Implementation and performance evaluation of parallel FFT algorithms

Somasundaram Meiyappan
Matriculation No.: HT023601A
School of Computing
National University of Singapore
Singapore
somasund@comp.nus.edu.sg

Abstract

Fast Fourier Transform (FFT) algorithms are widely used in many areas of science and engineering. Some of the most widely known FFT algorithms are Radix-2 algorithm, Radix-4 algorithm, Split Radix algorithm, Fast Hartley transformation based algorithm and Quick Fourier transform. In this paper, the first three algorithms listed are implemented in the sequential and MPI (message passing interface) parallel forms and their performances are compared. The algorithms are implemented in two parallel styles of algorithms such that we reduce communication overhead. We also see the effect of the inter-connection network on the performance of the algorithms.

1. Introduction

One of the most widely used technique in science and engineering is the concept of Fourier Transform and other algorithms based on it. In signal processing, it is primarily used to convert an input signal in time domain into frequency domain and vice-versa. In the world of digital, signals are sampled in time domain. So, we have Discrete Fourier Transform (DFT) in the digital world. DFT is applied on a discrete input signal and we get the frequency characteristics of the signal as the output. Performing inverse DFT, which has a mathematical form very similar to the DFT, on the frequency domain result gives back the signal in the time domain. This means that the signal when converted into frequency domain will give us the various frequency components of the input signal and then can be used to remove certain unwanted frequency components. This concept can be used in image or audio compression and filters on communication signals to name a few.

Discrete Fourier Transform is a very computationally intensive process that is based on summation a finite series of products of input signal values and trigonometric functions. Its time complexity of the algorithm in $O(n^2)$.To increase the performance, several

algorithms were proposed which can be implemented in hardware or software. These set of algorithms are known as Fast Fourier Transforms (FFT). The first major FFT algorithm was proposed by Cooley and Tukey. Many FFT algorithms were proposed with a time complexity of *O(nlogn).* Some of them are Radix-2 algorithm, Radix-4 algorithm and Split Radix algorithm. In this paper, we discuss ways of parallelizing these algorithms to reduce the communication overhead.

2. Background

2.1. Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is a based on the sum of the product of the input signal values and the trigonometric functions. It deals with real numbers and is bounded. The DFT, X(k) of a finite length discrete signal, x(n) is computed by the equation given in figure 1 below. It is bounded because it assumes that the signal is periodic with a period equal to the length of the sequence x(n).

$$\mathbf{X(k)} = \sum_{\mathbf{n=0}}^{\mathbf{N-1}} \mathbf{x(n).e^{-j2pkn/N}} \qquad 1.a$$

$$\mathbf{x(n)} = \sum_{\mathbf{k=0}}^{\mathbf{N-1}} \mathbf{X(k).e^{j2pkn/N}} \qquad 1.b$$

Figure 1: Mathematical form of the Discrete Fourier Transform and its inverse

In computing the DFT of a finite length signal with length N, we require $4N^2$ multiplications and N(4N-1) additions. This gives us a time-complexity of $O(N^2)$. This means that for larger values of N, the computational time increases exponentially, which is not desirable. This sets the stage for Fast Fourier Transforms (FFT).

2.2. FFT Algorithms

To make the DFT operation more practical, several FFT algorithms were proposed. The fundamental approach for all of them is to make use of the properties of the DFT operation itself. All of them reduce the computational cost of performing the DFT on the given input sequence.
From the equation 1.a in figure 1,

$$W_N^{\mathbf{n}} = e^{-j2\pi kn/N}$$

This value of W^n is referred to as the *twiddle factor* or *phase factor*. This value of twiddle factor being a trigonometric function over discrete points around the 4 quadrants of the two dimensional plane has some symmetry and periodicity properties.

$$\text{Symmetry property: } W_N^{k+N/2} = -W_N^k$$

$$\text{Periodicity property: } W_N^{k+N} = W_N^k$$

Figure2: Periodicity and symmetry property

Using these properties of the twiddle factor, unnecessary computations can be eliminated.

Another approach that can be used is the divide-and-conquer approach. In this approach, the given single dimensional input sequence of length, N, can be represented in a two-dimensional form with M rows and L columns with N = M x L. It can be shown that DFT that is performed on such a representation will lead to lesser computations, N(M+L+1) complex additions and N(M+L-2) complex additions. Please note that this approach is applicable only when the value of N is composite.

2.2.1. Radix-2 FFT algorithm

This algorithm is a special case of the approaches described earlier in which N can be represented as a power of 2 i.e., $N = 2^v$. This means that the number of complex additions and multiplications gets reduced to N(N+6)/2 and $N^2/2$ just by using the divide-and-conquer approach. When we also begin to use the symmetry and periodicity property of the twiddle factor, it can be shown that the number of complex additions and multiplications can be reduced to $N\log_2 N$ and $(N/2)\log_2 N$ respectively. Hence, from a $O(N^2)$ algorithm, the computational complexity has been reduced to O(NlogN).

The entire process is divided into $\log_2 N$ stages and in each stage N/2 two-point DFTs are performed. The computation involving each pair of data is called a *butterfly*. Radix-2 algorithm can be implemented as Decimation-in-time (M=N/2 and L=2) or Decimation-in-frequency (M=2 and L=N/2) algorithms. Figure 3 below gives the decimation-in-frequency form of the Radix-2 algorithm for an input sequence of length, N=8. Figure 4 below gives the decimation-in-time form of the same.

