A Parallel Application of the Fourier Transformation

Justin Spidell – Brett Sumser

December 2021

Abstract

In this project we implemented many versions of the Fourier Transform. Namely, we implemented a parallel Discrete Fourier Transform (hereby referred to as the DFT), a iterative version of the Fast Fourier Transfrom (hereby referred to as the FFT), and a MPI version of the Parallel DFT. Our main results in this project concerned converting analog signals into digital (discrete) signals, then preforming a Fourier Transform on the signals to ascertain the pitches in the original audio. We also explored image manipulation with the Fourier Transform, and examined parallelized implementations of converting images to grayscale. Our project was written in C++ and a little bit of python, and we made use of the OpenMPI, OpenMP, and LodePNG libraries.

Introduction

The Fourier Transform is an important mathematical concept. Discovered in 1822 by Joseph Fourier, it has applications in digital signal processing, convolution in neural networks, image recognition and even speech processing. A Fourier Transform can be described as a "mathematical operation that changes the domain (x-axis) of a signal from time to frequency," [4]. The Fourier transform is denoted by adding a circumflex to the symbol of a function:

$$f \to \hat{f}$$

The Fourier transform is defined as:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} dx \tag{1}$$

Where x represents time, and ξ represents frequency.

For our purposes, specifically the conversion of signals and image processing, we need to use the non-continuous (discrete) version of the Fourier Transform. Unfortunately, the DFT is on the slower side for algorithms, being performed in $O(n^2)$. It is defined as:

$$\textit{Let } x_0, \ ..., x_{N-1} \textit{ be complex numbers},$$

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2i\pi kn}{N}} \quad k = 0, \ ..., \ N-1 \tag{2}$$

Using Euler's identity, we can transpose the function to:

$$\textit{Let } x_0, \ ..., x_{N-1} \textit{ be complex numbers},$$

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left(\frac{-2\pi kn}{N}\right) + i \sin\left(\frac{-2\pi kn}{N}\right) \quad k = 0, \ ..., \ N-1 \tag{3}$$

This will be our main implementation, as it neatly separates the real and imaginary components.

To achieve a FFT, we must go a step further. By splitting the DFT into two subsections, we can achieve a DFT with less computations and better speeds. The FFT is much faster than the DFT, being performed in $O(n \log n)$ time. The FFT can be easily implemented using a recursive or iterative programming method, but there are benefits to using an iterative approach; Mainly the ability to be parallelized. The FFT is a radix-2 algorithm, meaning that it is really two interleaved DFTs of size $N/2$. The FFT can be defined as:

$$Let\ x_0,\ ..., x_{N-1}\ be\ complex\ numbers,$$

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{\frac{-2i\pi km}{N/2}} - e^{\frac{-2i\pi k}{N}} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{\frac{-2i\pi km}{N/2}}$$

$$X_{k+N/2} = \sum_{m=0}^{N/2-1} x_{2m} e^{\frac{-2i\pi km}{N/2}} - e^{\frac{-2i\pi k}{N}} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{\frac{-2i\pi km}{N/2}}$$

$$for\ k = 0,\ ...,\ N-1 \tag{4}$$

Of course our implementation applies Euler's Identity to split the function into real and imaginary components, but this will be left as an exercise to the reader.

Methodology

Implementation of the Algorithms

As stated previously, we used C++ to implement our various Fourier Transforms. Our most basic DFT algorithm was written with two loops, calculating a sum using every input for each value in our output.

The FFT (Cooley-Tukey) takes a much different form. In this algorithm, we first must pad the input with 0's, this will produce some unavoidable error into the calculation, however it is negligible. We have to pad with 0's because of the nature of the FFT, it is a radix-2 algorithm, and must have an input that is a power of two. This is expensive on our time, but the benefit we get from using the FFT outweighs the cost. Next, we must bit reverse the indicies of the input. This means given an input of size 16 (4 bits) that a number located at index 3 (0011) must be switched with the number located at 12 (1100). Although bit operations are very fast, this still can be an expensive operation. However, the speed of the FFT far outweighs the cost of the re-indexing. Finally, We start the main body of the FFT. We used an iterative implementation, so the main body consists of two for loops, one controlling the flow of the algorithm, and the other calculating the partial sums.

