COMPARATIVE ANALYSIS OF FFT ALGORITHMS IN SEQUENTIAL AND PARALLEL FORM

Michael Balducci, Ajitha Choudary, Jonathan Hamaker

Parallel DSP Group
Department of Electrical and Computer Engineering
Mississippi State University
Mississippi State, Mississippi 39762
{balducci, ajitha, hamaker}@erc.msstate.edu

ABSTRACT

There have been a large number of Fast Fourier Transform (FFT) algorithms which have been developed over the years. Among these are the Radix-2 algorithm, Radix-4 algorithm, Split-Radix algorithm, Decimation-in-Time-Frequency algorithm (DITF), Quick Fourier Transform (QFT), and the Fast Hartley Transform (FHT). However, there has not been much prior work where the user is given a single interface to poly-functional implementation that transparently optimizes space and time complexity.

In this paper we present the implementation and benchmarking of the sequential and parallel versions of the above mentioned FFT algorithms. All algorithms have been rigorously compared based on computational time, object size, code size, data dependence (real or complex) and the number of mathematical operations involved in the computations. The results of this endeavor will serve as a frame for creating an object-oriented FFT environment which will automatically choose the most efficient algorithm for a given platform, data set, and order or other user-specified criteria.

1. INTRODUCTION

The development of Fast Fourier Transform (FFT) has evolved over many years. The first major breakthrough was the Cooley-Tukey algorithm developed in the mid-sixties which resulted in a flurry of activity on FFT's. [1] [2] Further research led to the development of the Fast Hartley Transform and Split-Radix algorithm. Recently two new algorithms have also emerged: Quick Fourier Transform and the Decimation-in-Time-Frequency algorithm. Now research has to be geared towards finding which of these algorithms is most efficient. The problem that arises is the inherent trade-off between computation speed, memory usage and the algorithm complexity. Instead, we must attempt to find the most efficient algorithm under the constraints of a particular application. Our efforts will accomplish this task by benchmarking each algorithm under a variety of constraints. The benchmark statistics will be used to create an automated environment capable of choosing the most efficient algorithm for a given application.

The FFT is one of the important and most widely used Digital Signal Processing (DSP) algorithms. FFT algorithms are efficient methods of calculating the DFT. The DFT converts the input signal from the discrete time domain, x(n), to the discrete frequency domain, X(w), and vice versa. This is very useful in eliminating the unwanted noise signal from any communication signal (information bearing signal) being analyzed. Once the signal is converted into the frequency domain the noise or unwanted signal frequencies can be effectively filtered. Then, by using the inverse FFT, the communication signal can be converted back to the time domain. The FFT has many wide-ranging applications in nearly every signal processing field including speech, image processing, communications, cellular phones, modems, and digital control systems. [3]

For most applications computation time plays a significant role in the use of FFT's. The more time spent computing means more efficient utilization of resources; hence, more data can be processed in the same time. The computation time can be reduced using the symmetry, periodicity, etc. of the DFT. The computation time can also be reduced using parallelism in FFT's. By using the FFT algorithms in parallel, the data set can be separated into smaller blocks. The different blocks of data can then be processed at the same time across multiple processors. There is, of course, a trade-off with this method: overhead of communication between processes. To make efficient use of the parallel method the communication time must be minimized.

1. Theory of FFTs

The Fourier transform, *H(w)*, of a signal, *h(t*), is given by the equation of Figure (1a), where *h(t)* is the time-domain signal, *t* is the time and *w* is the angular frequency. The assertion of this equation is that any time-domain signal can be "transformed" into a function of frequency. A plot of this frequency-dependent function gives the frequency content of a particular signal over the entire frequency spectrum. The resulting function contains the magnitude and phase information for each frequency point in the spectrum. This process of conversion from the time to frequency-domain is invertible using the Inverse Fourier Transform of Figure 1b. By the inverse transform, a frequency-domain signal is transformed back to the time domain. [1] [3]

$$H(w) = \int_{-\infty}^{\infty} h(t)\, e^{j2\pi t} dt \qquad 1a$$

$$h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(w)\, e^{-j\omega t} \qquad 1b$$

Figure 1. Forward and Inverse Fourier Transform

The Discrete Fourier transform (DFT) is the digital equivalent of the Fourier Transform. It is a bounded length sequence which is more practical than the infinite summation of the Fourier Transform. The DFT assumes that the input signal is periodic with a period equal to the length of the input sequence. The Discrete Fourier transform is defined as shown in Figure (2a) and its inverse is shown in Figure (2b) [7].

$$X(k) = \sum_{n=0}^{N-1} x(n)\, e^{\frac{-j2\pi kn}{N}} \qquad 2a$$

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k)\, e^{\frac{j2\pi kn}{N}} \qquad 2b$$

Figure 2 Forward and Inverse Discrete Fourier Transform

The DFT is one of the most important concepts in digital signal processing but it is not useful for practical applications. In computing the DFT directly, there are $4N^2$ multiplications and N(4N-1) additions, therefore the computation time is of the order N^2. This type of complexity is not satisfactory for large N. We desire complexities which are linear with data size. At this time, no algorithm is available with such performance but we can derive a class of algorithms which give an efficient alternative to the DFT. These algorithms are collectively known as Fast Fourier Transforms (FFTs).