2.2.2. Radix-4 FFT algorithm

This algorithm is similar to Radix-2 algorithm; but here, four point DFTs are performed instead of two point DFTs as in Radix-2 algorithm. This also means that the length of the input sequence, N, should be a power of 4, 4^v. It can be shown that the number of complex multiplications and additions have been reduced to $(3N/8)\log_2 N$ and $(3N/2)\log_2 N$ respectively. Thus, it is more computationally efficient than Radix-2 FFT algorithm.

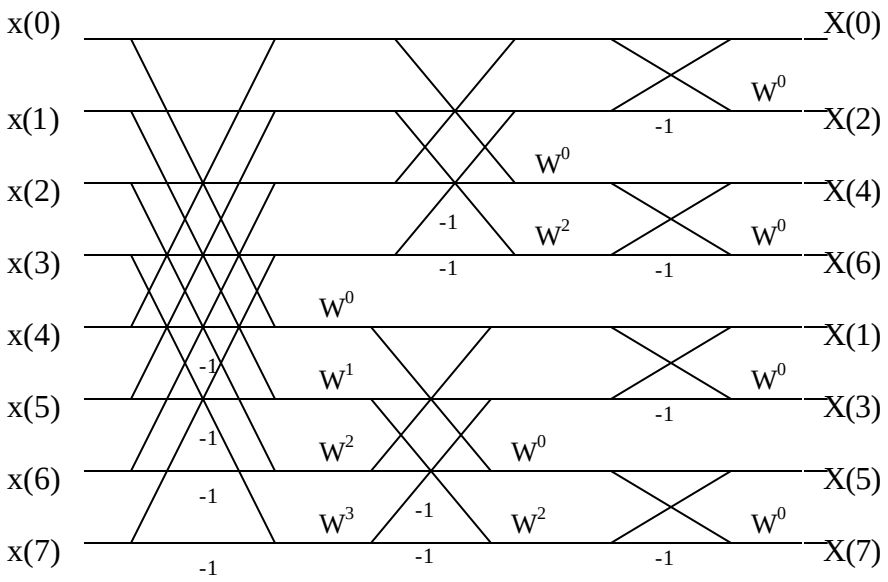


Figure 3: 8-point Radix-2 FFT: Decimation in frequency form

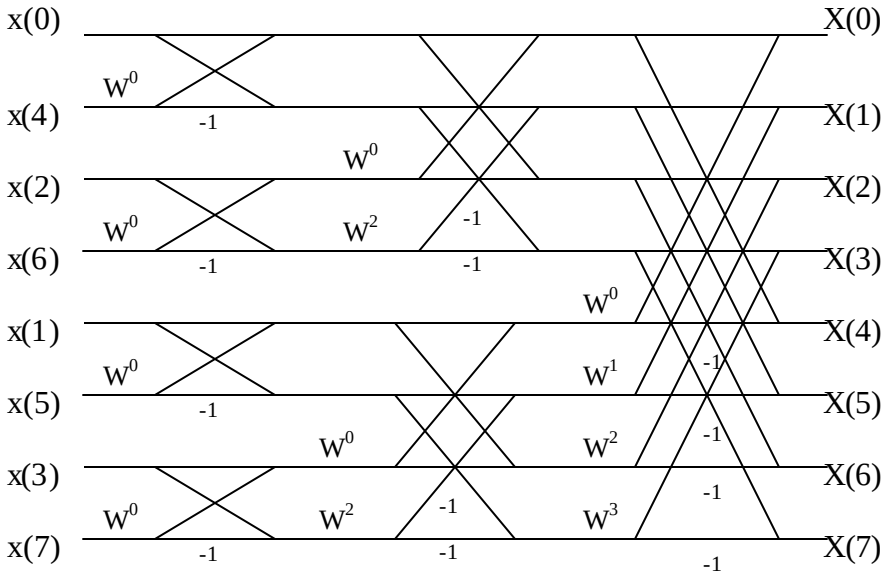


Figure 4: 8-point Radix-2 FFT: Decimation in time form

2.2.3. Split Radix FFT algorithm

Looking at figure 4, it can be observed that all the even numbered points of the DFT can be performed independent to the odd numbered points. This property is used in Split Radix FFT algorithm. As the name suggests, part of the algorithm is computed using Radix-2 algorithm and other part is computed using Radix-4 algorithm. As seen earlier, Radix-4 algorithm is more computationally efficient than the Radix-2 algorithm; but has the disadvantage that length of the input sequence should be a power of 4. The Split Radix algorithm brings together the advantages of the two algorithms.

2.2.4. Comparison

It can be shown that of the sequential form of the three algorithms, Radix-4 is the fastest and Radix-2 is the slowest. Split Radix performs somewhere in the middle. For more literature on these, [1] is a good reference.

3. MPI Parallel FFT algorithms

3.1. Approaches to parallelization

There are many forms of parallel systems available viz., shared memory multiprocessors and message based multi-processors. With the increasing popularity of clusters, MPI (message passing interface) has become a popular form of writing parallel programs for massively parallel multi-processors. Hence, we discuss here algorithms for MPI based systems.

