Multithreaded Algorithms for the Fast Fourier Transform

Parimala Thulasiraman†, Kevin Theobald, Ashfaq Khokhar† and Guang R. Gao

Department of Electrical and Computer Engineering
140 Evans Hall
University of Delaware
Newark, DE 19716
{thulasir,theobald,ggao}@capsl.udel.edu, ashfaq@eecis.udel.edu

†Authors for correspondence:thulasir@capsl.udel.edu, ashfaq@eecis.udel.edu

Abstract

In this paper we present fine-grained multithreaded algorithms for the Fast Fourier Transform (FFT) problem. The FFT problem has been formulated in two distinct, unique ways. The first approach, the receiver-initiated algorithm, realized the FFT iterations as a parent-child relationship while fully exploiting the underlying parallelism. The number of threads is directly proportional to the input size. The second approach, the sender-initiated algorithm, follows a data-flow model based on the producer-consumer style of programming and can be adopted to different architectural parameters for achieving high performance. We implement the proposed algorithms on the EARTH (Efficient Architecture for Running THreads) platform. For both the algorithms we analytically calculate the number of threads created in the algorithms and present asymptotic and experimental results in the paper. Our implementation results show that receiver initiated algorithm performs poorly when number of processors is small. However, for large number of processors, both the algorithms perform well, yielding execution times of only 10 msec for an input of 16 K data points on a 64 processor machine, assuming each processor running at 140 MHz clock speed.

1 Introduction

Traditionally, digital image/signal processing algorithms are computationally intensive because of the large amount of data involved in the underlying applications. For example, a typical multispectral image may have a resolution of 8192×8192 pixels with 8 bits per pixel and 125 spectral bands, resulting in a spatial data set containing more than 8 Gbytes for each scene. Similarly, application of inverse scattering techniques to obtain material properties of the objects in a target image involves solving large sparse system of linear equations where matrices typically grow as big as $100,000 \times 100,000$. Performing image transform operations such as FFT, DCT, or Wavelet, in real time on such large data sets requires high performance computing [Pit93, Gol65, Ung58, Pea68]. This paper investigates multithreaded algorithms and implementations for Fast Fourier Transform (FFT) on multithreaded computing paradigms.

On parallel machines the image/signal data is partitioned into blocks, the blocks are distributed to processors, and any non-trivial image operation requires communication among processors to complete the task. Two types of latencies are normally related to multiprocessor systems. The latency that occurs due to remote memory access or data transfers is termed as *Communication latency* [Hwa93]. The other important latency event is *synchronization latency* [Hwa93]. In some algorithms, two events A and B executed on two different processors may be dependent on one another thereby inhibiting event A to continue until the other event B that it depends on has occurred. In such situations the processor executing event A would have to just wait. Less time is wasted if the latency is hidden by allowing the waiting processor to execute another task. Thus useful work is performed by the waiting processor. There are many techniques at the software and hardware levels (such as superscalar, superpipelined, VLIW, prefetching) [Hen96, Sto80] to hide or tolerate such latencies. But the most general technique is *multithreading*. Multithreading tries to overlap computation with communication by means of *threads* (a thread is a set of instructions executed sequentially) thereby tolerating latencies.

The FFT problem has been studied extensively on images [Pra93, Cho93, Jam93] which can be characterized as a 2D matrix with some data points. In general the 2D-FFT algorithm can be realized by performing the 1D-FFT on every row of a matrix, transpose the resulting matrix and perform the 1D-FFT on every column. The 1D-FFT problem is vitally important in solving the 2D-FFT problem and so in this paper as a first step we concentrate on developing efficient implementation of the 1D-FFT algorithm on a multithreaded architecture. The FFT on an input of N data points requires (N/2) log_2(N/2) complex multiplication operations which take most of the computation time for large data sets. Therefore, parallel processing becomes highly necessary to solve the problem. The FFT problem has been studied on various parallel machines. It can be well parallelized using shuffle exchange network [Ang92, Bur92, Sto71] . Other parallel implementations have been performed on linear arrays [Tho83], hypercubes [Ang90, Ang92], and mesh architectures [Cyp89, Kam87]. Sohn et. al. [Soh97] has studied the FFT problem on EM-X multithreaded architecture. Given N points (N is a power of 2) and P processors, N/P points are partitioned and distributed to each of the processors. The iterative FFT algorithm is implemented on each processor by creating h threads in each processor to

handle N/P points. Each thread operates on (N/Ph) points. It is claimed that on the EM-X architecture, 2 to 3 threads perform the best overlap with communication. Matteo Frigo and Steven Johnson [Fri99] have developed a set of library C functions called codelets to compute the DFT for arbitrary image size and real or complex numbers. A compiler called *genfft* has been developed that takes the input N at compile time and generates a set of optimized codelets to calculate the DFT for N points. At runtime, they use a dynamic programming algorithm to determine the best set of codelets to execute. The algorithm is portable and adaptable on various architectures. A multithreaded version of the Cooley-Tukey DFT algorithm using the divide and conquer approach has also been developed using the multithreaded language Cilk [Lei99].

Due to imbalance in computation and communication in an FFT algorithm on parallel platforms, it makes it an ideal candidate for multithreaded platforms. In this paper, we study the FFT problem on non-preemptive multithreaded architectures. We develop two different dataflow style algorithms for the FFT problem.

The first algorithm is a fine grained algorithm referred to as the *receiver-initiated* algorithm. In this scheme a parent-child relationship is established between threads, while fully exploiting the underlying parallelism in the application. The number of threads is directly proportional to the input size. The second algorithm, called the *sender-initiated* algorithm, is a coarse-grained algorithm, where the number of threads can be set to be equal to the number of processors. This algorithm models the FFT problem as the producer consumer problem. This algorithm can be adopted to different architectural parameters for achieving high performance.

The platform used to study our experiments is EARTH- *Efficient Architecture for Running THreads* [Hum96, The99]-which is a fine-grained, non-preemptive, dataflow architecture. We define a fine-grained multithreaded paradigm as the one that has abundant number of threads, assumed balanced work load across processor, and cost of switching between threads is minimal [The99]. We present analytical and experimental results for both the algorithms. Our implementation results show that receiver initiated algorithm performs poorly when number of processors is small. However, for large number of processors, both the algorithms perform well, yielding execution times of only 10 msec for an input of 16 K data points on a 64 processor machine, assuming each processor running at 140 MHz clock speed.

The rest of the paper is organized as follows: Section 2 presents the multithreaded algorithms. Analytical results are presented in Section 3. The experimental framework of the EARTH model is given in Section 4. Performance results of the algorithms are presented in Section 5. Our observations and conclusions are presented in Section 6.

2 Fine-Grained Multithreading Algorithm

The FFT problem may be solved recursively or iteratively. In general, iterative version of the FFT algorithm is more suitable for distributed memory parallel machines [Kum94, Akl89, Cor90, Lei92]. In the following we present two dataflow style algorithms for the FFT problem.

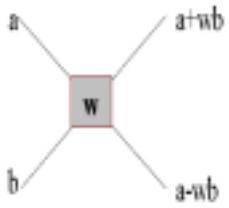


Figure 1: The Butterfly Operation

These algorithms differ from each other in terms of dataflow style and in number and size of threads employed.

2.1 Receiver-Initiated Algorithm

The *receiver-initiated* algorithm is a fine-grain multithreaded algorithm.

Let us assume we have N (N=2^m) data elements and P (P=2^p) processors. The data is block partitioned into N/P data points and is distributed to P processors. Each point or data element is a complex number. Initially, each processor performs the *butterfly* (figure(1)) computation for its local data points in a single thread. A butterfly computation can be described as follows: a and b are points or complex numbers.The upper part of the butterfly operation computes the summation of a and b with a twiddle factor w while the lower part computes the difference. At each iteration for N/P data points, N/2P summations and N/2P differences are performed in each processor.

