Parallel Fast Fourier Transforms for Large 1D Arrays

C. A. de Koning[1]

December 7, 2000

[1]Advisor: W. Matthaeus

1 Introduction

It is hard to think of any other branch of mathematics that has as many practical applications as Fourier analysis. The mathematics of Fourier analysis has been used for years to study physical phenomena from a wide variety of scientific and engineering fields; for example, heat transfer, wave propagation, circuit analysis, vibrations, optical systems, and spectral analysis.

Fourier analysis contains three distinct but notably similar fields. These three fields are the Fourier series, the Fourier transform, and the discrete Fourier transform. From a physics point of view, the discrete Fourier transform, or DFT, generally arises as a computer tool that is used in the calculation of a continuous Fourier transform.

The definition of a Fourier transform states that

$$B(z) = \int_{-\infty}^{\infty} b(k)\, \mathrm{e}^{\imath k z} dk; \tag{1}$$

that is, the Fourier transform associates a function in configuration space, $B(z)$, with a function in wave-number space, $b(k)$. The inverse Fourier transform equals

$$b(k) = \frac{1}{2\pi} \int_{-\infty}^{\infty} B(z)\, \mathrm{e}^{-\imath k z} dz. \tag{2}$$

2 Discrete Fourier Transform

With this brief introduction to the Fourier transform, let us see how the discrete Fourier transform emerges as a natural computational technique. For all computer implementations of the forward Fourier transform, Eq. 1, the input function, $b(k)$, must be sampled and stored as a sequence of discrete values, $b_0, b_1, \ldots, b_{N-1}$. Consequently, from a practical viewpoint, the input function, $b(k)$, must be assumed to equal zero outside some finite interval. In mathematical terminology, the function $b(k)$ has bounded support on the interval $-\frac{K}{2} < k < \frac{K}{2}$. If $b(k)$ is not naturally zero outside this interval, it is necessary to truncate the function. Although this will introduce errors, it is unavoidable. The sampling scheme mentioned above is obtained by dividing the interval $[-\frac{K}{2}, \frac{K}{2}]$ into N subintervals of length $\Delta k = K/N$. Consequently, $k_\ell = \ell \Delta k$ for $\ell = -\frac{N}{2} \ldots \frac{N}{2}$. Similarly, we impose the condition that the output function, $B(z)$, have bounded support on the interval $-\frac{L}{2} < z < \frac{L}{2}$. By subdividing the interval $[-\frac{L}{2}, \frac{L}{2}]$ into N subintervals of length $\Delta z = L/N$ with $z_q = q\Delta z$ for $q = -\frac{N}{2} \ldots \frac{N}{2}$, we obtain the sampling scheme for the output function. It is important to note that the choice of Δk, Δz, and N is not an arbitrary process; these quantities are related by the so-called reciprocity relations:

$$KL = 2\pi N, \tag{3}$$

$$\Delta k \Delta z = \frac{2\pi}{N}, \tag{4}$$

Keeping these ideas in mind, the forward Fourier transform, Eq. 1, can be written as

$$\int_{-\infty}^{\infty} b(k)\, \mathrm{e}^{\imath kz} dk = \int_{-K/2}^{K/2} b(k)\, \mathrm{e}^{\imath kz} dk, \tag{5}$$

because $b(k)$ has bounded support. Furthermore,

$$\int_{-K/2}^{K/2} b(k)\, \mathrm{e}^{\imath kz} dk \approx \sum_{\ell=-\frac{N}{2}}^{\frac{N}{2}-1} b(k_\ell)\, \mathrm{e}^{\imath k_\ell z_q}\, \Delta k. \tag{6}$$

Finally, by making use of the inherent symmetries in the complex exponential function, we obtain the standard form for the DFT,

$$B(q) = \sum_{\ell=0}^{N-1} b_\ell \, \mathrm{e}^{2\pi\imath\ell q/N}, \tag{7}$$

where

$$\begin{aligned} b_\ell &\equiv b(k_\ell)\Delta k, \\ B(q) &= B(z_q). \end{aligned} \tag{8}$$

The inverse DFT is given by

$$b_\ell = \frac{1}{N} \sum_{q=0}^{N-1} B(q) \, \mathrm{e}^{-2\pi\imath\ell q/N}. \tag{9}$$

The above analysis, showing how the DFT emerges from the Fourier transform, is admittedly sketchy. A rigorous derivation of the DFT, which naturally leads to the Shannon sampling theorem and the Nyquist rate, can be found in Weaver [1989]. In addition, a practical guide outlining the proper use of the DFT is available from Briggs and Henson [1995].

To estimate the computation time of the DFT, define the complex number

$$W \equiv \mathrm{e}^{2\pi\imath/N}; \tag{10}$$

therefore, Eq. 7 equals

$$B(q) = \sum_{\ell=0}^{N-1} b_\ell \, W^{\ell q}. \tag{11}$$

In other words, the vector b_ℓ is multiplied by a matrix whose $(\ell, q)^{\mathrm{th}}$ element is $W^{\ell q}$. Since the matrix multiplication requires N^2 complex multiplications, the DFT has a complexity of $\mathcal{O}(N^2)$. The Ws are generally called *twiddle factors*.

3 Fast Fourier Transform

The fast Fourier transform, or FFT, is a computational technique of sequentially combining progressively larger weighted sums of data samples so as to produce the DFT. By exploiting symmetries in the DFT, the FFT significantly reduces the number of operations that are necessary to calculate the DFT. Note that the FFT is not a new transform; rather, it is an efficient algorithm for calculating the DFT.

The existence of an FFT algorithm became generally known in 1965, from the work of J. W. Cooley and J. W. Tukey, who in turn had been prompted by R. L. Garwin of IBM Yorktown Heights Research Center. Retrospectively, we now know that a few clever individuals independently discovered, and in some cases implemented, the FFT as much as 20 years prior to 1965. In fact, according to Press et al. [1989], one of the earliest discoveries of the FFT was by Danielson and Lanczos in 1942.

Several 1D FFT algorithms are available; namely,

- Cooley-Tukey algorithm;
- Pease algorithm;
- Stockham autosort algorithm;
- transpose algorithm;
- higher radix algorithms.