3.1.1. All butterflies in parallel

This approach would mean that all butterfly computations can be performed in parallel. As we see in figure 3, all butterflies in a stage can be performed in parallel and then at the end of the stage, the results can be gathered. Now all nodes can perform computation on the result of the first stage in parallel and output of the second stage can be gathered again and so on. While this provides maximum scope for parallelism, there is a significant communication overhead in this process as at the end of each stage all the processors should communicate with each other and the system should not proceed to the next stage until all the output from the first stage is generated. Moreover, the communication pattern among the processors is also not uniform. This adds to the complexity of the system. Hence, this algorithm is not implemented.

3.1.2. Scope for parallelism increases with stages

Let us consider the decimation-in-frequency form of the length 8 Radix-2 algorithm. In the first stage, all the data is mingled and we have to perform a single 8-point DFT in the first stage. In the second stage, it visibly breaks into two separate 4-point DFTs and in the third stage, we have eight separate 2-point DFTs. Michael Balducci et al, in [2], have used an algorithm that partly resembles this and the previous algorithm in section 3.1.1. Figure 5 below shows us the parallel composition of this algorithm. The text in the oval is

the elements on which computation is performed and the number below the oval is the processor that would perform the computation. Solid lines indicate communication.

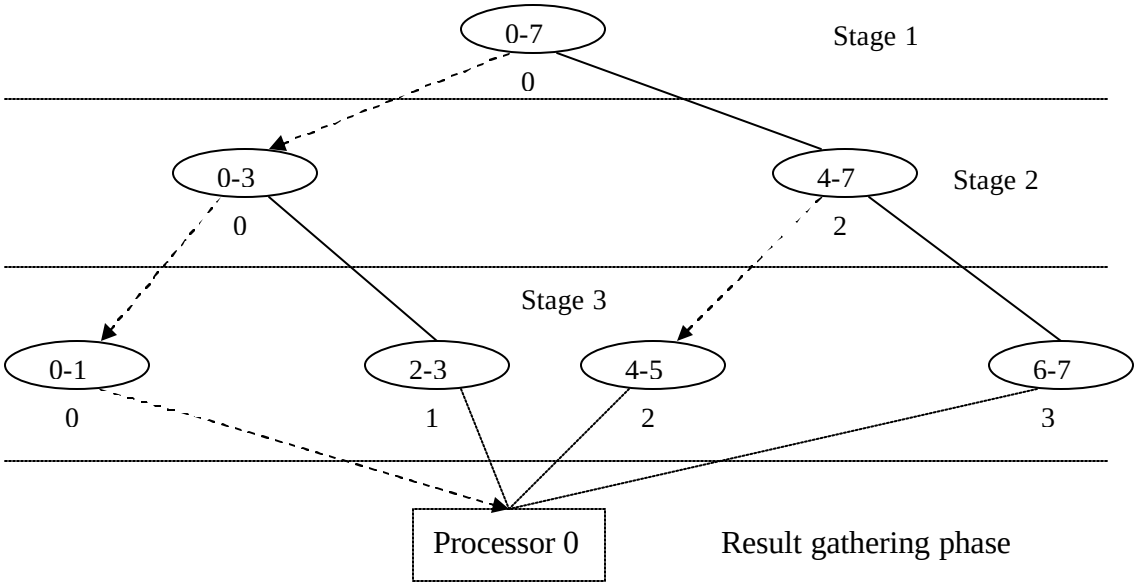


Figure 5: Parallel composition for 8-point Radix-2 FFT in Decimation in frequency form on a 4 processor system

In the decimation-in-time form, the arrows in the above figure need to be reversed and the stages should be numbered from bottom-up. Please note that in this, we have more parallelism in the first stage and the number of divisions clearly decreases by a factor of two. So, this is just the inverse of decimation-in-frequency form. But the advantage of this form is that the result is already in the master node by the end of the last stage whereas in the case of decimation-in-frequency form, we need a result gathering phase after the last FFT stage.

The disadvantage of this approach is that while only one processor will be active in the first stage of the decimation-in-frequency form, all the other available processors will be idle in the first stage. But the number of processors used will be increased in the subsequent stages.

3.1.3. Truly parallel computation: "Communicate twice" algorithm

This algorithm is applicable only to the decimation-in-frequency form. As the name suggests, this algorithm will reduce the communication overhead involved. In section 3.1.2, we saw that processor 2 waits for the result of the first stage from processor 0 before proceeding with stage 2 computations. Since processor 2 already is aware of the data set that it is waiting for, it could as well perform the same computation that processor 0 is performing and then use the data set for its stage 2 computations. In this approach, we need an initial broadcast of all the input data; but such a broadcast is not needed in the approach mentioned in 3.1.2. Figure 6 below represents the parallel

computations performed by the processors in various stages. It can be noted that some processors would be performing redundant computations as some other processors will also be computing the same operation on the data set. But this approach will be very attractive for fast processor nodes connected with a relatively higher latency interconnection network.

