A PARALLEL FFT ON AN MIMD MACHINE *

appeared in Parallel Computing, Vol. 15, pp. 61-74, 1990

Amir Averbuch Eran Gabber Boaz Gordissky †
Yoav Medan ‡

† Department of Computer Science
School of Mathematical Sciences
Tel-Aviv University
Tel-Aviv 69978 Israel
‡ IBM Scientific Center
Nuclear Engineering Building
Technion City, Haifa 32000, Israel
E-Mail: amir@MATH.TAU.AC.IL

Abstract

In this paper we present a parallelization of the Cooley-Tukey FFT algorithm that is implemented on a shared-memory MIMD (non-vector) machine that was built in the Dept. of Computer Science, Tel Aviv University. A parallel algorithm is presented for one dimension Fourier transform with performance analysis. For a large array of complex numbers to be transformed, an almost linear speed-up is demonstrated. This algorithm can be executed by any number of processors, but generally the number is much less than the length of the input data.

Key words: Parallel Algorithm, parallel FFT, shared-memory MIMD machine

1 Introduction

The Discrete Fourier Transform (DFT) and its accelerated version, the Fast Fourier Transform (FFT), has become a necessary tool in many various scientific and engineering applications. Since its introduction ([1]) in 1965 it is a target for an extensive research to implement an efficient FFT on general purpose, serial and parallel, machines and on special hardware. The FFT is an ideal candidate to be implementated on a vector processor by careful pipelining and scheduling of the stages of the computation.

*This work was supported by the Basic Research Foundation administered by thee Israeli Academy of Sciences and part by a research grant from National Semiconductor, Israel. Short version of the paper appeared in the 1989 International Conference on Parallel Processing, pp. III-63 - III-70, August 8-12, 1989, Chicago.

The FFT algorithm implementation, written in a high-level source language, is specifically tailored for a particular architecture. Nevertheless, it can be easily implemented on a vector processor since the idea for this implementation came from a successful vectorization of the FFT ([12],[16]) which achieved the full potential speed of vector processors. We will demonstrate very clearly that the architecture dependent parameters are a major factor in speeding up the computation. The main result is that for large arrays (i.e. 64K or greater), the speedup is almost linear. However, if the number of array elements to be transformed is below some threshold, then there is almost no speedup.

This paper is organized as follows. In section 2 we review the various existing architectures on which the FFT has been implemented. In section 3 we present the parallel one dimensional FFT algorithm. There we present a parallel algorithm with four stages to perform the FFT in parallel. In section 4 we describe the 1-dimensional Generalized FFT (GFFT) which includes the DFT as a particular case. The parallel one-dimensional GFFT is the algorithm we implement on our parallel machine. The machine architecture and the specific implementation is described in section 5. We describe the associated hardware and software architecture that we utilize. In addition to the implementation description we detail the integration of the algorithm and the architecture, give a high-level description of the implementation whith an emphasis on the relationship amongst the algorithm and the shared bus & shared memory architecture in order to obtain an optimal performance. The features that compromise a good match between the architecture and the algorithm are described. We will address computational issues such as memory strides, vector length, communication overhead. In section 6 we review and analyze the results based on execution speeds of various sizes of one dimensional FFT and in section 7 we present the performance analysis based on the match between the algorithm and the MIMD architecture.

2 Vector and Multiprocessor FFTs

The FFT is an ideal candidate to be implemented on a vector processor. Its structure, which is mainly a computation of an inner product between vectors, is perfectly suitable to most commercially available vector processors. There is a substantial amount of literature on this subject, [4]-[7],[12],[15]-[16], to name a few. The vectored realizations of the FFT are the main source for the new ideas for the proposed implementation of the FFT on multiprocessors.

References [2] and [3] provide a review of several multiprocessor FFTs that were developed for multiprocessor (and vector multiporcessors) with shared memory and multiprocessors with message passing.

A large variety of multiprocessors exist today and they can be grossly classified as either shared memory or message passing. All inter-processor communication in a shared-memory architecture is done through shared memory. The shared memory may hold global data, be used to emulate message passing, or to store local results. The shared memory may be either uniformly accessed from all processors, divided among processor clusters, or form a hierarchy. The FFT was implemented on the IBM RP3 ([10]-[11]) which is a shared memory multiprocessor, and the multiprocessors are connected to shared memory by a multistage interconnection network. In [3] there is an implementation of the FFT on an ALLIANT

FX/8 which is a vector multiprocessor. The vector multiprocessors consist of several vector processors connected to shared memory. processor.

All inter-communication in a message passing architecture is done by message passing through communication links. Each processor has some local memory. Common data must be duplicated in all processors because there is no shared memory that is globally accessible. The Hypercube is an example of such an architecture. In [3] there is an analysis of a hypercube FFT. [8]-[9] do not specifically discuss the FFT but they focus on the communication complexity of the general algorithms on the hypercube.

Recently, parallel algorithms for one and two-dimensional FFT over a wide variety of interconnection networks were considered in [21] except a binary tree. The problem of computing the two-dimensional FFT on a tree-structured parallel computer is addressed in [22].

