# Binary Exchange - A Parallel FFT Implementation

Nikolaos Tatarakis
nta@kth.se

## Abstract

*This paper focuses on the implementation and evaluation of Fast Fourier Transform (FFT) algorithms, and in particular, it aims to examine how could they be parallelized and how do they scale for different amounts of input data. Firstly, we will examine the FFT algorithm, and provide a basic theoretical background as well as simple properties it has. Then, we will refer to variations of the FFT algorithm and how could it be implemented for parallel architectures; one basic algorithm will be examined in depth, namely, the Binary - Exchange. Experiments on small to very large scale are used to demonstrate the parallel performance against the serial.*

## 1. Introduction

The discrete-time Fourier transform (Discrete Fourier Transform - DFT) is an algorithm widely used in science. It plays an important role in many scientific and technological applications such as waveform analysis, solving linear partial differential equations, the calculation of convolution, digital signal processing, image processing and telecommunications. The DFT, is a linear transformation that depicts $N$ uniform samples of one cycle of a periodic signal on an equal number of points that represent the spectrum of the signal frequencies. Its computational complexity is $O(N^2)$ and therefore it is judged as rather unsatisfactory.

In 1965, Cooley and Tukey [1], created an algorithm for calculating an $N$-point DFT ($N$-point DFT) with complexity $O(Nlog_2N)$ . The algorithm they created, was a major breakthrough in the field of DFT algorithms and it is usually referred to as a Fast Fourier Transform (Fast Fourier Transform - FFT). Due to the widespread use of this algorithm, there have been many efforts to implement FFT algorithms on parallel architectures. There are many different forms of FFT algorithms. In this paper we will be dealing with a particular case of one-dimensional parallel FFT algorithm, namely, Binary Exchange [5] [2] [3].

### 1.1. The Discrete Time Fourier Transform

The DFT, is the most important discrete transform used to perform Fourier analysis in many practical applications as we mentioned earlier. It basically converts a finite sequence of equally-spaced samples of a function into an equivalent-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency. The sequence of $N$ complex numbers $x_0, x_1, \ldots, x_{N-1}$ is transformed into an $N$-periodic sequence of complex numbers $X_0, X_1, \ldots, X_{N-1}$ so that:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi ikn/N}, k = 0, 1, \ldots, \frac{N}{2} - 1 \qquad (1)$$

where $i$ is the imaginary unit. Note that we will use the following notation for the phase vector:

$$W_N = e^{-2\pi i/N} \qquad (2)$$

We can observe that for each value of $k$, the direct computation of $X(k)$ involves $N$ complex multiplications and $N-1$ complex additions. Consequently, to compute all $N$ values of the DFT requires $N^2$ complex multiplications and $N^{2-N}$ complex additions.
It is easy to prove that the following proprieties are valid for each $W_N$:

$$\text{Symmetry property: } W_N^{k+N/2} = -W_N^k \qquad (3)$$

$$\text{Periodicity property: } W_N^{k+N} = W_N^k \qquad (4)$$

So we can conclude that the direct computation of the DFT is basically inefficient because it does not exploit those facts. Some of the FFT algorithms, that are computationally efficient, are described in the next section and exploit these two basic properties.

### 1.2. Theory of the Fast Fourier Transform

To begin with, in our computations, we make the assumption that $N$ is a power of 2. The FFT is based on the fact that an $N - point$ DFT may be divided into two

$(N/2 - point)$ DFTs, as follows:

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n W_N^{kn} \\ &= \sum_{n=even}^{N-1} x_n W_N^{kn} + \sum_{n=odd}^{N-1} x_n W_N^{kn} \\ &= \sum_{n=0}^{N/2-1} x_{2n} W_N^{2kn} + \sum_{n=0}^{N/2-1} x_{2n+1} W_N^{(2n+1)k} \end{aligned} \quad (5)$$

However, $W_N^2 = W_{N/2}$, so, we can express the sum in the following way:

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} W_{N/2}^{kn} + W_N^k \sum_{n=0}^{N/2-1} x_{2n+1} W_{N/2}^{nk} \quad (6)$$