All of these algorithms compute the same result; namely, the DFT. However, they differ in order that intermediate computations are stored in memory. It is this difference that makes one or the other more suitable for a particular architecture or application [Swartztrauber, 1984].

The Cooley-Tukey algorithm, which is the most common, does all intermediate computations in place; that is, it does not need a separate array to store intermediate computations. The Pease and Stockham algorithms are not done in place. This may pose a problem in a situation where available computer memory is limited, either because there is only a small amount of memory on the computer or, more likely, because the problem size is large.

The computational phase in the Cooley-Tukey and Pease algorithms computes the transform in a permuted order. When an ordered transform is required, a separate order phase, called bit-reversal, is used to unscramble the coefficients. This order phase is relatively time consuming; as a result, the Stockham autosort algorithms are valuable, because they compute the transform in the proper order without an explicit order phase. This property is particularly important on parallel computers, where sorting and transfer of data between different processors can be just as expensive, if not more so, than arithmetic operations.

The choice between the Cooley-Tukey and transpose algorithm depends upon communication related parameters of the machine in use, the number of processors available, and the size of the problem to be solved [Gupta and Kumar, 1993]. (See Sect. 7 for more details.)

All the above algorithms are generally assumed to be radix-2 FFTs; that is, the number of data points N, can be written as $N = 2^n$. In a higher radix, or mixed radix, FFT, the base can be an arbitrary number.

3.1 Cooley-Tukey Algorithm

The starting point for the Cooley-Tukey algorithm is Eq. 7,

$$B(q) = \sum_{\ell=0}^{N-1} b_\ell \, \mathrm{e}^{2\pi\imath\ell q/N}.$$

After dividing the above sum into even, $\ell = 2m$, and odd, $\ell = 2m+1$ terms, we get

$$B(q) = \sum_{m=0}^{N/2-1} b_{2m} \, \mathrm{e}^{2\pi\imath q(2m)/N} + \sum_{m=0}^{N/2-1} b_{2m+1} \, \mathrm{e}^{2\pi\imath q(2m+1)/N}. \tag{12}$$

After a simple rearrangement, Eq. 12 equals

$$B(q) = \sum_{m=0}^{N/2-1} b_{2m} \, \mathrm{e}^{2\pi\imath qm/(N/2)} + \mathrm{e}^{2\pi\imath q/N} \sum_{m=0}^{N/2-1} b_{2m+1} \, \mathrm{e}^{2\pi\imath qm/(N/2)}. \tag{13}$$

Finally, let $M = N/2$; therefore,

$$B(q) = \sum_{m=0}^{M-1} b_{2m} \, \mathrm{e}^{2\pi\imath qm/M} + \mathrm{e}^{2\pi\imath q/N} \sum_{m=0}^{M-1} b_{2m+1} \, \mathrm{e}^{2\pi\imath qm/M}. \tag{14}$$

This process can be repeated until each term consists of a pair of a one-point DFTs,

$$\tilde{B}(q) = \sum_{m=0}^{0} b_m \, \mathrm{e}^{2\pi\imath qm} + \mathrm{e}^{\pi\imath q} \sum_{m=0}^{0} b_{2m+1} \, \mathrm{e}^{2\pi\imath qm}. \tag{15}$$

or

$$\begin{aligned} \tilde{B}(\text{even}) &= b_0 + b_1 \\ \tilde{B}(\text{odd}) &= b_0 - b_1. \end{aligned} \tag{16}$$

Notice that the output sequence of a one point DFT equals the input sequence. The computational flow indicated in Eq. 16 is shown graphically in Fig. 1.

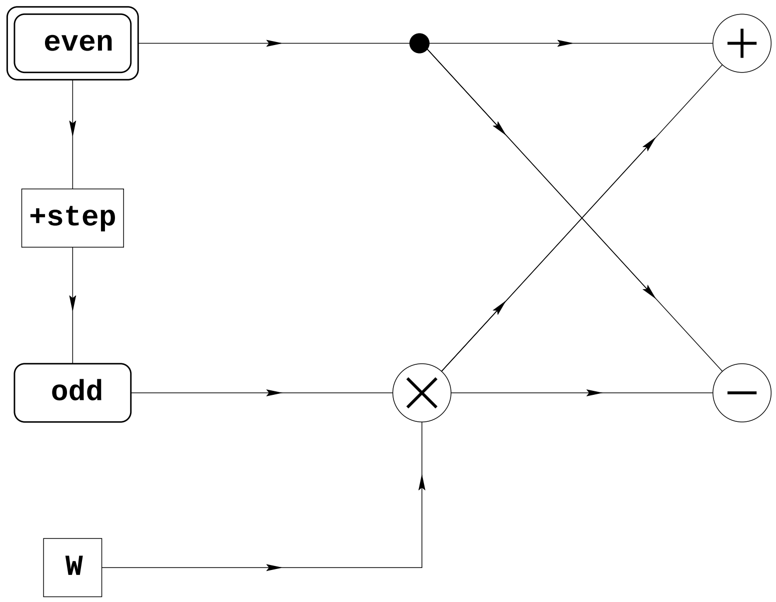

Figure 1: Two-point sequential FFT computational flow.

The heart of the Cooley-Tukey algorithm deals with sequentially recombining progressively larger weighted sums of data samples so as to produce the DFT. Initially, there exists N sets of one-point DFTs. On the first iteration, the Cooley-Tukey algorithm combines $N/2$ pairs of one-point DFTs to make $N/2$ sets of two-point DFTs. This step is indicated in Eq. 16. On the second iteration, the Cooley-Tukey algorithm combines $N/4$ pairs of two-point DFTs to make $N/4$ sets of four-point DFTs. This process continues till we have 2 $N/2$-point DFTs, which are combined to make an N-point DFT. This final step is indicated Eq. 14. The Cooley-Tukey algorithm evidently consists of $\log_2 N$ iterations. In addition, notice that the combination of any pair of data samples involves the calculation of a twiddle factor.

The computational flow for the entire FFT is often depicted in a butterfly-

Figure 2: Sixteen-point sequential FFT computational flow.

