n= 64
Summation=split input into
k=log2 (N)-2 = 4
odd and even parts.
Odd vectors of lengths= (2, 4, 8, 16)
Even vectors of lengths= (2, 4, 8, 16)
Via DCT-II computation followed by post processing
DCT-IV of Even parts
DST-IV of Odd parts
Sort
DCT-IV output - real part
DST-IV output -imaginary part

Figure 6: Block diagram of FFTP

`

5.2 Parallel Implementation of FFTP

After going over the basic working of the FFTP computation, the parallel implementation of it is discussed briefly below. The parallel implementation of FFTP has the advantage that the computations required can be fairly equally divided among processes without the overhead of communication generally involved in FFT computation. This is perhaps the main advantage of using this method. Also reasonable gain is possible without much complication due to this feature.

Direct calculation of the 2D DFT is simple, but requires a very large number of complex multiplications as we know. Assuming all of the exponential terms are pre-calculated and stored in a table, the total number of complex multiplications needed to evaluate the 2D DFT is N^4. The number of complex additions required is also N^4. Two techniques can be employed to reduce the operation count of the 2D-DFT transform. First, the row-column decomposition method partitions the 2D-DFT into many one-dimensional DFTs. Row-column decomposition reduces the number of complex multiplications from N^4 to 2 N^3 .The second technique to reduce the operation count of the 2D-FFT transform is the Fast Fourier transform (FFT). The FFT is a shortcut evaluation of the DFT.

The FFT is used to evaluate the one-dimensional DFTs produced by the row-column decomposition. The number of complex multiplications is reduced from 2 N^3 for direct evaluation of each DFT to N^2 log N for FFT evaluation using the row-column decomposition method. The 2D-FFT transform technique: A 2-D FFT is an intrinsically parallel operation; a 1-D FFT is applied separately to each row and column of a matrix.

`

This means that in the first part of calculation, it will calculate the FFT in one dimension of all the column elements. Take those values put back in the matrix of the image. Then in the next part, it will take all the rows of the image matrix; apply the 1 Dimensional FFT on it. Put these values back in the image matrix.

This technique has tremendous potential in terms of increasing the speed of execution of the computationally complex arithmetic involved in the calculation of the Fourier transform.

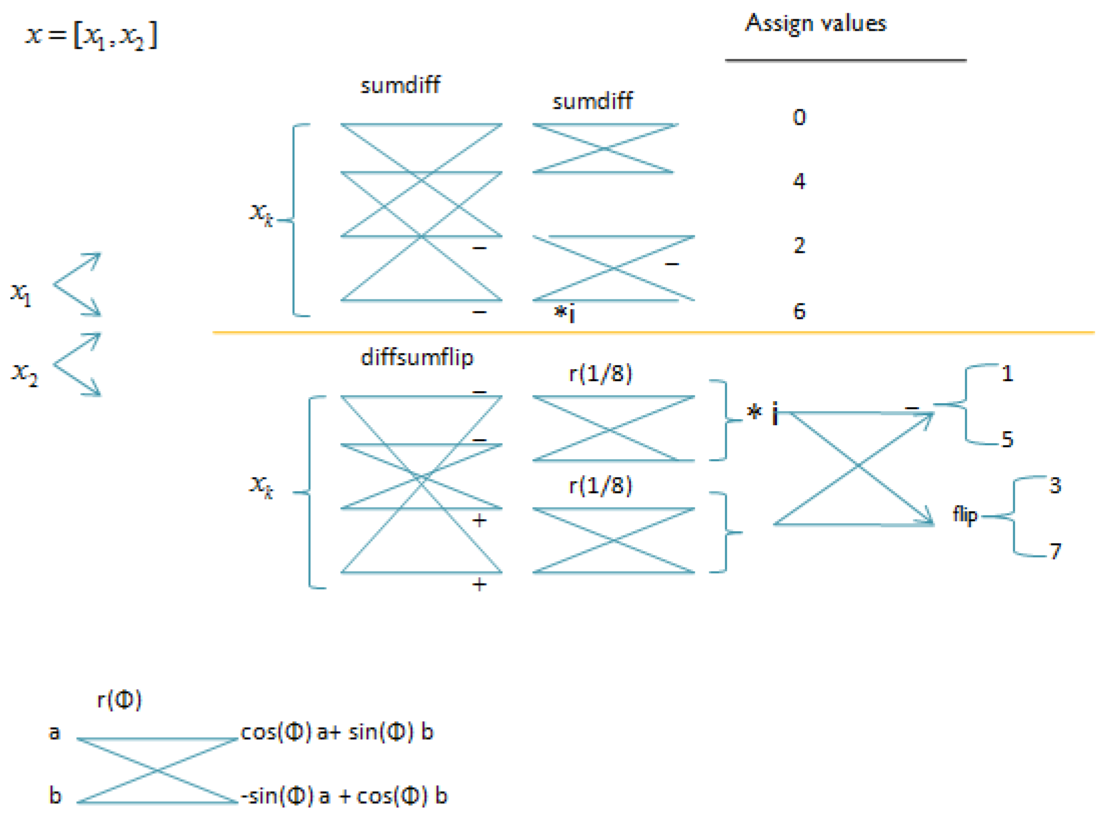


Figure: 7 Fast Fourier Transform for N=8 data points

`

CHAPTER VI

COMPUTATIONAL COMPLEXITY OF PARALLEL FFT AS COMPARED TO CONVENTIONAL FFT

As we have seen, the structure of the Fast Fourier Transform in parallel differs from the conventional style FFT. Though it is simple in structure, we know that the conventional Fast Fourier Transform itself is computationally complex to calculate. It involves the total number of complex multiplications N^4 and the number of complex additions of N^4 needed to evaluate the 2D DFT. As N increases with size of an image, it becomes practically impossible to carry out all the calculations and on top of that the time required for that is long. On the other hand, the Fast Fourier Transform, as stated earlier, is an efficient way to calculate the Fourier Transform and it reduces the large number of complex additions and multiplications to $N^2 \log N$. The Fast Fourier Transform which acts in a parallel fashion is very complicated structurally.

Also, when making it parallel in our approach, we are trying to divide the total number of computations in a number of processors and these divided tasks will be given simply to individual processors which will carry them out in parallel. In this entire process, what is really important is to make sure that each processor gets equal amount of load and also make sure the overhead of communication is relatively low. This makes dividing the entire task into a number of equally important processors. Developing a

`

strategically efficient algorithm and providing faster computational times for MRI images is the main goal of this proposal.

`

CHAPTER VII

FASTEST FOURIER TRANSFORM IN THE WEST [16]

7.1 Introduction

“This section describes the Fastest Fourier Transform in the West [16]. It is a portable package in the C language for computing one or multi-dimensional complex discrete Fourier transforms (DFT). This stands as a benchmark in publicly available FFT packages. Extensive benchmarking demonstrates that the FFTW is typically faster than all other DFT software, including the well-known FFTPACK and the code from Numerical Recipes. More interestingly, FFTW is competitive with or better than proprietary, highly-tuned codes such as Sun’s Performance Library and IBM’s ESSL library. The Fastest Fourier Transform in the West essentially is an implementation of the Cooley-Tukey fast Fourier transform. Three main key ideas are:

- The computation of the transform is performed by an *executor* consisting of highly-optimized, composable blocks of C code called *codelets*.
- At runtime, a *planner* finds an efficient way (called a *plan*) to compose the codelets. Through the planner, FFTW adapts itself to the architecture of the machine it is running on. In this way, FFTW is a *single* program that performs efficiently on a variety of architectures.

`

- The codelets are automatically generated by a *codelet generator* written in the Caml Light dialect of ML. The codelet generator produces long, optimized, unreadable code, which is nevertheless easy to modify via simple changes to the generator.

Despite its internal complexity, what makes the Fastest Fourier Transform in the West easy to use is the fact that the user (See Figure 1) interacts with FFTW only through the planner and the executor; the codelet generator is not used after compile-time. FFTW provides a function that creates a plan for a transform of a certain size. Once the user has created a plan, she can use the plan as many times as needed. A good thing about the Fastest Fourier Transform in the West is that it is not restricted to transforms whose size is a power of 2. The executor implements the well-known Cooley-Tukey algorithm, which works by factoring the size N of the transform into N = N1*N2. The algorithm then recursively computes N1 transforms of size N2 and N2 transforms of size N1. The base case of the recursion is handled by the codelets, which are hard-coded transforms for various small sizes. They emphasize that the executor works by explicit recursion, in sharp contrast with the traditional loop-based implementations. The recursive divide-and-conquer approach is superior on modern machines, because it exploits all levels of the memory hierarchy: as soon as a sub-problem fits into the cache, no further cache misses are needed in order to solve that sub-problem. Their results contradict the folk theorem that recursion is slow. Moreover, the divide-and-conquer approach parallelizes naturally in Cilk. The Cooley-Tukey algorithm allows arbitrary choices for the factors N1 and N2 of N. The best choice

`

depends upon hardware details such as the number of registers, latency and throughput of instructions, size and associability of caches, structure of the processor pipeline, etc.

```
 fftw_plan plan;
int n = 1024;
COMPLEX A[n], B[n];
/* plan the computation */
plan = fftw_create_plan(n);
/* execute the plan */
Fftw (plan, A);
/* the plan can be reused for other inputs
of size n */
fftw(plan, B);
```

This is simplified example of FFTW's use. The user must first create a plan, which can be then used at will. In the actual code, there are a few other parameters that specify the direction, dimensionality, and other details of the transform.

Most high-performance codes are tuned for a particular set of these parameters. In contrast, FFTW is capable of optimizing itself at runtime through the planner, and therefore the *same* code can achieve good performance on different kinds of architecture. We can imagine the planner as trying all factorizations of N supported by the available codelets, measuring their execution times, and selecting the best. In practice, the number of possible plans is too large for an exhaustive search. In order to prune the search, they

`

assume that the optimal solution for a problem of size N is still optimal when used as a subroutine of a larger problem. With this assumption, the planner can use a dynamic-programming algorithm to find a solution that, while not optimal, is sufficiently good for practical purposes. The solution is expressed in the form of byte-code that can be interpreted by the executor with negligible overhead. Their results contradict the folk theorem that byte-code is slow. The codelet generator produces C subroutines (codelets) that compute the transform of a given (small) size. Internally, the generator itself implements the Cooley-Tukey algorithm in symbolic arithmetic, the result of which is then simplified and unparsed to C. The simplification phase applies to the code many transformations that an experienced hacker would perform by hand. The advantage of generating codelets in this way is twofold. First, we can use much higher radices than are practical to implement manually (for example, radix-32 tends to be faster than smaller radices on RISC processors). Second, they can easily experiment with diverse optimizations and algorithmic variations. For example, it was easy to add support for prime factor FFT algorithms within the codelets. Savage [18] gives an asymptotically optimal strategy for minimizing the memory traffic of the FFT under very general conditions. Their divide-and-conquer strategy is similar in spirit to Savage's approach. The details of their implementation are asymptotically suboptimal but faster in practice.

7.2 Performance

In this section, we present the data from two machines: a 167-MHz Sun Ultra SPARC-I and an IBM RS/6000 Model 3BT (Figures 8 through 12) [16]. The

`

performance results are given as a graph of the speed of the transform in MFLOPS versus array size for both one and three dimensional transforms. The MFLOPS count is computed by postulating the number of floating point operations to be $5N \log_2 N$, where N is the number of complex values being transformed. This metric is imprecise because it refers only to radix-2 Cooley-Tukey algorithms. All the benchmarks were performed in double precision. A complete listing of the FFT implementations included in the benchmark is given in Table 1. Figure 2 shows the results on a 167MHz Ultra SPARC-I. FFTW outperforms the Sun Performance Library for large transforms in double precision, although Sun's software is faster for sizes between 128 and 2048. In single precision (Figure 4) FFTW is superior over the entire range. On the RS/6000 FFTW is always comparable or faster than IBM's ESSL library, as shown in Figures 3 and 6. The high priority that was given to memory locality in FFTW's design is evident in the benchmark results for large, one-dimensional transforms, for which the cache size is exceeded. Especially dramatic is the factor of three contrasts on the RS/6000 (Figure 3) between FFTW and most of the other codes.