In general, it is known [Kum94] that an FFT with blocked data distribution of N elements on P processors requires communication for $\log P$ iterations and terminates at $\log N$ iterations. At the end of the $\log N - \log P$th iteration, the latest computed values for N/P data points exist in each processor. Therefore during the first $\log N - \log P$ iterations, a sequential FFT algorithm can be used inside each processor. At this point, the processors switch to the multithreaded phase of the algorithm described below.

Conceptually, the multithreaded phase starts from the output. Conisder the N output data points at the end of the $\log N$th iteration. The butterfly computation for any data point in this iteration requires two data points from the previous iteration (Figure(1)).

Therefore, considering the dataflow style, each processor, for each of its N/P local data points, sends out two threads: one to itself and another to the destination processor holding the other data point. The set of parameters of a thread is comprised of a function name, destination processor id., and iteration number. These threads are sent to obtain the data values computed at the previous iteration. At a particular iteration i the processors upon receiving these threads send out two more threads with iteration number *i-1*. This process continues until the $\log Pth$ iteration is reached. At this point, the latest locally computed data element is transferred to the corresponding requester. When the requester receives the two data elements, the butterfly operation is performed.

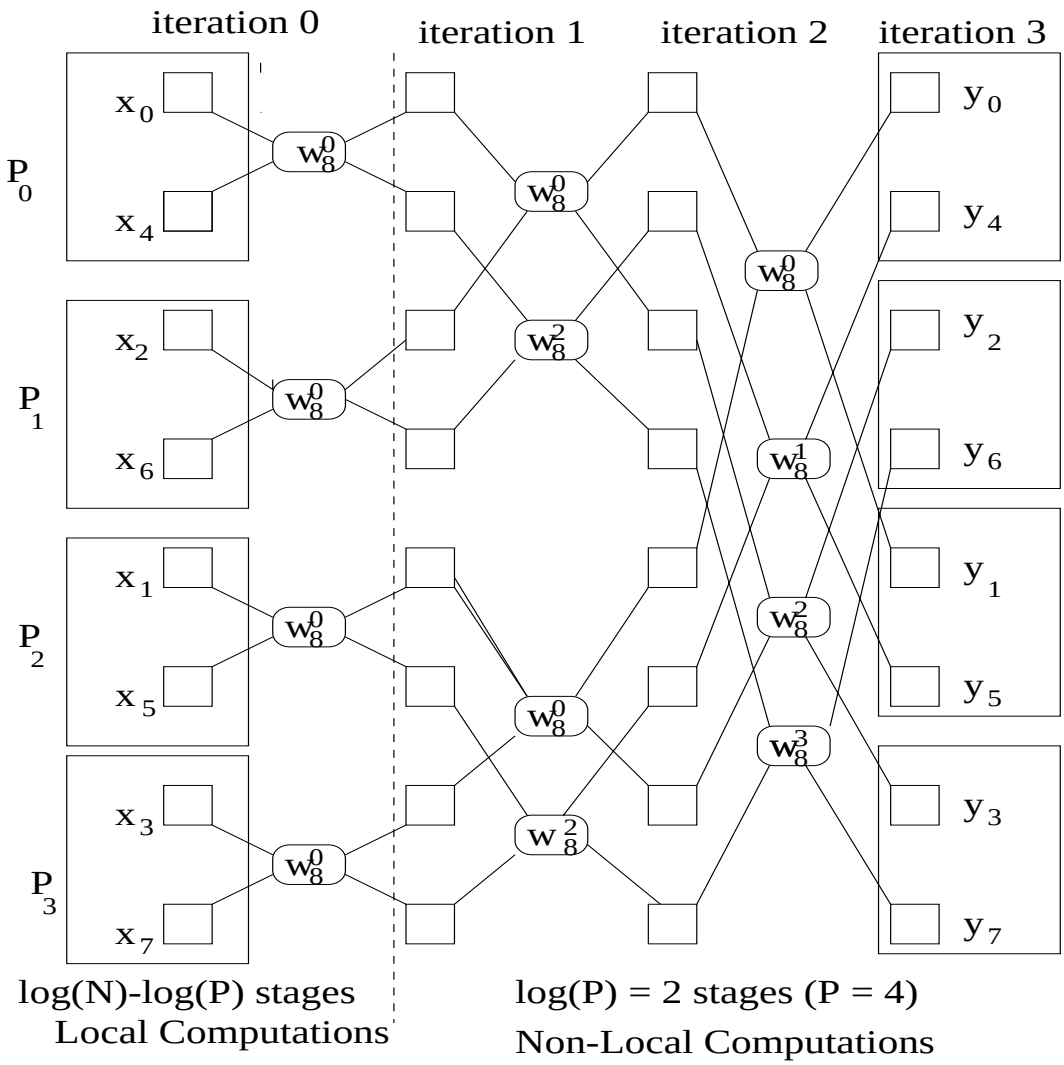


Figure 2: The Cooley-Tukey [Coo77] signal flow graph (N=8)

Note that the butterfly computation is performed only after the two data values has arrived for the two threads sent out. The thread computing the butterfly computation is therefore synchronized by two signals. The arrival of these signals acknowledges the arrival of the two data elements computed at the previous iteration.

The above algorithm can be illustrated with an example. Consider Figure(2) with (N = 8) data elements and (P = 4) processors. Each processor has 2 elements. P_0 has x_0,x_4, P_1: x_2,x_6, P_2: x_1,x_5 and P_3: x_3,x_7. The first $\log N - \log P$ iterations are performed locally by each processor. And therefore the computed values of the butterfly operation for each data element is available at the end of the $\log N - \log P$ iterations. The processors then switch to the multithreaded version of the algorithm.

Let us consider one particular data element y_7 at $\log Nth$ iteration, that is at iteration 3 (refer to Figures(2) and (3). The receiver initiated approach starts from y_7 and proceeds

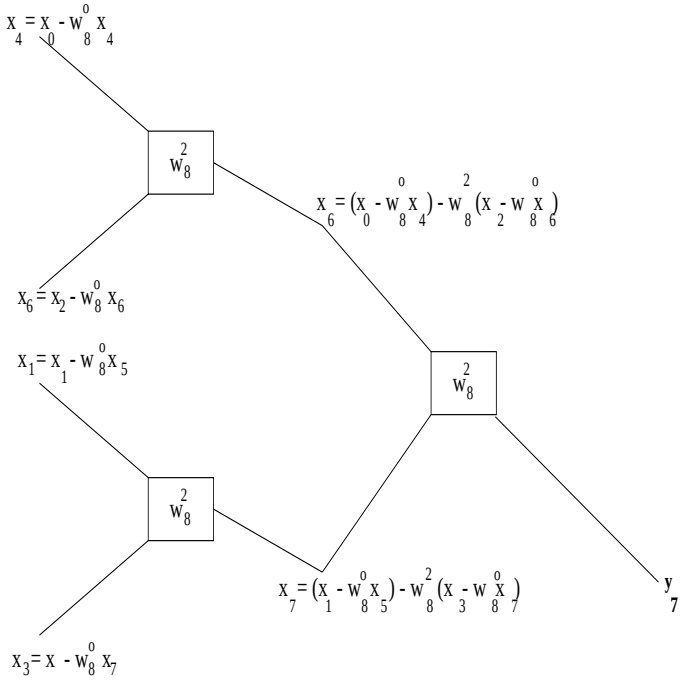


Figure 3: One pass of the flow graph

backwards for $\log P = 2$ iterations.

P_3 which holds y_7 sends out two threads: its mate processor P_1 and itself, P_3, for the computed data elements x_6 and x_7 at iteration 2. Therefore, P_3 has to receive two signals, one each from P_1 and P_3. Of course, at iteration 2, x_6 and x_7 have not yet been computed. Therefore, consider the actions of processor P_1 at iteration 2. This involves: Processor P_1 upon receiving and executing the thread from P_3 sends out two more threads to P_0 and P_1 for data elements at iteration 1. At iteration 1, the latest locally computed data values exist and P_0 and P_1 transfer x_4 $(x_0 - w_8^0 x_4$) and x_6 $(x_2 - w_8^0 x_6$) respectively to processor P_1 which requested these data at iteration 2. At this point P_1 computes the butterfly operation and sends the result back to P_3 which requested it at iteration 2. Now P_3 has received one signal and one data element. Similarly the same type of communication is performed to receive the second signal. When the 2 signals arrive at P_3 the butterfly operation is performed and y_7 is computed. The flow graph of y_7 is shown in fig(3).

