Parallel Fast Fourier Transformation Algorithm
Dustin Waldron
Abstract:

In this paper, I will introduce the significance of Fourier transformations. Next, I explain the concepts of discrete and fast Fourier transforms. I will then describe the problem a parallel Fast Fourier Transformation aims to solve. Finally, I will describe a parallel implementation of the Fast Fourier Transformation utilizing the 14 steps of Jeff Edmonds “How to Think about Algorithms”, described in Section 1.2, and time and space analysis as well.

Introduction:

In the physical world, almost everything can be represented in waves. Scientists and engineers often deal with systems that oscillate or vibrate. Trigonometric functions play a fundamental role in modeling such situations in the physical world. The problem for engineers and scientists is how to represent signals and send them in various forms.

Fourier Transformations, in the realm of science and engineering, are valid representations of waves in the real world. Fourier Transformations convert signals or data from the time domain to the frequency domain. The time domain is a representation of a wave over an infinite amount of time. The frequency domain is a representation of a wave over an infinite range of frequencies. A Fourier Transformation is reversible, meaning a signal may be modeled in the time domain and be transformed to be modeled in the frequency domain and vice versa. The time domain of a wave can be represented as a sum of sinusoids, which provide a best approximation for the wave, as seen in the following equation:

\[ Me^{iwt} = A \cos wt + jB \sin wt \]

Where w represents the frequency of a wave, A, B, and M are constants, and t represents time. The frequency domain of a wave can be represented as a sum of magnitudes and phases, or real and imaginary numbers respectively, as seen in the following formula:

\[ x(t) = A \sin(\omega t + \phi) = A \sin(2\pi f t + \phi) \]

While Fourier Transformations may be used in representing continuous signals, engineers and scientists use functions that are represented by finite sets of discrete values. Additionally, data is often collected in or converted to such a discrete format. Signals and waves may be divided into N equally-spaced subintervals. This fact give rise to a specific type of Fourier Transformations called Discrete Fourier Transforms (DFT). DFT’s are used for computer-based frequency domain analysis. There are many applications to DFT’s including spectral analysis, denoising audio signals, file compression, and sound filtering. A DFT for wave x may be written as:

\[ X_n = \sum_{k=0}^{N-1} x_k e^{-j2\pi kn/N} \]
and the inverse Fourier transform is

\[ x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{i2\pi kn/N} \]

Fast Fourier Transformations (FFT) are a numerically efficient method to calculate DFT’s. FFT’s were first introduced by Carl Friedrich Gauss in 1805, and further improved by J. W. Cooley and J.W. Tukey in 1965. The benefit of implementing a FFT is the exploitation of symmetries of \( e^{i2\pi k/n} \) in the computation of a DFT due to complex conjugate symmetry and periodicity in \( n \) and \( k \). The symmetry of odd and even terms of the DFT produces the following formula:

\[ X_n = \sum_{m=0}^{N/2-1} X_{2m} e^{-2\pi j mk/N} + e^{-2\pi j N/N} \sum_{m=0}^{N/2-1} X_{2m+1} e^{-2\pi j mk/N} \]

For each addition, the element \( e^{-2\pi j N/N} \) is what is known as the twiddle factor, \( \omega \).

With modern advancements in processing, parallelization of FFT has been widely investigated. While there are numerous FFT algorithms, I will focus on a parallel divide-and-conquer type radix-2 Cooley-Tukey FFT.

**Problem Statement:**

The problem I will solve with the parallel radix-2 Cooley-Tukey FFT will be transforming a one dimensional array, representing \( N \) elements of a wave, in place from the time domain to the frequency domain. This is important to solve based on the applications described earlier in this paper, and for the sake of reducing the time to convert a wave from the time domain to the frequency domain. Take for example the serial implementation of the radix-2 Cooley-Tukey FFT, a wave \( x \) is divided into eight discrete parts in the time domain. The indexes of the elements are ordered in bit reversed order. Suppose element \( N \) is at index four. Four represented in binary of size 3 bits is 100. If the bits of four are rearranged in reverse order, the value is 001, or one. The parallel radix-2 Cooley-Tukey FFT algorithm also requires bit reversal of element indexes. Once all elements are bit reversed, the elements will be transformed and rearranged in a butterfly crossing. This butterfly crossing is ideal for a parallel implementation because each butterfly calculation is independent of other butterfly calculations. The butterfly crossings and calculations continue for the 8 elements until the entire wave are converted into the frequency domain. This process can be represented in the diagram below:
The algorithm description for the parallel radix-2 Cooley-Tukey FFT is as follows:

