Project Report

Parallel Fast Fourier Transform

Elouan Gros

Rémy Seassau

Nazila Sharifi Amina

November, 2022

1 Fast Fourier Implementation

We based our implementation of the FFT algorithm off of the book by Jeff Erickson [2]. More specifically, we use the Cooley-Tukey radix-2 fast Fourier transform algorithm based on the divide-and-conquer paradigm.

Our methodology was to first implement the algorithm and its inverse before focusing on parallelism. We give in the following subsections a reminder of the algorithm and present some implementation details in C++.

1.1 Forward

We give the pseudocode of the implemented algorithm in the appendix (Algorithm 1). Notice that the input is an array of reals while the output array has values in $\mathbb{C}$. We elect for our implementation to use a vector of `double` for the input and a vector of `std::complex<double>` for the output. Thinking ahead to the parallelization, we store the result in a vector taken as an input by reference. Finally, note that we take our input to be of size 2^k by padding the input with 0 if necessary.

1.2 Inverse

We compute the inverse again according to [2]. The pseudocode is included in the appendix (Algorithm 2) with the differences between the inverse and the forward direction highlighted in red. Note here that both the input and output in the implementation use the `std::complex<double>` type.

1.3 Parallelizing

We first notice that the recursive calls are done in such a way that we need the result of both the recursive calls before we can finish a function call. Paired with this is the idea that we want our threads to divide the work as much as possible. In order to achieve this, we start our recursive calls with a number of threads that we want to use. While we have more than two threads available, we use a new thread to do one of our recursive calls.

We illustrate in Figure 1 the successive recursive calls of each thread where n is the size of the input and m is the number of threads at our disposition. Each node represents the number of threads available to a thread at that point. We also provide the pseudocode for this parallelization in (Algorithm 3) where the parallelization additions are highlighted in blue.

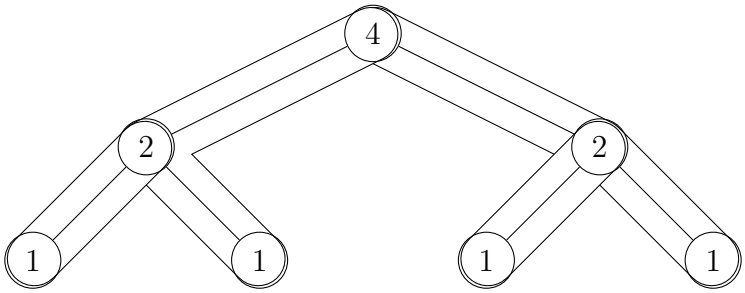


Figure 1: Representation of thread creations with $n = m = 4$

2 Benchmarking

2.1 Prediction Accuracy

We decide to use the FFT to "compress" the weather data of Taourirt Izakarn, Morocco. Indeed, taking the weather data, we start by compute the FFT on it. We are then left with an array of complex coefficients. We can then eliminate a percentage of the coefficients by sorting them by importance (since we are in $\mathbb{C}$, we just take the norm to get the importance). Then computing the inverse FFT on our filtered array gives us an approximation of the data that can be stored with less coefficients. We illustrate this process with Figure 3.

Using the same data, we can compute for different percentage thresholds the absolute and relative error of our approximation. We give a plot of this data in Figure 2.

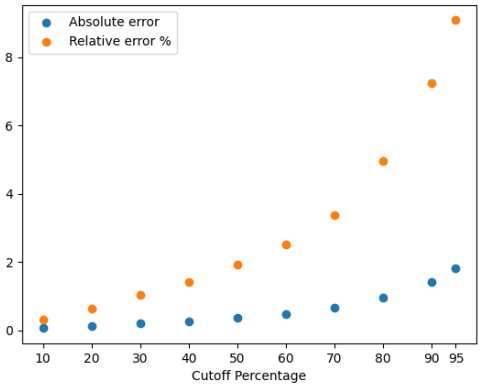
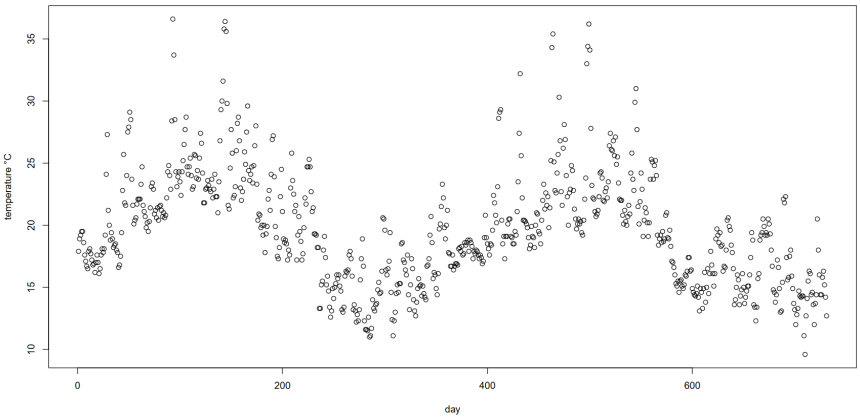
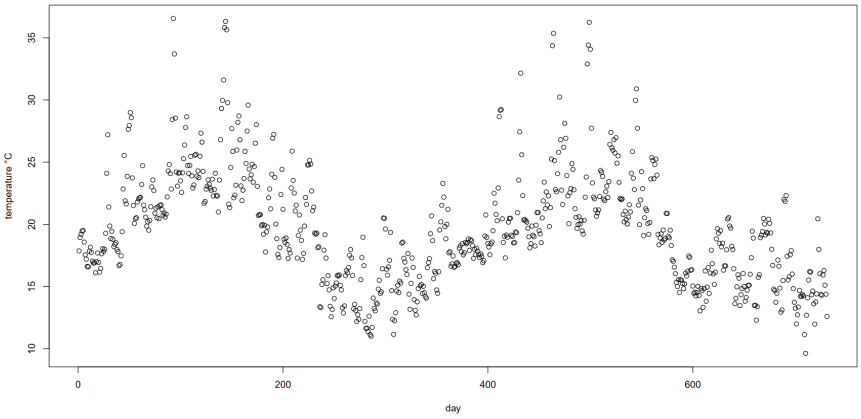


