COP5522
PROJECT REPORT

DAE HYUN SUNG AND FAROOQ MAHMUD

1. Introduction

In this project we demonstrate optimizations made on a recursive implementation of a Fast Fourier Transform (FFT). The FFT algorithm takes samplings of multiple sine waves for the period 2π and sums the samples to create a complex waveform upon which to perform the FFT. We successfully recreated a recursive version of the well-known Cooley-Tukey algorithm whose main advantage is its $\Theta(n \log n)$ runtime as opposed to the $\Theta(n^2)$ runtime of the straightforward implementation [2]. After benchmarking the recursive implementation we proceeded to benchamark an iterative version adapated from an open source implementation [4]. The iterative version showed drasitaclly reduced runtimes. Lastly, we improved the runtime of the iterative implementation by parallelizing two **for** loops using the OpenMP library. Subsequent sections cover the testing methodology and compare the execution times of three implementations based on the varying number of threads. Lastly the scalability and efficiency metrics of the parallelized iterative implementation are presented.

All experiments were performed on the Ubuntu 20.04 operating system equipped with an Intel I7 2.8GHz processor having four physical cores and eight logical cores. All programs were run with the **-O3 gcc** compiler optimization flag.

The algorithm exepects input sizes that are powers of two. There are solutions involving non-powers of two, but those complicate the code without providing any additional insight into the performance [2]. In the parallel case, the number of threads must be powers of two. This assumption simplifies the code.

Appendix A lists the structure of the code files for the three implementations. Instructions for building the three executables are in Appendix B. Instructions for running the three executables are in Appendix C.

The video demonstration for this project can be viewed at https://youtu.be/iUrCV7SN1nw.

2. Testing Methodology

To ensure meaningful runtime values, the input sample size must be adequately large. For the tests, a sample size of 2^{22} samples is used. The tests generate three cosine waves. The first waveform has frequency of one and an amplitude of ten. The second waveform has frequency of two and an amplitude of four. The third waveform has frequency of four and an amplitude of five (Figure 1). We generate the samples over a period of 2π, and the summed waveform (Figure 2) is used as the input for the FFT calculation. The FFT results are shown in Figure 3

FIGURE 1. Input waveforms.