3 1-Dimensional Parallel FFT

Let $\{x(n) : 0 \leq n \leq N-1\}$ be N samples of a time function. The DFT is defined

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \quad 0 \leq k \leq N-1 \quad W_N = e^{-i\frac{2\pi}{N}} \tag{1}$$

As we mentioned in section 2, most of the multiprocessor FFT implementations are based on the vector realization of the FFT. Our algorithm is a modification of [15] which is a fully vectorized mixed-radix algorithm, that does not require any special tor operations. This yields the current modification and implementation on our shared-memory parallel MIMD machine.

Assume that

$$N = N_1, \ldots, N_p \tag{2}$$

Conceptually, we can think of a one-dimensional array x of dimension N to be a $p-$ dimensional array x' of dimension $N_p \times N_{p-1} \times \ldots \times N_1$. That is

$$x'(n_p, \ldots, n_i, \ldots, n_1) = x(n_p + n_{p-1}N_p + \cdots n_1 N_2 N_3 \cdots N_p) \tag{3}$$

$$n_i = 0, 1, \ldots, N_i - 1 \quad i = 1, 2, \ldots p$$

The first step of the algorithm is an "index reversal" which is generalization of "bit reversal". In our case, this corresponds to a p -dimensional transposition. Therefore, we define another p-dimensional array x_0 in the following way:

$$x_0(n_1, \ldots, n_p) = x'(n_p, \ldots, n_1) \tag{4}$$

This step is done in parallel.

From (3) we have that

$$x_0(n_1, \ldots, n_p) = x(n_p + n_{p-1}N_p + \cdots + n_1 N_2 N_3 \cdots N_p) \tag{5}$$

We treat the array X of lengnth n as a p -dimensional array x_p

$$x_p(k_1,\ldots,k_p) = X(k_1 + N_1k_2 + \cdots + N_1N_2\cdots N_{p-1}k_p) \tag{6}$$

$$k_i = 0,\ldots,N_i - 1 \qquad i = 1,\ldots,p$$

Note that x_p and X are equivalent arrays. Substituting (5) and (6) in (1), we obtain

$$x_p(k_1,\ldots,k_p) = \sum_{n_1,n_2,\ldots,n_p} x_0(n_1,\ldots,n_p)W_N^{f_pg_p} \tag{7}$$

where

$$f_p = n_p + n_{p-1}N_p + \cdots n_1N_2N_3\cdots N_p \tag{8a}$$

and

$$g_p = k_1 + N_1k_2 + \cdots + N_1N_2\cdots N_{P-1}k_p \tag{8b}$$

and the general definition of these functions is

$$f_j = f_j(n_1,n_2,\ldots,n_j) = n_j + n_{j-1}N_j + \cdots + n_1N_2N_3\cdots N_j \tag{9a}$$

and

$$g_j = g_j(k_1,k_2,\ldots,k_j) = k_1 + N_1k_2 + \cdots + N_1N_2\cdots N_{j-1}k_j \tag{9b}$$

$$j = 1,\ldots,p$$

Define

$$L_j = N_1N_2\cdots N_j \quad j = 1,2,\ldots,p \tag{10}$$

(9a) and (9b) can be written recursively using (10) as:

$$f_j = n_j + f_{j-1}N_j \qquad g_j = g_{j-1} + L_{j-1}k_j \tag{11}$$

From the above

$$\frac{f_jg_j}{L_j} = f_{j-1}k_j + \frac{f_{j-1}g_{j-1}}{L_{j-1}} + \frac{n_jg_{j-1}}{L_j} + \frac{n_jk_j}{L_j} \qquad j = 1,\ldots,p \tag{12}$$

Define $f_0 = g_0 = 0$ and $L_0 = 1$. From [3] and the above, the DFT of (1) can be performed in $j = 1,\ldots,p$ successive stages in which $\frac{N}{N_j}$ one-dimensional DFT's of X_j , of length N_j , are performed along the rows of the partially transformed X_{j-1} :

$$X_j(K_j) = \sum_{n_j=0}^{N_{j-1}} X_{j-1}(K_{j-1}){W_{L_j}}^{n_jg_{j-1}}{W_{N_j}}^{n_jk_j} \tag{13}$$

where

$$K_j = n_{j+1},\ldots,n_p,k_1,\ldots,k_j$$

$$L_j = \prod_{l=1}^{j} N_l$$

$$g_j = \sum_{l=1}^{j} k_l L_{l-1} \quad (L_0 = 1)$$

$$n_j = 0, \ldots, N_j - 1$$

and $X_0(K_0)$ is obtained from $x(n)$ by an "index-reversal" transformation. At the end of the $p-th$ stage, $X(k) = X_p(K_p)$.

In (13), the second term represents the twiddle factor scaling the $j-th$ stage ([15]). The last term represents the DFT's of length N_j.

