Parallel Fast Fourier Transform

1st Hazel Kim
Department of Computer Science
Kalamazoo College
Kalamazoo, MI
hazel.kimh@gmail.com

2nd Michael Orwin
Department of Computer Science
Kalamazoo College
Kalamazoo, MI
mcokzoo@icloud.com

***Abstract*—The Fast Fourier Transform (FFT) is an important algorithm in computer science, which sees many applications within signal processing, image processing, image compression, and more. Emergence of high-performance multiprocessor computers offers a new way to speed up this algorithm through parallelization. By reducing execution time, new applications become available which require the processing of massive data-sets.**

I. Introduction

The Fast Fourier Transform (FFT) is a divide-and-conquer algorithm for efficiently computing the Discrete Fourier Transform. The Fourier Transform is a function which decomposes a given signal as a function of time, down to its fundamental frequencies. This decomposition into relevant frequencies allows for data reduction, making processing quicker and storage sizes smaller while retaining the most important information. It is used in many different fields including engineering, applied mathematics, physics, finance, etc. The primary focus of this research was on parallelizing the FFT and its application toward digital signal processing and image processing.

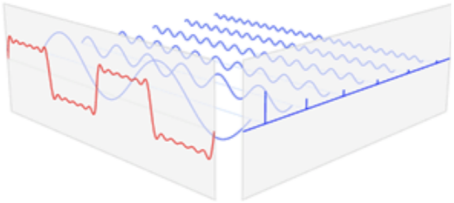

Fig. 1. Input signal and its decomposed trigonometric functions.

A. Mathematical Analysis of Fourier Transform

$$F(x) = \sum_{n=1}^{-1} f(x) e^{-2\Pi k x} \tag{1}$$

It is important to understand the mathematics behind the Fourier Transform in order to gain an intuitive understanding of how one can parallelize the algorithm. Using **Figure 1** as reference, the input signal in red will be decomposed into multiple trigonometric functions as seen in blue, by computing the Fourier Transform.

As the Fourier Transform equation **(1)** shows, the input data f(x) is multiplied by a particular exponential function whose summation then becomes the output signal. Specifically, for each specified frequency, the provided signal is wrapped around the complex plane such that its angle is a function of that specific frequency, and the amplitude is merely the modulus of the given complex value f(x). One can use the Euler Formula, represented in **Figure 2**, for clarity as to how signals are wrapped around the complex plane. The resulting output signal merely represents how well amplitudes or magnitudes align with each other when wrapped around the complex plane with the given frequency. For better understanding, one can see additionally in **Figure 3** the cyclical nature of the Euler Formula.

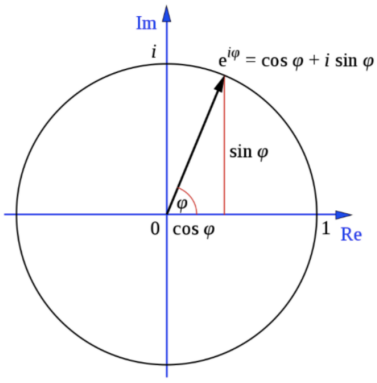

Fig. 2. Eulers formula: mathematical constant e.

II. Implementation

A. Discrete Fourier Transform

The algorithm explained in equation (1) can most aptly be described as the Discrete Fourier Transform (DFT). Due to the characteristics of the algorithm the input function needs to be discrete and finite. Using a rudimentary evaluation of this function one can achieve a time complexity of $O(n^2)$, which is extremely slow for larger data-sets.

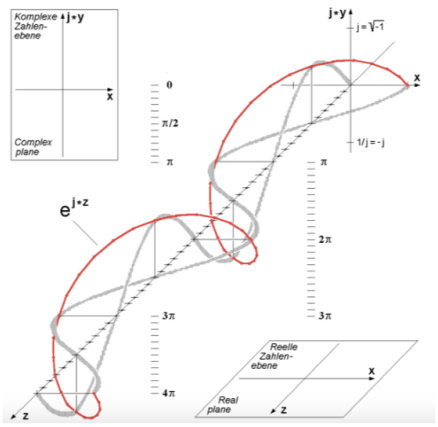

Fig. 3. Three-dimensional visualization of Euler's formula.

