Efficient Bit Reversal Algorithms in Parallel Computers

Rami A. Al Na’mneh*
Jordan University of Science and Technology, Irbid, 22110, JORDAN

Khalid A. Darabkh† and Iyad F. Jafar†
University of Jordan, Amman, 11942, JORDAN

Abstract

Fast Fourier Transform is widely used in many areas of engineering and mathematics such as digital signal and image processing systems, and polynomial multiplication. Furthermore, it has an important role in performing parallel simulations such as plasma simulation, weather forecasting, and dynamic fluids. Bit-reversal routine is considered to be an essential part of FFT and that is because of high possibility of degrading the overall execution time of FFT application if it is not perfectly designed. In this paper, we propose a new formulation for Numerical Recipes bit-reversal sequential algorithm. The formulation is based on independent steps; hence it can be efficiently implemented on parallel computers. Two algorithms that utilized the new formulation are proposed, namely, the All-to-all and Swapping algorithms. Experimental results of the proposed algorithms show that a reasonable speedup can be achieved, especially for large data sizes, when compared to the sequential implementation of the bit-reversal. Additionally, the results show that the Swapping algorithm performs better than the All-to-all algorithm.

Key Words: Bit-reversal, fast fourier transforms, parallel computers, performance analysis and time complexity, hierarchy transpose, speedup.

1 Introduction

The Fast Fourier Transform (FFT) is known as a class of algorithms that can effectively compute the Discrete Fourier Transform (DFT) of a time series of discrete data samples through carrying out the calculation of its coefficients iteratively, which consequently leads to remarking savings in its computational time [3, 7]. The complexity of processing the Fourier transform of an N-tuple prior to FFT algorithm is $N^2$ but this complexity is reduced to $N \log_2(N)$ when incorporating FFT [7]. However, FFT was originally developed by J. W. Cooley and J.W. Tukey in 1965 [8] and is considered the most important numerical method and computational tool in the last century that can efficiently simplify, through available digital computers, the signal analysis such as filter simulation and power spectrum analysis [3, 7]. Moreover, it facilitates the complicated analysis in many interesting fields such as engineering, applied mathematics, astronomy, cryptography, physics, and computational finance [2]. As further evidence of its importance, it is used in several benchmarks for parallel computers such as NASA parallel benchmarks and High-performance computing (HPC) challenge [18]. The professional abilities of solving PDEs in multiplying large polynomials, digital signal processing, and computational fluid dynamics are considered as a part of its wide applications [18].

There are new FFT algorithms which are proposed after Cooley and Tukey FFT algorithm such as split-radix algorithms [11], mixed-radix algorithms [28], prime factor algorithms [6], and multidimensional FFT algorithms [19] which mainly come up with efficiency improvements. However, there are many papers that proposed efficient implementations of the above mentioned FFT algorithms on single computers. These algorithms aim to reduce the number of multiplications and additions to evaluate FFT [6, 8, 11, 14-15, 19, 37]. Other algorithms introduce an optimization on the memory through reducing the number of loads and stores which consequently decrease the total time of FFT [1, 16, 21, 23, 34]. The following subsection briefs ideas proposed to parallelize FFT algorithms. In subsection B, details about bit-reversal, FFT, as well as their relation to each other are provided. On the other hand, parallel algorithms proposed for implementing bit-reversal are discussed in subsection C. In the last subsection of the introduction, the proposed approach in this paper and its contributions are summarized.

