Parallel and Distributed Systems: Paradigms and Models
Huffman coding

Pasquale Esposito
ID: 649153
p.esposito8@studenti.unipi.it

a.a. 2022/2023

1 Introduction

A Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The aim of this project is to implement a compressor, based on Huffman code, providing a serial and two parallel (native C++ threads and FastFlow) versions.

In this report, the main decisions for the implementation are discussed, and some execution tests are analysed in order to have a comparison of the performances among the different implementations.

2 Problem evaluation

The Huffman compression can be split into six main steps:

1. Reading the file from the disk
2. Counting characters' frequencies
3. Building the Huffman Tree and the code table
4. Huffman encoding of the file
5. ASCII encoding of the file
6. Writing the file to the disk

The steps worth parallelizing are 2, 4, 5; in the following, there is an explanation about how they work.

2.1 Counting characters' frequencies

This phase consists in taking a string as input and counting the frequencies of each character inside it; the expected result is a list of *key* - *value*, where the *key* is the character and the *value* is the number of occurrences of it in the input. From a theoretical point of view, the solution for this problem can be implemented using a **map** pattern and a block distribution policy, i.e. splitting the input string into chunks of size len/nw (where len is the length of the file text and nw is the number of workers to use for the computation). In case there are some remaining characters, the last worker manages them. Furthermore, a **barrier** is needed in order to have a global frequency table of the characters; in fact, step 3 needs the global table to create the Huffman tree and the codes change based on the frequencies of the symbols.
In this phase, the expected **overhead** is related to the creating of threads, the wait for their termination and the mutual access to the global frequency table in order to merge the partial result, computed by the workers, in the main table.

2.2 Huffman encoding of the file

Similarly to the previous step, the encoding step gets the content of the provided file as input, but the output of this phase is a binary string, created by using the code table previously generated by the former step. Specifically, there is a scan of the input content character by character and in another string is concatenated the code of that specific character, so that a string of all 0s and 1s is created. Also in this case, the **map** pattern is the one that fits for the parallelization of this step and the splitting phase is exactly

the same explained in the previous paragraph; moreover, also a **barrier** is needed again, since the next step needs the total binary string length in order to emit chunks with a length multiple of 8 to the workers.
In this phase, the expected **overhead** is related to the creation and wait for termination of threads. Also, there might be overhead caused by the unbalancing of uncommon and common symbols in the chunks given to the workers. In fact, since the Huffman code gives variable code lengths, some workers may get chunks with a higher number of rare symbols and the concatenation of their respective codes could take more time than the concatenation of shorter codes.

2.3 ASCII encoding of the file

Finally, the last analysed step consists of transforming the binary string into an ASCII string. This step takes as input the binary string provided from the previous step and gives as output an ASCII string. Also in this case, there is a scan of the binary string, taking substrings of 8 bits each and concatenating in another string the corresponding ASCII symbol. In case the binary string is not a multiple of 8, 0s are added as padding so that the last character can be mapped as an ASCII character as well. Again, a **map** pattern is used to parallelize the computation, creating chunks with a length multiple of 8 (the last worker might get more characters in case the binary string length is not multiple of 8). A **barrier** is needed in order to sort and concatenate all the intermediate results and write the resulting string to the disk.
The expected sources of **overhead**, in this case, are related to the creation and the wait for termination of threads.

In all the three steps just introduced a scan is present, and in fact the computation is highly dependent on the input file size. So the bigger the input file is, the slower the computation is, considering that three scans are necessary to compress the source.
To compute the ideal completion time and speed up, shown in the section 4.1, the following formulas have been used:

$$T_{c_{ideal}}(nw) = T_{seq} + \frac{T_{par}}{nw} + T_{ovh} * nw$$

where T_{seq} is the time spent to compute the non parallelizing part of the code, T_{par} is the time spent to compute the parallelizing part (in sequential), T_{ovh} is the overhead time, i.e. the time spent in creating and synchronizing

the threads, nw is the number of workers;
and

$$sp_{ideal}(nw) = \frac{T_{c_{seq}}}{T_{c_{ideal(nw)}}}$$

where $T_{c_{seq}}$ is the completion time of the sequential code.
The T_{ovh} time has been calculated only considering, as sources of overhead, the creation and termination of a thread, which has been computed after some experiments were specifically run; on average, this time is around 70 microseconds per worker. However, other sources of overhead exist in the proposed solution, such as the presence of barriers and the static policy adopted; therefore the measured speed up might be lower than ideal.