B. Fast Fourier Transform

In order to sidestep this slow runtime, other algorithms have been developed, which utilize either divide-and-conquer, convolutions, or a combination of both to achieve a time complexity of $O(nlog(n))$. In order to distinguish such algorithms, these are called Fast Fourier Transforms.

C. Cooley-Tukey Algorithm

In this paper specifically, an implementation of the Cooley-Tukey algorithm was developed in C (specifically a radix-2 decimation-in-time (DIT) version of the algorithm). Cooley-Tukey utilizes a method of divide-and-conquer allowing for the division of large DFT's into 2 DFT's of half the size.

In order to achieve fast results, an iterative version of the Cooley-Tukey algorithm was developed. The benefits of taking an iterative approach is that this cuts down on the time expenses for generation and allocation of new stack frames. Overall the algorithm can be broken down into 3 steps. These steps included:

1) Placing the dataset in bit-reversed order (by index)
2) Pre-computing the complex roots-of-unity utilized in the equation as a coefficient ($W_N = e^{i2\pi k}$)
3) Iteratively combining DFT's until the original DFT has been computed.

A visualization of this process can be seen in **Figure 4**. In general, each DFT is computed by computing the DFT associated with the odd and even indexed values. By pre-computing smaller DFT's in this manner one can see that the overall DFT can be computed by a simple linear combination of the smaller DFT's (where odd terms are multiplied by the respective coefficient, root-of-unity, based off their index and the given DFT's size).

Due to the way Cooley-Tukey computes the FFT, one can see that datasets must be a power of two. In some sense this can be seen as a downside, but also this can merely be resolved by padding the input signal with 0's to achieve the desired length. While this may be slower, it would be much faster than computing a DFT with time complexity of $O(n^2)$. Other algorithms exist such as Bluestein's algorithm or Radar's algorithm in which padding the input is not necessary, but these algorithms are much more complicated and slower. Both of these algorithms make use of the convolution theorem to compute two separate FFT's and then multiply them pointwise to achieve a FFT of the same length.

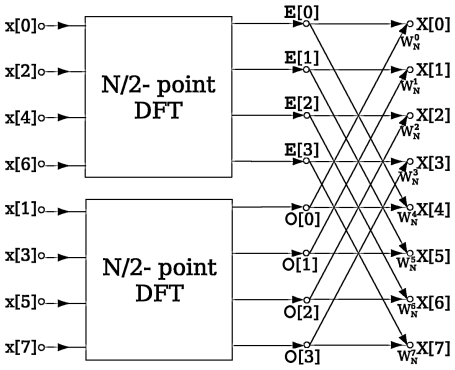

Fig. 4. A diagram for visualizing the process of computing the FFT using the Cooley-Tukey Algorithm.

D. Parallelization

For each of the three steps of the Cooley-Tukey algorithm, parallelization was extremely trivial using OpenMP's $parallel\ for$ statement and $collapse()$. Due to the nature of the algorithm, the order of DFT computation had to remain sequential (smaller DFT's computed first), but the remaining sections were easily parallelized.

E. Code

Algorithm 1 contains the general pseudo code for the Cooley-Tukey algorithm as described above.

Following the development of this algorithm in C and OpenMP, python was used for testing and graphical confirmation. Specifically, $ctypes$ was used to import a shared object file generated from the programmed $fft.c$ file. SciPy packages such as Matplotlib and NumPy were also utilized. Specifically, NumPy's implementation of the FFT allowed for a simple method of verification.

III. Result

A. Testing and Verification

In order to test the developed algorithm, a program was written to create artificial signal data. This data was outputted to a TSV. By utilizing TSV's, one could easily compare results of multiple FFT algorithms to make sure **Algorithm 1** was working correctly.