1.1 Parallel FFT Algorithms: Overview

In the past years much work was done to reduce and minimize the FFT computational time. In [33], a parallel one-dimensional FFT algorithm is proposed to run on clusters of vector symmetric multiprocessors (SMP) nodes, namely Hitachi SR 8000 which consists of 4 to 512 nodes where each node has pseudo-vector processing (PVP) and contains up to eight RISC microprocessors. This is mainly done through using the cyclic distribution that makes the communication between processors occur only once besides that the input and
output data are both in natural order. Moreover, an expansion to the innermost loop length of the well-known four-step FFT is made. In other words, the four-step FFT algorithm is completely altered into a five-step FFT algorithm. It is reported that when using Hitachi SR 8000, the five-step FFT-based parallel FFT algorithm is more advantageous than the four-step FFT-based parallel FFT algorithm and a performance of 61 GFlops is achieved. In [32] the optimum performance of FFT is compared on two-, three-, and four-dimensional torus and hypercube multicomputers. To utilize the full bandwidth of the interconnection network on every cycle, the known multiport communication algorithms are adopted for performance comparison. It is observed that the communication time over hypercube architecture is almost negligible compared to computational time while the opposite is completely true for torus architectures. As the dimension of the interconnections is gradually increased, torus performance is obtained comparable to hypercube. In [31] a significant contribution to the final computational time is made through developing parallel FFT algorithms to be efficiently run on a hypercube which in turn minimizes the inter-processor communication time. Furthermore, parallel FFT algorithms are also developed to be run on a vector multiprocessor system with shared memory (i.e., Alliant FX/8 multiprocessor). To fully utilize the potential of vector processing, these algorithms use long arrays where their elements are stored consecutively. In [20] FFT algorithms are parallelized and consequently run on multicomputers connected through mesh and hypercube interconnect networks. A complete scalability analysis is provided for both interconnect networks utilizing the known isoefficiency parallel performance metric, which refers to the rate at which the problem size must increase with the number of processors to maintain a constant efficiency. It is mentioned that when considering hypercube architecture, a speedup is obtained where it is linearly increased with the number of processors under a small increase in the problem size and this is limited by the bandwidth of communication channels and CPU speed.

When considering mesh architecture, the described scalability analysis shows that the parallel FFT algorithms can get benefit of large-scale mesh architectures as long as the bandwidth of communication channels increases as a function of the number of processors. Lastly, it is proven that implementing parallel FFT algorithms on a hypercube is more cost-effective than a mesh.

1.2 Bit Reversal and FFT: Overview

Many versions of the fast Fourier transform require a reordering of either the input or the output data that corresponds to reversing the order of the bits in the array index. Bit-reversal algorithms are used not only in FFT, but also in Fast Hartley Transform (FHT). Bit-reversal operation of the FFT accounts for a significant amount of the workload of the entire FFT computation. For example, in a length of 1024 split radix FFT, the bit-reversal requires about 20 percent of the time needed to perform the FFT itself [10]. If the bit-reversal is not performed properly, it can take a tremendous percent of the total time needed to perform the FFT. In other words, it is not easy to achieve a better performance when implementing bit-reversals either on uniprocessors or multiprocessors since the performance of bit-reversals is dependent mainly on how caches and memory hierarchies are used [38].

There are a lot of research articles that consider various memory hierarchies to report their effects on the performance of bit-reversal [4, 22, 30]. One more important usage to mention about bit-reversal algorithms besides its importance to FFT is that they are used as benchmark programs for evaluating the memory hierarchy of different digital machines. Additionally, they are widely used in image transposition and generalized sorting of multidimensional arrays.

There are many published bit-reversal permutation algorithms that run on single computers which are mainly targeted to solve the problem of data scrambling, namely shuffle, which is related to the first or last stage of FFT algorithms. These bit-reversal permutation algorithms are evaluated in many aspects, out of which are the number of program loops, memory size, and number of operations being as the most important factors [25]. Additionally, they can be categorized into two different classes. The first class of these proposed algorithms [12-13, 35] requires the use of auxiliary table (seed table) of size \( \sqrt{L/2} \) if \( L = 2^{2q+1} \) or \( \sqrt{L} \) if \( L = 2^q \), where \( L \) is the transform length and \( q \) is the length of the binary expansion of an index \( i \). The second class does not use seed table but is related to memory optimizations [9, 17, 24, 27, 36]. It is proven that the algorithms of first class run much faster than those found in the second class except for some cases described in [9].