If a vector of length N that has to be transformed can be represented as a $p-$ dimensional array where $N = N_1 N_2 \cdots N_p$, then the DFT of the vector can be done in p successive stages where each stage consists of the following phases:

1. Load $\frac{N}{N_j}$ vectors of length N_j.
2. Do $\frac{N}{N_j} - 1$ vector multiplications using $\frac{N}{N_j} - 1$ pre-computed twiddle factor vectors.
3. Compute the radix- N_j vector DFT.
4. Store $\frac{N}{N_j}$ vectors.

In phase 2 of the algorithm we have to compute the twiddle factor, i.e. $W_{L_j}^{n_j g_{j-1}} \quad j = 1, \ldots, p$ (13), but a new technique is presented in the next section to incorprate the computation of the twiddle factor into the computation of the FFT which will yield a 3-step parallel FFT. Such a DFT is called a Generalized FFT.

4 1-dimensional FFT and Generalized FFT (GFFT)

The one dimensional Discrete Fourier Transform (DFT) of a finite series $\{x(n) : 0 \le n \le N-1\}$ (real or complex) is the complex series given by

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \quad 0 \le k \le N-1 \quad W_N = e^{-i\frac{2\pi}{N}} \tag{14}$$

The number of complex multiplications for a DFT of N points when $N = 2^r$ is $\frac{N}{2} \log_2 N$. Here we are introducing one type of FFT which is called decimation in time by using the "basic butterfly operation":

$$X_N(k) = G_{\frac{N}{2}}(k) + W_N^k H_{\frac{N}{2}}(k) \quad 0 \le k \le \frac{N}{2} - 1 \tag{15a}$$

$$X_N(k + \frac{N}{2}) = G_{\frac{N}{2}}(k) - W_N^k H_{\frac{N}{2}}(k) \quad 0 \le k \le \frac{N}{2} - 1 \tag{15b}$$

where

$$G_{\frac{N}{2}}(k) = \sum_{m=0}^{\frac{N}{2}-1} x(2m) W_{\frac{N}{2}}^{km}$$

and

$$H_{\frac{N}{2}}(k) = \sum_{m=0}^{\frac{N}{2}-1} x(2m+1)W_{\frac{N}{2}}^{km}$$

The Generalized Discrete Fourier Transform, introduced in [13]-[14], of $\{x(n) : 0 \leq n \leq N-1\}$ with frequency shift parameter α, is defined as:

$$X_N(k) = \sum_{n=0}^{N-1} x(n)W_N^{(k+\alpha)n} \quad 0 \leq k \leq N-1 \quad W_N = e^{-i\frac{2\pi}{N}} \tag{16}$$

Using the same technique as in the classical FFT ((15a),(15b)) we have for the "shifted" DFT the following:

$$X_N(k) = G_{\frac{N}{2}}(k) + W_N^{(k+\alpha)}H_{\frac{N}{2}}(k) \quad 0 \leq k \leq \frac{N}{2}-1 \tag{17a}$$

$$X_N(k+\frac{N}{2}) = G_{\frac{N}{2}}(k) - W_N^{(k+\alpha)}H_{\frac{N}{2}}(k) \quad 0 \leq k \leq \frac{N}{2}-1 \tag{17b}$$

where

$$G_{\frac{N}{2}}(k) = \sum_{m=0}^{\frac{N}{2}-1} x(2m)W_{\frac{N}{2}}^{(k+\alpha)m} \tag{18a}$$

and

$$H_{\frac{N}{2}}(k) = \sum_{m=0}^{\frac{N}{2}-1} x(2m+1)W_{\frac{N}{2}}^{(k+\alpha)m} \tag{18b}$$

Hence, the difference between the FFT and the GFFT is in the value of the twiddle factor used to perform the butterfly operation.

In [16] a different interpretation of (13) is given such that a link between the GFFT and the vectorized version of the FFT ([15]) is established. From (13)

$$X_j(K_j) = \sum_{n_j=0}^{N_j-1} X_{j-1}(K_{j-1})W_{N_j}^{n_j(k_j+\frac{g_{j-1}}{L_{j-1}})} \qquad \frac{g_0}{L_0} = 0 \tag{19}$$

This is similiar to (16) where $\alpha_{j-1} = \frac{g_{j-1}}{L_{j-1}}$ $(\alpha_0 = 0)$. From (16)-(18b), Eq. (19) can be interpreted as the generalized in frequency FFT (GFFT) of X_{j-1}. During the initialization stage of the algorithm, $L_j = \prod_{l=1}^{j} N_l \quad j = 1, \ldots, p-1$ are pre-computed. In [6] it was proved that the arithemtic complexity of GFFT and FFT are the same, using a different set of twiddle factors. Therefore, we can eliminate the need for phase 2 of the algorithm. We should add that by using the vectored GFFT, we reduce both initialization time and memory requirements since the array α_{j-1} is smaller than the size of the twiddle factor array $W_{L_j}^{n_j g_{j-1}}$.

The elimination of phase 2 of the algorithm implies that the computation of $\frac{N}{N_j}-1$ vector multiplications using $\frac{N}{N_j}-1$ pre-computed twiddle factor vectors can be incorprated in the computation of the FFT, and is the main reason for the speedup achievement. This fact increases the locality of the computation (which will be explained later).