Then, we can write:

$$X_k = X_{even,k} + W_N^k X_{odd,k} \quad (7)$$

Where the $X_{even}$ and $X_{odd}$ are the $N/2$-DFTs of the sequences obtained by considering respectively only the even or only the odd components of the initial sequence: $x_0, x_1, \dots, x_{N-1}$. Now, we can observe that $X_{even}$ and $X_{odd}$ are periodic, with period $N/2$, so is true that:

$$\begin{aligned} X_{even,k+N/2} &= X_{even,k} \\ X_{odd,k+N/2} &= X_{odd,k} \\ W_N^{k+N/2} &= -W_N^k \end{aligned}$$

thus, we have that:

$$X_k = X_{\text{even},k} + W_N^k X_{\text{odd},k}, k = 0, 1, \dots, \frac{N}{2} - 1 \quad (8)$$

$$X_{k+N/2} = X_{\text{even},k} - W_N^k X_{\text{odd},k}, k = 0, 1, \dots, \frac{N}{2} - 1 \quad (9)$$

As we can easily see now, the computation of $X_k$ requires $2(N/2)^2 + N/2 = N^2/2 + N/2$ complex multiplications.

## 1.3. The Recursive FFT algorithm

If $N$ is a power of two, each of the above DFTs ($X_{even}$ and $X_{odd}$) can be divided in the same way into two smaller with the same size (half of the DFT from which it originated). This method is used to implement a recursive FFT algorithm. A pseudo code is given in Algorithm 1.

**Algorithm 1:** Recoursive_FFT

```
1  Recursive_FFT ( x, X, N, W )
2  if (N = 1) then
3      $X[0] = x[0]$;
4  else
5      Recursive_FFT($x_{\text{even}}, Q_{\text{even}}, N/2, W^2$);
6      Recursive_FFT($x_{\text{odd}}, T_{\text{odd}}, N/2, W^2$);
7      for i=0 : N-1 do
8          $X[i] =$
               $Q[i \bmod (N/2)] + W^i T[i \bmod (N/2)]$;
9      end
10 end
11 end Recursive_FFT
```

The size of the input that we perform the FFT, is reduced recursively at every step by a half. At each level of the recursion, the difference between the indices of the elements $(Q[i], T[i])$ involved in the calculation is reduced by the same factor. Since the size of the input in each recursion level is halved, the maximum number of recursion levels, for an initial input of size $N$, is $log_2 N$. The total number of arithmetical operations performed on each recursion level equals to $\Theta(N)$, so we can safely conclude that the complexity of the recursive algorithm is basically $\Theta(N log_2 N)$.

## 1.4. The Iterative FFT algorithm

The serial FFT algorithm can also be written in a non-recursive form. This form, facilitates the study of the parallel implementation of the FFT and that's why we present it before describing the actual parallel algorithm. The implementation for this algorithm can be basically derived by the recursive one, where each level of recursion is depicted as one iteration of a for loop. The pseudo code for this algorithm is given in Algorithm 2. Note that $(b_{r-1} b_{r-2} ... b_0)$ is the binary representation of $i$.

This algorithm has two main loops. The external one runs $N log_2 N$ times for an N-point FFT, and the inner loop $N$ times for each iteration of the outer loop. So it is obvious that the complexity of this algorithm is also $O(N log_2 N)$. It is important to pay attention to the line before the end of the inner loop (line 13): $R[i]$ is updated by using $S[j]$ and $S[k]$. It is important to see how the indexes $j, k$ are derived from $i$. Assuming that $N = 2^r$, where $0 \leq i < N$, the binary representation of $i$ is composed of $r$ bits.

Then, during the $m$ iteration of the outer loop $(0 \leq m \leq r)$, we get the index $j$ by setting the $m$-th digit of $i$ (that is $b_m$) to zero. Similarly we get the index $k$ by setting $b_m = 1$. I.e. the binary representations of $j, k$ differ only at bit $b_m$ $(k - j = 2^m)$. Below, we provide a dataflow diagram, Figure 1, where is it shown how the input elements for a 16-point FFT are being combined. Each node represents one data. Each node has two inputs, that is, the two edges

