

PARALLEL COMPUTATION OF THE INTERLEAVED FAST FOURIER  
TRANSFORM WITH MPI

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Ameen Baig Mirza

December, 2008

PARALLEL COMPUTATION OF THE INTERLEAVED FAST FOURIER  
TRANSFORM WITH MPI

Ameen Baig Mirza

Thesis

Approved:

---

Advisor  
Dr. Dale H. Mugler

---

Co-Advisor  
Dr. Tim O'Neil

---

Committee Member  
Dr. Kathy J. Liszka

---

Committee Member  
Dr. Wolfgang Pelz

Accepted:

---

Department Chair  
Dr. Wolfgang Pelz

---

Dean of the College  
Dr. Ronald F. Levant

---

Dean of the Graduate School  
Dr. George R. Newkome

---

Date

## ABSTRACT

Fourier Transforms have wide range of applications ranging from signal processing to astronomy. The advent of digital computers led to the development of the FFT (Fast Fourier Transform) in 1965. The Fourier Transform algorithm involves many add/multiply computations involving trigonometric functions, and FFT significantly increased the speed at which the Fourier transform could be computed. A great deal of research has been done to optimize the FFT computation to provide much better computational speed.

The modern advent of parallel computation offers a new opportunity to significantly increase the speed of computing the Fourier transform. This project provides a C code implementation of a new parallel method of computing this important transform. This implementation assigns computational tasks to different processors using the Message Passing Interface (MPI) library. This method involves parallel computation of the Discrete Cosine Transform (DCT) as one of the parts. Computation on two different computer clusters using up to six processors have been performed, results and comparisons with other implementations are presented.

## ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr Dale Mugler for assigning me this project and his constant support and co-operation until the project completion.

I would also like to thank my co-advisor Dr. Tim O'Neil for his guidance and support without him the project couldn't have been implemented in parallel, I would also like to thank Dr. Kathy Liszka and Dr. Wolfgang Pelz for their time and effort and especially for their valuable suggestions on parallelizing the Fast Fourier transform.

I would also like to thank my friends Mahesh Kumar, Radhika Gummadi, and Venkatesh Pinapala who helped me during the implementation and final phases of this project.

A special thanks to OSC (Ohio Supercomputer Center) for making the FFT to work on supercomputer machines which helps me to attain more accurate and optimized results.

Finally, thanks to my family who were always with me supporting me to achieve better results and I think without their support I would have been lost. Mom and Dad, I would not have made this success without you.

## TABLE OF CONTENTS

|   | Page |
|---|------|
| LIST OF TABLES .....                              | viii |
| LIST OF FIGURES .....                             | ix   |
| CHAPTER   |      |
| I. INTRODUCTION .....                             | 1    |
| 1.1 Discrete cosine transform (DCT).....          | 1    |
| 1.2 Fast Fourier transforms (FFT) .....           | 2    |
| 1.3 Message passing interface (MPI).....          | 2    |
| 1.4 Contributions and outline.....                | 3    |
| II. LITERATURE REVIEW .....                       | 5    |
| 2.1 Fastest Fourier Transform in the West.....    | 6    |
| 2.2 Carnegie Mellon University spiral group ..... | 7    |
| 2.2.1 DFT IP Generators .....                     | 8    |
| 2.2.2 DCT IP Generators .....                     | 8    |
| 2.3 Cooley-Tukey FFT algorithm .....              | 9    |
| 2.4 Summary .....                                 | 10   |
| III. MATERIALS AND METHOD.....                    | 11   |
| 3.1 DCT using the gg90 algorithm .....            | 11   |
| 3.2 DCT using the lifting algorithm.....          | 14   |

|   |    |
|---|----|
| 3.2.1 DCT using the lifting algorithm for 8 data points .....                   | 15 |
| 3.3 Fast Fourier Transform .....  | 16 |
| 3.3.1 FFT using the gg90 algorithm.....   | 16 |
| 3.4 Construction of n=8 point FFT in parallel .....                             | 18 |
| 3.5 FFT using 16 data point .....   | 20 |
| 3.6 Summary .....   | 21 |
| IV. RESULTS AND DISCUSSION .....  | 22 |
| 4.1 Hardware configuration of OSC machine.....                                  | 22 |
| 4.2 Hardware configuration of the Akron cluster .....                           | 23 |
| 4.3 Discrete cosine transforms .....  | 23 |
| 4.3.1 DCT using the lifting algorithm.....                                      | 23 |
| 4.3.2 Comparison of the lifting algorithm on UA and OSC using 1 processor ..... | 25 |
| 4.4 Comparison of the gg90 and lifting algorithm.....                           | 26 |
| 4.5 Fast Fourier transforms .....   | 28 |
| 4.5.1 Real case FFT using 1 processor .....                                     | 28 |
| 4.5.2 Comparisons of the real case FFT using 1 processor .....                  | 29 |
| 4.5.3 Complex FFT using 2 processor .....                                       | 30 |
| 4.5.4 Comparisons of the complex case FFT using 2 processor.....                | 32 |
| 4.5.5 Complex case FFT using 6 processor .....                                  | 34 |
| 4.5.6 Comparison of complex case FFT using 1, 2 and 6 processor .....           | 36 |
| 4.5.7 Comparison of complex case FFT in parallel with FFTW 3.2 .....            | 37 |
| 4.6 Summarys .....  | 39 |

|  |    |
|--|----|
| V. CONCLUSION.....                                   | 40 |
| 5.1 Future work.....                                 | 40 |
| REFERENCES .....                                     | 41 |
| APPENDICES .....                                     | 43 |
| APPENDIX A. TABLES SHOWING THE ACTUAL TIMINGS .....  | 44 |
| APPENDIX B. C CODE FOR FAST FOURIER TRANSFORMS ..... | 50 |

## LIST OF TABLES

| Table   | Page |
|---|------|
| 2.1 Operation counts for DFT and FFT .....                                      | 6    |
| 4.1 Comparing DCT lifting algorithm on 1, 2 and 4 processors .....              | 24   |
| 4.2 Comparing the lifting algorithm at UA and OSC on 1 processor.....           | 25   |
| 4.3 Comparing the gg90 and lifting algorithm at UA cluster on 1 processor ..... | 27   |
| 4.4 Real case FFT using 1 processor .....                                       | 28   |
| 4.5 Comparison of real case FFT on 1 processor.....                             | 29   |
| 4.6 Complex case FFT on 2 processor.....  | 32   |
| 4.7 Comparing complex case FFT using 2 processor .....                          | 33   |
| 4.8 Complex case FFT on 6 processor.....  | 35   |
| 4.9 Complex case FFT on 1, 2 and 6 processors .....                             | 36   |
| 4.10 Comparison of FFT and FFTW3.2 .....  | 38   |

## LIST OF FIGURES

| Figure  | Page |
|---|------|
| 2.1 N=8 point decimation in frequency FFT algorithm.....      | 10   |
| 3.1 gg90 formula for calculating cosine and sine values.....  | 12   |
| 3.2 Sum-difference for four input data points.....            | 12   |
| 3.3 Last steps in DCT.....                                    | 13   |
| 3.4 DCT for 8 data points .....                               | 13   |
| 3.5 Lifting step for two data points.....                     | 14   |
| 3.6 DCT using lifting step for 8 data points.....             | 15   |
| 3.7 Sum-difference operation for the input data points .....  | 17   |
| 3.8 FFT for n=8 data points .....                             | 19   |
| 3.9 FFT for n=16 data points .....                            | 20   |
| 4.1 Comparing lifting algorithm on 1, 2 and 4 processors..... | 24   |
| 4.2 Comparing lifting algorithm on 1 processor.....           | 26   |
| 4.3 Comparing gg90 and lifting algorithm on 1 processor ..... | 27   |
| 4.4 Real case FFT on 1 processor .....                        | 29   |
| 4.5 Comparison of real case FFT on 1 processor.....           | 30   |
| 4.6 Implementation of FFT on 2 processor.....                 | 31   |
| 4.7 Complex case FFT on 2 processor.....                      | 32   |
| 4.8 Comparison of complex case FFT using 2 processor .....    | 33   |

|      |   |    |
|------|---|----|
| 4.9  | Implementation of FFT on 6 processor.....     | 34 |
| 4.10 | Complex case FFT on 6 processor.....          | 35 |
| 4.11 | Complex case FFT on 1, 2 and 6 processor..... | 37 |
| 4.12 | Comparison of FFT and FFTW 3.2.....           | 38 |

# CHAPTER I

## INTRODUCTION

The discrete Fourier transform has a wide range of applications. More specifically it is used in signal processing to convert the time domain representation of a signal to the frequency domain. However the process of conversion is very expensive. Hence an alternate way to compute the discrete Fourier transform is to use the Fast Fourier Transform (FFT). This project deals with a new idea of solving the FFT by dividing the whole problem into parallel blocks and assigning them to parallel nodes to obtain better timings.

### 1.1 Discrete Cosine Transform (DCT)

The DCT is central to many kinds of signal and image processing applications, particularly in video compression. The DCT divides an image into discrete blocks of pixels each block of pixels has a different importance, for any given finite set of data points from a real world signal. Similar to the FFT, the DCT transforms a signal or image from a spatial domain to the frequency domain. It does so by expressing a function or signal in terms of sums of sinusoidal waveforms that vary in amplitude and frequency.

In particular, there exists a DCT for every Fourier related transform, but the DCT can only be used only for real data points. There are eight standard discrete cosines transform, the most common variant of this is being the type-II DCT, which is referred

to as simply DCT. In this thesis we try to build a type-IV DCT. The difference between the type-II and type-IV DCT is that the type-II DCT will generate two data points for the given input whereas the type-IV DCT will generate blocks of four data points for given input data points.

## 1.2 Fast Fourier Transform (FFT)

The main reason why the FFT came into use is to compute discrete Fourier transforms. It is an efficient algorithm to compute discrete Fourier transforms with a complexity of  $O(N \log N)$ , where  $N$  is the number of data points, as compared to complexity of  $O(N^2)$ . For any given finite set of data points taken from a real-world signal, the FFT expresses the data points into their component frequencies. It is also useful in solving the major inverse problem of reconstructing a signal from given frequency data. The FFT are also of great importance in a wide variety of applications including digital signal processing, solving partial differential equations and quick multiplication of large integers. The FFT is also known to be the fastest algorithm to multiply two polynomials.

## 1.3 Message Passing Interface (MPI)

There is a continual demand for greater computational speed from a computer system than is currently possible [2]. There are some specific applications like weather forecasting, manufacturing applications, engineering calculations and simulations, which must be performed quickly. High-speed systems are greatly needed in these areas. One way to increase the computational speed is to use multiple processors to solve a problem.

The problem is split into parts, each of which is performed by a separate processor in parallel. When the multiple processors work in parallel they need an interface by which they can communicate. The MPI is a library used by multiple processors to send messages back and forth using send and receive commands. This approach provides a significant increase in performance.

MPI's goals are performance, scalability and portability. These features make MPI the most dominant model used in high performance computing today. It has become the de facto standard for communication between different processors both for shared memory and distributed memory. MPI-1 is the standard for traditional message-passing using shared memory between different processors, whereas MPI-2 is standard for remote memory, with parallel input/output and dynamic processing using distributed memory for different processors

In this thesis we attempt to compute the FFT in parallel. We design the algorithm in such a way that the problem is split into parts with each part executed in parallel and the final result gathered at the end. We use the MPI library to communicate between multiple processors.

#### 1.4 Contributions and Outline

In this research, we present the following contributions that are implemented in the course of designing FFT.

1. We implemented the DCT and FFT. The DCT is built using two new approaches, a gg90 algorithm and a lifting algorithm.
2. We describe a different ways of implementing the FFT by making the FFT run in parallel with the DCT, thus making the entire Fourier transform run in parallel.

The rest of the thesis is organized as follows

1. Chapter 2 will give detailed information on DCT, FFT and the MPI library. It also talks about the implementation of FFT by the “fast Fourier transform in the west”. Mellon University’s “spiral group”, which has the best timings for the DCT and FFT, will also be discussed.
2. Chapter 3 will describe the algorithms pertaining to the DCT which we implemented by using two different algorithms. We use the naming conventions the “gg90 algorithm” and the “lifting algorithm”. We also explain how we are implementing the FFT in parallel and embedding the DCT along with FFT.
3. Chapter 4 describes the results which we have obtained by implementing the DCT using the gg90 algorithm and the lifting algorithm, followed by FFT results which we have obtained by making it run on different processors. Here we also take an opportunity to explain why the gg90 algorithm is preferred in constructing the FFT.
4. Chapter 5 describes the conclusions and future work. It also suggests new ways of obtaining better timings for the FFT and also provides enhancements that can be made to this algorithm.

## CHAPTER II

### LITERATURE REVIEW

Discrete Fourier transforms are primarily used in signal processing. They are also used in processing information stored in computers, solving partial differential equations, and performing convolutions. The discrete Fourier transform can be computed efficiently using the FFT algorithm.

The FFT has various applications including digital audio recording and image processing. FFTs are also used in scientific and statistical applications, such as detecting periodic fluctuations in stock prices and analyzing seismographic information to take “sonograms” of the inside of the Earth [3]. Due to the vast usage of FFT different algorithms have been developed over time. We will discuss some of the FFT algorithms which are currently being used.

The discrete Fourier transform of length  $N$  requires time complexity to be  $O(N^2)$  whereas the time complexity of FFT is  $O(N \log_2 N)$ , where  $N$  is the number of data points. The following table shows the significant difference between the operation counts for the DFT and FFT algorithms [1].

Table 2.1 Operation counts for 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       |

## 2.1 Fastest Fourier Transform in the West (FFTW)

The Fastest Fourier Transform in the West package developed at the Massachusetts Institute of Technology (MIT) by Matteo Frigo and Steve G. Johnson. FFTW is a subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data [4].

FFTW 3.1.2 is the latest official version of FFTW. Here is a list of some of FFTW's more interesting features [4]:

1. FFTW supports both one one-dimensional and multi-dimensional transforms.
2. The input data can have arbitrary length. FFTW employs  $O(n \log n)$  algorithms for all lengths, including prime numbers.
3. FFTW supports fast transforms of purely real input or output data.
4. FFTW with versions above 3.0 supports transforms of real even/odd data.