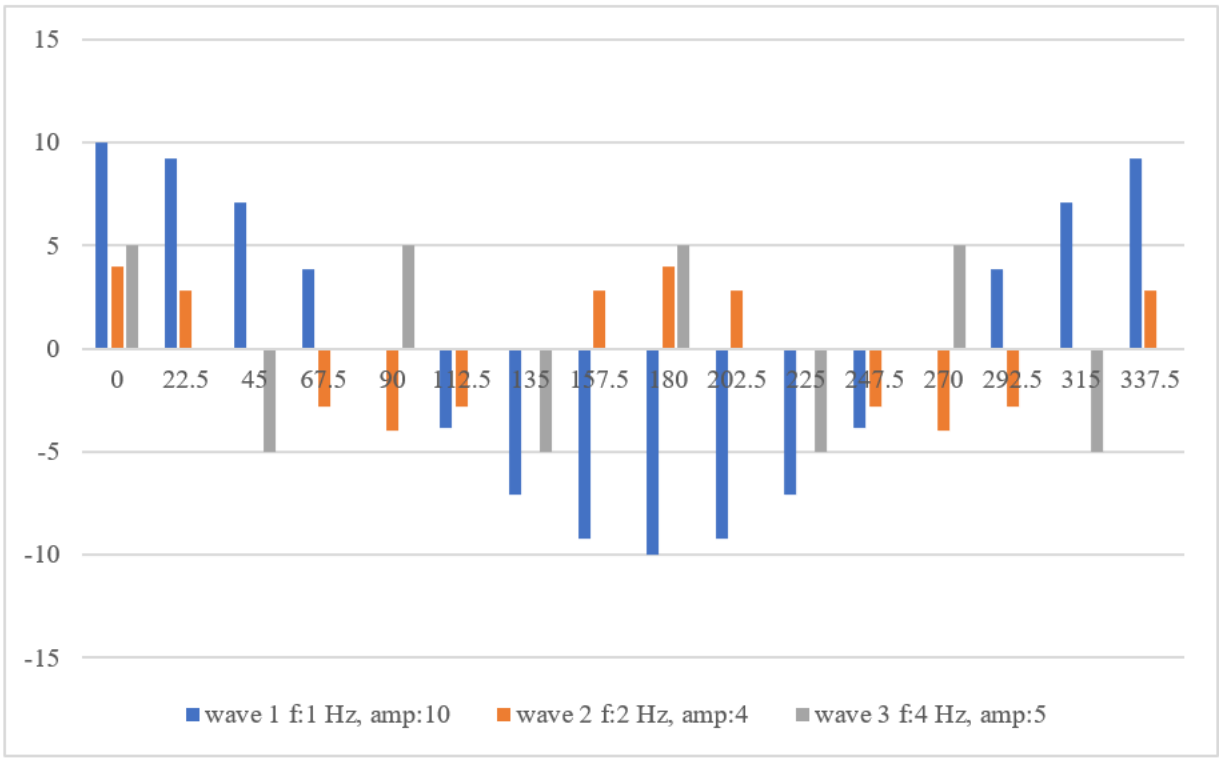


FIGURE 2. Summed waveforms.