**Algorithm 2:** Iterative_FFT

```
Iter_FFT ( x, X, N, W )
$r = logN$;
for $i$=0 : N-1 do
    $R[i] = x[i]$;
end
for $m$ = $r$ - 1 : 0 do
    for $i$ = 0 : N-1 do
        $S[i] = R[i]$;
    end
    for $i$ = 0 : N-1 do
        $j := (b_{r-1}...b_{m+1}0b_{m-1}...b_0)$;
        $k := (b_{r-1}...b_{m+1}1b_{m-1}...b_0)$;
        $R[i] := S[j] + S[k]W^{b_{m-1}b_m...b_0 0...0}$
    end
end
for $i$: = 0 to N-1 do
    $X[i] = R[i]$;
end
end Iter_FFT
```

that go towards it. Each edge in the graph transfers the data which is located at the 'start' of the edge, multiplied by the factor $W_p$, that is located at the 'end' of the edge. Absence of $W_p$ factor indicates that $W_p = 1$. The two inputs to a node are being combined with the addition operation. Thus, for example for node $X_2(2)$ at $m = 2$ applies that $X_2(5) = X_1(1) + X_1(5)W_8$.

### 1.5. Calculation of the phase vector

In the iterative algorithm we saw that we must evaluate $W^{b_m b_{m-1}...b_0 0...0}$. The power of $W$ can be basically obtained by using the following method:

i We write the index $i$ in binary form with $r = log_2 N$ bits, $(b_{r-1}b_{r-2}...b_0)$.

ii We right slide $m$ bits by filling with zeros the new bits.

iii We Reverse the sequence of the bits.

Let's see for example, in Figure 1, node $X_3(8)$. Since $m = 1$, $r = 4$ and $i = 8$, we have $i = 1000_{<2>}$. Right sliding one position and after reversing the result we obtain $0010_{<2>}$ so we have exactly what we need: $W^2$.

### 1.6. Reconstruction of output

Something that we silently hid until this point, is, the fact that the output of FFT is not arranged properly. In short, the output has not the form

$$X = X_0, X_1, \cdots, X_{N-1}$$

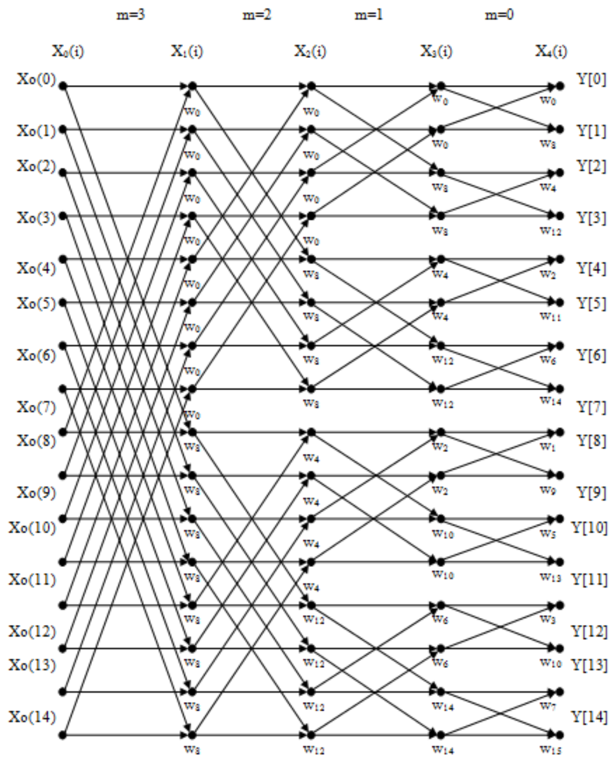

Figure 1. Scheme for a 16 FFT points

but is arranged according to the bit-inversion process (bit reversal process).To get the output with the correct order it is enough to perform the method of the bit reversal. So, if the data output is called $Y[i]$, we simply write $i$ in its binary form and then we invert the bits of its binary representation. Finally, we proceed by exchanging $Y[i]$ with $Y[k]$, where, $k$ is the number of the binary inverted $i$.

## 2. A FFT algorithm for parallel architectures

In this part, we will dive into the algorithm for calculating the FFT on parallel architectures, namely the Binary Exchange.

### 2.1. The Binary-Exchange Algorithm

We will examine the distribution of the data, how this data is assigned to the processes as well as the communication scheme of the algorithm. To begin with, we will consider the case where each process is occupied with the processing of one element and then we will generalize it.