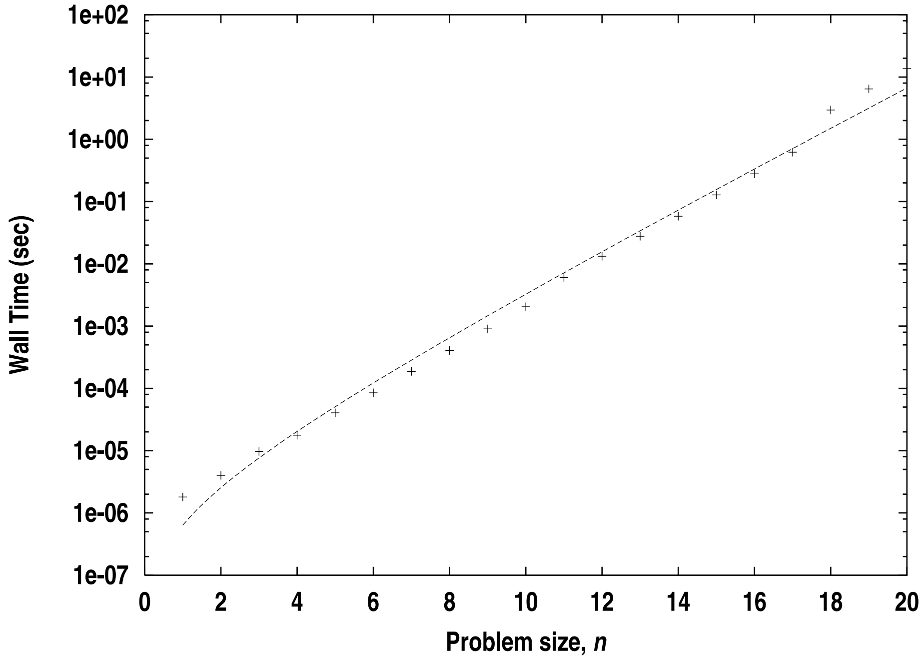

Figure 3: Total computational wall-time for a sequential FFT.

diagram, such as Fig. 2 for a 16-point transform.

According to Fig. 1, for each pair of data samples there are four basic arithmetic operations, plus the calculation of one twiddle factor. In addition, Fig. 2 indicates that there are $N/2$ pairs of data samples calculated on each iteration. Since the Cooley-Tukey algorithm consists of $\log_2 N$ iterations, the sequential time complexity for an *unordered* FFT equals

$$\begin{aligned} T_{comp} &= t_c\Big(1 \otimes + 3 \oplus + 1 \text{ twiddle}\Big)\frac{N}{2}\log_2 N \\ &= t_c\,\frac{5}{2}\,N\log_2 N, \end{aligned} \tag{17}$$

where t_c is the time for one arithmetic operation. Figure 3 shows experimental results for the total computational time, T_{comp}, for sequential FFTs of various sizes, n, where the total number of data points equals $N = 2^n$. The experimental wall-time was measured by repeatedly cycling through the Cooley-Tukey algorithm. The total time for an experimental run is on the order of minutes, where the number of cycles through the algorithm equals several thousand for large N, up to several million for very small N. The best fit curve to this data indicates that $t_c = 1.27 \times 10^{-7}$ sec for the CIS DEC Alpha cluster. This value is in sharp contrast to the experimental time for a basic arithmetic calculation, such as add or multiply, which, on the CIS DEC Alpha cluster, equals $t_c \sim 3 \times 10^{-9}$ sec. In fact, even a more involved computation, such as finding z^q or e^z for complex z, obtains $t_c \sim 3 \times 10^{-9}$ sec.

4 General Computation Issues

A basic FFT program contains the following elements:

- Input and output of data sequence.
- Reorder data via bit reversal.
- Calculate the DFT via the Cooley-Tukey, or some other, algorithm.

Although all aspects of the program are important, from a mathematical perspective, the third point is the most interesting.

It is worth noting that an input sequence of double complex numbers requires $16N$ bytes of memory; therefore, a moderately large data sequence of $N = 2^{20}$ requires more than 16 MB of memory and a data sequence of $N = 2^{22}$ requires 67 MB of memory. Clearly, with this much data, the tasks of input, output, and reordering are candidates for parallelization. Nevertheless, this paper will focus on computational techniques for performing the mathematical aspects of the FFT.

4.1 Calculating the Twiddle Factors

An important step in the Cooley-Tukey algorithm is the calculation of the twiddle factors. The two most straightforward methods for calculating the twiddle factors are to calculate them in place or to obtain them from a pre-calculated look-up table. There are pros and cons for each method.

On the positive side, calculating the twiddle factors in place requires the minimum amount of memory; the problem is that it takes the maximum amount of time.

On the other hand, if the twiddle factors are obtained from a look-up table, memory usage increases. Total computation time can be reduced with this method, in particular if multiple FFTs of the same length are performed.

On iteration j, the twiddle factors have the form

$$W_j(q) = \exp\left(\frac{2\pi\imath q}{2^j}\right). \tag{18}$$

Consequently, because of the symmetries in the complex exponential, there are 2^{j-1} unique twiddle factors that have to be calculated on iteration j. Summing over all iterations, we find that the size of the twiddle table equals

$$N_{twiddle} = \sum_{j=1}^{\log_2 N} 2^{j-1} = N - 1, \tag{19}$$

or $16(N-1)$ bytes of memory for double complex numbers. Once again, for a moderate problem of $N = 2^{20}$, the look-up table requires 16 MB of memory; therefore, a look-up table is *not* a wise use of memory for large 1D data sets.

Equation 18 indicates that a compromise solution exists. Notice that

$$W_j(q) = W_J^q \tag{20}$$

where

$$W_J \equiv \exp\left(\frac{2\pi\imath}{2^J}\right); \tag{21}$$

therefore, we can store W_J in a look-up table and raise it to the appropriate power in the FFT algorithm. Since the Cooley-Tukey FFT algorithm consists of $\log_2 N$ iterations, the look-up table requires only $16 \log_2 N$ bytes of memory for double complex numbers. Even for an astronomical problem size of $N = 2^{100}$ data points, the look-up table would only require 1.6 KB.

