Parallel FFT Algorithms on Network-on-Chips

Jun Ho Bahn, Jungsook Yang and Nader Bagherzadeh
Department of Electrical Engineering and Computer Science
University of California, Irvine - Irvine, CA 92697-2625
Email : {jbahn,jyang12,nader}@uci.edu

Abstract

This paper presents several parallel FFT algorithms with different degree of communication overhead for multiprocessors in Network-on-Chip(NoC) environment. Three different methods of parallel FFT are presented. One is the reference parallel FFT for comparison, and the other two with well-distributed computation as well as reduced communication overhead. By evenly distributing parallel computation tasks which uses data locality, the execution time for completing each stage of FFT can be reduced. Moreover, by optimizing data exchanges we minimize the communication overhead. Depending on the communication regularity, one can select appropriate parallel FFT algorithm. By using the simulation results of our cycle-accurate SystemC NoC model with a parameterizable 2-D mesh architecture, and the performance analysis in time as well as complexity, our proposed algorithms are shown to outperform other parallel FFT algorithm or high-speed DSP implementations.

1. Introduction

The discrete Fourier transform (DFT), which is a linear transformation that maps n regularly sampled points from a cycle of a periodic signal, like a sine wave, onto an equal number of points representing the frequency spectrum of the signal [11], is of great importance to scientific and engineering fields since it is used by many applications from digital signal processing solutions to partial differential equations.

The fast Fourier transform (FFT) by Cooley and Tukey [8] computes the DFT reducing the complexity from $O(n^2)$ to $O(n \log_2 n)$. Since then, many variations of the FFT have been suggested.

There are many variations of FFT algorithm but in this paper we use the basic radix-2 decimation-in-time (DIT) FFT on N number of data where N is a power of 2, is considered for simplicity. DIT FFT takes the divide-and-conquer approach to decomposing the input data $x[n]$ into smaller subsequences and applies sub-DFT for each of them.

Many new variations of FFT algorithm appear to compute the transform as parallel as possible to do the same task in a shorter time. However, when it comes to parallel computing, limiting the communication overhead so that it would not hide the speedup, is very important. In this paper we study the efficient data distribution that will lessen the communication overhead in addition to shortening the computation time.

With the development of network-interconnected on-chip processors, a parallel FFT algorithm which exploits the on-chip grid processing elements is needed. Thus, in this paper, we introduce parallel radix-2 DIT FFT algorithms which maximize the data parallelism as well as minimize the communication overhead resulting higher performance in multiprocessor SoC. Our algorithms show the improvement over the existing parallel algorithms because

1. the resources are used in a balanced manner so that the divided task for each PE finishes in shorter time
2. the data locality is also well utilized to minimize the communication load
3. the concurrency of communication and computation in NoC environment can hide communication overhead.

Rest of the paper is organized as follows: In Section 2 we discuss the background of the parallel FFT on various architectures. In Section 3 we briefly describe the NoC based multi-processor platform to be assumed. In section 4 we discuss 3 different FFT algorithms and complexity analysis. In Section 5 the simulation result is presented. In Section 6, the analysis of proposed algorithms in complexity and speedup is provided. And the conclusion follows in the final section.

2. Background

Cui-xiang et al. [9], deduced the radix-2 decimation-in-time FFT algorithm into parallelizable sequential form, and

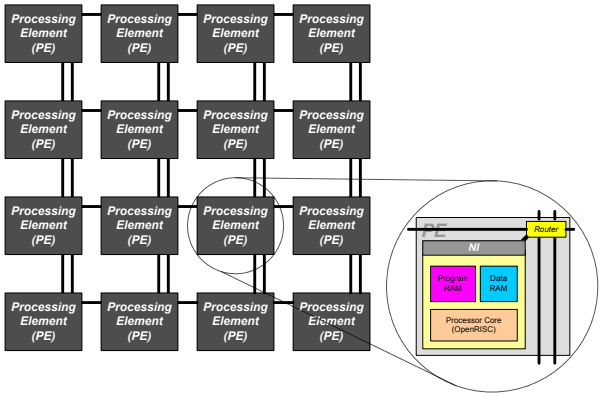


Figure 1. Prospective NoC-based multi-processor system platform in 4x4 mesh

transformed it to the new parallel FFT algorithm reducing the complexity of DFT to $O((n \log_2 n)/p)$, where p is the number of processors. They have suggested parallel radix-2 decimation-in-time FFT algorithm that are executed in parallel on the multiprocessors.