When implementing this algorithm on MIMD machine the main problem is how to exploit the architecture to perform the required data organization between each stage of the proposed algorithm. The data flow graph for the computation is illustrated in Figure 1. The main problems encountered were how to avoid the steady decrease in the vector length at successive stages, how to avoid costly data unscrambling, and optimal load balancing.

Figure 1: Data flow for parallel FFT for $N = 2 \times 4 \times 2$ points

5 Implementation

5.1 MMX parallel Machine

The MMX parallel processor[18-19] is an experimental shared memory multiprocessor built at the computer science department of Tel-Aviv University from commercial components. It was intended for research in software development for multiprocessors and not for hardware innovation.

The MMX processor consists of 4-8 National Semiconductor DB32016 processor boards connected by an industry standard bus (Multibus I) to 3 MB of shared memory. Each processor board contains a CPU, MMU, floating point coprocessor, interrupt control unit, two RS232 serial ports and 2MB of local RAM. The serial lines are the only I/O device connected to the parallel processor. They are used to control the processors, download the programs and perform I/O to and from the host.

The host computer (a SUN3 server) is connected to each of the processor boards by the serial lines. It is used for program development, cross compilation and downloading. Although the MMX parallel processor is currently a single user machine, its operation is simple and flexible, because it is controlled by a custumized windows system running on any workstation connected to the host computer via the LAN. The MMX processor structure is illustrated in the following figure:

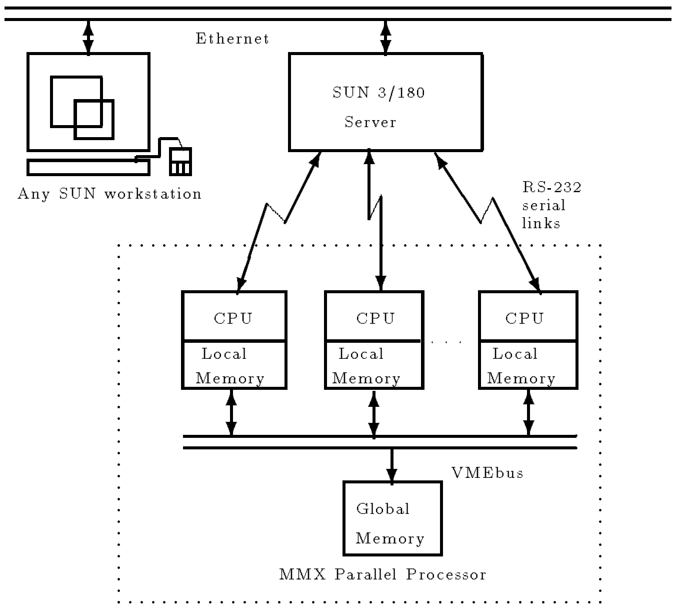

Figure 2: MMX Parallel Machine

The MMX operating system is a run time executive, which provides services to a single user program running on the MMX parallel processor. The MMX operating system is linked together with the user program to create a single object file, which is downloaded into the parallel processor.

MMX provides multitasking and scheduling services to user programs. It also supports blocking and non-blocking message passing via mailboxes, blocking semaphores, dynamic memory management of local and global memory, memory protection using the MMU, clock services and low level operations like spin locks and atomic fetch&add operation. All I/O operations are performed by the GNX run time library, which is supplied by National Semiconductor. It also supports dynamic task creation, automatic load balancing by task migration, tasks synchronization by semaphores, inter-task communication by mailboxes and time-sharing between tasks.

The MMX parallel processor and its operating system has been in constant use since summer 1987. It is now serving more than 10 regular users, who use it for research in software environments, image processing, signal processing and applied mathematics.

5.2 The specific implementation

The algorithm is written in C with MMX services. The performance is optimal when the vectors are equal in length, i.e $N = M^p \quad (N_1 = N_2 = \cdots = N_p)$. The algorithm is performed in p stages, where in each stage $\frac{N}{M}$ GFFT operations are performed on vectors of length M. The $\frac{N}{M}$ GFFT operations are performed in parallel depending on the number of processors. It supports any number of processors.

The parallel program is executed by calling the procedure *main _ FFT* with the following parameters:

N The number of points to be transformed.

$M\,p$ Decomposition of N to $N = M^p$.

$\log M$ The number of bits that represents the number $M-1$. This is utilized by the "index reversal" of the input vector.

Inverse TRUE to perform Inverse FFT, else FALSE.

Vec_real, Vec_image Pointers to real and imaginary parts of the input.

The performance in shared memory & shared bus architecture has been degraded because of the need to access the global memory. This access is done through MULTIBUS which is slower than the bus that is used by the local memory. When a processor wants to access the global memory while the bus is used by another processor then the processor has to wait until the bus is freed. We tried to avoid a frequent use of the global memory. To accomplish the above the following design decisions were taken into consideration:

1. In MMX we can define two types of tasks: *Global* and *local* . Global task can reside in any processor and during the execution it can travel from one processor to another. On the other hand, the local task processor-id is predetermined by the program when the local task is generated. During the execution period the local task will reside on this processor. Local task achieves better performance than a global task due to the fact that its stack is located in the local memory which belongs to the processor that executes this task.