**Specifications:**

Given a data set of \( N \) elements, construct two arrays \( x_r \) and \( x_i \) to represent real and imaginary components of each element of a wave \( x \), which are sorted in bit reversed order. Transform the real and imaginary components of each element in the wave from the time domain to the frequency domain and return the two converted lists \( x_r \) and \( x_i \) with components in the frequency domain.

**Basic Steps:**

First, create variables level to represent the current stage of the transform, step to signify which twiddle will be used, and threadId to represent a thread calculating each step component. Next create variables \( k \) to represent the twiddle index, \( i \) to represent the source array index, and \( j \) the destination array index. Then declare arrays \( w_r \) and \( w_i \) to represent the real and imaginary components of the twiddle index, as well as arrays \( x_r \) and \( x_i \) to represent the real and imaginary components of each element in the wave. Finally, establish SF representing the scale factor to prevent data overflow during computation. Calculate the \( x_r \) and \( x_i \) array values for the present level and the next level, until all the wave is transformed into the frequency domain.

**Measure of Progress:**

The measure of progress is the value of the level, which represents the phase number at a given time. The total number of levels are equal to \( \log_2 N \) for \( N \) elements in the wave being transformed.
The Loop Invariant:

The loop invariant is that while iterating through levels 1 through \( \log_2 N \), the real and imaginary components of wave \( x \) processed in the previous level are represented in the frequency domain.

Main Steps:

The first of the main steps are for each level, launch \( \frac{N}{2} \) threads. Each thread will perform calculations on its assigned element of wave \( x \), in place, based on the thread id. The threads will transform the even and odd indexed elements of the input arrays \( x_r \) and \( x_i \) from the time domain to the frequency domain. Once all threads are finished, proceed to the next level until all elements of \( x_r \) and \( x_i \) are completely transformed.

Make Progress:

For each iteration, the elements of \( x_r \) and \( x_i \) are transformed from the time-domain to the frequency domain for the current level. Once completed, increment the value of level and repeat the process and the algorithm makes progress.

Maintain Loop Invariant:

For each iteration of level, the values of the \( N \) elements are transformed by threads of the previous level will be representing the frequency domain. Every thread will calculate the appropriate values for arrays \( x_r \) and \( x_i \) without affecting the values of arrays \( x_r \) and \( x_i \) of another thread.

Establishing the Loop Invariant:

Initially, before the \( \frac{N}{2} \) threads are launched, wave \( x \) is represented as one element. By definition, the wave \( x \) is already represented in the frequency domain before any calculations by the algorithms are made, which establishes the loop invariant.

Exit Condition:

The exit condition is when all phases are completed and all elements of arrays \( x_r \) and \( x_i \) are represented in the frequency domain, at which time the algorithm returns the two arrays \( x_r \) and \( x_i \).

Ending:

If all phases are completed the algorithm has completely transformed the elements of wave \( x \) into the frequency domain, return arrays \( x_r \) and \( x_i \).
Termination and Running Time:

For $N$ twiddle calculations, and $N/2$ threads each phase for each level for $\log_2 N$ phases, the total number of computations is $N \log_2 N + N$. This results in overall computation time of $O(N \log_2 N)$.

Special Cases:

There are no special cases for this algorithm.

Coding and Implementation Details:

algorithm ParallelFFT(xr, xi, wr, wi)
(pre-cond): array xr contains the real components of wave x in the time domain, array xi contains the imaginary components of wave x in the time domain, both sorted in bit reversed order. Arrays wr and wi contain the real and imaginary components of the $N$ twiddle factors.
(post-cond): array xr and xi contains the real components of wave x in the frequency domain, begin

step = N
loop

(loop-invariant) the elements transformed by the previous phase are represented in the frequency domain

%Make progress while maintaining loop invariant
nextLevel = level << 1
step >>= 1
parallel do