Among them are parallel FFT algorithms on parallel architecture such as vector processors. Swarztrauber [15] presented many implementations of the parallel FFT algorithm on vector processors. FFTs on shared-memory parallel architecture were studied [10, 14]. Since then many parallel FFT algorithms on various architectures have been studied by many other researchers [4, 6, 13].

Recently, the parallel FFT was suggested and mapped to a CDMA-based star-topology network-on-chip architecture in [12]. The authors suggested two mapping methodologies of a 16-FFT computation on the star-topology NoC architecture.

Efficient data distribution of the FFT algorithm using the Indirect Swapping Network (ISN) to improve data locality was suggested in [7]. However, in this approach, the distribution of twiddle factors on each processor becomes irregular and transformed data may not be in-order, resulting in additional reordering steps as post-processing. Our algorithms do the same thing by reducing the communication by half compared to the traditional one, but provide some regularity in arranging twiddle factors as well as residing data. Therefore, no additional post-processing such as reordering is needed.

3. NoC-based System Platform

Communication requirements of a conventional SoC system made of numerous cores can not be fully satisfied using single or multi-layer buses because of their poor scalability and bandwidth limitation among cores. The notion of utilizing Network-on-Chip (NoC) technology for the future generation of high performance and low power chips for myriad of applications, in particular for wireless communication and multimedia processing, is of great importance. For this purpose, we proposed a multi-processor system platform adopting NoC techniques as shown in Figure 1. As a component of system platform, the fundamental NoC techniques including the router architecture and a generic network interface (NI) are defined and implemented. Also as a component of processing elements, multiple compact OpenRISC cores are designed where some specific blocks such as I/D-cache, I/D-MMU (memory management unit), debug unit are omitted from the original OpenRISC core for simplicity and cost minimization. These compact OpenRISC cores are interconnected throughout 2D meshed network where each of OpenRISC core is encapsulated with network related components such as our generic network interface block and router as shown in Figure 1. Each router for on-chip interconnection, uses a wormhole flow control and adaptive routing algorithm as a method of reducing latency and improving network utilization. The delivery of data for each PE is done by packetized data communication throughout networks. For instance, each of OpenRISC cores can be reconfigured by loading instructions delivered throughout networks from the external host. Different from bus architectures, the network can allow simultaneous communication among nodes. This is one of factors utilized by optimization of communication overhead in our proposed parallel FFT algorithms.

4. parallel FFT algorithms

Our FFT algorithms can be separated into three steps. The first step is preprocessing where data is rearranged in bit-reverse order and divided by p blocks so that N/p data items are sequentially fed to each processing element(PE) from PE[0] to PE[p-1], where p is the total number of PEs and N is the size of data for FFT. The second step is the actual transformation. During the second step where the transform is executed, it can be further decomposed into sequential execution and parallel execution. The sequential FFT refers to the first $\log_2(N/p)$ stages of N-point FFT where N/p-point FFT is performed within each PE since the data to be transformed reside in the local memory of each PE. On the other hand, the parallel FFT represents the rest of $\log_2 p$ stages of original N-point FFT where data exchanges are needed because the index distance between a butterfly is larger than N/p.

In this section, three different parallel FFT algorithms are explained and compared of their performance. For the implementation, the FFT is divided by $\log_2 N$ stages of iterations. During each of the first $\log_2(N/p)$ stages, the $N/2p$-

butterfly computation is executed where no data exchange is needed between PEs. And in the next $\log_2 p$ stages, either N/p-butterfly computation or $N/2p$-butterfly operation is performed where $N/2p$ data exchange occurs depending on methods. The selection of upper or lower half of data for communication in $N/2p$ data exchange affects residence of local data in each PE and the distribution of resultant data which determines the complexity of the third step called the post-processing step. On the other hand, in order to ensure the synchronization of data in data exchanges between PEs, some mechanism of sending data and checking arrival of receiving data is necessary.

In these three steps, the preprocessing and the first $\log_2(N/p)$ stages of butterfly are common to all three algorithms discussed in this paper.