In this scheme, a *parent-child* relationship is established between threads. This parent-child relationship and the synchronization signals which act as acknowledgment signals allow efficient multithreading. It also ensures the correctness of the program without any data race conditions or corruption of data. Also, there are equal number of threads per processor thereby balancing the work load. For $\frac{N}{P}$ data points per processor, $2^i \frac{N}{P}$ threads are sent out, where $i = 1 \cdots \log P$. The processors execute the butterfly computation of each of these threads as per the arrival of signals and these could be in any order. Therefore, a processor either sends out threads or performs computations; it never sits idle. The algorithm efficiently overlaps computation with

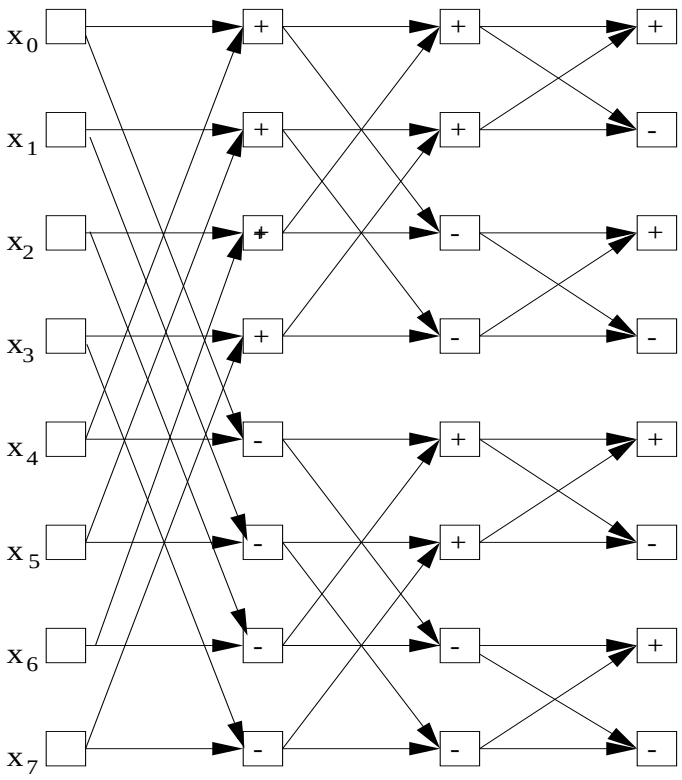


Figure 4: Signal Flow Graph [Gen66]

communication. It is easy to show that the total number of threads created is $2(P-1)N$ and the number of butterfly cmputations performed is $N \log P$. The analytical section will explain the complexity analysis in more detail.

2.2 Sender-Initiated Approach

In the *sender-initiated* algorithm, the number of threads is fixed at compile time to be equal to $\frac{N}{B}$, where B is the *block size*, consisting of *contiguous* data elements. The $\frac{N}{B}$ threads are distributed to each of the processors in a round-robin fashion, thereby balancing the load across the processors. Each processor performs the FFT computation on its B data points.

In the FFT algorithm, each data point requires a mate data point to compute the butterfly operation. The mate may be located in a different thread in a different processor. In this case, a thread for each of its points sends the recently computed value to the thread containing its mate point. The sending and receiving of information requires certain amount of synchronization between the producing and consuming threads. Also, the mate points change at each iteration during the execution of the algorithm. However the communication and synchronization is performed at a block level. That is each thread, computes the values for its B points and uses a split phase transaction operation to move data to its mate thread. Note that, all the mate points are located within the same thread which makes sending and receiving easier. This is

possible due to the contiguous distribution of points in each thread.

As mentioned above, each thread comsumes data from previous iteration and produces data for next iteration. This producer-consumer function is realized as a second level thread (a thread within a thread), called fiber. We explain the concept of second-level threads as follows. The data is produced in a producer thread and using a split-phase transaction operation, the produced values are delivered to the corresponding consumer thread in another processor. The consumer thread in the other processor is activated when it receives a synchronization signal from its mate thread. Note that at each iteration, the threads have to determine the location of its mate thread and set up the synchronization slots appropriately during runtime. Therefore, the producer and consumer threads act as second level threads (fibers) within a threaded function. The synchronization slots act as acknowledgment signals and the second-level threads comprise a data-flow style of programming.

We illustrate the above producer-consumer approach with an example (Figure 4) (We have represented the signal flow graph differently [Gen66] for easier explanation of the sender-initiated algorithm). Assume N=8, P=4 and B=2. Then there are N/B = 4 threads. Points x_0, x_1 are in thread 0; x_2,x_3 in thread 1; x_4,x_5 in thread 2; x_6,x_7 in thread 3. These threads are distributed to each of the 4 processors (thread 0 is executed by P_0, thread 1 by P_1,etc.,). In Figure 4, all edges going upwards are marked positive (+) and all edges going downwards are marked negative (-). This indicates that a+bw or a-bw is computed at + and - marked points respectively. Consider the first iteration of the algorithm. The mate points of x_0, x_1 in thread 0, P_0 are x_4, x_5 and are located in thread 2, P_2. Thread 0 computes $x_0 w^n$, $x_1 w^n$ (where n=0 or 1 ... or 3) and sends the computed values to the consuming mate thread (which is in thread 2 of P_2). Similarly the consuming thread of thread 0 in P_0 receives the computed values ($x_4 w^n$, $x_5 w^n$ (where n=0 or 1 ... or 3) from the producing thread of thread 2, P_2. In the next iteration, thread 0's mate thread is thread 1. The setting up of synchronization slots between the threads is performed at the start of the new iteration. Therefore, it is clear that the sender-initiated approach follows a producer-consumer data flow format while the receiver-initiated approach follows the parent-child format.

3 Analytical Results

3.1 Receiver-Initiated Algorithm

In this approach, given N points and P processors, the data points are partitioned into block of size N/P and each block is assigned to one processor. The total number of threads in the system for the multithreaded (communication/computation) portion of the algorithm is as follows.

Consider a particular point y_i at log(P)th iteration. Initially it sends two threads. These threads at log(P)-1th iteration in turn send two more threads for a total of four threads. This continues for log(P) iterations. This basically forms a binary tree starting from each data point y_i. That is,

$$
\begin{aligned}
2^1 + 2^2 + ... + 2^{log(P)} &= 2^1(2^0 + 2^1 + ... + 2^{log(P-1)}) && (1)\\
&= 2 \times \frac{2^{log(P)} - 1}{2 - 1} && (2)\\
&= 2 \times (2^{log(P)} - 1) && (3)\\
&= 2(P-1)N && (4)\\
&= 2NP - N && (5)\\
& && (6)
\end{aligned}
$$

The number of threads is $2(P-1)N$ and the number of butterfly computation for the log P iterations is $\log P$ iterations is $\log PN$. Note that the number of computations has increased by a factor of 2 compare to $\frac{N}{2}\log P$. However, the advantage is that computation of the FFT for each point can be carried independent of the others points.

In the following, we sketch the proof of correctness for the multithreaded phase of the algorithm. Detailed proof will be available in the full version of the paper.

Proof of Correctness:

Given N points and P processors. There are log(N) iterations in total in the FFT computation. In the algorithm first $\log N - \log P$ iterations are local computations performed by each processor on its local data set and $\log P$ iterations are non-local computations. The resulting locally computed values of the butterfly operation are available at the end of the $\log N - \log P$ iterations. The processors then switch to the multithreaded phase of the algorithm for log(P) stages. In the following, we prove that any data point in the final iteration is correctly computed.

