CSE305 Project Report
Fast Fourier Transform

Francisco Moreira Machado, Khoa Do
Bachelor of Science
Ecole Polytechnique

June 13, 2024

1 Introduction

In this project, we present implementations and applications of the *Discrete Fourier Transform (DFT)*, focusing on exploring parallel implementation of the approaches. Our methods are based on the *Cooley-Tukey Fast Fourier Transform (FFT)* algorithm, a sequential algorithm that provides $O(n \log n)$ computation time complexity, which we expand to provide efficient parallel algorithms for the DFT.

We also implement two important applications of the DFT: Data Compression and Polynomial Multiplication. These serve to show the efficacy of our algorithm, but also to compare the performance of Fourier transform-based approaches against traditional methods. Additionally, we implement the *Number-theoretic Transform (NTT)* with analogous FFT algorithms, which allows us to accurately compute integer polynomial multiplication.

The code as well as all necessary instructions to run this project are in `https://github.com/fcomoreira1/CSE305-FFT`

2 Background

The DFT consists of a transformation on a sequence $(X_n)_{n=0}^{N-1}$ of complex numbers into another sequence of complex numbers $(Y_n)_{n=0}^{N-1}$ defined by

$$Y_k := \sum_{n=0}^{N-1} X_n e^{-i2\pi \frac{kn}{N}}. \tag{1}$$

We also define the *Inverse DFT (IDFT)* as the inverse transformation, which is given by

$$X'_k := \frac{1}{N} \sum_{n=0}^{N-1} Y_n e^{i2\pi \frac{kn}{N}}, \tag{2}$$

in which case, as expected, $(X_n) = (X'_n)$. We will denote these transformations as DFT and $IDFT$.

Conceptually, the DFT finds a "frequency domain" representation of the original data, representing the data as a combination of separate frequencies. Due to this property, we implemented a compression algorithm that retains the frequencies with the largest coefficients in absolute value. Later in this report, we show how this compression performs, especially when looking at time series data with periodic patterns.

Another important property of DFT transformation is the Convolution Theorem [2]. If we let the cyclic convolution operator $*$ be such that

$$((X_k)_{k=0}^{N-1} * (Y_k)_{k=0}^{N-1}) = \left(\sum_{l=0}^{N-1} X_l Y_{(k-l) \pmod N} \right)_{k=0}^{N-1},$$

and $\cdot$ be the pointwise product of sequences, then the theorem gives us that

$$X * Y = IDFT(DFT(X) \cdot DFT(Y)) \tag{3}$$

Moreover, one may recall that polynomial multiplication is closely related to convolution operations in sequences. Indeed, if we let $P(x) = \sum_{k=0}^{n-1} a_k x^k$, $Q(x) = \sum_{k=0}^{m-1} b_k x^k$, $N = n+m-1$ and we extend the sequence of coefficients $(a_k), (b_k)$ with 0's to have both size N, then we can observe that for (c_k) being the coefficients of PQ, we have $(c_k) = (a_k) * (b_k)$. Therefore, with an FFT of complexity $O(N \log N)$ we can use identity (3) to compute polynomial multiplication in a much better complexity than the straight-forward $O(N^2)$ approach.

An important variant of the DFT that we also explore in the context of polynomial multiplication is the NTT. The formulation of the forward and inverse NTT are very similar to the one of DFT, with the changes being that operations are taken mod p, a prime, and instead of $e^{i2\pi kn/N}$, we use ω^{kn} with ω being a primitive root mod p [4]. Therefore computing integer polynomial multiplication modulo p can be expressed in terms of this transform, which we extend by making p large enough to ensure the equality also holds in the ring of integer polynomials.

It is worth noticing that algorithms that work to compute DFT can be directly adapted for the NTT. For this transformation to yield the correct result, it must hold that N divides $p-1$, therefore part of an algorithm that uses NTT to multiply polynomials involves finding such a suitable p (which may be large). Therefore, picking a valid p yields an algorithm for integer polynomial multiplication that relies solely on integer arithmetic.