### 2.2. One element per process

This case can be seen in Figure 2. As shown, each process $P_i(0 \leq i < n)$ initially stores the $x[i]$, and finally produces the $X[i]$. In each iteration of the outer loop of the iterative algorithm, process $P_i$ updates the value of $R[i]$ (as seen in the line of the algorithm just before the end of the inner loop). To carry out the updates, the process $P_i$ (with

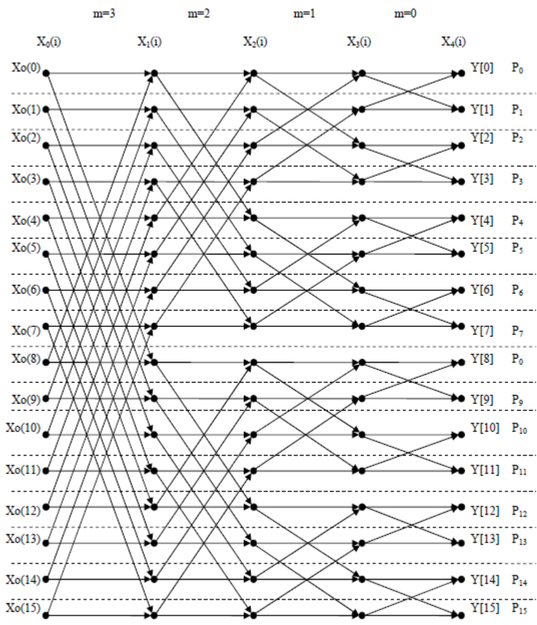

Figure 2. Scheme for the 16-point FFT calculated on 16 processes.

the element $x[i]$) needs one element of $S$, let $S[k]$, where $k$ differs from $i$ by a single bit. In the first iteration of the outer loop, $i$ and $k$ differs only in the most significant bit. So, for example, the processes $P_0 \cdots P_7$ communicate with the processes $P_8 \cdots P_{15}$ respectively. Similarly, in the next cycle, the processes which communicate with each other differs in the second most significant bit and so on and so forth. It is interesting to notice, that, since each process communicates with other processes that their addresses differ by only one bit, then, hypercube topology seems to be an ideal approach to implement the Binary Exchange algorithm. In hypercube topology, each node is only connected with other nodes that their addresses differ by only one bit to this node's address. In each of the $log_2 N$ iterations of the algorithm, each process $P_i$ performs a complex addition and a complex multiplication, and it exchanges a complex number with another process. That means, that, there is a balanced load per cycle. Therefore, we need $\Theta(log_2 N)$ time to execute the algorithm, using hypercube topology.

### 2.3. Multiple data per process

Consider now that we have $N$ data uniformly divided into $p$ processes, where $N > p$. Let us also assume that both the $N$ and $p$ are powers of 2, i.e. $N = 2^r$ and $p = 2^d$. We divide the input into blocks of $N/p$ consecutive data and then assign each block to a process. An interesting feature of this partitioning is that if $(b_{r-1} b_{r-2} \cdots b_0)$ is the binary representation of $i$, then $R[i]$, $S[i]$ assigned to a process with address $(b_{r-1} \cdots b_{r-d})$. I.e. the $d$ most significant bits of

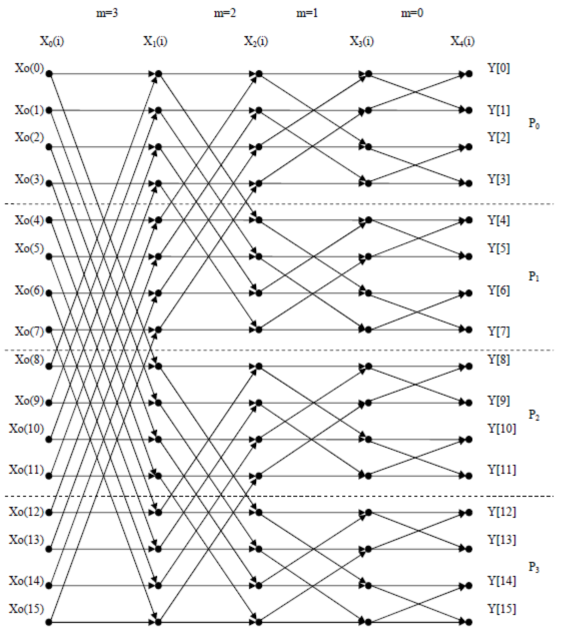

Figure 3. Scheme for the 16-point FFT calculated on 4 processes. (note the dotted lines that divides the processes into groups of four)

