Master Degree in Computer Science

Parallel and Distributed systems project: Huffman Coding

Francesco Caprari:580154

Academic Year: 2022-2023

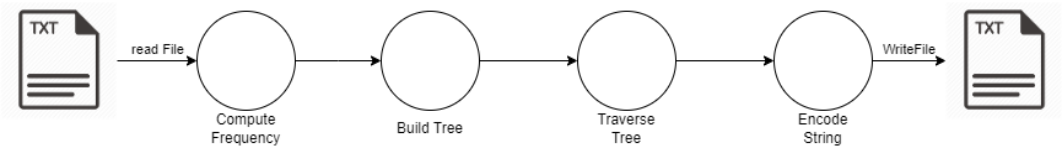
Contents

1 The Problem

The purpose of the project is to parallelize **Huffman encoding** by providing an implementation using native C++ threads and another using the patterns provided by the FastFlow library.

2 Sequential Implementation

The first phase involves scanning the text to record the frequency of each character. The second phase is where the Huffman tree is constructed and traversed, while in the third phase, the string is encoded. Finally, there are the last two phases where ASCII encoding is applied and the string is written to the output file.



By conducting several tests on a **5-megabyte** file using the **sequential implementation** of the algorithm, it was noticed that the construction and traversal of the Huffman tree come at a significantly lower cost compared to computing character frequencies and performing the actual string encoding. In this context, also the operation of writing the compressed file incurs a significant cost, this is because the procedure to create the compressed file involves a procedure that traverses the binary string character by character to create a byte and write it to the output file.
Result of the **Sequential algorithm** of a file of **5MB** express in microseconds.

- The average costo to read the file: **10166** usecs.
- The average cost to compute the frequencies: **48124** usecs.
- The average cost to build the Huffman tree and traverse it: **15** usecs.
- The average cost to encode the string: **108722** usecs.
- The average cost to write the string and compute the ASCII encode: **485452** usecs.

.

3 Exposing Parallelism

Given the previous result, the computation of character frequencies and the string encoding is parallelized in both the native **C++ thread** implementation and in **FastFlow**. The frequency calculation phase can be parallelized using the **map-reduce pattern**. The sting is divided into equal parts based on the number of threads, each thread will generate a list of pairs <**k, freq(k)**> for its portion of the text, and then the various lists is merge into a single list. Thanks to the previously generated list, the Huffman tree is constructed sequentially, and then, still sequentially, the tree is traversed and the dictionary is created that associates each character with its binary encoding string. Another possibility would have been for the threads to directly save the results into a shared vector using a lock on the structure.
To encode the text, similar to what happens during the frequency calculation, the text is divided into a number of **chunks** of the same size, equal to the number of threads. Each thread iterates through its portion of the text character by character and, using the dictionary, obtains the encoding, which is then concatenated with the encodings of the previous characters. Later, there is a phase where we reconstruct the encoded string by assembling all the substrings generated by each thread, ensuring that the order is maintained. Naturally, reassemble the various substrings into a single string introduces a significant **overhead** for calculating the result.
The writing to the file can also be parallelized, taking into consideration the transformation of the character string into the byte string, also in this case you can apply the same pattern used for encoding. It could also parallelize the **construction** of Huffman Tree using multiple threads working on different parts of the tree. In my implementation, i considered the case of **data parallelism** where i store the entire text in a string and then apply various computations. Another possibility is to process each portion of the text as i read it, using parallelism in a **streaming context**, in this scenario, instead of using a *map*, i would have used a *farm*.

4 Native thread implementation

For the implementation using native threads, the **fork-join model** is used and mutexes or condition variables are avoided. For calculating the character frequencies, a set of threads is launched into execution, calculating the same function. Each thread works on a different portion of the string and the **chunks** have the **same size** given by the formula: length of the string divided by the number of threads. The result is saved in a *vector<unordered_map<char,int»* as large as the number of thread, each thread save the frequency of characters in the substring, using chracters as keys. At this point, there is an overhead caused by reconstructing a single *unordered_map<char,int>* in a *sequential way* from the various *unordered_map* returned by each thread. To avoid reconstructing the final map sequentially, it can be used a shared vector and access it with mutual exclusion every time a position is modified, but of course, this would naturally lead to overhead due to the locks whenever access to the data structure is performed. Another possible alternative to using *unordered_map* is to use an *vector* of 128 positions. The construction and traversal of the tree is done sequentially, the encoding of each character is save in an *unordered_map*, which uses the character as

the key and associates it with the corresponding bit string. On the other hand, for the **construction** of the encoded sequence, i once again use the fork-join, the string is divided into chunks, just like in the frequency calculation phase model. I use a *vector<strings>*, as large as the number of threads, where each thread saves the encoded substring it has generated, this is followed by a reduction phase to obtain the encoded string from the various substrings.
The phase preceding the **file writing** can also be parallelized, where starting from the previously created string, a byte string is generated. Similar to how encoding is done, each thread works on a substring, processing in groups of eight characters, which are transformed into a byte, to achieve this, padding is applied to the string to ensure its length is a multiple of eight.