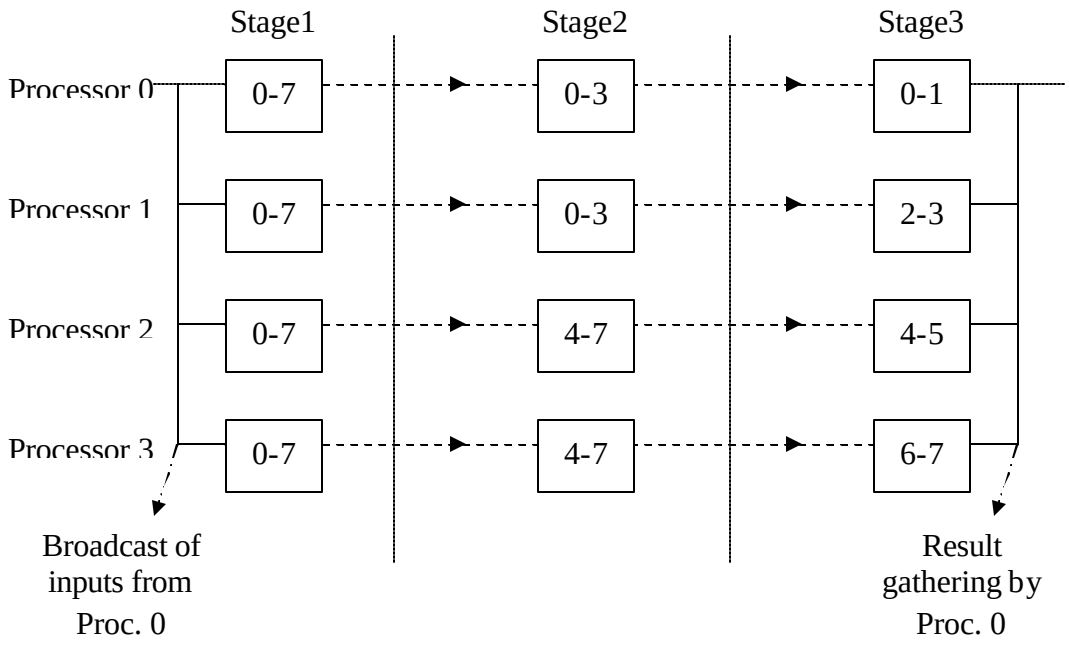


Figure 6: Parallel communication and computational structure for 8-point DIF Radix-2 FFT algorithm implemented as described in section 3.1.3

3.2. Implementation

The algorithms, mentioned in sections 3.1.2 and 3.1.3, have been implemented in C++ using Standard Template Libraries (STL) and Message Passing Interface (MPI).

It can be noted form figure 5 that we divide the inputs to a stage into two sets in the case of Radix-2 or Split Radix algorithm and four sets in the case of Radix-4 algorithm. But this does not mean that this algorithm can be implemented using 2^v or 4^v processors only. It can be implemented using any number of processors. When we cannot divide the data elements at any stage, then the processor that was supposed to divide the data set will continue to perform computations on the entire data set from the previous stage. This can be seen in figure 7 below, which is the 3 processor version of figure 5. In this approach, we map out the entire communication map for each stage for all the processors before beginning to compute the FFT. Blocking MPI communication was used instead of non-blocking communication as it will not make any difference in performance in this case and just increase the amount of memory required because of the requirements of MPI::COMM_WORLD.Isend() method.

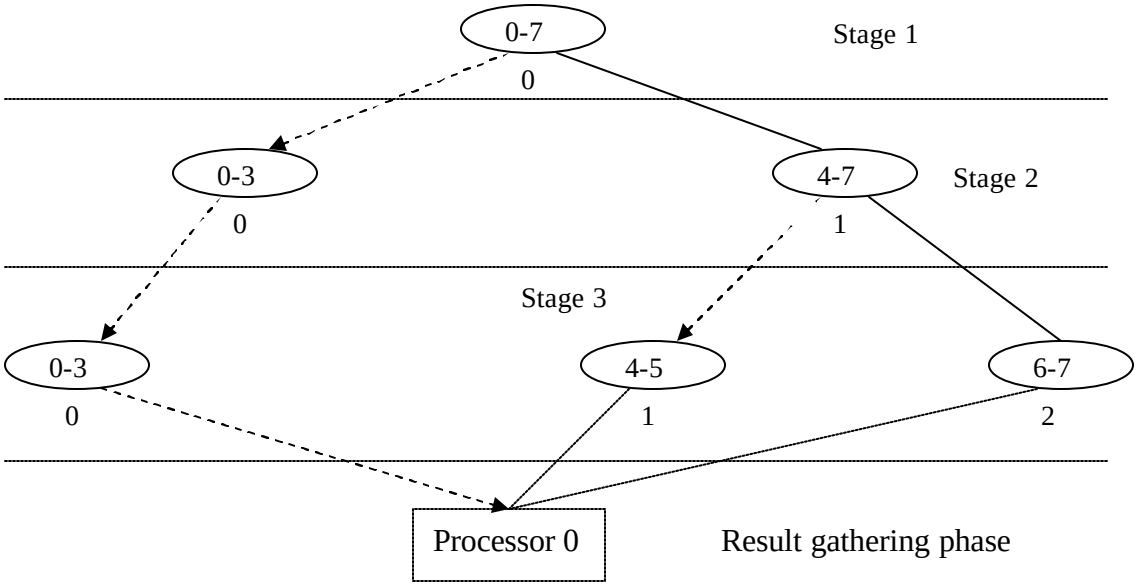


Figure 7: Parallel composition for 8-point Radix-2 FFT in Decimation in frequency form on a 3 processor system

For the approach in 3.1.3, we initially compute just the data elements or rather results expected from all the processors at the end of the last stage. The determination if a division should be executed in a particular stage by a processor is done on the fly by the FFT kernel. Computations on a division can be done in two ways i.e., the DFT operation can be either computed for both the upper and lower half of the division or computed for either the lower or the upper half of the division. This would save us some additions and multiplications. The results in this paper are only based on the former way of implementation. The latter way, which will further improve performance, can be done as a future work. After the last stage, we can perform a MPI::COMM_WORLD.Gatherv() to receive the various result from the nodes.