5. Efficient handling of multiple, strided transforms, which lets the user transform multiple arrays at once, transform one dimension of a multi-dimensional array, or transform one field of a multi-component array.
6. Portability to any platform with a C compiler

The FFTW has obtained more accurate and optimized results for FFT. They achieved more extensive benchmarks on a variety of platforms. Their code is machine independent. The same program without any modification performs well on all most all the architectures. Since the code of FFTW is available for free download, we configured it on the Akron Cluster ([cluster2.cs.uakron.edu](http://cluster2.cs.uakron.edu)). We will discuss the comparison of FFTW results with our algorithm results in the next chapter. FFTW's performance is typically superior to any other publicly available FFT software. The authors of FFTW give three reasons for making their code more superior and faster than others [4]:

1. FFTW uses a variety of algorithms and implementation styles that adapt themselves to the machine.
2. FFTW uses an explicit divide-and-conquer methodology to take advantage of the memory hierarchy.
3. FFTW uses a code generator to produce highly optimized routines for computing small transforms.

## 2.2 The Carnegie Mellon University Spiral Group

The main goal of Carnegie Mellon University's "spiral group" [5], is to push the limit of automation in software and hardware development and to provide optimization for digital signal processing (DSP) algorithms and other numerical kernels. They provide a number of online generators for solving DFT and DCT. Their method of solving the

DFT is using generators, to create the code for a specific case of  $N$  (where  $N$  is the number of data points) which will fetch them more enhanced timings and more compact code.

### 2.2.1 DFT IP Generator

The Spiral DFT IP generator [24] is a fast generator for customized DFT soft IP cores. The user provides variety of input parameters like size of DFT, scaling mode, data width, constant width, parallelism, twiddle storage method, and FIFO threshold that control the functionality of the generated core. All these parameters control resource tradeoffs such as area and throughput as well. After filling in the parameters in the input form, the resources are dynamically estimated. The output generated is synthesizable Verilog code for an  $n$ -point DFT with parallelism  $p$ .

### 2.2.2 DCT IP Generator

The Spiral DCT IP generator [25] is a fast generator for customized DCT. The user provides input parameters like DCT size, data width, constant width, data ordering, scaling mode, parallelism, constant storage method and FIFO threshold that control the functionality of the generated core. The input parameters also control resource tradeoffs such as area and throughput. The output generated from the generator is a synthesizable Verilog code for an  $n$ -point DCT (type II) with parallelism  $p$ .

Both the DFT and DCT generators take the same input parameters and generate the specific Verilog code for the specific value of  $N$ , where  $N$  is number of input data points. Since the code is specific for specific value of  $N$  the time generated is very less.

### 2.3 Cooley-Tukey FFT algorithm

The Cooley-Tukey FFT algorithm is the most common algorithm for developing FFT. This algorithm uses a recursive way of solving DFT of any arbitrary size  $N$  [22]. The technique divides the larger DFT into smaller DFT. Which subsequently reduce the complexity of their algorithm. If the size of the DFT is  $N$  then this algorithm makes  $N=N_1.N_2$  where  $N_1$  and  $N_2$  are smaller DFT's. The complexity then becomes  $O(N \log N)$ .

Radix-2 decimation-in-time (DIT) is the most common form of the Cooley-Tukey algorithm, for any arbitrary size  $N$ . Radix-2 DIT divides the size  $N$  DFT's into two interleaved DFT's of size  $N/2$ . The DFT is defined by the formula [21]

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

Radix-2 divides the DFT into two equal parts. The first part calculates the Fourier transform of the even index numbers. The 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 an 8-point FFT algorithm is shown:

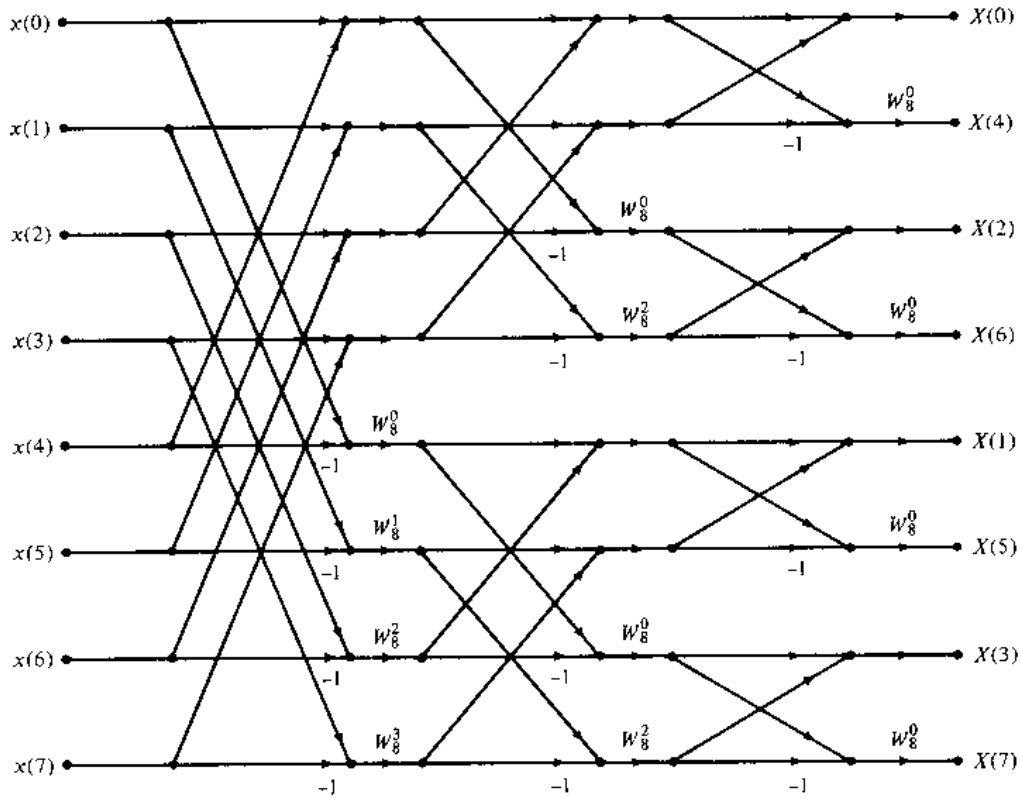


Figure 2.1  $N = 8$ -point decimation-in-frequency FFT algorithm.

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

#### 2.4 Summary

In this chapter, we discussed the FFT models which are currently being used. However the models provide sequential versions of the FFT. Little amount of research has been done on developing a parallel version of the FFT. We developed this approach of implementing the FFT in parallel.

## CHAPTER III

### MATERIALS AND METHODS

In this thesis the DCT is computed by using two different algorithms; the gg90 and the lifting algorithm. We present gg90 algorithm.

#### 3.1 DCT using the gg90 algorithm.

This algorithm involves three main steps, for  $n=8$ .

1. Reorder the input data points.
2. Calculate the cosine and sine values using the gg90 formula.
3. Calculate sum-difference step for the given input points.

For the given input data points for a vector of length 8, first change the order of the input points. In order to do that, the data points are stored in two different blocks: the even elements are stored in one block and the odds in the other in reverse order. For example the 8 data points  $x_0$  to  $x_7$  would be re-order as  $x_0, x_7, x_2, x_5, x_4, x_3, x_6, x_1$ . Calculate the cosine and sine values by using the formula.

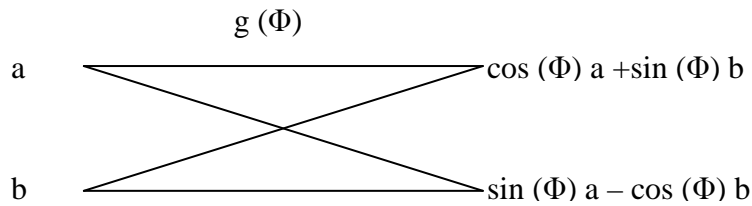


Figure 3.1 gg90 formula for calculating cosine and sine values

Figure 3.1 shows for two data points  $a, b$ , the output value of  $a$  is calculated as  $\cos(\Phi)a + \sin(\Phi)b$  and the output value of  $b$  is calculated as  $\sin(\Phi)a - \cos(\Phi)b$ .

The sum-difference step Figure 3.2 involves sum and difference operations on the input data points. The points are divided into two halves. The top half performs the addition operation and the bottom half performs the difference operation.

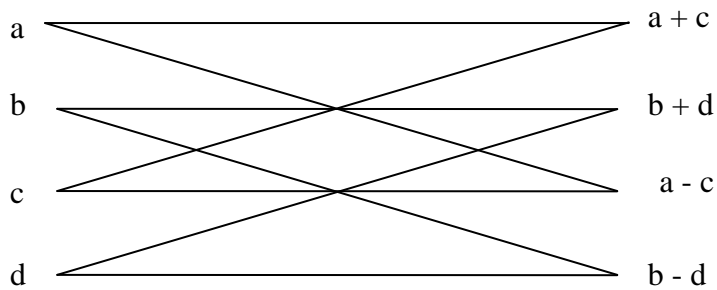


Figure 3.2 Sum-difference for four input data points

There is a last computation step involved when the data point size is two, as seen in Figure 3.3, the computation that needs to be performed is a sum-difference of two data points divided by the square root of 2.

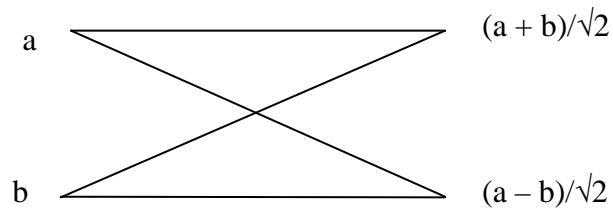


Figure 3.3 Last steps in DCT

Figure 3.4 shows the complete discrete cosine transform computation for 8 data points [13].

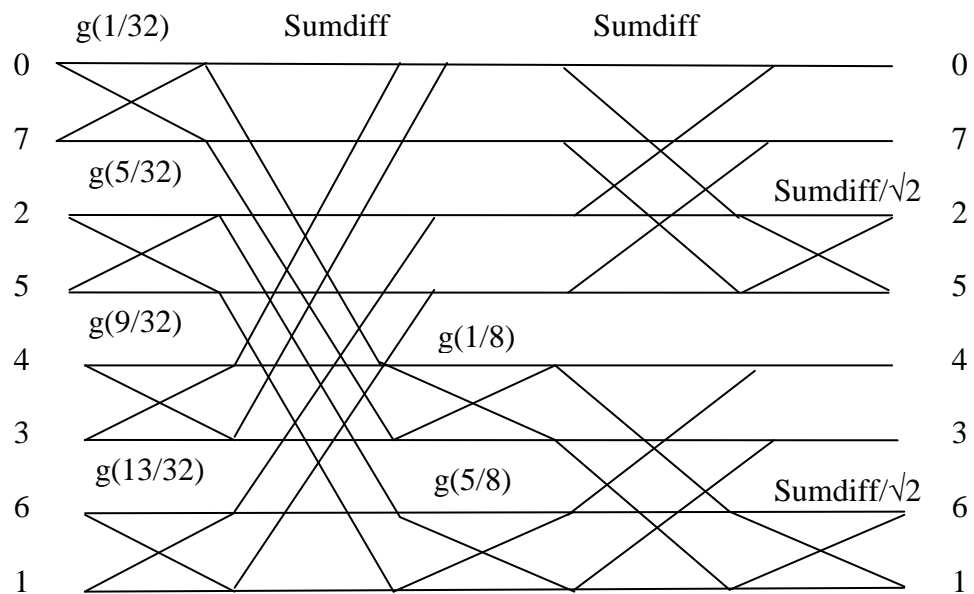


Figure 3.4 DCT for 8 data points

As demonstrated in Figure 3.4 the given input data points are reordered and the cosine and sine angles are computed. The output values are then operated on by the sum-difference method. After this first step, the problem is partitioned into two halves. The top half only requires a sum-difference operation, but the bottom half needs a sum-

difference and also a cosine and sine multiplication. Finally the last step carries out the sum-difference operation and then divides by the square root of 2.

### 3.2 DCT using the lifting algorithm

The steps involved in the lifting algorithm are similar to that of the gg90 algorithm, except for one step where the cosine and sine are calculated [13, 14].

For the case of  $N=8$  data points:

1. Reorder the input data points.
2. Calculate the cosine and sine values using the lifting formula.
3. Calculate the sum-difference step for the given input points.

Lifting performs the cosine and sine multiplication different from the gg90 algorithm. After calculating the cosine and sine values for the given input data points, we tabulate the  $R$  value, which is derived by the formula  $R=(c-1)/s$  where  $c$  and  $s$  are the cosine and sine values.

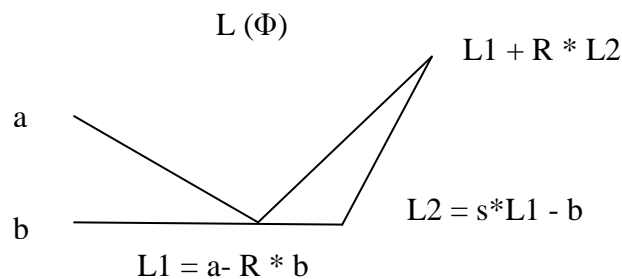


Figure 3.5 Lifting step for two data points

Computed:  $c = \cos(\pi/8)$ ,  $s = \sin(\pi/8)$   $R = (c-1)/s$

Figure 3.5 shows that for two data point a, b only the R and sine values are used. First calculate an intermediate value  $L1=a-R*b$  which is used to derive the values for the two data points. The first second data output value is calculated by the formula  $s*L1 -b$ . This value is stored in L2 which is then used to compute the output for the first data point by the formula  $L1+R*L2$ .