3 Implementation

The main components of this project are the following:

- The **sequential** implementation of the algorithm, present in the `huffman_seq.cpp` file;
- The parallel implementation using **native C++ threads**, present in the `huffman_thread.cpp` file;
- The parallel implementation using **FastFlow** framework, present in the `huffman_ff.cpp` file;
- The implementation of some **utility** functions, in order to compute the Huffman tree and codes, present in the `huffman_alg.hpp` file.

In the following section, the native thread and FastFlow versions of the parallelized code are presented.

3.1 Thread implementation

The parallelized part of the code, as previously mentioned, are steps 2, 4 and 5.

3.1.1 Counting characters' frequencies

Step 2 is implemented following a **map pattern**, so the input string is split into approximately equal chunks (the last worker may have fewer characters to count than the other ones) and each worker counts the characters'

frequencies that are in its assigned chunk, storing the values in a local frequency table. Once this phase ends, using a `mutex`, in order to preserve the correctness of the result, each worker updates a global frequency table, adding its local results to it.

3.1.2 Huffman encoding of the file

For the Huffman encoding step, the procedure is pretty similar to the previous one, so the input string is, again, split in chunks and processed following a **map pattern**. Each worker receives a chunk and the code table; the map function they compute consists of taking, consecutively, each character from the input chunk and concatenating in a new string the code of that specific symbol. Once the map function is over, the worker writes in the specific vector slot it receives as input the binary string it computed. In this case, there is no need to use any kind of procedure to preserve the correctness of the result, since the slot where the thread writes is not shared with the other threads.

3.1.3 ASCII encoding of the file

Finally, also in this last parallelizing step, the pattern followed is the **map** one. Once again, after the binary string has been computed (concatenating all the results given as output from each thread), the binary string is split in chunks, taking care that each chunk is a multiple of 8. In case the string length is not a multiple of 8, 0s are added at the end of the string as padding in order to let the last character be mapped to an ASCII symbol as well.
Each worker receives a chunk as input and, starting from the beginning of the string, it gets substrings of 8 bits and maps it to an ASCII symbol; each ASCII symbol is concatenated in a new string and, once the mapping phase is over, the worker assigns the resulting string to a vector slot passed as input. Also in this case, there is no need for a mechanism to avoid the interference of writing among all the workers.

3.2 FastFlow implementation

The FastFlow implementation has been performed using three `ParallelFor`, one for each parallelizing step. The parameter passed to the `ParallelFor` constructor is only the number of workers `nw`.

4 Experiments

The experiments have been executed using text files of 1 MB, 10 MB and 100 MB size, respectively. The expected behaviour is a lower speed up on a small size file, i.e. the 1 MB file, while, vice versa, a good speed up with a big size file, i.e. the 100 MB file; this behaviour is due to the overhead generated by the creation and synchronization of threads, so the gain is higher in big data computations. The following plots show the completion time, speed up and scalability of computing the compression of the aforementioned files; the execution times reported are the average execution times; each computation has been executed five times in order to reduce the variance in the results.

4.1 Results

In the following table, the completion time of the three implemented versions working with a 100 MB file are reported, and in Fig. 1, 2, 3, 4, 5, 6 plots are reported, showing the completion time, speed up and scalability.

nw	seq	threads	ff	seq (no IO)	threads (no IO)	ff (no IO)
1	6.6	7.5	7.27	6.4	7.37	7.08
2	-	4.7	4.36	-	4.52	4.17
4	-	3.4	2.62	-	3.22	2.44
8	-	2.56	1.74	-	2.38	1.56
16	-	2.1	1.47	-	1.92	1.3
32	-	1.69	1.35	-	1.53	1.16
64	-	1.69	1.54	-	1.5	1.34

Table 1: Completion time (in seconds) of the Huffman coding performed on the 100 MB file.