When comparing with NumPy's built in fft function, the resulting FFT computations were identical. However, this was only for when data-sets were powers of two. For different sized data-sets the resulting plot merely had increased resolution due to the 0-padding necessary. Speeds between the two algorithms were extremely comparable. **Figure 5** and **Figure 6** show an

Algorithm 1 Parallelized Radix-2 DIT Algorithm

function FFT(a[], length)
    `#pragma omp parallel for`
    for $i \leftarrow 0$ `to` $length$ **do**
        Initialize $swap\_index = bit\_reverse(i, length)$
        if $swap\_index < i$ **then**
            Initialize $temp = a[i]$
            $a[i] = a[swap\_index]$
            $a[swap\_index] = temp$

    Initialize w_ns to hold $length$ complex numbers
    `#pragma omp parallel for`
    for $k \leftarrow 0$ `to` $length$ **do**
        $w\_ns.real = cos(2 * \pi * k/length)$
        $w\_ns.imaginary = -sin(2 * \pi * k/length)$

    Initialize $num\_iterations = num_bits(length - 1)$
    for $i \leftarrow 0$ `to` $num\_iterations$ **do**
        Initialize $outer = round(pow(2, num\_iterations - i - 1))$
        Initialize $inner = length/(2 * outer)$

        `#pragma omp parallel for collapse(2)`
        for $j \leftarrow 0$ `to` $outer$ **do**
            for $k \leftarrow 0$ `to` $inner$ **do**
                Initialize $even\_index = k + inner * 2 * j$
                Initialize $odd\_index = k + inner * 2 * j + inner$
                Initialize $wn1 = outer * (even_index\%(inner * 2))$
                Initialize $wn2 = outer * (odd_index\%(inner * 2))$

                Evaluate and multiply the complex values at $even_index$ and odd_index (multiply by root-of-unity)
                Store these values in new_even and new_odd

                $a[even\_index] = new_even$
                $a[odd\_index] = new_odd$

example of the artificial signal generated and the output to the custom FFT algorithm written.

After verifying the algorithm was working, other use-cases were programmed. One such example is merely computing a 2D FFT. To compute a 2D FFT, one computes an FFT along the rows of a data-set followed by along the columns of a data-set. This process is extremely relevant to image processing. Example images are seen in **Figure 7** and **Figure 8**.

Finally, a small program was written using the *sounddevice* package in python which allowed for the generation of a spectogram. For this spectogram, the researchers sectioned mic data collected in real-time, into overlapping chunks. Subsequently the FFT algorithm was computed on each chunk and then results displayed using matplotlib.

Figure 9 provides a beautiful example of this process; a spectogram of playing all the white notes on a piano. Looking closely at this image, one can even see the instruments harmonics being displayed.

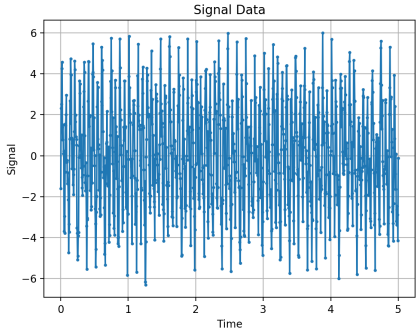

Fig. 5. A plot of artificial signal data generated for testing purposes

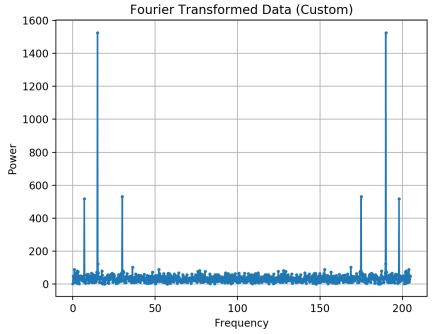

Fig. 6. A plot of the resulting output of the custom FFT algorithm

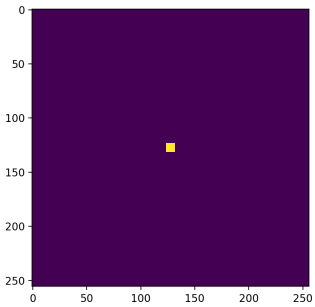

Fig. 7. A test image containing an 8x8 highlighted square in the center of the image

B. Parallelization Speedups and Efficiencies

Following the verification of the programmed FFT, the algorithm was timed using the built-in timing functions in OpenMP. Overall, one can see that **Algorithm 1** is weakly

