Parallel Fast Fourier Transform in C++

Pedro Cabral

June 2023

1 Introduction

In this project, we present a parallel implementation of the Fast Fourier Transform (FFT) algorithm in C++ [1]. We compare three DFT transform algorithms: the naive $O(n^2)$ algorithm, the recursive 2-radix $O(n \log n)$ algorithm, and an iterative $O(n \log n)$ algorithm. Moreover, we compare two parallelization methods: one implemented with the standard C++ functionality and one using the OpenMP library. We explore all the implementation details in Section 2.

We also implement two applications of our FFT code. In Section 3, we discuss our implementation of a Time Series Compressor using the FFT algorithm. Additionally, in Section 4, we present our code that multiplies two integer polynomials without rounding errors. We implement the FFT algorithm modulo primes to accomplish this.

All the instructions on compiling and using the files are in the README located in the GitHub repository.

2 FFT Implementation

2.1 Sequential Implementation

We implement our DFT functions using similar function signatures to the STL ones. We pass iterators to our DFT functions specifying the sequence to transform and the destination where the output should be stored. Each DFT function is of the form:

```
template <typename InputIt, typename OutputIt>
void DFT(InputIt first, InputIt last, OutputIt d_first);
```

We also ensure the function works when `first` and `d_first` point to the same memory location.

Our Inverse Discrete Fourier Transform (IDFT) shares the implementation with the DFT. Both functions internally call the `ImplDFT` function, which adjusts the root of unity to be used based on each case.

Each of the three methods has the same names `DFT/IDFT`, but are in different namespaces: `base_dft, recursive_fft, iterative_fft`. Each namespace corresponds to a different implementation of the DFT algorithm. This same convention is kept for the parallelized code implementation.

[1] GitHub repository: https://github.com/Gompe/FFT_CSE305

2.2 The Parallelizer Abstraction

We introduce a new abstraction called the `Parallelizer` for the parallel implementation of the DFT algorithms. All parallelization in the DFT code comes from two specific situations:

1. Performing a for loop in parallel.

2. Calling functions in parallel in a Divide-and-Conquer algorithm.

Therefore, a `Parallelizer` is a `struct/class` that implements the functions:

```
void parallel_for(int first, int last, std::function<void(int)> func);
void parallel_calls(std::vector< std::function<void(void)> > funcs);
```

The first function calls the function `func` for all integers in the range `[first, last)` in parallel. The second function calls each function in the container `funcs` in parallel. The only constraint we assume is that the specified operations above can be performed in parallel without race conditions.

We implement the `Parallelizer` abstraction in two different ways. The first way, which we call the `FixedThreadsParallelizer`, keeps track of the number of threads currently running in an `std::atomic<int>` and the maximum allowed number of threads that it can create. By default, it relies on the `std::thread::hardware_concurrency()` function to decide the number of threads it will use. In the `parallel_for` function, it simply splits the range `[first, last)` equally among the available threads. On the other hand, the function `parallel_calls` is more interesting. We implement a Thread Safe Queue and push all functions to be called into the queue. Then the threads evaluate functions one by one and pop new functions from the queue when they are done. This process continues until the queue becomes empty.

The second implementation of the `Parallelizer` is the `OmpParallelizer`. Here we use the compiler pragma:

```
#pragma omp prallel for
```

before executing any loop.

2.3 Parallel Algorithms

For the parallel implementation, we expand our function signature to:

```
template <typename InputIt, typename OutputIt, typename Parallelizer>
void ParallelDFT(InputIt first, InputIt last, OutputIt d_first,
                 const Parallelizer& parallelizer);
```

Implementing the parallel versions of the algorithms is similar to the sequential ones. The difference is that we use the `parallelizer` functionality in the key places where operations can be performed in parallel.

2.4 Comparison

In this section, we compare the runtime of each algorithm for a wide range of input sizes [2]. Table 1 summarizes the results. Notice that for a small input size, thread creation's overhead does not compensate for the parallelism gained from multithreading. Indeed, for inputs of size up to 2^12, the sequential iterative algorithm is by far the best. For larger inputs, the multithreaded algorithms start to outperform the sequential ones. Notice that the `OmpParallelizer` and the `FixedThreadParallelizer` produce comparable runtimes. Moreover, notice that the Iterative Algorithm significantly outperforms the Recursive Algorithm for large inputs. We conjecture that there are two reasons for this:

1. The for loop only requires creating threads at the beginning of its execution.
2. It is much harder to balance the work done by each thread evenly in the recursion case.