A notable program is the one labeled 'CWP' in the graphs, which sometimes surpasses the speed of FFTW for large transforms. Unlike all other programs tried, CWP uses a prime-factor algorithm instead of the Cooley-Tukey FFT. CWP works only on a restricted set of transform sizes. Consequently, the benchmark actually times it for a transform whose size (chosen by CWP) is slightly larger than that used by the rest of the codes. It is included on the graph since, for many applications, the exact size of the transform is unimportant. The reader should be aware that the point-to-point comparison of CWP with other codes may be meaningless: CWP is solving a bigger problem and, on

`

the other hand, it is choosing a problem size it can solve efficiently. The results of a particular benchmark run were never entirely reproducible. Usually, the differences from run to run were 5% or less, but small changes in the benchmark could produce much larger variations in performance, which proved to be very sensitive to the alignment of code and data in memory. It was possible to produce changes of up to 10% in the benchmark results by playing with the data alignment (e.g. by adding small integers to the array sizes). More alarmingly, changes to a single line of code of one FFT could occasionally affect the performance of another FFT by more than 10%. The most egregious offender in this respect was one of our Pentium Pro machines running Linux 2.0.17 and the gcc 2.7.2 compiler. On this machine, the insertion of a single line of code into FFTW slowed down a completely unrelated FFT (CWP) by almost a factor of twenty. Consequently, we do not dare to publish any data from this machine. We do not completely understand why the performance Pentium Pro varies so dramatically. Nonetheless, on the other machines, the overall trends are consistent enough to give confidence in the qualitative results of the benchmarks. Our benchmark program works as follows. The benchmark uses each FFT subroutine to compute the DFT many times, measures the elapsed time, and divides by the number of iterations to get the time required for a single transform.

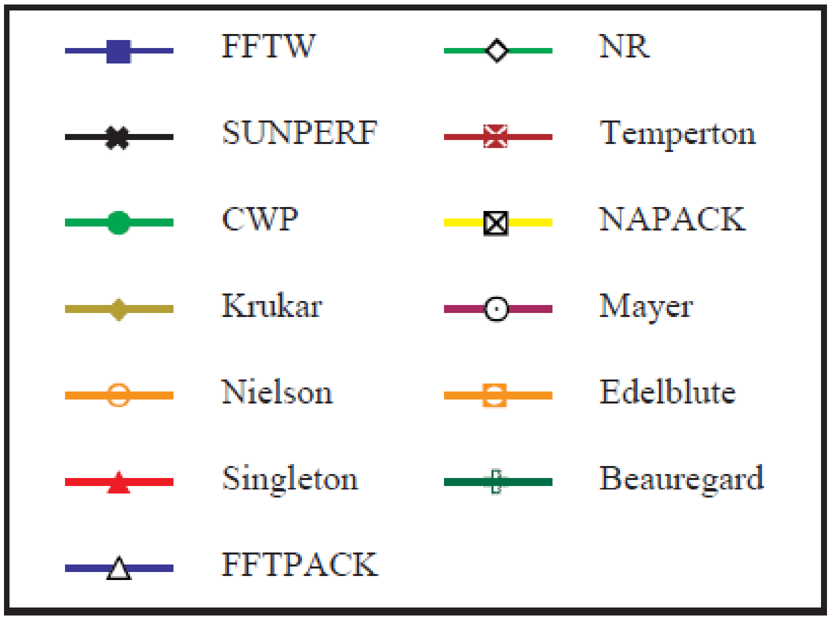


Figure 8: Notation for different graphs

`

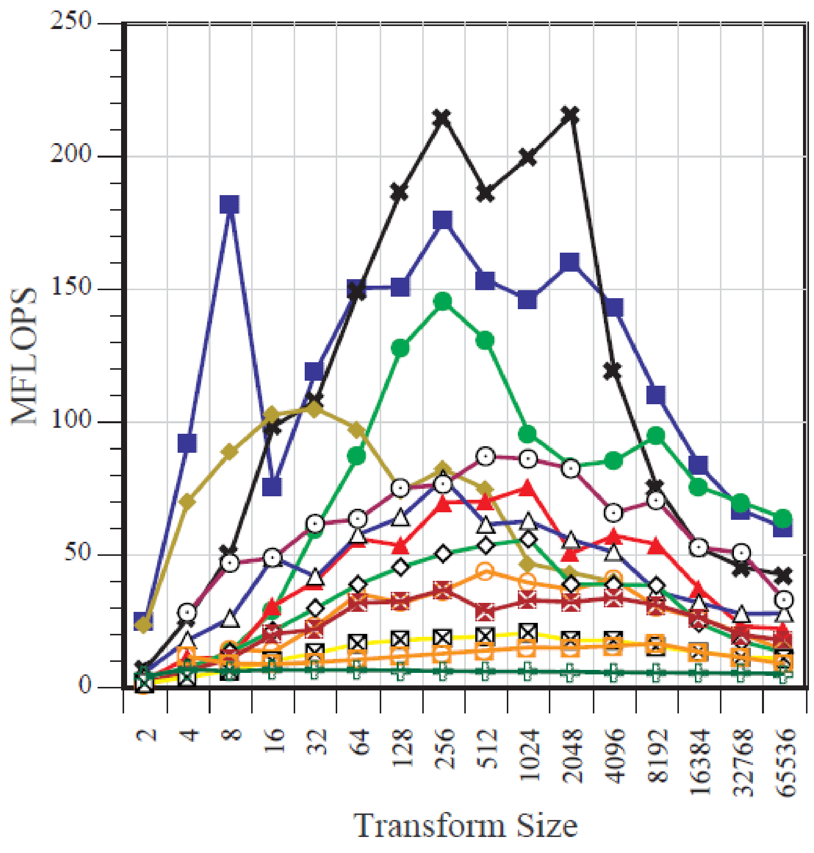


Figure 9: Comparison of 1DFFTs on a SunHPC 5000 (167MHzUltraSPARC-I).Compiled with cc -native -fast -xO5 -dalign. SunOS 5.5.1, cc version 4.0. [16]

`

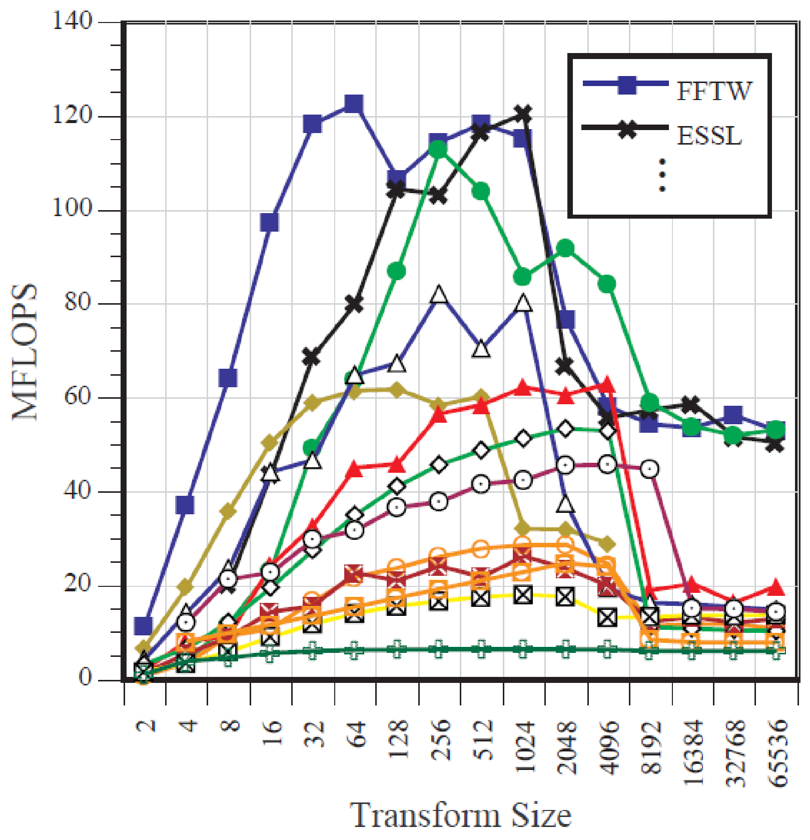


Figure 10: Comparison of 1D FFTs on an IBM RS/6000, model 3BT, running AIX version 3.2. Compiled with cc -O3 -qarch=pwrx -qtune=pwrx. [16]

`

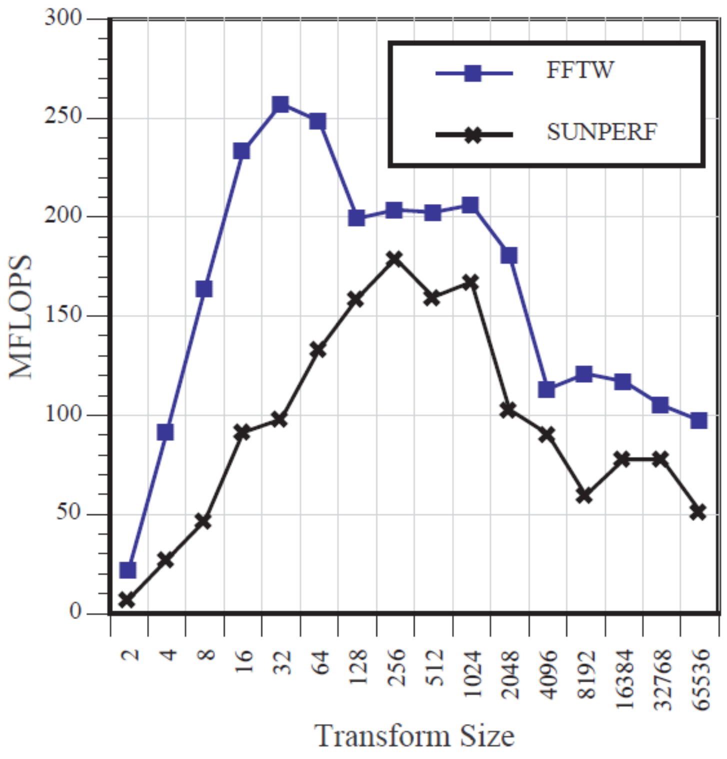


Figure 11: Comparison of FFTWwith the Sun Performance Library on the UltraSPARC for single precision 1D transforms. [16]

`

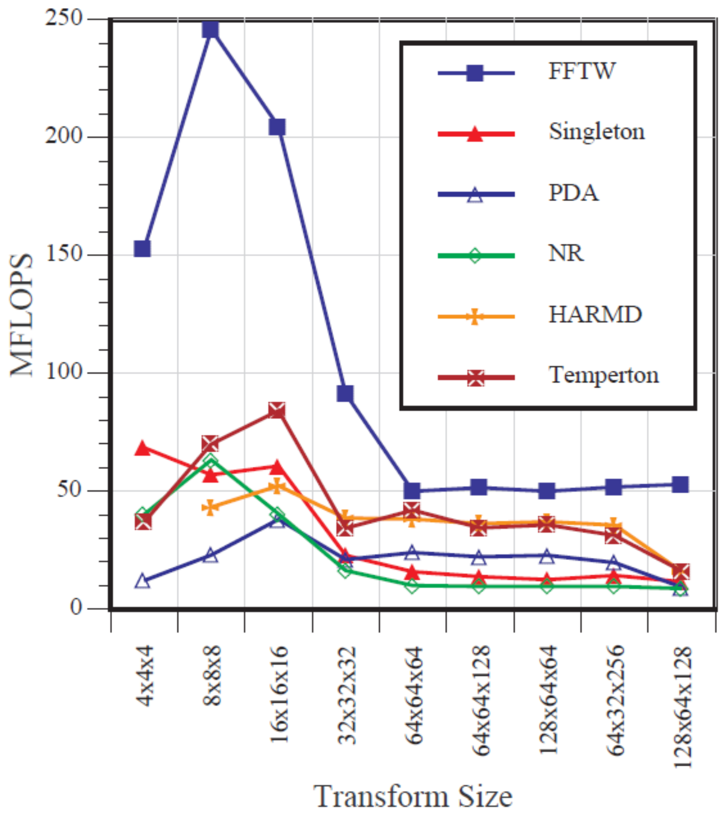


Figure 12: Comparison of 3D FFTs on the UltraSPARC. [16]

