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 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 extracting raw data from the audio files using the QAudioDecoder class provided by Qt, we intend to use one-dimensional Fast Fourier Transforms (FFT) to process the signals. For our approach on parallelizing the FFT, we plan on taking a distributed approach which evenly divides the work each thread does when calculating the FFT over N total samples. The algorithms for the sequential and parallel Fast Fourier Transforms will be discussed in more detail in section II-C.

1) Project Architecture: The core of the project resides in the FTController, which spawns DistributedFFTWorkerThreads using an array to hold each one.

For the distributed DFT, the number of samples is calculated and if the result is 0, it exits. The raw data is taken as a char array and is iterated through as a short array to retrieve the actual data from the samples. Then the calculation for the range of data that the thread will process is performed, based on the thread's worker ID. Each sample is evaluated from the short array and multiplied by the real and imaginary terms which are represented with cosine and sine respectfully. This product is added to the current sum for the current iteration of frequency k.

After the summation is finished, the magnitude of the current sum is evaluated and stored as the maximum sum if it is the largest amplitude calculated so far. The point of the graph is stored as an (x, y) coordinate in the final vector output, which represents a frequency bin x and corresponding amplitude y.

The output vector is iterated only on the lower half, where the frequencies that are within the scope of our desired frequency are plotted in the Qt chart for frequency visualization.

The point vector is then sent to the FTController where it normalizes the amplitude values between 0 and 1 based on the maximum summation we calculated.

For the distributed FFT, the Cooley-Tukey FFT algorithm is utilized. First, the length of the real and imaginary vectors are checked to be of the same length. Once that is verified the number of levels for the FFT are computed by bit shifting the length. If the length is not a power of 2, then both the real and imaginary vectors must be resized to the next highest power of 2, automatically padding with 0s.

The sine and cosine calculations are done next, followed by bit-reversing the addressing permutation. Next the Cooley-Tukey decimation-in-time radix-2 FFT algorithm is performed. Finally, the output vector is filled by calculating the magnitude of the complex element and updating the maxSum atomic member variable with the greatest magnitude. The corresponding frequency bin from the current index is calculated using the FFTUtils class and if the frequency is a duplicate from a previous index it is skipped. The frequency and amplitude are added to the output vector as a pair.

Additionally, the FTController class is able to terminate running threads by requesting an interruption and waiting on clean terminations of currently running threads. This allows for clean and safe thread termination in our application in the case that the user decides to load a new audio file while the distributed FFT calculation is still processing. If the thread notices that an interruption was requested, it will clear its data. This check occurs when the thread is iterating through each sample.

The core structure of our application is represented via the class diagram in Figure 1. The AudioFileStream class handles decoding the raw audio data from an audio file via Qt's QAudioDecoder class and writes the data to the Spectrograph's buffer. When the AudioFileStream finishes writing all the audio data to the buffer it calls the Spectrograph calculateSpectrum() method which tells the FTController class to start spawning the 8 DistributedFFTWorkerThreads. Under the hood, the FTController class automatically sets the worker IDs for each thread and passes the required audio data to each thread, also providing support for clean termination of these threads if needed by the Spectrograph.

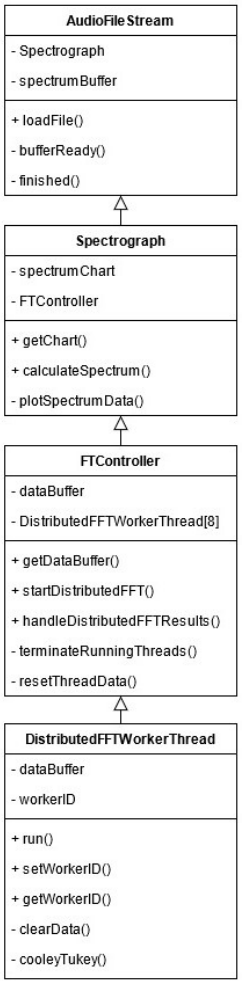

Fig. 1. Class diagram of the core classes of our application.

B. Plan Outline

An outline of how our project development was structured and the steps we took to ensure we achieved our goals 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 trigonometric functions cosine and sine using Euler's Identity $e^{ix} = cosx + isinx$, as shown in equation (2).

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

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 2.