Figure 2: Absolute and relative error percentage of our approximation

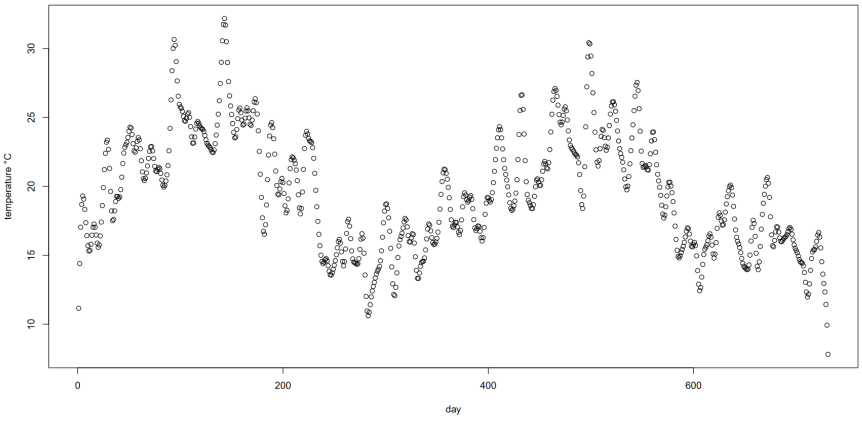
Figure 3: Approximation of Temperature data at Taourtit Izakarn



(a) Full data



(b) Data recovered from eliminating 10% of the coefficients



(c) Data recovered from eliminating 90% of the coefficients

2.2 Computation Speed

In order to test the speed of computation we consider the problem of polynomial multiplication. Indeed, this problem involves a straightforward application of the FFT and its inverse with a very minimal step in between the two.

An important note is the decision we made earlier of padding our vectors until they reach size 2^k for some $k \in \mathbb{N}$ means that we are better off testing on polynomials of size 2^k since the polynomials of size $2^k < n < 2^{k+1}$ will take about the same computation time as those of size 2^{k+1}. Computing the time for various sizes of the polynomials and for numbers of threads between 1 and 8, we see an emerging pattern that can be observed in Figure 4.

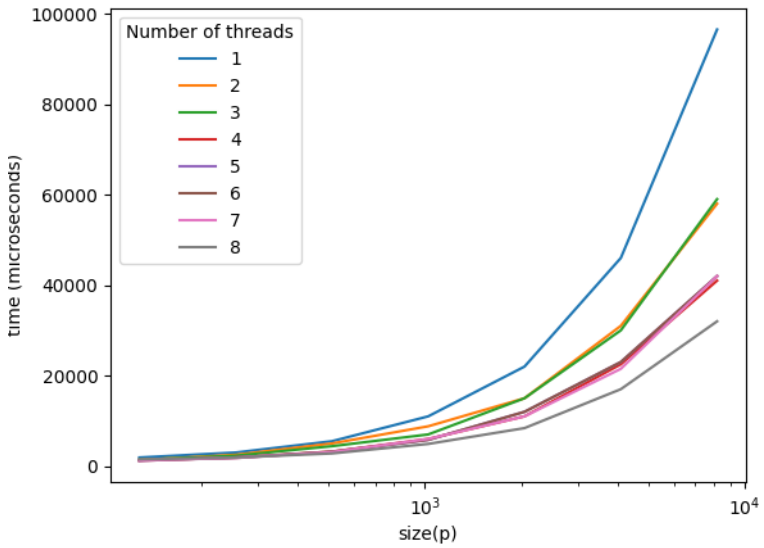


Figure 4: Time of computation over size of the input (log scale)

Indeed, when the number of threads m is not a power of 2, we find that it shares similar times as when the number of threads is the largest k such that $2^k \leq m$. Thinking back to Figure 1, this makes sense. Indeed, consider Figure 5. Notice that in the presented case the entire right half of the tree is left to one thread. Because we cannot compute the top-level without the result of both recurrent calls, the time it takes is equal to the maximum of the children. It should now make sense that we only see improvements in runtime when the number of threads reaches a power of 2.