`

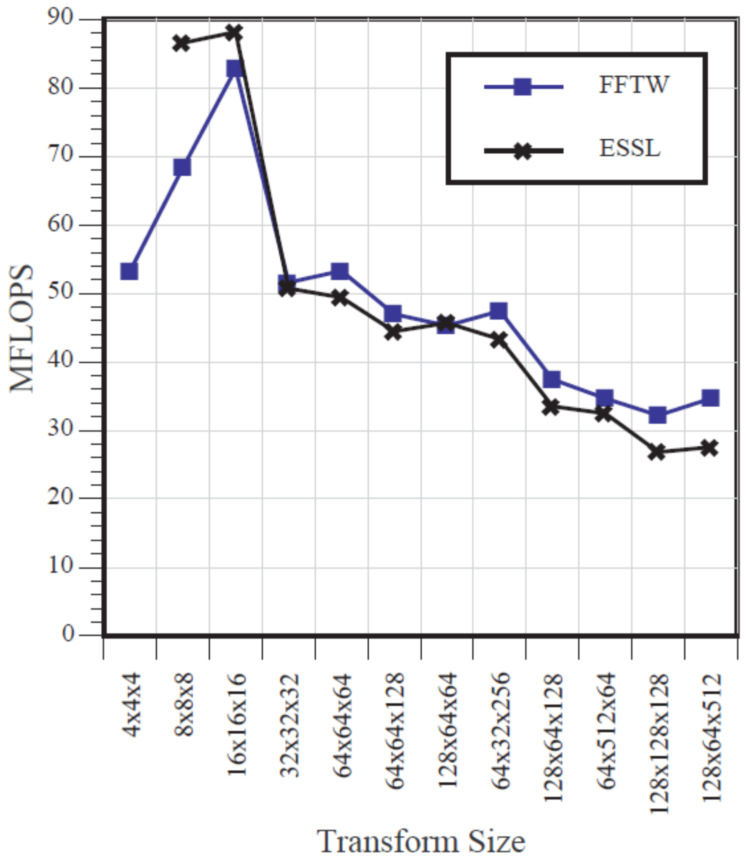
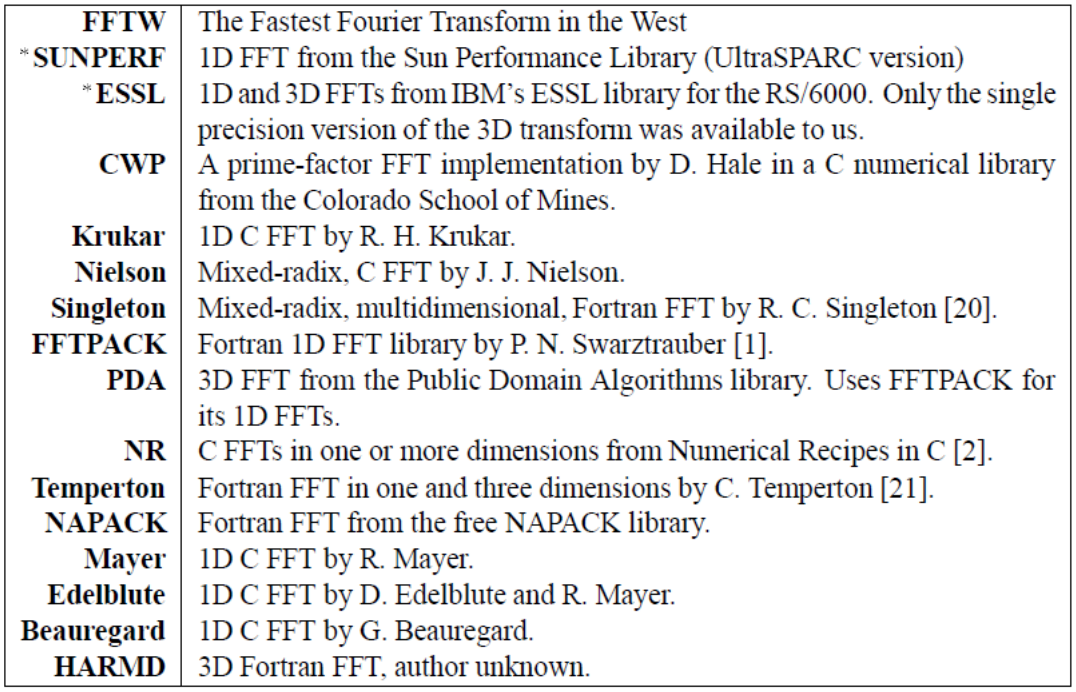


Figure 13: Comparison of 3D transforms from FFTW and the ESSL library on the RS/6000, in single precision. [16]

`

Table 2: Description of the programs benchmarked. All codes are generally available except for the entries marked with an asterisk, which are proprietary codes optimized for particular machines. [16]



FFTW	The Fastest Fourier Transform in the West
* **SUNPERF**	1D FFT from the Sun Performance Library (UltraSPARC version)
* **ESSL**	1D and 3D FFTs from IBM's ESSL library for the RS/6000. Only the single precision version of the 3D transform was available to us.
CWP	A prime-factor FFT implementation by D. Hale in a C numerical library from the Colorado School of Mines.
Krukar	1D C FFT by R. H. Krukar.
Nielson	Mixed-radix, C FFT by J. J. Nielson.
Singleton	Mixed-radix, multidimensional, Fortran FFT by R. C. Singleton [20].
FFTPACK	Fortran 1D FFT library by P. N. Swarztrauber [1].
PDA	3D FFT from the Public Domain Algorithms library. Uses FFTPACK for its 1D FFTs.
NR	C FFTs in one or more dimensions from Numerical Recipes in C [2].
Temperton	Fortran FFT in one and three dimensions by C. Temperton [21].
NAPACK	Fortran FFT from the free NAPACK library.
Mayer	1D C FFT by R. Mayer.
Edelblute	1D C FFT by D. Edelblute and R. Mayer.
Beauregard	1D C FFT by G. Beauregard.
HARMD	3D Fortran FFT, author unknown.

After each FFT, however, it is necessary to reinitialize the array being transformed (iteration of the DFT is a diverging process). The time for these re-initializations is measured separately and subtracted from the elapsed time mentioned above. Instead of reinitializing the input array after each transform, one could alternatively follow the transform by the inverse transform in each iteration. Many FFT implementations compute an un-normalized DFT, however, and thus it would be necessary to have an additional loop to scale the data properly. They did not want to

`

measure this additional cost, since in many applications the scale factor can easily be absorbed into other multiplicative factors to avoid the extra multiplications and loads.

It was an intention to benchmark C FFTs, but much of the public-domain software was written in Fortran. These codes were converted to C via the free f2c software. This raises some legitimate concerns about the quality of the automatic translation performed by f2c, as well as the relative quality of Fortran and C compilers. Accordingly, when compared the original Fortran FFTPACK with the f2c version. On average, the C code was 16% slower than the Fortran version on the RS/6000 and 27% slower on the Ultra SPARC. The Fortran code was never faster than FFTW."

`

CHAPTER VIII

MISAL'S WORK: BENCHMARK

In this Nilimb's thesis work [17], a parallel implementation of a new method called the parallel Fast Fourier Transform (FFTP) was developed. This method has certain structural advantages, which are particularly suitable for parallel implementation of the FFT. These were as listed below:

1. This method allows independent computation of different parts of the input data. After the processing, the result can be sorted to get the final output.
2. The sequential version of this method already closely matches in speed to the currently available fast algorithms. For real input data, the sequential version of this code has a competitive flop count as compared to other FFT methods. For integer input data (such as gray scale images), the sequential version has the lowest flops (floating point operations) compared to other methods such as the radix-2 and the split-radix methods.
3. The algorithm structure was comparatively simple, allowing for an easy implementation of the algorithm in sequential and parallel versions.
4. The user had the option of doing a faster computation, which was coarser. For example, for the case of a vector of length 16, the user could choose to only perform the calculation of 8 values. Those values could be used to provide a

`

coarse output of the transform, based on the higher resolution. It would be a trade-off between speed of computation and coarseness of the transform. If desired, the other values could be computed later and included in a higher resolution version of the transform at that time.

There were four important developments made by Dr. Mugler before the start of that work. They were as follows

1. The symmetry property of the centered matrix.
2. The sorting concept for computing the shifted Fourier transform by the centered Fourier transform.
3. The relation of DCT-IV and DST-IV to the centered Fourier transform for fast computation of the centered Fourier transform.
4. The tree structure for computing the conventional FFT through the centered Fourier transform.

So far in all the work that Dr.Mugler did on the development of Fast Fourier Transform, the work developed by Nilimb stands as the benchmark. The timings for different cases of input data size with different number of processors are still the best. As said before, apart from him, another graduate student worked on the same development with different approach. Ameen Mirza from the department of computer science, introduced different concepts like the lifting function, unfortunately his work did not yield good timings for calculation of the Fourier Transform. In this section we are trying to see the results of Nilimb's work and we are going to compare it with our results later on.

`

In the benchmark the desktop configuration was Pentium-IV, 512 MB RAM 1.67 GHz (Clock cycle: 0.000000279 seconds). Microsoft VC++ 6.0 compiler (optimization turned on) and Windows XP operating system were used. The timing was measured in the number of clock cycles needed for the FFT computation for that particular value of *N*. [17]

Table 3: Benchmark results for real 1-Dimensional input vector. Entries are the number of clock cycles needed. [17]

N	FFTP	Ooura	Mayer	FFTW E	FFTW M	Duhamel	Num. Recipe
32	36	111	32	14	8	45	27
64	45	116	47	21	11	89	41
128	62	138	74	78	17	181	70
256	88	199	132	111	47	376	140
512	201	325	263	167	99	790	295
1024	335	580	513	433	219	1673	678
2048	653	1126	1140	806	481	NA	1485
4096	1352	2341	2304	1624	1081	NA	308989

`

Table 4: Parallel Benchmark results for real 1-Dimensional input vector on 2, 4 processors [17]

N	T_1P	T_2P	T_4P	S_2P	S_4P
256	6.87E-05	3.81E-05	2.69E-05	1.8	2.55
512	2.04E-04	1.12E-04	7.40E-05	1.83	2.76
1024	4.86E-04	2.67E-04	1.70E-04	1.82	2.86
2048	1.17E-03	6.10E-04	3.74E-04	1.91	3.11
4096	2.80E-03	1.35E-03	8.15E-04	2.08	3.44
8192	7.00E-03	3.34E-03	1.95E-03	2.1	3.59
16384	1.81E-02	8.86E-03	5.28E-03	2.05	3.44
32768	4.69E-02	2.39E-02	1.48E-02	1.96	3.17
65536	1.16E-01	5.90E-02	3.67E-02	1.97	3.17