2. Algorithms

The Fast Fourier Transforms included here use several different approaches in reducing the computation cost of calculating the fourier coefficients. One method employed by several of the algorithms is the divide-and-conquer approach. These algorithms use the fact that a input sequence of length N can generally be broken into a number of smaller sequences. Since the DFT has an order of complexity N^2, breaking the summation into β sections of length-N/β results in a reduction of complexity as shown in Figure 3. [4]

$$\text{Complexity} = \left(\frac{N}{\beta}\right)_0^2 + \left(\frac{N}{\beta}\right)_1^2 + \underset{\cdots}{\cdots} + \left(\frac{N}{\beta}\right)_\beta^2 = \frac{N^2}{\beta}$$

Figure 3 Complexity of Divide-and-Conquer approach. Note the reduction in complexity from the $O(N^2)$ of the DFT.

Another approach is to utilize the periodicity of the DFT. From Figure 4, it can be seen that the multiplying factors of the DFT exhibit both horizontal and vertical symmetry about the unit circle when N is a power of two. Thus, by realizing that the coefficients repeat, redundant calculations can be eliminated. [3]

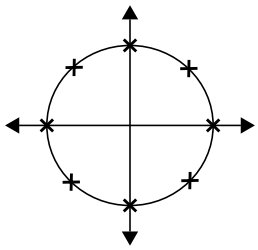


Figure 4 Plot of W_N on the unit circle. This complex function exhibits both horizontal and vertical symmetry when N is a power of two.

2.1. Radix-2 and 4 Algorithms

2.1.a. Sequential Form

By limiting our data-length to the form $N = R^V$ we can define a class of FFTs known as radix algorithms. These algorithms successively decompose a single N-point DFT into R segments of N/R-point DFTs. The most widely used of these radix algorithms is the Radix-2 and the Radix-4. Each uses the periodic properties of the DFT to attain higher efficiency levels. Radix algorithms can be implemented by either using Decimation-In-Time (Cooley and Tukey) or by Decimation-In-Frequency (Sande and Tukey). Each of these algorithms reduces the number of operations from $O(N^2)$ to $O(N \log_2 N)$. The drawback is that the data must be of a specified length. This problem can be avoided by zero-padding with no loss of information. [1] [5]

2.1.b. Parallel Form

The parallel structure of the radix algorithms is understood by considering the fact that the DFT can be decomposed into smaller independent DFTs. By performing each of these smaller DFTs concurrently, we can take advantage of this parallelism. Figure 5 shows two stages of a Decimation-In-Time FFT. In this figure, one can see that the 8-point DFT is decomposed into two 4-point DFTs. These, in turn, are each decomposed into two 2-point DFTs.[6]

.

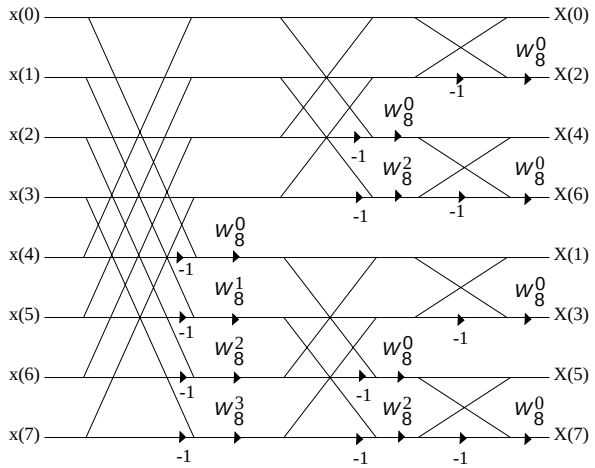


Figure 5 Flowgraph of an 8-point Decimation-in-Frequency FFT. The independence of the odd and even sample calculations allows for evaluation on separate processors

2.2. Split-Radix Algorithm

2.2.a. Sequential Form

By observing Figure 5, it can be seen that even index points can be calculated independently of the odd indexed points. This leads to the possibility of making use of more than one algorithm for the data set. The increase in computational efficiency of the higher order Radix-4 is attractive, but the limitation in data sequence lengths is a hindrance. The Split-Radix utilizes the fact that the data points can be decomposed into even and odd indices to employ both Radix-2 and 4 algorithms. A Radix-2 is performed on the even index points. The odd points are then decomposed into two N/4 point sequences where a Radix-4 approach is taken. In making use of these two techniques, the Split-Radix acquires an increase in computational efficiency over the Radix-2 while retaining the its ability to perform on any power of two. [7]

2.2.b. Parallel Form

As discussed above, the Split-Radix algorithm is composed of a Radix-2 and two Radix-4 components. These components are independent of each other and, thus, can be performed in parallel separate processes. Figure 6 shows the flow of this process. Also, notice that the Radix-2 and Radix-4 components can each be performed as parallel computations (see section 2.1.b). In this manner, we can achieve maximum utilization of parallel hardware.