4.1. common part

Among three steps, the first and sequential part in the second step are common to all three different FFT algorithms. In the first step, the data of size N is bit-reversed (see the line 3 in Algorithm 1). Then the reordered data is sequentially segmented into p blocks of which the data size is N/p, and fed across the PEs (see the lines 4 to 8 in Algorithm 1).

Algorithm 1 Preprocessing

```
1:  Input: a = (a_0, a_1, ..., a_{n-1})
2:  Output: b = (b_0, b_1, ..., b_{n-1})
3:  b = bitReverse(a)
4:  for i ← 0 to p − 1 do
5:      for k ← 0 to N/p − 1 do
6:          P[i].c[k] ← b[i ∗ N/p + k]
7:      end for
8:  end for
```

Once N/p data are received by each PE, N/p-point FFT will be performed as the computation of the first $\log_2(N/p)$ stages of original N-point FFT. During this computation, there is no communication between PEs because all N/p-point FFT use the received N/p data locally as shown in Algorithm 2.

Algorithm 2 Local N/p-point FFT on each PE

```
1:  Input: c = (c_0, c_1, ..., c_{N/p−1})
2:  Output: c = (c_0, c_1, ..., c_{N/p−1})
3:  for i ← 0 to log_2(N/p) − 1 do
4:      l = 2^i, q = N/2l, z = w^q
5:      for k ← 0 to N/p − 1 do
6:          if ((i ∗ N/p + k) mod l = (i ∗ N/p + k) mod 2l) then
7:              c[k] = c[k] + c[k + l] ∗ z^m
8:              c[k + 1] = c[k] − c[k + l] ∗ z^m
9:          end if
10:     end for
11: end for
```

4.2. last $\log_2 p$-stage butterfly with data exchange

Now we describe three different parallel FFT algorithms suitable for multiprocessor environment. The next three methods are performed from the $\log_2(N/p)$-th stage to the $(\log_2 N - 1)$-th stage where the communication between PEs is needed. They are differentiated by the complexity of communication patterns and communication overhead. The first method was proposed by Cui-xiang et al. [9] which is based on N/p butterfly operations on each PE. Though the communication pattern of this method is comparatively simple, the required communication overhead is bigger than the others. The second method is enhanced by the even computation load as $N/2p$ butterfly operations and the reduced communication overhead. In third method, the communication overhead is minimized and the regularity of data distribution and twiddle factors in each PE is maintained by constructing a generic rule for selecting upper/lower half of $N/2p$ data in each PE.

Algorithm 3 method I

```
1:  Input: c = (c_0, c_1, ..., c_{N/p−1})
2:  Output: c = (c_0, c_1, ..., c_{N/p−1})
3:  j = log_2(p) + 1
4:  for e ← 0 to log_2(p) − 1 do
5:      t = 2^e, l = 2^{(e+log_2(N/p))}, q = n/2l, z = w^q, j = j−1, v = 2^j
6:      for i ← 0 to p − 1 do
7:          if (i mod t = i mod 2t) then
8:              receive data block from (i + p/v)th PE and store them into
                c[N/v] − c[N/v + N/p − 1]
9:              for k ← 0 to N/p − 1 do
10:                 m = (i ∗ N/p + k) mod l
11:                 c[k] = c[k] + c[k + N/v] ∗ z^m
12:                 c[k + N/v] = c[k] − c[k + N/v] ∗ z^m
13:             end for
14:             Send transformed data in c[N/v] − c[N/v + N/p − 1] to the
                (i + p/v)th PE.
15:         else
16:             Send the data of this PE to the (i − p/v)th PE.
17:             After the transformation, Receive data from (i − p/v)th PE and
                store them into c.
18:         end if
19:     end for
20: end for
```

Method I As shown in Algorithm 3, N/p-pair butterfly computation is performed in one PE while the other paired PE just sends whole N/p data to the corresponding PE and waits until transformed data return. In terms of load balance, this algorithm shows inefficiency. For communication overhead, $2N/p$ data are exchanged at every stage where two N/p data transfers are needed for sending and receiving on an idle PE.