4.2 Results analysis

As predicted, the speed up reached with the 100 MB file is higher than the one reached with the 1 MB file. In fact, as you can notice in Fig. 6, the highest speed up gained in the 100 MB file tests is 4.3 (with the native threads implementation) and 6.17 (with the FastFlow implementation), the first one obtained using 50 threads and the second one with 60, versus 3.16 (native threads implementation) and 3.54 (FastFlow implementation), the

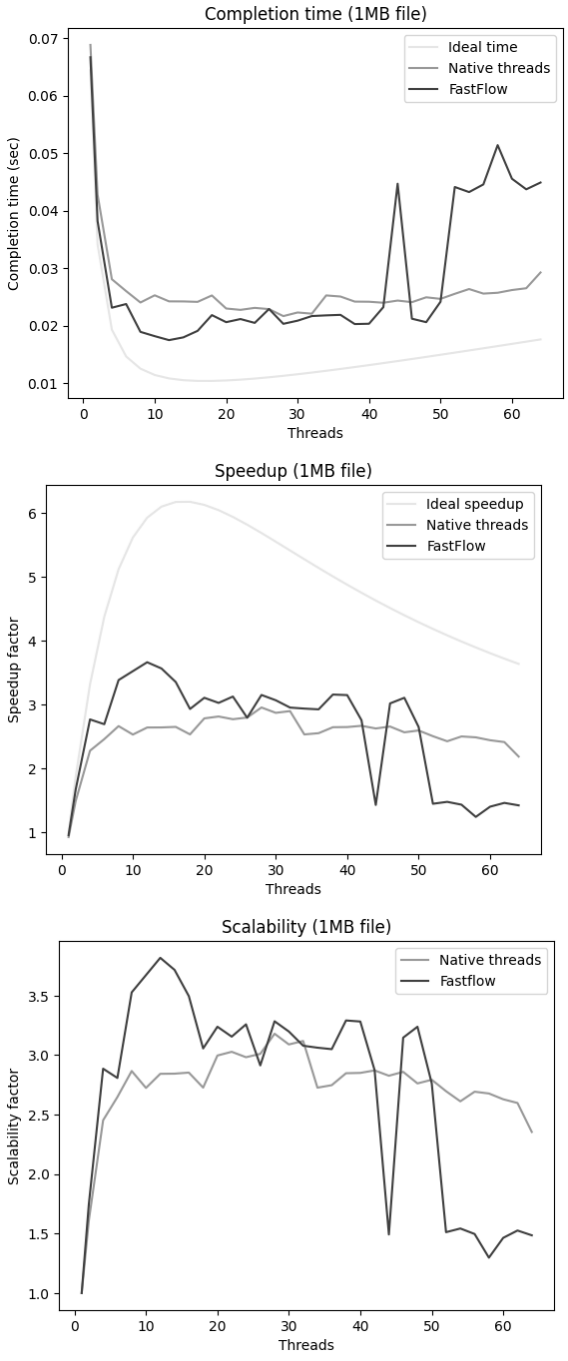

Figure 1: Results of 1 MB file.