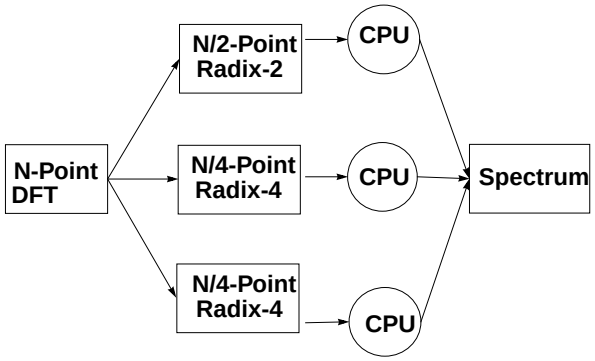


Figure 6 Parallel composition of the Split-Radix FFT and FHT.

2.3. Fast Hartley Transform

2.3.a. Sequential Form

The Discrete Hartley Transform (DHT), shown in Figure 7, takes the approach that fewer is better. Since complex arithmetic requires four real multiplications for every complex multiplication and two real additions for every complex addition, it is computationally very expensive. In addition to requiring more operations, complex numbers also require more memory since a complex number consists of two real coefficients. The DHT reduces the number of computations and memory used by simplifying the kernel of the DFT to real valued variables. [8]

$$X_H(k) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n \left[\cos\left(\frac{2\pi kn}{N}\right) + \sin\left(\frac{2\pi kn}{N}\right) \right]$$

Figure 7 The Discrete Hartley Transform

This new transform shares many of the properties of the DFT. Due to the similarity between the DFT and the DHT, the techniques employed by FFT to calculate the DFT can be applied to the DHT. This allows for an even greater reduction in the number of computations. Although the DHT requires fewer computations that the DFT, there is a drawback in calculating the fourier coefficients using the DHT. Additional computations are required to convert from the DHT to the DFT. However, since the relationship is linear, the computation cost is minimal. The relationship between the DHT and DFT is shown in Figure 8. [8]

$$Re(X(k)) = \frac{(X_H(k) + X_H(N-k))}{2}$$

$$Im(X(k)) = \frac{(X_H(k) - X_H(N-k))}{2}$$

Figure 8 Relations for conversion between Hartley coefficients and Fourier coefficients

2.3.b. Parallel Form

In our efforts, we chose the Split-Radix form of the FHT. The parallel form of this algorithm follows the form of the Split-Radix DFT shown in Figure 6. The difference occurs in the end when the Hartley coefficients must be converted to Fourier coefficients. This step introduces additional computations which can be performed in parallel. [9]

2.4. Quick Fourier Transform

2.4.a. Sequential Form

Whereas most FFTs use the periodic properties of the DFT to reduce complexity, the Quick Fourier Transform (QFT) uses the symmetry of the cosine and sine terms in the DFT to decrease the number of complex calculations. The QFT is constructed by breaking the data set into real cosine terms and imaginary sine terms and those into their even and odd parts. The length-(N+1) cosine terms and length-(N-1) sine terms can be recursively decomposed into two length-(N/2 +1) Discrete Cosine Transforms and two length-(N/2-l) Discrete Sine Transforms respectively as shown in Figure 9. Using this decomposition process we reduce the complexity of the DFT to a complexity of O(N $\log_2$N). Another important aspect of the QFT is that all complex operations occur at the last stage of the decomposition. This makes it well suited for real data. However, complex data can be processed by taking real transforms of the real and imaginary portions of the input separately and combining the results. [10]

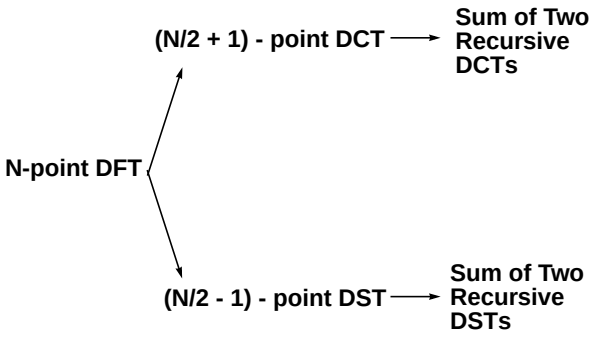


Figure 9. Flow Diagram of the Quick Fourier Transform

2.4.b. Parallel Form

The QFT is recursively decomposed into DCTs and DSTs according to the tree structure shown in Figure 10. We see from this figure that the operations at the leaves of the tree are independent of the other leaves' operations. Thus each leaf can be evaluated by a separate process and the results returned to the parent. We take the approach of allowing all processors to

traverse the QFT's tree until a point is reached where all processors can be utilized by a separate leaf. This approach does limit the number of processors to be a power of two but this is a reasonable limitation for today's hardware. Figure 10 shows how each processor is allocated to a branch of each subtree and the communication paths between each processor.