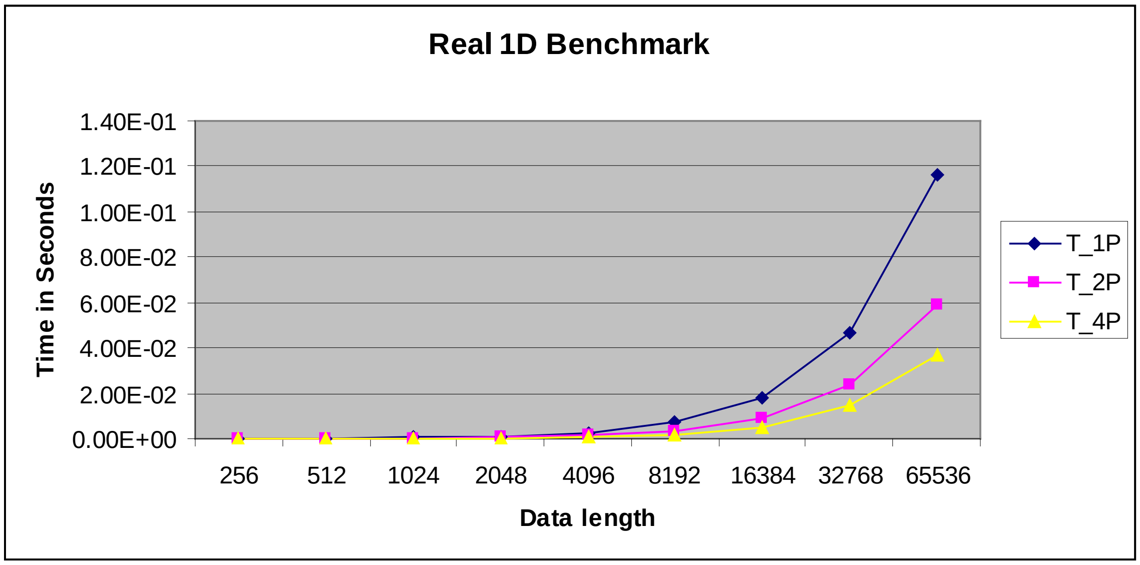


Figure 14 Parallel Benchmark Results on 2, 4 processors [17]

`

8.1 1-Dimensional real and complex case mapped onto 4 processors

The FFTP can be mapped onto four processors easily as well. The odd and even vectors can be computed separately. This splits the computation further into four parts as can be seen in the figure. A point to note is that the processors end up with parts of the output without any communication until the end.

The complex case is slightly different due to the summation in the interleave stage. In the interleave stage, add the sine transform of the real part to the cosine transform of the imaginary part and vice-versa. So each processor should compute sine and cosine transforms which it is going to add later in the interleave stage.

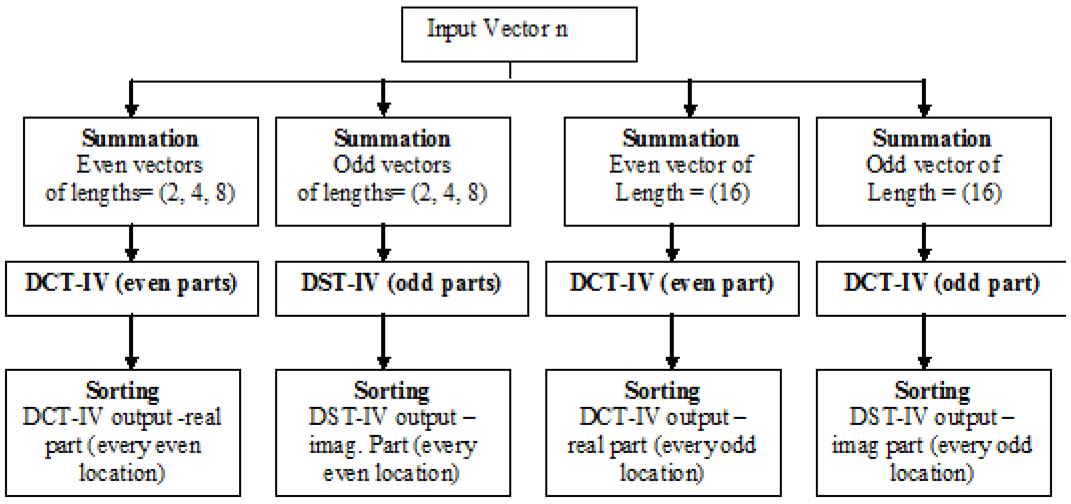

Input Vector n
Summation Even vectors of lengths= (2, 4, 8)
Summation Odd vectors of lengths= (2, 4, 8)
Summation Even vector of Length = (16)
Summation Odd vector of Length = (16)
DCT-IV (even parts)
DST-IV (odd parts)
DCT-IV (even part)
DCT-IV (odd part)
Sorting DCT-IV output -real part (every even location)
Sorting DST-IV output – imag. Part (every even location)
Sorting DCT-IV output – real part (every odd location)
Sorting DST-IV output – imag part (every odd location)

Figure 15: Block Diagram 1-Dimensional real and complex case mapped onto 4 processors

`

CHAPTER IX

RESULTS AND DISCUSSION

9.1 1 D FFT Transform

As we discussed in the chapter IV about the design behind the 2 dimensional Fast Fourier Transform, what we were going to do is take 1 dimensional Fast Fourier Transform of all the rows , then store all the values of transform back in to the 2 dimensional image array. After that we would take 1 dimensional Fast Fourier Transform of the column which will complete 2 dimensional Fast Fourier Transform. In this entire computation, the 1 dimensional Fast Fourier Transform plays the most important role. So we first designed and analyzed the 1 dimensional Fast Fourier Transform for two cases: N=16 and N=128 length input data. In each case, these computations involved detailed cases depending up on the nature of program flow and the number of processor used to process the data. So we have total following cases:

a. N=16, Processor 4 Parallel using MPI
b. N=16, Processor 1 Serial using MPI
c. N=16, Processor 1 Serial without MPI
d. N=128, Processor 4 Parallel using MPI

`

e. N=128, Processor 1 Serial using MPI

f. N=128, Processor 1 Serial without MPI

We compared these cases against time and tried to analyze each based on its efficiency to calculate the transform. The following data was collected while doing the analysis.

Table 5: Case a-One Dimensional FFT Timings n=16 without MPI and Parallelizing with 1 Processors

No	Time(ms)
1	25
2	20
3	25
4	25
5	30
6	25
7	20
8	20
9	30
10	25

`

Table 6: Case b- One Dimensional FFT Timings n=16 with MPI -without Parallelizing 1 processor

No	Time(us)
1	82.02
2	80.01
3	86.92
4	80.11
5	79.95
6	81.94
7	84.01
8	72.92
9	78.99
10	83.98

`

Table 7: Case c- One Dimensional FFT Timings n=16 with MPI and Parallelizing 4 processor

Iteration No	Total Time (us)	Individual Processor timings			
		0	1	2	3
1	422.96	78.1136	74.07	63.93	42.96
2	404.995	83.07	270.98	69.09	43.92
3	398.039	82.09	264.008	66.88	44.07
4	419.88	80.99	87.06	58.89	44.03
5	429.05	83.93	73.93	73.07	42.9
6	409.93	75.9	82.917	62.95	44.05
7	423.923	80.96	80.1	63.03	43.96
8	426.02	81.99	74.002	63.94	43.92
9	417.06	77.09	71.07	60.11	44.02
10	407.97	81.93	83.06	59.95	43.99
11	458.03	34.92	109.026	91.98	38.97
12	454.034	32.95	104.039	96.98	38.038
13	445.89	40.06	110.07	89.9	37.9
14	458.04	47.92	101.91	107.06	37.99
15	460.91	30.99	114.08	99.97	37.99
16	470.92	33	108.97	97.95	38.99
17	431	27.11	100.98	98.06	37.89
18	469.9	44.08	110.89	89.89	36.93
19	461.03	35.99	111.91	88.92	37.88
20	494.1	32.1	101.91	112.09	39.05

`

Table 8: Case d One Dimensional FFT Timings n=128 without MPI and Parallelizing 1 Processor

No	Time (ms)
1	100
2	105
3	100
4	100
5	100
6	105
7	100
8	100
9	100
10	105

Table 9: Case e One Dimensional FFT Timings n=128 with MPI -without Parallelizing 1 processor

No	Time (us)
1	138.89
2	139.11
3	137.04
4	137.91
5	137.06
6	133.89
7	137.93
8	139.09
9	133.96
10	140.05

`

Table 10: Case f- One Dimensional FFT Timings n=128 with MPI and Parallelizing 4 Processor

Iteration	Total	Individual Processor Timings			
No	Time (us)	0	1	2	3
1	517.05	87.91	354.88	100.99	49.93
2	543.06	83.05	147.92	96.1	47.95
3	514.92	86.88	358.07	102.951	48.95
4	537.91	80.94	145.96	94.05	49.99
5	530.03	80.03	365.88	107.05	49.04
6	528.97	81.97	137.012	98	47.93
7	529.99	85.07	356.39	112.918	47.99
8	532.96	87.02	171.97	110.1	48.1
9	533.92	87.11	361.94	103.05	49.06
10	517.91	85	362.91	108.07	50.02