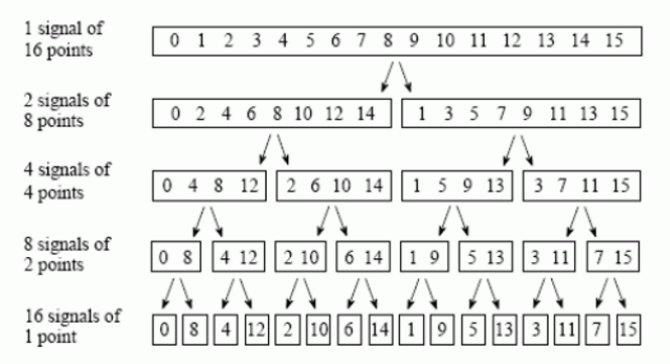

Fig. 2. Example of interleaved decomposition of a 16-point input signal [2].

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).

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

This equation is simplified in terms of cosine and sine for code simplicity in the implementation, similar to equation (2). 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.

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 with zeros to the next power of 2 if needed, using bit-shifting to find the size and resizing the std::vector.

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 perform the FFT on for that particular thread.

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

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. The output $data_i$ is the array slice of the input $signal$ array for thread i.

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 single local vector of (frequency, magnitude) points, which is used to plot the Frequency vs. Amplitude graph in our application.

E. Experimental Results

Figure 3 displays our performance results of the sequential and parallel implementations of the Fast Fourier Transform

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

Fig. 3. Data sets and computation times for each sample audio file.

tested on three different WAV audio files. Obviously, the computation time may vary among different systems due to hardware capabilities, but we generally expect the speedup factor to be around the same.

The last data set, which contains the 440Hz-30s.wav file, has over 1.3 million total samples in the signal, which is sufficient 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 Figure 3. 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 discrete Fourier transforms have a variety of applications for the science and engineering fields [1].Though due to how inefficient the base form of Fourier transforms are, the Cooley-Tukey Fast Fourier transform was developed in 1965 that was later expanded on [4]. These modified implementations make up the state-of-the-art research in digital signal processing using Fourier transforms. Although there are several implementations, they are typically those that improve on the constant factors, none of which that obtain a better upper asymptotic bound of $O(NLog(N))$. Although the efficiency of each algorithm is typically determined based on the total amount of operations which involves the addition and multiplication count aiming for less operations but yielding same results. Further expanding on the FFT algorithms to further increase performance on distributed and multiprocessor systems, variations of the algorithms were parallelized. Where some utilize inter-processor communications, and others with no communications. A comparison between the two approaches to parallelizing the FFT is provided in [5], with an end result of no utilization of inter-processor communications performing better on smaller data sets, and utilization of inter-processor communications performing more efficient on larger data sets. In the below subsections, the different implementations are explained in more detail.

A. Radix 2 Decimation in Time

This RAD 2 DIT algorithm makes use of the divide and conquer methodology. Through exploiting the symmetrical properties of the DFT the summation can be broken up into computations of even and odd sample points [6]. The sub trees of even and odd points are then continuously broken up into smaller transforms. Due to the computations not being in a natural ordering, it is necessary to use bit-reversed addressing permutations to obtain the natural ordering of the results. As said by Douglas L. Jones in [7], it is considered one of the simplest Power-of-two FFT algorithms. That being the case, it serves as a great approach to begin with due to its efficiency relative to its simplicity.

B. Split Radix FFT

Similar to the Radix 2 DIT, the sample points are split into even and odd indices. Although there is a key difference where the even indexed terms uses a radix-2 index map, and the odd indexed terms uses a radix-4 index map. Since the even and odd indices are computed independently, it is possible to split them into the two different index mappings. The number of computations required deduce to $4nLog(n)$ as opposed to the Radix 2 DIT's $5nLog(n)$ [6]. Due to the efficiency provided by the split radix approach, it is preferred to the Radix 2 approach, as the implementation is not much more difficult to expand to.

C. Prime Factor Algorithm

The idea behind this algorithm is to reorder the DFT computations in such a way that any redundancies are exposed, overall allowing for the computation of the DFT to be reduced [7]. This approach is not as widely used as the Power-of-two FFT's such as Split radix and Radix-2 DIT discussed above. One characteristic about this algorithm is that it serves to remove the twiddle factors and remaining with multiples of short length, the opposite effect that is achieved with the Power-of-two algorithms. Although this approach is not as widely used, it yields similar operation count on similar length samples [7], making this approach not inferior to the latter.