Let $x_i^{log(P)}$ and $x_j^{log(P)}$ reperesent the final output values of two data points x_i and x_j at the end of the log(P)th iteration. To compute the output values by a butterfly operation, requires two data points from $\log P - 1th$ iteration. Assume $x_i^{log(P)}$ and $x_j^{log(P)}$ require the values of $x_i^{log(P)-1}$ and $x_j^{log(P)-1}$ at log(P)-1th iteration to compute $x_i^{log(P)-1}$+w$x_j^{log(P)-1}$ and $x_i^{log(P)-1}$-w$x_j^{log(P)-1}$ respectively. According to the algorithm $x_i^{log(P)}$ and $x_j^{log(P)}$ send two TOKENS to receive $x_i^{log(P)-1}$ and $x_j^{log(P)-1}$ data points (each TOKEN is nothing but a threaded function). So, in fact at $\log P - 1th$ iteration, $x_i^{log(P)-1}$ and $x_j^{log(P)-1}$ have received two *different* TOKENS, one from $x_i^{log(P)}$ and the other from $x_j^{log(P)}$. Notice that these are two different instantiations of of the TOKENS and have no relationship between them. For each TOKEN received, two more instantiations of the TOKENS are sent out for previously computed data values. Therefore, a parent-child relationship is established between the TOKENS. This process continues for $\log P$ iterations at which point, x_l^0 , $l = 1 \cdots N$ values are

readily available. This parent-child relationship for any data point can be shown as a rooted binary tree. We show that the root is correctly computed. This implies that there is no data race problems and therefore no corruption in data between TOKENS that do not have a parent-child relationship and also between iterations of the TOKENS that have this relationship.

Assume that there are N data points, x_l, $l = 1 \cdots N$. In the above scenario, $x_i^{log(P)}$ and $x_j^{log(P)}$ both require $x_i^{log(P)-1}$ and $x_j^{log(P)-1}$. This is true for all data points. In general, at each iteration, one of the points needed to compute the butterfly compuation is its own data point calculated at the previous iteration.

For the proof, let us only consider the TOKENS traveling from $x_i^{log(P)}$ which we will call the root. Consider the TOKEN received at iteration k for the data point x_i^k. As usual the TOKEN sends out two TOKENS, one to x_i^{k-1} and the other to its' mate point for values computed at iteration k-1. A parent-child relationship is now established between the threee TOKENS. The algorithm stores a counter, incremented to 2 to indicate that two TOKENS were sent. This serves as an acknowledgement signal. In the next and further iterations, the data points that received the TOKENS will in turn send out two more TOKENS and increment its counter by 2. Since each TOKEN is a new threaded function invocation at each iteration, the counter value assigned belongs to the corresponding threaded function and therefore it is not shared by TOKENS that are not in a parent-child relationship. By the process of recursion we notice that the TOKENS travel for $\log P$ iterations and the TOKENS traveling from $x_i^{log(P)}$ is a rooted binary tree. The TOKENS received at the end of the $\log Pth$ iteration send the resulting values x_i^0 $(i = 1 \cdots N)$ to the parent TOKEN that requested it and also decrement the corresponding counter by 1. This indicates to the parent TOKEN that an acknowledgement has been received. When the counter value reaches 0, the parent TOKEN performs the butterfly computation. The parent TOKEN in turn sends the resulting values to its parent and decrements the counter by 1. This process continues until the root is reached. At which point the computation stops. Notice that the parent TOKEN will not compute the butterfly operation until it has received two resulting values from its children. The parent-child relationship allows the algorithm to avoid data race conditions and computes the data points correctly.

3.2 Sender-Initiated Approach

In this approach, the number of processors dedicated to the system has no baring to creating the threads. For a given block size, B, and N points there are N/B threads in the system.

- **Case 1:** $B = N$

$N/B = 1$ thread. A sequential algorithm in this case.

- **Case 2:** $B < N$

$N/B = b$ **threads. Given** P **processors,** b/P **threads per processor.**

- $P = b$ **: 1 thread per processor. Therefore, this leads to a very coarse-grained approach.**
- $P > b$ **: 1 thread per processor and only** b **processors will be utilized.**
- $P < b$ **: b/P threads per processors. threads are distributed to processors in a round robin fashion.**

The number of threads in the algorithm can be adopted to the architectural features of the target platfom. In the following we only skecth the proof of correctness.

Proof of Correctness:

We prove that at each iteration, using the split phase trasaction operation, indeed the correct mate values from the *previous* iteration are received and used for the computation before moving to the next iteration and that there are no race conditions.

The sending and receiving of information requires certain amount of synchronization between producing and consuming threads. At each iteration, the mate point changes. Therefore the location of the mate has to be determined during the course of the algorithm. The sender-initiated algorithm is designed such that the resulting computed values are sent to the mate points in bulks since all the mate points are located within the same thread. If the mate points are located within the same processor, then there is no need for synchronization between the producer and consumer threads. Therefore, assume the mate points are located in another processor.

Let T_i **and** T_j **be two different threads located in two different processors,** P_i **and** P_j **respectively. Assume, that the points between these two threads form a butterfly computation at a particular iteration k. This implies that** T_i **and** T_j **are partners. At iteration k, each of the points,** x_l, $(l = 0 \cdots B)$ **in thread** T_i **and** T_j **compute** $x_l w^n$, $(n = 0, 1, \ldots)$. **This computation is performed by the producer thread or fiber within thread** T_m, $(m = i or j)$. **The computed values are now sent to their partner thread. At iteration k-1, the processor and their threads have determined the location of their mate thread and set up the proper synchronization mechanisms. Therefore, at iteration k, the computed values are automatically sent to the location of their mate threads. That is,** T_i **sends the results to** T_j **and vice versa. The data is transferred and the consumer thread of its partner is signaled. This indicates that the data has been received and it is the consumer threads' job to perform the**

butterfly computation. Data transfer and synchronization are combined into one atomic operation (split-phase transaction operation). The signaling operation can be implemented using a 1-bit flag that is allowed to be set by the producer and reset by the consumer. Therefore, even if the threads are ahead by one iteration (which is the most that can happen), the producer thread cannot synchronize its partner consumer thread unless the synchronization flag is reset. The activation of the consumer thread indicates that the next iteration can begin. At this point, the synchronization bit is reset, to synchronize with the next set of mate points.

The setting and resetting of the synchronization allows allows proper coordination between the producer and consumer threads and allows the program to compute without any data race problems.

4 Experimental Framework

EARTH (Efficient Architecture for Running THreads) [Hum96] is a multithreaded program execution model targeted to high-performance of parallel and distributed multiprocessing. The EARTH platform supports latency tolerance by efficient exploitation of fine-grained parallellism available in many applications. In the EARTH programming model, code is divided into threads that are scheduled atomically using dataflow-like synchronization operations [Hum94, Hum96]. These "EARTH operations" comprise a rich set of primitives, including remote loads and stores, synchronization operations, block data transfers, remote function calls and dynamic load balancing. EARTH operations are initiated by the threads themselves. Once a thread is started, it runs to completion, and instructions within it are executed in sequential order.[1] Therefore, a conventional processor can execute a thread efficiently, even when the thread is purely sequential. For this reason, it is possible to obtain single-node performance close to that of a purely sequential implementation, as shown in our earlier work and recapitulated in [Hum96].

Conceptually, each EARTH node consists of an *Execution Unit* (EU), which executes the threads and a *Synchronization Unit* (SU), which performs the EARTH operations requested by the threads. The EARTH multiprocessor structure is shown in Figure(5). The current hardware designs for EARTH use an off-the-shelf high-end RISC processor for the EU and custom hardware for the SU [Maq95a, Maq95b]. However, other implementations are also possible.

In the EARTH programming model, a programmer can express parallelism by utilizing two form of threads: *first-level* and *second-level* threads. First-level threads are declared as threaded functions. When a trreaded function is invoked, a thread is spawned to execute the function. Note that the caller thread will

[1] Instructions may be executed out of order, as on a superscalar machine, as long as the semantics of the sequential ordering are obeyed.

