Multithreaded Algorithms for the Fast Fourier Transform

Parimala Thulasiraman* Kevin B. Theobald* Ashfaq A. Khokhar* Guang R. Gao*

ABSTRACT

In this paper we present fine-grained multithreaded algorithms and implementations for the Fast Fourier Transform (FFT) problem. The FFT problem has been formulated using two distinct approaches based on the dataflow concepts. The first approach, referred to as the receiver-initiated algorithm, realizes the FFT iterations as a parent-child relationship while fully exploiting the underlying parallelism. The second approach, referred to as 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. The implementations of the proposed algorithms have been carried out on the EARTH (Efficient Architecture for Running THreads) platform. For both the algorithms, we analyze the ratio of remote vs local threads and study its impact on the experimental results. Our implementation results show that for certain block sizes on fixed problem size and machine size, the receiver-initiated approach performs better than the sender-initiated approach. 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.

Categories and Subject Descriptors

F.2 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*complexity measures, performance measures*

General Terms

Fine-Grained, Multithreading, Dataflow Architecture, Parallel Algorithms, Non-Preemptive

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

1. INTRODUCTION

Traditionally, digital image/signal processing algorithms are computationally intensive due to 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 transform operations such as FFT, DCT, or Wavelet, in real time on such large data sets requires high performance computing [20, 19, 11]. The Fast Fourier Transform (FFT) [5] has been studied extensively as a frequency analysis tool in diverse application areas such as audio, signal, and image processing [18], and several other real time data applications [7, 16]. In general the FFT based frequency analysis of a multidimensional data set can be realized by performing 1D-FFT alternately on each dimension of the data interleaved with data transpose steps. The 1-D FFT on an input of N data points requires (N/2) log_2(N/2) complex multiplication operations which takes most of the computation time for large data sets. The FFT problem has been studied on various parallel machines [13]. It can be well parallelized using shuffle exchange network [2, 23]. Other parallel implementations have been performed on linear arrays [25], hypercubes [1, 2], and mesh architectures [12, 11]. Two types of latencies are normally embedded in parallel implementations: communication latency (due to remote accesses) and synchronization latency (due to data dependencies) [10]. Conventional message passing MPPs do not yield high performance if such latencies are frequent in the parallel solutions employed. Several techniques at the software and hardware levels (such as superscalar, superpipelined, VLIW, prefetching) [8] have been used to hide or tolerate both communication and synchronization 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. This paper investigates multithreaded algorithms and implementations for Fast Fourier Transform (FFT) on fine-grained multithreaded computing paradigms. We define a fine-grained multithreaded paradigm as the one that has abundant number of threads and the cost of switching between threads is minimal [24]. The imbalance in computation and communication in a parallel FFT algorithm and the global nature

of the embedded communication patterns makes it an ideal candidate for multithreaded platforms.

Sohn et. al. [22] have 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 [6] 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 [15].

In this paper, we study the FFT problem on non-preemptive multithreaded architectures. In a non-preemptive model a thread once started runs to completion. Assuming this model of computation, 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 algorithm a parent-child relationship is established between threads, while fully exploiting the underlying parallelism in the FFT problem. The number of threads scales linearly with the product of input size and the number of processors. The second algorithm, referred to as the *sender-initiated* algorithm is a coarse-grained algorithm, where the number of threads can be set based on the architectural characteristics of the target platform. This algorithm models the FFT problem as a producer-consumer problem and can be adopted to different architectural parameters for achieving high performance.

We have used the EARTH- *Efficient Architecture for Running THreads* [9, 24] platform for our implementations, which is a fine-grained, non-preemptive, dataflow architecture. We present analytical and experimental results for both the algorithms. Our implementation show that for certain block sizes on fixed problem size and machine size, the receiver-initiated approach performs better than the sender-initiated approach. 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. For reference purposes, we have also implemented the best known sequential algorithm for the FFT on a single node MANNA. The algorithm takes 866ms for performing FFT on 2^{16} data points on an i860 processor [24].

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.

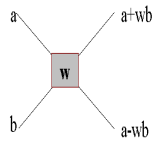

Figure 1: The Butterfly Operation

2. FINE-GRAINED MULTITHREADED 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 [13, 14]. In the following we present two dataflow style multithreaded algorithms for the FFT computation based on the iterative solution. These algorithms differ from each other in terms of dataflow style and in number and sizes (weights) of the threads employed.