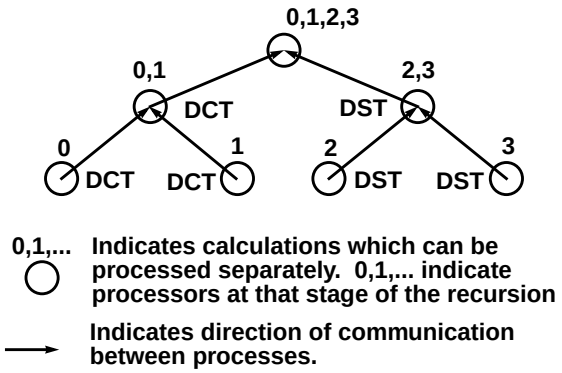


Figure 10 Parallel structure of the QFT and DITF algorithms

2.5. Decimation-in-Time-Frequency Algorithm

2.5.a. Sequential Form

The Decimation-In-Time-Frequency (DITF) algorithm uses both the Radix-2 Decimation-in-Time (DIT) and Decimation-in-Frequency (DIF) algorithms to form a new recursive algorithm. The DITF is based on the observation that the DIT algorithm has a majority of its complex operations towards the end of the computation cycle and the DIF algorithm has a majority towards the beginning. The DITF makes use of this fact by performing the DIT at the outset and then switching to a DIF to complete the transform as shown by the block diagram of Figure 11. Combining these algorithms comes at the cost of computing complex conversion factors at the point of switching. [11]

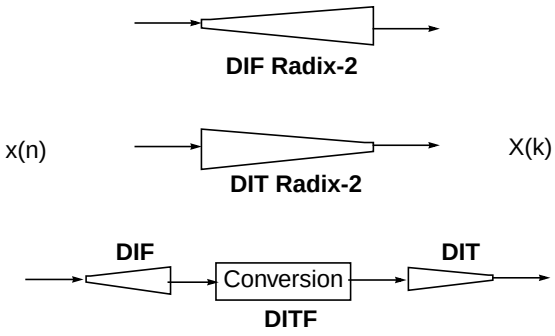


Figure 11 Block Diagram of DITF algorithm. Notice that the portions of the DIT and DIF with fewer computations are used by the DITF.

2.5.b. Parallel Form

Similar to the QFT, the DITF uses a recursive approach to the decomposition of the DFT by using both a recursive DIT and a recursive DIF. As with the QFT, the parallel structure of the DITF is described by Figure 10. Unlike the QFT, the DITF is highly communication dependent. Each leaf process must communicate at least once for each frequency point calculated. Thus, the cost of the tree structured approach for the DITF is too high. Instead the approach we take for the DITF is one of data splitting. Each processor is allocated N/p frequency data points to calculate, where p is the number of processors and assuming (N mod p) = 0.

3. Implementation

Implementation of the FFT algorithms consisted of two phases: sequential coding and parallel coding. The sequential FFT routines have been freely available in the public-domain for years. These FFT routines have been refined over a matter of months and years and each has been tweaked to the point of maximal efficiency.

The parallel implementations were developed directly from the sequential code. To do this, we first carefully analyzed each FFT routine for structural parallelism. Secondly, we analyzed the required communication costs of the parallel structure. Taking both of these into account, we designed the parallel routines to best take advantage of the algorithm's parallelism while minimizing communication costs.

3.1. Parallel Communication Tool

In order to use multiple processors, the data must first be distributed. Thus, there is a need for a method by which to transfer data between processes. The Message Passing Interface (MPI) provides a means by which to transfer data between processes conveniently. Since MPI is a communication library, it allows for inner-workings of the communication protocol to occur transparent to the calling code. MPI processes are "self-aware" in the sense that each is assigned a numeric rank at start-up that uniquely identifies it. The transfer of data using MPI is accomplished by making a call to one of the library's function. The calling code specifies the transfer by passing sending process's rank, the receiving process's rank, the data type, and the number of data elements to be passed. In addition to point-to-point communication, MPI supports collective and group communication. This allows for a reduction in the overhead in sending to a group (broadcast) and receive from a group (gather) of process. [12] [13]

Since the MPI libraries have been ported to many operating systems (OS)/achetectures, porting a parallel program to another does not require any change in the communication calls. MPI takes care of the hardware details, allowing the programmer to worry only about the software implementation. This allows for MPI code to transcend platforms.

4. Testing Procedures

Giving an objective evaluation of the algorithms requires an extensive knowledge of how each compares over a wide range of circumstances. In particular we want to collect statistics which relate directly to application constraints including factors such as memory, speed, data length, and hardware capabilities. We also desired test methods which give consistent and repeatable results.

4.1. Criteria

In choosing criteria by which to evaluate the various algorithms, consideration was given to the different constraints that would be imposed by a particular application. The criteria for benchmarking were computations speed, memory usage, number of processors available, input data size, code size, object size, and number of mathematical operations (adds, multiplications, and binary shifts). Computation speed was selected as the core criteria for comparison since the most efficient method is generally the most desirable one. However, since the amount of memory available is not static from machine to machine, memory was also included as a measure of efficiency. For many parallel applications, an increase in the number of processors available directly correlates to the speed-up. This is due to the fact that the number of processors limits the degree to which the data sequence can be decomposed. As was illustrated in theory section, decreasing the length of the data sequence for an $O(N^2)$ operation reduces the net number of operations which must be performed. The number of mathematical operations was included since it is directly related to the computation time and the hardware requirements. The additions and multiplications were broken in to floating point and integer operations due to the fact that floating point operations are much more costly in computation time than are integer operations. Input sequences of differing lengths were included to examine increase/ decrease in the overall cost of the communication.