### 3.2.1 DCT using the lifting algorithm for 8 data points

The given input data points are reordered. The even values are stored in order and the odd values are stored in reverse order. The lifting formula is applied to compute the cosine and sine values followed by a sum-difference step. After the sum-difference operation the problem is divided into two halves. The first half performs the normal sum-difference step and the bottom half performs lifting and the sum-difference step. As seen from Figure 3.6, the last step involves the sum-difference and then division by the square root of 2.

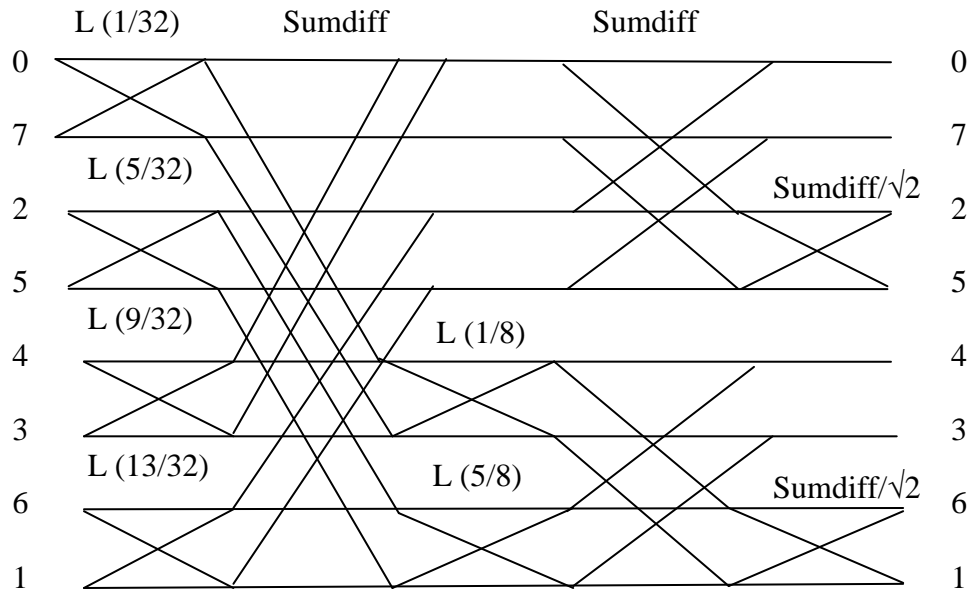


Figure 3.6 DCT using lifting step for 8 data points

Both algorithms have been implemented on The University of Akron cluster and tested on the Ohio Supercomputer cluster for comparison. We discuss the results of both these algorithms in Chapter 4.

As we have seen, for both algorithms, the input data points are reordered before applying the DCT. Hence the results produced are also not in the correct order. A re-ordering step is used to bring the values back into original order. We applied the re-ordering step for the lifting algorithm, but the timings went very high because the processor interaction time surpasses the actual time to compute the DCT values. In the gg90 algorithm, instead of re-ordering the final data points, we perform that step in the FFT itself. By doing this the timings were very satisfactory.

### 3.3 Fast Fourier Transform

In this thesis we implemented a FFT algorithm which runs in parallel with the DCT [14]. As we have seen from the above section 3.1 and 3.2, implemented two different algorithms for computing the DCT and hence used it further in the FFT. We chose to proceed with the gg90 algorithm for computing the FFT instead of the lifting algorithm because it is seen from the implementation that the accuracy of the gg90 algorithm is much higher than that of the lifting algorithm.

#### 3.3.1 Fast Fourier transform algorithm using gg90 algorithm

In order to compute the FFT there are two major steps involved, applied many times in succession.

1. Calculate the sum-difference step for the given input points.
2. Compute the DCT using the gg90 algorithm.

As seen from figure 3.7 for the four data points a, b, c and d the sum-difference step breaks the data points into two halves. The first half performs the addition operation and the second half performs the difference operation.

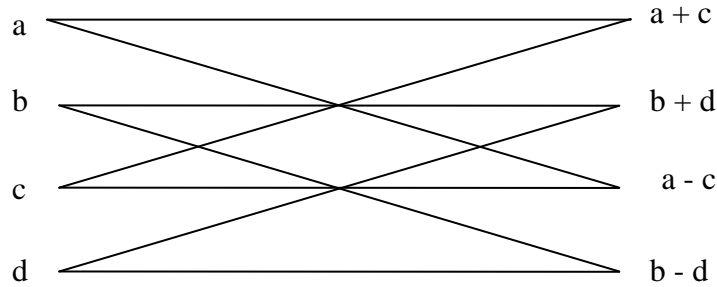


Figure 3.7 Sum-difference operations for the input data points

Since we have already computed the DCT using the gg90 algorithm, for the FFT we fix the DCT and make the entire FFT run in parallel.

When computing the FFT for any input of real data points, the first step is to compute the sum-difference. After this initial step the FFT is broken into two halves. The first part performs only the sum-difference operation. The bottom half calls the sub routine which calculates the DCT using the gg90 method. Both halves run in parallel independent of one another. After computing the resulting values, the master processor receives the computed values and performs a few final re-ordering steps to generate the output values. The FFT for complex values is computed by computing the real and imaginary part separately on two different processors. For example in order to calculate  $a + i(b)$ , where a and b are real and imaginary data points, the real part is computed on processor 0 and the imaginary part on processor 1. These two processors work

independently of each other, and each internally calls the DCT. Thus not only does the FFT run in parallel but also the DCT runs in parallel within it.

We implemented the FFT using cases of real and complex input values. We discuss and compare our results with previous FFT implementations [1] and show ways to improve the timings by using multiple processors in parallel in the next chapters.

### 3.4 Construction of n=8 point FFT in parallel

We now discuss how we have constructed the FFT in parallel for a small case of 8 data points. For the given 8 data points, the initial step is to perform the sum-difference operation. After the first step, the data points are halved. The first half is independent of the bottom half. The top half calculates only the sum-difference operation for the four data points. The bottom half performs a diffsumflip which means flipping the data points after the sum- difference operation. The output is then multiplied with cosine and sine values by the gg90 formula.

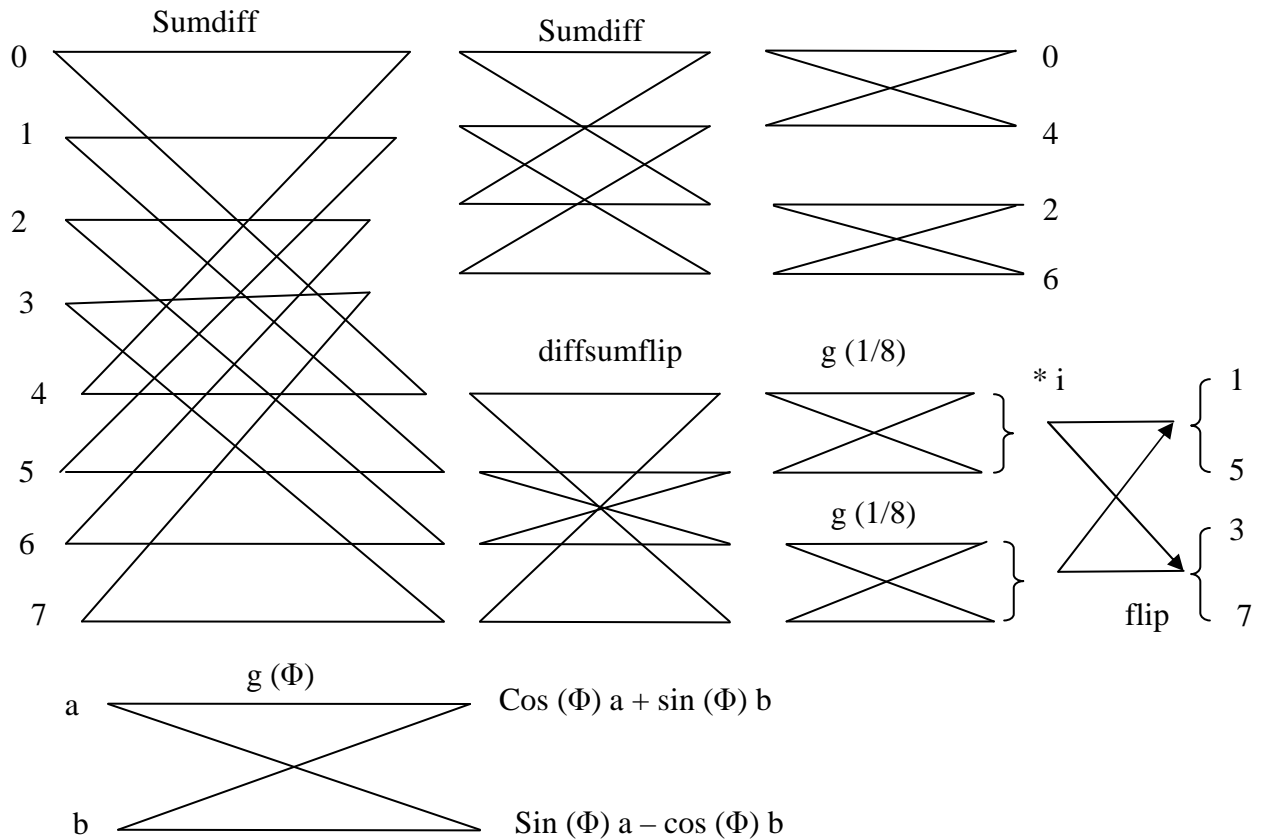


Figure 3.8 FFT for n=8 data points

As seen from Figure 3.8, the gg90 formula for two data points a, b is  $\cos(\Phi)a + \sin(\Phi)b$  and  $-\sin(\Phi)a + \cos(\Phi)b$ . A final flip is carried out in the bottom half to output the data points in the correct order. In other words the bottom half is the same as that of the DCT transform for four data points.

In our experiments we have made the bottom and top halves independent of one another. We programmed the two halves to run in parallel, thus making the FFT run in parallel. Since the DCT is working in the bottom half, we worked to achieve having both the FFT and DCT run in parallel.

### 3.5 FFT using 16 data points

Figure 3.9 below shows the FFT using 16 data points [14]. After the initial step of a sum-difference, the top half and bottom half work independent of each other.

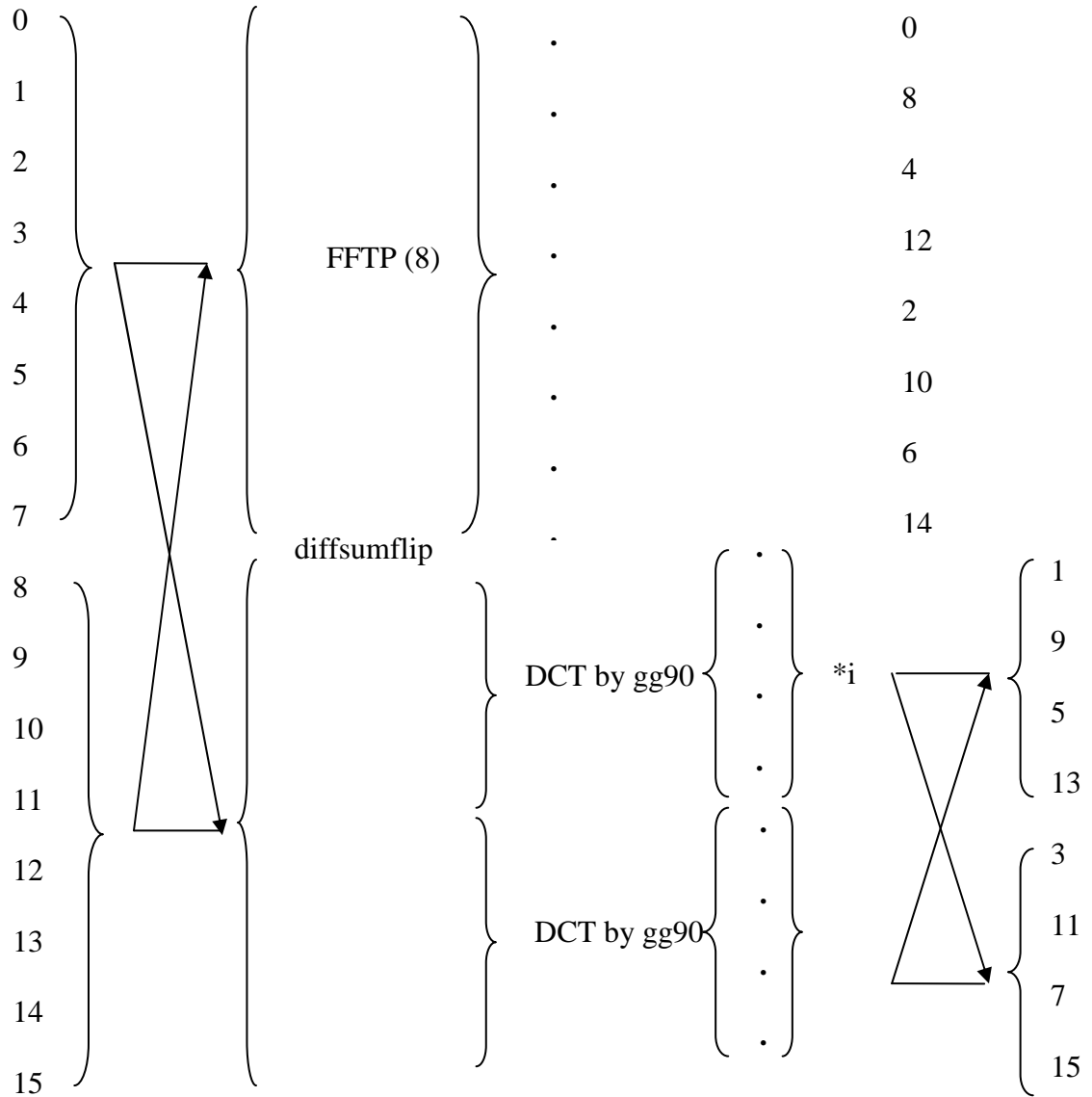


Figure 3.9 FFT using 16 data points

The top half works as in the 8 point FFT. The bottom half calls the internal DCT sub routine. Both halves work on different processors and with the final result reported back to the master processor.