2.1 Receiver-Initiated Algorithm

The *receiver-initiated* algorithm is a fine-grain multithreaded algorithm based on the Cooley-Tukey style [4] of the FFT signal flow graph.

Let us assume we have N (N=2^m) data elements and P (P=2^p) processors. A *butterfly* computation (Figure(1)) is performed on each of the data points in every iteration and there are altogether $\log N$ iterations. The butterfly computation can be conceptually 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. In each iteration, there are N/2 summations and N/2 differences.

In general, a parallel algorithm for FFT, with blocked data distribution of N elements on P processors, involves communication for $\log P$ iterations and terminates after $\log N$ iterations. If we assume shuffled input data at the beginning, the Cooley-Tukey Style (Figure 2), the first $\log N - \log P$ iterations require no communication. Therefore, during the first $\log N - \log P$ iterations, a sequential FFT algorithm can be used inside each processor. At the end of the $(\log N - \log P)th$ iteration, the latest computed values for $\frac{N}{P}$ data points exist in each processor. The receiver-initiated algorithm consists of two phases, the sequential phase and the multithreaded phase. The mulithreaded phase of the algorithms starts at the end of the $\log N - \log P$ iterations.

Conceptually, the multithreaded phase starts from the final output. Consider 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)). For the multithreading phase, the algorithm works on only $\frac{N}{2}$ output data points. The remaining $\frac{N}{2}$ data points can be generated with just one additional local butterfly operation for each point. Therefore, given P processors, each processor computes FFT for only $\frac{N}{2P}$ data points at the final output.

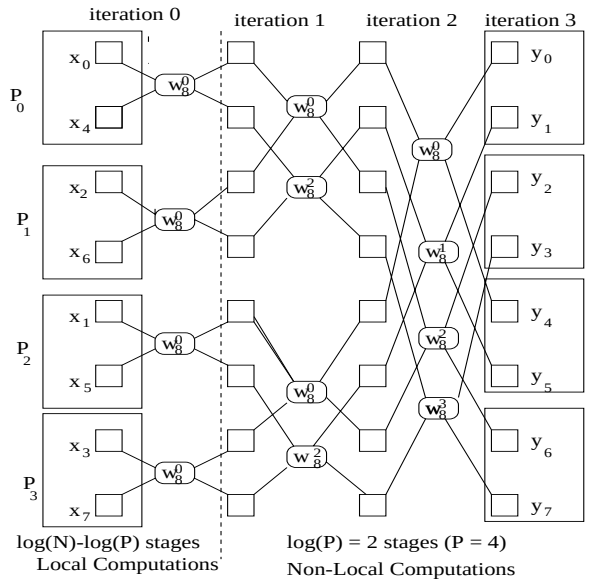

Figure 2: The Cooley-Tukey signal flow graph (N=8)

Considering the dataflow style, each processor, for each of its $\frac{N}{2P}$ 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 number, 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.

Note that the butterfly computation is performed only after the two data values have 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. Since the multithreaded algorithm is performed on only $\frac{N}{2}$ points, each processor contains $\frac{N}{2P}$ output data points. That is for this example, P_0 has y_0, P_1: y_4, P_2: y_2 and P_3: y_6. 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_0 at $\log N$-th iteration, that is at iteration 3 (refer to Figure(2)). The receiver initiated approach starts from y_0 and proceeds backwards for $\log P = 2$ iterations.

P_0 which holds y_0 sends out two requests using two separate threads: one to its mate processor P_2 and the other to itself for the computed data elements x_1 and x_0 at iteration 2, respectively. Of course, at iteration 2, x_0 and x_1 have not yet been computed. Therefore, consider the actions of processor P_2 at iteration 2. This involves: Processor P_2 upon receiving and executing the thread from P_0 sends out two more threads to P_2 and P_3 for data elements at iteration 1. At iteration 1, the latest locally computed data values exist and P_2 and P_3 transfer x_1 $(x_1 + w_8^0 x_5)$ and x_3 $(x_3 + w_8^0 x_7)$ respectively to processor P_2 which requested these data at iteration 2. At this point P_2 computes the butterfly operation and sends the result back to P_0 which requested it at iteration 3. Now P_0 has received one data element. Similarly the same type of communication is performed to receive the second data element. When the two data points have arrived at P_0 the butterfly operation is performed and y_0 is computed. Now the processor P_0 holding y_0 computes its mate's value y_4 at the last iteration.