`

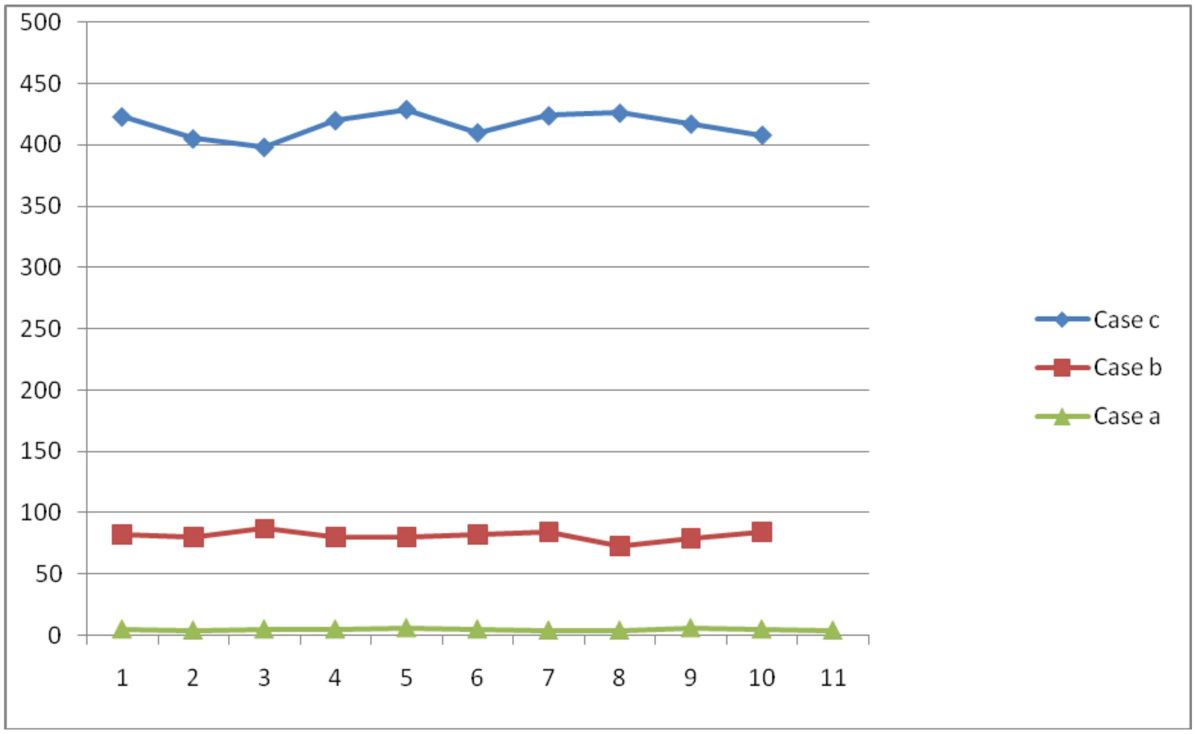


Figure 16: Different case results for input data size N=16

`

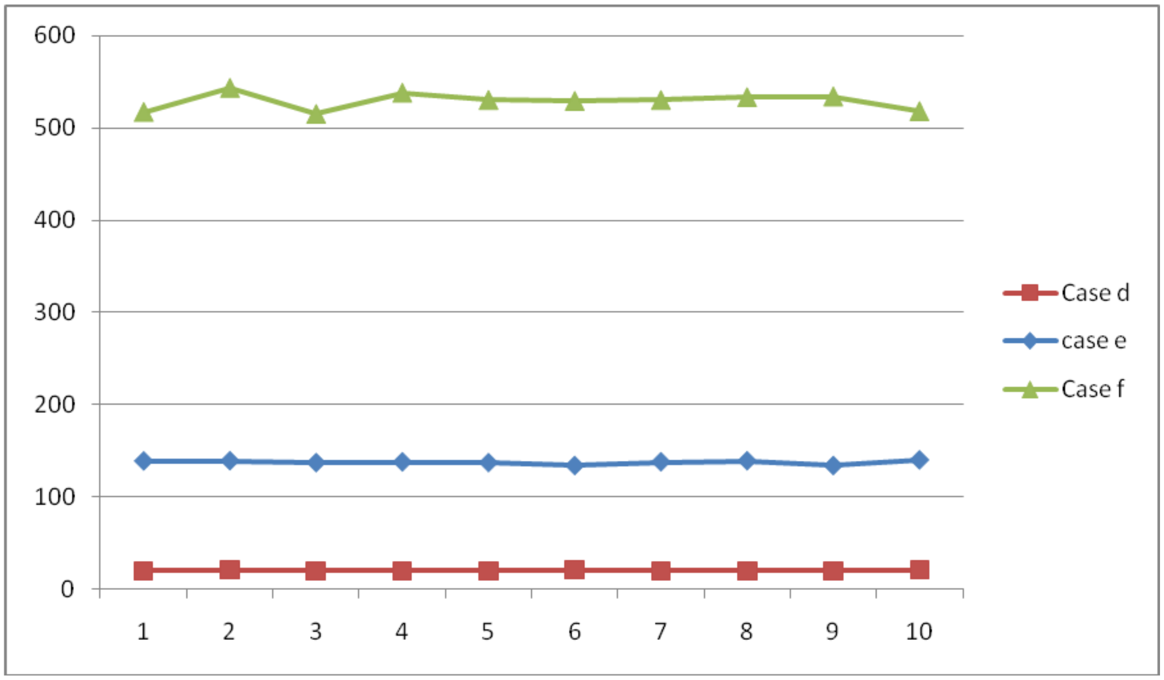


Figure 17: Different case results for input data size N=128

9.2 2-Dimensional Case Dilemma

While dealing with the 2 dimensional case, we need to take the 1 dimensional Fast Fourier Transform of all the rows one at a time and then take the 1 dimensional Fast Fourier Transform of all the resulting columns. To develop a prototype, we did this computation in MATLAB, checked against the built-in function for the FFT. It gave the same accuracy of the magnitude. But while dealing with the same in the C language, we had some issues. The C language is not designed like MATLAB to deal exclusively with mathematical issues. It is a higher level computer language and has some basic mathematical functions. In the process of computing the 2 Dimensional Fast Fourier

`

Transform of the image, it is necessary to read an image as an array of values with rows and columns, then access those rows and columns individually and take the Fast Fourier Transform. In all these things, it was hard to access the image array and then manipulate it row by row and then column by column in the C language. We tried all the things like reading the image array as a text file or a csv file then manipulate it. But then the only way out of this situation was to enter each row or column each time we want to take Fast Fourier Transform. This sounds easy when we are dealing with small image array like 16 * 16 array. But as the image size increases, it becomes computationally impossible. So what we thought is that we will take 1 D Fast Fourier Transform of one row then column and to establish efficiency of our algorithm, we will simply calculate from that.

9.3 CORDIC Function Problem

Our second approach was to implement the CORDIC Function for mathematical functions. As explained before the CORDIC Function uses the bit shift method to calculate the mathematical multiplications rather than the usual approach. We believed that it had the potential to reduce the time required to do the mathematical computation. We designed a program for rotation and calculation of few mathematical computations using the CORDIC Function. When we executed the program and recorded the time required, we came to know that with the present hardware cluster of processors at the university, it was hard to determine the timings in microseconds. When we executed the 1 dimensional Fast Fourier Transform program one processor case, timings were in

`

microseconds. To determine the timings for part of such program wouldn’t be practical. So we had to drop the idea of using the CORDIC Function.

9.4 Accuracy

To test the accuracy of the C program, output values of the Fast Fourier Transform computation in C language were compared to the output values of MATLAB program that we developed earlier. These matched to many degrees of precision, showing that the c program was computing correct results.

9.5 Comparison with previous results

This section compares our results with the results of Misal [10] and Mirza [19], previous graduate students at the University of Akron. As mentioned before, we applied the new approach to the input real valued data vector length N=16 and N=128. In Misal’s work and Mirza’s work, they considered different input data sizes. The only case that was matching is N=128 of Mirza’s work. So here we are summarizing the comparison of both results.

Mirza tried his approach on the University of Akron and OSC cluster. Following is the result from the University of Akron cluster required for one run of the Fast Fourier Transform computation. The new approach column in the following table gives the time required for one run of computation with the algorithm we developed running on one processor.

`

Table 11: Comparison results N=128 one processor case

New Approach	Mirza's Result
100 us	113 us

Table above shows that when the input data size is 128 with 1 processor, Mirza’s approach took 113 microseconds for one run of Fast Fourier Transform computation while our new approach took 100 microseconds for the same. Therefore the new approach we implemented appears to make the computation of Fast Fourier Transform faster than Mirza’s approach. There can be some other factors which are responsible for difference such as difference in hardware, number of times iterations taken, operating systems etc.

`

CHAPTER X

CONCLUSIONS

The goal of the research was to develop a new algorithm for the 2 dimensional Fast Fourier Transform which is used in the reconstruction of the images in MRI using parallel processing. For that purpose, we first studied the previous developments in the same area of research. Two of the students of Dr. Mugler had already worked on the same type of research with different approaches, and there was FFTW algorithm developed by a group in MIT. We analyzed the works of these two students, and then we developed our own new approach to the same problem computing FFT using parallel processing. For that we considered two major cases depending up on the size of input, N=16 and N=128.

While studying previous development in this research area, we came to the conclusion that the algorithm implemented by Misal had efficient timings. The idea behind Misal’s code was to execute individual subroutines for different sizes of input data. It doesn’t execute the whole algorithm each time, it just intelligently computes part of the entire code depending up on the input data size. This paper described another approach. While studying previous algorithms, we reached the interesting conclusion that, in the computation of the Fast Fourier Transform, the most time consuming part is the calculation of sine, cosine parts and their use in mathematical rotation. We decided to

`

deal with that in our new approach. We determined that we would provide the sine, cosine components calculation as a pre-computed resource to the program. That is, we calculate all the sine, cosine components and store in the processor memory and just access those components whenever there is a need for those values in the program. That is how we designed algorithm and analyzed its efficiency. For that we considered different cases based on the number of processors used to execute the program, data size of input, whether the computation is parallelized or not.

In the N=16 case, the input data size is 16. For this case, we considered three different settings. The first was with four processors and using MPI in parallel, the second was one processor using MPI and the third is one processor without implementing MPI. We calculated the timings for each setting with 10 iterations. In the first setting, four processors with implementing MPI in parallel, we computed timings for each processor as well to get the idea of how much time each processor takes. We had the same setting for another case with input data size N=128.

When we designed the entire Fast Fourier Transform computation algorithm in parallel with completely new approach, we thought it had the potential to compute the transform efficiently. But when we recorded timings for each setting, the results were different than expected. Theoretically with any input data size the case with 4 processors implemented in parallel should have been most efficient while the case with 1 processor without parallelizing should be taking more time. But in contrast to this belief, the case with 4 processors with parallelizing took more time to compute the Fast Fourier Transform while the case with 1 processor without parallelizing turned out to be most efficient for both data input sizes. So the conclusion of entire study was rather than

`

parallel implementation, the sequential execution of the Fast Fourier Transform computation was more efficient with this new approach.

In the future, it will be interesting to study the implementation of this concept in a higher level computer language rather than C such as Java. That may help to deal with the problem of accessing rows and columns in a two dimensional array.

`

BIBLIOGRAPHY

1 Z. P. Liang and P. C. Lauterbur, “Principles of Magnetic Resonance Imaging, A signal processing perspective”, IEEE Press, New York, 2000.

2 D. G. Nishimura, “Principles of Magnetic Resonance Imaging”, April 1996.

3 E. M. Haacle and Z. P. Liang, “Challenges of Imaging Structure and Function with MRI”, *IEEE Transactions on Medicine and Biology*, Vol.19, pp 55 - 62, 2000.

4 “MRI Basics: MRI Basics”. Accessed September 21, 2007, from Website: http://www.cis.rit.edu/htbooks/mri/inside.htm

5 “MRI Physics: MRI Physics”. Accessed May 11, 2008, from Website: http://www.mri-physics.com/

6 M. R. Smith, S. T. Nichols, R. M. Henkelman and M. L. Wood, “Application of Autoregressive Moving Average Parametric Modeling in Magnetic Resonance Image Reconstruction”, *IEEE Transactions on Medical Imaging*, Vol. M1-5:3, pp 257 - 261, 1986.

7 F. J. Harris,“On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform”, Proceedings of the IEEE. Vol. 66, January 1978.

8 Fourier Transform accessed November 2008, http://en.wikipedia.org/wiki/Fourier_transform

9 Z. P. Liang, F. E. Boada, R. T. Constable, E. M. Haacke, P. C. Lauterbur, and M. R. Smith, “Constrained Reconstruction Methods in MR Imaging”, *Reviews of MRM*, vol. 4, pp.67 - 185, 1992.

10 Misal, A Fast Parallel Method of Interleaved FFT for Magnetic Resonance Imaging, The University of Akron, Master’s Thesis [May 2005].

11 Barry Wilkinson and Michael Allen, Parallel Programming- Techniques, and Applications using Networked Workstations and Parallel Computers, 1999, Prentice Hall.

`

12 James W. Cooley and John W. Tukey, An algorithm for the machine calculation of complex Fourier series, Math. Computing 19, pp.297–301, 1965

13 Jon Louis Bentley, Writing Efficient Programs, 1982, Prentice-Hall

14 D.H.Mugler, The New Interleaved Fast Fourier Transform, private communication

15 M.L Wood and R. M. Henkelman, ”Truncation artifacts in magnetic Resonace Imaging ”, *J. Magn. Res. Med.* Vol. 2, 1985.

16 www.fftw.org/fftw-paper.pdf

17 www.fftw.org

18 J. E. Savage, “Space-time tradeoffs in memory hierarchies,” Tech. Rep. CS 93-08, Brown University, CS Dept., Providence, RI 02912, October 1993.