### 3.6 Summary

In this chapter, we demonstrated algorithms for the DCT using the gg90 and lifting algorithms. We also explained how the DCT is fixed in the FFT algorithm using suitable examples for 8 and 16 data points.

## CHAPTER IV

### RESULTS AND DISCUSSION

Many major benchmarks have been obtained during the implementation phase of the parallel computation of the DCT and FFT. We tested our application both on The University of Akron cluster (UA), and on the glenn cluster located at the Ohio Supercomputer Center (OSC).

#### 4.1 Hardware configuration of the OSC cluster [7]

##### 1. 877 System x3455 compute nodes

Dual socket, dual core 2.6 GHz opterons

8 GB RAM

48 GB local disk space in /tmp

##### 2. 88 system x3755 compute nodes

Quad socket, dual core 2.6 GHz opterons

64 GB (2 nodes), 32 GB (16 nodes), or 16 GB (70 nodes) RAM

1.8 TB (10 nodes) or 218 GB (76 nodes) local disk space in /tmp

##### 3. One e1350 Blade Center

4 Dual Cell based QS20 blades

Voltaire 10 Gbps PCI express adapter

4 System x3755 login nodes

Quad socket 2 dual core 2.6 GHz Opterons

8 GB RAM

4. All parts connected by 10 Gbps Infiniband

#### 4.2 Hardware configuration of the Akron cluster

1. 46 compute nodes

Intel ® Pentium ® D CPU 3.00 GHz

2 GB RAM

CPU MHz is 3000.245

2. Dual gigabit networks on private switches are used for cluster communications; one for diskless operations, the other dedicated to MPI traffic. Only the front node communicates to the outside world.

#### 4.3 Discrete Cosine Transform

As we discussed earlier in Chapter 3, we started building the DCT type IV, using two different algorithms. We present the results of the DCT computation using the lifting algorithm and the gg90 algorithm, followed by a comparison of these two algorithms.

##### 4.3.1 Discrete cosine transforms using lifting algorithm

Table 4.1 below shows the relative timings generated for the DCT for values of N using the lifting algorithm. The program is implemented on one, two and four processors. A comparison graph is also displayed. These tests are implemented on the UA cluster. It is observed from the Figure 4.1 that the DCT on one processor gives better timings than 2 and 4 processors because the DCT requires a re-ordering step. Also when running on 2 and 4 processors, the processors require extensive message passing which increases the overall computation time. The table describes only the relative timings on the UA

clusters, as it list the ratio of the timings on either 1 or 4 processors to the timings on 2 processors. For actual timings, refer to Appendix A.

Table 4.1 Comparing DCT lifting algorithm on 1, 2 and 4 processors

| N    | Lift_1 | Lift_2 | Lift_4 |
|------|--------|--------|--------|
| 8    | 0.27   | 1      | 1.43   |
| 16   | 0.133  | 1      | 0.919  |
| 32   | 0.09   | 1      | 1      |
| 64   | 0.07   | 1      | 1.14   |
| 128  | 0.06   | 1      | 1.19   |
| 256  | 0.04   | 1      | 1.067  |
| 512  | 0.04   | 1      | 1.011  |
| 1024 | 0.03   | 1      | 1.012  |
| 2048 | 0.03   | 1      | 0.99   |
| 4096 | 0.02   | 1      | 0.99   |
| 8192 | 0.028  | 1      | 1.002  |

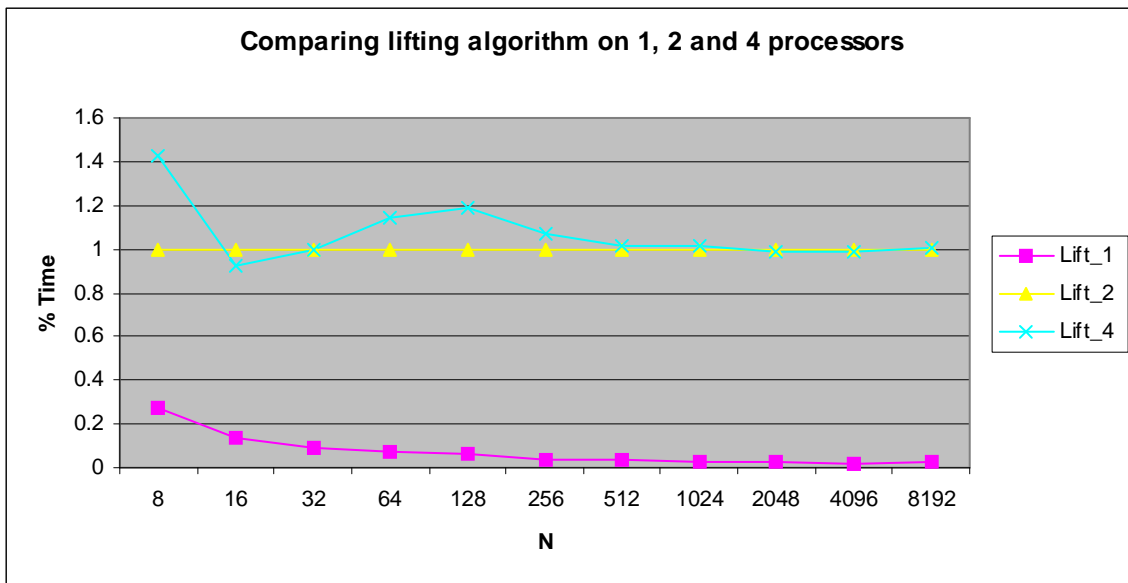


Figure 4.1 Comparing lifting algorithm on 1, 2 and 4 processors

#### 4.3.2 Comparison of lifting algorithm on UA and OSC using 1 processor

Table 4.2 below shows the comparison of the lifting algorithm for the DCT using one processor both using the UA cluster and the OSC cluster. From Figure 4.2 the graph, it is observed that the UA cluster performs better than the OSC cluster when the value of N is small. This is because all nodes in the OSC cluster are Quad core. Though the program utilizes only one processor, all four processors are initiated. As the value of N increases, the OSC cluster gives comparatively better timings. The table describes only the relative timings on both of the clusters. For actual timings, refer to Appendix A.

Table 4.2 Comparing the lifting algorithm at UA and OSC cluster on 1 processor

| <b>N</b> | <b>UA_1Processor</b> | <b>OSC_1 Processor</b> |
|----------|----------------------|------------------------|
| 8        | 0.6                  | 1                      |
| 16       | 0.62                 | 1                      |
| 32       | 0.67                 | 1                      |
| 64       | 0.67                 | 1                      |
| 128      | 0.76                 | 1                      |
| 256      | 0.845                | 1                      |
| 512      | 1.06                 | 1                      |
| 1024     | 1.18                 | 1                      |
| 2048     | 1.33                 | 1                      |
| 4096     | 1.57                 | 1                      |
| 8192     | 2.03                 | 1                      |
| 16384    | 2.38                 | 1                      |
| 32768    | 2.81                 | 1                      |

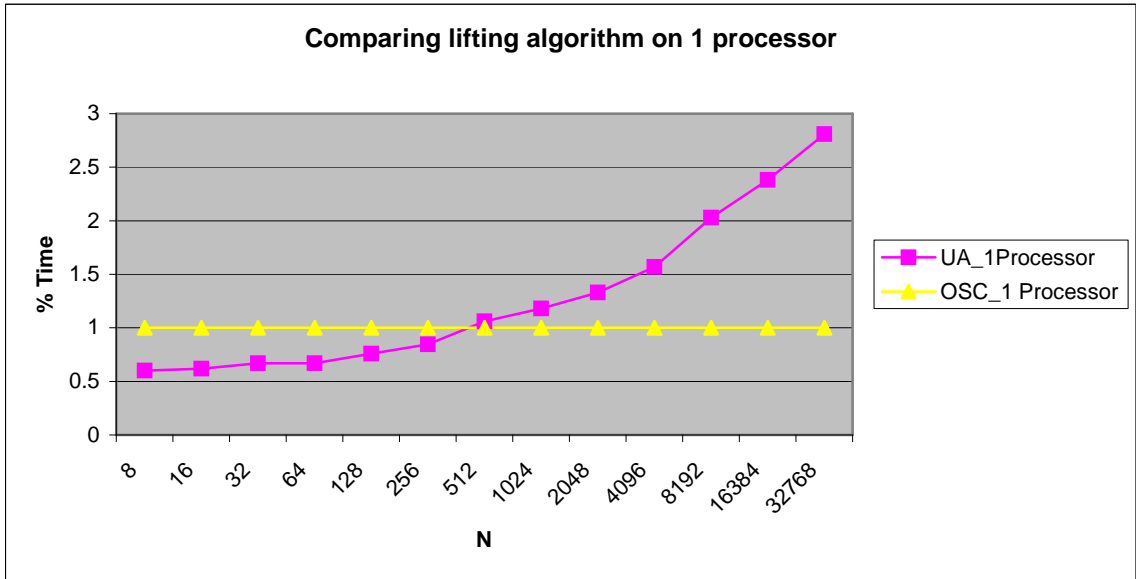


Figure 4.2 Comparing lifting algorithm on 1 processor

#### 4.4 Comparison of the gg90 and lifting algorithm

After implementing lifting algorithm we found out that the timings were not so impressive, that is due to the fact that the final re-ordering step in DCT is taking more processor time. In gg90 algorithm we eliminated the final re-ordering step and take care of this step in FFT algorithm. Table 4.3 below shows that gg90 algorithm gives better timing than lifting algorithm. The table describes only the relative timings on the clusters. In order to see the actual timings, refer to Appendix A.

Table 4.3 Comparing the gg90 and lifting algorithm at UA cluster on 1 processor

| N     | UA_1 gg90 | OSC_1 Lift |
|-------|-----------|------------|
| 128   | 0.61      | 1          |
| 256   | 0.49      | 1          |
| 512   | 0.54      | 1          |
| 1024  | 0.52      | 1          |
| 2048  | 0.47      | 1          |
| 4096  | 0.48      | 1          |
| 8192  | 0.47      | 1          |
| 16384 | 0.47      | 1          |
| 32768 | 0.48      | 1          |

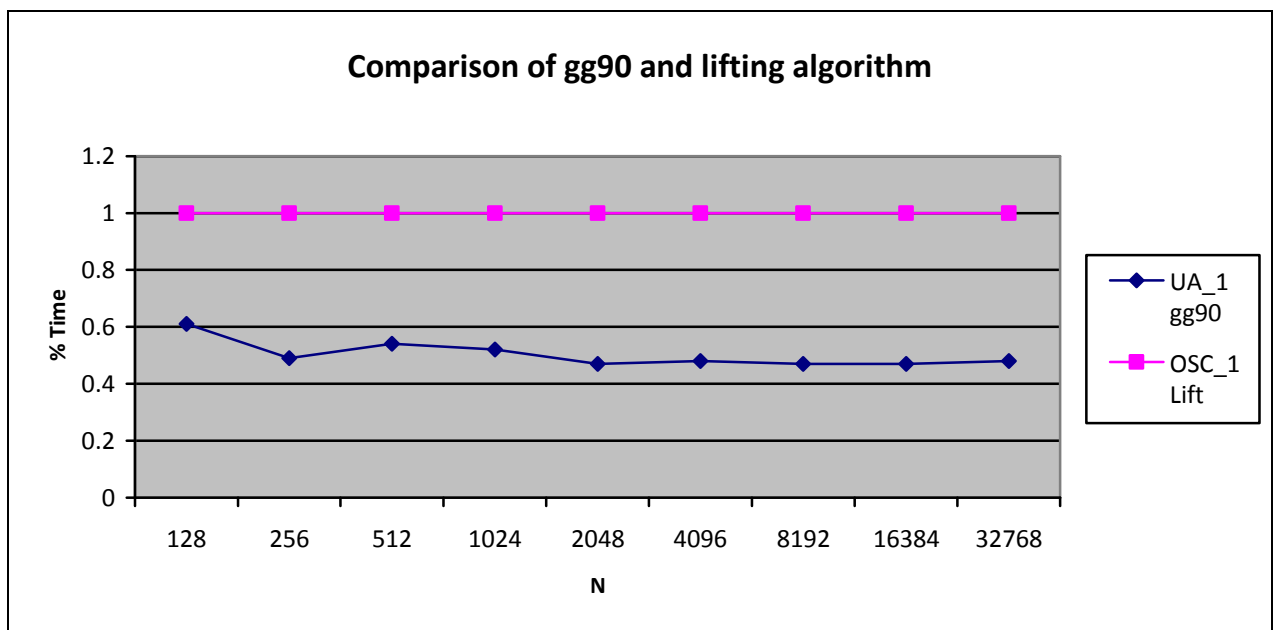


Figure 4.3 Comparing gg90 and lifting algorithm on 1 processor

## 4.5 Fast Fourier Transform

The FFT has been implemented in this study for input that is both real-valued and complex-valued. For the complex-valued case, the FFT is implemented by using 1, 2 and 6 processors.

### 4.5.1 Real case FFT using 1 processor

Table 4.4 below shows the comparison of the FFT for the real case on the UA cluster and the OSC cluster. It is seen from the figure 4.4 that the timings on the OSC cluster are comparatively better than those on the UA cluster because of the architecture of the machine, and because each node on the OSC cluster has 8GB of memory which is three times faster than the memory on the UA cluster. The table describes only the percentage timings on both the clusters. For actual timings, refer to Appendix A.