In the above 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 $\frac{N}{2P}$ data points per processor, $2^i \frac{N}{2P}$ threads are sent out, where $i = 1 \cdots \log P$. The processors execute the butterfly computation for each related pair of threads as per the arrival of data points 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 communication. Note that in this algorithm, butterfly computations over same data elements are computed in different processors, giving rise to redundant computation load. However, the algorithm can be easily adapted to varying degrees of parallelism and synchronization overheads. The analytical section explains the complexity analysis in detail.

2.2 Sender-Initiated Approach

The *sender-initiated* algorithm is based on the Gentleman-Sande [7] signal flow graph for the FFT problem. 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 B data points.

The Gentleman-Sande signal flow graph (Figure 3) can be viewed as each data point requiring a mate data point for performing the FFT computation. The mate may be located in a different thread in a different processor depending on the FFT iteration. In this case, a thread (called the consumer thread) for each of its points sends the recently computed value to the thread (called the producer thread) containing the mate points. The sending and receiving of information requires certain amount of synchronization between the producing and consuming threads to be set up apriori. Also, the

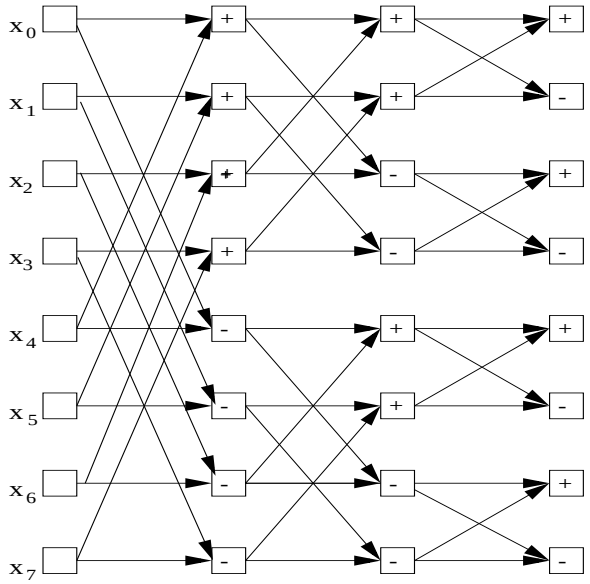

Figure 3: Gentleman-Sande Signal Flow Graph

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(s).

As mentioned above, each thread consumes data from the previous iteration and produces data for the 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 3) (We have represented the signal flow graph differently [7] for easier explanation of the sender-initiated algorithm). Assume N=8, P=4 and B=2. Then there are $\frac{N}{B} = 4$ 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 3, 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.

3. ANALYTICAL RESULTS

3.1 Receiver-Initiated Algorithm

In the Receiver-Initiated algorithm, given N points and P processors, the data points are partitioned into block of size $\frac{N}{P}$ and each block is assigned to one processor. This algorithm can be implemented incorporating varying degrees of parallelism depending on the target architecture and the number of processors. In the following, we analyze this algorithm under two such scenarios.

In the first scenario, we exploit full parallelism by generating maximum number of threads at the expense of redundancy in the butterfly computations at different processing nodes. In the second scenario, we limit the number of threads generated to completely avoid redundant computations at the expense of synchronization overheads between different processors. The implementation results are reported only for the first scenario. In both the scenarios, the ratio of local versus remote threads are studied. The performance of the algorithm lies in the balance between the number of remote and local threads and on their overlapped asynchronous scheduling.

3.1.1 Scenario 1:

Consider a particular point y_i at $\log P$-th iteration during the multithreaded phase of the algorithm. Initially, it sends out two requests in the form of two threads. These threads at $\log P - 1$-th iteration in turn send two more threads for a total of four threads. This process continues over $\log P$ iterations. The thread generation process in this manner can be viewed as a binary tree of height P starting from each of the data point y_i in the final iteration. The internal nodes of such a tree correspond to the threads performing butterfly computations and the arcs correspond to the threads gathering the data. Thus the number of threads performing the (remote) communication is $2(P-1)$ and the number of local threads performing the local computations is $(P-1)$. There are $N/2$ such binary trees of parent-child threads, one corresponding to each even indexed data point in the final output column. The odd indexed data will be automatically computed due to the butterfly computation in the final iteration. Therefore, total number of threads is $N(P-1)$ and the number of butterfly computation for the final $\log P$ iterations is $\frac{N}{2}(P-1)$.