1.3 Parallel Bit Reversal Permutations: Overview

All of previously mentioned parallel FFT algorithms use the sequential code of bit-reversal algorithm, and do not try to parallelize it. In fact, it is a very challenging problem at the side of parallel processing. This is because implementing bit-reversal algorithms on multi-computers is not an easy task since as mentioned earlier that the implementation of bit-reversals should be cache effective. However, there are few papers which implement bit-reversal on parallel computers. In [38], the cache-optimal techniques of blocking, buffering, and padding are examined first to implement bit-reversal on uniprocessors. It is reported that there are integrations between these techniques so that the cache locality is effectively exploited which are blocking with padding, blocking for translation-lookaside buffer (TLB), and padding for TLB. After that, these integrations are tested on two commercial SMP multiprocessors which are Sun E-450 and HP 9000 V2200. However, no speedup is reported. Additionally, there is no performance analysis found to indicate the need for testing on these commercial multiprocessors. But, it is reported that implementing these integrations on crossbar interconnect network is much more efficient than the well-known bus topology. Furthermore, it is very important to
mention that the data access contention to the shared memory is a potential bottleneck on SMPs. No further exploration about these techniques is discussed here since they fall out of the interest of this paper. However, there is a relevant work to ours where the well-known Elster’s algorithm, which is used to produce the vector representation of bit-reversal, is parallelized utilizing the idea of splitting the total array length (one dimension) among available processors without any kind of optimization. For details about this work please see [5]. The execution environment of this work is described as follows: Cilk 5.2 multithreaded computing, extension of C, is considered to be the software environment. There is a lack of details about the hardware environment except that the code is run on a four-processor machine. However, the speedup of 2.4 and 2.6 are reported under the conditions where the data sizes are 2048x2048 and 4096x2048, respectively.

1.4 Our Approach

In this paper, a new formulation of the Numerical Recipes bit-reversal sequential algorithm is proposed such that it can be efficiently parallelized. Actually, Numerical Recipes and Elster are very popular sequentially algorithms for evaluating the bit-reversal. However, the new formulation computes the reversal process through ordering the data set into a two dimensional array and relies on performing bit-reversal on the columns and the rows separately. This has the effect of reducing the time complexity. Based on this formulation, two novel parallel algorithms are introduced, namely, the All-to-all and Swapping algorithms. In the All-to-all algorithm, the new bit-reversal formulation in addition to hierarchy array transposition are exploited for parallelization. The array transposition is performed through all-to-all communication between the processors, hence the name All-to-all. On the other hand, the Swapping algorithm parallelizes the bit-reversal process through swapping the rows of the corresponding bit-reversed indices between the processors. The parallel programming environment used is message passing interface (MPI). The proposed algorithms are tested on 14 distributed computers connected together using Myrinet switch and for different data sizes and number of processors. The results are very promising in terms of the achieved speedup for both algorithms when compared to the previous work [5] when considering the same number of used processors and data sizes. Moreover, the effect of data size and number of processors on speedup for both algorithms is investigated. Additionally, the experimental results are explained and justified through a detailed complexity analysis of the two algorithms.

The rest of the paper is organized as follows. In Section 2, the new formulation of the bit-reversal process is presented. The proposed algorithms that utilize the new formulation are explained in Section 3. Section 4 includes the performance analysis for the All-to-all and for the Swapping algorithms. Experimental results and discussion are presented in Section 5. Finally, the paper is concluded in Section 6.

