Parallel fast Fourier transform

Emanuel Ravemyr

Christian L. Thunberg

1 Abstract

In this report a parallel version of the FFT is implemented and analysed. The FFT-algorithm used in this report is the: sequential, unordered, radix-2 Cooley-Tukey algorithm for one dimension using Binary-Exchange for parallelisation. The computational time of the algorithm and its speedup is reported. The algorithm is implemented using MPI and applied on the sound data from the WAV-version and MP3-version of a sound file. The obtained data sets of the frequency spectrum are compared.

It turns out that the chosen algorithm suited for parallelisation and it is noted that the WAV-file has a broader frequency spectrum than the MP3-file of the same sound. Likely a consequence of the MP3-file being compressed while the WAV-file is not.

2 Background

Fourier Transforms are a central concept in mathematics, with strong applications in Signal processing and data science. It allows for investigating the properties of signals from the perspective of frequencies instead of time, in which it is often captured. It also allows for some challenging computations, as convolutions under Fourier transforms are simple multiplications.
In numerical applications, Fourier transforms are still of interest. A discrete version can be computed in $\mathcal{O}(N^2)$, however, there is a faster way. This is the FFT (Fast Fourier Transform). The perhaps most known implementation is that of Cooley-Tukey which is a radix-2 recursive method, which runs in $\mathcal{O}(Nlog_2(N))$.

As FFT is of interest in many applications, it is of course of interest to many to create even faster running algorithms, often by means of parallelisation. There are many works on parallel FFT, and some of these implementations are easily accessible for use, for example FFTW [1] (Fastest Fourier Transform in the West) which is an open source FFT library for C and Fortran. and cuFFT which is a FFT library for CUDA, which is a language for GPU programming.

2.1 Zero-Padding

In many applications, it is interesting to look at the signal of interest with a higher resolution than normally achieved. This can be done by adding zeroes to the array, e.g. zero padding. Not only does this allow for a closer look at the signal, but it also allows us to ensure that the computed array is always of length $N = 2^p$, which is desirable for most implementations of FFT, as it allows for efficient computational strategies.

3 Method Development

FFT can be implemented in many ways, both sequentially and recursively. For this project, we have chosen to implement the sequential, unordered, radix-2 Cooley-Tukey algorithm for one dimension using Binary-Exchange for parallelisation. This decision is based upon the availability of literature and how well suited the algorithm is for parallel implementation. The algorithm works by manipulating the binary representations of the indexing integers to find the pairs which to add to deliver the FFT. This is done in $\mathcal{O}(Nlog_2(N))$ operations according to the following algorithm:

```
1.   procedure ITERATIVE_FFT(X, Y, n)
2.   begin
3.      r := log n;
4.      for i := 0 to n - 1 do R[i] := X[i];
5.      for m := 0 to r - 1 do        /* Outer loop */
6.          begin
7.             for i := 0 to n - 1 do S[i]:= R[i];
8.             for i := 0 to n - 1 do /* Inner loop */
9.                 begin
     /* Let (b0b1 ··· br -1) be the binary representation of i */
10.                   j := (b0...bm-10bm+1···br -1);
11.                   k := (b0...bm-11bm+1···br -1);
12.                   R[i] := S[j] + S[k] × ω^(bm bm-1···b0 0···0);
13.            endfor;    /* Inner loop */
14.     endfor;           /* Outer loop */
15.     for i := 0 to n - 1 do Y[i] := R[i];
16.  end ITERATIVE_FFT
```

Figure 1: Pseudo code of the unordered radix-2 Cooley-Tukey fft-algorithm for one dimension. $\omega = e^{2\pi i/N}$. (The algorithm is taken from [2].)