Algorithm	2^6	2^{12}	2^{18}	2^{24}
SeqBase	49us	159,068us	∞	∞
SeqRecursive	6us	460us	55.6ms	5.1s
SeqIterative	**4us**	**377us**	57.0ms	4.8s
FixedThreadsBase	439us	17,640us	∞	∞
FixedThreadsRecursive	85us	1,464us	102.0ms	7.7s
FixedThreadsIterative	984us	2,0103us	37.3ms	**3.1s**
OmpBase	2,081us	16,723us	∞	∞
OmpRecursive	48us	2,057us	104.4ms	7.2s
OmpIterative	13us	2,211us	**23.2ms**	3.5s

Table 1: Comparison of the proposed methods for a different number of input sizes.

3 Time Series Compression

We used the FFT algorithm as a tool to compress periodic-like time series. Our `compressor` code has two main functionalities: `compressor::Compress` and `compressor::Decompress`. The `Compress` function takes a time series and a desired `num_frequencies`. It performs the FFT on the data and keeps only the `num_frequencies` frequencies with the largest amplitude. Notice that this compresses the data, as only `sizeof(int) + 2 * sizeof(FloatType)` bytes are needed to store each frequency data: (The int represents the frequency, and the 2 FloatType represents the Phase/Amplitude of this frequency). Therefore, the compressed data size is determined uniquely by the provided `num_frequencies`.

Our code takes data from stdin, compresses the data internally, reconstructs it, and then outputs the reconstruction to stdout. We provide a Python file to plot the comparison between data and reconstruction.

[2]Experiments performed on an Intel(R) Core(TM) i7-10870H CPU @ 2.20GHz, 2208 Mhz, **8 Core(s), 16 Logical Processor(s)**. The code was compiled with GCC and the -O2 flag.

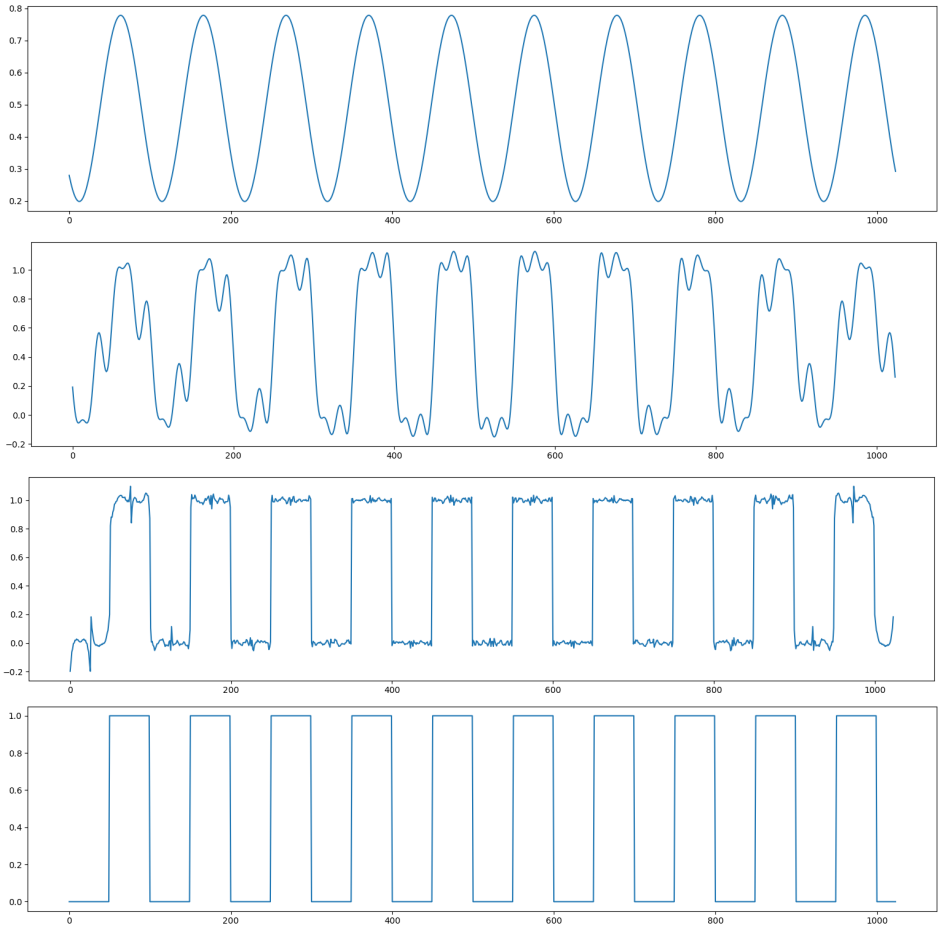

Figure 1: Compression results. From top to bottom: 2 frequencies (MAE: 0.331), ten frequencies (MAE: 0.131), 200 frequencies (MAE: 0.017), and original data. The data consists of 1024 data points.