4.2. Testing Methods

Evaluation of the above criteria involves many issues which are not readily apparent. Key amongst these is processor loading. One of our most important criteria is speed. Unfortunately this measurement cannot be completely decoupled from the loading of the hardware. To obtain consistent measurements of speed, we must eliminate the effects of fluctuations in processor loading. We accomplish this in two ways: test on unloaded processors and use an iterative testing method.

We ran our timing tests on relatively unloaded processors. These processors were loaded only by our programs and system operations. This eliminated the latency introduced when another user’s program is executing on our test machines.

An iterative approach to testing was also used to reduce the transients of processor loading. This method involved running each test for each algorithm for a large number of iterations. For instance, a compute intensive system operation may have been started in the middle of one test. This test taken alone would produce invalid and inconsistent results. On the other hand, performing this same test over a large number of iterations would average out that invalid result and produce repeatable results.

We also used a variety of methods for calculation of the other statistics. Computation speed is measured using the standard unix system utility ‘clock()’. For evaluation of memory usage, we developed a floating point class which has the feature of accumulating a count for each byte of memory allocated. This gives a

very efficient method of viewing the dynamic memory management. Counting of mathematical operations was accomplished in a manner similar to the memory counts.

5. Results

The observed computation time, shown in Figure 12 and Table 1, illustrates that execution time was of order Nlog(N) for all algorithms with respect to the data size The Radix-4 had the least computation time of all algorithms evaluated, but is limited to data sizes which are a power of four. The Radix-2 was somewhat slower than the more computationally efficient Radix-4. For the odd powers of two, the Split- Radix had the lowest computation time. Overall, the Split-Radix ranked between the Radix-2 and Radix-4. This was expected since the Split-Radix makes uses of both the Radix-2 and Radix-4. Therefore, the computation time for the Split-Radix is a weighted average of the computation time of its two sub-components. The FHT, which utilizes a Split-Radix topology, is somewhat slower than the Split-Radix FFT. This is due to additional computation time necessary to transform the Hartley coefficients to Fourier coefficients.

The ranking of the algorithms based on lowest memory usage, shown in Figure 13 and Table 2, follows the Ranking for least computation time with one exception. The DFT was the slowest, but uses the least amount of memory.

Algorithm	Speed N = 64	Speed N = 128	Speed N = 256	Speed N = 512	Speed N = 1024	Speed N = 2048	Speed N = 4096
DFT	61900	248600	996400	3994300	16009500	21280728	30576662
RADIX-2	1000	2100	4400	9400	19900	42100	90900
RADIX-4	700	---	3500	---	16000	---	72200
SRFFT	900	1800	3800	7900	16800	35400	76400
FHT	1000	2100	4600	9500	20500	43600	97800
QFT	1100	2500	5600	12300	26900	60700	129300
DITF	49400	165700	596900	2214000	6889400	7735873	11668681

Table 1: Computation Time for Each Algorithm (See Figure 12)

Algorithm	Data Type	Float Mults	Float Adds	Int Mults	Int Adds	Bin Shifts	Memory (bytes)
DFT	Real	2097152	2097152	0	1049600	0	8
	Complex	4194304	4194304	0	1049600	0	8
RADIX-2	Real	20480	30720	0	15357	1024	12308
	Complex	20480	30720	0	15357	1024	12308
RADIX-4	Real	15701	28842	336	8877	2738	4172
	Complex	15701	28842	336	8877	2738	4172
SRFFT	Real	4668	11722	494	11545	2335	12332
	Complex	10016	25488	502	12448	2937	12332
FHT	Real	9352	15006	0	4695	2123	4328
	Complex	18704	32056	0	8367	4246	16416
QFT	Real	4224	14722	8	34517	157	12288
	Complex	8448	31492	16	70058	316	24576
DITF	Real	48894	47163	0	16927	1	446464
	Complex	52996	51796	0	34878	2	462848