2 New Formulation of Bit Reversal

Generally, bit-reversal is a permutation operation in which elements in a data set are rearranged by bit-level processing of their indices. For a data set with $M$ elements, $n$ bits ($\log_2 M$) are required to represent the indices of the elements. In this case, bit-reversal is essentially the permutation in which the index of the $i^{th}$ element written in binary with digits $b_{n-1}b_{n-2}…b_1b_0$ is transferred to the index with reversed digits $b_0 b_1 b_2…b_{n-1}b_{n-2}…b_{n-1}$. Such permutation can be achieved in different ways [26, 29]. However, the new bit-reversal formulation presented here in this paper is proposed to facilitate parallelization. This can be better explained through the following example which considers applying the bit-reversal procedure on a 16-element data set, i.e., 4 bits are required to represent the indices of the elements which range from 0 to 15. For each binary value $b_0b_1b_2b_3$ that represents the element index, it is divided into two groups of bits; $b_0b_1$ and $b_2b_3$. The first group includes the upper $n/2$ bits while the lower group has the lower $n/2$ bits. Next, bit-reversal is applied to each group of bits to obtain $b_1b_0$ and $b_3b_2$ and then groups are swapped and combined to form the bit-reversed index $b_1b_0b_3b_2$. For example, if the index to be reversed has the value of 5 (0101)2, then upper group is 01 and the lower group is 01. Reversing the bits in each group results in 10 and 10, respectively; this in turn gives the reversed index value of (1010)2 after swapping the two groups. This approach in computing the bit-reversal can be realized through array operations. In this case, the data set is arranged column-wise in a two-dimensional array as shown Figure 1a. In this arrangement, one can think of the column index to be the upper $n/2$ bits of the index value while the lower $n/2$ bits are the row index. Thus, each index of the array is now the concatenation of the row index and the column index. Next, bit-reversal is applied to the columns as shown in Figure 1b which results in swapping the columns of the array, except for the first and last columns since their bit-reversed indices are the same as the original. Afterwards, the array is transposed and bit-reversal is applied to the columns of the transposed array as shown in Figure 1c and d. The algorithms proposed in this paper to parallelize the bit-reversal exploits this approach in order to distribute the workload on multiple processors as explained in the following section.

3 Proposed Algorithms

This section explains how the new formulation of bit-reversal is exploited for parallelization in two different algorithms, namely, the All-to-all and Swapping algorithms. The first algorithm is based on all-to-all communication between processors while the second one relies on swapping the data rows between processors.

3.1 The All-to-All Algorithm

In order to explain the parallelization of bit-reversal using the All-to-all algorithm, an example of 16-element data set and
Figure 1: Illustration of bit-reversal formulated using two-dimensional array

(a) Original data arranged in 2D array
(b) Data after bit-reversal of the columns
(c) Data after transposition
(d) Data after bit-reversal of the columns

1) The data set is rearranged into a two-dimensional array of size $\sqrt{n} \times \sqrt{n}$, where $n$ is the number of elements. In this example, a 4x4 array is used and the elements are arranged column-wise. Actually, the same result is obtained if data is arranged row-wise; however, the rows will be exchanged with the columns.

2) The master processor divides the new array equally between all other processors. Thus, each processor gets $\left\lceil \frac{\sqrt{n}}{P} \right\rceil$ rows of the array or $\left\lceil \frac{\sqrt{n}}{P} \right\rceil \times \sqrt{n}$ elements, where $P$ is the number of processors. In this example, the two processors, $P_0$ and $P_1$, get two rows of the original data set.

3) The data subset assigned to each processor undergoes bit-reversal operation for its columns such that the indices 00, 01, 10, and 11 become 00, 10, 01, and 11, respectively. Actually, this is equivalent to preserving the first and last columns while swapping the remaining columns.

4) The data subsets in each processor are transposed using all-to-all communication. Actually, this is the major step in this algorithm in which array transposition for the blocks of data subsets are exchanged between processors. The transpose operation is done in all-to-all communication through hierarchy transpose, which is explained as follows. The data subset assigned to each processor is partitioned into $P$ blocks. The notation $Block(i,j)$ is used to refer to the $j^{th}$ block in the $i^{th}$ processor. Next, the $j^{th}$ block in the $i^{th}$ processor is transposed locally. However, for the remaining $P-1$ blocks, the $j^{th}$ block is sent row-wise to the $j^{th}$ processor which arranges the received block column-wise in its buffer. This operation is illustrated in Figure 2 and is detailed in general form in Figure 3. As a matter of fact, transposing the array in all-to-all communication can be overlapped with computation in which the data of size $\left\lceil \frac{\sqrt{n}}{P} \right\rceil \times \left\lceil \frac{\sqrt{n}}{P} \right\rceil$ is packed as illustrated in Procedure 1 which lists the pseudo-code for transposing the array through overlapping.

5) The transposed data subsets for each processor undergo bit-reversal operation for the columns as explained in Step 3.