3 Implementation

For a general structure of the project, the DFT implementations are in `dft.cpp`, the NTT implementations are in `ntt.cpp`, the Polynomial Multiplication functions are in `polymult.cpp`, and Compressing functions are provided in `compression.cpp`. Additionally, we define a `IntegerModP` class that efficiently handles mod p arithmetic as well as essential functions for the NTT such as finding a primitive root mod p.

In this section, we will discuss the implementations of the FFT focusing on DFT, however for almost all approaches, there are alternative functions of the same name that take `IntegerModP` inputs and perform an NTT. Additionally, we write `Complex` as a shorthand for `std::complex<double>` from the complex `c++` library.

3.1 Cooley-Tukey's algorithm

Cooley-Turkey algorithm, named after J.W.Cooley and John Tukey, is one of the most popular Fast Fourier Transform (FFT) algorithms. To transform a sequence of composite length N, the algorithm decomposes N into a product of smaller number $N = N_1 N_2$, and recursively performs a smaller-length transformation to obtain a computation time $O(N \log N)$ for composites with small prime divisors. In this project, we implemented the radix-2 case of the algorithm, setting the length N to be powers of 2.

The sequential version of the algorithm (as well as the reverse transformation) is declared in the header file `dft.h` as follows, and implementation is found in `dft.cpp`:

```
void fft_radix2_seq_(const Complex *x, Complex *y, int n, int d = 1);
void ifft_radix2_seq_(const Complex *y, Complex *x, int n, int d = 1);
void fft_radix2_seq(const Complex *x, Complex *y, int n);
void ifft_radix2_seq(const Complex *y, Complex *x, int n);
```

The complex array `x` contains the values of the input sequence, and `y` will hold the Fourier coefficients. The number `n` is the length of the sequence and is a power of 2. As an implementation detail, our algorithm only uses constant memory overhead and does not make copies of the input sequence. For that, we use the parameter `d` to control the jump size of input sequences and help with the recursive calls. The wrapper functions (that do not end in underscores) are provided for experimenting convenience.

We refer to [3] for more information on the Cooley-Tukey FFT algorithm, as well as the radix2 case.

3.2 Two strategies to parallelize

3.2.1 Divide and Conquer approach

Our first approach to parallelize Cooley-Tukey's radix2 FFT algorithm is to apply the Divide-and-Conquer (DAC) principle when implementing the recursive calls, whenever the length `n` is large enough.

```
void fft_radix2_parallel_dac_(const Complex *x, Complex *y,
                              int n, int d) {
    // When n is small
    if (n <= 1024) {return fft_radix2_seq_(x, y, n, d);}

    // Recursive calls
    std::thread subthread = std::thread(&fft_radix2_parallel_dac_,
                                        x, y, n/2, 2*d);
    fft_radix2_parallel_dac_(x + d, y + n / 2, n / 2, 2 * d);
    subthread.join();

    // Merging
    ...
```

The idea is to reduce the time complexity of the expensive recursive calls by carrying them out in parallel. With that, we can theoretically obtain an algorithm that has complexity $O(n)$. However, the approach does not specify the number of threads one takes. As a result, to certain hardware, the creating and running of too many threads may reduce the performance of the function. Indeed, we do not parallelize the merging step, as we expect the recursive calls to create a large enough number of threads. We analyze the experiments on this approach in the following sections.

3.2.2 Even-load approach

Another approach we have is to specify the number of thread `n_thread < n` beforehand (which we constrain to be a power of 2) and treat the algorithm as in the radix-`n_thread` case:

```
void fft_radix2_parallel_our(const Complex *x, Complex *y,
                             int n, int n_thread) {
    // Trivial case
    if (n <= 128) {
        fft_radix2_seq_(x, y, n);
        return;
```

```
}

// Recursive calls
Complex *z = (Complex *) std::malloc(n*sizeof(Complex));
std::vector<std::thread> threads(n_thread-1);
for (int i = 1; i < n_thread; i++) {
    threads[i-1] = std::thread(&fft_radix2_seq_,
                x+i, z+i*n/n_thread, n/n_thread, n_thread);
}
fft_radix2_seq_(x, z, n/n_thread, n_thread);

// Syncing
for (int i = 1; i < n_thread; i ++) {threads[i-1].join();}

// Merging
...
```

The idea is that the majority of the workload is divided evenly to the threads, giving a near-optimal solution. This comes with the slight cost of merging the obtained output in the array `z`; we parallel the merging step with `n_thread` threads as well to counter this overhead.