Where N is assumed to be a power of two. The algorithm iterates from 0 to $log_2(N)$ in the outer loop, and each inner loop contains an order $\mathcal{O}(N)$ operations, meaning that the algorithm is in order $\mathcal{O}(Nlog_2(N))$ which is expected from FFT. In each step of the outer loop, we find the "partner" of each element in the array by changing the m-th most significant bit in the binary representation of the index of the element. This can be seen in the following illustration from the book [2]

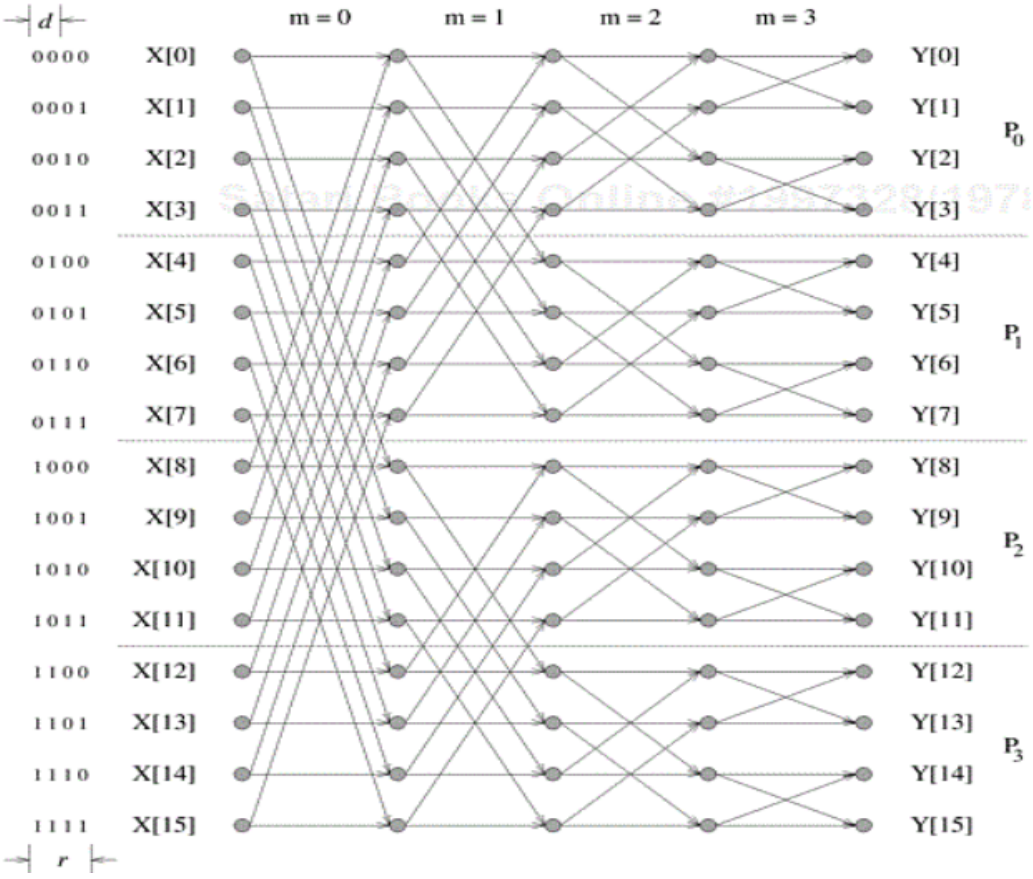

Figure 2: Butterfly pattern with binary representations of indices. (The Illustration is taken from [2].)

Introducing parallelisation in this algorithm is not very challenging. We can simply distribute the N data points linearly over the P processors, leaving each processor with N/P points. Here we assume that $P = 2^d$. A consequence of this is that we only have communication in the first d iterations of the outer loop, as we find the communication partners of an element by manipulating the m-th most significant bit. After d iterations, we no longer need to fetch elements from outside of the processor itself. In each of these communication steps, N/P points of data are exchanged. Using this, we can create an expression for the computational time of the parallel Binary-exchange FFT algorithm.

$$T_p = d(t_s + t_{comm}N/P) + rt_cN/P) \tag{1}$$

Where t_s is the startup time for communication, t_{comm} is the time it takes to transfer one piece of data and t_c is the computation time of one FLOP. Since $d = log_2(P)$ and $r = log_2(N)$, and the fact that our serial algorithm is $T_s = t_cNlog_2(N)$, we find that the speedup is

$$T_{sp} = \frac{T_s}{T_p} = \frac{t_cNlog_2(N)}{t_clog_2(N)(N/P) + log_2(P)(t_s + t_{comm}N/P)} \tag{2}$$