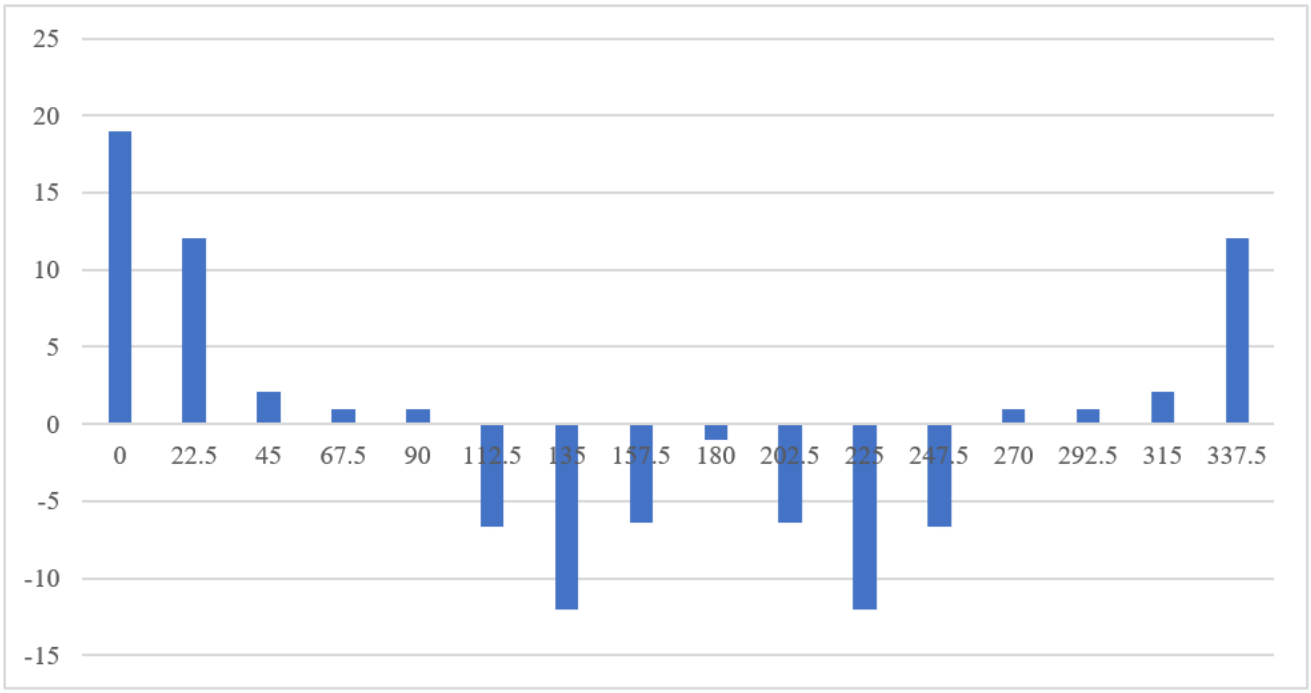
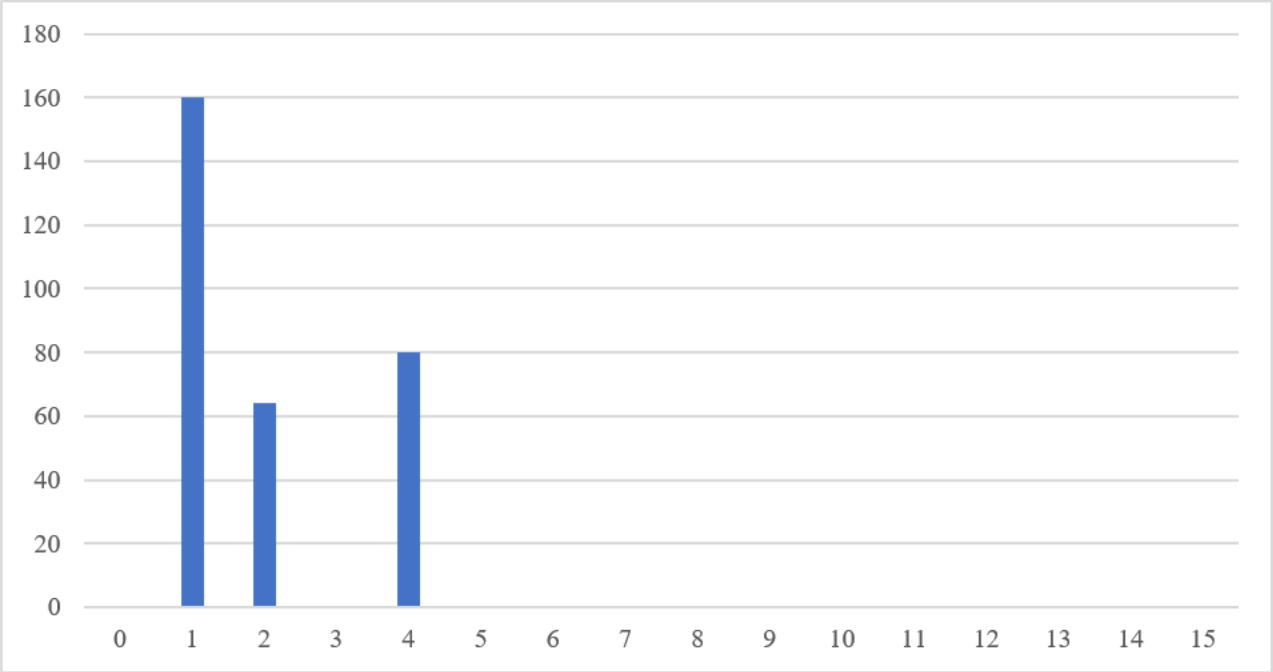