the index of each input element are the binary representation of the process address to which this element belongs to. This property plays an important role in the calculation of the communication pattern of the algorithm. In figure 3, we show that elements that differ in $d = 2$ most significant bits are assigned to different processes. However, all of the elements whose index is the same in the two most significant bits, they assigned to the same process. As we have already mentioned, in a N-point FFT, the outer loop is being executed $r = log_2 N$ times. During the loop, the data which differ in the $m$ most significant bit, are combined to carry out the calculation. Elements which are combined in the first $d$ iterations, belong to different processes, while the pairs of elements in the last $r - d$ iterations, they belong to the same process. So there is no communication in the last $r - d$ iterations. Furthermore, all elements needed by a process at some iteration $(i)$ (from the first $d$ iterations), they belong to only one different process; that is the process that its label differ at the $i$ most significant bit. Finally, in each communication step, $N/p$ data are being transferred.

## 3. Implementation

This section will deal with how to implement the FFT algorithm which we analyzed in the previous chapter. In order to avoid the implementation details that are not necessary for understanding the general philosophy of the algorithm, the description will be given using pseudo code.

## 3.1. The Parallel Algorithm (Binary Exchange)

The algorithm that we give below, is basically the part of the code that is being executed by each process of the parallel program. Using the same notation of the previous sections, $x$ is the initial series and $X$ is his DFT. The algorithm is parametrized according to the address of the process that is running it. By address, we mean the rank of the process, such as $0 \leqslant rank < p$ . Note that the $computeW$ function is calculating the power of the $W$ similarly to what we explained before and the $splitpoint$ permits to compute the butterfly communication between the right processes. After computing the Algorithm 3, we proceed by calling the $MPI\_Gather$ for reassembling all our result in one unique vector.

**Algorithm 3:** Parallel FFT

```
ParallelFFT( x, X, N, rank, size)
$r = logN$;
dataChunk $= N/size$;
$start = rank *$ dataChunk;
for $m$=0 : $r$ do
    for $i$ = 0 : dataChunk do
        Sk[i]=S[i]=x[i];
    end
    bit=left_shift(1) of r-m-1 positions;
    notbit=bitwise_complement(bit);
    splitpoint=size/(left_shift(1) of m+1 positions);
    if splitpoint $> 0$ then
        MPI_sendrec(rank $\pm$ splitpoint);
    end
    else
        $Sk[i] = S[i]$ for all i
    end
    i=start;
    for $l$=0:dataChunk do
        $j =$ (i & nobit)% dataChunk;
        $k =$ (i | bit)% dataChunk;
        $W = computeW(i, r, m)$;
        $R[l] := S[j] + S[k] * W$;
        i++;
    end
end
for $i$=0:dataChunk do
    $X[i] = R[i]$;
end
end ParallelFFT
```

## 3.2. Analysis of the performance

As we have already analyzed, elements combined during the first $d$ iterations belong to different processes, and pairs of elements combined during the last $(r - d)$ iterations belong to the same processes. Hence, this parallel FFT algorithm requires inter-process interaction only during the first $d = logp$ of the $logN$ iterations. There is no interaction during the last $(r - d)$ iterations. Furthermore, in each iteration from the first $d$, all the elements that a process requires come from exactly one other process. Each interaction operation exchanges $N/p$ words of data. We will use the following notation:
$t_s$ the startup time for the data transfer;
$t_w$ the per-word transfer time;
$t_c$ is the time of a complex multiplication and addition;

Therefore, the time spent in communication in the entire algorithm is $t_s log(p) + t_w(N/p)log(p)$. A process updates $N/p$ elements during each of the $logN$ iterations. If a complex multiplication and addition pair takes time $t_c$, then the parallel run time of the binary-exchange algorithm is:

$$T_p = t_c \frac{N}{p} log_2 N + t_s log_2 p + t_w \frac{N}{p}. \tag{10}$$

The speed up is:

$$\begin{aligned} S &= \frac{t_c N log_2 N}{T_p} \\ &= \frac{p N log_2 N}{N log N + (t_s/t_c) p log_2 p + (t_w/t_c) N log_2 p}. \end{aligned} \tag{11}$$