The performances of this approach are analyzed in the following sections.

3.3 Bluestein's algorithm

Bluestein's algorithm [1] provides an efficient way to perform DFT with complexity $O(n \log n)$ for any length n that is not neccessarily powers of 2, albeit several constant slower than the Cooley-Tukey algorithm for composites. The algorithm is based on the following transformation:

$$nk = \frac{-(k-n)^2}{2} + \frac{n^2}{2} + \frac{k^2}{2}$$

Therefore we have that

$$y_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} = e^{-\frac{\pi i}{N} k^2} \sum_{n=0}^{N-1} \left(x_n e^{-\frac{\pi i}{N} n^2} \right) e^{-\frac{\pi i}{N} k^2}$$

The sum now becomes a convolution, and we can again apply the Convolution Theorem [2] to deduce an $O(n \log n)$ complexity using the implemented radix-2 algorithms.

We implemented the Bluestein's algorithm with a template that takes in radix2 fft functions to obtain fast calculation of the convolutions. The use of different versions (sequential, dac or even-load approach) of Cooley-Tukey algorithm yield different analogs of Bluestein's algorithm.

4 Experiments

Now that we have established an overview of the methods we develop, we will include and discuss multiple experiments and benchmarks we have done to test them. It is worth noticing that in general, we will use input sequences with a size that is a power of two because of the radix-2 variant. Elapsed-real times are measured using `std::chrono` and times included are obtained by averaging several runs. Additionally, the tests are compiled with `g++ -O3` flag and conducted in a machine with an Intel(R) Xeon(R) W-1290P CPU @ 3.70GHz, with 10 cores and 20 available threads.

4.1 Speed comparison of methods

We run the performance tests by picking random sequences of length N (X_k) (let us say each entry is picked from a uniform distribution in $[0, 1]$) and measuring the elapsed-real time to compute $IDFT(DFT((X_k)))$. We opt for this approach as it also helps ensure the correctness of our algorithms. For a baseline for this testing, we use a simple function to compute the DFT of an input sequence by strictly following the definition of the transform. Table 1 summarizes our findings.

N	Baseline	Seq	DAC	Our 2	Our 4	Our 8	Our 16
2^{14}	46359	2.47	1.58	1.45	0.84	1.07	1.62
2^{15}	202551	5.19	3.17	3.02	1.68	1.56	2.15
2^{16}	831266	11.34	4.83	6.10	3.37	2.16	3.08
2^{17}	-	22.55	9.14	13.11	6.97	4.40	5.39
2^{18}	-	46.91	17.00	27.97	14.66	8.86	11.01
2^{19}	-	103.58	33.76	59.19	32.25	18.87	23.02
2^{20}	-	258.32	71.79	144.74	74.48	42.82	48.95
2^{21}	-	589.92	139.69	349.59	183.76	105.96	105.44
2^{22}	-	1283.18	285.47	713.08	360.02	227.29	220.20
2^{23}	-	2701.44	574.34	1506.14	830.94	514.91	469.877
2^{24}	-	5647.56	1174.98	2939.63	1525.68	941.34	950.00
2^{25}	-	11626.40	2426.13	6136.62	3240.34	1942.23	1936.31
2^{26}	-	23807.20	5077.54	12768.00	6627.19	4104.59	3939.55

Table 1: *Benchmark of different FFT methods in milliseconds. Seq, DAC, and Our T, mean sequential radix-2, divide-and-conquer radix-2, and even-load radix-2 using T threads, respectively.*

From this table, the most noticeable fact is that the baseline is severely outperformed by Cooley-Tukey-based algorithms. We also remark that for all the methods we explored, the growth of the elapsed time is roughly linear, as expected.