3.2 The Swapping Algorithm

Adopting the example used to explain the bit-reversal by the All-to-all algorithm, the first three steps of the Swapping algorithm are essentially the same as those of the All-to-all algorithm where the data is rearranged into a two-dimensional array and divided equally between the processors, then bit-reversal is applied to the columns. The main difference lies in the fourth step where instead of utilizing the hierarchy transposition of the blocks between the processors, the Swapping algorithm proceeds by swapping rows between processors. This swapping is illustrated in Figure 4 and is explained as follows. The rows’ indices of the original two-dimensional data are first bit-reversed using a lookup table. Then, the reversed index of each row in each processor is used to determine the processor number with which swapping is to be performed in addition to the index of the row in that processor to be swapped with by

$$\text{Processor Number} = \left\lfloor \frac{\text{Row Bit_reversed Index}}{\text{Number of Rows}} \right\rfloor \quad (1)$$

and

$$\text{Row Index} = \text{MOD} (\text{Row Bit_reversed Index}/\text{Number of Rows}) \quad (2)$$
where the number of rows is $\sqrt{N}/P$ per processor. For the example under consideration, the original rows’ indices are 0 and 1 for $P_0$, and 2 and 3 for $P_1$. After bit-reversal of the rows’ indices, they become 0 and 2 for $P_0$ and 1 and 3 for $P_1$. Actually, bit-reversal in this step is performed using a lookup table. According to equations (1) and (2), row 0 in $P_0$ should be swapped with row 0 in $P_0$. In other words, it is not swapped. On the other hand, row 1 in $P_0$ is swapped with row 0 in processor $P_1$. The same applies to the rows in processor $P_1$. It is important to mention here that bit-reversal is applied on the indices only and no rearrangement of the elements is performed. The result of this reversal is used to determine the destination processor and the row index to swap with. The general algorithm that performs this swapping operation between $P$ processors is illustrated in Procedure 2.

4 Performance Analysis

As mentioned in the first section, the bit-reversal step in the computation of the FFT comprises a significant portion of the computational time. In the proposed work, experimentation revealed that the percentage of the sequential computational time of bit-reversal is between 15–30 percent of the total time of FFT. In fact, these values can be related to two causes: data size and how caches and memory hierarchies are used in the implementation. In other words, when large data sizes are used, cache conflict misses increase which in turn increase the total sequential time of bit-reversal and consequently degrade its performance. Accordingly, it is imperative to investigate the performance of the proposed algorithms analytically in terms of the computation and communication overhead times. The total time spent to execute a certain task $T_p$ on parallel computers is basically the sum of the computation and the communication overhead times, $T_{\text{computation}}$ and $T_{\text{communication}}$, respectively. The computational component is a function of the computational rate of the individual processors and the workload while the penalty associated with the parallelization process is represented by the communication overhead term. For the case of the All-to-all algorithm and with a data set of $n = \sqrt{n_x} \sqrt{n_y}$ elements, the processing time of the algorithm is obtained as follows. The algorithm basically consists of two bit-reversal operations, steps 3 and 5, and one transpose operation that is done through all-to-all communication. For
the bit-reversal operations, the complexity of the best algorithms to compute it is $O(n)$ [26, 29] which is for Elster’s bit-reversal sequential algorithm but it is $O(n \log n)$ for Numerical Recipes bit-reversal sequential algorithm [26] that is employed in this paper. Hence, the total complexity for these two steps is $2\sqrt{n}$. Thus, the computational time is $2Tc\sqrt{n}$, where $Tc$ is the time per computational step. As for the transpose operation in the All-to-all algorithm, step 4, if the number of processors is $P$, then each processor will have