Considering the first $\log N - \log P$ iterations as well, the processors perform local butterfly computations on $\frac{N}{P}$ points over these iterations. This can be realized as a sequential FFT algorithm over $\frac{N}{P}$ data points using a single thread in each processor. Therefore, there are P local threads per iteration for $\log N - \log P$ iterations. However, each thread in this case is performing $\frac{N}{2P} \log \frac{N}{P}$ butterfly computations. For the next $\log P$ iterations, the multithreaded algorithm is performed.

Summarizing, the total number of local threads is $P+\frac{N}{2}(P-1)$ and the total number of remote threads is $N(P-1)$. The ratio of local threads versus remote threads is:

$$\frac{\frac{N}{2}(P-1)+P}{N(P-1)} = O(1)$$

The total number of of butterfly computations:

$$(P(\frac{N}{2P}\log\frac{N}{P}) + \frac{N}{2}(P-1) = O(NP), \text{for} P > \log N$$

3.1.2 Scenario 2:

In the above analysis, each of the $\frac{N}{2}$ points requests data from its mate processors by means of threads and these requests yield a binary tree pattern. However, the $\frac{N}{2}$ binary trees created are not necessarily *unique*, thereby, duplicating work. The algorithm can be implemented by realizing only unique trees, decreasing the amount of computation and avoiding to send unnecessary number of threads. For example, if y_i and y_j follow the same path, then y_i performs the butterfly computations and communicates the computed value to y_j eliminating y_j from performing the same computations as y_i. This can be realized as follows: Initiate $2 * \frac{N}{2}$ threads at the Nth iteration, each collecting a unique data point generated at the N-1th iteration. Next, initiate only $4*\frac{N}{4}$ out at N-1th iteration. This process continues for $\log P$ iterations, at which point, $2^{\log P} * \frac{N}{2^{\log P}}$ threads are sent out. Therefore, in the multithreaded phase of the algorithm, the total number of threads involving communication is $N \log P$ and the number of butterfly computations is $\frac{N}{2}\log P$.

Summarizing, the total number of local threads including the sequential phase is $P + \frac{N}{2}\log P$, and the ratio of local to remote threads:

$$\frac{(P+\frac{N}{2}\log P)}{N\log P} = O(1).$$

The total number of butterfly computations:

$$(P(\frac{N}{2P}\log\frac{N}{P}) + \frac{N}{2}\log P = O(N\log N)$$

3.2 Sender-Initiated Approach

In the Sender-Initiated approach, the algorithm can be formulated as if the number of processors in the system has no bearing to creating the threads. For a given block size B and N data points, there are $\frac{N}{B}$ threads in the system.

- Case 1: $B = N$
$\frac{N}{B} = 1$ thread. A sequential algorithm in this case.

- Case 2: $B < N$
$\frac{N}{B} = b$ threads. Given P processors, $\frac{b}{P}$ threads per processor.

  - $P = b$: 1 thread per processor, this leads to a very coarse-grained approach.
  - $P < b$: $\frac{b}{P}$ threads per processors, threads are distributed to processors in a round robin fashion.
  - $P > b$: not an interesting case.

The number of threads in the algorithm can be adopted to the architectural features of the target platform.

Each thread consists of producer and consumer fibers (microthreads). However, in the analysis we will treat the fibers and threads as the same. In the multithreading phase of the algorithm, the first $\log P$ iterations, the number of remote threads is $N/B \log P$ when $P \leq N/B$. The number of local threads is $N \log P$ during the multithreaded phase. However, there is an additional cost of $N \log P$ over $\log P$ iterations for setting up the N synchronization slots at each iteration. The ratio of local to remote threads is :

$$\frac{(P+N\log P)}{N/B\log P}$$

When $\frac{N}{B} = P$, this yields a very coarse grained algorithm with local to remote threads ratio of $O(\frac{N}{P})$. When B is a small constant, the algorithm is highly fine-grained with local to remote threads ratio of $O(1)$. The number of butterfly computations in all the cases is $\frac{N}{2}\log N$.

Note that in terms of the complexity analysis, the sender initiated approach is comparable to the second scenario of the receiver-initiated algorithm.