So the efficiency is:

$$E = \frac{1}{1 + \dfrac{t_s p log_2 p}{t_c N log_2 N} + \dfrac{t_w log_2 p}{t_c log_2 N}}. \tag{12}$$

Note that if we want the output distributed in the same way as the input, the computational time increases according to the method used to reverse the bits.

## 4. Experimental Results

The experimental procedure we followed in order to evaluate the algorithm is as follows; Firstly, we implemented the the recursive serial algorithm as described in Algorithm 1 section with complexity $\Theta(N log_2 N)$, and, compared it against the parallel algorithm for the following input sizes: $2^{10}, 2^{14}, 2^{19}, 2^{20}, 2^{22}$ and $2^{24}$. The comparison held against 2, 4, 8, 16, 32, and 64 processes. For the parallel implementation, the MPI function for code profiling used, that is, `MPI_Wtime()` and time-averages over 10 repetitions of each experiment used to get accurate data with small standard deviation.

In Figure 4 and 5, we can see the Speed Up and the Efficiency ($\eta = S(P)/P$) plots respectively. We notice, that, for a small problem size such as $2^{10}$ we don't obtain much speed up after using more than eight processes; In fact, when the number of processes is increased dramatically we notice that performance deteriorates compared to the best serial algorithm as it has reached a saturation point after just 8 processes.

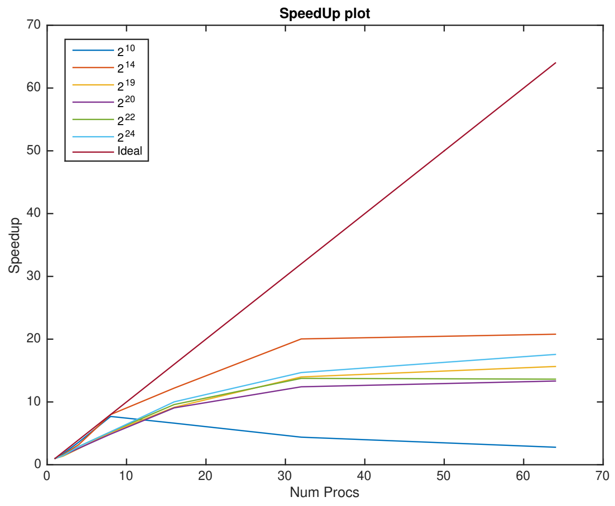

Figure 4. Speed-Up for different $P$ size and problem size.

We found out that the best speed up is obtained for a medium problem, of size $2^{14}$ followed by problem of very large size, $2^{24}$. It is also interesting to see that for problem sizes $2^{10}$ & $2^{14}$ the speed up is nearly linear for up to eight processes and we can see in Figure 5 that their efficiency tends to 1 for this number of processes. The raw data for the full computations times for all experiments can be seen in Table 1, in addition, we provide in Table 2 the communication times for the case with $2^{24}$ data input size.

To show the correctness of the algorithm, we evaluated the $sinc()$ function, $sinc(x) = \frac{\sin(\pi x)}{\pi x}$, in a range of $N = 128$ points, used the parallel algorithm to perform the FFT of the generated input sequence and used Matlab to interpret and visualize the results. As we can see in Figure 6, a Fourier transformation of the $sinc()$ function in time domain corresponds to a rectangular pulse in the frequency domain, which is the expected outcome [4].