19 Mirza’s Parallel computation of the interleaved Fast Fourier Transform with MPI, The University of Akron, Master’s Thesis [December 2008]

`

APPENDIX

C CODE FOR FAST FOURIER TRANSFORM

```
/* Final FFT n= 16 in parallel   */

 /* Running with Implementation of Parallel Approach Sept 23 , 2009    */


#include<stdio.h>
#include<math.h>
#include <stdlib.h>
#include<mpi.h>
#define PI 3.1415926535


void sumdiff(double v[],int d1);
void sumdiff1(double v[],int d2);
void C4genCS1(double x[],int n);
void fft08a(double x[],int n);    /*  xoutfinal1 []*/
void rr90cs(double pin[],double CS[],double SC[], int n);
void oddhalf08a(double x[],double C[], double S[],int n);
int getlog(int n);
void ord1(int n);
void genC1S1(int n);
void genC2S2(int n);
double y3[64]={0};
double Y3[64]={0};;
double xout[64]={0};
double xouttest[64]={0};
double  ordb[64]={0};
double xoutfinal1[64]={0};
double xoutfinal2[64]={0};
double xoutfinal3[64]={0};
double ord[64]={0};
double pout_final [64]={0};
double x0[64]={0};
double xt[64]={0};
double x1[64]={0};
double x2[64]={0};




int main(int  argc, char *argv[])
       {
              int i,j,m;
              double x[32]={0};
              double x1a[32]={0};
```

`

```
              double x2a[32]={0};
              double sumx[32]={0};
              double diffx[32]={0};



                     int myrank,tag=1234, count,numprocs;
                            double mytime;
                            MPI_Status status;

                            MPI_Init(&argc, &argv);
                            mytime=MPI_Wtime();
                            MPI_Comm_rank (MPI_COMM_WORLD,&myrank);
                            MPI_Comm_size (MPI_COMM_WORLD, &numprocs);

                            int n=16;

                                       double
C[]={0.7071,0.9239,0.9808,0.5556,0.9952,0.8819,0.6344,0.2903,            0.9988,
0.9700,0.9040,0.8032,0.6716,0.5141,0.3369,0.1467,0.9997,0.9925,0.9757,0.9495,

       0.9142,0.8701,0.8176,0.7572,
0.6895,0.6152,0.5350,0.4496,0.3599,0.2667,0.1710,0.0736,0.9999,0.9981,0.9939,0.
9873,0.9783, 0.9670,0.9533,0.9373,0.9191,0.8987,0.8761,

       0.8514,0.8246,0.7958,0.7652,0.7327,0.6984,0.6624,0.6249,0.5858,0.5453,0.5
035,0.4605,0.4164,0.3713,0.3253,0.2785,0.2311,0.1830, 0.1346,0.0858,0.0368};

                                          double    S[]={0.7071,0.3827,    0.1951,
0.8315,0.0980,0.4714,   0.7730,0.9569,0.0491,0.2430,   0.4276,0.5957,   0.7410,
0.8577,                              0.9415,                              0.9892,
0.0245,0.1224,0.2191,0.3137,0.4052,0.4929,0.5758,0.6532,0.7242,
0.7883,0.8449,0.8932,0.9330,0.9638, 0.9853, 0.9973, 0.0123, 0.0613, 0.1102,
                                                                 0.1589,
0.2071,0.2549,0.3020,0.3484,0.3940,0.4386,0.4822,   0.5246,   0.5657,   0.6055,
0.6438, 0.6806, 0.7157,0.7491, 0.7807, 0.8105,0.8382, 0.8640, 0.8876, 0.9092,
0.9285, 0.9456,0.9604,0.9729,0.9831, 0.9909, 0.9963,  0.9993};


                            if (myrank ==0)
                                          {
                                                 printf("\n \t N=16 case with n=4
processors \n");

                                                 printf("\n \t Processor 0 starts
here \n");




                                   int g=1;

                                   for( i=0;i<n;i++)
                                   {
                                          //g=g+1;
                                          x[i]=g;
                                          printf("\n \t i is %d   \t Input array
Index is %g \n ",i,x[i]);
                                          g++;
                                   }
```

`

```
                                    for(i=0,j=n/2;i<n/2 && j<n;i++,j++)
                                         {
                                           x1a[i]=x[i];
                                           x2a[i]=x[j];
                                          // printf("\n \t X2 is %g \n",x2[i]);
                                           sumx[i]=x1a[i]+x2a[i];
                                           diffx[i]=x1a[i]-x2a[i];
                                         }
                                    /*for(i=0;i<n/2;i++)
                                         {
                                            printf("\n    \t    SUMX    is    %g
\n",sumx[i]);
                                         }*/


                                        /* for(i=0;i<n/2;i++)
                                                {
                                                 printf("\n   \t   DIFFX   is   %g
\n",diffx[i]);
                                                }*/


                                                 MPI_Send(sumx,8,MPI_DOUBLE,1,
tag,MPI_COMM_WORLD);
                                                 MPI_Send(diffx,8,MPI_DOUBLE,2,
tag,MPI_COMM_WORLD);
                                  }

                                        if(myrank==1)
                                          {
                                                        printf("\n \t Processor 1
starts here \n");


       MPI_Recv(sumx,8,MPI_DOUBLE,0,tag,MPI_COMM_WORLD, &status);



                                                 fft08a(sumx,n/2);

                                                 /*for(i=0;i<n/2;i++)
                                                              {
                                                                     printf("\n
xoutfinal1 array is %g \t %d \n ",xoutfinal1[i],i);

                                                              }*/

                                        /*     for(i=0;i<n;i++)
                                                        {
                                                              printf("\n       \t
xoutfinal2 ( fft08a ) array is %g  \t %d \n",xoutfinal2[i],i);
                                                        }*/

                                                 MPI_Send(xoutfinal1
,8,MPI_DOUBLE,3, tag,MPI_COMM_WORLD);


                                        }

                                        if (myrank==2)
                                               {
```

`

```
                                                                  printf("\n
\t Processor 2 starts here \n");


     MPI_Recv(diffx,8,MPI_DOUBLE,0,tag,MPI_COMM_WORLD, &status);



                                   oddhalf08a(diffx,C,S,n/2);

                                   /*for(i=0;i<n/2;i++)
                                        {
                                             printf("\n      \t
xoutfinal3 (oddhalf08a) array is %g  \t \t %d \n",xoutfinal3[i],i);
                                        }*/


                                   MPI_Send(xoutfinal3
,8,MPI_DOUBLE,3, tag,MPI_COMM_WORLD);


                                   }

                    if(myrank==3)
                         {
                              printf("\n \t Processor 3 starts here
\n");


     MPI_Recv(xoutfinal1,8,MPI_DOUBLE,1,tag,MPI_COMM_WORLD, &status);

     MPI_Recv(xoutfinal3,8,MPI_DOUBLE,2,tag,MPI_COMM_WORLD, &status);


                              int dq=0;
                              for(i=0,j=1;i<n && j<n;i+=2,j+=2)
                                        {

     xoutfinal2[i]=xoutfinal1[dq];

     xoutfinal2[j]=xoutfinal3[dq];
                                             dq++;
                                        }
                              for(i=0;i<n;i++)
                                   {
                                        printf("\n \t xoutfinal2
array is %g  \t \t %d \n",xoutfinal2[i],i);
                                   }
                              mytime = MPI_Wtime()-mytime;
                              mytime = mytime* 1000000;
                              printf("\n \t  Timing from rank %d is
%lf us \n", myrank, mytime);


                    }
               MPI_Finalize();
               return 0;

     }

void fft08a(double x[], int n)
```

```
`


{

              int i,j,k,l,m,n2,p,nspace,nstart,nc,J,m2,m3,ab,cd,xy,yz;
              n2=n/2;
              int ax=n/4;

              double temp[32]={0};
              double temp1[32]={0};
              double temp2[32]={0};
              double temp3[32]={0};
              double xbothalf[32]={0};
              double pair[32]={0};





         m=n; nspace=2; nstart=2;




     J=getlog(n);

        //printf("\n Int J is %d ",J);

     for(nc=1;nc<(J-1);nc++)
              {

                sumdiff(x,m);

        for(j=0;j<n;j++)
           {
                            x[j]=y3[j];
                     //   printf("\n Array y3 in fft08a is %g \t ",y3[j]);
           }



         m2=m/2; ab=0; m3=m2/2;

              // printf("\m \t the m is %d, m2 is %d, ax is %d ",m,m2,ax);

         for(i=m2;i<m;i++)
           {
                            temp[ab]=y3[i];
               //printf("\n temp in fft08a is %g \t",temp[ab]);
             ab++;
           }
         cd=0;
         for(k=(m2-1);k>=m3;k--)
           {
             temp1[cd]=temp[k]-temp[cd];
             //printf("\n temp1 array in fft08a is %g \t",cd,temp1[cd]);
             cd++;
           }

         yz=m2/2; xy=0;
         for(l=(m3-1);l>-1;l--)
                      {
                             temp2[xy]=temp[l]+temp[yz];
```

`

```
                            //printf("\n  temp2  array  in  fft08a  is  %g  \t  %d
\n",temp2[xy],xy);
                           xy++; yz++;
                           }


                 C4genCS1(temp1,ax);

                 for(i=0;i<m3;i++)
                   {
                     temp1[i]=xout[i];
                                   //printf("\n Array xout in fft08a   is %g \t
%d \n  ",xout[i],i);
                                  }


                 /*for(i=0;i<2;i++)
                   {
                                        printf("\n  Array  xout  in  main   is  %g
",xout[i]);
                        printf("\n array temp1 in fft08a is %g \n ",temp1[i]);
                   }*/

                           C4genCS1(temp2,ax);



                 for(i=0;i<m3;i++)
                   {
                     temp2[i]=xout[i];
                   }
                 /*for(p=0;p<m3;p++)
                                   {

                                         printf("\n  Outout  array  temp2    in
fft08a %g \t %d \n ",temp2[p],p);
                                   }*/

                            for(i=0;i<m3;i++)
                                  {
                                        temp3[i]=temp2[i]+(-1*temp1[i]);
                                  }
                           int i1=0;int i2=m3-1;
                           for(i=0,j=m3;i<(m2/2) && j<m2;i++,j++)
                           {
                                  xbothalf[i]=temp2[i1]-(-1*temp1[i1]);
                                  xbothalf[j]=temp3[i2];
                                  i1++;i2--;
                           }



                           /*for(i=0;i<n/2;i++)
                             {
                                  printf("\n \t xbothalf array in fft08a is %g
\t %d \n ",xbothalf[i],i);
                             }*/



                           ord1(n/2);
```

`

```
                                double ord[64]={0}; int iw=0,A1=0;
                                double ordb1[64]={0};

                                for(i=0;i<m2;i++)
                                        {
                                                ordb1[i]=ordb[i];
                                                //printf("\n ordb1 array in fft08   is
%g \t %d \n ",ordb1[i],i);
                                        }



                                for(i=(nstart-1);i<n;i+= nspace)
                                        {
                                                iw=ordb1[A1];
                                                xoutfinal1[i]=xbothalf[iw];
                                                A1++;

                                        }



                                /*for(i=0;i<n;i++)
                                        {
                                                printf("\n xoutfinal1 array in FOR LOOP
fft08  is %g \t %d \n ",xoutfinal1[i],i);
                                        }*/

                        nstart=nstart+nspace/2;
                        m=m/2; nspace=2*nspace;
                        ax=ax/2;

                        //printf("\n \t nstart is %d nspace is %d and m is %d \n
",nstart,nspace, m);
        }



                double xx[256]={0};

            for(i=0;i<16;i++)
                                        {
                                                xx[i]=x[i];
                                        }
                sumdiff(xx,4);
                 for(i=0;i<4;i++)
                                        {
                                                xx[i]=y3[i];
                                        }

                /*for(i=0;i<2;i++)
                                        {
                                                xx[i]=y3[i];
                                        }*/
                sumdiff(xx,2);
                for(i=0;i<2;i++)
                                        {
                                                xx[i]=y3[i];
                                        }
                /*for(i=0;i<16;i++)
                                        {
```