4. Results

4.1. Measurement method

The main and the only criteria used here for benchmarking the algorithms is the time taken for computation and communication. We use MPI::COMM_WORLD.Wtime() to measure the wall-time. We include all computations and communication involved including the initial broadcast and the final result gathering phase, if applicable. The laying out of the communication map is also included in the calculation of the total time taken by the process. Thus, all overheads involved in performing a parallel computation are taken into account when measuring the time except the initial loading of the program in the processors.

4.2. Comparison of the two parallelizing algorithms

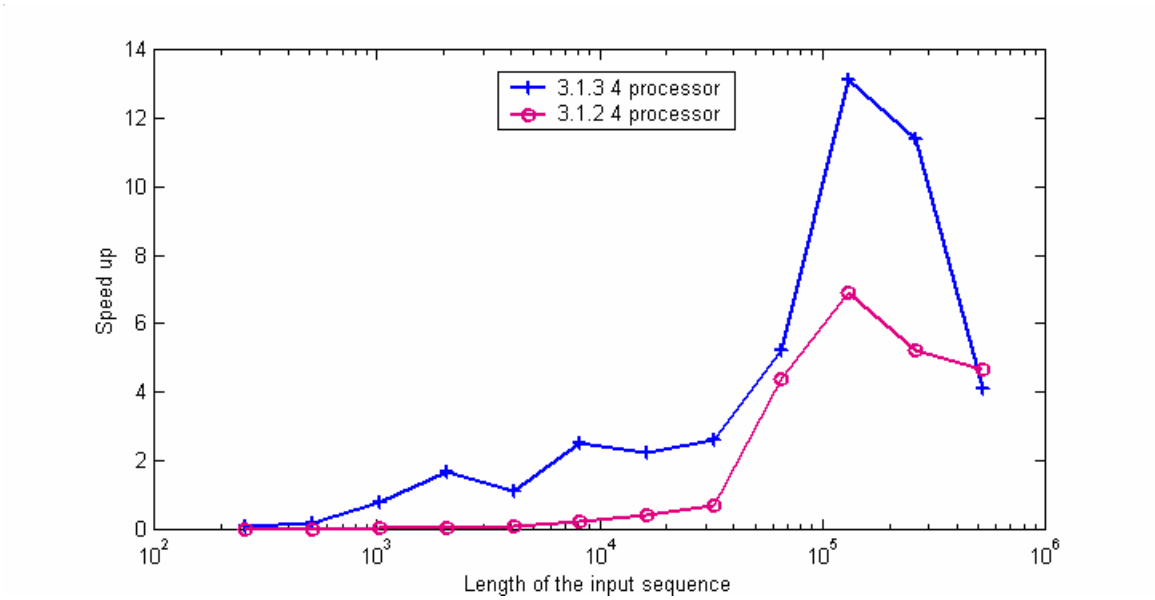


Figure 8: Comparison of the Radix-2 algorithm implemented in 4 processor system with the algorithms in section 3.1.3 and section 3.1.2. We can see that the *Communicate twice* algorithm clearly outperforms the other one.

