Comparison of Sequential and Parallel Fast Fourier Transform

Rami Jurdi
Zachary Noble
Gregory Freitas
Jayden Bendezu

***Abstract*—This research is in the Digital Signal Processing field where our aims are to improve the performance of real-time signal processing using parallel processing techniques coupled with 1 dimensional Fast Fourier Transforms. It has been done before by other researchers implementing multi-dimensional Fast Fourier Transforms in a multi-threaded context. The purpose of our research is to gain a better understanding of parallel processing techniques and digital signal processing. Thus the main goal is to observe the outcome of implementing a multi-threaded Fast Fourier transform algorithm and learn from the state-of-the-art research.**

I. Introduction

The Fast Fourier transform (FFT) is an algorithm that uses Discrete Fourier transforms (DFT) on a time sequence to convert a signal, usually based on time, to a signal based in the frequency domain. This allows us to analyze a sequence of time-values by decomposing it into bins of different frequencies. The Fast Fourier transform is used in many applications ranging from digital signal processing, sampling, pitch correction software, and wave analysis.

The direct computation of the DFT results in n^2 multiplications and $n(n-1)$ additions, making the computation greatly expensive for sufficiently large n. Over the course of many years of research done by researchers in the field, more efficient algorithms were developed for computing DFTs, such as the Cooley-Tukey FFT algorithm. This algorithm reduces the time complexity of the computation of Fourier transforms from $O(n^2)$ to $O(nlogn)$ [1].

In this paper, we present our process of implementing the parallel FFT algorithm, specifically a variant of the Cooley-Tukey algorithm, and conduct performance comparisons between the sequential and parallel versions of the algorithm in C++.

II. Target Problem

The main problem at hand is to create an application that is able to process signals provided to the program as audio files such as .WAV, .MP3, and .MP4. Then displaying the audio files as a spectrograph of its signals, which is calculated using the Fast Fourier Transform algorithm. This problem can be broken into several steps of what we need to achieve.

- A GUI to interact with.
- Accepting audio files via local upload.
- Decoding the raw audio file data.
- Computing the parallelized Fast Fourier Transform on the audio signal.
- Displaying the detected frequencies in a Frequency vs. Amplitude chart.

A. Approach

Tackling some of the above sub-problems. In order to create a GUI to interact with, we will write the program using the QT C++ GUI framework to simplify taking audio files from disk and decoding the raw data. Through the use of provided libraries of QT the basic functionality of loading files and playing/decoding audio is handled and we use them on a higher level. This will allow us to focus more on the problem of processing signals for the audio files. Then by handling extracting raw data from the audio files using the QAudioDecoder class provided by QT, we intend to use one-dimensional Fast Fourier Transforms to process the signals. (How do we plan to parallelize the FFT is what would be discussed here briefly since will be explained thoroughly within the algorithms subsection).

1) Project Architecture:

B. Plan Outline

An outline of how our project development was structured is detailed below:

1) Setup programming environment using Qt C++.
2) Add support using Qt multimedia APIs to load and decode audio files from disk.
3) Stream the audio data to an output device such as speakers or headphones.
4) Display a waveform of the audio.
5) Implement single-threaded Fast Fourier Transforms.
6) Implement multi-threaded Fast Fourier Transforms.
7) Display the frequency vs. amplitude data calculated from the Fast Fourier Transform, in order to validate correctness.
8) Conduct experimental tests comparing multi-threaded implementation vs sequential implementation. Observe any performance boosts if any.

C. Algorithm

1) Discrete Fourier Transform: The Discrete Fourier Transform (DFT) can be calculated as a summation of the product

of the sample value and a complex exponential term, shown in equation (1).

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-j2\pi kn}{N}} \tag{1}$$

Where x_n denotes the sample value at index n, j the imaginary number, k the desired frequency, and N the total number of samples. The output of the DFT X_k is the Fourier Transform coefficient for frequency k, which in our program is the magnitude of the real and imaginary output of the DFT, normalized between 0 and 1 with 1 indicating a large response from the DFT for that frequency k.

An equivalent summation can be derived in terms of the trigonemetric functions cosine and sine using Euler's Identity $e^{ix} = cosx + isinx$, as shown in equation (2).

$$X_k = \sum_{n=0}^{N-1} x_n[cos(\frac{2\pi}{N}kn) - jsin(\frac{2\pi}{N}kn)] \tag{2}$$

This version of the DFT makes it easier to compute the real and imaginary components of the equation in code through the use of the std::complex class in the C++ STL.