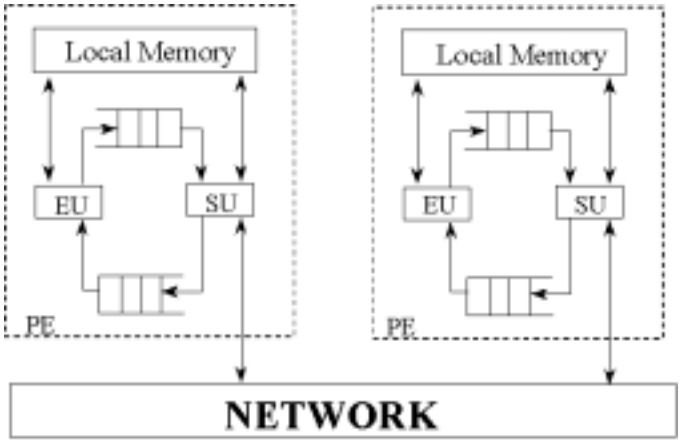


Figure 5: EARTH Multiprocessor

continue its own execution without waiting for the return of the forked threaded function.The body of a function can be further partitioned into *fibers* [The99]. These fibers are referred to as second-level threads. Whenever a user suspects that an operation may incur unpredictable latencies, the user can choose to use an EARTH *split-phase* transaction operation. In a split-phase transaction, data transfer and synchronization are combined into an atomic operation to avoid potential race conditions in the network. A thread need not block until a synchronization signal is received when using this operation. It may execute other instructions. A synchroniztion signal may trigger the spawning of other threads. For example, an user may decide to put the consumer who will need the result of the long latency operation in a different fiber. The producer threaded may synchronize the consumer thread when its data is ready. This ensures that a fiber can be executed in a non-preemtive fashion avoiding any waste of processor resources. The EARTH runtime system will hide the latency by multithreading as long as the program has enough parallelism to generate threads or fibers.

Currently, programs running on EARTH are written in *EARTH Threaded-C*, which extends the C language with multithreading instructions. It is clean and powerful enough to be used as a user-level, explicitly parallel programming language.

The EARTH fine-grain multithreading has the following key features or benefits among others. 1) The programming model with both thread and fiber supports, can be used to express, both naturally and efficiently, both the parallel control/flow dependence at desired level and data dependence in programs. 2) split-phase communication/synchronization operations designed for variable and unpredictable latencies. It supports both network syncronization and memory latencies for both word-wide and block-wide moves. 3) efficient thread-level dynamic load balancing.

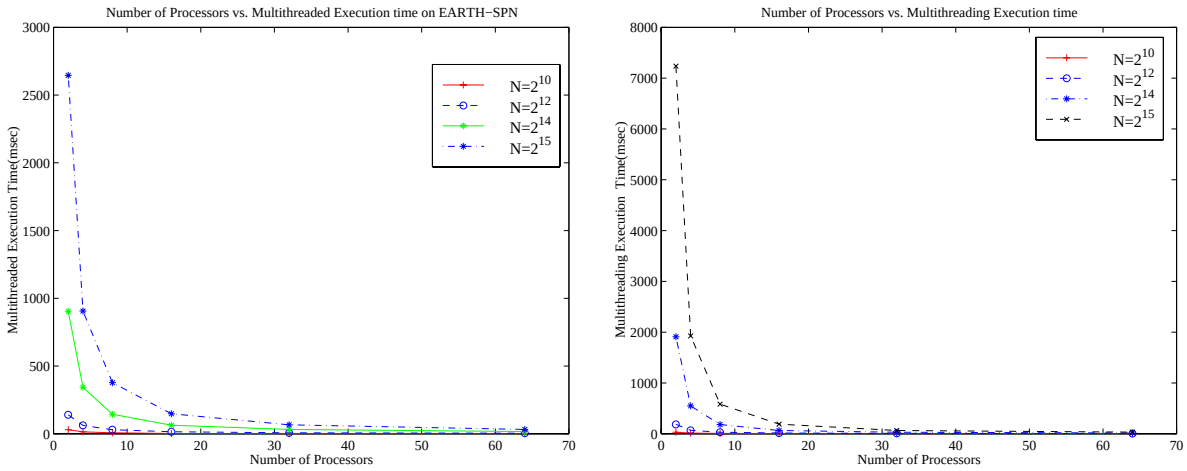
Applications for which a good task distribution can not be determined statically at compile time, EARTH provides an instruction in which the programmer can simply encapsulate a function invocation as a TOKEN. These TOKENs may be executed by any processor that is idle thereby providing automatic load balancing. If the programmer chooses to execute a thread at a predetermined processor, EARTH also provides an instruction in which the programmer can encapsulate a function invocation as a INVOKE. The processor identification number is provided in this instruction. 4) non-preemptive threads (called fibers) with ultra-lightweight context and low thread initiation overhead.

The implementations reported in this paper have been developed using Threaded C and the performance results have been obtained on the EARTH-MANNA platform. MANNA (*Massively parallel Architecture for Numerical and Non-numerical Applications*) is a multiprocessor platform built by GMD-FIRST. Each processing node consists of two Intel i86x XP RISC CPUs (similar to the Intel Paragon), but without the OS "firewall" to facilitate runtime system research and experiments. The nodes operate at 50MHZ clock and each node has 32 MB of dynamic RAM. A 20-node EARTH-MANNA platform is available at the University of Delaware.

5 Performance Results

In this section, we discuss the performance results for the algorithms presented in the previous sections. The algorithms have been implemented in the Threaded-C language on SEMi (Simulator for EARTH, MANNA and i860).

There are two configurations supported by the SEMi simulator. These are called EARTH-MANNA-D and EARTH-MANNA-S configurations. In section 4, we explained that the EARTH EU and SU emulate the two processors of the MANNA machine. This is called the *dual processor* (DUAL) version or EARTH-MANNA-D. But since most multiprocessors have only one CPU per node, we also have a *single processor* (SPN) implementation where only one processor of the MANNA machine emulates both the EU and SU. With only a single CPU to execute both the program code and the multithreading support code, it is necessary to find an efficient way to switch from one to the other. The EARTH operations are therefore replaced by in-line code in the EU to carry out these operations rather than sending the requests to the SU. For some simple operations, doing them in-line in the EU may take less of the EU's time than sending the request to the SU [The99]. We have experimented with both these configurations for both the sender-initiated and receiver-initiated algorithms.



(a) SPN (b) DUAL

Figure 6: Receiver-Initiated Algorithm: Scalability w.r.t machine size for the log(P) stages

5.1 Receiver-Initiated Approach

In the receiver-initiated algorithm, we partition N input points into N/P contiguous points and distribute them to each of the processors. The first $\log N - \log P$ iterations are local computations and the last $\log P$ iterations require remote communication realized as a multithreaded phase in this algorithm. In Figure(6), we show the performance of the the mutithreaded phase of the algorithm only.