FIGURE 3. FFT results.



To test correctness, we perform the FFT and get the correct resulting FFT graph with the correct frequency values and relative amplitudes in the input waves. We will take the runtime average of ten runs, and use the average values to normalize runtime fluctuations. We use the same sample size to test the three implementations. We then compare the runtimes between all three implementations, and analyze the scalability and efficiency of the parallelized openMP algorithm.

3. Comparison of Serial and Iterative Serial Runtimes

Figure 4 shows the runtimes of the recursive and serial iterative implementations for an input size of 2^{22}. Recursion generally performs worse than its iterative counterpart due to stack maintenance overhead [1]. The iterative version is faster because the calculations involve iterating over local variables, i.e. there is a less stack maintenance overhead.

4. Parallel Iterative Implementation

The iterative algorithm was parallelized using the OpenMP library. Two **for** loops were parallelized. The code for these methods are on lines 50 - 124 in **fft.c**. Figure 5 shows the runtimes for varying number of threads.

FIGURE 4. Serial runtime vs. iterative serial runtime.

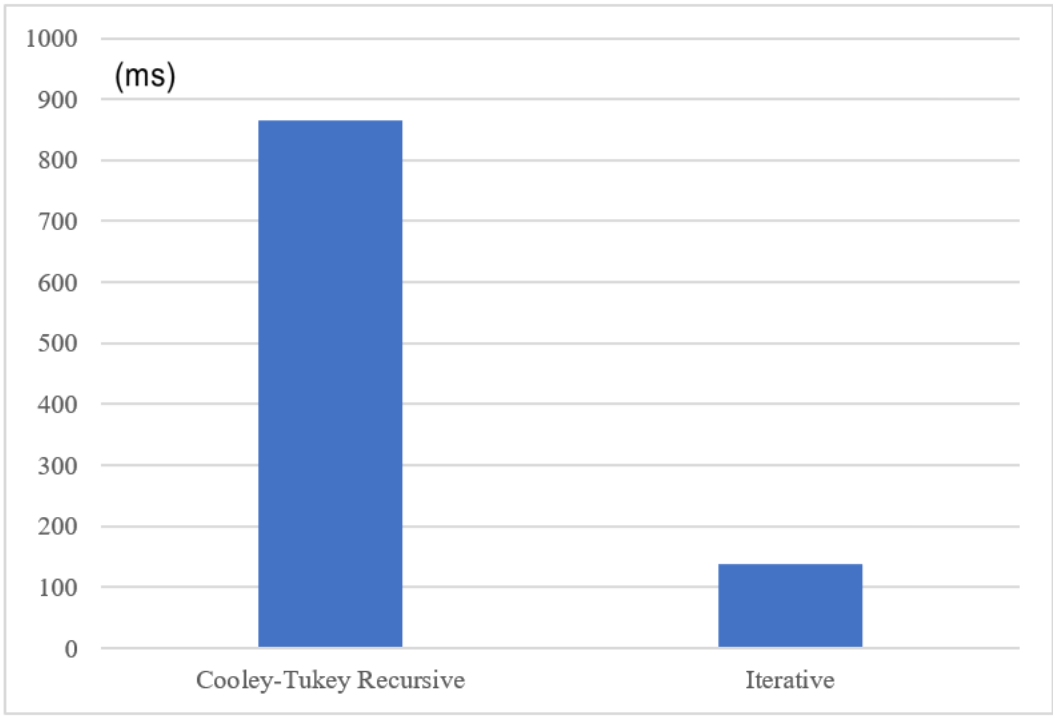
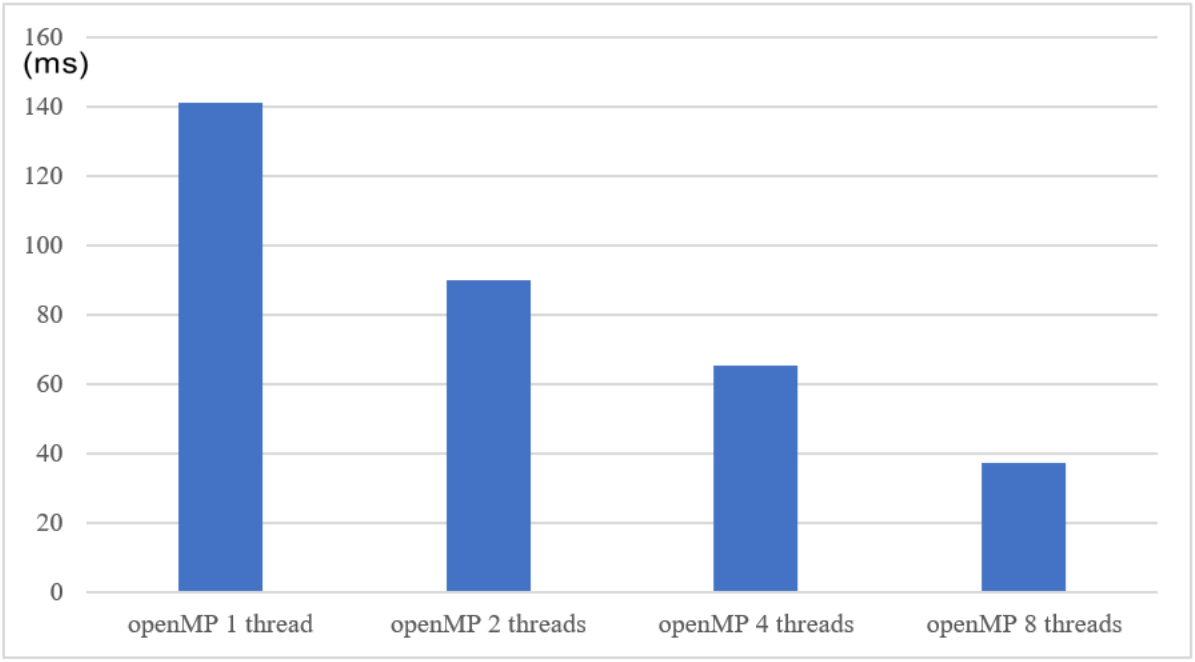


FIGURE 5. Parallel runtimes.