Now if we focus on comparing the sequential and parallel methods, for larger input size we see about a 6 times improvement with Even-load 8/16 to the sequential one.

We point out that the DAC function has access to all 20 threads of the CPU, while we limit the one for the even load functions. Indeed, through the more controlled approach of even-load, we avoid the creation of a large number of threads, which is necessary for the DAC counterpart. We attribute this as the main reason to the consistently worse performance of DAC with relation to even-load with a larger number of available threads. Moreover, as we observe diminished gained in performance by going from 8 to 16 threads for the even-load method, we do not expect the possible extra 4 threads to make a significant performance improvement.

4.2 Compression

We experiment with the use of Fourier Transform-based methods on compressing and approximating Weather datasets, as they are usually composed of strong seasoning factors. We use 2 datasets: one is the Daily Climate time series data, to which we used the daily temperature of the full dataset with length 1471, and the other is the Max Planck Weather Dataset, to which we used only the first temperature measured per day, giving a time series of length 2921. both of the datasets taken from the website Kaggle.

The idea is to use either the Cooley-Tukey radix2 FFT algorithm, which works for a sequence of length being a power of 2, or Bluestein's algorithm, which works for a sequence of any integer length, to transform the sequence. Assuming the weather data is highly seasonal, after applying the Fourier Transform, for a small integer k, we keep k frequencies with the highest magnitude, and apply the reverse Fourier Transform to obtain the compressed version of the original data.

In the case of the Cooley-Tukey algorithm, we pad the sequence with zeros for a power of 2 length and remove the extra terms at the end of the process.

The results are shown below. Bluestein's algorithm runs slower than its counterpart but yields much less error for the same compression rate, due to the padding of zeros making it hard for the Fourier Transform to highlight the dominant frequencies.