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

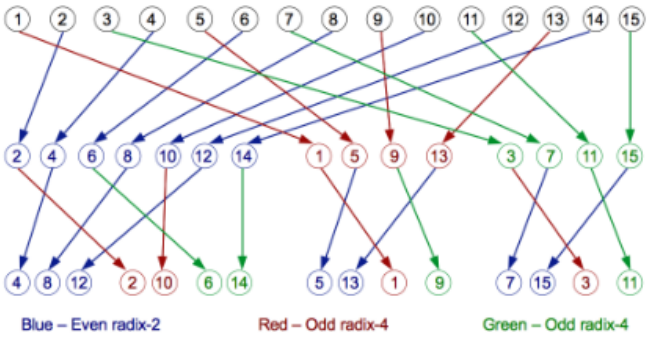

Fig. 4. Example of a split radix approach [8]

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.

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 4.

This approach to splitting up the computation in practice improves performance by 20% [9].

Another improvement we could have made to the Cooley-Tukey radix-2 DIT algorithm was pre-computing the $\frac{N}{2}$ cosine and sine values and re-using those values for each thread, since each of the 8 threads ends up re-computing the same cosine/sine functions independently anyways. This could end up saving us $O(n)$ time in pre-computation for the Cooley-Tukey algorithm; however, the overall runtime complexity remains $O(nlogn)$ due to the main computation loops.

V. Related Work

The implementation that we went with was the Radix 2 Decimation in Time algorithm. Parallelizing it without inter-processor communications, where a constant number of 8 threads are utilized. It varies from the methods used in [10], which instead made use of an arbitrary number of processes *P*.

There are various applications of the Discrete Fourier Transform in different fields of study. We found topics ranging from digital image processing to modeling the earth's surface.

When conducting remote sensing from satellites for modeling the earth's surface, one of the most important problems is modeling surface—fragments of dynamic surfaces such as the waves of the ocean and landforms that slowly shift as well. The use of a Discrete Fourier Transform is recommended by some researchers, where the timing of implementing the transform is crucial for improving the quality and accuracy. The use of parallel computations helps to decrease the implementation time. By decreasing the length of time, more of the model surface-fragment can be generated, which would improve the accuracy and quality of the overall model. [11]

Another related research paper that was found had a focus on the possible parallelization techniques for multidimensional hypercomplex discrete Fourier transform (HDFT). The HDFT has been mainly utilized in image and multidimensional signal processing and can be given by the following algebraic expression:

$$\sum_{n_1,\dots,n_d=0}^{N-1} f(n_1,\dots,n_d)W^{<m,n>} = \prod_{k=1}^{d} w_k^{m_k n_k}, w_k^N = 1$$

One notable aspect of the transform is that the N-th roots w_k from unity are found in different sub-algebras that are isomorphic to a complex algebra of some 2^d algebra B_d. Computing a multidimensional transform takes great effort when the dimensionality increases. In generating the hypercomplex spectrum, an analog of the inner parallelism of the Cooley-Tukey scheme was used. [12]

Another related publishing had a focus on creating a more accurate calculation of the Discrete Fourier Transform. In some cases, the Discrete Fourier Transform is insufficient in approximating the continuous Fourier transform. For example, a function such as $h(t) = e^{-50t}, t \in [0,1]$, the error on DFT $\{\mathrm{h}\}$ around f = 64 decreases to approximately $N^{-1/3}$. Which means that N must increase by a factor of 1000 in order to decrease the error by a factor of 10. The paper presented a method to provide accurate approximations of the continuous Fourier transform with a similar time complexity to the Fast Fourier Transform. The assumption of signal periodicity is no longer rigid and allows to compute numerical Fourier transforms in a broader domain of frequency than the usual half-period of the DFT. This behavior is highly recommended in image processing since it obtains the Fourier transform of an image without the usual interference of the periodicity of the classical DFT. [13]

One popular and very efficient implementation of FFT's comes from a library called FFTW (Fastest Fourier Transform in the West) developed by Matteo Frigo and Steven G. Johnson at MIT. The library was developed with portability and flexibility in mind, allowing computation of real and complex-valued input of arbitrary size and dimension in $O(nlogn)$ runtime. Additionally, it supports many variants of the Cooley-Tukey algorithm, and chooses the variant that it estimates to be the most preferable in the specific situation. For example, it switches to using either Rader's or Bluestein's FFT algorithm for computing the FFT on prime-sized input, and chooses among variants of the Cooley-Tukey algorithm for composite sized input. FFTW was written in C and is currently used as the matrix package for calculating FFTs in MATLAB.