`

```
                                        printf("\n \t Array xx in fft08a %g \n
",xx[i]);
                                  }*/


              int iw2=0;
              for(i=0;i<=n;i=(i+n/2))
                                  {
                                         xoutfinal1[i]=xx[iw2];
                                         iw2++;
                                  }
              /*for(i=0;i<16;i++)
                                  {
                                         printf("\n $$$$$$$$$$$$$$$$$ xoutfinal1
array in fft08a is %g \n ",xoutfinal1[i]);
                                  }*/



              xx[2]=temp[0]+temp[1];
              xx[3]=temp[0]-temp[1];

              int iw3=2;
              for(i=n/4;i<=n;i=(i+n/2))
                                  {
                                         xoutfinal1[i]=xx[iw3];
                                         iw3++;
                                  }


              /*for(i=0;i<16;i++)
                                  {
                                         printf("\n  xoutfinal1  array  in  fft08a
is %g \t %d \n ",xoutfinal1[i],i);
                                  }*/




}

void oddhalf08a(double x[],double C[], double S[],int n)
       {
              int i,j,k,p,l,m,m2,m3,nc,yz,xy,ax;
              double temp[32]={0};
              double temp1[32]={0};
              double temp2[32]={0};
              double temp3[32]={0};
              double xbothalf[32]={0};

              for(i=0;i<n;i++)
                     {
                            temp[i]=x[i];
                            //printf("\n Temp array is %g  \t %d \n",temp[i],i);
                     }
              m2=n; nc=1; m3=m2/2;

              int cd=0;
                      for(k=(m2-1),i=0;k>=m3 && i<m3;k--,i++)
                        {
                             temp1[i]=temp[k]-temp[i];
```

`

```
                          //printf("\n  temp1 array in fft08a is %g \t  %d \n
",temp1[i],i);
                          cd++;
                     }

                   yz=m2/2; xy=0;
                   for(l=(m3-1);l>-1;l--)
                            {
                                 temp2[xy]=temp[l]+temp[yz];
                                //  printf("\n  temp2  array  in  fft08a  is  %g
\t",temp2[xy]);
                                 xy++; yz++;
                                }


                 C4genCS1(temp1,m3);

                 for(i=0;i<m3;i++)
                   {
                     temp1[i]=xout[i];
                                // printf("\n Array xout in oddhalf08a  is %g
\t %d \n",xout[i],i);
                                }


                 /*for(i=0;i<m3;i++)
                   {
                     printf("\n array temp1 in fft08a is %g \n ",temp1[i]);
                   }*/



                           C4genCS1(temp2,m3);


                 for(i=0;i<m3;i++)
                   {
                     temp2[i]=xout[i];
                   }
                /* for(p=0;p<m3;p++)
                                 {

                                        printf("\n  Array  temp2   in  fft08a  %g
\t %d \n ",temp2[p],p);
                                 }*/
                           /*
                            for(i=0;i<m3;i++)
                                {
                                       temp3[i]=temp2[i]+(-1*temp1[i]);
                                }*/

                           int i1=0;int i2=m3-1;
                           for(i=0,j=m3;i<(m2/2) && j<m2;i++,j++)
                                {
                                       xbothalf[i]=temp2[i1]-(-1*temp1[i1]);
                                       xbothalf[j]=temp2[i2];
                                       i1++;i2--;
                                }



                           /*for(i=0;i<n;i++)
```

`

```
                              {
                                   printf("\n \t xbothalf array in fft08a is %g
\t %d  \n ",xbothalf[i],i);
                              }*/



                            ord1(8);

                            double ord[256]={0}; int iw=0,A1=0;


                            /*for(i=0;i<m2;i++)
                                   {
                                          printf("\n ordb array in fft08   is %g
\n ",ordb[i]);
                                   }*/




                            for(i=0;i<n;i++)
                                   {
                                          iw=ordb[A1];
                                          //printf("\n Index array in fft08   is
%g \n ",iw);

                                          xoutfinal3[i]=xbothalf[iw];
                                          A1++;

                                   }


                            /*for(i=0;i<n;i++)
                                   {
                                          printf("\n xoutfinal3 array is %g \t %d
\n ",xoutfinal3[i],i);
                                   }*/



       }

void C4genCS1(double x [],int n)
      {
                     int i,n2,n3,n4,j,cj,mj,m,k,ab,cd,q,z1;
                     int xin[]={1,2,3,4,5,6,7,8};
                     double c,s,a,b,y,z;
                     double xtemp[256]={0};
                     double x1[128]={0};
                     double x2[128]={0};

                     double P[128]={0};
                     double p1[128]={0};
                     double p2[128]={0};

                     double C2c[64]={0};
                     double S2c[64]={0};
                     double pb[128]={0};
                     double temp2[128]={0};
                     double temp3[128]={0};
                     double temp4[128]={0};
```

```
`


                    //pout_final[]={0};


                    double
C[]={0.7071,0.9239,0.9808,0.5556,0.9952,0.8819,0.6344,0.2903,            0.9988,
0.9700,0.9040,0.8032,0.6716,0.5141,0.3369,0.1467,0.9997,0.9925,0.9757,0.9495,
                                        0.9142,0.8701,0.8176,0.7572,
0.6895,0.6152,0.5350,0.4496,0.3599,0.2667,0.1710,0.0736,0.9999,0.9981,0.9939,0.
9873,0.9783, 0.9670,0.9533,0.9373,0.9191,0.8987,0.8761,

       0.8514,0.8246,0.7958,0.7652,0.7327,0.6984,0.6624,0.6249,0.5858,0.5453,0.5
035,0.4605,0.4164,0.3713,0.3253,0.2785,0.2311,0.1830, 0.1346,0.0858,0.0368};

                    double  S[]={0.7071,0.3827,  0.1951,  0.8315,0.0980,0.4714,
0.7730,0.9569,0.0491,0.2430,  0.4276,0.5957,  0.7410,  0.8577,  0.9415,  0.9892,
0.0245,0.1224,0.2191,0.3137,0.4052,0.4929,0.5758,0.6532,0.7242,
0.7883,0.8449,0.8932,0.9330,0.9638, 0.9853, 0.9973, 0.0123, 0.0613, 0.1102,
                                         0.1589,
0.2071,0.2549,0.3020,0.3484,0.3940,0.4386,0.4822,   0.5246,   0.5657,   0.6055,
0.6438, 0.6806, 0.7157,0.7491, 0.7807, 0.8105,0.8382, 0.8640, 0.8876, 0.9092,
0.9285, 0.9456,0.9604,0.9729,0.9831, 0.9909, 0.9963,  0.9993};



                    //double  C1[]={0.9988, 0.9700, 0.9040, 0.8032,    0.6716,
0.5141,  0.3369,  0.1467,0.9952,0.8819, 0.6344,  0.2903,   0,0,0,0,    0.9808,
0.5556, 0,0,0,0,0,0,   0.9239,0,0,0,0,0,0,0};

                                               //double
C1[]={0.9988,0.9700,0.9040,0.8032,0.6716,0.5141,
       0.3369,0.1467,0.9952,0.8819,0.6344,0.2903,0,0,0,0,                0.9808,
0.5556,0,0,0,0,0,0,0, 0.9239,0,0,0,0,0,0,0};      /* for n=64 */

                                               double      C1[]={0.9952,0.8819,
       0.6344,      0.2903,      0.9808,  0.5556, 0,0,   0.9239,0,0};  /* for
n=32 */


                                               //double  S1[]={0.0491,  0.2430,
0.4276,0.5957,      0.7410,        0.8577,        0.9415,0.9892,        0.0980,
       0.4714,0.7730,0.9569,0,0,0,0,0.1951,0.8315,0,0,0,0,0,0,
0.3827,0,0,0,0,0,0,0 };    /* for n=64 */

                                               double
S1[]={0.0980,0.4714,0.7730,0.9569,       0.1951,0.8315,0,0,0.3827,0,0,0};    /*
for n=32 */



                    /*double  C1[]={0.9997,  0.9925,  0.9757,  0.9495,  0.9142,
0.8701,  0.8176,  0.7572,0.6895,0.6152,  0.5350,0.4496,  0.3599,0.2667,0.1710,
0.0736, 0.9988, 0.9700,0.9040, 0.8032,0.6716, 0.5141, 0.3369, 0.1467,
                           0,0,0,0,0,0,0,0,
0.9952,0.8819,0.6344,0.2903,0,0,0,0,0,0,0,0,0,0,0,0,0.9808,
0.5556,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0.9239,0,0,0,0,0,0,0,0,0,0,0,0,0,0};*/
```

```
`


                    /*double   S1[]={   0.0245,   0.1224,0.2191,0.3137,0.4052,
0.4929,     0.5758,     0.6532,0.7242,0.7883,     0.8449,0.8932,0.9330,0.9638,
0.9853,0.9973,0.0491,0.2430,0.4276, 0.5957, 0.7410,0.8577, 0.9415, 0.9892,
                    0,0,0,0,0,0,0,0,0.0980,                      0.4714,0.7730,
0.9569,0,0,0,0,0,0,0,0,0,0,0,0,0,      0.1951,0.8315,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0.3827,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};                              */


                    printf("\n \t size of input array in C4genCSa is %d ",n);

                    n2=n/2; n3=n/4;
                    int mm2=1;int LL=n2;
                    int m2=1;int L=n2;

                    int j2;
                    /*for(i=0;i<n;i++)
                      {
                        printf("\n Array x in C4genCSa is %g\t %d  \t",x[i],i);
                      }*/

                                  for(i=0,j=n-1;i<n && j>0;i++,j-=2)
                                         {
                                                x0[i]=x[j];
                                                //printf("\n  Array  x0  is  %g  \t
%d \n",x0[i],i);
                                         }

                                  for(i=0,j=1;i<2*n && j<2*n;i++,j+=2)
                                         {
                                                xt[i]=x[i];
                                                //printf("\n  Array  xt  is  %g  \t
%d \n",xt[i],i);
                                         }

                                  for(i=0,j=1;i<n && j<n;i++,j+=2)
                                         {
                                                xt[j]=x0[i];
                                                //printf("\n  Array  xt  is  %g  \t
%d \n",xt[j],j);
                                                //printf("\n  Array  x0  is  %g  \t
%d \n",x0[i],i);
                                         }

                                         /*for(i=0;i<n;i++)
                                           {
                                                printf("\n Array xt is %g \t %d
\n",xt[i],i);
                                           }*/
                                  if(n==2)
                                  {
                                                c=C[1];
                                                s=S[1];
                                                xout[0]=  c*x[0]+s*x[1];
                                                xout[1]=-s*x[0]+c*x[1];
                                                /*for(i=0;i<2;i++)
                                                       {
                                                              printf("\n       \t
!!!!!!!  The array xout in C4genCSa is %f \t  %d   \n",xout[i],i);
                                                       }*/
                                         goto Endofloop;
                                  }
```

`

```
                                  else
                                        {
                                        for(j=1;j<=n/2;j++)
                                                {
                                                    c=C1[j-1];
                                                    s=S1[j-1];
                                                    j2=(2*j)-1;
                                                    //printf("\n j2 is  %d \t
c= %f \t s= %f \n",j2,c,s);

                                                    float NN,MM;

                                                    NN=(c*xt[j2-1]+s*xt[j2]);
                                                    MM=(-s*xt[j2-
1]+c*xt[j2]);

                                                    xt[(j2-1)]=NN;
                                                    xt[(j2)]=MM;


                                                    /*printf("\n  Array  xt  in
C4genCSa is %g \t %d \n",xt[j2-1],j);
                                                    printf("\n   Array   x   in
C4genCSa is %g \t %d \n",xt[j2],j);*/

                                              }

                                                    /*for(i=0;i<n;i++)
                                                      {
                                                          printf("\n    Array
xt is %g \t %d \n",xt[i],i);
                                                      }*/


                                        sumdiff(xt,n);

                                        /*for(i=0;i<n;i++)
                                                {
                                                    printf("\n Array y3 is %g
\t %d \n",y3[i],i);
                                                }*/
                                        for(i=0,j=n/2;i<n/2,j<n;i++,j++)
                                                {
                                                      p1[i]=y3[i];
                                                      p2[i]=y3[j];
                                                      //printf("\n  Array  p1
is %g \t %d \n",p1[i],i);
                                                }




                            int iw=0;
                            if(n==4)
                                  {
                                        for (i=0;i<2;i++)
                                                {
                                                    xout[i]=p1[iw];
```