4. EXPERIMENTAL FRAMEWORK

In the following, we briefly describe the EARTH model and platform that have been used in our experiments.

EARTH (Efficient Architecture for Running THreads) [9] 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 parallelism available in many applications. In the EARTH programming model, code is divided into threads that are scheduled atomically using dataflow-like synchronization operations [9].

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 current hardware designs for EARTH use an off-the-shelf high-end RISC processor for the EU and custom hardware for the SU [17]. 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 threaded function is invoked, a thread is spawned to execute the function. Note that the caller thread will 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* [24]. 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 synchronization 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 thread may synchronize the consumer thread when its data is ready. This ensures that a fiber can be executed in a non-preemptive 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 are written in *T*hreaded-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 programming model has been realized on a MANNA platform. MANNA (*Massively parallel Architecture for Numerical and Nonnumerical 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.

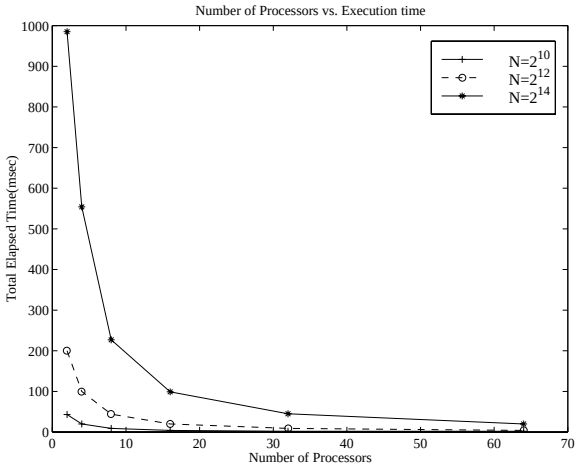

Figure 4: Receiver-Initiated Algorithm: Scalability w.r.t machine size with varying problem size on EARTH-SPN(Total Elapsed Time)

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 the simulator for EARTH- MANNA called SEMi.

There are two configurations supported by the SEMi simulator: 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 [24]. We have experimented with both these configurations for both the sender-initiated and receiver-initiated algorithms. For reference purposes, we have also implemented the best known sequential algorithm for the FFT on a single node MANNA. The algorithm takes 866ms for performing FFT on 2^{16} data points on an i860 processor [24].

5.1 Receiver-Initiated Approach

In the receiver-initiated algorithm, we partition the N output points into $\frac{N}{P}$ 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.

Figures(4,5) show the performance results with varying problem size. The total execution time of the entire FFT algorithm is depicted in these figures on EARTH-SPN and EARTH-DUAL. This includes both log(N)-log(P) iterations of local computation and log(P) iterations involving remote computation/communication (multithreading). Notice that for small number of processors the execution time is steep but as the machine size increases the performance is significantly improved. For small values of P the number of threads to be handled is relatively large and that is the reason for higher execution times in such cases. For example, if $N = 2^{14}$, and $P = 64$, each processor contains 2^8 data points per processor ($\frac{N}{P}$ data points). The number of threads generated in the system is $N(P-1)$ which is $(63) * 2^{14}$. Since at each iteration a processor requires a mate to compute its butterfly computation, each processor sends out a thread to its mate processor requesting data. Each processor is either busy sending a thread requesting data or busy handling the request. This is performed by each processor at every iteration. Therefore, the processors load is equally balanced and performing either one of the tasks, eliminating the need to be idle at any point in time. The algorithm has overlapped computation with communication appropriately, thereby, producing a near-linear speedup.

Comparing the performance on SPN and DUAL configurations, we observe that if we flood the system with enough parallel threads the performance of the multithreading implementation is improved significantly as the number of processors is increased. One implication is that as long as there are enough parallel threads in the system, the processors are never idle.

The performance results with respect to varying machine size on the processors are depicted in Figures(6,7). As the figures show, with the increase in the problem size the execution time decrease for varying processors size. The relative speedup is about 50% on 64 processors.