Up to this point, we have assumed the use of double complex numbers. The justification for this choice is related to the twiddle factors. For large N, a single precision algorithm cannot distinguish between W^q and W^{q+1}; therefore, it is necessary to use double precision. For example, let $N = 2^{20}$ and $q = 1$. Then

$$\begin{aligned} \mathrm{real}(W^1) &= 0.99999999998204725, \\ \mathrm{real}(W^2) &= 0.99999999992818922. \end{aligned} \tag{22}$$

In a single precision algorithm, the computer would obtain $\mathrm{real}(W^1) = \mathrm{real}(W^2) = 1$.

5 Parallel Computation Issues

Given that the FFT is one of the most important algorithms in computational science (it has been estimated that up to 40% of Cray cycles are dedicated

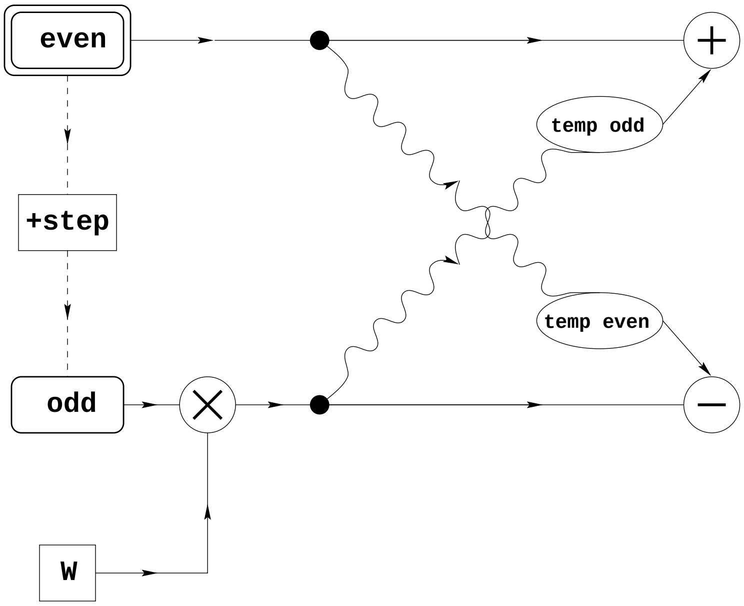

Figure 4: Two-point parallel Cooley-Tukey FFT computational flow.

to computing the FFT [Gatlin and Carter, 2000]), it is obviously worthwhile to parallelize the code. Since the FFT has a sequential time complexity of $\mathcal{O}(N \log N)$, the ideal cost-optimal parallel time complexity equals $\mathcal{O}(\log N)$ with N processors.

Zooming in on a pair of data samples, the parallel computational flow diagram corresponding to Fig. 1, is shown in Fig. 4. The parallel computational flow diagram for the entire Cooley-Tukey algorithm, corresponding to Fig. 2, is shown in Fig. 5.

There are several points worth noting about the parallel Cooley-Tukey FFT. First, when the number of processors, $P = 2^p$, and the number of data points, $N = 2^n$, with $P \leq N$, then the parallel Cooley-Tukey algorithm has two distinct phases; in p of the iterations the processors need to communicate.

Figure 5: Sixteen-point parallel FFT computational flow on four processors.

For the remaining $n-p$ iterations, the data samples to be combined are all on the same processor. Furthermore, during the communication phase, all N/P data samples required by a particular processor are available to it from a single processor. Finally, the computational complexity of the parallel algorithm is equal to the computational complexity of the sequential algorithm.

Each communication stage involves the transfer of N/P words between pairs of processors. Since there are $\log_2 P$ iterations with communication, the time complexity for communication equals

$$T_{comm} = \Big(t_w \frac{N}{P} + t_\ell\Big)\, 2 \log_2 P, \tag{23}$$

where t_w is the time to exchange one word between processors. In this paper we will always assume that a word is 16 bytes long, corresponding to a double complex number. The latency, which measures the start-up time for communication, equals t_ℓ; in other words, the latency measures the time it takes to send a word of zero length. The time complexity for computation in the parallel Cooley-Tukey FFT equals

$$\begin{aligned} T_{comp} &= t_c\Big(1 \otimes + 3 \oplus + 1 \text{ twiddle}\Big)\frac{N}{2P}\log_2 N \\ &= t_c\,\frac{5}{2}\frac{N}{P}\log_2 N. \end{aligned} \tag{24}$$

Note that the above time complexity analysis is for an *unordered* parallel Cooley-Tukey FFT. The total time complexity for the parallel unordered Cooley-Tukey FFT equals $T_{par} = T_{comp} + T_{comm}$.

The speed-up, S, a measure of relative performance between a parallel

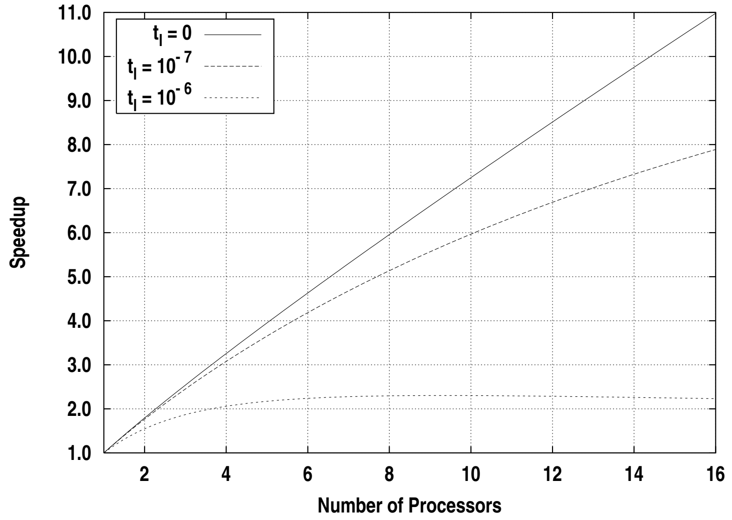

Figure 6: Effect of latency on the theoretical performance of a Cooley-Tukey FFT for $N = 2^7$ and $t_w = t_c$.