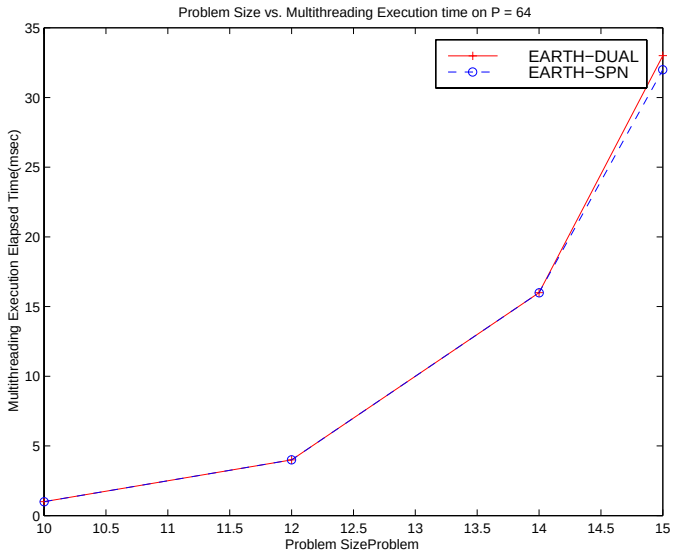
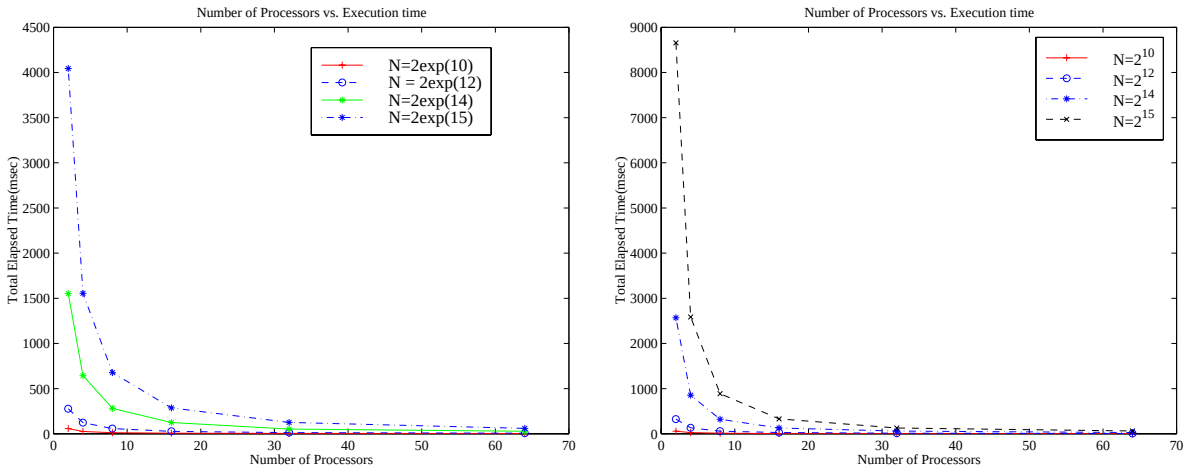


Figure 7: Receiver-Initiated Algorithm: Scalability w.r.t problem size for log(P) stages with P = 64

Note that in Figure(7), on a 64 processor machine using various problem sizes,



(a) SPN (b) DUAL

Figure 8: Receiver-Initiated Algorithm: Scalability w.r.t machine size with varying problem machine size (Total Elapsed Time)

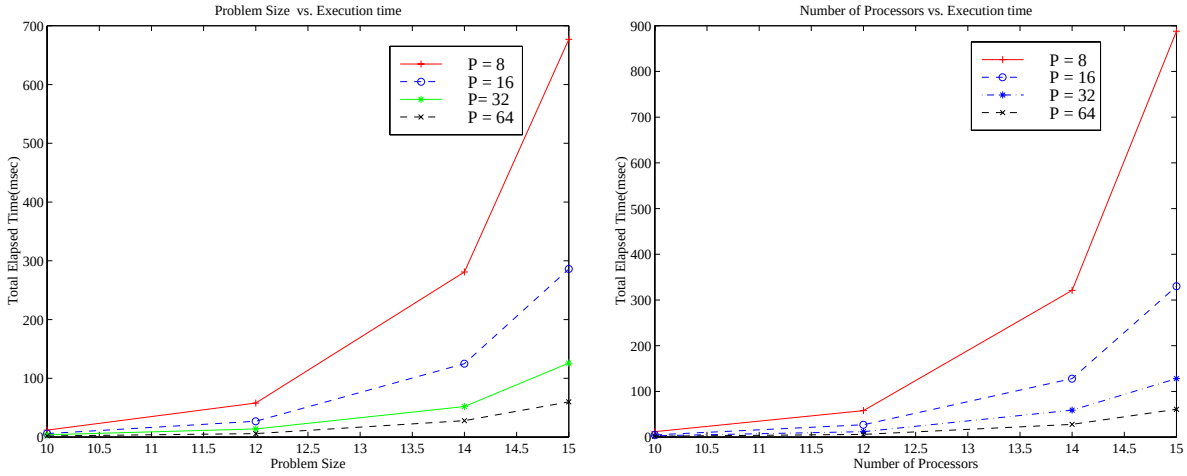
the remote communication in the algorithm does not degrade the performance of the algorithm. For P = 64, N = 2^{15}, the number of threads in the system is $2 * 2^{15} * (64 - 1)$. This is quite a large number of threads in the system. However, we can observe from Figure(7) a near linear speedup. Multithreading has safely guarded against any performance degradation by appropriately overlapping computation with communication. The latencies (synchronization and remote memory access) are efficiently hidden.

Figure(8) shows the scalability results with varying problem size. The total elapsed execution time of the whole FFT algorithm is depicted in these figures on EARTH-SPN and EARTH-DUAL. This includes both log(N)-log(P) local computations and log(P) remote computation/communication (multithreading) portion of the algorithm.

In the receiver-initiated approach $2N(P-1)$ threads are generated. For $N = 2^{15}$, $P = 64$, there are $2 * 63 * 2^{15}$ threads and for P=2, there are $2 * 1 * 2^{15}$ threads. For small values of P the number of threads to be handled is relatively large and that is the reason for the poor perfromance of the algorithm for such small values of P.

Comparing the performance on SPN and DUAL configurations, we observe that if we flood the system with enough parallel threads the performance results of the multithreading implementation as the number of processors increases produces considerable difference in the execution time. One implication is that as long as there are enough parallel threads in the system, the processors are never idle.

The scalability results with respect to varying problem size on the processors are depicted in Figure(9).



(a) SPN (b) DUAL

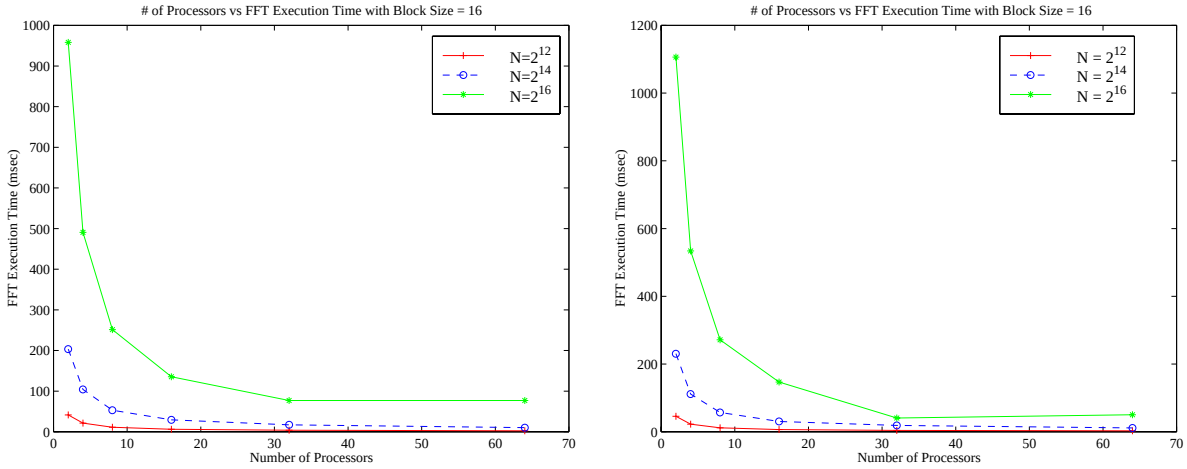
Figure 9: Receiver-Initiated Algorithm: Scalability w.r.t problem size with varying machine size (Total Elapsed Time)

For $logN - \log P$ iterations, each processor performs the local FFT algorithm on its local data set. This is a very sequential algorithm. At the end of this iteration the processors switch to the multithreaded portion of the algorithm for $\log P$ iterations. We see that there is a near-linear speedup for varynig problem size on different processors. And the execution decreases as the number of processors increases. Overall observation is that if there are enough parallel threads and there is a way of overlapping computation with communication then fine-grained multithreading is very effective as seen in the above figures.