Algorithm 4 method II

Input: $c = (c_0, c_1, ..., c_{N/p-1})$
Output: $c = (c_0, c_1, ..., c_{N/p-1})$
$j = log_2(p) + 1$
for $e \leftarrow 0$ to $log_2(p) - 1$ **do**
  $t = 2^e, l = 2^{(e+log_2(N/p))}, q = n/2l, z = w^q, j = j-1, v = 2^j$
  for $i \leftarrow 0$ to $p - 1$ **do**
    if (i mod t = i mod $2t$) **then**
      Send upper $N/2p$ data stored in $c[\frac{N}{2p}] - c[\frac{N}{p} - 1]$ to the $(i + p/v)^{\text{th}}$ PE.
      Receive $N/2p$ data from $(i + p/v)^{\text{th}}$ PE and store them to upper $N/2p$ location of $c[]$.
      /*transform*/
      for $k \leftarrow 0$ to $\frac{N}{2p} - 1$ **do**
        $m = (i * N/p + k) \bmod l$
        $c[k] = c[k] + c[k + \frac{N}{2p}] * z^m$
        $c[k + \frac{N}{2p}] = c[k] - c[k + \frac{N}{2p}] * z^m$
      end for
      Send upper $N/2p$ of transformed data stored in $c[\frac{N}{2p}] - c[\frac{N}{p} - 1]$ back to the $(i + p/v)^{\text{th}}$ PE.
      Receive the $N/2p$ transformed data from $(i+p/v)^{\text{th}}$ PE and store them back to upper $N/2p$ location of $c[]$.
    else
      Send lower $N/2p$ data stored in $c[0]-c[\frac{N}{2p}-1]$ to the $(i-p/v)^{\text{th}}$ PE.
      Receive $N/2p$ data from $(i - p/v)^{\text{th}}$ PE and store them to lower $N/2p$ location of $c[]$.
      /*transform*/
      for $k \leftarrow 0$ to $\frac{N}{2p} - 1$ **do**
        $m = (i * N/p + k + N/2p) \bmod l$
        $c[k] = c[k] + c[k + \frac{N}{2p}] * z^m$
        $c[k + \frac{N}{2p}] = c[k] - c[k + \frac{N}{2p}] * z^m$
      end for
      Send lower $N/2p$ of transformed data stored in $c[0] - c[\frac{N}{2p} - 1]$ back to the $(i - p/v)^{\text{th}}$ PE.
      Receive the $N/2p$ transformed data from $(i-p/v)^{\text{th}}$ PE and store them back to lower $N/2p$ location of $c[]$.
    end if
  end for
end for

Method II In this enhanced method, each PE does not send out whole data block. Instead it only sends out either of the upper or the lower $N/2p$ data to the paired PE. By doing this, each PE performs $N/2p$-butterfly computation. As a result, the computation load among PEs is well balanced. Details of this method is provided in Algorithm 4. In order to keep the regularity in communication pattern between PEs, either of $N/2p$ computed data should return to the original PE resulting in an increase for the communication overhead. With comparison to the previous method, the amount of data exchange is the same as $2N/p$, but two $N/2p$ data transfers are needed for sending and receiving on each PE. As a result, the communication latency can be reduced maximally in half of previous method.

Method III Basic approach of the third method is very similar with [7]. However, with respect to the regularity of assigning twiddle factors and the distribution of transformed results, [7] needs additional effort to control the index of twiddle factors and make transformed results in-order, which adds programming complexity and increases the total execution time. By observing the communication pattern in method II, some redundancy can be detected. If one of $N/2p$ data in the given PE may be kept during the whole FFT computation, the step of returning from the paired PE and sending to the paired PE for next stage can be merged without losing regularity in twiddle factors as well as the distribution of transformed data. To do this, a generic rule to keep either of upper or lower half of data locally for each PE is formulated as shown in Figure 2. For simple explanation, the property of each PE is set by either BLACK or WHITE. Being black is assumed to keep the lower $N/2p$ data while being white is to keep the upper $N/2p$ data, and