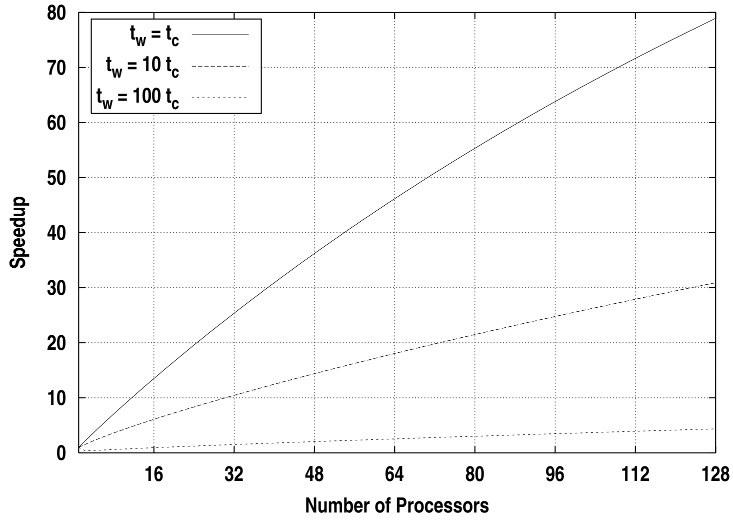

Figure 7: Effect of the communication rate on the theoretical performance of a Cooley-Tukey FFT for $N = 2^{20}$.

system and a single processor system equals

$$S = \frac{T_{seq}}{T_{par}} = \frac{P}{1 + \frac{4}{5}\left[\frac{t_w}{t_c} + \frac{t_\ell}{t_c}\frac{P}{N}\right]\frac{\log_2 P}{\log_2 N}}. \tag{25}$$

Equation 25 indicates that there are a few factors will effect overall performance. The latency will only effect performance if $P \sim N$. This is shown in Fig. 6 for $N = 2^7$. The primary influence on parallel performance is due to the ratio of the communication time to computation time, t_w/t_c. The effect of the communication rate is shown in Fig. 7 for $N = 2^{20}$. Note that because $N \gg P$, latency is not important.

5.1 Communication

A parallel implementation of the Cooley-Tukey FFT uses a specific form communication known as bidirectional communication. In other words processor **A** sends to processor **B** and processor **B** sends to processor **A**. In bidirectional communication, deadlock can be a problem if both processors send large blocks of data at approximately the same time, as indicated in Fig. 8. Deadlock occurs because processor **A** has possession of the buffer. Processor **B** wants to use the buffer, but can't till it empties the buffer, and it can't empty the buffer till it uses the buffer.

MPI offers several alternative methods for safe communication:

- Alternate order of send and receive on processor pairs; that is, processor **A** first calls `MPI_SEND` and then `MPI_RECV` but processor **B** first calls `MPI_RECV` and then `MPI_SEND`.

- Use the combined send and receive routine: `MPI_SENDRECV`.

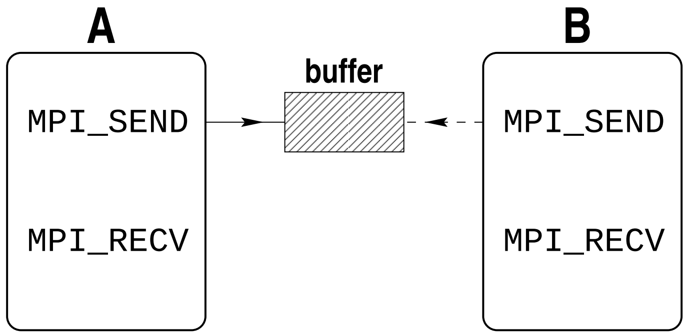

Figure 8: An example of deadlock during bidirectional communication.

- Use nonblocking or immediate routines: `MPI_ISEND`, `MPI_IRECV` along with the separate routines `MPI_WAIT` or `MPI_TEST` to detect completion.

5.2 Load Balancing

The parallel Cooley-Tukey algorithm does not achieve maximal load balancing because only the "odd" processors need to compute the twiddle factors (see Fig. 4). If the "odd" processor computes the twiddle factors during a nonblocking communication in which the even processor sends its data to the odd processor, then the effect of the load imbalance will depend on the communication to computation speed. In other words, if the communication speed is slow relative to the computation speed, then the "odd" processor may finish computing all the twiddle factors before the nonblocking communication is complete.

In fact, this will almost always be the case, since

$$\tau_{comm} = t_w \frac{N}{P} + t_\ell \tag{26}$$

and

$$\tau_{comp} = 2t_c \frac{N}{P}. \tag{27}$$

Therefore

$$\frac{\tau_{comm}}{\tau_{comp}} = \frac{1}{2}\frac{t_w}{t_c} + 2\frac{P}{N}\frac{t_\ell}{t_c}. \tag{28}$$

For large data sets, the latency term is negligible, and so

$$\frac{\tau_{comm}}{\tau_{comp}} \sim \frac{1}{2}\frac{t_w}{t_c}, \tag{29}$$

where $t_w/t_c > 1$ for most present day systems.

6 Results

Actual results for this project are sketchy, at best. A parallel 1D Cooley-Tukey algorithm does exist and it reproduces results predicted by a continuous Fourier transform! For example, it can be shown, that if

$$b(k) = \frac{1}{1+k^2}, \tag{30}$$

then

$$B(z) = \pi \mathrm{e}^{-|z|}. \tag{31}$$

The parallel FFT results for the sequence based on Eq. 30 are shown in Fig. 9.

The problem is that the parallel FFT program refuses to run on more than 2 processors; whether this is the fault of the programmer or the CIS DEC Alpha cluster remains to be seen. Furthermore, the measured speed-up

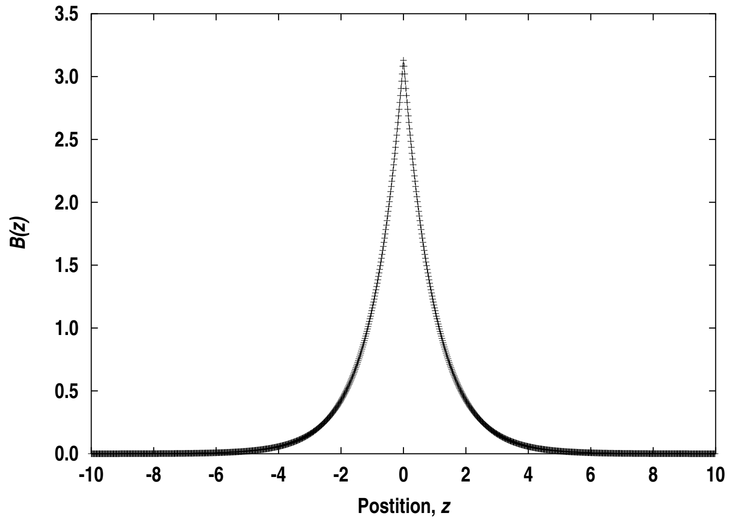

Figure 9: Verifying the accuracy of the parallel 1D Cooley-Tukey FFT.