Table 2: Table of mathematical operations for each algorithm

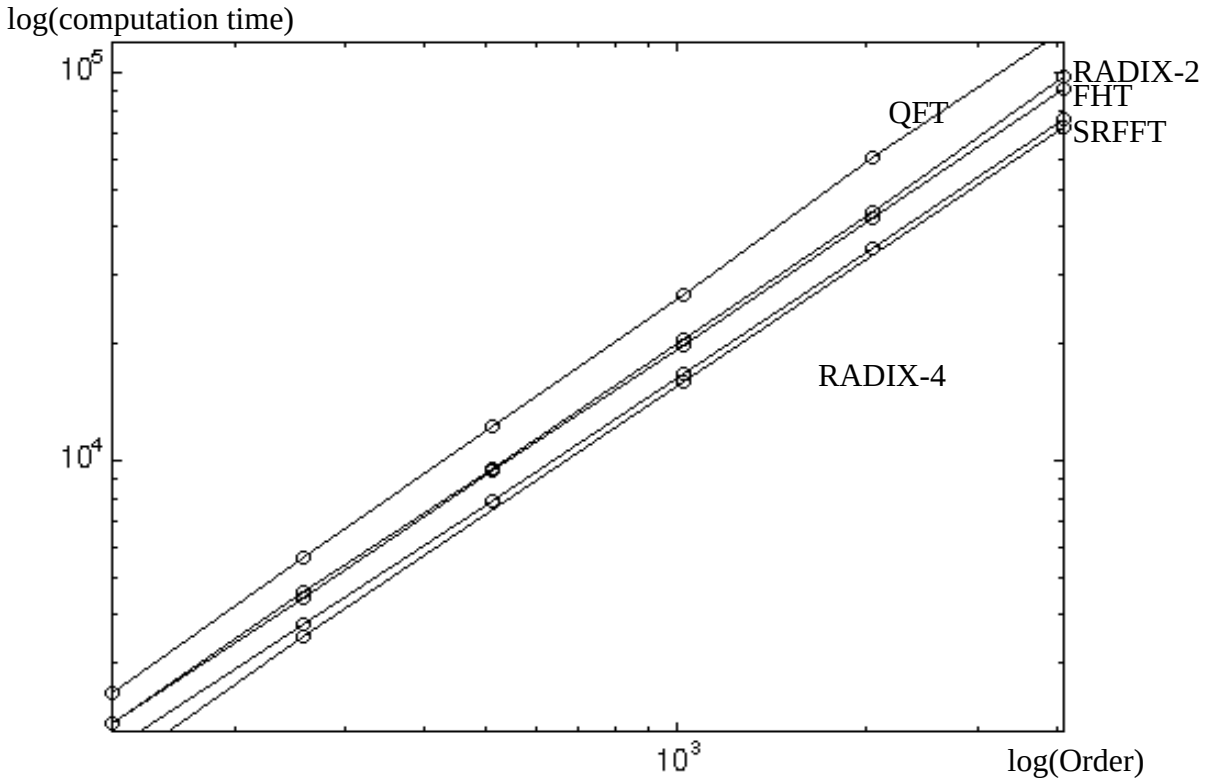


Figure 12 Log-Log Plot of Computation Time vs Order. Notice the N log(N) shape of the curves.

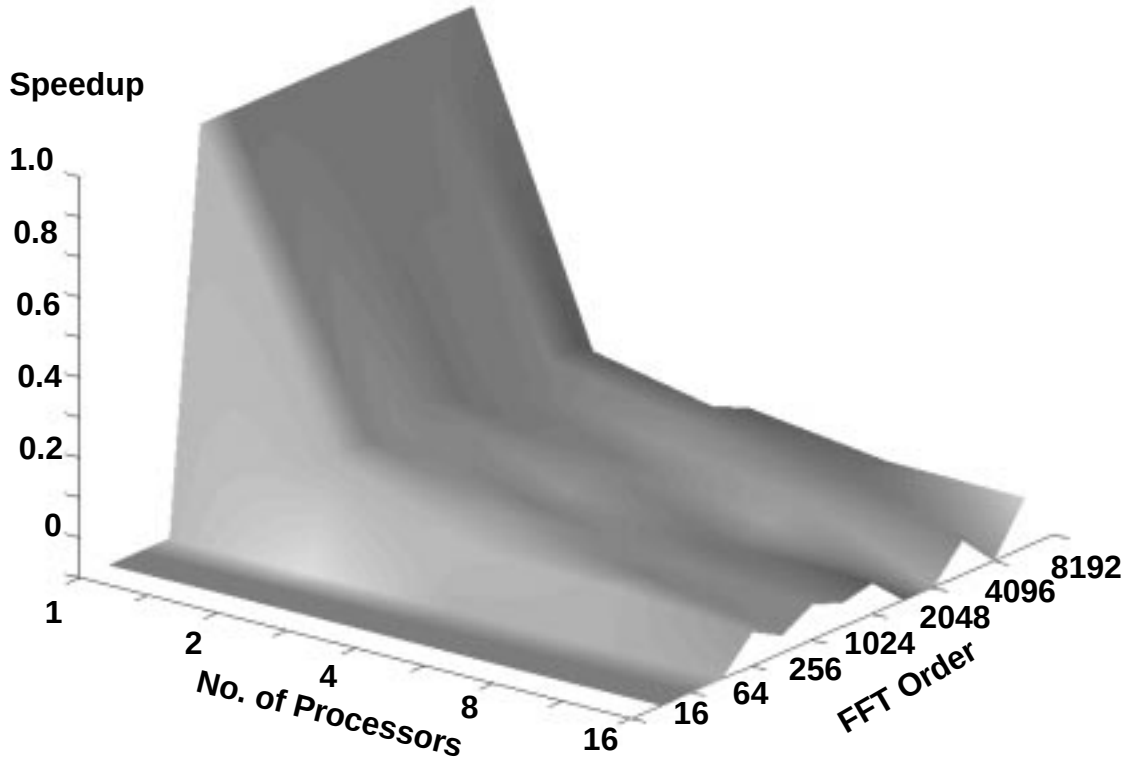
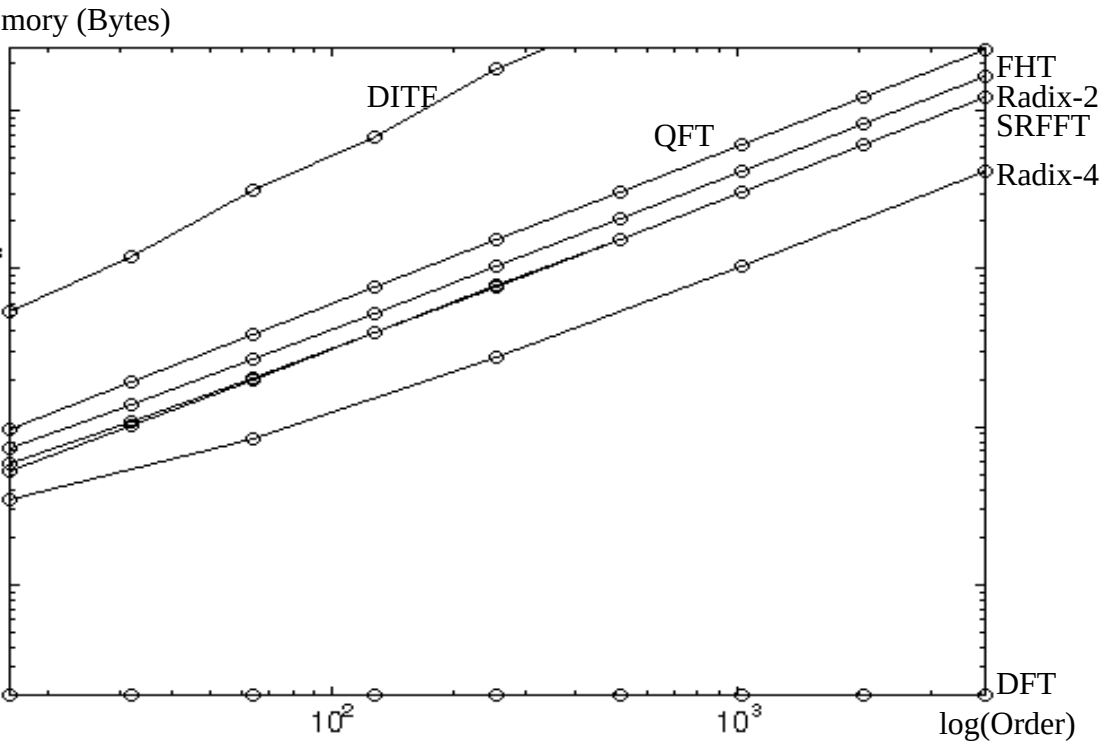