5 Fast Flow implementation

The parallel pattern of FastFlow is a *map*, i parallelize the frequency calculation part, the string encoding part and the ASCII encoding, just like i do with native threads. For the frequency calculation, i use a **parallel_for_reduce** each thread accumulates the result in a local *unordered_map* and then writes the result in the reduce phase to the global map. To perform string encoding, *parallel_for_idx* is used on the original string, where each thread encodes a substring and saves the result in a position of the vector. This is followed by a sequential phase in which the vector is traversed, and the complete string is reconstructed, of course, this step represents an **overhead** in the computation of encoding. For the ASCII transformation, a padding is applied to the string first, ensuring its length is a multiple of eight. Then, using a *parallel_for_idx* with a chunk size that is a multiple of eight, the results are saved in a vector of strings. Subsequently, these strings are combined to construct a single string, which is written to the binary file, so similarly to the previous case, there is an overhead for reconstructing the result. With Fast Flow, one alternative approach was to utilize a combination of *ff_node*, incorporating an emitter and a collector using the *ff_Farm*, both for the frequency calculation phase and the encoding phase.

6 Experiment Result

The algorithms were tested on a **1-megabyte**, **5-megabyte** and a **10-megabyte** file, with the number of tested workers being powers of 2 up to 64, for both native threads and fast flow. Let's compare their performances in terms of speedup and scalability.

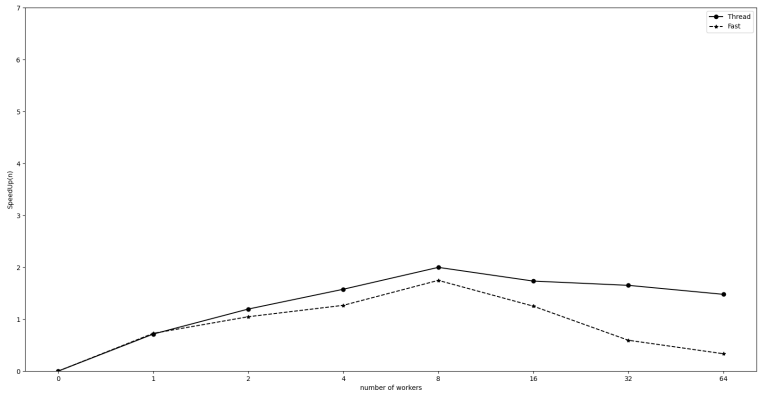


Figure 1: Speedup Chart, 1 Megabyte file, no I/O

In the **1-megabyte file**, it is observed that especially in the fast flow implementation (Fig.1), there is not a significant improvement, this is due to the small size of the file, resulting in too much overhead that prevents significant results.
For the **10-megabyte file**, let's compare the different outcomes among various implementations, considering **Scalability (Fig.3)**, **Speedup (Fig.4)**, and **execution time** in correlation with the number of **workers** used **(Fig.2)**. Let's also take a look at the differences in these values, considering input and output operations or not.