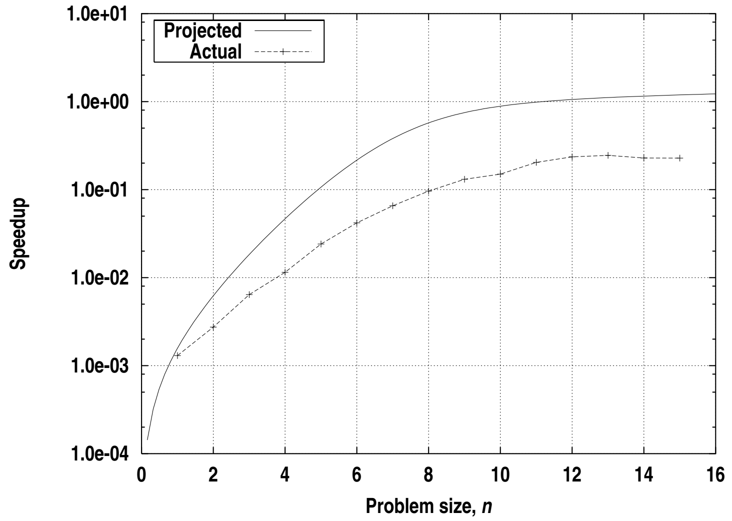

Figure 10: Comparing actual and projected speed-up for two processors on the CIS DEC Alpha cluster; performance parameters for the cluster are given in the text.

on two processors for varying problem size does not agree with the projected speed-up. Projected speed-up is given by Eq. 25; for the CIS DEC Alpha cluster, the following parameters were experimentally measured:

- $t_c = 1.27 \times 10^{-7}$ sec,
- $t_w = 1.60 \times 10^{-6}$ sec,
- $t_\ell = 2.00 \times 10^{-4}$ sec.

Results comparing the actual and projected speed-up are shown in Fig. 10. Although the measured speed does not equal the projected speed-up, the results are similar; it is possible that the measured parameters are not correct.

The remaining figures show the projected speed-up for a variety of different clusters available at the University of Delaware. Performance parameters for each system are indicated in the figure. Actual performance of an FFT will likely be one or two orders of magnitude below its maximum speedup, unless the computation is formulated in a way that takes advantage of the complex architectural features [Swartztrauber, 1984].

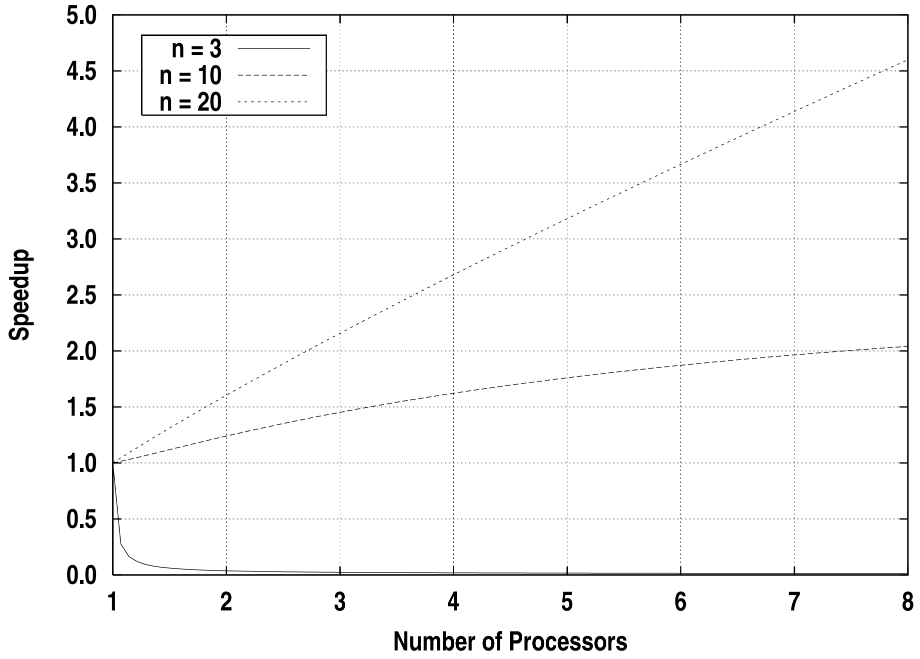

Figure 11: Projected 1D Cooley-Tukey FFT performance on the DEC Alpha cluster, where the computation rate equals 3.85 MFLOPS; the bandwidth equals 10 MB/s or $\frac{t_w}{t_c} = 6.15$ for double complex words; and the latency equals 200 μs or $\frac{t_\ell}{t_c} = 7.7 \times 10^2$.