Which can be simplified into

$$\frac{t_clog_2(N)P}{t_clog_2(N) + log_2(P)t_{comm} + Plog_2(P)t_s/N} \tag{3}$$

Taking the limit as $P \to N$ we get

$$T_{sp} \to \frac{1}{1/N + (t_s/(Nt_c) + t_{comm}/(Nt_c)} = \frac{N}{1 + t_s/t_c + t_{comm}/t_c} \tag{4}$$

We see here that the speedup is very large in this theoretical case, especially as the data grows large. (Obviously this is practically limited by number of accessible processors. While it is not hard to imagine an array with hundreds of thousands of elements, having this many processors is often not feasible) The efficiency is

$$E = \frac{T_{sp}}{P} = \frac{1}{1 + (Plog_2(P)t_s)/(t_cNlog_2(N)) + (log_2(P)t_{comm})/(t_clog_2(N))} \tag{5}$$

Applying the same limit of $P \to N$ yields

$$E = \frac{T_{sp}}{P} \to \frac{1}{1 + t_s/t_c + t_{comm}/t_c} << 1 \tag{6}$$

Which is very small as startup time is far larger than computation time. This shows that using a large number of processors means that communication time takes up most of the time spent.

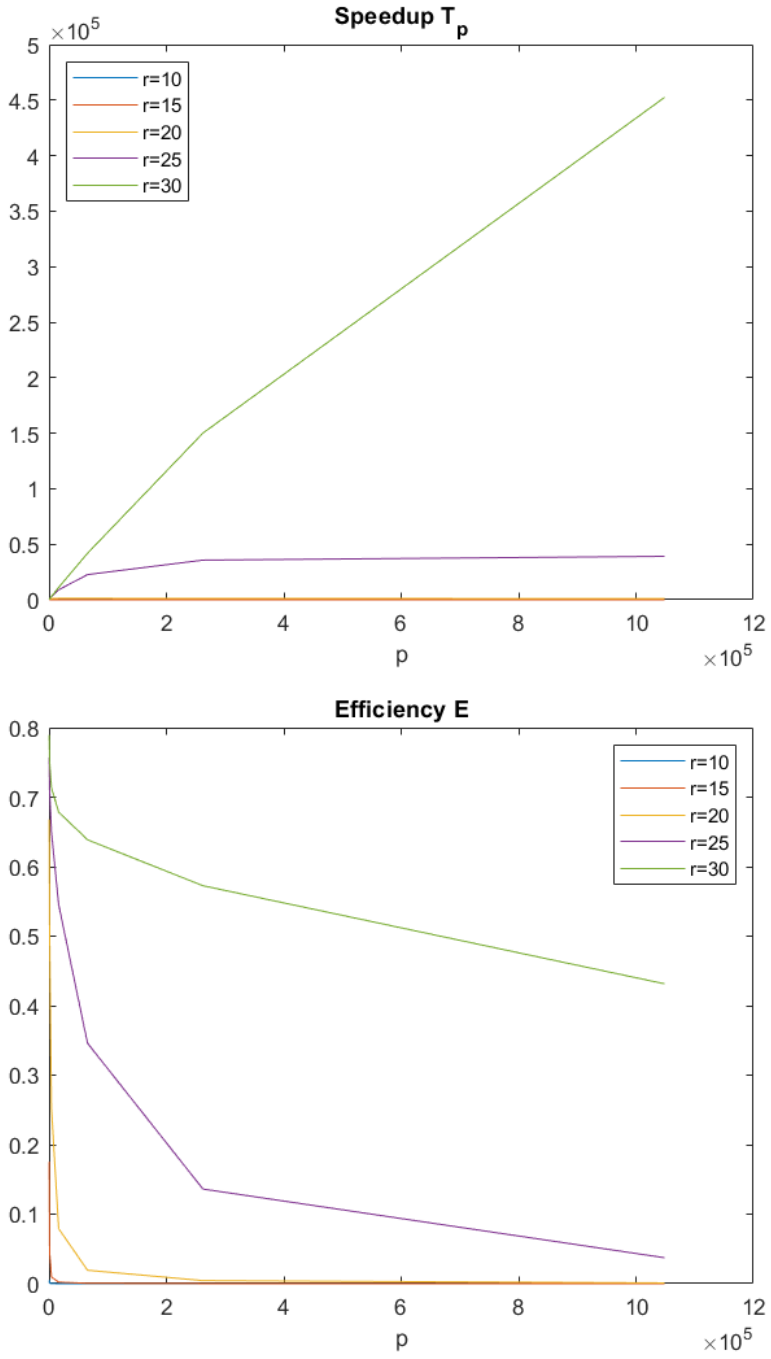

Figure 3: Speedup and efficiency of parallel FFT, $r = \log_2(N)$.

From figure (3) we see that speedup is larger with large N and large P, and that efficiency is large for large N, but grows smaller as P grows. For these results, we have assumed $t_s = 1000t_c = 1000t_{comm}$

3.1 Data

The data used in this report is a sound clip found on freesound.com by user ERH. [3] The clip was chosen due to the size (0.5 minutes) and the variation of frequencies that occur in the clip. The file is sampled at 44.1kHz and used under a Creative Commons license. Both a WAV-version and a MP3-version (of the same file) were collected in order to make a comparison of the two formats. It is worth noting that the WAV-file (5.1MB) is several times larger than the MP3-file (469kB) which might have an impact on the results of the algorithm when run on the two files. The sound data is generated using matlab's audioread function