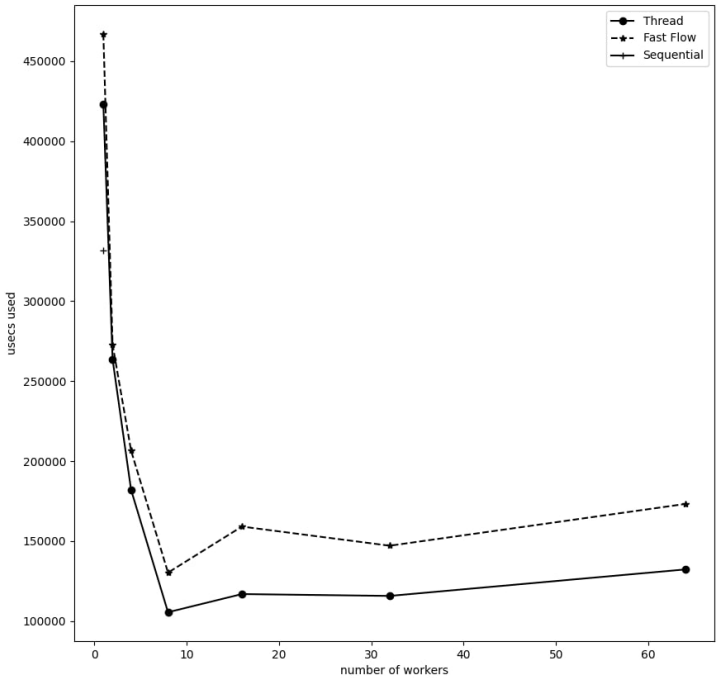


Figure 2: Usecs Chart, 10-megabyte file

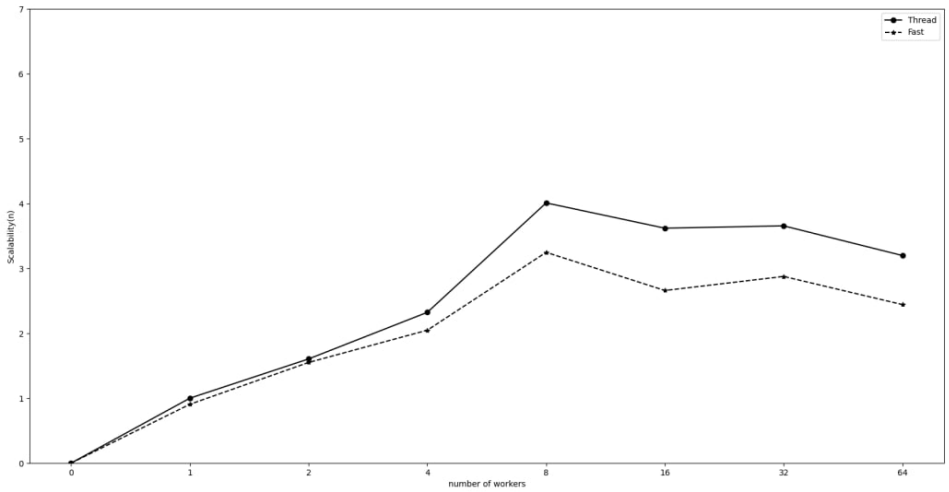


Figure 3: Scalability Chart, 10-megabyte file

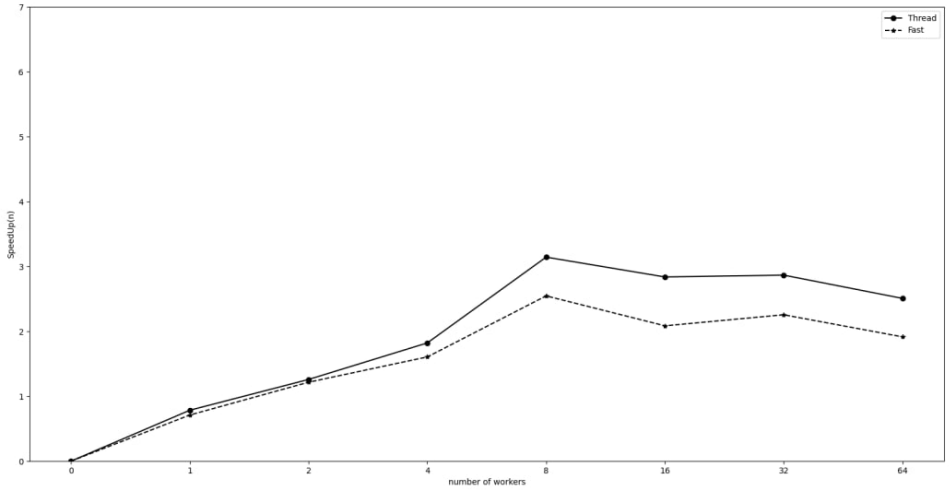


Figure 4: SpeedUp Chart, 10-megabyte file

As evident from the figures, on the **10-megabyte** file, i achieve favorable results with both fast flow and native threads, however, in the case of fast flow, Speedup **(Fig.4)** and Scalability **(Fig.3)** decrease more rapidly. With threads, we achieve a level of scalability that surpasses even 4.0, while for the Speedup, we exceed 3.0. Naturally, as i test on various files, it becomes evident that i experience better performance as the file size increases.