Algorithm 5 method III

Input: $c = (c_0, c_1, ..., c_{N/p-1})$
Output: $c = (c_0, c_1, ..., c_{N/p-1})$
$j = log_2(p) + 1$
for $e \leftarrow 0$ to $log_2(p) - 1$ **do**
  $t = 2^e, l = 2^{(e+log_2(N/p))}, q = n/2l, z = w^q, j = j-1, v = 2^j$
  for $i \leftarrow 0$ to $p - 1$ **do**
    if (i mod t = i mod $2t$) **then**
      $i' = i + p/v$
      if ($t = 1$) **then**
        Send upper/lower $N/2p$ data stored in $c[\frac{N}{2p}] - c[\frac{N}{p} - 1]$ to the $(i + p/v)^{\text{th}}$ PE.
        Receive $N/2p$ data from $(i + p/v)^{\text{th}}$ PE and store them to upper/lower $N/2p$ location of $c[]$.
      end if
      /*transform*/
      for $k \leftarrow 0$ to $\frac{N}{2p} - 1$ **do**
        $m = (i * N/p + k) \bmod l$
        $c[k] = c[k] + c[k + \frac{N}{2p}] * z^m$
        $c[k + \frac{N}{2p}] = c[k] - c[k + \frac{N}{2p}] * z^m$
      end for
    else
      $i' = i - p/v$
      if ($t = 1$) **then**
        Send lower $N/2p$ data stored in $c[0] - c[\frac{N}{2p} - 1]$ to the $(i - p/v)^{\text{th}}$ PE.
        Receive $N/2p$ data from $(i - p/v)^{\text{th}}$ PE and store them to lower $N/2p$ location of $c[]$.
      end if
      /*transform*/
      for $k \leftarrow 0$ to $\frac{N}{2p} - 1$ **do**
        $m = (i * N/p + k + N/2p) \bmod l$
        $c[k] = c[k] + c[k + \frac{N}{2p}] * z^m$
        $c[k + \frac{N}{2p}] = c[k] - c[k + \frac{N}{2p}] * z^m$
      end for
    end if
    if ($e \neq p - 1$) **then**
      if (i' mod $2t$ = i' mod $4t$) **then**
        Send upper/lower $N/2p$ of transformed data stored in $c[]$ back to the $(i' + 2p/v)^{\text{th}}$ PE.
        Receive the $N/2p$ transformed data from $(i'+2p/v)^{\text{th}}$ PE and store them back to upper/lower $N/2p$ location of $c[]$.
      else
        Send upper/lower $N/2p$ of transformed data stored in $c[]$ back to the $(i' - 2p/v)^{\text{th}}$ PE.
        Receive the $N/2p$ transformed data from $(i' - 2p/v)^{\text{th}}$ PE and store them back to upper/lower $N/2p$ location of $c[]$.
      end if
    end if
  end for
end for

vice versa. The rule starts from 2x2 mesh. In order to construct such a selection map, the neighboring PE keeps the other part of $N/2p$ data than the current PE and the diagonal PE keeps the same part of $N/2p$ data as the current one. With 2x2 mesh which is assumed as a single PE, the selection map for 4x4 mesh can be built in a similar manner. That is, the map for adjacent 2x2 mesh is complemented by the original one for 2x2 mesh. And the map for the diagonal 2x2 mesh is the same as the original one. In the same fashion, a selection map for bigger mesh can be constructed.

By using the constructed selection map for the given mesh, either of upper or lower $N/2p$ data always reside in the corresponding PE and the other $N/2p$ data can be directly forwarded during the last $\log_2 p$ stages. Therefore, in this method, the communication overhead can be further reduced by half with comparison to method II. Details of this method is provided in Algorithm 5.

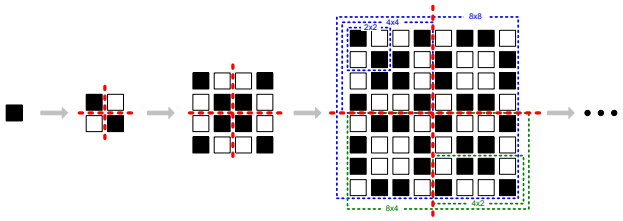
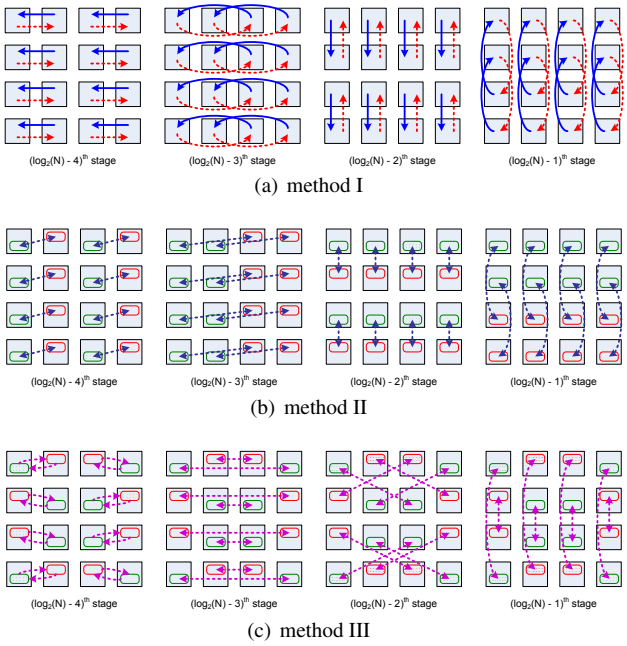