Additionally, FFTW has parallel subroutines for the FFT calculations, including one-dimensional, multi-dimensional (with shared memory), and multi-dimensional (with distributed memory) versions. In FFTW's one-dimensional parallel implementation, which uses the Cooley-Tukey algorithm, the recur-

sive sub-problems of the original input are divided equally among different threads. [14]

VI. Open Research

The top-performing platforms best suited for FFT's in the current day are distributed-memory supercomputers [8]. Current day GPU's are by no means terrible, but by comparison to supercomputers, the focus of FFT computations will most likely remain on the supercomputing platforms for some time. Although there are still resources available further pushing the research within the FFT field. As mentioned in section V, FFTW has become an increasingly popular open-source library for furthering FFT research. It serves well as it is easily accessible and flexible to allow engineers of other fields to make improvements and further develop the library. Relative to the current standing of the field overall, it seems the need for more efficient FFT's will become necessary as the increase in size of datasets will demand it. Even more so the improvement of parallel FFT's will be required, as Ben Karsin says, "For many applications, sequential FFT algorithms are simply not powerful enough" [8]. The growing use cases for the FFT will certainly in the future provide new and improved algorithms or findings.

VII. Our Contributions

After performing experiments with sequential and parallel implementations of the FFT, the execution times were recorded and proven that the parallel implementations ran more efficiently, based on trial data. This contributes to the fact that parallel implementations of the Fast Fourier transform are more effective for large N. Also, that the FFT algorithm is well suited for parallelization. Since there are many applications for FFT calculation, any improvements to the speed of the algorithm are vital when calculating large samples of *N*.

VIII. Conclusion

Overall, the research was based on a difficult topic, so most of all learning and understanding the math behind it proved to be the toughest hurdle. It was imperative to understand the base discrete Fourier transform, to be able to understand its optimization and then implement it. A more straight forward approach of no inter-processor communication was chosen for simplicity. It was easier to follow and understand with it being lock free and not having any conflicting memory accesses while values were being computed. Also, due to the nature of the problem being relatively distributed, it seemed natural to apply a distributed approach to calculating the FFT. The performance gains made with our parallelized implementation were a relatively large improvement, as the sequential FFT was about 6 times as slow, making a significant improvement over large number of *N* samples.

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." http://www.dspguide.com/ch12/2.htm, 1998. [Online].

[3] Nayuki, "Free small fft in multiple languages." https://www.nayuki.io/page/free-small-fft-in-multiple-languages, 2021. [Online].

[4] "The cooley-tukey fast fourier transform algorithm." https://eng.libretexts.org/@go/page/2013, 2021. [Online].

[5] R. Al Na'mneh, D. 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.

[6] Manish Soni and Padma Kunthe, "A general comparison of fft algorithms," no. 713229618.

[7] Douglas L. Jones and Ivan Selesnick, *The DFT, FFT, and Practical Spectral Analysis*. Connexions, 2010.

[8] B. Karsin, "Parallel fast fourier transform literature review." https://benkarsin.files.wordpress.com/2013/12/fft-review2.pdf, 2013. [Online].

[9] 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.

[10] G. Xie and Y. Li, "Parallel computing for the radix-2 fast fourier transform," in *2014 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science*, pp. 133–137, 2014.

[11] Cosimo Stallo, Marina Ruggieri, Sabino Cacucci, Donatella Dominici, Vasily Popovich, Christophe Claramunt, Manfred Schrenk, and Kyrill Korolenko, *Information Fusion and Geographic Information Systems (IF AND GIS 2013): Environmental and Urban Challenges*. Lecture Notes in Geoinformation and Cartography, Springer-Verlag Berlin Heidelberg, 1 ed., 2014.

[12] M. Chicheva, M. Aliev, and A. Yershov, "Parallelization techniques for multidimensional hypercomplex discrete fourier transform," pp. 413–419, 09 2005.

[13] N. Beaudoin and S. S. Beauchemin, "An accurate discrete fourier transform for image processing," in *Object recognition supported by user interaction for service robots*, vol. 3, pp. 935–939 vol.3, 2002.

[14] M. Frigo and S. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.