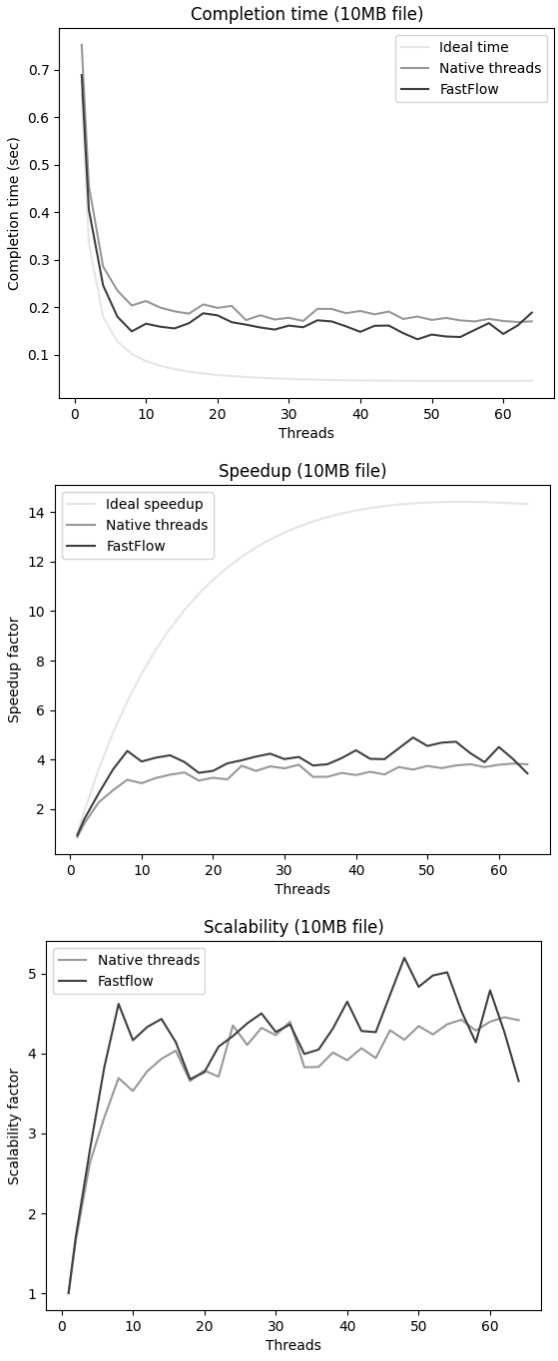

Figure 2: Results of 10 MB file.

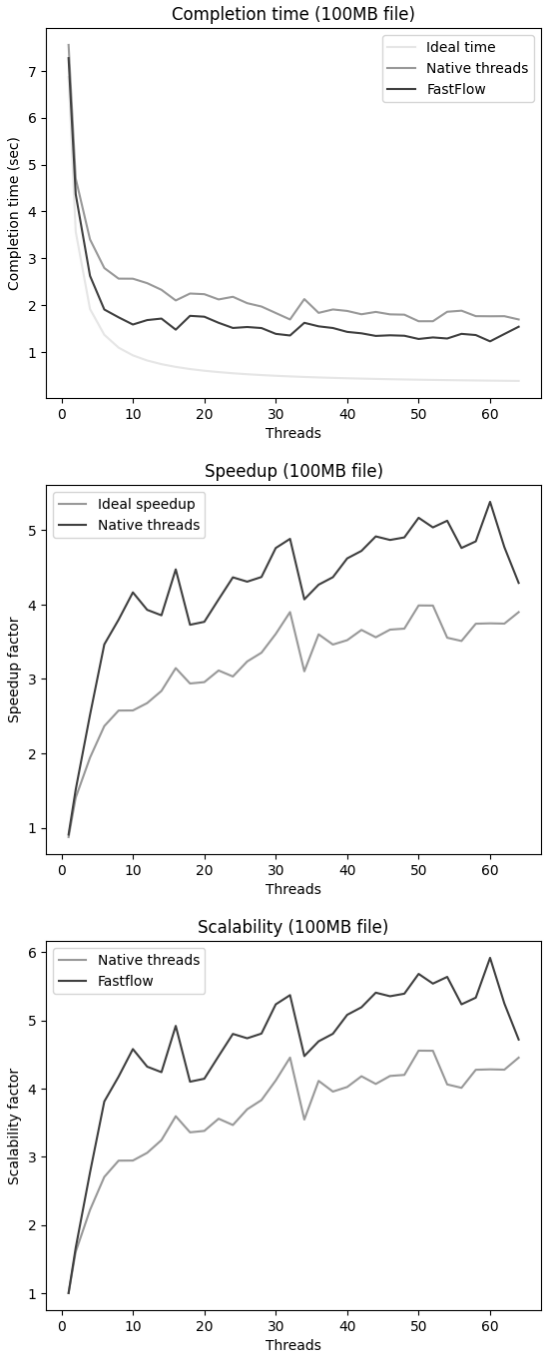

Figure 3: Results of 100 MB file.