Figure 2. Procedure of constructing generic selection maps for variable size of meshes

In the previous method, transformed data need to return to the PE which sends out the original data. However, in this method, PE sends out them directly to the next destination PE which needs data in the following stage. The rationale of this algorithm is simplicity and regularity for communicating data and keeping twiddle factor stationary in order to reduce communication overhead.

Figure 3 provides examples of communication pattern of the 64-point FFT onto 16 PEs using different methods. At each stage, data communication paths are slightly different from each other. The performance on each method will be varied. Also, the way of the data exchange between PEs such as the number of data exchanges and whether the computed data items return or not, is different. Figure 3(a) illustrates the communication pattern in method I where a full line arrow represents N/p data sending and a dotted line arrow represents N/p data return after completion of butterfly operations. The PE receiving N/p data performs N/p butterfly operations while the PE sending N/p data just waits for returning data. Therefore, each PE does not get the same workload. That is the waste of resource and leads to performance degradation that could have increased if the efficient data mapping was used. Figure 3(b) shows the communication pattern in method II where round rect-



(a) method I

(b) method II

(c) method III

Figure 3. Communication Pattern of 64-point FFT in 4x4 mesh

angles represent either upper or lower $N/2p$ data block and a bidirectional arrow represents $N/2p$ data sending and returning after butterfly operation. In method II, exploiting data locality efficiently and sending only $N/2p$ number of data per PE, the communication cost is reduced also the computation time is minimized by distributing the workload over the PEs in a balanced manner. In method III as shown in Figure 3(c) where a bidirectional arrow has a meaning of sending $N/2p$ data, more optimization in reducing the communication load is achieved by removing $N/2p$ data returning which exists in method II.

5. Experiment

5.1. Simulation Environment

The algorithms proposed in this paper were implemented and tested on our cycle-accurate SystemC simulator modeling Network-on-Chip environment. The components of the simulator are various number of compact OpenRISC processing elements and the associated routers and shared communication channels which connect the routers. The compact OpenRISC processing element has local program/data memory which is directly connected with OpenRISC core for fast access. The C program for each OpenRISC core was automatically generated from the kernel C which is described based on the proposed methods. Finally the

binary code for each PE was compiled using OpenRISC toolchains. The router model used in this system is the same as [5]. The overall operating clock frequency is assumed to be 100 MHz. The proposed parallel FFT algorithms are implemented and run by varying system parameters such as the number of PEs and the size of FFT. Data and the twiddle factors are represented in 16-bit real and 16-bit imaginary in 2's complement format. 14 bits are used for the fractions and scaling is performed to avoid overflow. The input data are assumed to be fed in packet format from the external host which is connected to PE[0]. Therefore, the corresponding input data for each PE are delivered throughout network.

5.2. Simulation Result

The three different parallel FFT algorithms described in this paper were tested and the execution time for completing all stages was measured. The result of the simulation is shown in Table 1 where pFFTv1, pFFTv2, and pFFTv3 represent the reference parallel algorithm method I, II, and III, respectively.