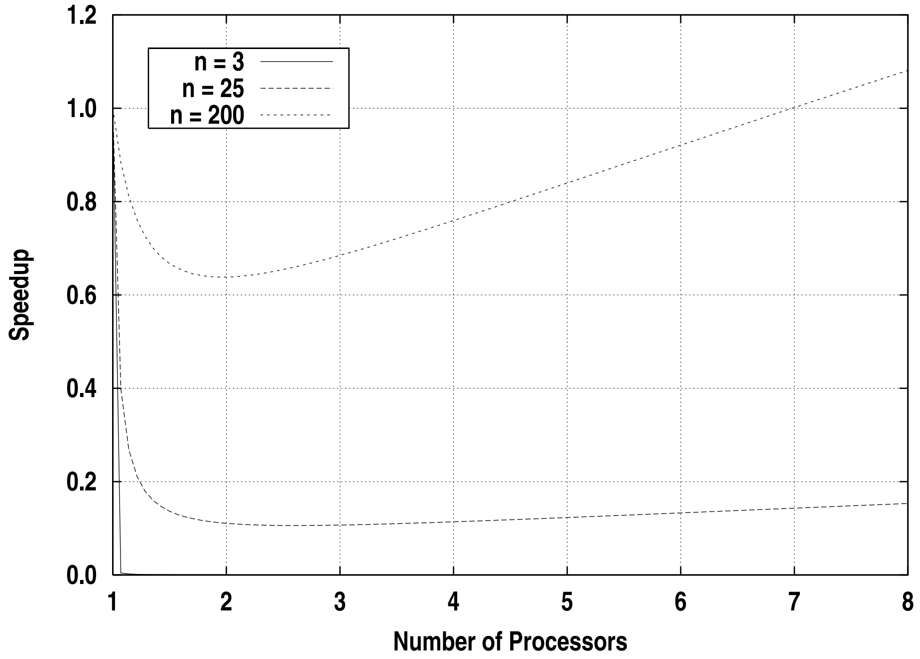

Figure 12: Projected 1D Cooley-Tukey FFT performance on the DEC Alpha cluster, where the computation rate equals 333 MFLOPS; the bandwidth equals 10 MB/s or $\frac{t_w}{t_c} = 533$ for double complex words; and the latency equals 200 μs or $\frac{t_\ell}{t_c} = 6.7 \times 10^4$.

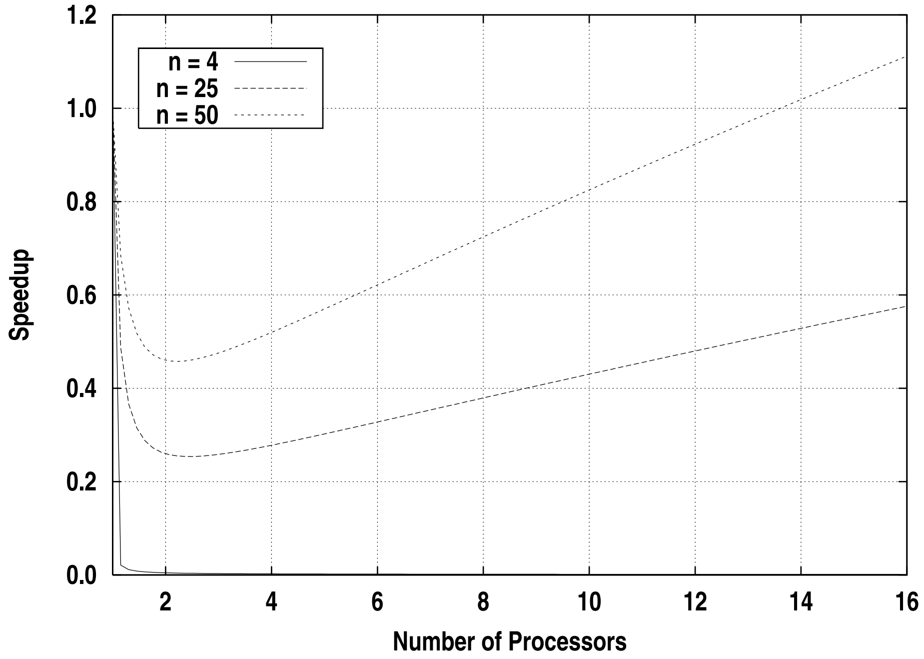

Figure 13: Projected 1D Cooley-Tukey FFT performance on Wulfie, where the computation rate equals 153 MFLOPS; the bandwidth equals 12 MB/s or $\frac{t_w}{t_c} = 209$ for double complex words; and the latency equals 100 μs or $\frac{t_\ell}{t_c} = 1.5 \times 10^4$.

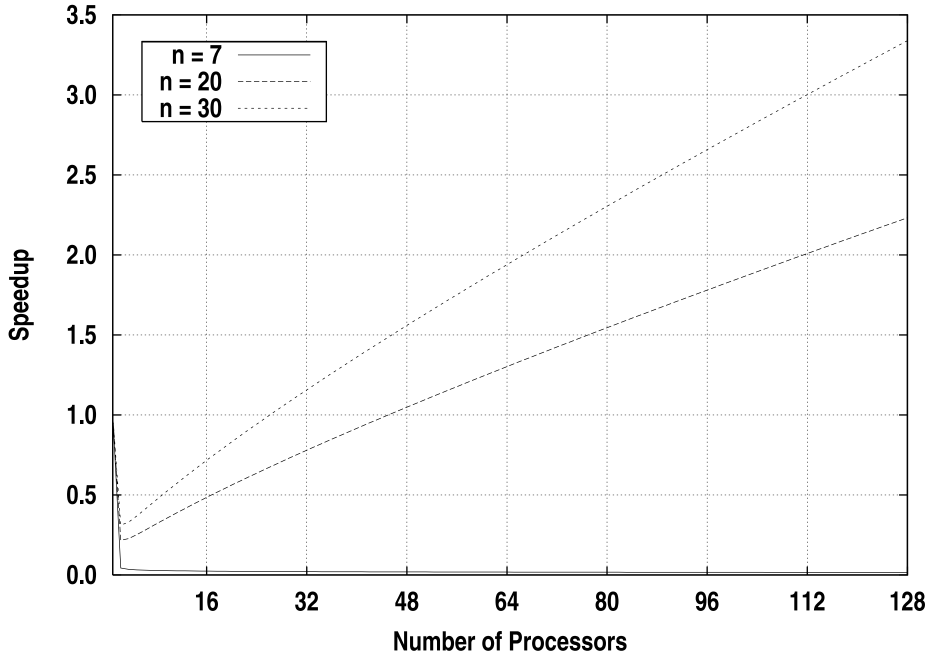

Figure 14: Projected 1D Cooley-Tukey FFT performance on Samson, where the computation rate equals 1000 MFLOPS; the bandwidth equals 80 MB/s or $\frac{t_w}{t_c} = 200$ for double complex words; and the latency equals 10 μs or $\frac{t_\ell}{t_c} = 10^4$.