Figure 15 Typical results for the parallel algorithms. No speedup was achieved for the majority of the parallel algorithms



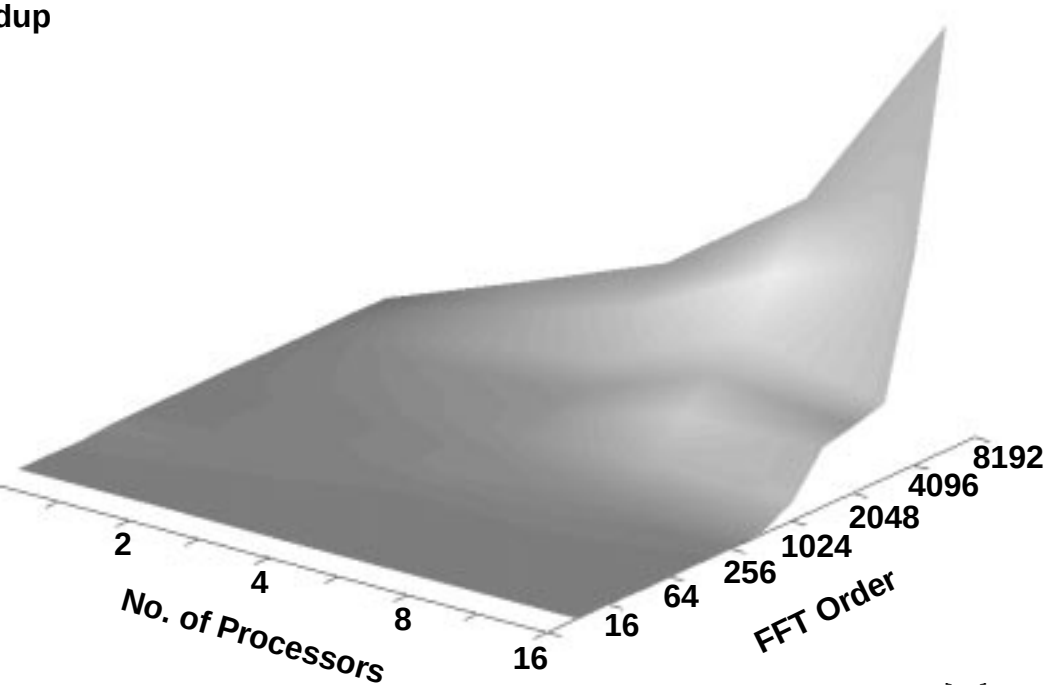
Log-Log Plot of Memory Usage vs Order for Each Algorithm.

alternative approach would be to distribute the data unevenly to allow more communication / execution overlap. That is, the master node would keep a larger portion of the data for itself. This would allow the

dup

inually process data while the other ess, and send back their portions. de is completing its computations at the processed data from other nodes n uneven data distribution approach re overlap, but still suffers from the den.