As mentioned in the introduction, the runtime complexity for direct computation of the DFT is $O(n^2)$ which makes it unusable in real-time applications for significantly large sample sizes; however, the main idea behind the DFT computation is used in the FFT algorithm, but the computation is repeatedly sub-divided instead, achieving better overall performance.

2) Fast Fourier Transform: It is essential that we use an implementation that yields a better runtime as in the case of our program, operating on larger audio files will begin to take much longer and become more inefficient. Thus we will be working with the Fast Fourier transform algorithm to reduce our upper asymptotic bound. The specific FFT algorithm we used is the Cooley-Tukey radix-2 Decimation in Time (DIT) algorithm, which repeatedly divides a DFT of size N into two interleaved DFTs of size $N/2$.

The interleaving of DFTs is achieved by splitting the even-indexed samples and odd-indexed samples of the original N sized DFT, computing them using the DFT formula, and combining the results to produce the DFT of the original signal. This assumes that the original DFT size N is a power of 2, which can be ensured by using zero-padding in applications where the original audio file might not be a power of 2. A visualization of how the input signal is divided in the Cooley-Tukey radix-2 DIT algorithm is shown in Figure 1.

The first level of splitting of the DFT can be written as two summations, one operating on even-indices and one on odd indices, as show in equation (3).

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{\frac{-j2\pi k(2n)}{N}} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{\frac{-j2\pi k(2n+1)}{N}} \tag{3}$$

This equation is simplified in terms of cosine and sine for code simplicity in the implementation, similar to equation (2). The algorithm for the Cooley-Tukey radix-2 DIT algorithm also makes use of bit-reversal permutation addressing, which swaps the real/imaginary signal input elements based on the bit reversal of their respective index in the signal. This is needed because the splitting of even/odd index elements in the calculation of the FFT leads to unordered output, which this permutation can prevent by pre-processing the bit-reversals in $O(n)$ runtime. Overall, the runtime for the FFT is $O(nlogn)$, which is a significant improvement from the $O(n^2)$ runtime from the direct computation of the DFT.

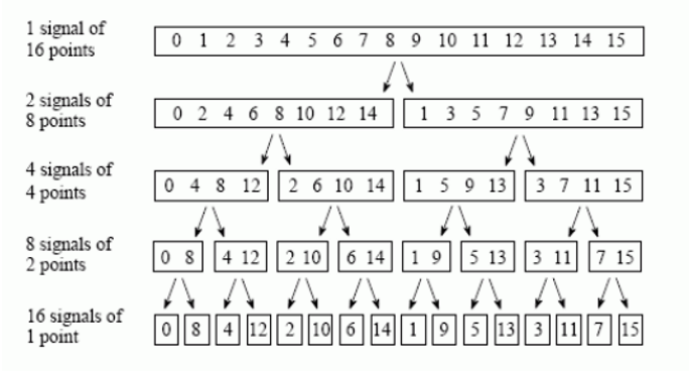

Fig. 1. Example of interleaved decomposition of a 16 point input signal [2].

The sequential implementation in C++ of the Cooley-Tukey radix-2 DIT algorithm that we used in our application came from Project Nayuki [3] and was modified for our use-case.

D. Parallelized Fast Fourier Transform

As explained in section II-A, our application implements a DistributedFFTWorkerThread class which computes the FFT on a slice of the original input of size N. This implementation makes use of the QThread API from the Qt 5 C++ core library. We decided to use 8 total FFT worker threads with each thread operating on an evenly-split amount of samples, padded to the next power of 2 if needed, using bit-shifting to find the size and resizing the std::vector, padded with 0s.

Each thread is spawned in the FTController class which assigns each thread a unique ID from 0-7, which is used in the range calculation (4) of the indices to calculate the FFT on.

$$data_i = signal[i*N/p, (i+1)*N/p-1](i = 0, ..., p-1) \tag{4}$$

Where N is the total size of the $signal$ input array, i is the unique worker ID, and p is the total amount of worker threads.

This range calculation (4) is a relatively common approach to distributing work between multiple threads, allowing for concurrent work to be done, and was used in the research paper by Zhong Cui-xiang et al. [1], in which they utilize a multiprocessor system to speed up the computation of the 1-D FFT radix-2 DIT algorithm.

For our approach we only utilize threads which operate on separate sections of the signal input, ensuring thread-safe computation without the use of any mutual-exclusion locks. Each thread, however, needs to relay the computed output back to the FTController through the use of thread-safe

signals available in the Qt Core library, which could lead to a potential bottleneck for a greater amount of threads. Finally, the FTController combines the split output of the worker threads into a local vector of (frequency, magnitude) points, which is used to plot the Frequency vs. Amplitude graph in our application.