Procedure 1: Pseudo code for hierarchy transpose in the all-to-all algorithm

```
IN: P is the number of processors
begin
    j = block index;
    MPI_Init();  //Initialize MPI
    MPI_Comm_rank();  //Get the rank for each processor
    MPI_Comm_size();  //Get the number of processors
    Blocks = Divide_Data()  // Divide $i$th processor data
               into $P$ blocks each of size $\sqrt{\frac{N}{P}} \times \sqrt{\frac{N}{P}}$
    j = 0;
    While ($j < P$ && $j != i$) do
        MPI_Isend (blocks($j$) to $j$th processor);
               //Non-blocking send row-wise
        MPI_Irecv (blocks(rank) from $j$th processor);
               //Non-blocking receive column-wise
        Increment $j$;
    end while
    MPI_Waitall();  // Waits for all given communications to complete
    MPI_Finalize();  //Terminate MPI
end
```

the data subset in each processor is divided equally into $P$ blocks of size $\sqrt{\frac{n}{P}} \times \sqrt{\frac{n}{P}}$. Thus, each processor transposes one block with size $\sqrt{\frac{n}{P}} \times \sqrt{\frac{n}{P}}$ locally and sends one block with size $\frac{\sqrt{n}}{P} \times \frac{\sqrt{n}}{P}$ to each of the $P-1$ processors in the system. Generally, the communication overhead per processor is given by
Procedure 2: Pseudo code for swapping the row between processors in the swapping algorithm

**IN:** P is the number of processors

```
begin
    MPI_Init(); // Initialize MPI
    MPI_Comm_rank(); // Get the rank for each processor
    MPI_Comm_size(); // Get the number of processors
    i = processor rank;
    rows = the set of rows assigned to rank
    R = \sqrt{n}/P // Number of rows
    index = the set of bit-reversed indices of the rows;
    j = 0;
    loop (j=0 to R-1)
        processor = floor(index(j)/R); // Index of processor to receive the row
        destination = Mod(index(j)/R) // Index of the row to swap in the receiving processor
        if processor != i then
            MPI_send (rows(j) to processor); // Blocking send
            MPI_recv (rows(destination) from processor); // Blocking receive
        endif
        increment j;
    end loop
    MPI_Finalize(); // Terminate MPI
end
```

\[ h_{All-to-All} = T_s + mT_w \]  
(3)

where \( T_w \) is the effective communication time per 4-bytes supported by the inter-processor communication network, \( T_s \) is the start-up time associated with all basic communication operations, and \( m \) is the message size. Accordingly, for \( P-1 \) blocks, the total communication overhead in the All-to-all algorithm is

\[ h_{All-to-All}(n, P) = (T_s + T_w \frac{n}{P^2})(P - 1) \]  
(4)

Consequently, the total processing time of the All-to-all algorithm is the sum of the computation time and the communication overhead time

\[ T_P^{All-to-All} = (T_s + T_w \frac{n}{P^2})(P - 1) + 2T_c \sqrt{n} \]  
(5)

For the Swapping algorithm, basically it involves bit-reversal of the columns as a first step with computational complexity of \( T_c \sqrt{n} \). As for the second step, it is also a bit-reversal operation; however, it requires swapping the rows between processors based on the result of bit-reversal of the rows indices as explained in Section 2 (subsection B). With \( P \) processors in the system and data set with \( n = \sqrt{n} \times \sqrt{n} \) elements, each processor will have \( \frac{\sqrt{n}}{P} \times \sqrt{n} \) elements and it will swap \( \sqrt{n} \) elements in each swap. Therefore, the communication overhead associated with this step is given by

\[ h_{Swapping}(n, P) = (T_s + T_w \sqrt{n})w \]  
(6)

where \( w \) is the number of swaps performed. Therefore, the total processing time of the Swapping algorithm is given by

\[ T_P^{Swapping}(n, P) = (T_s + T_w \sqrt{n})w + T_c \sqrt{n} \]  
(7)

In order to compare the performance of the two algorithms, consider the difference between their processing times

\[ D = T_P^{All-to-All} - T_P^{Swapping} \]  
(8)

which is equivalent to

\[ D = (T_s + T_w \frac{n}{P^2})(P - 1) - (T_s + T_w \sqrt{n})w + T_c \sqrt{n} \]  
(9)

after substituting equations (5) and (7). Clearly, the Swapping algorithm performs better if \( D > 0 \). Investigating equation (9) shows that the factor that favors the Swapping algorithm over the All-to-all algorithm is basically the number of swaps \( w \) since all other parameters are the same in both algorithms. The upper bound of \( w \) below which the Swapping algorithm performs better than the All-to-all algorithm, i.e., \( D \geq 0 \), can be obtained by solving

\[ (T_s + T_w \frac{n}{P^2})(P - 1) + T_c \sqrt{n} \geq (T_s + T_w \sqrt{n})w \]  
(10)