Table 1. Cycle counts for parallel FFT algorithms

# of FFT	single	2x2			4x4			8x8		
		pFFTv1	pFFTv2	pFFTv3	pFFTv1	pFFTv2	pFFTv3	pFFTv1	pFFTv2	pFFTv3
32	3149	2629	2802	2267						
64	7073	3989	3846	3311	3762	4609	2680			
128	16269	7120	6250	5715	4793	5289	3360			
256	37424	14364	12527	11939	6917	6717	4788	5503	7265	4827
512	85237	29546	24856	24322	11528	10230	8760	7114	8138	5702
1024	191885	62774	52379	51885	22427	18293	16718	10050	10357	7726
2048	427190	134761	113186	112544	44993	35119	33560	16780	14977	12273
4096	941762	289806	245595	244903	91498	69591	68057	31882	25600	22944
8192	2058984	621878	532609	531676	190101	144607	142613	62630	47456	44802
16384	4469507	1329997	1150615	1149152	398403	305511	302882	128194	94995	92147
32768	9380584	2834265	2474675	2472211	836961	649386	645440	258356	188763	172303

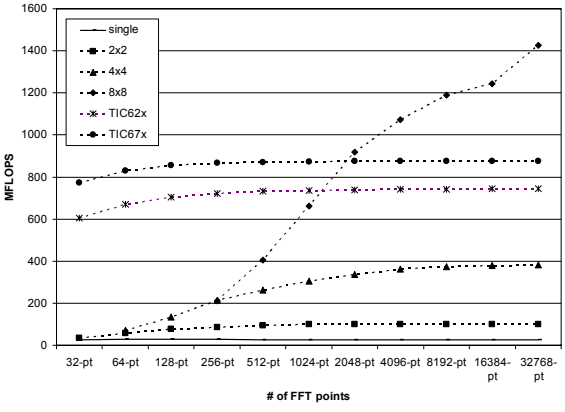


Figure 4. benchFFT comparison in speed

The simulation shows that parallel FFTs require fewer clock cycles for computingtransform on the same size of data as the number of processing elements increases. Also, the proposed algorithms such as methods II and III outperform the reference parallel FFT algorithm, method I. After running the simulation with different system parameters, the performance gain of method II over the reference parallel FFT method I is greater than 24% in 8x8 PE model when the data size is larger than 2^{13}. The parallel FFT method III shows 33% of performance gain when the data size is 2^{15} and the average performance gain is 25%. As a result, the proposed algorithms are effective in utilizing the parallel processing capability of the architecture and achieve scalable performance.

When the performance results are compared with the cycle counts of FFT implemented on TI C62x and C67x architecture [2, 3] where the formula of cycle counts are $(2N + 7)\log_2 N + 34 + N/4$, and $2N \log_2 N + 42$, respectively, our proposed algorithms spend less clock cycles. According to benchFFT [1], the results can be normalized by MFLOPS which is a scaled version of the speed, defined by:

$$MFLOPS = \frac{5N \log_2 N}{time\ for\ one\ FFT\ in\ \mu sec.} \quad (1)$$

where N is the total number of FFT points. Figure 4 shows the normalized performance graph of method III with TI DSP's based on benchFFT. In Figure 4, TIC62x and TIC67x are assumed to run at 300 MHz and 350 MHz, respectively. Therefore, even though the operating clock frequency is much lower than TI DSP's, i.e. 100 MHz, our proposed algorithms show better performance starting with 2048-point FFT in 8x8 mesh. If the operating clock frequency is set to be as TI's, then overall performance of the proposed methods in 4x4 and 8x8 mesh outperform.

6. Speedup Comparison

The speedup $Gain$ of the parallel algorithm can be compared with the sequential one, defined by $Gain = T_{serial}/T_{parallel}$. Equation 2 and Equation 3 represent the speedup gain by method II and method III, respectively.

$$\begin{aligned} Gain &= \frac{t_c N \log_2 N}{t_c(N/2p)\log_2(N^2/p) + 2Ct_w(N/2p)} \\ &= \frac{2pt_c N \log_2 N}{2t_c N \log_2 N + 2Ct_w N - t_c N \log_2 p} \\ &= \frac{pT_c}{T_c + C'} \end{aligned} \quad (2)$$

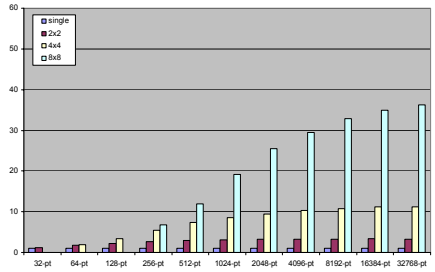
where $T_c = 2t_c N \log_2 N$ and $C' = 2Ct_w N - t_c N \log_2 p$.