We observe that in the above figures, the SPN configuration performs better than the DUAL configuration. In the

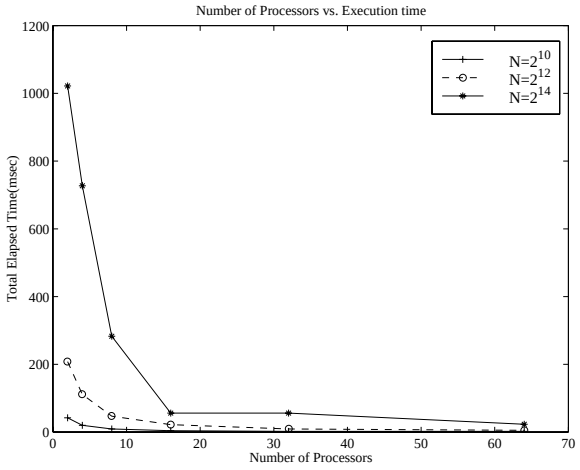

Figure 5: Receiver-Initiated Algorithm: Scalability w.r.t machine size with varying problem size on EARTH-DUAL(Total Elapsed Time)

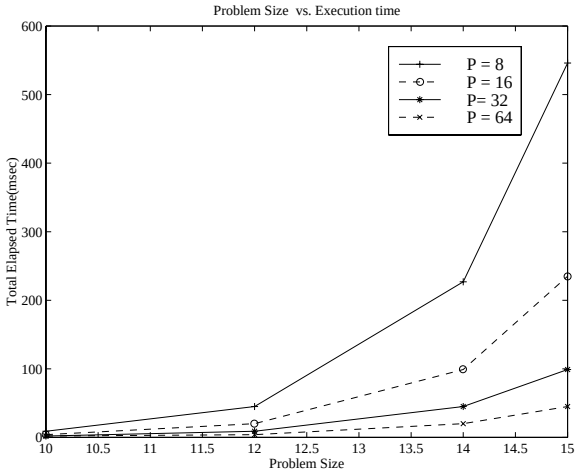

Figure 6: Receiver-Initiated Algorithm: Scalability w.r.t problem size with varying machine size on EARTH-SPN(Total Elapsed Time)

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. In SPN, 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

Figures(8,9) show 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 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.

Figures(10,11) show 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 $\frac{N}{B} = 16$ 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 is 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.

We have noticed poor scalability in the sender-initiated approach for a fixed block size on different machine sizes. The number of threads is proportional to the number of blocks and is independent of the number of processors. 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.

Figures(12,13) compare the performance results of the two approaches (receiver versus sender initiated) on an input of size 2^{14} with two different block sizes for the sender-initiated method. The comparison is between the total elapsed time in both cases. Note that for a block size of 1, the receiver-initiated approach performs slightly better than the sender-initiated approach. The reason behind this is that for N = 2^{14} there are $\frac{N}{B} = 2^{14}$ threads generated. Each data point

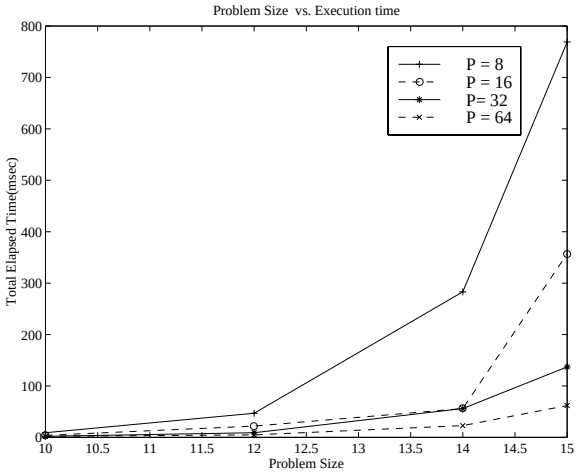

Figure 7: Receiver-Initiated Algorithm: Scalability w.r.t problem size with varying machine size on EARTH-DUAL(Total Elapsed Time)

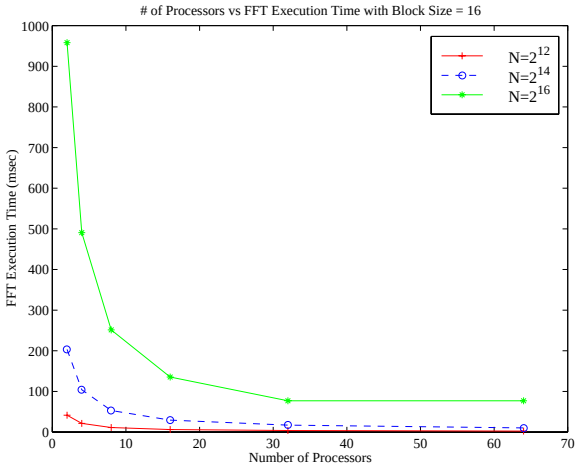

Figure 8: Sender-Initiated Algorithm:Scalability w.r.t to machine size with varying problem size and fixed block size on EARTH-SPN