Audio

The audio processing portion of this project was surprisingly simple. With a working DFT or FFT, the only step to filter the output into the corresponding pitches is to find frequencies that have large imaginary components. Naively implemented, it is as simple as saving the index of a large imaginary value.

Parallization

Parallelization of the Fourier Transforms went smoothly in general. DFT can be naively parallelized for a decent speedup by adding a $\#pragma\ omp\ for$. No *critical* or *atomic* call is need: as each thread creates one output in our array, there is no opportunity for memory errors or race conditions. FFT is certainly a more complicated algorithm to parallelize. Unfortunately, we could not achieve better results than and un-parallelized Cooley-Tukey implementation. We believe the cause of this lies in the overhead of creating new threads each and every iteration of the loop.

SIMD

Single Instruction Multiple Data prcoessors are also known as vector processors, and allow a single instruction to process multiple pieces of data in the same clock cycle. The way that SIMD works is by packing several pieces

of data into one data word. This allows the instruction loaded to act on each piece of data from a single instruction. This has applications in situations where large amounts of data are being manipulated. With respect to the Fourier Transform, this can be applied to image manipulation, audio processing, or other functions.

C/C++ contains SIMD functions called vector intrinsics. These are implemented in the compiler, allowing them to load orders of magnitude faster than common library functions. The found that they could use these intrinsics to produce code up to four times faster, and even more in certain cases! For this project, getting a handle on these intrinsics would allow for an increased knowledge in C/C++, while also providing experience in possible future scenarios with performance critical code.

Difficulties

Everything did not go as smoothly as planned for the project. There were problems encountered with parallelizing aspects of the project, including various issues with image loading, MPI, and attempting to implement SIMD for increased performance.

Parallizing the FFT

We had some trouble Parallizing the FFT. We could not get the speed to an ideal place, but had some ideas on how to approach the problem. To make the FFT run faster, we would parallelize with a thread pool fixed at compile time. The amount of threads is equal to N/B, with B being the block list for a list of contiguous data pieces. Threads are then distributed between processors round-robin style, which enables load balancing across available processors. In the end, each processor will perform an FFT computation on B data pieces. This implementation would have been ideal for implementation on the FFT, sadly, we could not make it work in time.

Image Loading

With regards to loading image data for transforming, there were a couple of difficulties. The Fourier Transform can only transform images that are in greyscale, so it makes sense to also look for ways to apply parallelism to the conversion of images to grayscale. The first issue encountered was finding a

library that was simple to use with little overhead and dependencies to get running. The first attempted library we used was OpenCV. This ended up requiring a chain of dependencies, increasing bloat in the project and ended up being difficult to use as well. LodePNG ended being the library we used for image loading, and was extremely simple to get running being only a single header file. With modification and more time, you could also change how it saves image data from being a vector or C style array to a struct of arrays for the RGB values, allowing easier memory aligning for SIMD operations.

SIMD/Intrinsics

After discussing the merits of SIMD and Instricsics in C++ with Professor Choi, we reached the decision to focus developement time into MPI. With the nature of converting grayscale images for use in the Fourier Transform, the RGB pixels need to be summed, and then the average of that value is applied to each RGB component. SIMD could be used to sum the first 2 RGB components, provided that they are memory aligned. However, you would still need to add the last RGB compenent, and then perform a division and assign the value to all of the pixel values. It was brought to our attention that although this is possible, the speedup overall would probably not really be worth the effort to get it running.