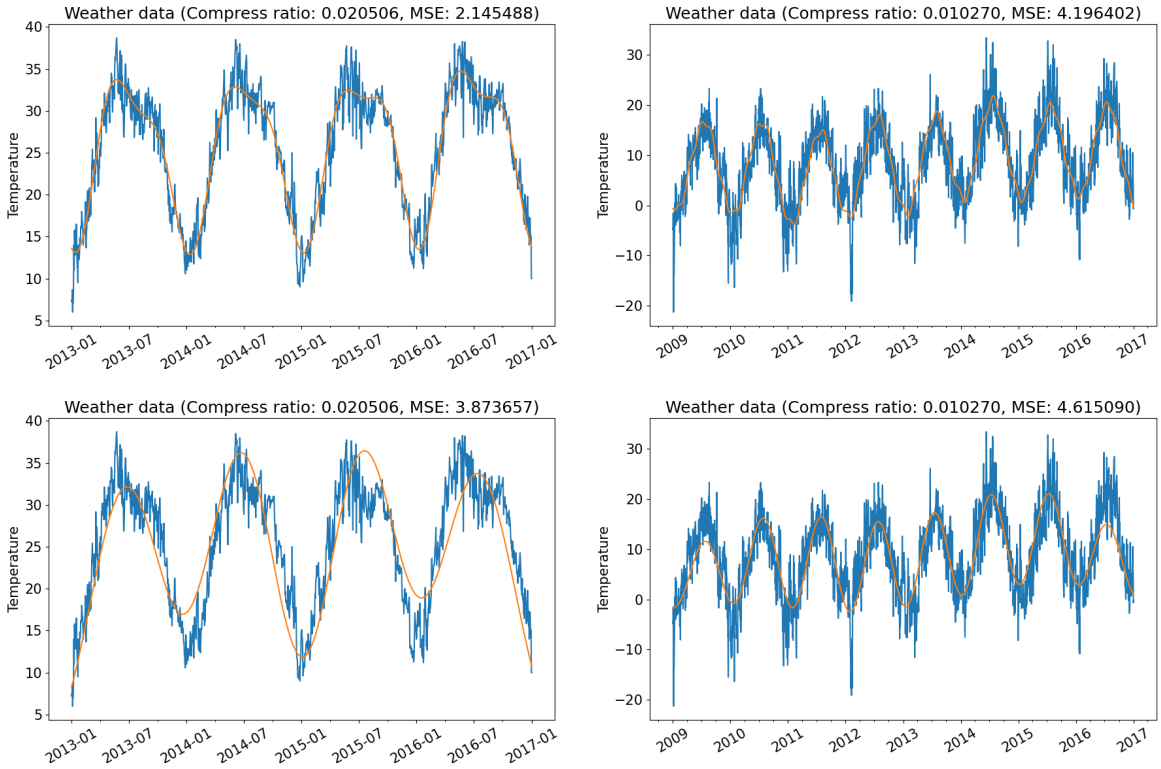

Figure 1: *Plot of compresses sequence for temperatures in the Daily Climate time series data dataset (first column) and the Max Planck Weather Dataset (second column), using Bluestein's algorithm (first row) and Cooley-Tukey radix-2 algorithm (second row). With the same compression rate, due to having to fit the padded zeros, the Cooley-Tukey radix-2 method performs worse than its alternative, albeit a little faster.*

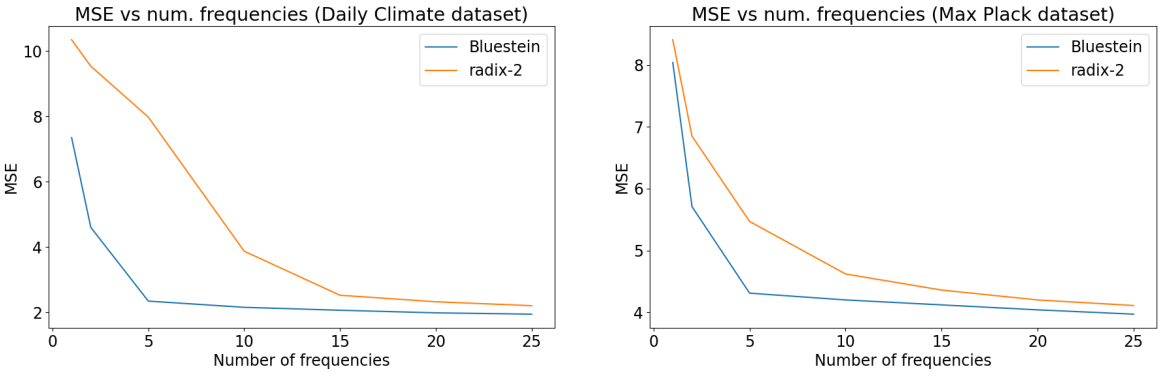

Figure 2: *Plot of MSE as function of the number of compression frequencies, for the Daily Climate time series data dataset (first column) and the Max Planck Weather Dataset (second column). The frequencies found by Bluestein's algorithm always yield lower MSE.*

The conclusion is that padding the sequence with zeros prevents the DFT algorithms from reading and outputting the correct dominant signals. This is expected behavior and is demon-

strated by the experiments.

4.3 Speed comparison in the context of Polynomial Multiplication

Now let us use the context of the problem of fast polynomial multiplication to test our implementations. Here we compare how the divide-and-conquer and even-load radix-2 FFTs compare to the straightforward quadratic polynomial multiplication algorithm. In particular, we also compare the application of FFT to DFT and NTT and assess their correctness. For inputs, we pick two integers polynomials with degree N a power of two, with coefficients randomly chosen from $[-10, 10]$ in order to avoid overflows related to the size of the prime p.

We recall from the background part, that to compute the product of two polynomials using DFT/NTT, we first compute the forward transform of both polynomials in the form of sequences, then the pointwise product of the transforms, followed by the inverse transform of the outcome. Here we include the elapsed-real time to run all these operations.