If we assume \( P-1 \approx P \), then the upper bound of \( w \) that makes the Swapping algorithm perform better than the All-to-all algorithm is given by

\[ w \leq \frac{T_s P + T_w \frac{n}{P}}{T_s + T_w \sqrt{n}} \]  
(11)

which is \( O(\sqrt{n}) \) for large data sets.

5 Results and Discussion

The proposed algorithms for bit-reversal are implemented using MPI programming and tested on a 14-node cluster. The
cluster is based on a homogeneous set of Intel\textsuperscript{TM} Pentium\textsuperscript{®} 4 2.4 GHz with 512k cache and 256MB DDR-SDRAM. The software configuration operating system is Redhat 7.3, Kernel 2.4.18 and with gcc compiler. The cluster is connected through a Myrinet switch. The size of data sets used in the experiments varies from 512x512 to 8192x8192 elements in steps of power of 2. The number of processors used in the experiments is 2, 4, and 8. Performance evaluation included 10000 experiments and the processing time is measured using the MPI\_Wtime() function. The sequential implementation of bit-reversal is based on the Numerical Recipes algorithm [26].

The processing time to compute the bit-reversal for the given data sets using sequential processing, the All-to-all algorithm, and Swapping algorithm are given in Tables 1 and 2. Comparing the corresponding numbers in the two tables shows how the proposed algorithms have lower processing time when compared to the sequential processing. For the proposed algorithms, one-to-one comparison shows how the Swapping algorithm has lower processing time than the All-to-all algorithm for different data sizes and number of processors. For a better representation and to compare the performance of the All-to-all and Swapping algorithms, the speedup for different cases is computed by

\[
    \text{Speedup} = \frac{T_{\text{Sequential}}}{T_{\text{Parallel}}} \quad (12)
\]

where \(T_{\text{Sequential}}\) is the execution time using single processor \((P=1)\) while \(T_{\text{Parallel}}\) is the execution time using parallel processors \((P>1)\). Figures 5 through 7 show the speedup charts for the proposed algorithms for different data sizes and when the number of processors is 2, 4, and 8, respectively. It is apparent in these figures how the Swapping algorithm has higher speedup values than the All-to-all algorithm. However, the following observations can be made about these figures. First, in theory, the speedup achieved through parallelization should equal the number of processors used. Actually, this can be claimed to be true for the cases of large data sizes as shown in the figures where the speedup approaches the theoretical value as the data size increases. Nonetheless, for small data sizes, the speedup is lower than theoretical limit. This can be justified as follows. If it is assumed that \(T_{\text{Parallel}} = T_{\text{Sequential}}/P\) in equation (12), then as the data size increases \((n \rightarrow \infty)\), the computational time \(T_{\text{Sequential}}\) increases and the contribution of \(T_{\text{communication}}\) in the denominator of (12) becomes smaller. This in turn pushes the speedup towards the upper limit which is the number of processors \(P\). On the other hand, when the data size decreases \((n \rightarrow 0)\), the dominant factor that determines the speedup becomes \(T_{\text{communication}}\), i.e., the communication overhead outweighs the actual computation time.

Another observation regarding the speedup is related to the difference between speedup achieved by the proposed algorithms. In this case, note how the difference in speedup between the two algorithms decreases as the data size
increases. This can be explained in light of the discussion presented in Section 4. Specifically, consider equation (9) which relates the processing times of the All-to-all and Swapping algorithms, and the fact that the number of swaps $w$ is $O(\sqrt{n})$ for very large $n$, as given in equation (11). If $w$ is substituted in equation (9), then for very large $n$, the processing times for both algorithms are $O(n)$. Thus, the difference between the processing times approaches zero as the data size increases.