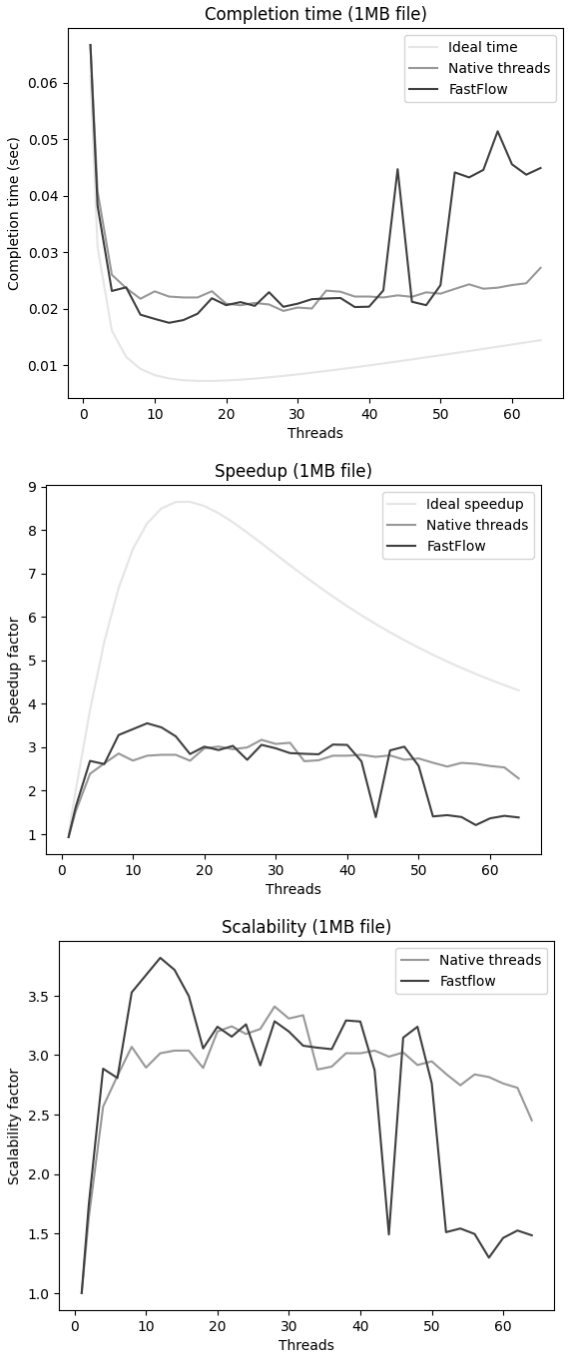

Figure 4: Results of 1 MB file (no I/O time).

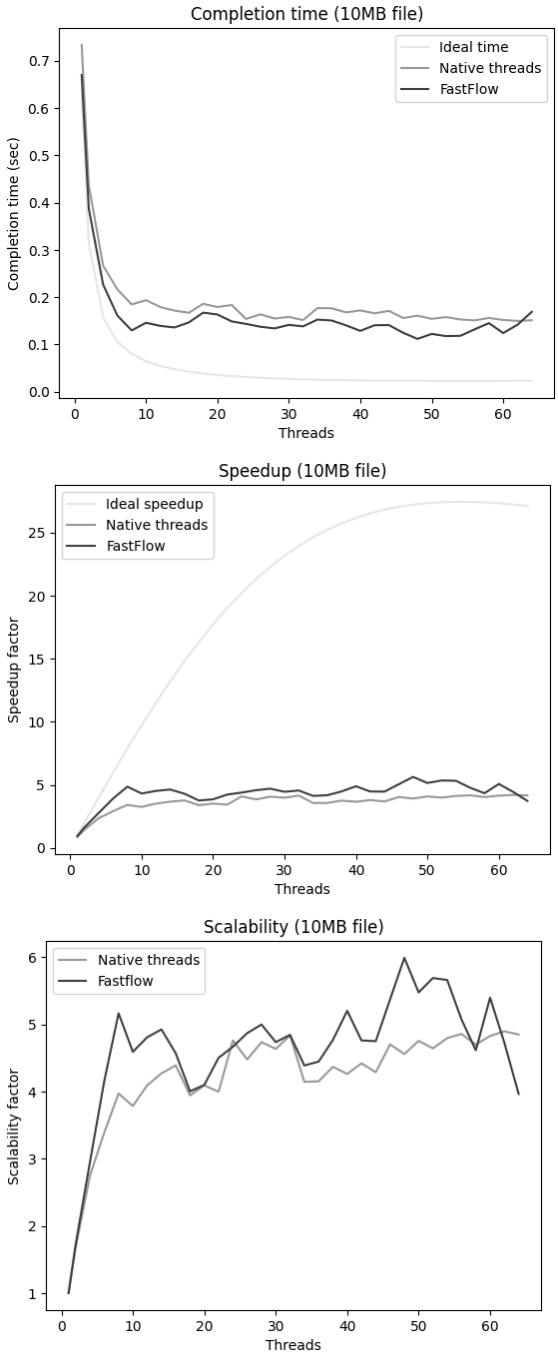

Figure 5: Results of 10 MB file (no I/O time).