is now a thread. These threads are distributed in a round robin fashion to each of the processors. For P = 64, each processor contains, 2^8 threads. The synchronization primitives between these threads have to be set up dynamically at runtime. At each iteration, there are 64 synchronization primitives that need to be coordinated. This is very time consuming. Thus, for large number of processors, the receiver-initiated approach performs better. When P = 2, the number of threads per processor in the sender-initiated approach is $\frac{2^{14}}{2} = 2^{13}$. For the receiver initiated approach there are $N(P-1)$ threads (2^{14}). However, as the figure shows the execution time in both approaches for $P = 2$ is approximately the same. In the sender-initiated approach, due to the number of synchronization slots that needs to be set up between the two processors for the threads at each iteration during runtime creates performance degradation. And for the receiver-initiated approach there are too few processors to handle the huge number of threads.

However when $B = 2$, the number of threads is 2^{13} for the sender-initiated approach. There are now two points per thread. Again for $P = 64$ processors there are 2^7 threads per processor with two points per thread. The butterfly computation within the points in the thread is sequential. Note that in Figure(13), for small number of processors, the synchronization slot assignment does not affect the performance of the algorithm. This is mainly because one synchronization slot is set up for both the two points in a thread. In general this is true when N, P and B are powers of 2. The algorithm works such that the mate points for a particular thread reside in the same thread of its mate processor. It is therefore not necessary to set up a sync slot for each of the two points in the thread, but rather assign one sync slot per thread. This greatly reduces the synchronization slot compute time. For only two processors the synchronization is between the two processors only. However for 64 processors, the synchronization mechanism is between all 64 processors which changes dynamically at runtime. So, each processor needs to set up 2^7 synchronization slots at runtime. This is again time consuming. Thus, the execution time is no better than the receiver-initiated approach for large number of processors. For $P = 2$, there are 2^6 threads per processor compared to 2^{14} threads in the receiver-initiated method. Therefore, the sender-initiated method performs better.

As the block size increases the number of points per thread also increases. The computation within a thread is sequential. Since there is one synchronization slot assignment per thread the set up greatly reduces as the block size reduces. Also, increasing the block size makes the problem coarse-grained in the sender-initiated approach.

In conclusion, it is safe to say that for a given problem size and machine size, the receiver-initiated method performs better than the sender-initiated approach for smaller block sizes. In the sender initiated method, the block size have to be properly defined to get good performance. The more coarse grained the problem gets, the better the results are. For large number of processors, the receiver-initiated method has always performed either equally or better.

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. We achieve 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 showed that for certain block sizes on fixed problem size and machine size, the receiver-initiated approach performed

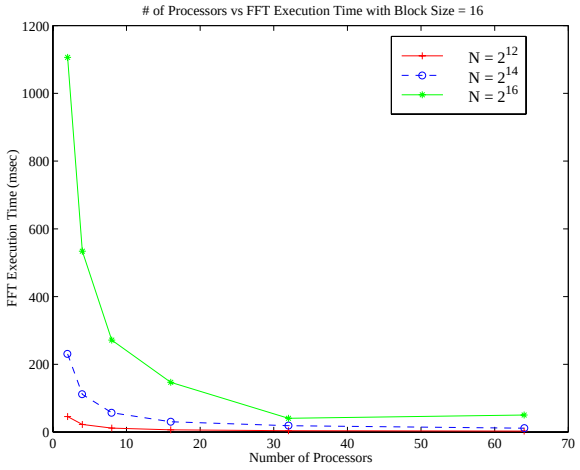

Figure 9: Sender-Initiated Algorithm:Scalability w.r.t to machine size with varying problem size and fixed block size on EARTH-DUAL