The emphasis in the implementation was to rely on using the maximum number of local tasks.

The following considerations are direct implications of the need to concentrate more on local tasks and local memory.

2. Priority was given to dynamic allocation of memory space in the local memory.

3. We tried to eliminate, as far as possible, the use of global variables.

4. In case a processor necessitates massive access to global memory, then the data in the global memory is copied to local memory. Then the computation is done in local memory and after completion of the computation the data is moved back to global memory.

5. We tried to eliminate, as far as possible, to use of message-passing among the tasks since message-passing is done through the use of global memory.

5.3 Integration of the algorithm and the architecture

As was explained before, in order to perform one-dimensional FFT of length $N = M^p$ the algorithm is performed in p stages where in each stage we performed $\frac{N}{M}$ GFFT operations on input vector of length N . These vectors contain the results from the previous stage since the computation is done in place. The $\frac{N}{M}$ input vectors to the GFFT algorithm are generated from reading the input vector while the reading is performed in step 1. The $\frac{N}{M}$ vectors which contain the results of the GFFT in the current stage are written in $\frac{N}{M}$ steps into the output vector which will be the input vector for the next stage.

The $\frac{N}{M}$ operations of the GFFT algorithm are performed in parallel. These operations are independent of each other. Therefore, the number of GFFTs that are performed in parallel in each stage is equal to the number of available processors.

During the execution two main tasks are created **Main_FFT** and **Slave_FFT** (see figure 3).

1. Main_FFT:

Only one task of this type is created. It is the first created task and it generates the rest of the tasks which are all of type **Slave_FFT** . In general, **Main_FFT** is used as a "supervisor" and "synchronizer" among the other tasks.

Its main tasks are:

- Performs general initializations.
- Responsible that the P stages of the algorithm are carried into effect.
- Signal the tasks of the type **Slave_FFT** to start to execute the next step of the algorithm.
- Avoid the situation in which part of the **Slave_FFT** starts to execute the next step of the algorithm while some of them are still busy with the current step of the algorithm.

- Perform all kinds of "administrative" types of work like the exchange among input vectors and output vectors during the completion of the execution in each step of the algorithm.
- To take care that the results of the algorithm are written in the user buffers.

2. Slave_FFT:

The number of available processors dictates the number of **Slave_FFT.** Therefore the number of **Slave_FFT** tasks is equal to the number of processors. These tasks are responsible for the main computational task of the algorithm. In each stage of the P stages algorithm each of the **Slave_FFT** task performs the following:

- Loads $\frac{N}{M \times \#tasks}$ vectors from the global memory into the local memory ($\#tasks = \#processors$).
- Computes for each of these vectors the twiddle factor.
- Perform GFFT on each of the vectors.
- Stores the results in global memory.

The GFFT algorithm is described in detail in the appendix.

In figure 3 there is a functional flow-chart that demonstrates the interaction among the tasks and their responsibilities.

Figure 3: Parallel FFT flowchart

6 Analysis of the results

The parallel FFT has been run on the MMX [18-19] system. To measure the speedup of the parallel FFT we execute a serial FFT on a single 32016 processor. The serial FFT is given without any system service call to MMX. This serial FFT is the fastest version of the FFT we can derive. Transform sizes, which vary from 256 to 65536 points, have been chosen. The FFT computation is done in radix-2,4 and 8.

The results are given in Tables 1-3. The first column (from left to right) shows the size of vector to be transformed. The other columns contain times for the parallel FFT when it runs on 1 to 4 processors. The speedup, defined as,

$$speedup \;=\; \frac{serial\ \ time}{parallel\ \ time}$$

is given under each parallel time. The serial time is given in the rightmost column. The time in the tables is given in seconds. These mesures do not include I/O time, preprocessing of sin and cos tables, and the twiddle factor which is initialized in the first FFT routine call.

FFT radix-2,radix-4 and radix-8 Measurements

#points	Parallel Time(#processors)				Serial Time (sec.)
	4	3	2	1	
256	0.17	0.22	0.28	0.49	0.36
Speed Up	2.12	1.64	1.29		
512	0.40	0.50	0.70	1.31	0.81
Speed Up	2.00	1.62	1.16		
1024	0.59	0.83	1.10	2.14	1.77
Speed Up	3.00	2.13	1.61		
4096	2.43	3.29	4.78	9.45	8.52
Speed Up	3.51	2.59	1.78		
16384	10.56	14.35	20.91	41.45	39.58
Speed Up	3.75	2.76	1.89		
65536	45.85	61.06	91.06	180.77	180.15
Speed Up	3.93	2.95	1.98		

Table 1: FFT radix-2 Algorithm timing

The entries in the above tables which are marked with †do not fit the processing. In other words, their size does not decompose into an optimal sub-problems. This is only happening in Radix-4,8.