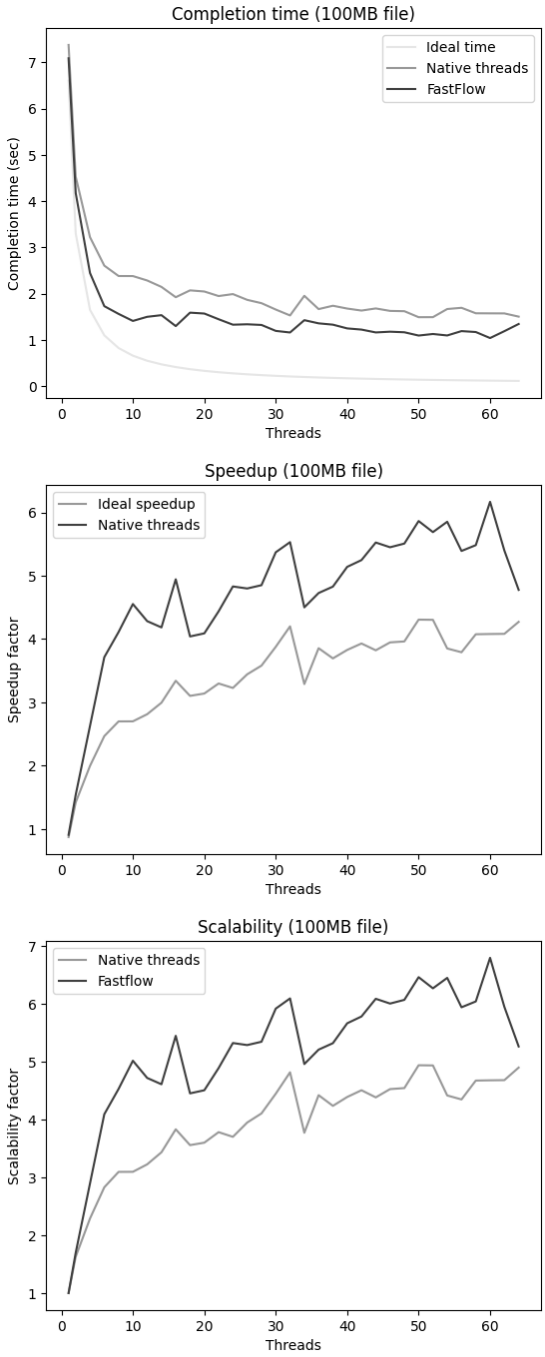

Figure 6: Results of 100 MB file (no I/O time).

first one obtained with 28 threads and the second one with 12, gained in the 1 MB file tests.
However, the expected results were much different from the obtained ones. The low speed up is probably due to the overhead that the static policy, used in both parallelizing implementations, introduced. In fact, as you can notice in Fig. 7, 8 and 9, the waiting time in the three parallelizing steps is quite high so that it introduces overhead for the threads' synchronization. In fact, considering for example step 3 of the algorithm, since the Huffman encoding has variable length codes, it may be the case that a thread gets a chunk with more rare symbols and since their code is longer than the most common symbols, the considered thread takes more time in mapping the symbols to the Huffman code. A possible solution to this overhead would probably be a cyclic strategy, splitting the strings in smaller chunks.