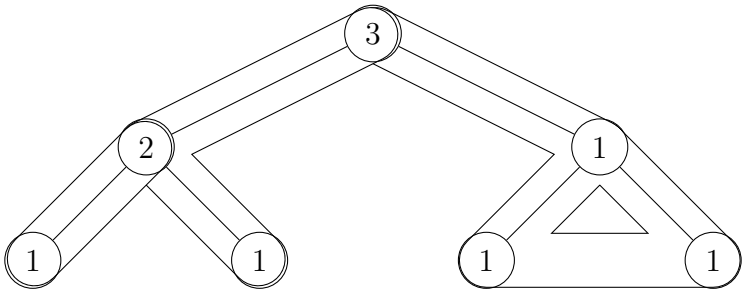


Figure 5: Representation of thread creations with $n = 4$ and $m = 3$

Fortunately this can be remediated by choosing a smarter parallelization, such as the one given in Section 4 of [1]. Indeed, this requires shifting away from a recursive FFT algorithm into an iterative one where it is easier to move the order of operations around to ensure that each thread can compute a more or less equal amount of operations. We would expect to see an increase in performance for every additional thread in such a way that the performance of the well parallelized FFT should be about equal to our naively parallelized FFT when the number of threads is a power of 2.

Nota Bene:

The code for this project can be found on the github page:

`https://github.com/El-BG-1970/fast_fourier`

Please note that the contributions are not necessarily reflective of how much work was done by each person as we often worked in a group on the same computer.

References

[1] Amir Averbuch et al. "A parallel FFT on an MIMD machine". In: *Parallel Computing* 15.1 (1990), pp. 61–74. ISSN: 0167-8191. DOI: https://doi.org/10.1016/0167-8191(90)90031-4. URL: https://www.sciencedirect.com/science/article/pii/0167819190900314.

[2] Jeff Erickson. *Fast Fourier Transforms*. URL: https://jeffe.cs.illinois.edu/teaching/algorithms/notes/A-fft.pdf.

A Algorithms

Algorithm 1: radix2fft

Input: An array P of size n

if $n = 1$ **then**
  return P

U = array(n/2);
V = array(n/2);
for $j \in [0, n-1]$ **do**
  U[j] = P[2j]; `// U = P[::2]`
  V[j] = P[2j+1]; `// V = P[1::2]`

U^* = radix2fft(U); `// Recursive calls`
V^* = radix2fft(V);

P^* = array(n);
$\omega_1 = e^{2\pi i/n}$; `// nth roots of unity`
$\omega = 1$;
for $j \in [0, n/2-1]$ **do**
  P^*[j] = $U^*[j] + \omega V^*[j]$;
  P^*[j+n/2] = $U^*[j] - \omega V^*[j]$;
  $\omega = \omega \cdot \omega_1$;

return P^*;

Algorithm 2: invradix2fft

Input: An array P of size n

if $n = 1$ **then**
  return P

U = array(n/2);
V = array(n/2);
for $j \in [0, n-1]$ **do**
  U[j] = P[2j]; `// U = P[::2]`
  V[j] = P[2j+1]; `// V = P[1::2]`

U^* = invradix2fft(U); `// Recursive calls`
V^* = invradix2fft(V);

P^* = array(n);
$\omega_1 = e^{-2\pi i/n}$; `// inverse nth roots of unity`
$\omega = 1$;
for $j \in [0, n/2-1]$ **do**
  P^*[j] = $(U^*[j] + \omega V^*[j])/2$;
  P^*[j+n/2] = $(U^*[j] - \omega V^*[j])/2$;
  $\omega = \omega \cdot \omega_1$;
return P^*

Algorithm 3: parallelfft

Input: An array P of size n, a number of threads m

if $n = 1$ **then**
  return P

U = array(n/2);
V = array(n/2);
for $j \in [0, n-1]$ **do**
  U[j] = P[2j]; `// U = P[::2]`
  V[j] = P[2j+1]; `// V = P[1::2]`

if $m \geq 2$ **then**
  t = spawn thread;
  U^* = t(parallelfft(U, $m/2$)); `// Run one of the calls in a thread`
  V^* = parallelfft(V, $m/2$);
  t.join();
else
  U^* = parallelfft(U, 1); `// If we ran out of threads`
  V^* = parallelfft(V, 1);

P^* = array(n);
$\omega_1 = e^{2\pi i/n}$;
$\omega = 1$;
for $j \in [0, n/2-1]$ **do**
  P^*[j] = $U^*[j] + \omega V^*[j]$;
  P^*[j+n/2] = $U^*[j] - \omega V^*[j]$;
  $\omega = \omega \cdot \omega_1$;

return P^*;