The overhead from the MMX services is best demonstrated when we examine the performance of the parallel FFT on a single processor.

The speedup increases with the problem size. This is due, in part, to the increase of the utilization of each processor in the parallel algorithm. Said differently, as the problem size increases, the amount of time spent in synchronizing barriers decreases. Since the algorithm transfer data among global and local memories, the inefficiency of such transfer becomes

#points	Parallel Time(#processors)				Serial Time (sec.)
	4	3	2	1	
256	0.16	0.20	0.24	0.43	0.30
Speed Up	1.87	1.5	1.25		
1024	1.01	1.25	1.79	3.44	1.45
Speed Up †	1.43	1.16	0.79		
4096	2.06	2.80	4.06	7.99	6.73
Speed Up	3.26	2.40	1.65		
16384	20.23	26.38	38.78	75.94	30.74
Speed Up †	1.51	1.16	0.79		
65536	38.23	50.85	75.79	150.18	138.22
Speed Up	3.61	2.71	1.82		

Table 2: FFT radix-4 Algorithm timing

#points	Parallel Time(#processors)				Serial Time (sec.)
	4	3	2	1	
512	0.35	0.42	0.58	1.09	0.59
Speed Up †	1.68	1.40	1.01		
4096	1.95	2.63	3.82	7.50	5.93
Speed Up	3.04	2.25	1.55		
32768	29.53	38.77	57.35	113.06	57.33
Speed Up †	1.94	1.47	0.99		

Table 3: FFT radix-8 Algorithm timing

insignificant when the problem size increases. The efficiency is defined as

$$efficiency \quad = \quad \frac{speedup}{number \ \ of \ \ processors}$$

In the GFFT-radix 2 of 65536 points, the efficiency is 0.93.

On the other hand, when the radix in the algorithm increases, the absolute parallel time and the serial time is decreasing, but the speedup decreases also. This is due to the fact that the serial algorithm in radix 4 and 8 "gains" more than the radix-2 serial algorithm. For example, for $4096 = 64 \times 64$ points, there is parallel GFFT on 64 points. Therefore, in radix-4 the number of multiplications is 576, and in radix-8 the number of multiplications is 512. On the other hand, the number of multiplications in the radix-2 parallel algorithm is 768. In the serial algorithm, in radix-4 there are 73728 multiplications, in radix-8 there are 65536 multiplications and in radix-2 there are 98304 multiplications.

7 Performance Analysis

To get near peak performance from the machine we note that the delivered speedup is dramatically better for much longer transforms. In this case, the parallel overhead is minimized because of the greater amount of work performed in the local processors. In addition, there is less traffic on the bus and less synchronization is required. In [23], on the other hand, the increase of the vector length may severely damage the data locality and therefore make poor use of the cache.

The GFFT that is described in this paper is fitted to this architecture and its features comprise a good match between the architecture (the increase of the locality of the computation) and the algorithm (the preprocessing of the twiddle factor which reduce substantially the amount of access to the global memory). The locality of the computation, which is based on local tasks, enables this parallel FFT to achieve an almost linear speedup and hence demonstrates an almost optimal integration of the algorithm and the architecture.

From examination of the results and by the structure of the algorithm it is impossible to improve performance in the range of short transforms. In the case of short transforms, the parallel overhead is dominates the gain in the performance since the processors are doing less and more time is spent on synchronization, message passing, use of semaphores and access to global memory.

Can we improve the current algorithm ? There are some points in the implementation which may be considered as candidates for change and thus will yield better performance:

- From the results (Tables 1-3) we have that the speedup using three processors is less than the speedup gain using four processors. One reason is that for odd number of processors the distribution of the work load is not uniform. Therefore, we have that some processors are busy while the other are idle. If another scheme of the work load which is more uniformily distributed is proposed then the anomaly in the speedup gain may be prevented.

- In the implementation of the algorithm we limit ourselves to the case where the transform size is $N = M^p$. But the algorithm can fit the case $N = N_1 \times N_2 \times \cdots N_p$. If the implementation was based on this decomposition we may have better results. On the other hand, in this general setup, it is impossible to hold pre-computed coefficient tables for each N_i and therefore we are forced to get them from the on-line computation of the mathematical functions sin and cos which may degrade the performance.

  But a potential algorithm for a mix-radix GFFT is given by the following analysis. If we denote by η the ratio between the computational complexity of the twiddle factor array and that of the frequency shift array, we have

$$\eta = \frac{\sum_{j=1}^{p} L_{j-1}(N_j - 1)}{\sum_{j=1}^{p} L_{j-1} - 1} \tag{20}$$

  Eq. (20) can be easily reduced to the following form:

$$\eta = \frac{L_p - 1}{\sum_{j=1}^{p} L_{j-1} - 1} \cong N_p - \frac{N_p}{N_{p-1}}$$

  An approximate upper limit for η is given by:

$$\eta \cong N_p - \frac{N_p}{N_{p-1}} - \cdots \tag{21}$$

  According to Eqs. (20-21) it can be concluded that:

- To increase the efficiency of a mixed-radix GFFT scheme it is recommended to factor the input vector in an ascending order of N_j.
- For equi-radix $N_j = r \quad j = 1, \ldots, p$, the bounds equals to $r - 1$. However, in this case it is suffice to compute only the corresponding arrays of the last stage as they are composed of sub-arrays of the preceding stages which can be referenced by proper indexing.
- The potential savings increase linearly with the radix of the transform kernel.