4.1. **Speedup and Efficiency.** Speedup and efficiency are two common metrics used for evaluating the performance of a parallel algorithm against the serial version [3]. Speedup is the improvement in parallel execution time over the serial implementation, as shown in Equation 1.

$$Speedup = \frac{T_{serial}}{T_{parallel}} \tag{1}$$

Efficiency is the relative improvement in execution time as the number of processes increase, as shown in Equation 2.

$$Efficiency = \frac{Speedup}{number\ of\ processes} \tag{2}$$

Table 1 reports the speedup and efficiency values for varying number of threads. Figures 6 and 7 depicts these values graphically.

TABLE 1. Speedup and Efficiency

Number of processes	Speedup	Efficiency
1	1.0	1.0
2	1.57	0.78
4	2.16	0.54
8	3.77	0.47

FIGURE 6. Speedup

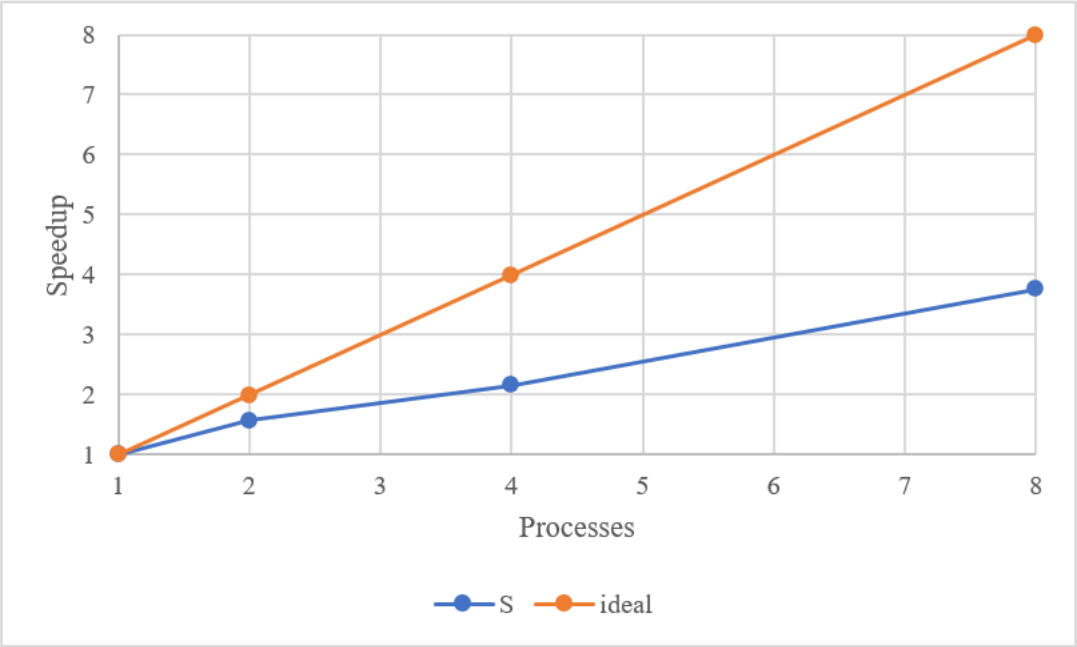
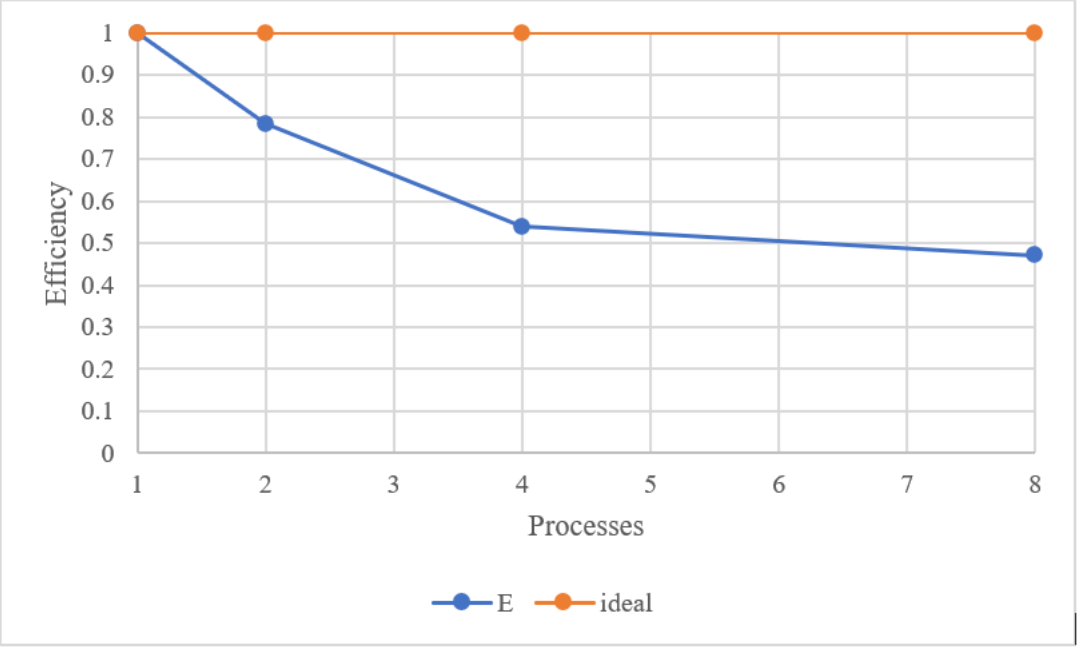