and stored in two text-files one for the WAV-data and one for the MP3-data. To make file reading easy and accurate the data is stored in scientific format with **exactly** 21 characters. This means the negative values were rounded to 14 significant figures and the positive values had 15 significant figures.

The file is converted into doubles using C and the array is then padded with zeros on the end to elongate the array so that its padded length is a power of two. Doing this improves to frequency resolution, by adding more bins in what is essentially interpolation in the frequency domain. Also note that the algorithm is unordered which means it is necessary to bit-reverse the output of the algorithm described in figure 1. This is done by matlab's Bitrevorder-function.

4 Analysis

Since we use zero padding in our algorithm, we must reevaluate the efficiency of the algorithm. Let N be the size of data input and N_{pad} be the size of the padded array. Note that $log_2(N) \leq log_2(N_{pad}) \leq log_2(N) + 1$ as N is not necessarily a power of two. The speedup is then

$$T_{sp} = \frac{t_c N log_2(N)}{t_c log_2(N_{pad})(N_{pad}/P) + log_2(P)(t_s + t_{comm} N_{pad}/P} \tag{7}$$

Since the padding only extends to the next power of two, $1 < N_{pad}/N < 2$ so

$$T_sp > \frac{t_c P log_2(N)}{(log_2(N) + 1)(2t_c) + P log_2(P)(t_s/N) + 2t_{comm} log_2(P)} \tag{8}$$

$$T_{sp} < \frac{t_c P log_2(N)}{log_2(N)(t_c) + P log_2(P)(t_s/N) + t_{comm} log_2(P)} \tag{9}$$

We see that the best case is bound from above by the theoretical speedup calculated in section 3. The worst case is not that much worse, since there is just some constants that makes the speedup slightly smaller. Obviously, the same behaviour follows for the efficiency of the implementation.

5 Results

Firstly, we present the results of the absolute value of the resulting FFT-vectors from our algorithm. We note that the two profiles are very similar, except from some small frequencies.