7 Future Work

Several authors suggest that a parallel transpose FFT performs better than the Cooley-Tukey FFT, especially for very large N [see, for example Swartztrauber, 1987; Ashworth and Lyne, 1988; Bailey, 1990; Gupta and Kumar, 1993]. Initial findings suggest that this is correct, but much more work is required to verify the initial projections.

The time complexity analysis for the parallel unordered transpose FFT states that

$$T_{comp} = t_c \frac{N}{P}\Big(\frac{5}{2}\log_2 N + 1\Big) \tag{32}$$

and

$$T_{comm} = t_w \, 2\frac{N}{P^2}\Big(P-1\Big) + t_\ell \, 2(P-1). \tag{33}$$

Using the single processor time complexity given in Eq. 17, the speed-up for the parallel transpose FFT equals

$$S_t = \frac{P}{1 + \frac{4}{5}\left[\frac{t_w}{t_c}\frac{P-1}{P} + \frac{t_\ell}{t_c}\frac{P(P-1)}{N} + \frac{1}{2}\right]\frac{1}{\log_2 N}}. \tag{34}$$

Projected performance results, relative to the parallel Cooley-Tukey FFT, are shown in Fig. 15; in both figures projected performance is based on the performance characteristics of Samson, where [Seckel, private communication]

- $t_c = 1.0 \times 10^{-9}$ sec,
- $t_w = 2.0 \times 10^{-7}$ sec,
- $t_\ell = 1.0 \times 10^{-5}$ sec.

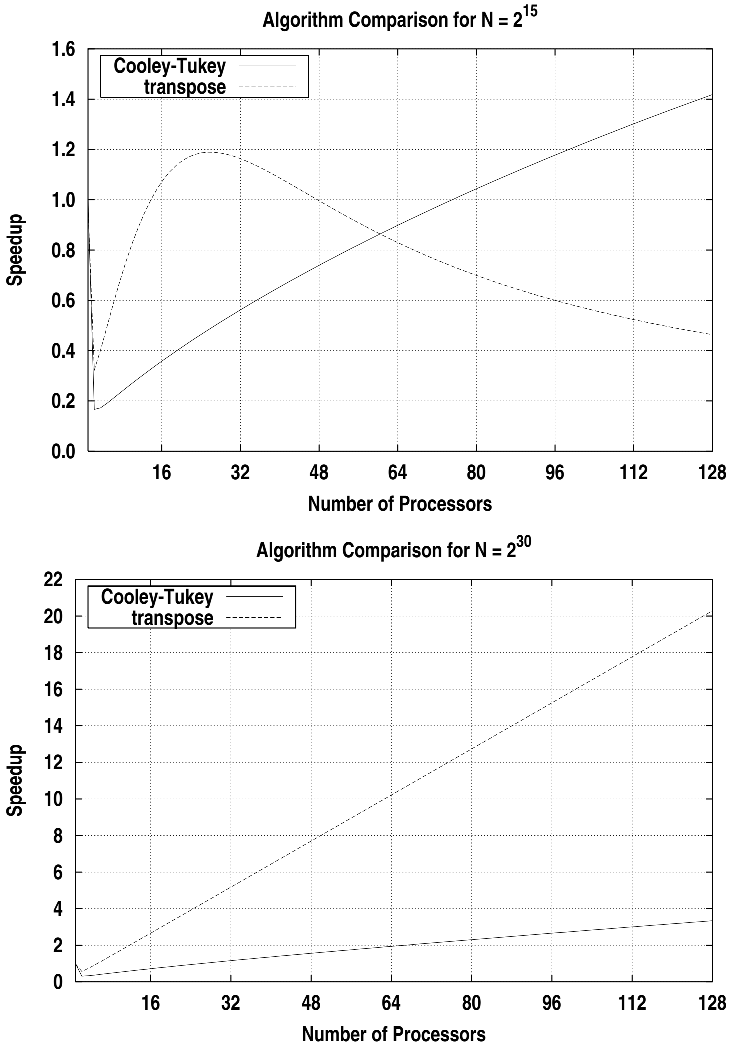

Figure 15: Projected 1D FFT performance on Samson for the parallel Cooley-Tukey and transpose algorithm. Projected results are shown for a small and large data set.

8 Conclusion

Although the ideal cost-optimal parallel time complexity equals $\mathcal{O}(\log N)$ with N processors, our time complexity analysis indicates that this ideal is a pipe-dream. The primary limiting factor with this many processors would be the latency. When the ratio $P/N \ll 1$, thereby eliminating latency effects, the projected speed-up is sub-linear because of the communication speed to computation speed, t_w/t_c. These factors must be appropriately weighed when deciding whether or not to parallelize a 1D FFT.

Furthermore, the naive assumption that a parallel algorithm must be faster than the best sequential algorithm is *not* necessarily correct. Of course, the only way that this can be determined before hand, is to carry out a thorough time complexity analysis, instead of automatically spending time and money to parallelize a sequential algorithm.

References

Mike Ashworth and Andrew Lyne. A Segmented FFT Algorithm for Vector Computers. *Parallel Computing*, 6:217–224, 1988.

David H. Bailey. FFTs in External or Heirarchial Memory. *Journal of Supercomputing*, 4:23–35, 1990.

William L. Briggs and Van Emden Henson. *The DFT: An Owners Manual for the Discrete Fourier Transform.* Society for Industrial and Applied Mathematics, 1995.

Kang Su Gatlin and Larry Carter. Faster FFTs via Architecture-Cognizance. *IEEE*, ?:249–260, 2000. Paper received from Lori Pollock, but publication details are unclear.

Anshul Gupta and Vipin Kumar. The Scalability of FFT on Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4:922–932, 1993.

W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 1989.

Paul N. Swartztrauber. FFT Algorithms for Vector Computers. *Parallel Computing*, 1:45–63, 1984.

Paul N. Swartztrauber. Multiprocessor FFTs. *Parallel Computing*, 5:197–210, 1987.

H. Joseph Weaver. *Theory of Discrete and Continuous Fourier Analysis.* John Wiley & Sons, Inc., 1989.