tion that eliminates the need for ta between processes is multi- reading uses several processes, all of to the same memory. Therefore, ne process transmitting its results to esult will be written to a memory processes can access. The processes o communicate with one another to ion between stages. That is, one o wait for another to complete the proceed. The limitation to this that the number of processes would number of processors on any one



4 Best case results for parallel algorithms (DFT). These results occur because data splitting was possible for the DFT.

6. Conclusions

Overall, the sequential algorithms proved to faster than their parallel counterparts. This can be accounted for by the additional time required to communicate between nodes. With the parallel algorithms, one node acts as a master distributing the data to other nodes. The master then becomes a slave which processes an equal portion of the data. The communication latency was such that the master node would complete its data processing cycle long before any of the other nodes since a send requires much less time than a receive. After completing its data processing cycle, the master node goes into a receive mode waiting to recollect the data processed by other nodes. This causes yet another delay due the imbalance of the send and receive times mentioned. The net result was sequential execution with waits inserted due to the communications latency.

In order for any parallel algorithm to be beneficial, there must be sufficient overlap between execution and communication. An even distribution of data provides for virtually no overlap. Given these results, an

7. Future Considerations

In our efforts thus far we have assembled the first unified collection of publicly available parallel FFT algorithms. The sequential code provides public-domain FFT code for a variety of algorithms under a single framework. The parallel code will give insight into the parallel coding of the various FFT algorithms.

Secondly, we have developed hard statistics for the complexity of the FFT algorithms. These statistics give the developer a clear picture of what will be required if a particular algorithm is used. Having these statistics takes the guess work out of development which was necessary to overcome the ambiguity of the big-O notation.

Perhaps, most importantly our work has laid the foundation for a class of “intelligent” programs which will automatically choose the best algorithm and execute it for a given set of user constraints. As the hardware available becomes more diverse, this type of software is becoming a necessity. This constraint driven program will also be a beneficial tool to the developer because it will give a quick way to test performance under changing design constraints.

There are various possibilities that must be explored

before the constraint driven software will become a reality. Foremost among these is to re-evaluate the parallel structure of the FFT code. Trying to cast the "tight" sequential code into a parallel form is not likely to be the best option. We should redevelop the implementation to use the parallel structure of the algorithm more efficiently.

Also, we must explore other parallel processing techniques. We have seen that the major impedance to parallel processing of FFTs is communication costs. We will explore other techniques (such as shared memory) which allow us to perform the FFTs with a reduced number of communication calls.

8. Acknowledgments

We would like to thank the following persons for their support: Dr. Joseph Picone, Dr. Tony Skjellum, and the members of each of their groups. Most importantly we thank Aravind Ganapathiraju for his leadership and interest in our efforts.

9. References

1. Oppenheim, Alan V., Ronald W. Shafer. *Discrete-Time Signal Processing.* Prentice Hall. Englewood Cliffs, New Jersey 1989. pp 587-610.

2. Bracewell, Ronald N. *The Fourier Transform and Its Applications, Second Edition.* McGraw-Hill Book Company. New York, 1978. pp356-381.

3. Proakis, John G. and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications, Third Edition.* Prentice Hall, Upper Saddle River, New Jersey, 1996. pp230-256, 394-494.

4. Roberts, Richard A., Clifford T. Mullis. *Digital Signal Processing.* Addison Wesley, Reading, Massachusetts, 1987. pp 148-162.

5. Blahut, Richard E. *Fast Algorithms for Digital Signal Processing.* Addison Wesley, Reading, Massachusetts, 1985. pp 114-152, 240-280.

6. Tatyana D. Roziner, et. al., "Fast Fourier Transforms Over Finite Groups by Multiprocessor Systems," *IEEE Trans. on ASSP*, vol. 38, no. 2, February 1990. pp 226-239.

7. P. Duhamel and H.Hollomann, "Split radix FFT algorithm," *Electron. Lett.*, vol. 20, pp. 14-16, Jan. 1984.

8. R. Bracewell. *The Hartley Transform* Oxford, England: Oxford Press, 1985, chapter 4.

9. "Implementing 2-D and 3-D Discrete Hartley Transforms on a Massively Parallel SIMD Mesh Computer." *Technical Report CS-TR-95-01*, University of Central Florida, Orlando, FL.

10. H. Guo, G.A. Sitton, C.S. Burrus. "The Quick Discrete Fourier Transform." *ICASSP94 Digital Signal Processing.* vol III. Institute for Electrical and Electronics Engineers. pp. 445-447, 1994.

11. Saidi, Ali. "Decimation-In-Time-Frequency FFT Algorithm." *ICASSP94*
Digital Signal Processing. vol III. Institute for Electrical and Electronics Engineers.
pp. 453-456, 1994.

12. "Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report Computer Science Department" *Technical Report CS-94-230*, University of Tennessee, Knoxville, TN, May 5 1994.

13. Snir, Marc. et. al. *MPI: The Complete Reference.* The MIT Press, Cambridge,Massachusetts, 1996.

14. Fox, Geoffrey C. et. al. *Solving Problems On Concurrent Processors*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.