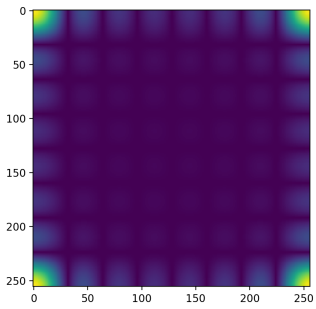

Fig. 8. The result of computing a 2D FFT on the provided test image

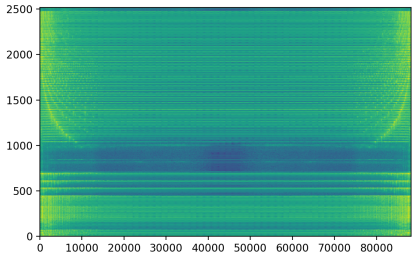

Fig. 9. A plot of artificial signal data generated for testing purposes

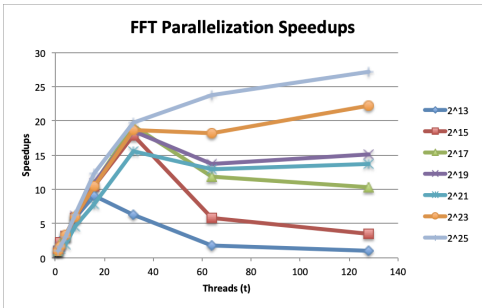

Fig. 10. A line chart comparing the speedups with increasing thread counts and problem size

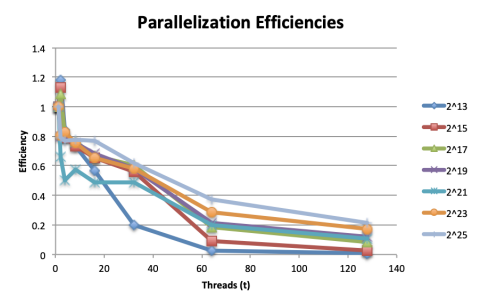

Fig. 11. A line chart comparing the efficiencies with increasing thread counts and problem size

scalable. The computed speedups and efficiencies can be seen in **Figure 10** and **Figure 11**.

Conclusion

Overall, the researchers were successfully able to program and parallelize a radix-2 DIT algorithm using OpenMP alongside c.

In order to improve the run-time of this algorithm in the future, other FFT algorithms could be programmed alongside other versions of the Cooley-Tukey algorithm. This would provide more versatility in the method of computation in order to achieve the fastest results. In addition, expansion and fleshing out of use cases would allow for a more readily usable package. All code included in this paper is located at https://github.com/jetstream5500/FFT.

References

[1] Anand, Aash. A Brief Study of Discrete and Fast Fourier Transforms. (2010, October 11). Retrieved January 30 2019, from https://www.math.uchicago.edu/~may/VIGRE/VIGRE2010/REUPapers/Anand.pdf

[2] Chu and George. FFT algorithms and their adaptation to parallel processing. (1997 September 22). Retrieved March 14, 2019, from https://ac.els-cdn.com/S0024379598100861/1-s2.0-S0024379598100861-main.pdf?_tid=1b0bc87f-b20a-4920-9ef8-bfb978263833&acdnat=1552587483_7710eadbfb1a9fe6c16792f6aee58519

[3] Discrete Fourier Transform. (n.d.). Retrieved January 30, 2019, from https://en.wikipedia.org/wiki/Discrete_Fourier_transform

[4] Douglas, B. (Producer). (2013, January 19). Introduction to the Fourier Transform (Part 2)[Video file]. Retrieved January 30, 2019, from https://www.youtube.com/watch?v=kKu6JDqNma8

[5] Fourier transform. (n.d.). Retrieved January 30, 2019, from https://en.wikipedia.org/wiki/Fourier_transform

[6] Govindaraju, Lloyd, Dotsenko, Smith, and Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. Retrieved March 14, 2019, from https://mc.stanford.edu/cgi-bin/images/7/75/SC08_FFT_on_GPUs.pdf

[7] LIU, B. (n.d.). Parallel Fast Fourier Transform. Retrieved January 30, 2019, from https://cs.wmich.edu/gupta/teaching/cs5260/5260Sp15web/studentProjects/tiba&hussein/03278999.pdf

[8] Parallel Fast Fourier Transform implementations in Julia. Retrieved March 14, 2019 from https://ocw.mit.edu/courses/mathematics/18-337j-parallel-computing-fall-2011/projects/MIT18_337JF11_FFT_rpt.pdf