`

```
                                                    iw++;
                                                  }
                                                c=C[0];
                                                s=S[0];
                                                xout[2]=c*(p2[0]+p2[1]);
                                                xout[3]=s*(-p2[0]+p2[1]);
                                                /*for(i=0;i<4;i++)
                                                       {
                                                              printf("\n       \t
$$$$$$$  The array xout in C4genCSa is %f \t  %d   \n",xout[i],i);
                                                       }*/
                                                goto Endofloop;
                                  }


                           int i1=0;
                           double C2b[64]={0};
                           double S2b[64]={0};

                           int pcounter=1;

                           int J1=getlog(n)-2;
                            int block;


                           for(pcounter=1;pcounter<3;pcounter++)
                                  {
                                                if (pcounter==1)
                                                       {
                                                              for(i=0;i<n;i++)
                                                              {

       P[i]=p1[i1];
                                                                            i1++;
                                                              }

                                                              //printf("\n     \t
@@@@@ PCOUNTER is %d \n",pcounter);



                                  for(j=1;j<J1+1;j++)
                                  {
                                         int L4=L/4;
                                         int i1=0;
                                         for(i=L4/2;i<L4;i++)
                                                       {
                                                              C2c[i1]=C[i];
                                                              S2c[i1]=S[i];

                                                              //printf("\n     \t
C2c is %g \t %d  \n",C2c[i1],i1);
                                                              //printf("\n     \t
S2c array is %g \t %d \n",S2c[i1],i1);
                                                              i1++;
                                                       }


                                                for(k=1;k<(m2+1);k++)
                                                       {
```

`

```
                                                                    int
kk=(1+L*(k-1));

       //printf("\n \t $$$$$ L is %d \t m2 is %d \n",L, m2);

                                                                    int i5=0;
                                                                    for(i=kk-
1;i<(L*k);i++)
                           {
                                  block=i;
                                  //printf("\n  \t  block  is  %d  \t  k  is  %d
\n",block,k);
                                  pb[i5]=P[i];
                                  i5++;
                                                                           }

       /*printf("\n \t pb array is %g \t %d \n",pb[0],0);

       printf("\n \t pb array is %g \t %d \n",pb[1],1);*/




sumdiff(pb,L);


       for(i=0;i<L;i++)
                                  {
                                         temp2[i]=y3[i];
                                         //printf("\n \t temp2 array is %g \t %d
\n",temp2[i],i);
                                         }

                                  int ie=0;

       for(i=L/2;i<L;i++)
                           {
                                         temp3[ie]=temp2[i];
                                  //printf("\n  \t  temp3  array  is  %g  \t  %d
\n",temp3[ie],ie);
                           ie++;
                           }

       //printf("\n \t @@@@@ L is %d \n",L);

       rr90cs(temp3,C2c,S2c,L/2);
                    int iww=0;

       for(i=L/2;i<L;i++)
                    {
                                  temp2[i]=pout_final[iww];
                                  //printf("\n  \t  temp2  array  is  %g  \t  \t  %d
\n",temp2[i],i);
                                  iww++;
                    }

                                                                           int
iq=0;

       for(i=(1+(L)*(k-1));i<=(L*k);i++)
                                                                           {
```

```
`


                    P[i-1]=temp2[iq];
              //printf("\n array P in subroutine C4genCSa  is %g \t i is %d \t
J1 is %d\n",P[i], i, J1);
                    iq++;
             }

      /*for(i=0;i<n/2;i++)
             {
      printf("\n \t The array P in C4genCSa is %f \t i is %d \n",P[i],i);
             }*/

                                                              }




                                                L=L/2; m2=2*m2;

                                                //printf("\n \t $$$$$$ m2 is %d
\t L is %d \t j is %d \t k is %d \n ",m2,L,j,k);

                                                /*if (L<5)
                                                {
                                                       break;
                                                }*/

                                  }




                                   }
                                          /*
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ */


                                                else
                                                       {
                                                                     {
                                   int q1=0

      for(i=n/8;i<n/4;i++)
             {
                           C2b[q1]=C[i];
                           S2b[q1]=S[i];
                           /*printf("\n \t C2b is %g \t %d  \n",C2b[q1],q1);
                           printf("\n    \t    S2b    array    is    %g    \t    %d
\n",S2b[q1],q1);*/
                    q1++;
                                                                      }



      rr90cs(p2,C2b,S2b,n/2);


      for(i=0;i<n/2;i++)
{
              P[i]= pout_final[i];

//printf("\n \t P array in ELSE LOOP is %g \t %d \n",P[i],i);
```

```
`



}

       //     printf("\n \t @@@@@ PCOUNTER is %d \n",pcounter);





                            int jw;
                            for(jw=1;jw<(J1+1);jw++)
                                   {

                                          int LL4=LL/4;
                                          int i1=0;
                                          int kj;
                                          for(i=LL4/2;i<LL4;i++)
                                                        {
                                                               C2c[i1]=C[i];
                                                               S2c[i1]=S[i];

                                                               /*printf("\n     \t
C2c is %g \t %d  \n",C2c[i1],i1);
                                                               printf("\n  \t  S2c
array is %g \t %d \n",S2c[i1],i1);*/
                                                               i1++;
                                                        }


                                                 for(kj=1;kj<=mm2;kj++)
                                                        {

                                                                      int
kk=(1+LL*(kj-1));

       //printf("\n \t $$$$$ LL is %d \t mm2 is %d \n",LL,mm2);

                                                                      int i5=0;
                                                                      for(i=kk-
1;i<(LL*kj);i++)
              {
                     block=i;
              //printf("\n \t block is %d \t k is %d \n",block,k);
              pb[i5]=P[i];
                     i5++;
              }



sumdiff(pb,LL);


       for(i=0;i<LL;i++)
       {
              temp2[i]=y3[i];

       //printf("\n \t temp2 array is %g \t %d \n",temp2[i],i);
       }
              int ie=0;

       for(i=LL/2;i<LL;i++)
              {
```

`

```
        temp3[ie]=temp2[i];
                //printf("\n \t temp3 array is %g \t %d \n",temp3[ie],ie);
                      ie++;
               }


        rr90cs(temp3,C2c,S2c,LL/2);
                      int iww=0;

        for(i=LL/2;i<LL;i++)
                      {
                                    temp2[i]=pout_final[iww];
                      //printf("\n \t temp2 array is %g \t %d \n",temp2[i],i);
                             iww++;
                      }


               int iq=0;

        for(i=(1+(LL)*(kj-1));i<=(LL*kj);i++)
                      {
                                    P[i-1]=temp2[iq];
                              //printf("\n array P in subroutine C4genCSa   is %g
\t i is %d \t J1 is %d\n",P[i], i, J1);
                             iq++;
                      }

        /*for(i=0;i<n/2;i++)
                      {
                             printf("\n  \t  The  array  P  in  second  for  loop  in
C4genCSa is %f \t i is %d \n",P[i],i);
                      }*/
                                                          }



                                                    LL=LL/2; mm2=2*mm2;

                                                    //printf("\n \t  mm2 is %d \t LL
is %d \t jw is %d \t kj is %d \n ",mm2,LL,jw,kj);

                                                    /*if (L<5)
                                                    {
                                                           break;
                                                    }*/
                                                    /*int iww=0;
                                                    for(i=(1+(1)*(n/2));i<=n;i++)
                                                           {
                                                                  xout[i-1]=P[iww];
                                                                  P[iww]=0;
                                                                  iww++;
                                                           }*/


                                     }

                               }
               }

}
```

`

```
                            int iww=0;
                            for(i=(1+(pcounter-1)*(n/2));i<=(pcounter*n/2);i++)
                                                   {
                                                          xout[i-1]=P[iww];
                                                          iww++;
                                                   }

                            /*for(i=0;i<64;i++)
                                   {
                                          printf("\n  \t  The  array  P  in  C4genCSa
is %f \t  %d  \n",P[i],i);
                                   }*/

                            /*for(i=0;i<n;i++)
                                   {
                                          printf("\n \t ******  The array xout in
C4genCSa is %f \t  %d   \n",xout[i],i);
                                   }*/
                     Endofloop:printf("\n \t End of C4genCSa Subroutine \n ");
        }
}


         void sumdiff(double v[],int d1)
     {
       int i,j,k,m,n,n1;
       int a,xy,yz;


               /*double x1[64]={0};
       double x2[64]={0};*/

       n=d1; n1=n/2;


       for(i=0;i<n1;i++)
         {
           x1[i]=v[i];
         }
       a=0;
       for(j=n1;j<n;j++)
         {
           x2[a]=v[j];
           a++;
         }
       for(k=0;k<n1;k++)
         {
           y3[k]=x1[k]+x2[k];
         }

       xy=0; yz=0;
       for(m=n1;m<n;m++)
         {
           y3[m]=x1[xy]-x2[yz];
           xy++;yz++;
         }
       }



int getlog(int n)
   {
```

`

```
     if(n==4)return(2);
        if(n==8)return(3);
     if(n==16)return(4);
     if(n==32)return(5);
     if(n==64)return(6);
        if(n==128)return(7);
        if(n==256)return(8);
     else return(0);
   }

void ord1(int n)
    {
      double veca[256];
      veca[0]=0;
      veca[1]=2;
      veca[2]=1;
      veca[3]=3;

         double vecb[256]={0};
         double z1[256]={0};
      int klev,i,j,nv,s1,k,J;
      nv=2;

      J=getlog(n);
      //printf("\n \t J in subroutine ORD1 is %d ",J);

          if(J==2)
              {
                     for(j=0;j<n;j++)
                     {
                       ordb[j]=veca[j];
                      // printf("\n \t Array ordb in subroutine %g ",ordb[j]);
                     }
              }

         else {

                       for(klev=1;klev<=J-2;klev++)
                            {
                              nv*=2;

                              for(i=0;i<=(n/2)-1;i++)
                                   {
                                     vecb[i]=2*veca[i];
                                   }
                              s1=(2*nv)-1;


                             for(i=0;i<nv;i++)
                                   {
                                     z1[i]=s1-vecb[i];

                                   }
                              for(i=nv,k=nv-1;k>=0 && i<(2*nv);k--,i++)
                                   {
                                     vecb[i]=z1[k];
                                     //    printf("\n    \t    array    vecb    is
%g",vecb[i]);
                                   }
                              for(i=0;i<(nv*2);i++)
                                   {
                                     veca[i]=vecb[i];
```

`

```
                                //printf("\n array veca is %g",veca[i]);
                              }

                       }
                  for(j=0;j<n;j++)
                       {
                         ordb[j]=vecb[j];
                         //printf("\n  \t  Array  ordb  in  subroutine  %g
",ordb[j]);
                       }
         }
    }

 void rr90cs(double pin[],double CS[],double SC[], int n)
              {
                       int i,j,k,l,m,mj,xy,ab,cd,uv;

                       /*double C[]= {0.7071,0.9239,0.9808,0.5556};
                       double S[]={0.7071,0.3827,0.1951,0.8315};*/
                       /*double C[]={0.7071,0.9239};
                       double S[]={0.7071, 0.3827};*/
                       /*double      C[]={0.7071,      0.9239,0.9808,0.5556,
0.9952,0.8819,0.6344,0.2903};
                       double
S[]={0.7071,0.3827,0.1951,0.8315,0.0980,0.4714,0.7730,0.9569};*/
                       int temp[64]={0};
                       double c,s,a,b,x,y;
                       double pout[32][2]={{0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},
                                                                  {0,0},};

                              m=n;
```