FIGURE 7. Efficiency



5. Conclusion

In this assignment, the serial recursive, serial iterative, and parallel implementations of the Cooley-Tukey Fast Fourier Transform algorithm were compared. Not surprisingly, the recursive version was the slowest. The serial iterative version was much faster due to less stack maintenance overhead. Parallelizing the iterative implementation improved the runtime even more as evidenced by the increased speedup metric. The key takeaways from the project are (1) prefer an iterative solution over a recursive one if performance is of prime importance and (2) consider parallelizing an iterative algorithm using OpenMP as a way to increase performance further without over-complicating the code.

References

[1] Randal Bryant and David R. O'Hallaron. *Machine-Level Representation of Programs*, page 253–254. Pearson, 2016.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Polynomials and the FFT*, pages 909–919. Mit Press, 2009.

[3] Peter S. Pacheco. *Parallel Hardware and Parallel Software*, page 58–61. Elsevier Science and Technology, 2013.

[4] Rshuston. Rshuston/fft-c: Fft implemented in ansi c.

Appendix A. Code Files

The three implmentations are contained in three separate folders according to Table 2.

Table 2. Code files

Implementation	Folder name
Serial	fft_serial
Serial iterative	fft_iterative
Parallel	fft_openMP

Appendix B. Building the Executables

Each implementation's folder contains a makefile. To build the executable for the serial implementation for example, run **make clean** followed by **make** in the fft_serial folder. The makefiles generate the following executables:

(1) **fft_serial/fft_serial.exe**

(2) **fft_iterative/fft_iterative.exe**

(3) **fft_openMP/fft_openMP.exe**

Appendix C. Running the Executables

Table 3 shows the command lines used to run the programs. The first argument of each program is the number of samples as a power of two. For example, to run a program with 16 samples, the first argument is 4. The second argument to the parallel implementation is the number of threads which must be a power of two. If **--print** is passed to any program as the last parameter, the input and output waveforms are printed to the console.

Table 3. Command lines

Implementation	Command
Serial	**./fft_serial [samples]**
Serial iterative	**./fft_iterative [samples]**
Parallel	**./fft_openMP [samples] [threads]**