## 5. Conclusion

In this paper, we demonstrated a parallel implementation of the well known FFT algorithm, namely, Binary Exchange. Experimental results on a wide range of input sizes showed that it is possible to achieve a rather considerable Speed Up (sometimes it is even approaching linearity) compared to the serial version of the algorithm. This fact,

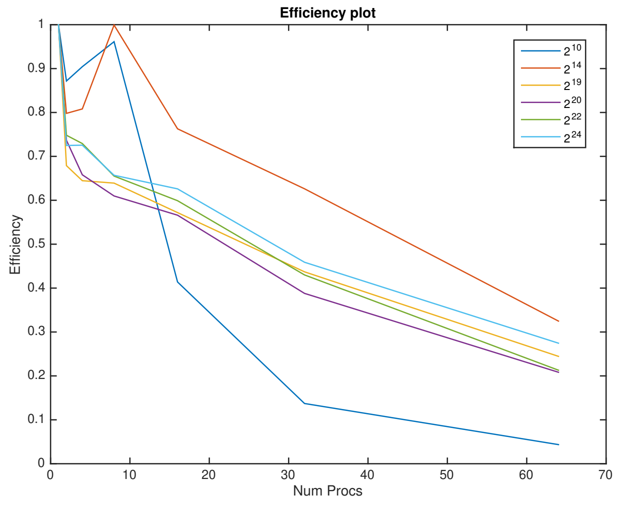

Figure 5. Efficiency for different $P$ size and problem size.

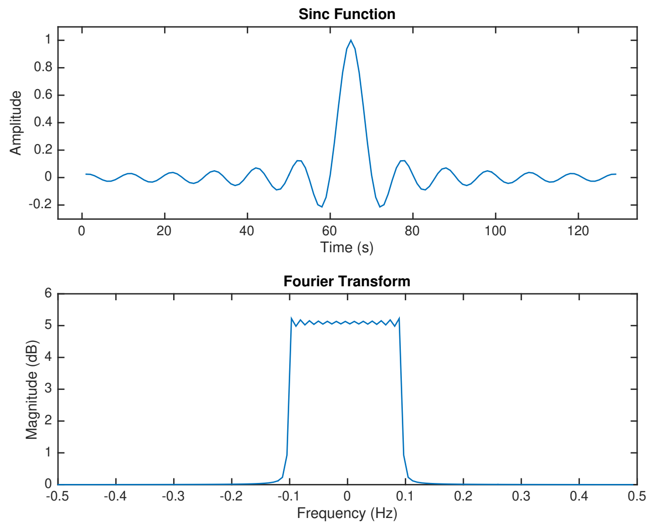

Figure 6. FFT of a $sinc$ corresponds to a rectangular pulse.

makes Binary Exchange algorithm a very attractive choice for computing the Fourier Transform of large sequences, especially when the computational resources are available.

| | Serial | 2 Processes | 4 Processes | 8 Processes | 16 Processes | 32 Processes | 64 Processes |
|---|---|---|---|---|---|---|---|
| $2^{10}$ | 0.003722 | 0.002135 | 0.001029 | 0.000484 | 0.000562 | 0.000848 | 0.001336 |
| $2^{14}$ | 0.057731 | 0.036172 | 0.017861 | 0.007223 | 0.004730 | 0.002881 | 0.002777 |
| $2^{19}$ | 1.248665 | 0.919630 | 0.484247 | 0.244212 | 0.136436 | 0.089280 | 0.079791 |
| $2^{20}$ | 2.438644 | 1.655293 | 0.926443 | 0.499844 | 0.269259 | 0.196435 | 0.182918 |
| $2^{22}$ | 10.144808 | 6.779875 | 3.477813 | 1.935549 | 1.058389 | 0.737257 | 0.744276 |
| $2^{24}$ | 42.573740 | 29.369165 | 14.669548 | 8.100109 | 4.250414 | 2.899349 | 2.423137 |

Table 1. Raw Data. All times are in seconds (s).

| | 2 Processes | 4 Processes | 8 Processes | 16 Processes | 32 Processes | 64 Processes |
|---|---|---|---|---|---|---|
| $2^{24}$ | 0.040560 | 0.019947 | 0.008986 | 0.008913 | 0.005262 | 0.009755 |

Table 2. Raw Data, Communication time. All times are in seconds (s).

## References

[1] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[2] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, Inc., New York, NY, USA, 1987.

[3] P. N. Swarztrauber. Multiprocessor ffts. *Parallel computing*, 5(1-2):197–210, 1987.

[4] E. W. Weisstein. Rectangle function. *Wolfram Research, Inc.*, 2002.

[5] B. Wilkinson and C. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. An Alan R. Apt book. Pearson/Prentice Hall, 2005.