Table 4.4 Real case FFT using 1 processor

| <b>N</b> | <b>UA_1Processor</b> | <b>OSC_1 Processor</b> |
|----------|----------------------|------------------------|
| 128      | 0.91                 | 1                      |
| 256      | 0.8669               | 1                      |
| 512      | 0.938                | 1                      |
| 1024     | 1.008                | 1                      |
| 2048     | 1.03                 | 1                      |
| 4096     | 1.15                 | 1                      |
| 8192     | 1.19                 | 1                      |
| 16384    | 1.18                 | 1                      |
| 32768    | 1.12                 | 1                      |

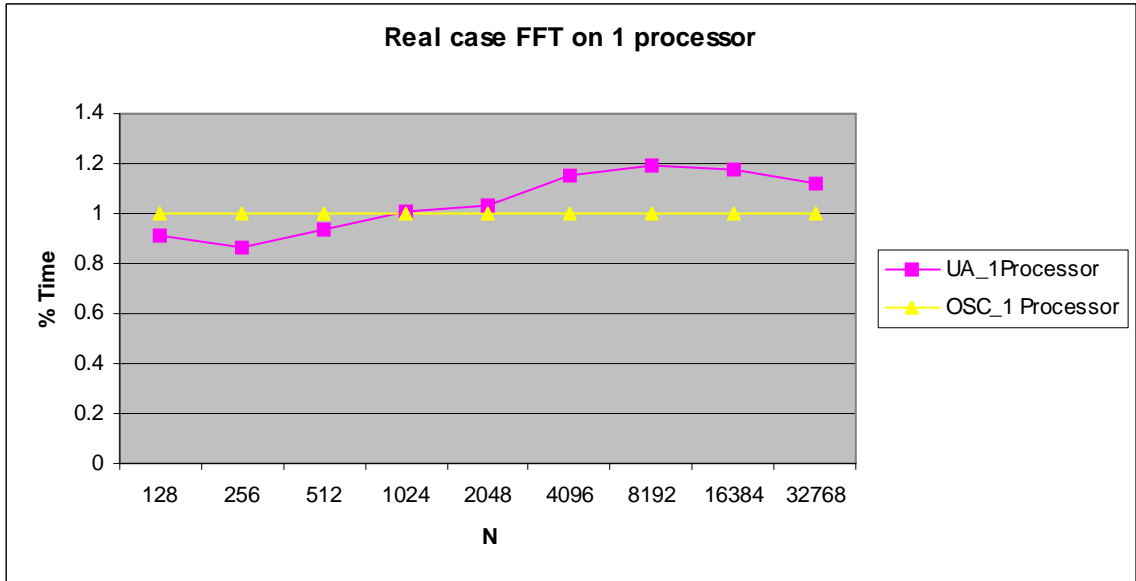


Figure 4.4 Real case FFT on 1 processor

#### 4.5.2 Comparisons of the real case FFT using 1 processor

Table 4.5 below shows the comparison of FFT timings with [1]. It is observed from Figure 4.5 our FFT on one processor performs better for large values of N. The table describes only the percentage timings on both the clusters. For actual timings, refer to Appendix A.

Table 4.5 Comparison of real case FFT on 1 processor

| <b>N</b> | <b>UA_1Processor</b> | <b>OSC_1 Processor</b> | <b>Misal [1]</b> |
|----------|----------------------|------------------------|------------------|
| 256      | 0.8669               | 1                      | 0.29             |
| 512      | 0.938                | 1                      | 0.45             |
| 1024     | 1.008                | 1                      | 0.62             |
| 2048     | 1.03                 | 1                      | 0.93             |
| 4096     | 1.15                 | 1                      | 1.14             |
| 8192     | 1.19                 | 1                      | 1.42             |
| 16384    | 1.18                 | 1                      | 1.38             |
| 32768    | 1.12                 | 1                      | 1.64             |

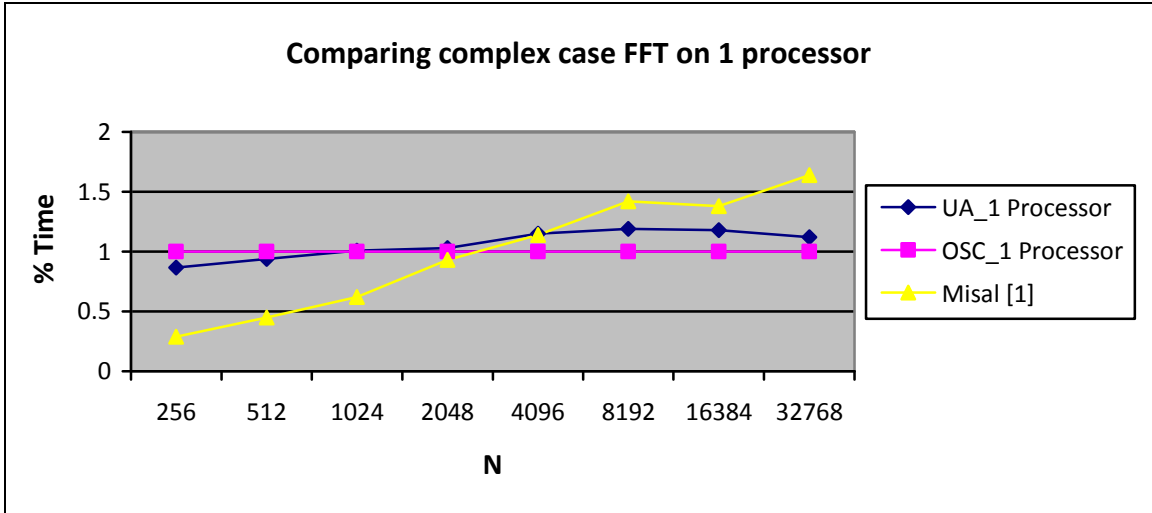


Figure 4.5 Comparison of real case FFT on 1 processor

#### 4.5.3 Complex case FFT using 2 processors

We consider both the real and imaginary case for complex-valued FFT. The Figure 4.6 below shows how the FFT has been implemented in parallel using two processors. When the execution begins, processors 0 and 1 receive the real and imaginary parts of the input, respectively. Both processors work independently on the real and imaginary input. Once processor 1 completes execution, it sends its results to processor 0. The 0<sup>th</sup> processor will then combine the real and complex parts to generate the complete FFT.

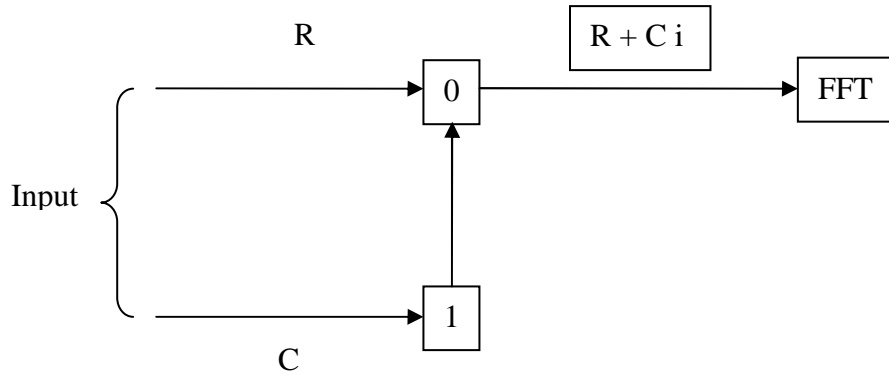


Figure 4.6 Implementation of FFT on 2 processors

Table 4.6 below shows comparative results when the FFT has been implemented on 2 processors, comparing the UA processor and the OSC cluster. The comparison graph is shown below in figure 4.7. From the graph, it is seen that as the value of N increases, the timings on the UA processor slowly decrease and tend to match up with the OSC cluster. This is because the UA cluster has a latency time of 50 cycles whereas OSC latency time is 0. In the initial cases the timings are affected. However, once both of the processors get enough input to process, at this point the latency time is overshadowed by the execution time. The table describes only the relative timings on both the clusters; for actual timings refer to Appendix A.

Table 4.6 Complex fast Fourier transform on 2 processors

| N     | UA_2 Processor | OSC_2 Processor |
|-------|----------------|-----------------|
| 128   | 1.73           | 1               |
| 256   | 1.49           | 1               |
| 512   | 1.54           | 1               |
| 1024  | 1.36           | 1               |
| 2048  | 1.4            | 1               |
| 4096  | 1.23           | 1               |
| 8192  | 1.289          | 1               |
| 16384 | 1.28           | 1               |
| 32768 | 1.17           | 1               |

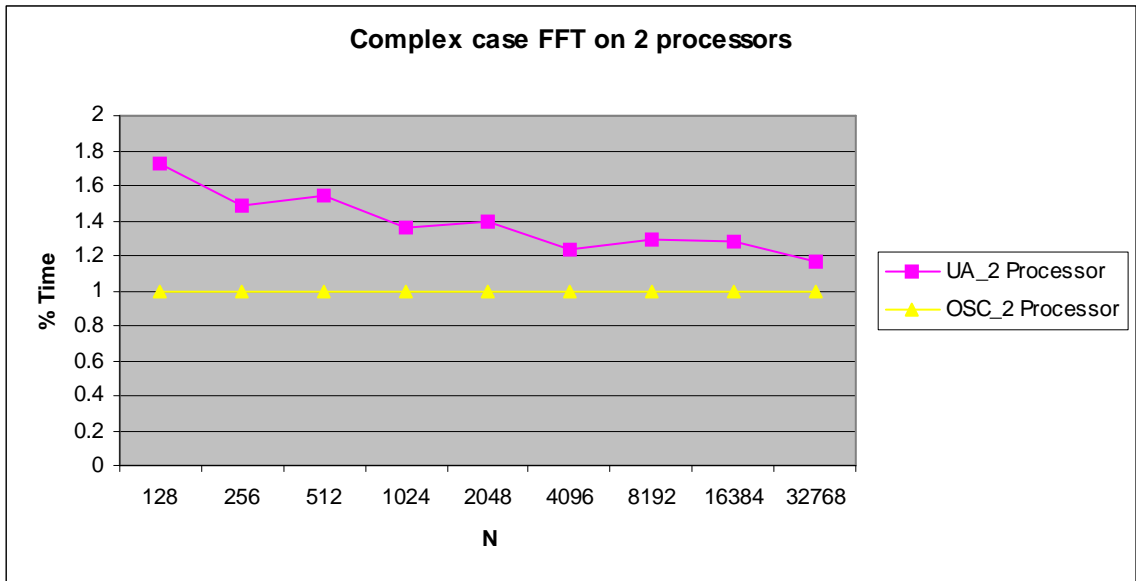


Figure 4.7 Complex case FFT on 2 processors

#### 4.5.4 Comparisons of the complex case FFT using 2 processors

Table 4.7 below shows the comparison of the FFT in parallel with [1] using 2 processors. The timings are calculated on both UA cluster and OSC cluster. From the table we observed that as the value of N increases, the FFT in parallel generates comparatively better timings than [1]. This is because when the value of N is small, the interaction time

between processors time is dominating, but as the value of N increases, both processors get enough data to work on. The method of this paper beats the timings from those in [1] once the value of N becomes 8192. As above, the table describes only the relative timings on both the clusters. For actual timings refer to Appendix A.

Table 4.7 Comparing complex case FFT using 2 processors

| N     | UA_2 Processor | OSC_2 Processor | Misal [1] |
|-------|----------------|-----------------|-----------|
| 256   | 1.49           | 1               | 0.236     |
| 512   | 1.54           | 1               | 0.38      |
| 1024  | 1.36           | 1               | 0.49      |
| 2048  | 1.4            | 1               | 0.61      |
| 4096  | 1.23           | 1               | 0.67      |
| 8192  | 1.289          | 1               | 1.07      |
| 16384 | 1.28           | 1               | 1.29      |
| 32768 | 1.17           | 1               | 1.54      |

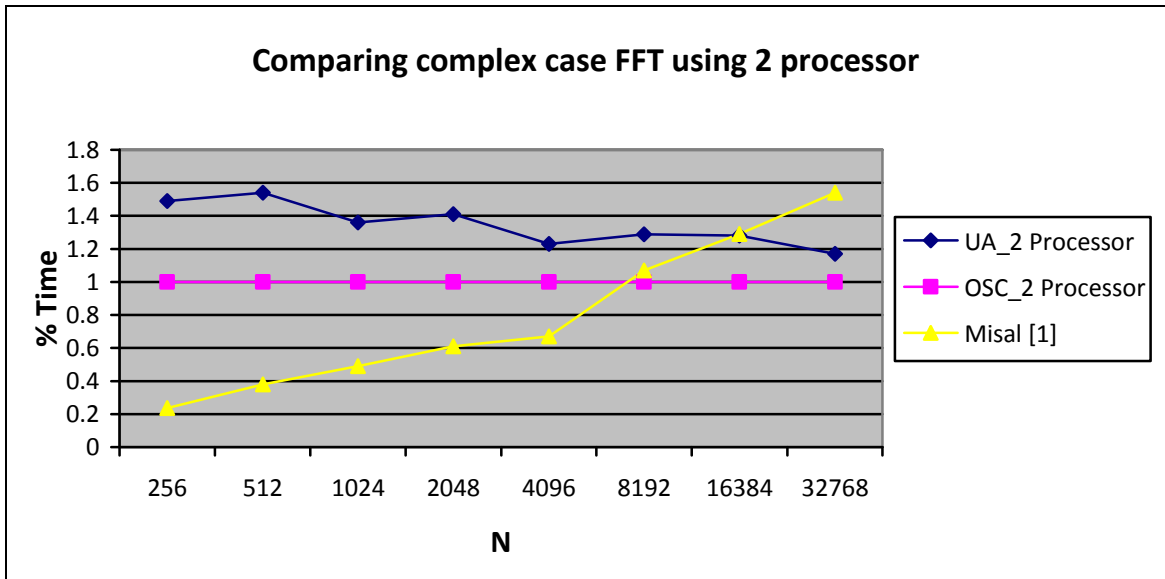


Figure 4.8 Comparison of complex case FFT using 2 processors

#### 4.5.5 Complex case FFT using 6 processors

Figure 4.9 below shows how the FFT is implemented by using 6 processors. Processors 0 and 1 get the real and complex input respectively. Processor 0 will then assign processors 2 and 3 to compute the DCT. Processors 2 and 3 will then work independently and send the message back to processor 0. Processor 1, which is working on complex input independent of processor 0, will assign processors 4, 5 to compute the DCT. Both processors 4 and 5 send messages back to processor 1 after computing the DCT. Processor 1 then gives the computed complex values to processor 0. Processor 0 will perform final computations and generate the FFT.