N	Baseline	Seq C	Seq N	DAC C	DAC N	Our 16 C	Our 16 N
2^{12}	15.75	2.61	22.36	1.64	6.08	1.53	4.40
2^{13}	61.50	4.33	40.59	2.53	7.98	2.30	6.86
2^{14}	245.63	9.03	86.53	4.59	14.20	3.98	12.50
2^{15}	996.71	18.69	182.08	9.64	29.77	6.75	24.76
2^{16}	3817.71	38.67	390.25	18.29	57.67	12.62	49.37
2^{17}	15198.60	79.73	400.83	35.82	112.68	24.45	100.65
2^{18}	63342.60	172.01	1747.89	70.54	228.31	43.01	211.98
2^{19}	-	433.93	3537.86	141.75	468.23	99.02	428.16
2^{20}	-	970.14	7619.90	255.40	954.68	196.69	900.86

Table 2: *Benchmark of integer polynomial multiplication in milliseconds. C means we use a DFT-based method, while N means an NTT-based method.*

We once again highlight the efficiency of all FFT-based methods compared to the baseline with a rapidly increasing elapsed-real time. If we first focus on the DFT-based methods, we observe a very similar comparison of performances between the Sequential, the DAC, and the Even-load methods.

More interesting to analyze, however, are the NTT transforms. It also holds for its implementations that all radix-2 methods severely outperform the baseline for larger inputs, and we have a clear improvement in elapsed-real time by taking the parallel methods. However, if we solely compare the NTT methods with the corresponding DFT, there are massive gaps in performance.

We attribute this partially to the usage of a user-defined `IntegerModP` in comparison with a highly optimized standard library type. Additionally, due to the constraints we discussed in the background section, the prime p gets large quickly as a function of the size of the polynomials and bound on coefficients. This implies in particular that computing the n-th primitive root modulo p is a relatively expensive call, as it consists of finding a primitive root and raising to the power $\frac{p-1}{n}$, which must be done at every recursive step of an NTT calculation. Moreover, each arithmetic operation is immediately followed by a $\pmod p$, which should add a significant overhead to the computations.

Finally, we also observe that the even-load approach once more outperforms the DAC. The different is less significant, however we believe that this is due to our lack of expertise in how to maximally optimize both methods as well as the `IntegerModP` arithmetic.

5 Conclusion

In this project, we have successfully implemented the Discrete Fourier Transform (DFT) with a focus on parallel processing. By leveraging the Cooley-Tukey Fast Fourier Transform (FFT) algorithm, we adapted its sequential $O(n \log n)$ time complexity to efficient parallel algorithms. This extension has shown significant improvements in computation speed and resource utilization, proving the viability of parallel approaches to DFT.

We highlighted the practical applications of the DFT for Data Compression by applying and analyzing two possible implementations, yielding slightly different compression results. Furthermore, we explore the Number-theoretic Transform (NTT) for a robust framework for integer polynomial multiplication. By applying FFT algorithms to the NTT, we study the accuracy and performance of this approach, as well as establish the shortcomings of the method in comparison with the DFT alternative to the problem.

If the opportunity presents in the future, we hope to come back and expand the project by optimizing the expensive prime operations of the NTT algorithm, benchmarking and optimizing more with Bluestein's algorithm, or implementing an analogue of Wolfram picture-curves.

6 References

[1] Wikipedia. Chirp Z-transform — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Chirp%20Z-transform&oldid=1171007425`, 2024. [Online; accessed 13-June-2024].

[2] Wikipedia. Convolution theorem — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Convolution%20theorem&oldid=1226718213`, 2024. [Online; accessed 13-June-2024].

[3] Wikipedia. Cooley–Tukey FFT algorithm — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Cooley%E2%80%93Tukey%20FFT%20algorithm&oldid=1209251205`, 2024. [Online; accessed 13-June-2024].

[4] Wikipedia. Primitive root modulo n — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Primitive%20root%20modulo%20n&oldid=1222449691`, 2024. [Online; accessed 13-June-2024].