5 How to run the project

In the submitted folder, there are the source files of the sequential, native thread and FastFlow implementations and **input** and **output** folders. In the **input** folder, there are the three files used to run the experiments, while the **output** folder is empty and the compressed files will be stored there, appending, before the name of the input file, either `compressed_seq_` or `compressed_thread_` or `compressed_ff_`.
Also, in the submitted folder a **Makefile** is present, which needs to be run with the command `make`. To execute an executable file with a specific file, you first need to add the input file to the **input** folder and then pass only the name of the file to the program:

```
./huffman_threads <file_name.txt> <nw>

./huffman_ff <file_name.txt> <nw>
```

Finally, in the submitted folder four scripts used to run the experiments are also present.

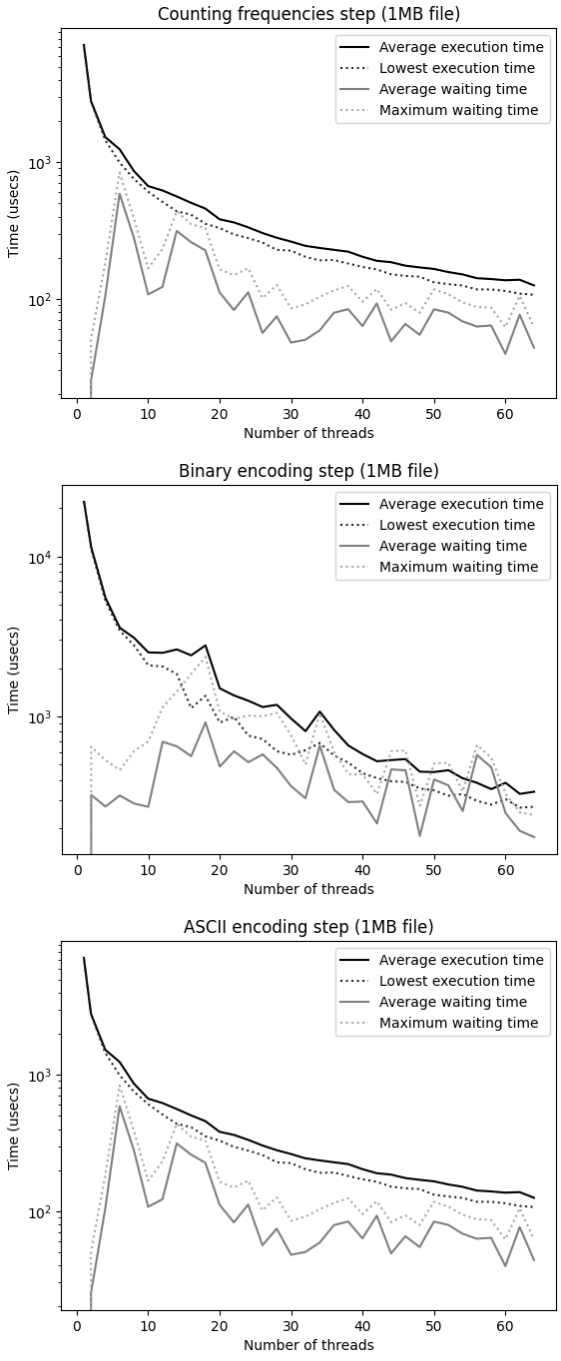

Figure 7: Execution time vs waiting time in 1 MB file tests.