4 Integer Polynomial Multiplication

One of the main applications of the FFT algorithm is fast polynomial multiplication. Assuming that we have two polynomials (with complex coefficients), $P(X) = p_0 + p_1 X + ... + p_n X^n$ and $Q(X) = q_0 + q_1 X + ... + q_n X^n$ of degree at most n, the multiplication formula:

$$r_k = \sum_{i=0}^{k} p_i q_{k-i}$$

gives us the coefficients of the polynomial $R(X) = P(X)Q(X)$ (assuming that the coefficients are 0 for indices out of range). This formula computes the product of two polynomials in $O(n^2)$.

With the Fast Fourier Transform algorithm, we can get a better complexity. Langrange's interpolation gives us a way to discover the coefficients of a polynomial R of degree N from the evaluation of R at $N + 1$ points $R(y_1), ..., R(y_{N+1})$. Moreover, notice that multiplying

N values $P(y_1)Q(y_1), ..., P(y_{N+1})Q(y_{N+1})$ can be done in $O(N)$. Therefore, if we find a fast way to evaluate P and Q at $N+1$ points and then convert the evaluations to the coefficients, we end up with a polynomial multiplication algorithm. The FFT is ideal for that because the FFT applied at polynomial coefficients is equivalent to evaluating the polynomial at the roots of unity ω_n^k where $\omega_n = \exp(2i\pi/n)$, which can be done in $O(n \log n)$.

Using our FFT code, we implement this algorithm in the `polynomial.h` file. Even though this algorithm works well with floating points and complex coefficients, it is unreliable for integer polynomial multiplication. Indeed, the intermediate conversions to floating point arithmetic inside the FFT algorithm may result in rounding errors. As a result, it would be possible to output integer coefficients that would be wrong by a small amount. In this section, we describe our implementation of fast integer polynomial multiplication. Our code performs the entire multiplication without using floating points.

We follow the approach of CLRS and compute the FFT modulo prime p. In this case, the roots of unity are derived from primitive roots of unity modulo p. We implemented the FFT modulo prime in the `modular_fft.h` file and called it *Modular FFT*. All number theory functionality is implemented in the file `number_theory.cc`. We can repeat the algorithm above from the Modular FFT to compute the multiplication of integer polynomials P, Q modulo a prime p. However, this is not enough to discover the coefficients of PQ. Indeed, any coefficient is only known modulo p. We solve this issue by performing modular FFT with two primes p_1, p_2. Moreover, we take primes p_1, p_2 such that their product is larger than 2^{33}. Therefore, by the Chinese Remainder Theorem, there is a single coefficient satisfying the two modulo relations that fit in the interval $[-2^{32}, +2^{32}]$, which is the range of `int`. With our `ChineseRemainderTheorem` function, we recover the `int` coefficients.

As a remark, many of these algorithms require calling FFT multiple times without data dependencies between different calls to FFT. In all these cases, we were careful to make those calls in parallel. For example, when we need to compute the DFT of P and Q, we perform these steps in parallel. We observed that the code runs about 40% faster when two different threads are used for these operations. For the Integer Polynomial Multiplication, we managed to get a $3\times$ speedup by using parallelism. We compute the Modular FFT of P, Q with respect to p_1 and p_2 in parallel. So we have 4 threads running concurrently. Moreover, we need to call `ChineseRemainderTheorem` for each coefficient of PQ to convert their remainders to an actual value. We perform this expensive for-loop using the `parallel_for` construct. This change alone made the code 30% faster.

Notice that the real polynomial multiplication will always be faster than the integer polynomial multiplication. Indeed, the algorithm for the real case is much simpler and performs fewer operations. Moreover, in our experiments, we noticed that rounding the output of the real polynomial multiplication gave the correct results with integer polynomials. Therefore, the integer polynomial multiplication for 32bit integers has more theoretical value than practical value. See Table 2 for a comparison of the runtime of each algorithm.

As a side note, the integer polynomial multiplication algorithm was hard to debug. For large degrees, the coefficients of PQ will almost surely overflow the 32 bits of `int`. This did not produce any visible errors, so it was not visible where the problem was coming from. For this reason, the current code has the pragma

```
#pragma GCC optimize "trapv"
```

that aborts the program whenever an overflow occurs.

Algorithm	2^6	2^{12}	2^{18}	2^{24}
Naive Multiplication	**46us**	72,105us	∞	∞
Integer Multiplication	814us	10,148us	741.7ms	60.5s
Real Multiplication	97us	**2,790us**	**297.0ms**	**24.8s**

Table 2: Comparison of the proposed methods for polynomial multiplication for a different number of input sizes.