We observe that in the above figures, the SPN configuration performs better than the DUAL configuration. In the SPN version there is a single processor that performs both the task of the EU and SU. That is, it handles the network communication/synchronization and computation aspect of the algorithm. However, this does not seem to degrade the performance of the algorithm and also its performance is better than the DUAL configuration which has two processors to perform the tasks of EU and SU. The EU performs all the EARTH operations efficiently without the need to send to the SU like in the dual processor which creates an overhead and wastes CPU time unnecessarily.

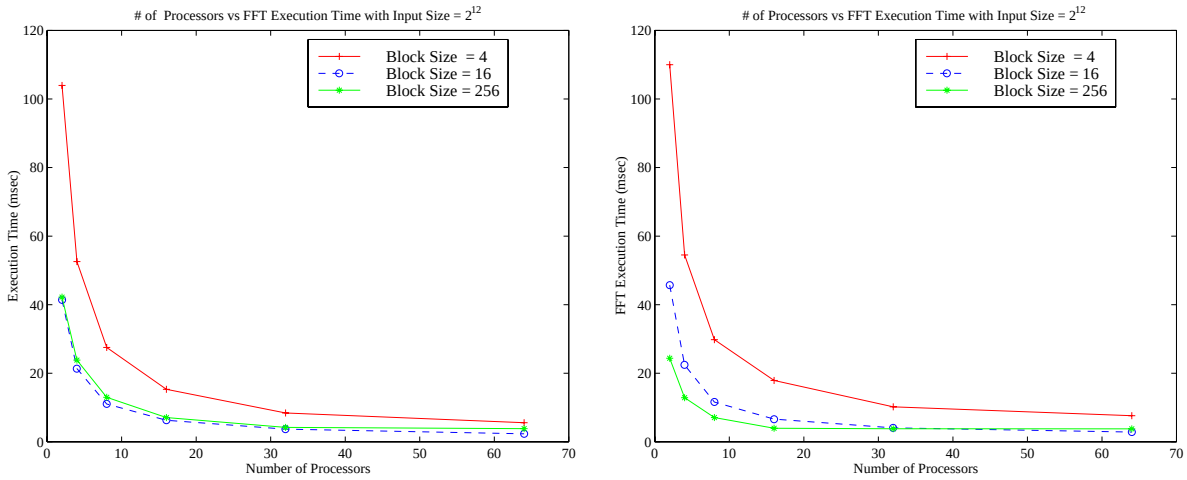
5.2 Sender-Initiated Approach

Figure(10) shows the scalability results as the input problem size increases for both the DUAL and SPN configurations. The number of points per thread is 16 (B = 16). Therefore for N = 2^{12} there are 256 threads and for N = 2^{16}, there are 4096 threads in the system. The EARTH-SPN version performs better than the EARTH-DUAL version for small number of processors, especially. However, for



(a) SPN (b) DUAL

Figure 10: Sender-Initiated Algorithm:Scalability w.r.t to machine size with varying problem size and fixed block size



(a) SPN (b) DUAL

Figure 11: Sender-Initiated Algorithm:Scalability w.r.t to machine size with varying block size and fixed problem size

large number of processors, we observe that the execution time in both cases is very minimal for all problem sizes. We see that the proper overlap of communication and computation has produced better results even with one processor performing both tasks. In the DUAL version, the overhead involved in sending messages to SU by EU creates a bottleneck every time the EU needs to communicate remotely, as mentioned earlier in the receiver-initiated approach. This, therefore, is the reason for poor performance for very small number of processors in the DUAL version as in the case of receiver-initiated approach.

Figure(11) shows the scalability results as the number of points per thread is increased on a fixed size, N = 2^{12}. For B = 256, the number of threads in the system is N/B = 2^{12}/ 2^{8}=16 threads. We observe after 16 processors , there is no change in the execution time. The maximum number of processors that will be kept busy using a round-robin load balancing fashion is 16 since there are only 16 threads in the system. There is not enough parallelism (threads) in the system to balance load on all processors. Beyond 16 processors, the others are idle. This is the reason for the stationary execution time after 16 processors for B = 256. However, for B = 4 (1024 threads) and B = 16 (256 threads), there are enough parallelism to keep all processors busy. Therefore, we see a gradual decrease in the execution time as the number of processors increases. The best result is obtained when there are 16 threads and 16 processors. This leads to a coarse-grained implementation with one thread in each processor. If there is more than one thread in the processor (e.g. 1024/64 = 16 threads/processor), each processor executes a thread to completion before switching to its next thread. There are B points in a thread. So each thread executes the FFT algorithm sequentially on its B points, then uses a split phase transaction to send the produced results to the consumer thread. It is after this split phase transaction operation that the processor switches to the next thread. This is the reason that the execution time for 32 processors on a block size of 4 is slightly more than that of block size 16. If we compare both SPN and DUAL versions, the SPN version does better and the same reasoning as explained previously holds.

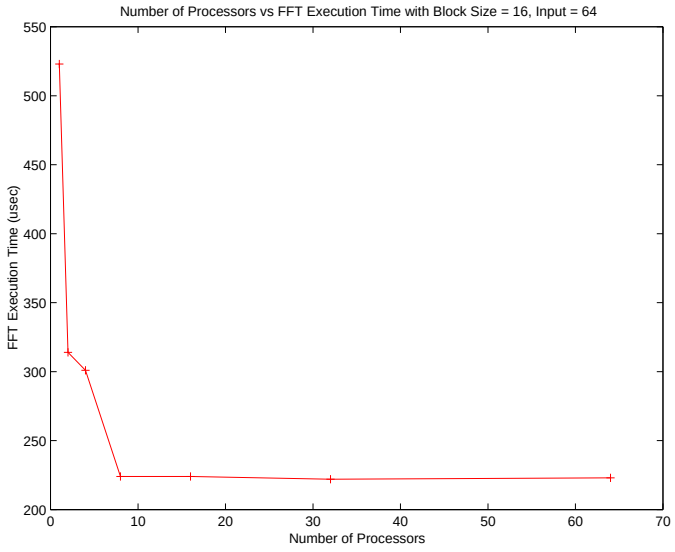


Figure 12: Sender-Iniitaited Algorithm:Scalability w.r.t machine size with block size=16,problem size=64

Figure(12) shows the result for N = 64, B = 16. The number of threads in this case is N/B = 4. For up to 4 processors, there is a linear decrease in the execution

time . But, starting from 8 processors, we see that there is no change in the execution time. The reason behind this is that there are not enough threads in the system for all the processors to be fully utilized. The four threads are distributed in a round-robin fashion to the first four processors. The other processors sit idle. This obviously indicates that one has to choose the appropriate block size to provide enough threads in the system for full load balancing of the processors. Note that in the sender-initiated approach, the number of threads is not directly proportional to the number of threads and therefore poor scalability.

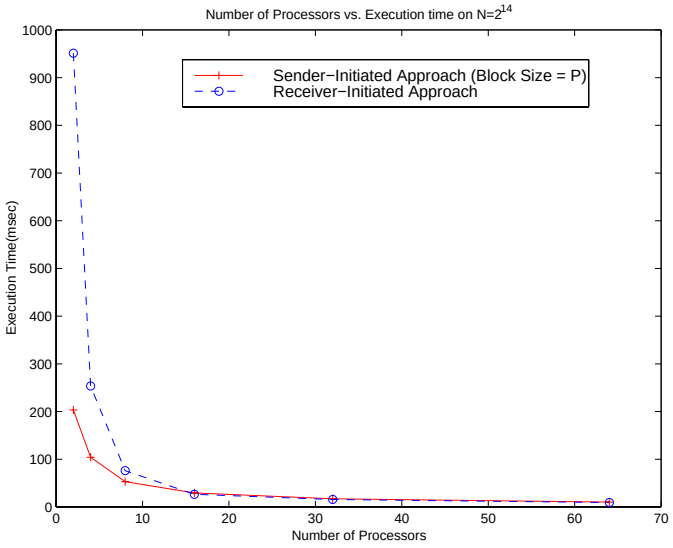


Figure 13: Comparison between the sender-initiated and receiver-initiated approaches

Figure(13) shows the scalability results between the two approaches on an input of size 2^{14}. The comparison is between the total elapsed time in both cases. For the sender-initiated approach, the block size (B) is set equal to the number of processors (P) (i.e., B = P). In the figure, it is clear that for small number of processors, the sender-initiated approach performs better. We examine this as follows: For the sender-initiated approach, the number of threads in the system is N/B = $2^{14}/16$ = 2^{10} regardless of the number of processors. In the receiver-initiated approach, the number of threads is 2(P-1)N. For P =2, there are $2(1)2^{14}=2^{15}$ threads. There are too many threads in the system for only two processors to handle. Therefore, the execution time is higher for the receiver-initiated algorithm. However, for P = 64, there are $2(63)2^{14}$ threads which is a lot more threads than the sender-initiated approach. The execution time, however, is approximately the same for both cases. This is because there is enough parallelism in the system and enough processors to handle the large number of threads, in the receiver initiated approach. To summarize, multithreading is very effective if there are enough parallel threads and enough processors to handle the load generated in the system. The absolute speedup of the algorithms for 64 processors is approximately 70%.