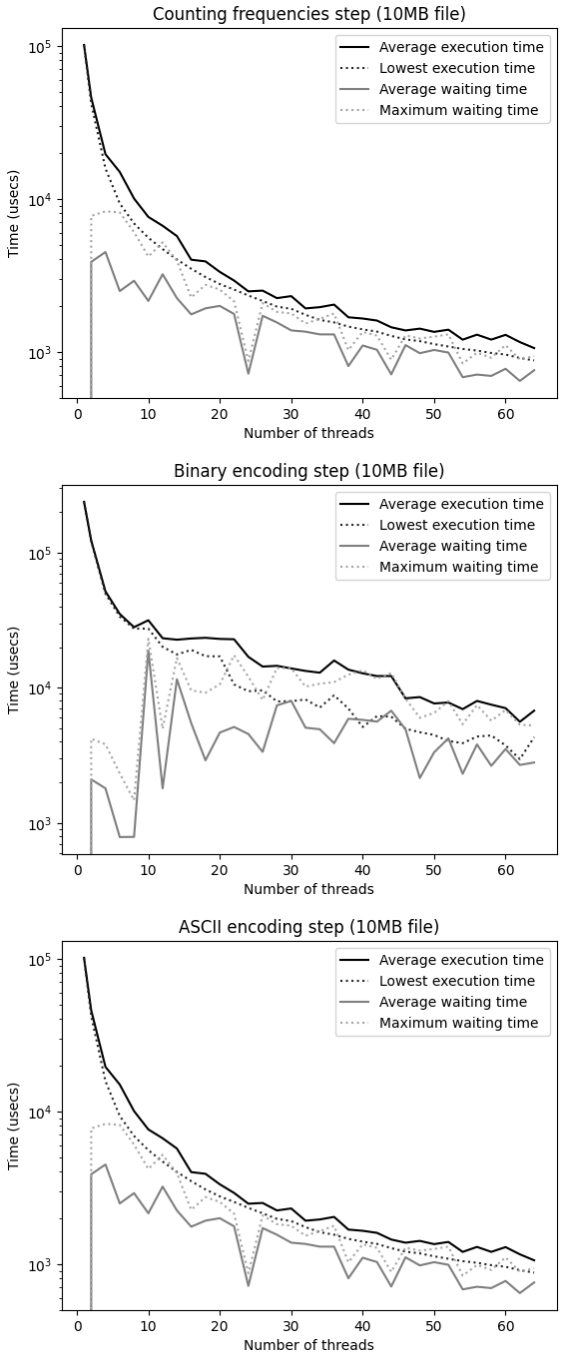

Figure 8: Execution time vs waiting time in 10 MB file tests.

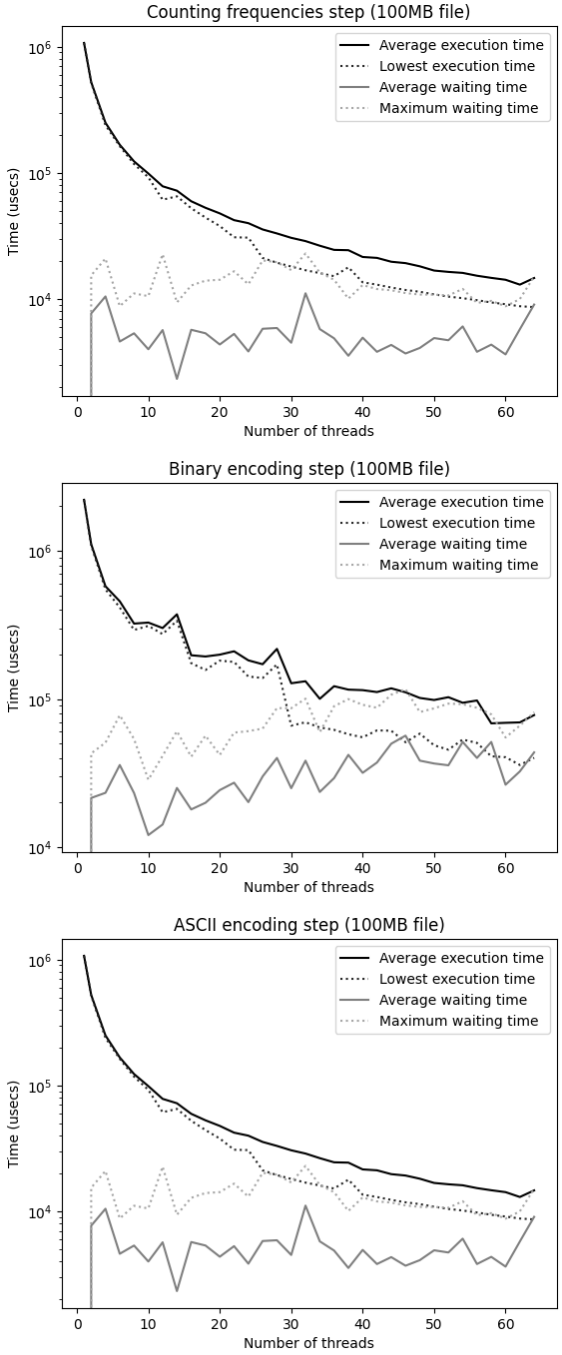

Figure 9: Execution time vs waiting time in 100 MB file tests.