$$Gain = \frac{t_c N \log_2 N}{t_c(N/2p)\log_2(N^2/p) + Ct_w(N/2p)}$$

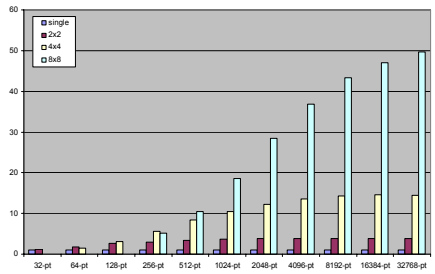
$$= \frac{2pt_cN\log_2 N}{2t_cN\log_2 N + Ct_wN - t_cN\log_2 p}$$
$$= \frac{pT_c}{T_c + C''} \quad (3)$$

where $C'' = Ct_wN - t_cN\log_2 p$.

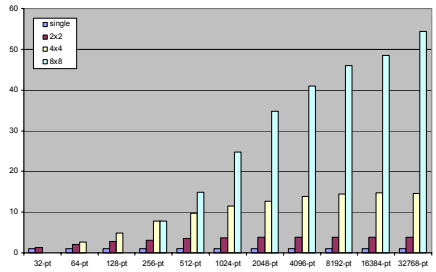
Figure 5(a), Figure 5(b), and Figure 5(c) show the speedup of various number of FFTs from single sequential execution for method I, method II, and method III, respectively. As the number of points in FFT, N increases, the speedup ratio reaches the number of processing elements, p, as shown in Eq. 2, and 3. Also, by comparing with Figure 5(a), our proposed methods such as method II, and III show higher speedup.



(a) method I



(b) method II



(c) method III

Figure 5. Speedup comparison

7. Conclusion

In this paper, we present several parallel FFT algorithms suitable for multiprocessor environment by equally distributing the data points across PEs and optimizing communication overhead. Specifically, in order to minimize the communication overhead and keep data arrangement regular, a generic data selection rule for data exchange is also introduced. The balanced work load and minimized communication overhead lead to fast operation of FFT. By using a cycle-accurate simulation and complexity analysis, we showed that our algorithms outperform other parallel FFT algorithms with similar specifications.

References

[1] FFT benchmark results. http://www.fftw.org/speed.

[2] TMS320C62x DSPs C62x core benchmarks from texas instruments. http://www.ti.com.

[3] TMS320C67x floating point DSPs C67x core benchmarks from texas instruments. http://www.ti.com.

[4] A. Averbuch, E. Gabber, B. Gordissky, and Y. Medan. A parallel FFT on an MIMD machine. *Parallel Computing*, 15(1-3):61–74, 1990.

[5] J. H. Bahn, S. E. Lee, and N. Bagherzadeh. On design and analysis of a feasible network-on-chip (NoC) architecture. In *ITNG '07*, pages 1033–1038, Washington, DC, USA, 2007. IEEE Computer Society.

[6] D. H. Bailey. FFTs in external or hierarchical memory. In *PPSC*, pages 211–224, 1989.

[7] S. Barua, R. K. Thulasiram, and P. Thulasiraman. Improving data locality in parallel fast Fourier transform algorithm for pricing financial derivatives. *ipdps*, 14:235a, 2004.

[8] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematical Computation*, 19:297–301, 1965.

[9] Z. Cui-xiang, H. Guo-qiang, and H. Ming-he. Some new parallel fast Fourier transform algorithms. In *PDCAT '05*, pages 624–628, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Z. Cvetanovic. Performance analysis of the FFT algorithm on a shared-memory parallel architecture. *IBM Journal of Research and Development*, 31(4):435–451, 1987.

[11] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *An Introduction to Parallel Computing*. Addison Wesley, 2003.

[12] D. Kim, M. Kim, and G. E. Sobelman. Parallel FFT computation with a CDMA-based network-on-chip. In *ISCAS (2)*, pages 1138–1141, 2005.

[13] C. V. Loan. *Computational frameworks for the fast Fourier transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[14] A. Norton and A. J. Silberger. Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared-memory architectures. *IEEE Trans. Comput.*, 36(5):581–591, 1987.

[15] P. N. Swarztrauber. FFT algorithms for vector computers. *Parallel Comput.*, 1:45–63, 1984.