6 Conclusions

In this paper, we have presented two multithreaded algorithms for the FFT problem: receiver-initiated and sender-initiated. In the receiver-initiated approach the multithreaded version of the algorithm due to its fine-grain communication/computation ratio produced superb results for large number of processors. This algorithm extracts full parallelism in the FFT computation while allowing redundancy. We saw a near linear speedup as the number of processors increases, even when there are large number of threads in the system. In the sender-initiated approach the number of threads in the system is fixed at runtime and can be independent of the number of processors. We observed that the best result is obtained when there is one thread per processor which produces a coarse-grained implementation. Our implementation results show that receiver initiated algorithm performs poorly when number of processors is small. However, for large number of processors, both the algorithms perform well, yielding execution times of only 10 msec for an input of 16 K data points on a 64 processor machine, assuming each processor running at 140 MHz clock speed. Overall, the sender initiated algorithms gave the best performance for smaller machine sizes, while for large machine sizes both the algorithms performed equally well.

References

[Akl89] Akl S.G. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Ang90] Angelopoulos G. and Pitas I. Parallel implementation of 2-d fft algorithms on a hypercube. In *Proc. Parallel Computing Action, Workshop ISPRA*, Dec. 1990.

[Ang92] Angelopoulos G., Ligdas P. and Pitas I. Two-dimensional fft algorithms on parallel machines. In *Transputing for Numerical and Neural Network Application, G.I. Reijns, editor*, IOS Press, 1992.

[Bur92] Burkhardt H., Lang. B., and Noelle M. Aspects of parallel image processing algorithms and architectures. In *In H. Burkhardt, Y. Neuvo and J.C. Simon, editors, From Poxels to Features II*, pages 65–84, 1992.

[Cho93] Cho-Chin Lin, V.K. Prasanna, and A.A Khokhar. Scalable parallel extraction of linear features on mp-2. In *Workshop on Computer Architectures for Machine Perception*, pages 352–361, New Orleans, Louisiana, 1993. IEEE Computer Society Press.

[Coc67] Cochran W.T and Cooley J.W et.al. What is the fast Fourier transform? *IEEE Transactions on Audio and Electroacoustics*, 15:45–55, 1967.

[Coo77] Cooley J.W. and Lewis P.A. and Welch P.D. *The Fast Fourier transform and its application to time series analysis*. Wiley, New York, 1977. In statistical Methods for Digital Computers.

[Cor90] Cormen T.H, Leiserson C.L. and Rivest R.L. *Introduction to Algorithms*. The MIT Press, 1990.

[Cyp89] Cypher R. and Sanz J.L.C. SIMD architectures and algorithms for image processing and computer vision. In *IEEE Trans. Acoustics, Speech and Signal Processing*, volume 37(12), pages 2158–2174, Dec. 1989.

[Fri99] Frigo M. and Steven. Fftw. In *http://theory.lcs.mit.edu/ fftw*, 1999.

[Gen66] Gentleman W.M and Sande G. Fast Fourier transforms for fun and profit. In *Proc. 1966 Fall Joint Computer Conference AFIPS 29*, pages 563–578, 1966.

[Gol65] Golay M.J.E. Apparatus for counting bi-nucleate lymphocytes in blood. In *US Patent, 3214574*, 1965.

[Hen96] Hennesey J.L. and Patterson D.A. *Computer Architecture: A quantitative Approach, Second Edition*. Morgan Kaufmann,Inc., San Francisco,CA, 1996.

[Hum94] Hum H.H.J, Maquelin O., Theobald K.B., Tian X. and Gao G.R. Building multithreaded architectures with off-the-shelf microporcessors. In *Proc. of the 8th Intl. Parallel Processing Symp.*, pages 288–294, Cancún, Mexico, Apr. 1994. IEEE Comp. Soc.

[Hum96] Hum H.H.J. et. al. A study of the earth-manna multithreaded system. In *Intl. J. of Parallel Programming*, volume 24(4), pages 319–347, Aug. 1996.

[Hwa93] Hwang K. . *Advanced Computer Architecture: Parallelism,Scalability, Programmability*. McGraw-Hill,Inc., New York,NY, 1993.

[Jam93] Jamieson L.H, Delp E.J et.al. A library based program development environment for parallel image processing. In *Scalable Parallel Library Conference*, pages 187–194, Mississippi State University, Mississippi, 1993.

[Kam87] Kamin R.A. and Adams G.B. Fast fourier transform algorithm design and tradeoffs on the cm-2. In *Proc. Workshop Comput. Arch. Pat. Anal. Mach. Intell.*, pages 184–191, Oct. 1987.

[Kum94] Kumar V. and Grama A. et. al. *Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Company, 1994.

[Lei92] Leighton F.T. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann, San Mateo, California, 1992.

[Lei99] Leiserson C. Cilk. In *http://supertech.lcs.mit.edu/cilk*, 1999.

[Loa92] Loan C.L. Computational frameworks for the fast fourier transform. *SIAM Journal, Frontiers in Applied Mathematics*, 1992.

[Maq95a] Maquelin O. Load balancing and resource management in the adam machine. In *Advanced Topics in Dataflow Computing and Multithreading (G. R. Gao , Lubomir Bic, and Jean-Luc Gaudiot, eds)*, pages 307–323. IEEE Comp. Sci. Press, 1995.

[Maq95b] Maquelin O. et. al. Costs and benefits of multithreading with off-the-shelf risc processors. In *Proc. of the First Intl. EURO-PAR Conf.*, pages 117–128, Stockholm, Sweden, Aug. 1995. Springer-Verlag.

[Opp83] Oppenheim A.V. and Willsky A.S. *Signals and Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1983.

[Pea68] Pease M.C. An adaptation of the fast Fourier transform for parallel processing. *Journal of the ACM*, 15:252–264, 1968.

[Pit93] Pitas I. *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks.* John Wiley and Sons, New York, NY, 1993.

[Pra93] Prasanna V.K, Cho-Li Wang and Khokhar A.A. Low level vision processing on connection machine cm-5. In *Workshop on Computer Architectures for Machine Perception*, pages 117–126, New Orleans, Louisiana, 1993. IEEE Computer Society Press.

[Soh97] Sohn A., Kodama Y., et.al. Fine-Grain Multithreading with the EM-X. In *Ninth ACM Symposium on Parallel Algorithms and Architectures*, pages 189–198, Newport, Rhode Island, June 1997.

[Sto71] Stone H.S. Parallel processing with the perfect shuffle. In *IEEE Trans. Computers, C-20*, pages 153–161, 1971.

[Sto80] Stone H.S, editor. *Introduction to Computer Architecture.* Science Research Associates, Chicago, 1980.

[The99] Theobald K.B. *EARTH: An Efficient Architecture for Running Threads*, 1999.

[Tho83] Thompson C.D. Fourier transforms in VLSI. *IEEE Transactions on Computers*, 32:1047–1057, 1983.

[Ung58] **Unger H. A computer oriented toward special problems. In *Proc. IRE*, volume 46, pages 1744–1750, 1958.**