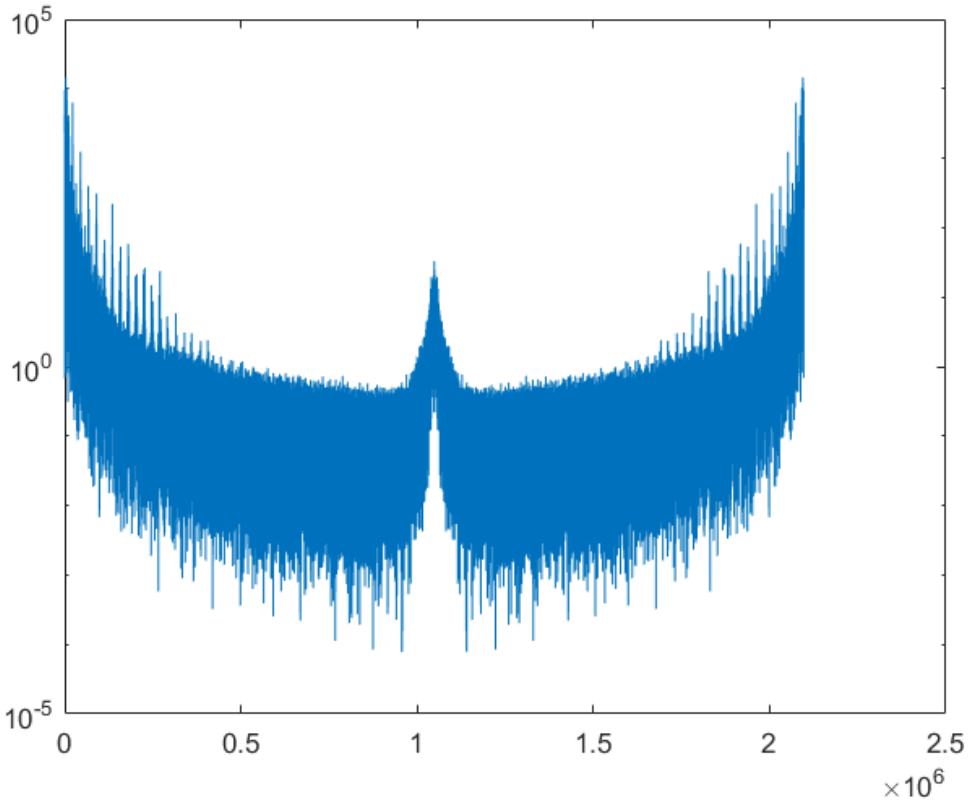

Figure 4: FFT of the WAV-file.

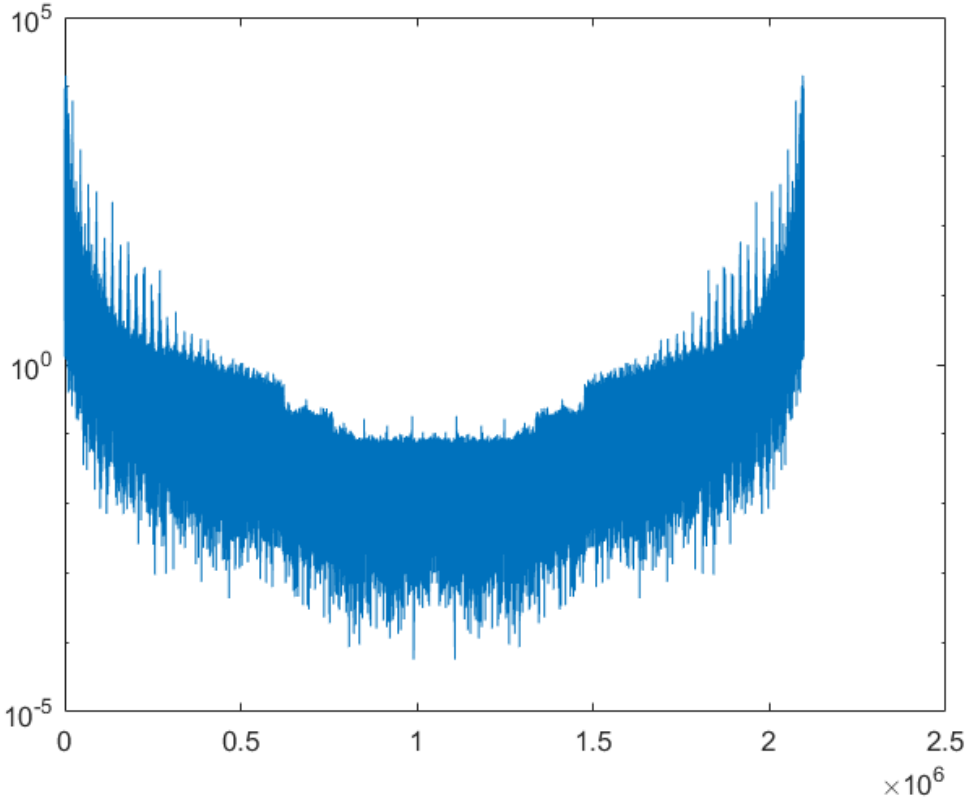

Figure 5: FFT of the MP3-file.