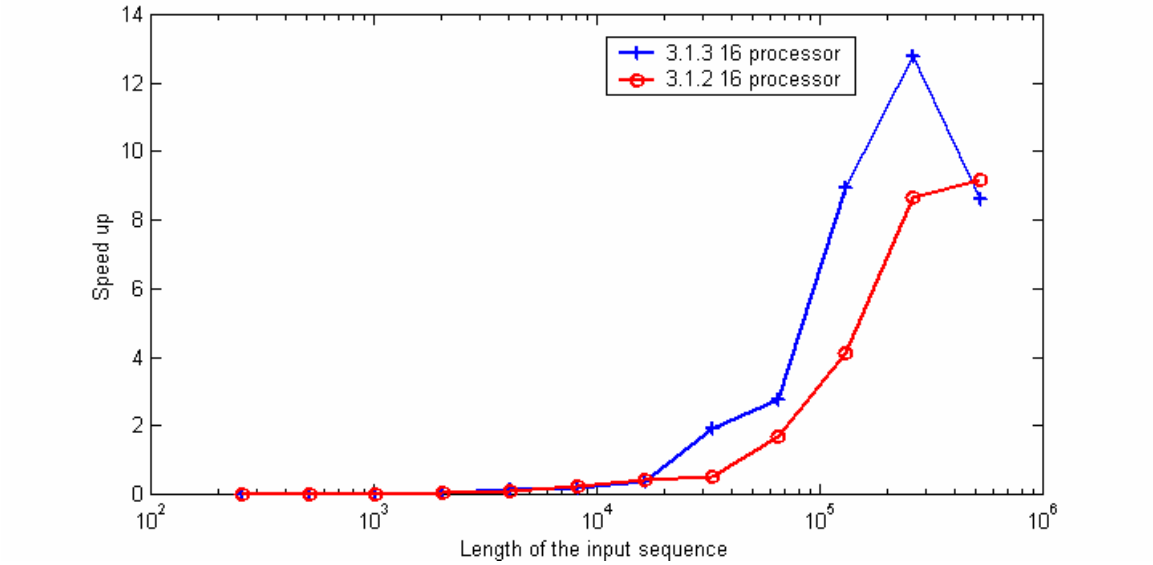


Figure 9: Comparison of the Radix-2 algorithm implemented in 4 processor system with the algorithms in section 3.1.3 and section 3.1.2. We can see that the *Communicate twice* algorithm clearly outperforms the other one. Please note that when we increase the number of processors from 4 to 16, we need to wait for a little longer before it outperforms the single processor implementation. In the 4 processor system, we achieve this when the input sequence is of length 2048; but in a 16 processor system, we achieve that only when length of the input sequence is 32768.

In both the above cases, it has to be noted that when the input size reaches 524288, the tree based algorithm described in section 3.1.2 slightly outperforms the other algorithm.

4.2. Performance of Radix-2, Radi-4 and Split Radix FFT algorithms

From the graphs in the previous section, we see that the *Communicate twice* algorithm in section 3.1.3 works better in the sample space than the one described in section 3.1.2. So, in this section, we compare the above three FFT algorithms when their parallel implementation is based on the *"communicate twice"* algorithm. The communication medium used was *Gigabit Ethernet*. The graphs indicate the speed-up that we obtain using an *np*-processor based cluster when we vary the input size. The blue horizontal line with dots, represent the speed-up for the single processor system which is always 1.

In all the three graphs, it can be observed that for lower values of the input size, less than 256, the single-processor implementation clocks a better time than all the multi-processor system. Beyond an input size of 256, the 2-processor implementations of all three algorithms begin to match and decrease the time taken for computation of the FFT by the single processor implementation. At this point, the computational and communication loads match the computational load of the single processor implementations. So, for lesser input sizes, communication became the overhead. It can also be noted that for different values of *np*, the point of cross-over is different. The higher the number of processors used, the higher is the minimum input size needed to make the system profitable when compared to the single processor implementations. This is true for all the three algorithms.

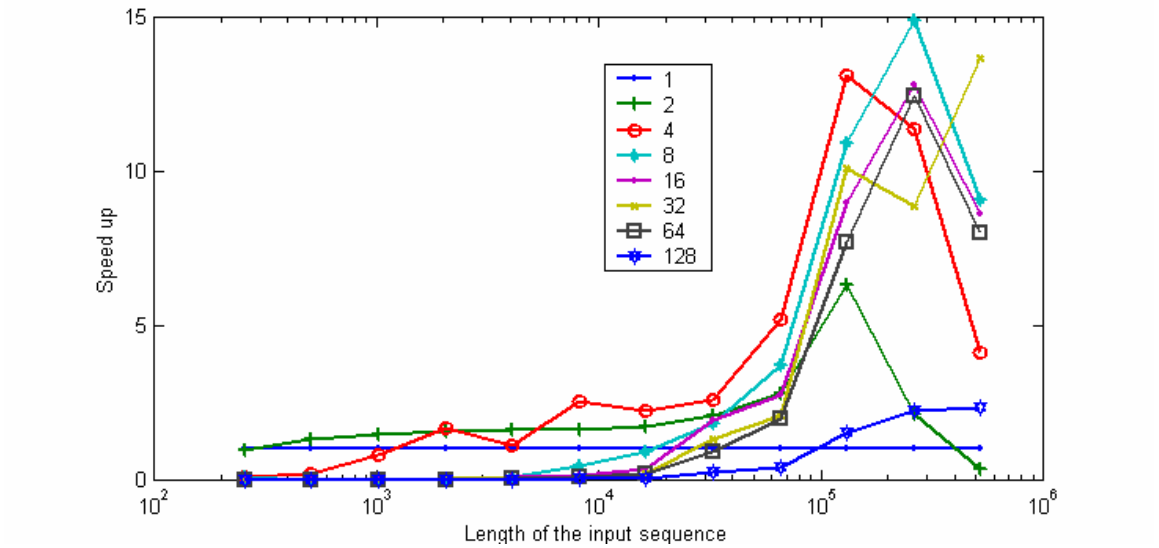


Figure 10: Radix-2 FFT algorithm implemented using *"Communicate twice"* (section 3.1.3) algorithm