MPI

MPI was very harsh to us throughout the course of the project. The DFT seemed like an ideal candidate for MPI at first, but it wasn't an easy process. Besides MPI being difficult to implement on a project that already had many functions, like audio reading and image loading, it proved impossible to implement successfully even with a generated signal. The main issue we ran into was that no matter what we tried, MPI_Gather would never successfully gather the data. The function returned $MPI\_SUCCESS$, even though the array created in the process would never have anything inside it but 0's.

Results

Speed

Our results were collected using Talapas, tested using an input array of size 131072.

DFT	N/A
PDFT	60.4222s
Cooley-Tukey	0.14108s
Cooley-Tukey (Critial)	2.32468s
Cooley-Tukey (Locks)	1.31673s
numpy (Industry Standard)	0.00323s

The default DFT algorithm was incredibly slow. With and input so large, It could not keep up in the slightest, eating up all the execution time. We could not get the parallel FFT's to match the speed of the default FFT, and the Industry Standard FFT, numpy, which uses a popular C++ FFT library, beat us handily.

Audio

Utilizing our implementation of the Discrete Fourier Transform, we were able to successfully filter out individual pitches in musical chords. Shown in the graph below, our implementation was able to accurately detect the correct frequencies of the notes in different chords, represented as wav file recordings.

Conclusion

In this attempt to provide a more parallelized version of the Fourier Transform, we learned many lessons. The trade-off between the time it takes to implement parallel version of a concept and the performance returns it brings definitely mirrors the real world trade offs that one encounters in software developement. By creating a implementation of the Fast Fourier Transform, we gained insight into modern image and audio compression software, as well commonly found analog to digital conversion software. One can clearly see how a parallelized implementation could be helpful in a situation such as color grading for 4k and above video data, or filtering noise from hi-fidelity

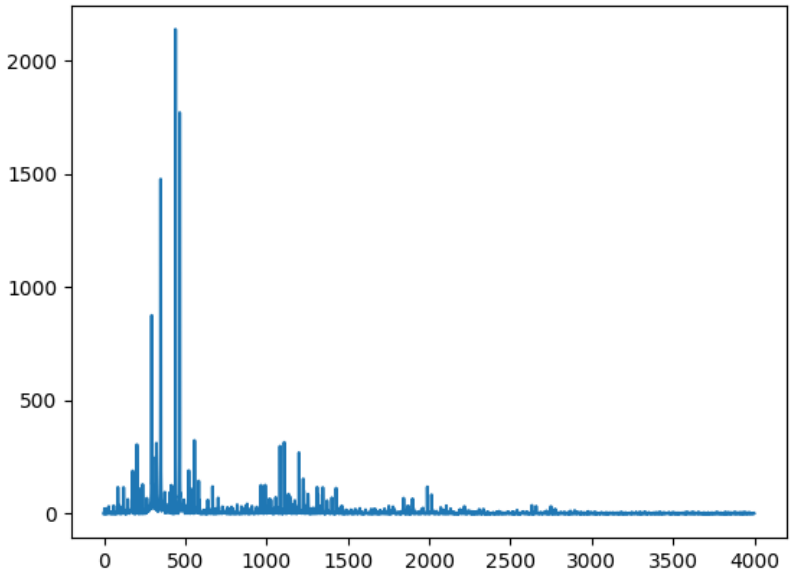

Figure 1: A $Dminflat6$ Chord

audio signals. We attempted to implement a more sophisticated parallelized version of the FFT, using concepts applied from class such as SIMD and multithreading/processing. We are excited to be able to have access to the hardware capable of applying these concepts for parallelized computing. The University of Oregon's supercomputer will be applied for testing and use of our implementation, hopefully with success. This is a lofty goal, and will take a good amount of work, but the end result will be something to be proud of.

References

[1] Anthony Blake. Computing the fast fourier transform on simd microprocessors, 2012.

[2] Konstantin. Improving performance with simd intrinsics in three use cases, 2020.

[3] Bo Liu. Parallel fast fourier transform, N/A.

[4] Cory Maklin. Fast fourier transform, 2019.

[5] Parimala Thulasiraman. Multithreaded algorithms for the fast fourier transform, 2020.