- If the input to the algorithm is "organized" in advance (i.e. "index reversal" has been performed already) then there will be a performance gain. Although index reversal is done in parallel, it dictates massive access to the global memory and this a bottle neck in improving the performance.

We should add that we believe that this method will yield the same of type of results for two-dimensional GFFT. Due to limitations of storage in the current version of the machine we can not test it right now. This problem will be resolved partially in the next version which will run on the latest product of National Semiconductor microprocessor-DB32532. We can not prove now the validity of this method to a speed up of long sequences of two-dimensional GFFT.

The code of the serial and parallel versions of the GFFT algorithm can be obtained from the authors.

8 Conclusions

We have described an efficient algorithm for computing 1-dimensional FFT on a shared bus & shared memory parallel MIMD machine. We observe that in order to get a speedup the length of the vector to be transformed should be large enough.

The same method can be used to derive a parallel 2-dimensional parallel FFT.

The open question is whether there is an optimal number of processors such that the speedups level off with increasing the number of the processors ? In other words, what is the tradeoff between the speedup and the number of processors on this type of architecture ?

The MMX is just one representative of the class of MIMD computers. It would be useful to implement this algorithm on other types of multiprocessors.

A planned project is to incorporate the parallel FFT into the FACR(l) algroithm [24] for the direct solution of Poisson's equation. Another planned project is to use an existing software tool, developed by us, to have an automatic porting of the code to a message passing type of machine architecture.

It should be mentioned that hyperbolic differential equation using this parallel FFT based on [25] was implemented on the MMX [26]. For large array of numbers an almost linear speed-up was demonstrated. Pseudospectral methods were used to solve the hyperbolic differential equation.

9 References

1. Cooley, J.W., and J.W. Tukey, An Algorithm for the Machine Computation of Complex Fourier Series, *Math. Comp. 19, pp.297-301 (1965).*

2. Hwang, K., F.A. Briggs, Computer Architecture and Parallel Processing , *McGraw Hill, 1986.*

3. Swarztrauber, P.N., Multiprocessor FFTs, *Parallel Computing 5, pp.197-210(1987).*

4. Fornberg, B., A vector implementation of the Fast Fourier transform, *Math. Comput. 36, pp.189-191 (1981).*

5. Korn, D.G., J.J. Lamboitte, Computing the fast Fourier transform on a vector computer, *Math. Comput. 33, pp.997-992 (1979).*

6. Swarztrauber, P.N., FFT algorithms for vector computers, *Parallel Computing 1, pp.45-63(1984).*

7. Swarztrauber, P.N., Vectorizing the FFTs, *in: G. Rodrigue, ed., Parallel Computations, Academic Press, NY, pp. 490-501, 1982.*

8. McBryan, O.A., E.F. Van de Velde, Hypercube algorithms and implementations, *SIAM J. Sci. Statist. Comput.*

9. Pease, M.C., The indirect binary n-cube microproceesor array, *IEEE Trans. Comput. 26, pp.458-473(1977).*

10. Norton, A., A.J. Silberger, "Parallelization and Performance Analysis of the Cooley-Tukey FFT Algorithm for Shared-Memory Architectures, *IEEE Trans. on Computers, Vol. C-36, No. 5,pp.581-591, 1987.*

11. Cvetanovic, Z., Performance analysis of the FFT algorithm on a shared-memory parallel architecture, *IBM J. of R&D, Vol. 31, No. 4, pp.419-510, 1987.*

12. Agarwal., R.C., J.W. Cooley, Fourier transform and convolution subroutines for the IBM 3090 Vector Facility, *IBM J. of R&D, Vol. 30, No. 2, pp.145-162, 1986.*

13. Bongiovanni, C., P. Corsini and G. Frosini, One-Dimensional and Two-Dimnesional Generalized Discrete Fourier Transform, *IEEE Trans. ASSP, Vol. ASSP-24, pp.97-99, 1976.*

14. Corsini P. and G. Frosini, Properties of the Multidimensional Generalized Fourier Tranform, *IEEE Trans. on Computers, Vol. AC-28, pp.819-830, 1979.*

15. Agarwal, R.C., An Efficient Formulation of the Mixed-Radix FFT Algorithm, *Proc. Intl. Conf. Computers, Systems, and Signal Processing, pp. 769-772, Bangalore, India, Dec. 10-12, 1984.*

16. Agarwal., R.C., J.W. Cooley, Vectorized Mixed Radix Discrete Fourier Transform Algorithms, *Proc. of the IEEE, Vol. 75, No. 9, pp.1283-1292, 1987.*