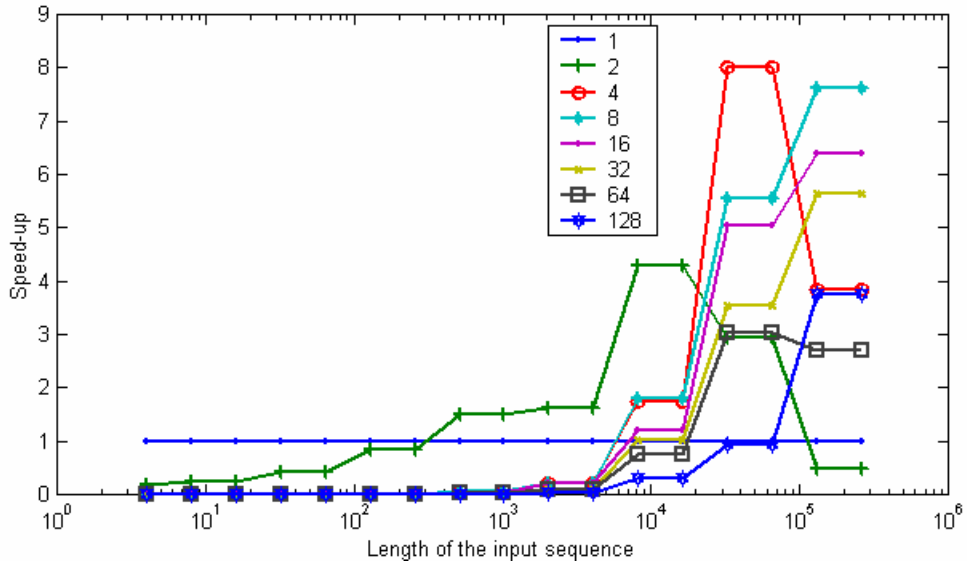


Figure 11: Radix-4 FFT algorithm implemented using *"Communicate twice"* (section 3.1.3) algorithm

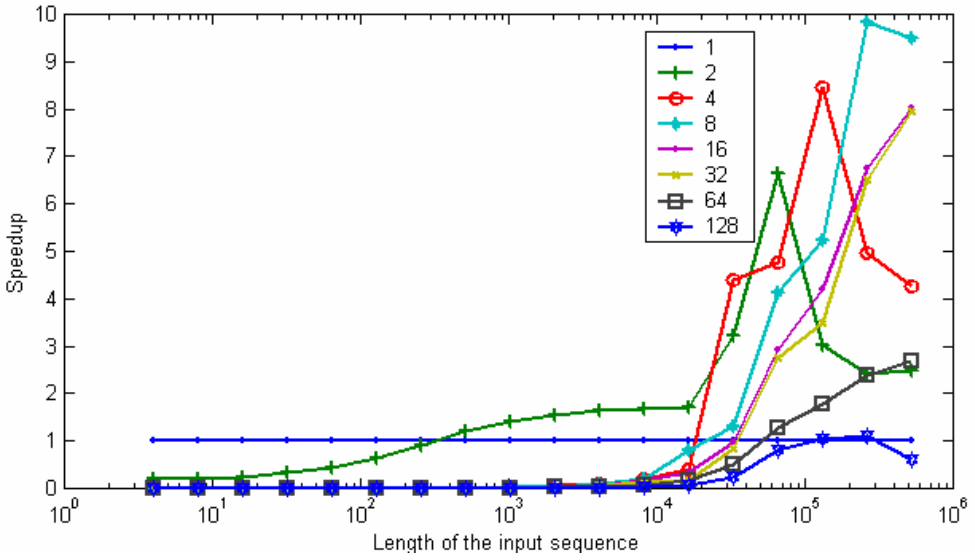


Figure 12: Split Radix FFT algorithm implemented using *"Communicate twice"* (section 3.1.3) algorithm

Please note the contrast between the performance comparisons of the parallel algorithms in [2] and in this paper. [2] mentions that no speed-up was achieved even when the input size reaches 8192. Some of the reasons could be the algorithm itself which places too much communication overhead or it could be the communication medium or protocols that was available during that time (1996).

For a given value of *np* in each algorithm, has a sweet-spot. This sweet spot though slightly varies between the three algorithms; but almost falls within a certain range. The table 1 (below) lists the number of processors to use when given a particular input size. This could actually vary with different communication mediums and processor architecture.

Input size range	**Number of processors to use**
1-256	1
256 -16384	2
$16384 – 1.5x10^5$	4
$1.5x10^5 – 1x10^6$	8

Table 1: Static heuristic to determine the number of processors to use given the input size in a MPI based cluster of Pentium IV machines

Another interesting observation that one could make is that the speed-up obtained in Radix-2 algorithm is higher than the speed-up obtained on Radix-4 and Split Radix algorithm when increasing the number of processors. Please note that the sequential version of Radix-4 FFT algorithm performs better than the Split Radix and Radix-2 algorithms. The number of stages in a Split Radix and Radix-4 implementation for an input of size N is $\log_2 N$ whereas for the Radix-4 implementation, it is $\log_4 N$. Both our parallelizing approaches are built on the fact that parallelism increases with increasing number of stages and divisions. Also, the Radix-4 and Split Radix implementations use STL classes to dynamically compute and store the division to execute in each stage; but Radix-2 does not. This also cost the Radix-4 and Split Radix implementations dearly.

4.3. Different communication medium: Myrinet Vs Gigabit Ethernet

This graph below gives us a comparison between Myrinet and Gigabit Ethernet for an input of size 16384. It can be noted that when the communication medium improves, the performance of the *"communicate twice"* algorithm increases communication tremendously. The weight on the communication overhead in determining the speed-up reduces.

5. Conclusions

Fast Fourier Transform (FFT) is used widely in many scientific, engineering and mathematical applications. In same cases, it is used to analyze a huge set of input data. Hence, parallel FFT algorithms are desirable. Parallelizing the sequential and simple FFT algorithms will be beneficial to control code complexity and minimize execution time of the process.