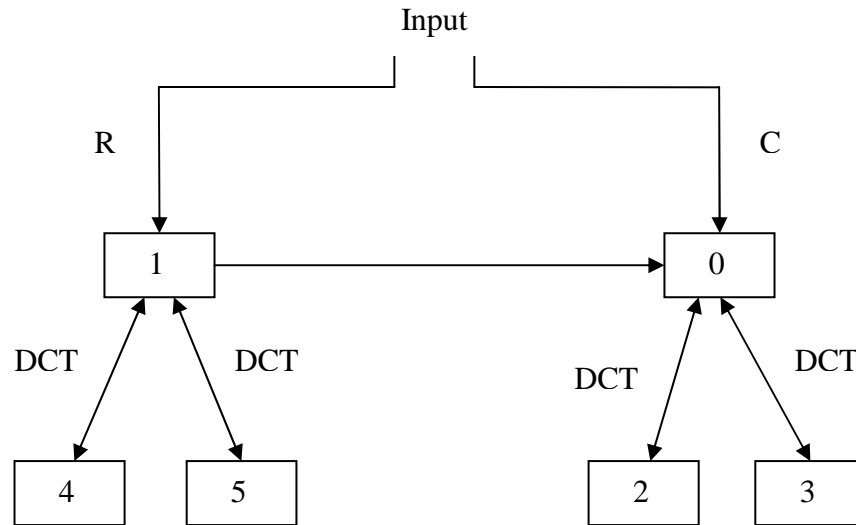


Figure 4.9 Implementation of FFT on 6 processors

Table 4.8 below shows the comparison of timings on both the UA cluster and the OSC cluster. The comparison graph figure 4.10 is shown below. It is observed that the performance of the OSC cluster is better than that of the UA cluster because of the high

latency time of the UA cluster. While using 6 processors, the latency time will be approximately 6\*50, whereas the latency time is zero at the OSC cluster. The table describes only the relative timings on both clusters, for actual timings refer to Appendix A.

Table 4.8 Complex case FFT on 6 processors

| N     | UA_6 Processor | OSC_6 Processor |
|-------|----------------|-----------------|
| 128   | 2.53           | 1               |
| 256   | 2.65           | 1               |
| 512   | 3.04           | 1               |
| 1024  | 3.53           | 1               |
| 2048  | 2.92           | 1               |
| 4096  | 3.35           | 1               |
| 8192  | 3.45           | 1               |
| 16384 | 4.34           | 1               |

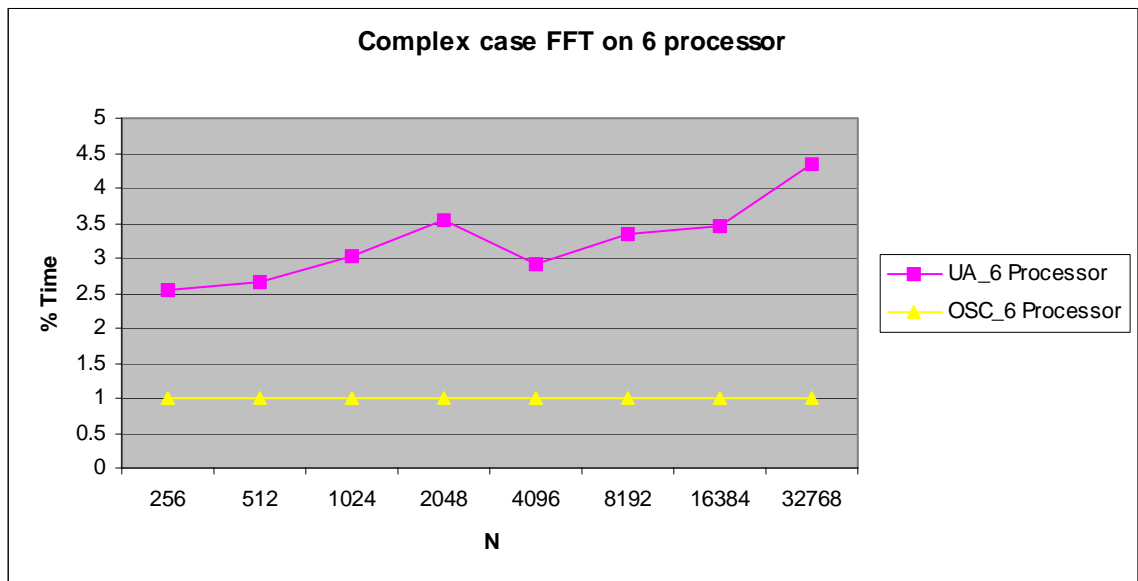


Figure 4.10 Complex case FFT on 6 processors

#### 4.5.6 Comparison of complex case FFT using 1, 2 and 6 processors

Table 4.9 below shows the comparison of the FFT using 1, 2 and 6 processors. With an increase in the value of N, the FFT running on 6 processors gives a more optimized time. It is seen that for small cases of N the message passing time between 6 processors dominates the overall time to generate the FFT, but as the value of N increases the FFT using 6 processors gives a comparatively better timing than those using 1 and 2 processors. The table describes only the relative times in order to see the actual timings refer to Appendix A.

Table 4.9 Complex case FFT on 1, 2 and 6 processors

| <b>N</b> | <b>OSC_1 Processor</b> | <b>OSC_2 Processor</b> | <b>OSC_6 Processor</b> |
|----------|------------------------|------------------------|------------------------|
| 128      | 0.94                   | 1                      | 3.523                  |
| 256      | 0.87                   | 1                      | 2.62                   |
| 512      | 0.866                  | 1                      | 1.61                   |
| 1024     | 0.89                   | 1                      | 1.18                   |
| 2048     | 0.86                   | 1                      | 1.07                   |
| 4096     | 1.12                   | 1                      | 1.13                   |
| 8192     | 0.911                  | 1                      | 0.99                   |
| 16384    | 0.876                  | 1                      | 0.85                   |

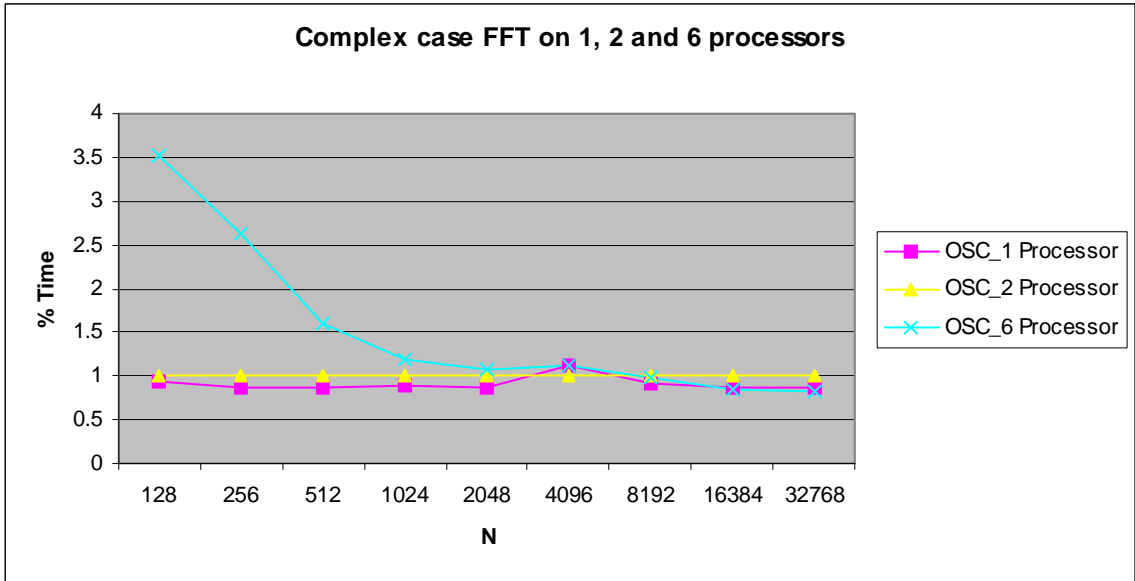


Figure 4.11 Complex case FFT on 1, 2 and 6 processors

#### 4.5.7 Comparison of complex case FFT in parallel with FFTW 3.2

Table 4.10 below shows the comparison of the FFT in parallel with the (FFTW) [4]. It is seen from the table that as the value of N increases, the performance of the FFT in parallel improves. The table describes only the relative time on the clusters, for actual timings refer to Appendix A.

Table 4.10 Comparison of FFT and FFTW 3.2

| N       | FFT | FFTW 3.2 |
|---------|-----|----------|
| 2048    | 26  | 1        |
| 4096    | 25  | 1        |
| 8192    | 20  | 1        |
| 16384   | 17  | 1        |
| 32768   | 16  | 1        |
| 65536   | 15  | 1        |
| 131072  | 13  | 1        |
| 262144  | 13  | 1        |
| 524288  | 10  | 1        |
| 1048576 | 10  | 1        |

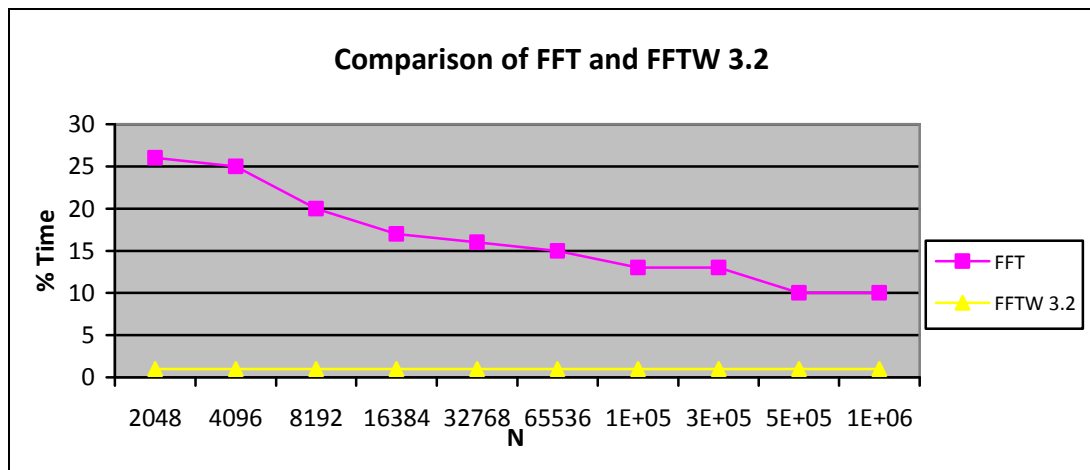


Figure 4.12 Comparison of FFT and FFTW 3.2

## 4.6 Summary

In this chapter, we showed the experimental results for the DCT using the lifting and gg90 algorithms. We explained how the gg90 algorithm performs better than the lifting algorithm. Finally conducted experiments for the FFT and compared our results with those from previous thesis [1]. All experiments are carried out on both the UA and OSC clusters.

## CHAPTER V

### CONCLUSIONS

The research contribution of this thesis provides a new parallel implementation of the FFT that involves the Type IV DCT. This implementation was coded in C and tested on both the UA cluster and the OSC cluster. This parallel FFT allows the entire FFT to be computed in parallel. This application can be used when a fast FFT computation needs to be done. The results are promising, but the speedup when using two or six processors has not yet been shown to provide a significant increase.

#### 5.1 Future work

There is future work where in the user can improve the performance of the FFT by increasing the number of processors. This would require using all of the processors with a load-balancing technique. The general FFT program works with any number of input points. In order to achieve better timings, generators can be implemented which will create fewer lines of code for small values of  $N$  and thus give better timings. All well known FFT developers in the market currently use generators.

## REFERENCES

- 1 Nilimb Misal, A Fast Parallel Method of Interleaved FFT for Magnetic Resonance Imaging, The University of Akron, Master's Thesis [May 2005].
- 2 Barry Wilkinson and Michael Allen, Parallel Programming- Techniques, and Applications using Networked Workstations and Parallel Computers, 1999, Prentice Hall.
- 3 [http://www.bridgeport.edu/sed/projects/cs597/Summer\\_2002/kunhlee/index.html](http://www.bridgeport.edu/sed/projects/cs597/Summer_2002/kunhlee/index.html), online link [dated Summer 2002].
- 4 <http://www.fftw.org>, online link [dated August 10, 2008].
- 5 <http://www.spiral.net>, online link [dated August 12, 2008].
- 6 <http://www-unix.mcs.anl.gov/mpi>, online link [dated September 10, 2001].
- 7 <http://www.osc.edu/supercomputing/hardware>, online link [dated 2008].
- 8 <http://www.osc.edu/supercomputing/computing/opt/index.shtml>, online link [dated 2008].
- 9 A.Edelman, P.McCorquodale and S.Toledo, The Future Fast Fourier Transform, SIAM J.Sci. Computing 20, 1094-1114 (1999)
- 10 <http://cobweb.ecn.purdue.edu/~ace/jpeg-tut/jpgdct1.html>, online link [dated 2008].
- 11 C. Loeffler, A. Ligtenberg and G. Moschytz, Practical Fast 1-D DCT Algorithms with 11 Multiplications, Proc. Int'l. Conf. on Acoustics, Speech, and Signal Processing (ICASSP`89), pp. 988-991, 1989.
- 12 Jon Louis Bentley, Writing Efficient Programs, 1982, Prentice-Hall.
- 13 Zhongde Wang, Fast algorithms for the discrete W transform and for the discrete Fourier transform, IEEE Trans. Acoustics, Speech and Sig.Proc, v.32,pp. 803-816, Aug. 1984.
- 14 D.H.Mugler, The New Interleaved Fast Fourier Transform, private communication.