17. Medan, Y., Vectored Generalized DFT Revisited, *IBM Science and Technology IBM Israel research report 88.215, 1987.*

18. Gabber, E., Parallel Programming Using the MMX Operating System and its Processor, *Proc. of the 3rd Israeli Conf. on Computer Systems and Software Engineering, Tel Aviv, Israel, pp. 122-132, June 1988; Also Technical Report 107/88, Dept. of Computer Science, Tel-Aviv University, May 1988.*

19. Gabber, E., MMX - Multiprocessor Multitasking eXecutive User's Manual *Dept. of Computer Science, Tel-Aviv University, 1987.*

20. Evans, D.J., S. Mai, A Parallel Algorithm for the Fast Fourier Transform, *Parallel Algorithms & Architectures, M. Cosnard et al. (editors), Elsevier Science Publishers B.V. (North-Holland), pp. 47-60, 1986.*

21. Jamieson, L.H., P.T. Muller, H.J. Siegel, FFT Algorithms for SIMD Parallel Processing Systems, *J. of Parallel and Distributed Computing, 3, pp. 48-71, 1986.*

22. Gorin, A., A. Silberger, L. Auslander, Computing the Two-Dimensional FFT on a Tree Machine.

23. Gannon, D., Jalby, W., The Influence of Memory Hierarchy on Algorithm Organization: Programming FFTs on a Vector Multiprocessor, *in The Characteristics of Parallel Algorithms, Jamieson et. al. editors, MIT Press, pp. 277-301, 1987.*

24. Hockney, R. W., A fast direct solution of Poisson's equation using the Fourier analysis, *J. Assoc. Comput. Mach., 12, pp. 95-113, 1965.*

25. Gottlieb, D., Orszag, S.A., Numerical Analysis of Spectral Methods: Theory and Applications, *SIAM, vol. 26, 1977.*

26. Averbuch, A., Gabber, E., Gordissky, B., Neumann, H., Solving Hyperbolic Differential Equation using FFT on Parallel MIMD machine, *to appear.*

Appendix: The GFFT algorithm $N = M^p$

```
#tasks = Number of processors currently available on the system
N      = Number of sample points
Main_FFT( )

  - If the activation of the FFT is done for the first time or the FFT
    parameters were changed then
     BEGIN
         - Allocate global memory for the new buffers
         - Compute new lookup tables to hold the sin  and cos  which
           help to speedup the computation of the twiddle factor
         - Create new resources such as mail_box for message-passing and
           semaphores for synchronization.
     END.

  - Initialize semaphore to the value= -(#tasks-1)

  - For id=0 to #tasks-1 do
     BEGIN
         /* Create local slave tasks equal to the number of processors */
           Spawn Slave_task(id)
     END.

  - Wait on semaphore for all Slave tasks to finish initialization

  - Perform the P  stages of the algorithm, while at each stage
     BEGIN
         - Initialize semaphore to -(#tasks-1)
         - Send Do_FFT message to all slave tasks in order to start the
           new stage
         - Wait on semaphore until all the slave tasks complete theirs
           work
         - Swap the pointers of the input vectors with the output vectors in
           the current stage
     END.

  - Insure that the FFT results are in the calling procedure buffers
    If not then
     BEGIN
         - Send Copy_Buffers message to all slave tasks in order to copy
           the results into the user input vectors
         -Wait for all slave tasks to complete their work
     END
```

- Send termination message to all slave tasks
- Terminate

$0 \leq id \leq$ #tasks-1

```
Slave_FFT(id)

  - If the activation of the FFT is done for the first time or the FFT
    parameters were changed then
     BEGIN
         - Allocate new buffers in local memory
         - Store local buffers pointers in global memory
         - Initialize length M, sin and cos lookup tables
     END.

  - Read from global memory the pointers to local memory buffers

  - Perform proportional part of the indices reversal ( i.e indecies
    id x (N/#tasks) ... (id+1)x(N/#tasks -1) on the input vector of length N

  - Signal using semaphore that initialization has finished

  - Wait at the mail_box for a message from the Main_FFT task

  - While the received message is Do_FFT
     BEGIN  ( New stage )= N/(Mx#tasks) GFFT of length  M
         - For $ i=idx(N/(Mx#tasks)) to $ (id+1)x((N/(Mx#tasks))-1) do
             BEGIN
                 - Compute the new twiddle factor
                 - Load by stride of 1 length  M vector from the stage input
            vector (in global memory) into local buffer
            (i.e start loading at index ixM in global buffer)
                 - Compute GFFT(twiddle factor) on local buffer
                 - Store by stride of N/M the local result vector of length M
            in the stage output vector (in global memory)
            (i.e. start storing at index i in global buffer)
             END.
         - Signal using semaphore that this slave has finished current stage
         - Wait at the mail_box for a message from the Main_FFT task.
            (i.e. actually wait for all slave tasks to finish their work
            before starting a new stage)
    END.

  - If the received message is Copy_Buffers
    then BEGIN ( Copy the FFT results to the user input vector)
              - Perform a proportional part of the copy from the input
                buffer to the output buffer
              - Signal using semaphore that this slave has finish copying
```

```
        END.
```

- Terminate