5.1 Empirical time analysis

The FFT-implementation's time performance is analysed by running the algorithm on $P = 1, 2, 4, 16, 32, 64, 128$ processors on three different data sets with $N = 1\,024, 524\,288, 4\,194\,304$ data points respectively. The data sets contain random data sampled from a standard normal distribution.

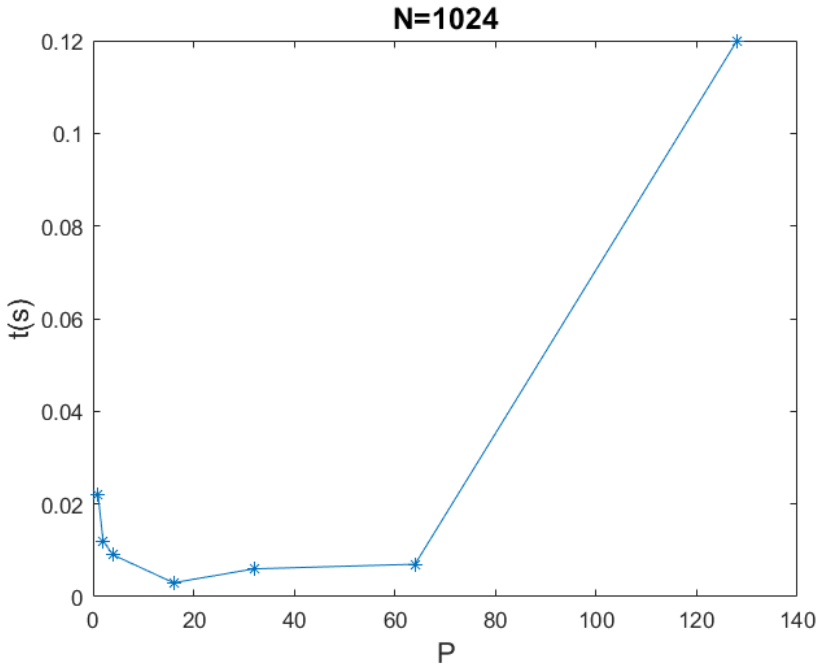

Figure 6: Time performance of parallel FFT

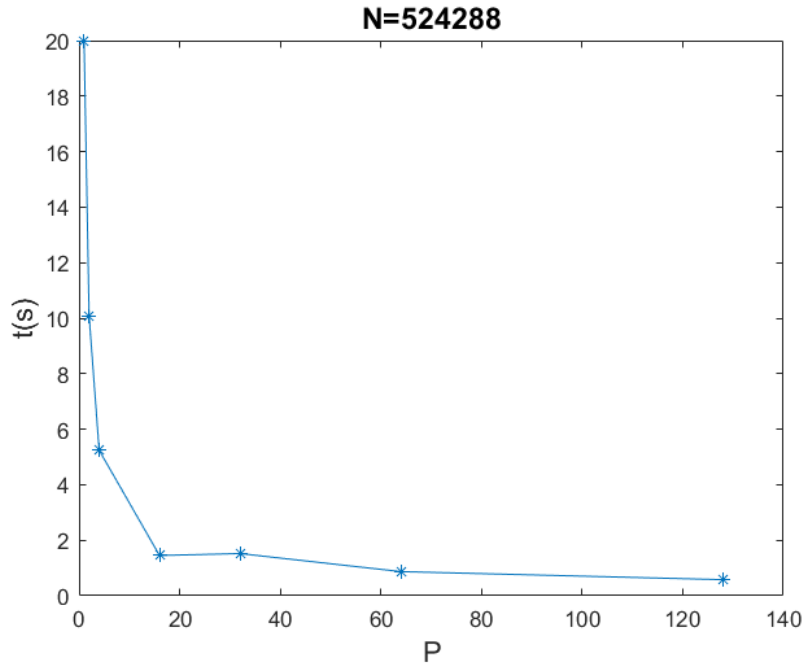

Figure 7: Time performance of parallel FFT

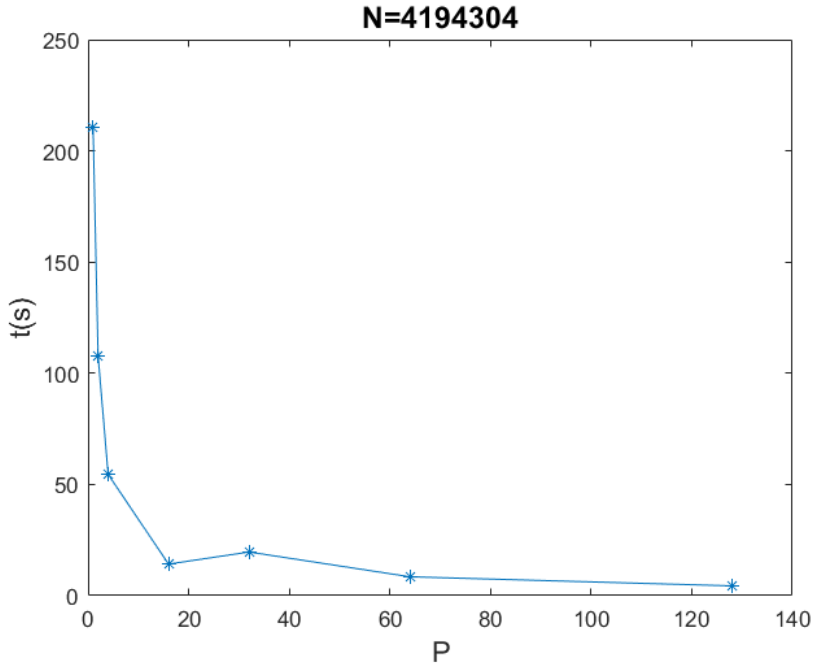

Figure 8: Time performance of parallel FFT

5.2 File size

Lastly we collect the WAV-file's size and the MP3-file's size in the following table.

Table 1: Collection of the files' sizes.

File-type	File size (bytes)	File size
WAV	5 120 112	5.1 MB
MP3	469 028	469.0 kB

6 Conclusion

Based on our FFT analysis, the frequency spectrum of the MP3 file is smaller. This is possibly a simple consequence of the fact that the MP3 is compressed while WAW-files are uncompressed, and therefore is smaller as well. It makes the MP3-file well suited for consumption, while the WAW-format is better adapted for professional settings where more information is critical.

We see that for large N, performance improves greatly with the addition of more processors.

References