%each thread launched performs calculations on xr and xi
%independently from other threads
k = step * (threadId%level)
i = (threadId/level)*nextLevel + (tid%level)
j = i + level
%calculate real and imaginary parts of twiddle factor
wr = w[k + N/4] >> 1
wi = -w[k] >> 1
%transform real and imaginary elements of wave x in place using formulas below
xr[j] = (xr[i] >> 1) - [(wr* xr[j]) >> SF - (wi*xi[j]) >> SF]
xi[j] = (xi[i] >> 1) - [(wr*xi[j]) >> SF + (wi*xr[j]) >> SF]
xr[i] = (xr[i] >> 1) + [(wr* xr[j]) >> SF - (wi*xi[j]) >> SF]
xi[i] = (xi[i] >> 1) + [(wr*xi[j]) >> SF + (wi*xr[j]) >> SF]
end do
level <<= 1
end loop
return xr and xi

end algorithm
Informal Proof:

Based on the fourteen steps previously described, the parallel radix-2 Cooley-Tukey FFT algorithm is correct. Given a wave $x$, the real and imaginary components of the wave are represented in arrays $x_r$ and $x_i$ respectively. Utilizing $N/2$ threads and the symmetry of twiddles and periodicity of the $N$ elements, each level of thread calculations will transform the $N$ elements of $x_r$ and $x_i$ from the time domain to the frequency domain. The loop invariant for this parallel radix-2 Cooley-Tukey FFT is that for each level, the elements transformed from the previous level must be in the frequency domain. The algorithm establishes this invariant when the wave is represented in one part, which is already a representation in the frequency domain. Each phase progression shows that the algorithm is making progress. As a result of the $\log_2 N$ total phases of calculations, where all elements are transformed in place by the $N/2$ threads, the algorithm terminates when the final result is a complete wave transformed from the time domain to the frequency domain. Based on the steps described in the fourteen points, the maintaining of the loop invariant, the progress made by the algorithm, and the correct final result of a wave transformed into the frequency domain, the algorithm must be correct.

Time and Space Analysis:

For the given problem, the parallel radix-2 Cooley-Tukey FFT converts a wave $x$, broken up into $N$ discrete elements, from the time domain to the frequency domain. From the implementation given above, the algorithm takes the wave of $N$ elements and divides the wave into even and odd components each being of size $\frac{N}{2}$. Using these two partitions, the algorithm takes $\frac{N}{2}$ elements and calculates the DFT for that set. The algorithm repeats this process for each partition until the wave is completely transformed from the time domain to the frequency domain. To calculate the time of computation and BigO complexity for two arrays of size $\frac{N}{4}$, the algorithm converts the elements into two arrays of size $\frac{N}{4}$. The mathematical representation is as follows:

$$N \text{ elements } \rightarrow 2^{\log_2 N} \text{ Even and Odd indexed DFTs}$$

$$\frac{N}{2}, \frac{N}{4}, ..., \frac{N}{2^p} = 1.$$ \quad \text{Where } p \text{ represents the number of partitions}

1 partition: $\frac{N}{2} \rightarrow 2 \left( \frac{N}{2} \right)^2 + N = \frac{N^2}{2} + N.$

$P$ partition: $\frac{N}{2^p} \rightarrow \frac{N^2}{2^p} + pN = \frac{N^2}{N} + N \log_2 N$

$\Rightarrow O(N \log_2 N)$ for $N$ elements
The calculated computation time is equal to the expected computation time comparable to the serial implementation of the serial radix-2 Cooley-Tukey FFT and the number of computations are accurately represented. Therefore, the time analysis for this algorithm as $O(N \log_2 N)$ is correct. Because the algorithm transforms the arrays $x_r$ and $x_i$ in place, no extra memory allocation is needed for the algorithm after variable declarations.

Conclusion:

In conclusion, the parallel radix-2 Cooley-Tukey FFT algorithm is an interesting approach to implement FFTs in parallel. The fourteen steps describing the algorithm, pseudo code, informal proof, and time analysis should leave one to believe the algorithm is correct. Furthermore, utilizing and optimizing this parallel algorithm further will help reduce the time needed to transform a wave represented in the time domain into the frequency domain. These faster FFTs will help scientists and engineers solve more complex problems and create innovative solutions for the modern world.
References


"Parallel Versions of FFTW." Parallel FFTW. Web. 14 May 2014.