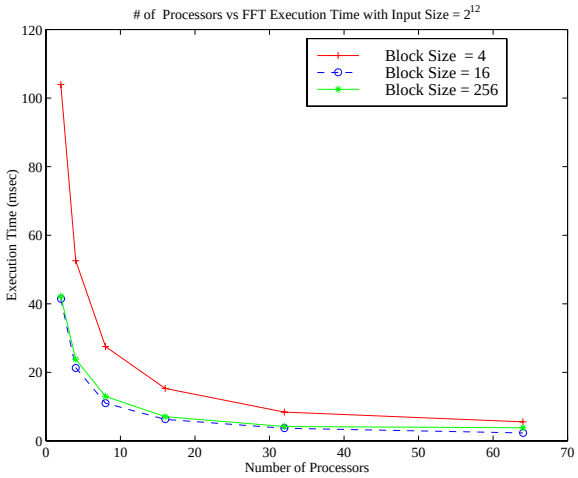

Figure 10: Sender-Initiated Algorithm:Scalability w.r.t to machine size with varying block size and fixed problem size on EARTH-SPN

better than the sender-initiated approach. 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 algorithm gave the best performance for smaller machine sizes and certain block sizes, while for large machine sizes both the algorithms performed equally well.

7. REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

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

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

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

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

[17] Maquelin O. et. al. Costs and benefits of multithreading with off-the-shelf risc processors. In

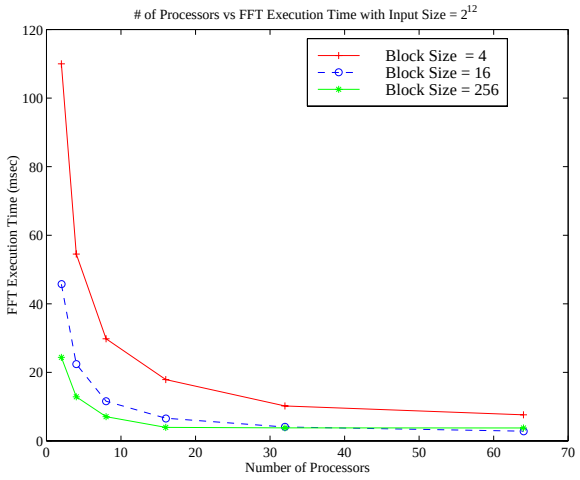

Figure 11: Sender-Initiated Algorithm:Scalability w.r.t to machine size with varying block size and fixed problem size on EARTH-DUAL

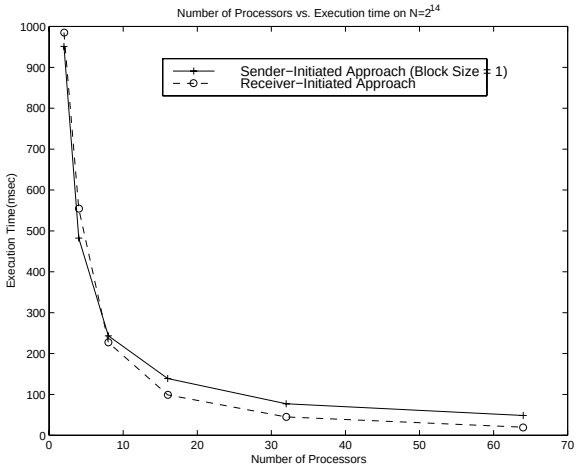

Figure 12: Comparison between the sender-initiated and receiver-initiated (half data size)approaches

Proc. of the First Intl. EURO-PAR Conf., pages 117–128, Stockholm, Sweden, Aug. 1995. Springer-Verlag.

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

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

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

[21] 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.

[22] 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.

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

[24] Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads.* PhD thesis, McGill, Montreal, May 1999.

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

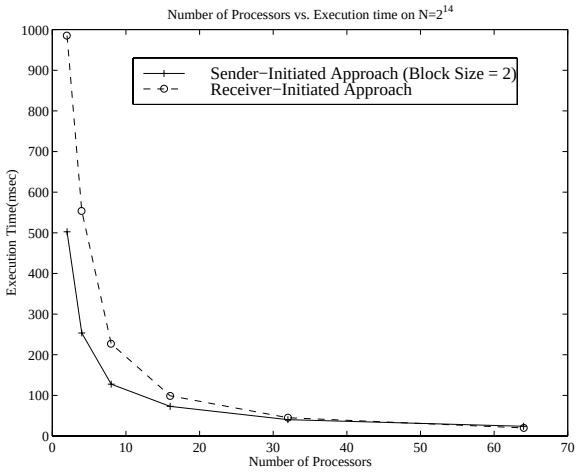

Figure 13: Comparison between the sender-initiated and receiver-initiated (half data size) approaches