[1] Steven G. Johnson Matteo Frigo. The design and implementation of fftw3. *Proceedings of the IEEE 93 (2)*, 2005.

[2] George Karypis Ananth Grama, Anshul Gupta and Vipin Kumar. *Introduction to Parallel Computing, Second Edition.* Addison-Wesley Professional, 2003.

[3] EHR. tension.wav.

7 Project outline

1. **Title:**
   Parallel radix-2 Cooley-Tukey fast Fourier transform

2. **Problem description:**
   We will parallelize the radix-2 vesrion of Cooley-Tukey FFT and pad the data with trivial functions if the number of data points are not a power of 2.

3. **Methods:**
   To solve the proposed problem we do the following.

   (a) Distribute the data on the different processors. Processor p's chuck of data is denoted $\texttt{arr}_p$ which has length N_p.

   (b) Pad each data chuck, $\texttt{arr}_p$, with zeros such that $N_p = 2^k$ for some constant k.

   (c) Perform the radix-2 Cooley-Tukey FFT on $\texttt{arr}_p$.

   (d) Print the result (with out the padded zeros) to a file.

   (e) Analyse the data in Matlab.

   (f) Analyse the program's efficiency.

4. **Software tools:**
   C, Open MPI and Matlab (and some of its libraries) for plotting and for verification of the result.

5. **Data set:**
   (Might change) We will compare a song from a CD saved as WAV-file with 44.1 kHz sampling rate and compare it to an mp3-compressed version of the same file.