It is important to mention that a previous work is done about parallelizing the known Elster’s bit-reversal sequential algorithm on a four-processor machine using Cilk 5.2 multithreaded computing C extension [5]. As a comparison with the proposed algorithms when considering the same number of processors, the speedup values of 2.4 and 2.6 are obtained for 2048x2048 and 4096x2048 data sizes, respectively [5] while in the All-to-all algorithm, the obtained values of speedup are 3.367 and 3.54 for 2048x2048 and
4096x4096 data sizes, respectively. Considering the Swapping algorithm, the achieved values of speedup are 3.63 and 3.89 for 2048x2048 and 4096x4096 data sizes, respectively. Taking into consideration positively that the speedup values of 3.54 and 3.89 obtained from All-to-all and Swapping algorithms, respectively are for data size which is larger than the one used to compare with (i.e., 4096x4096 not 4096x2048). Thus, it is very noticeable how much the performance of the proposed algorithms is better though the sequential complexity of Elster’s bit-reversal sequential algorithm is lower than that obtained for Numerical Recipes sequential bit-reversal that is employed in this paper.

6 Concluding Remarks

Fast Fourier Transform is widely used and still of interest since it plays a major role in several applications including telecommunications, image processing, time series and wave analysis, convolution, particle simulation, weather forecasting, Poisson’s equation solver, and digital signal processing. Bit-reversal is considered a very important step of the entire workload of FFT. It may degrade the overall performance of FFT significantly if it is not implemented efficiently. There are a lot of research articles, found in the literature, which aim to effectively implement the bit-reversal on both uniprocessors and symmetric multiprocessor systems. A few papers address implementing bit-reversal on either massively parallel processors or multicomputers (clusters) due to the possible risk resulted from communication overhead. In this paper, two novel parallel algorithms to implement the bit-reversal in the best possible shape, where the cache optimization is achieved, are proposed. These algorithms are named as All-to-all and Swapping algorithms and are tested on a cluster consisting of 14 nodes connected together via Myrinet switch. An empirical comparison between the two proposed parallel implementations of bit-reversal is made. Moreover, a detailed performance analysis, that includes run-time characteristics in the perspectives of communication overhead and computational time for both parallel implementations, is provided. The results show that an impressive speedup is achieved for both algorithms over the known Numerical Recipes sequential bit-reversal method especially when the data sizes and number of used processors get increased more and more. Additionally, significant speedups are obtained compared to those reported in a previous relevant work when considering the same data size and number of processors. It is also shown that the speedup obtained through adopting the Swapping algorithm is higher than the one obtained through adopting All-to-all algorithm for a fixed number of data sizes and number of processors. Many other interesting observations that validate the significance and correctness of our work are presented and explained to the extent level.

References


August 1989.


**Rami A. Al Na’mneh** is an Assistant Professor in the Department of Software Engineering at Jordan University of Science and Technology. He received his bachelor degree in Computer Engineering from the Jordan University of Science and Technology in 2000, the Master’s and Doctor’s of Philosophy Degrees from the University of Alabama in Huntsville in 2003 and 2006, respectively. His major research interests are in parallel programming, signal processing, and wireless communications.

**Khalid A. Darabkh** received his Ph.D. Degree in Computer Engineering from the University of Alabama in Huntsville, Alabama, USA in 2007. He is currently an Assistant Professor and the Chair of the Computer Engineering Department at the University of Jordan, Amman, Jordan. His main research interests include wireless and mobile communications, queuing theory, traffic management, multimedia systems and networking, congestion control architectures and resource allocation, parallel computing, and pattern recognition. Dr. Darabkh is a member of many honor societies such as Phi Kappa Phi, Tau Beta Pi, Eta Kappa Nu, and Sigma Xi.

**Iyad F. Jafar** received the B.S. Degree in Electrical Engineering from The University of Jordan in 2001, the M.S. Degree in Electrical Engineering from the Illinois Institute of Technology in 2004, and the Ph.D. Degree in Computer Engineering from Wayne State University in 2008. He is currently working as an Assistant Professor in the Department of Computer Engineering at the University of Jordan. His research interests are in signal and image processing, pattern recognition, and computer networks.