- 15 H.S.Hou, A fast recursive algorithm for computing the discrete cosine Transform, IEEE Trans. Acoustics, Speech and Sig.Proc, v.10, pp. 1455-1461, Oct. 1987.
- 16 <http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html>, online link [dated February 10,2006]
- 17 <http://web.engr.oregonstate.edu/~pancake/presentations/sdsc.pdf>, online link. [dated February 10, 2006]
- 18 [http://www.epcc.ed.ac.uk/computing/services/cray\\_service/documents/tech\\_reports/EPCC-TR97-01/node5.html](http://www.epcc.ed.ac.uk/computing/services/cray_service/documents/tech_reports/EPCC-TR97-01/node5.html), online link [dated February 10,2006]
- 19 <http://www.jjj.de/fft/> , online link [dated February 10,2006]
- 20 <http://momonga.t.u-tokyo.ac.jp/~ooura/fft.html>, online link [dated February 10,2006]
- 21 James W. Cooley and John W. Tukey, An algorithm for the machine calculation of complex Fourier series, Math. Computing 19, pp.297–301, 1965
- 22 P. Lee and F. Y. Huang, Restructured Recursive DCT and DST Algorithms, IEEE Trans. Signal Processing, v. 42 (7), pp. 1600-1609, 1994
- 23 Steven G. Johnson and Matteo Frigo, "A modified split radix FFT with fewer arithmetic operations", IEEE Trans. Signal Processing, in press (2006).
- 24 <http://spiral.net/hardware/dftgen.html>, online link [dated August 12, 2008]
- 25 <http://spiral.net/hardware/dctgen.html>, online link [dated August 12, 2008]
- 26 S. Winograd, "On computing the discrete Fourier transform," Math. Computation, vol. 32, no. 1, pp. 175–199, Jan. 1978
- 27 [www.geocities.com/ResearchTriangle/8869/fft\\_summary.html](http://www.geocities.com/ResearchTriangle/8869/fft_summary.html), online link [dated 10 February 2006].
- 28 Z. Cvetkovic and M.V. Pepovic, New fast recursive algorithms for the computation of discrete cosine and sine transforms, IEEE Trans. Sig.Proc., v. 40 (8), pp. 2083-2086, 1992.

## APPENDICES

## APPENDIX A

### TABLES SHOWING THE ACTUAL TIMINGS

#### 4.3.1 DCT using the lifting algorithm

Table A1 Timing in microseconds of lifting algorithm on 1,2 and 4 processors

| <b>N</b> | <b>Lift_1</b> | <b>Lift_2</b> | <b>Lift_4</b> |
|----------|---------------|---------------|---------------|
| 8        | 45.9          | 174           | 250           |
| 16       | 49.9          | 373           | 342.98        |
| 32       | 63.2          | 636           | 642.8         |
| 64       | 93.4          | 1312.98       | 1497          |
| 128      | 157.9         | 2529.99       | 3032.88       |
| 256      | 299.9         | 6643.0        | 7089.9        |
| 512      | 679.02        | 16555.1       | 16739.9       |
| 1024     | 1516.08       | 40896.93      | 41398.93      |
| 2048     | 4009.91       | 129486        | 129199.9      |
| 4096     | 12242.02      | 442861.9      | 441023.1      |
| 8192     | 40544.23      | 1436512       | 14398112      |

#### 4.3.2 Comparison of the lifting algorithm on UA and OSC cluster using 1 processor

Table A2 Timing in microseconds of lifting algorithm on 1 processors

| <b>N</b> | <b>UA_1Processor</b> | <b>OSC_1 Processor</b> |
|----------|----------------------|------------------------|
| 8        | 33                   | 55                     |
| 16       | 41                   | 66                     |
| 32       | 52                   | 77                     |
| 64       | 80                   | 119                    |
| 128      | 151                  | 198                    |
| 256      | 296                  | 350                    |
| 512      | 649                  | 608                    |
| 1024     | 1557                 | 1315                   |
| 2048     | 4094                 | 3059                   |
| 4096     | 12487                | 7912                   |
| 8192     | 41265                | 20285                  |
| 16384    | 148190               | 62259                  |
| 32768    | 572643               | 203109                 |

#### 4.4 Comparing the gg90 and lifting algorithm

Table A3 Timing in microseconds of gg90 and lifting algorithm on 1 processor

| <b>N</b> | <b>UA_1 gg90</b> | <b>OSC_1 Lift</b> |
|----------|------------------|-------------------|
| 128      | 124              | 202               |
| 256      | 215              | 431               |
| 512      | 437              | 799               |
| 1024     | 848              | 1621              |
| 2048     | 1658             | 3472              |
| 4096     | 3411             | 7018              |
| 8192     | 7225             | 15163             |
| 16384    | 15066            | 31153             |

#### 4.5.1 Real case FFT using 1 processor

Table A4 Timing in microseconds of FFT using 1 processors

| <b>N</b> | <b>UA_1Processor</b> | <b>OSC_1 Processor</b> |
|----------|----------------------|------------------------|
| 128      | 113                  | 123                    |
| 256      | 202                  | 233                    |
| 512      | 431                  | 459                    |
| 1024     | 799                  | 792                    |
| 2048     | 1621                 | 1563                   |
| 4096     | 3472                 | 3008                   |
| 8192     | 7018                 | 5961                   |
| 16384    | 15163                | 12846                  |
| 32768    | 31153                | 27781                  |

#### 4.5.2 Comparisons of the real case FFT using 1 processor

Table A5 Comparing timing in microseconds of FFT using 1 processor

| <b>N</b> | <b>UA_1Processor</b> | <b>OSC_1 Processor</b> | <b>[1]</b> |
|----------|----------------------|------------------------|------------|
| 256      | 202                  | 233                    | 68         |
| 512      | 431                  | 459                    | 204        |
| 1024     | 799                  | 792                    | 486        |
| 2048     | 1621                 | 1563                   | 1170       |
| 4096     | 3472                 | 3008                   | 2800       |
| 8192     | 7018                 | 5961                   | 7000       |
| 16384    | 15163                | 12846                  | 18100      |
| 32768    | 31153                | 27781                  | 46900      |

#### 4.5.3 Complex FFT using 2 processors

Table A6 Timing in microseconds of FFT using 2 processors

| <b>N</b> | <b>UA_2Processor</b> | <b>OSC_2 Processor</b> |
|----------|----------------------|------------------------|
| 128      | 260                  | 151                    |
| 256      | 415                  | 269                    |
| 512      | 721                  | 519                    |
| 1024     | 1266                 | 929                    |
| 2048     | 2401                 | 1704                   |
| 4096     | 4284                 | 3472                   |
| 8192     | 8458                 | 6582                   |
| 16384    | 17256                | 13775                  |
| 32768    | 35675                | 30410                  |

#### 4.5.4 Comparisons of the complex case FFT using 2 processor

Table A7 Comparing Timing in microseconds of complex case FFT on 2 processors

| <b>N</b> | <b>UA_2Processor</b> | <b>OSC_2 Processor</b> | <b>[1]</b> |
|----------|----------------------|------------------------|------------|
| 256      | 415                  | 269                    | 65.5       |
| 512      | 721                  | 519                    | 199        |
| 1024     | 1266                 | 929                    | 456.2      |
| 2048     | 2401                 | 1704                   | 1040       |
| 4096     | 4284                 | 3472                   | 2330       |
| 8192     | 8458                 | 6582                   | 7000       |
| 16384    | 17256                | 13775                  | 17900      |
| 32768    | 35675                | 30410                  | 47000      |

#### 4.5.5 Complex case FFT using 6 processors

Table A8 Timing in microseconds of FFT on 6 processors

| <b>N</b> | <b>UA_6Processor</b> | <b>OSC_6 Processor</b> |
|----------|----------------------|------------------------|
| 256      | 1923                 | 647                    |
| 512      | 2335                 | 838                    |
| 1024     | 4066                 | 1097                   |
| 2048     | 7789                 | 1832                   |
| 4096     | 13401                | 3851                   |
| 8192     | 27205                | 6609                   |
| 16384    | 51167                | 12737                  |
| 32768    | 120313               | 26613                  |

#### 4.5.6 Comparison of complex case FFT using 1, 2 and 6 processors

Table A9 Timing in microseconds of FFT on 1, 2 and 6 processor

| <b>N</b> | <b>OSC_1</b> | <b>OSC_2</b> | <b>OSC_6</b> |
|----------|--------------|--------------|--------------|
| 128      | 123          | 151          | 529          |
| 256      | 233          | 269          | 647          |
| 512      | 459          | 519          | 838          |
| 1024     | 792          | 929          | 1097         |
| 2048     | 1563         | 1704         | 1832         |
| 4096     | 3008         | 3472         | 3851         |
| 8192     | 5961         | 6582         | 6609         |
| 16384    | 12846        | 13775        | 12737        |
| 32768    | 27781        | 30410        | 26613        |

#### 4.5.7 Comparison of complex case FFT in parallel with FFTW 3.2

Table A10 Timing in microseconds of FFT and FFTW3.2

| <b>N</b> | <b>FFT</b> | <b>FFTW 3.2</b> | <b>Performance</b> |
|----------|------------|-----------------|--------------------|
| 2048     | 1832       | 68.96           | X26                |
| 4096     | 3851       | 150.66          | X25                |
| 8192     | 6609       | 328.31          | X20                |
| 16384    | 12737      | 748.81          | X17                |
| 32768    | 26613      | 1650            | X16                |
| 65536    | 61187      | 3980            | X15                |
| 131072   | 129389     | 9570            | X13                |
| 262144   | 291218     | 20910           | X13                |
| 524288   | 622134     | 58260           | X10                |

## APPENDIX B

### C CODE FOR FAST FOURIER TRANSFORMS

//Sept 10 Program is done with the addition of real and complex part

```
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
#include<mpi.h>
#include<math.h>
#include<stdlib.h>
#define PI 3.141592654
#define SQH 0.707106
int ordb[65536];
double sum2[65536];
double poutf[65536];

main (int argc, char *argv[])
{
    int myrank, numprocs,tag=1234,c1;
    int n, i, n1 = 0, m, space, nstart,nstart2,space2, ncounter, m2, j,
k, h1;
    n = atoi (argv[1]);
    double xout[n], J, mytime, x[n], x2[n], x4[n], temp[n], temp1[n],
temp2[n], xbothalf[n], xbothalf2[n], temp5[n], temp6[n], xlout[n],
x2out[n], fxout1[n],fxout2[n],fxout3[n],fxout4[n];

    void dct(double [], int);
    int getlog(int);

    MPI_Status status;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
    MPI_Comm_size (MPI_COMM_WORLD, &numprocs);

    if (myrank == 0)
    {
        fftporder4 (n);

        while(n1<n)
        {
            x[n1] = n1+1;
            n1++;
        }
    }
}
```

```

mytime = MPI_Wtime ();

J = getlog (n);

m = n;
space = 2;
nstart = 2;

for (ncounter = 1; ncounter <= (J - 2); ncounter++)
    {
    sumdiff2 (x, 0, m, n);

    for(i=0;i<n;i++)
        x[i]=sum2[i];

        m2 = m / 2;

    for (i = m2, j = 1; i < m; i++, j++)
        temp[j] = x[i];

    for (k = 0, i = m2, j = 1; i >= (m2 / 2) + 1; i--, j++, k++)
        temp1[k] = temp[i] - temp[j];

    k = 2*(n/4);
    MPI_Send(temp1, k, MPI_DOUBLE, 2, tag, MPI_COMM_WORLD);

    for (k = 0, i = m2 / 2, j = (m2 / 2) + 1; i >= 1, j <= m2;
        i--, j++, k++)
        temp2[k] = temp[i] + temp[j];

    k = 2*(n/4);
    MPI_Send(temp2, k , MPI_DOUBLE, 3, tag, MPI_COMM_WORLD);

    //DCT IMPLEMENTATION

    MPI_Recv (temp1, k , MPI_DOUBLE, 2, tag,MPI_COMM_WORLD,
&status);

        MPI_Recv (temp2, k , MPI_DOUBLE, 3, tag,MPI_COMM_WORLD,
&status);

    for (i = 0; i < (m2 / 2); i++)
        {
        xbothalf[i] = temp2[i];
        xbothalf2[i] = -1 * temp1[i];
        }

    for (i = (m2 / 2), j = (m2/2)-1; i<m2; i++, j--)
        {
        xbothalf[i] = temp2[j];
        xbothalf2[i] = temp1[j];
        }

```

```

nstart2 = nstart;
space2 = space;

    for (i = 0; i < m2,nstart<=n; i++)
    {
        ordb[i] = (ordb[i] / 2);
        fxout1[nstart] = xbothalf[ordb[i]];
        fxout2[nstart] = xbothalf2[ordb[i]];
        nstart=nstart+space;
    }

nstart = nstart2 + (space2 / 2);
m = m / 2;
space = 2 * space2;
}

//Last Step 4 Values
for (i = 0; i < 4; i++)
    temp1[i] = x[i];

sumdiff2 (temp1, 0, 4, n);

for (i = 0; i < 4; i++)
    x[i] = sum2[i];

sumdiff2 (x, 0, 2, n);

fxout1[1] = sum2[0];
fxout2[1] =0;

fxout1[1 + (n / 2)] = sum2[1];
fxout2[1+(n/2)]=0;

for (j = 0, i = 3; i <= 4; i++, j++)
    temp[j] = x[i-1];

fxout1[(n / 4) + 1] = temp[0];
fxout2[(n / 4) + 1] = -1 * temp[1];

fxout1[(n / 4) + (n / 2) + 1] = temp[0];
fxout2[(n / 4) + (n / 2) + 1] = temp[1];

        MPI_Recv (fxout3, n+1 , MPI_DOUBLE, 1, tag,MPI_COMM_WORLD,
&status);
        MPI_Recv (fxout4, n+1 , MPI_DOUBLE, 1, tag,MPI_COMM_WORLD,
&status);

    for (i = 1; i <= n; i++)
    {
        fxout1[i] = fxout1[i] - fxout4[i];
        fxout2[i] = fxout2[i] + fxout3[i];
    }

//          for(i=1;i<=n;i++)