E. Experimental Results

Below is our performance results of the sequential and parallel implementations of the Fast Fourier Transform tested on three different WAV audio files. The computation time may vary among different systems due to hardware capabilities.

Data Set 1			
Algorithm	File	File Size	Comp. Time
Sequential FFT	440Hz-1s.wav	88 KB	0.0051s
Parallel FFT	440Hz-1s.wav	88 KB	0.0014s

Data Set 2			
Algorithm	File	File Size	Comp. Time
Sequential FFT	440Hz-3s.wav	260 KB	0.023s
Parallel FFT	440hz-3s.wav	260 KB	0.0047s

Data Set 3			
Algorithm	File	File Size	Comp. Time
Sequential FFT	440Hz-30s.wav	2.5 MB	0.34s
Parallel FFT	440Hz-30s.wav	2.5 MB	0.058s

The last data set, which contains the 440Hz-30s.wav file, has over 1.3 million total samples in the signal, which is sufficient enough to compare runtimes of the sequential/parallel FFT algorithms. Each audio file was sampled at 44100 Hz, with a 16 bit sample size. So, a 30 second audio file sampled at 44100 Hz contains $30*44100 = 1323000$ total samples, which is used as input for the parallel/sequential FFT.

Additionally, each audio file had its FFT computed for 50 trials each, for both the sequential and parallel FFT algorithms, and the average runtimes were calculated and are displayed in the Comp. Time column in the table above. We observed a speedup factor of around 6x on the 30s audio file, utilizing 8 worker threads with our parallel FFT algorithm. As the number of samples increase, there is a greater performance increase between the sequential and parallel FFT implementations, as can be observed in the table.

III. State-of-the-Art Research

The papers that we will use to implement the parallel 1D Fast Fourier Transform algorithm are the *Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory* [4], the .....

IV. Further Improvements

Our implementation is only one of many implementation options we found in our research. In our research we found a few possible improvements on top of our current implementation. One possible improvement we could have added to our implementation could have been dynamically adjusting the number of threads we use for calculating our Fourier transform based on the number of samples we have in our audio file. We could have calculated some number of threads that would decrease the amount of padding required to reach a total number of samples that reach a power of 2, then we could theoretically decrease the total computation time by avoiding unhelpful computations.

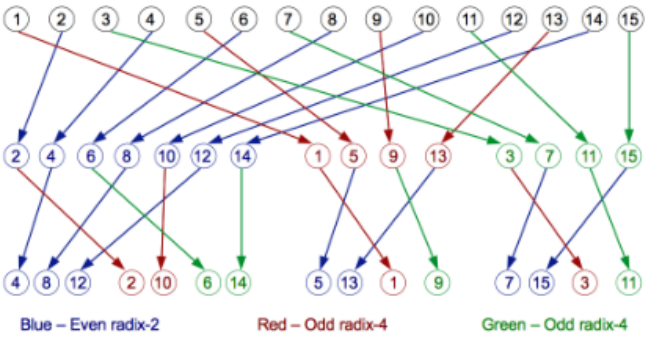

Fig. 2. Example of a split radix approach [5]

Another technique that we never were able to implement was something called the split radix algorithm. The way our algorithm works is that we split our sample points of audio into 2 equal groups in radix 2. It is split so that one thread recursively splits the even indices of the audio file into more threads equally, and a second thread recursively splits the odd indices equally between more threads. The split radix approach splits the even indices recursively from one whole chunk, but the odd indices are split into 2 separate threads, to then further be split similarly. This is illustrated in Figure 2.

This approach to splitting up the computation in practice improves performance by 20% [6].

V. Related Work

Here we will talk about how our work relates to currently worked on research.

VI. Our Contributions

References

[1] Zhong Cui-xiang, Han Guo-qiang, and Huang Ming-he, "Some new parallel fast fourier transform algorithms," in *Sixth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'05)*, pp. 624–628, 2005.

[2] Steven W. Smith, "The scientist and engineer's guide to digital signal processing," 1998. [Online].

[3] Nayuki, "Free small fft in multiple languages," 2021. [Online].

[4] R. Al Na'mneh, D. W. Pan, and R. Adhami, "Parallel implementation of 1-d fast fourier transform without inter-processor communications," in *Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory, 2005. SSST '05.*, pp. 307–311, 2005.

[5] B. Karsin, "Parallel fast fourier transform literature review," 2013.

[6] P. Duhamel and H. Hollmann, "Implementation of "split-radix" fft algorithms for complex, real, and real symmetric data.," *ICASSP '85. IEEE International Conference on Acoustics, Speech, and Signal Processing, Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '85*, vol. 10, pp. 784 – 787, 1985.