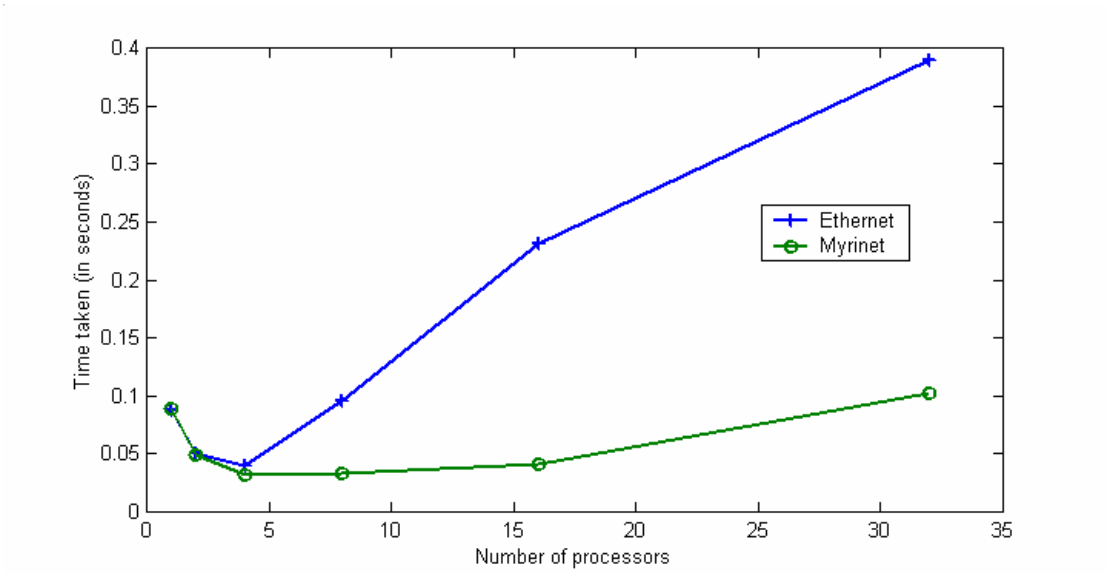


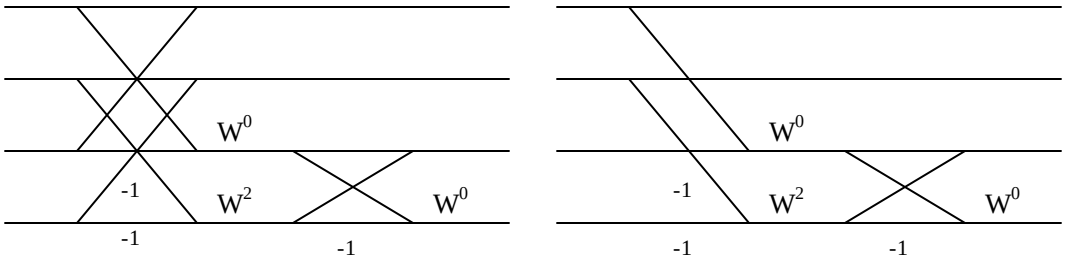
Figure 13: Myrinet Vs Ethernet : 16384 point Radix-2 FFT using *Communicate twice* algorithm

This paper establishes that FFT algorithms can be parallelized and we can also reduce the execution time unlike [2]. The *'communicate twice'* algorithm reduces execution time considerably than the algorithms used in [2]. But the engineering compromise here is that it is not a good cluster citizen. It increases the load of the processor. So, the cumulative CPU cycles spent on the operation is higher than the other algorithms. With increasing processing power of the processors, this might turn out to be a bearable commodity. The static heuristic in table 1 can be used as a guide designing intelligent FFT routines that automatically choose the number of processors to use.

6. Future work

The *"communicate twice"* parallel FFT algorithms are implemented such that the processors execute the butterfly operation is performed on all the inputs to the appropriate division in every stage. But that is not necessary and instead of performing computations to get the full butterfly, we can compute just half the butterfly (look at the figure 14 below) and reduce the execution time and the CPU load by half in each stage. This can be tried and benchmarked as well.

The implementations that were presented in this paper were actually parallel implementations of the hugely popular and less mathematically demanding FFT algorithms. There also exist other parallel FFT algorithms that are based on matrices and Kronecker products as in [3]. It can be studied and parallel versions of the algorithm can be implemented and the performance can be studied. Please note that the algorithm that was described in section 3.1.3 uses a concept that is slightly similar to [3]; but it does not exploit the many parallel algorithms that are available for matrix operations. Also, [3] is a parallel FFT algorithm from ground up.



a. a full-butterfly b. a half-butterfly

Figure 14: Concept of full and half butterflies in action when the processor in consideration needs to compute only the bottom half of the four points in the final stage. 'b' shows that the above halfs are not computed in the previous stage.

We also saw that one of the main reasons that we cannot implement the approach in section 3.1.1 was because of the significant amount of and non-uniform communication involved in an MPI system. But this is a very good candidate to be implemented in a shared memory system.

7. References

1. John G. Proakis and Dimitris G.Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, Third Edition, Prentice Hall, pages: 448-475

2. Michael Balducci, Ajitha Choudary and Jonathan Hamaker, *Comparative Analysis of FFT algorithms in sequential and parallel form*, Mississippi State University

3. Herbert Karner and Christoph W. Ueberhuber, *Parallel FFT algorithms with reduced communication overhead*, Institute for Applied and Numerical Mathematics, Technical University of Vienna