```

```

// printf("%lf %lf\n",fxout1[i],fxout2[i]);

        mytime = MPI_Wtime () - mytime;
        mytime = mytime * 1000000;
        printf ("\nTiming from rank %d is %lfus.\n", myrank,mytime);
    }        //if rank 0 ends here

if(myrank == 2)
{
    k =(n/4)*2;
    J = getlog(n);
    J = J-2;
    for(ncounter=J;ncounter>=1;ncounter--)
    {
        MPI_Recv (temp1, k , MPI_DOUBLE, 0, tag,MPI_COMM_WORLD,
&status);
        c1 = pow (2, ncounter);
        dct(temp1,c1);
        MPI_Send(temp1, k , MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
    }
}

if(myrank == 3)
{
    k =(n/4)*2;
    J = getlog(n);
    J = J-2;

    for(ncounter=J;ncounter>=1;ncounter--)
    {
        MPI_Recv (temp2, k , MPI_DOUBLE, 0, tag,MPI_COMM_WORLD,
&status);
        c1 = pow (2, ncounter);
        dct (temp2, c1);
        MPI_Send(temp2, k , MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
    }
}

if (myrank == 1)
{
    fftporder4 (n);

    while(n1<n)
    {
        x2[n1] = n1+1;
        n1++;
    }

    J = getlog (n);

    m = n;
    space = 2;
    nstart = 2;

```

```

for (ncounter = 1; ncounter <= (J - 2); ncounter++)
    {
    sumdiff2 (x2, 0, m, n);

    for(i=0;i<n;i++)
        x2[i]=sum2[i];

        m2 = m / 2;

    for (i = m2, j = 1; i < m; i++, j++)
        temp[j] = x2[i];

    for (k = 0, i = m2, j = 1; i >= (m2 / 2) + 1; i--, j++, k++)
        temp1[k] = temp[i] - temp[j];

    k = 2*(n/4);
    MPI_Send(temp1, k, MPI_DOUBLE, 4, tag, MPI_COMM_WORLD);

    for (k = 0, i = m2 / 2, j = (m2 / 2) + 1; i >= 1, j <= m2;
        i--, j++, k++)
        temp2[k] = temp[i] + temp[j];

    k = 2*(n/4);
    MPI_Send(temp2, k, MPI_DOUBLE, 5, tag, MPI_COMM_WORLD);

    MPI_Recv (temp1, k , MPI_DOUBLE, 4, tag, MPI_COMM_WORLD,
&status);

    MPI_Recv (temp2, k , MPI_DOUBLE, 5, tag, MPI_COMM_WORLD,
&status);

    for (i = 0; i < (m2 / 2); i++)
        {
        xbothalf[i] = temp2[i];
        xbothalf2[i] = -1 * temp1[i];
        }

    for (i = (m2 / 2), j = (m2/2)-1; i<m2; i++, j--)
        {
        xbothalf[i] = temp2[j];
        xbothalf2[i] = temp1[j];
        }

    nstart2 = nstart;
    space2 = space;

        for (i = 0; i < m2, nstart<=n; i++)
            {
            ordb[i] = (ordb[i] / 2);
            fxout3[nstart] = xbothalf[ordb[i]];
            fxout4[nstart] = xbothalf2[ordb[i]];
            nstart=nstart+space;
            }

```

```

        nstart = nstart2 + (space2 / 2);
        m = m / 2;
        space = 2 * space2;
    } //for counter endsa

//Last Step 4 Values

for (i = 0; i < 4; i++)
    temp1[i] = x2[i];

sumdiff2 (temp1, 0, 4, n);

for (i = 0; i < 4; i++)
    x2[i] = sum2[i];

sumdiff2 (x2, 0, 2, n);

fxout3[1] = sum2[0];
fxout4[1] =0;

fxout3[1 + (n / 2)] = sum2[1];
fxout4[1+(n/2)]=0;

for (j = 0, i = 3; i <= 4; i++, j++)
    temp[j] = x2[i-1];

fxout3[(n / 4) + 1] = temp[0];
fxout4[(n / 4) + 1] = -1 * temp[1];

fxout3[(n / 4) + (n / 2) + 1] = temp[0];
fxout4[(n / 4) + (n / 2) + 1] = temp[1];

        MPI_Send(fxout3, n+1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
        MPI_Send(fxout4, n+1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);

    } //if rank 1 ends here
if(myrank == 4)
{
    k =(n/4)*2;
    J = getlog(n);
    J = J-2;
    for(ncounter=J;ncounter>=1;ncounter--)
    {
        MPI_Recv (temp1, k , MPI_DOUBLE, 1, tag,MPI_COMM_WORLD,
&status);
        c1 = pow (2, ncounter);
        dct(temp1,c1);
        MPI_Send(temp1, k , MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
    }
}

if(myrank == 5)
{
    k =(n/4)*2;

```

```

J = getlog(n);
J = J-2;

for(ncounter=J;ncounter>=1;ncounter--)
{
    MPI_Recv (temp2, k , MPI_DOUBLE, 1, tag,MPI_COMM_WORLD,
&status);
    c1 = pow (2, ncounter);
    dct (temp2, c1);
    MPI_Send(temp2, k , MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
}
}
MPI_Finalize ();
return 0;
}

//*****DCT IMPLEMENTATION*****//
void dct (double xout[], int z1)
{
    long int n = 0;
    int L, m2, myrank, pcounter, l1, tag =1234, h1, h3, h4, h5, h6, K1,
h16, h7, h8, numprocs, i, j, cj, m, k,
    count;
    long int j4r[z1];
    double c, s, J1, angnum, mrs, ang, mult, C[z1], S[z1], C2b[z1],
S2b[z1],
    angle, angle1, temp[z1], xin[z1], temprs[z1], v[z1], S2[z1],
p1[z1],
    C2c[z1], S2c[z1], temp2rs[z1], p2[z1], mytime, temp1[z1],
block[z1],
    pair1[z1], pair2[z1], temp2[z1], temp3[z1], temp4[z1],
p[z1],xpin[z1];

    while (n < z1)
    {
        n++;
    }

    if (n == 2)
    {
        angle = PI / 8;
        c = cos (angle);
        s = sin (angle);

        xpin[0] = xout[0];
        xpin[1] = xout[1];

        xout[0] = (c * xpin[0]) + (s * xpin[1]);
        xout[1] = (-s * xpin[0]) + (c * xpin[1]);
    }
    else if (n == 4)
    {
        c = cos (PI / 16);
        s = sin (PI / 16);
    }
}

```

```

temp[1] = xout[0];
temp[2] = xout[3];
temp[3] = xout[2];
temp[4] = xout[1];

xin[0] = c * temp[1] + s * temp[2];
xin[1] = (-s * temp[1]) + c * temp[2];

c = cos ((5 * PI) / 16);
s = sin ((5 * PI) / 16);

xin[2] = c * temp[3] + s * temp[4];
xin[3] = (-s * temp[3]) + c * temp[4];

sumdiff2 (xin, 0, 4, z1);

temp[0] = sum2[2] * SQH;
temp[1] = sum2[3] * SQH;

xout[0] = sum2[0];
xout[1] = sum2[1];
xout[2] = temp[0] + temp[1];
xout[3] = -temp[0] + temp[1];
}
else
{
for (j = 1; j <= (n / 2); j++)
{
angnum = 4 * (j - 1) + 1;
mult = PI / (n * 4);
ang = angnum * mult;
C[j] = cos (ang);
S[j] = sin (ang);
}

for (j = 1; j <= (n / 4); j++)
{
cj = 2 * (j - 1);
m = n - 1;
//0,7,4,3 in the first iteration and 2 5 6 1 in the second
iteration
pair1[0] = xout[cj];
pair1[1] = xout[(m - cj)];
pair2[0] = xout[(cj + (n / 2))]; //xtemp(3:4)
pair2[1] = xout[(m - (cj + (n / 2)))]];

v[0] = (C[j] * pair1[0]) + (S[j] * pair1[1]);
v[1] = (-S[j] * pair1[0]) + (C[j] * pair1[1]);

v[2] = (C[j + (n / 4)] * pair2[0]) + (S[j + (n / 4)] *
pair2[1]);
v[3] = (-S[j + (n / 4)] * pair2[0]) + (C[j + (n / 4)] *
pair2[1]);

sumdiff2 (v, 0, 4, z1);

```

```

p1[(2 * j) - 2] = sum2[0];
p1[(2 * j) - 1] = sum2[1];
p2[(2 * j) - 2] = sum2[2];
p2[(2 * j) - 1] = sum2[3];
}

pcounter = 1;
while (pcounter <= 2)
{
  if (pcounter == 1)
  {
    for (i = 0, j = 1; i < n / 2; i++, j++)
    {
      p[i] = p1[i];
      count++;
    }
  } //if ends here
  else
  {
    count = 0;
    for (j = 1, i = 0; i < n / 2; i++, j++)
    {
      p[j] = p2[i];
      count++;
    }
    for (i = 1; (4 * (i - 1) + 1) <= n / 2; i++)
    {
      angle = 4 * (i - 1) + 1;
      angle1 = angle / n;
      C2b[i] = cos (PI * angle1);
      S2b[i] = sin (PI * angle1);
    }
    gg90cs (p, C2b, S2b, count, z1);

    for (i = 0, j = 1; j <= n / 2; j++, i++)
    {
      p[i] = poutf[j];
    }
  } //else ends here

  J1 = getlog (n);
  K1 = J1 - 2;
  m = n / 2;

  for (l1 = 1; l1 <= K1; l1++)
  {
    m2 = pow (2, (l1 - 1));
    L = m / m2;

    for (j = 1; (4 * (j - 1)) < L / 2; j++)
    {
      mrs = 4 * (j - 1) + 1;
      mrs = mrs / L;
      C2c[j] = cos (PI * mrs);
    }
  }
}

```

```

    S2c[j] = sin (PI * mrs);
}

for (k = 1; k <= m2; k++)
{
    h1 = L * (k - 1);
    h3 = (L * k) - 1;
    h6 = L * (k - 1);
    h16 = 0;
    while (h1 <= h3)
    {
        block[h1] = p[h1];
        h16++;
        h1++;
    }
    sumdiff2 (block, h6, h3 + 1, z1);

    for (i = 0; i < h16; i++)
    {
        temp2[i] = sum2[i];
    }

    h4 = (L / 2);

    for (i = 1; h4 < L; h4++, i++)
    {
        temp3[i] = temp2[h4];
    }
    gg90cs (temp3, C2c, S2c, L / 2, z1);

    for (i = 1, h5 = L / 2; h5 < L; i++, h5++)
    {
        temp2[h5] = poutf[i];
    }

    h7 = L * (k - 1);
    h8 = (L * k) - 1;

    for (i = 0; h7 <= h8; i++, h7++)
    {
        p[h7] = temp2[i];
    }
}
if (pcounter == 1)
{
    for (i = 0; i < n / 2; i++)
        xout[i] = p[i];
}
else
{
    for (j = 0, i = n / 2; i < n; i++, j++)
        xout[i] = p[j];
}
}

```

```

        pcounter++;
    }
}
//dct ends here

gg90cs (double pin1[], double C2b[], double S2b[], int count, int z1)
{
    int m, j, mj, mj2, i;
    double temp[z1], temp1[z1], pout3[z1], pout4[z1];
    m = count;

    if (m == 2)
    {
        poutf[1] = (pin1[1] + pin1[2]) * S2b[1];
        poutf[2] = (-pin1[1] + pin1[2]) * S2b[1];
    }
    else
    {
        for (j = 1; j <= m / 4; j++)
        {
            mj = (2 * j) - 1;

            temp[1] = pin1[mj];
            temp[2] = pin1[mj + 1];

            pout3[mj] = (C2b[j] * temp[1]) + (S2b[j] * temp[2]);
            pout3[mj + 1] = (-S2b[j] * temp[1]) + (C2b[j] * temp[2]);

            temp1[1] = pin1[((m / 2) + mj)];
            temp1[2] = pin1[((m / 2) + mj + 1)];

            pout4[mj] = (-S2b[j] * temp1[1]) + (C2b[j] * temp1[2]);
            pout4[mj + 1] = (-C2b[j] * temp1[1]) + (-S2b[j] * temp1[2]);
        }

        for (i = 1; i <= m / 2; i++)
        {
            poutf[i] = pout3[i];
        }
        for (i = 1, j = (m / 2 + 1); i <= m; i++, j++)
        {
            poutf[j] = pout4[i];
        }
    }
    //else ends
}

////////////////////////////////////
sumdiff2 (double xy3[], int r1, int r2, int z1)
{
    int f1, r, i, j, h, k, l, size1;
    size1 = z1;
    double x1[size1], x2[size1], xy2[size1];

    r = r2 - r1;

```

```

f1 = (r / 2);

for (i = r1, k = 0; i < r2; i++, k++)
    xy2[k] = xy3[i];

for (i = 0, k = 0; i < f1; i++, k++)
    x1[k] = xy2[i];

for (h = 0, j = f1; j < r; j++, h++)
    x2[h] = xy2[j];

for (k = 0; k < f1; k++)
    sum2[k] = x1[k] + x2[k];

for (l = f1, k = 0; l < r; l++, k++)
    sum2[l] = x1[k] - x2[k];
}
fftporder4 (int n1)
{
    double J;
    double veca[n1];
    veca[0] = 0;
    veca[1] = 2;
    veca[2] = 1;
    veca[3] = 3;
    double vecb[n1], z1[n1];
    int klev, i, nv, s1, k;
    nv = 2;
    int n=n1;
    J = getlog(n);
    for (klev = 1; klev <= J - 2; klev++)
        {
            nv *= 2;

            for (i = 0; i <= (n1 / 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];

            for (i = 0; i < (nv * 2); i++)
                {
                    veca[i] = vecb[i];
                }
        }
    for (i = 0; i < n1; i++)
        {

```

```
        ordb[i] = vecb[i];
    }
}
int getlog(int n){
    if(n==8) return(3);
    else if(n==16)return(4);
    else if(n==32)return(5);
    else if(n==64)return(6);
    else if(n==128)return(7);
    else if(n==256)return(8);
    else if(n==512)return(9);
    else if(n==1024)return(10);
    else if(n==2048)return(11);
    else if(n==4096)return(12);
    else if(n==8192)return(13);
    else if(n==16384)return(14);
    else if(n==32768)return(15);
    else if(n==65536)return(16);
    else return(0);
}
```