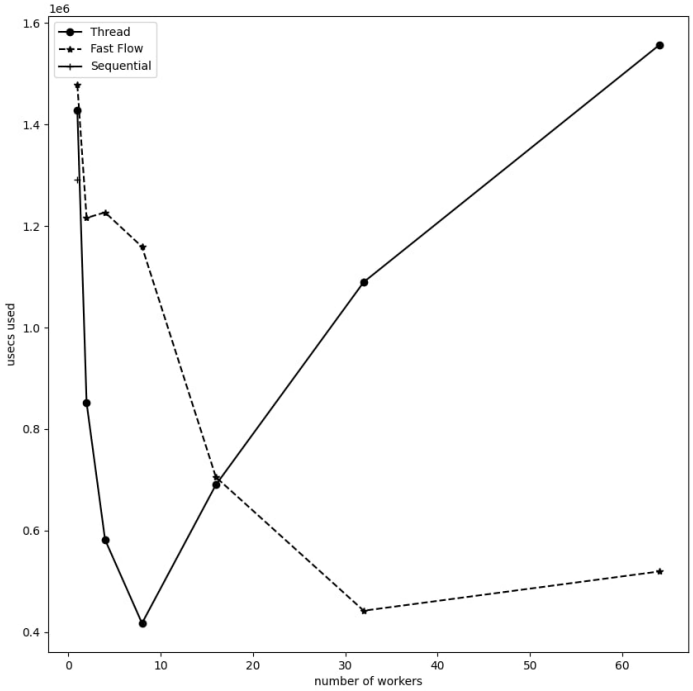


Figure 5: Usecs Chart, 10-megabyte file, I/O

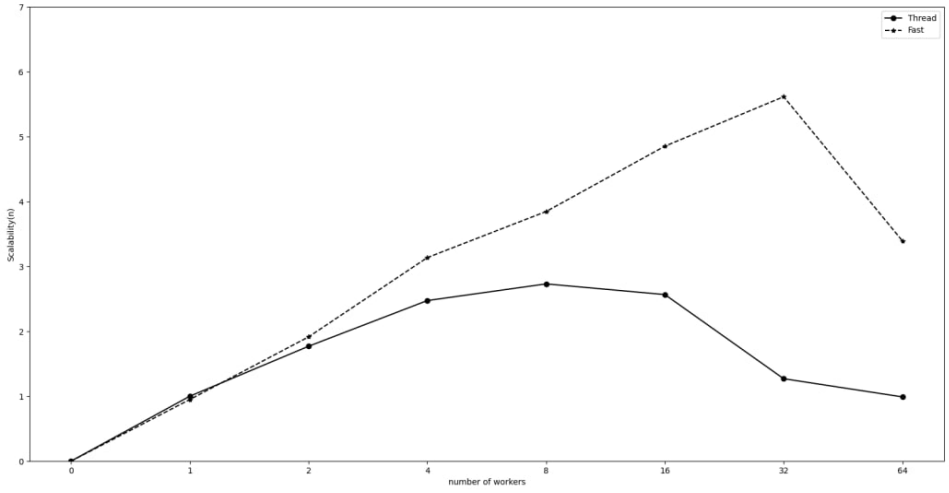


Figure 6: Scalability Chart, 10-megabyte file, I/O

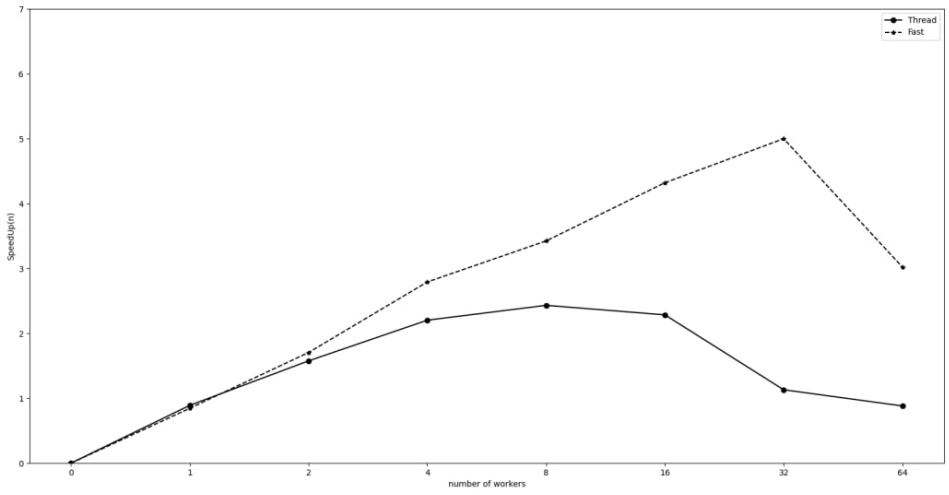


Figure 7: SpeedUp Chart, 10-megabyte file, I/O

From the **Figure:5**, i observe that when considering input and output operations, the performance deteriorates in the thread implementation. Specifically, the file writing phase involves translating the string into bytes, and in this function, i experience a significant overhead generated by the spawning and termination of the threads and from the concatenation of the strings, that worsens as the number of threads increases. In fact in the native threads implementation, using more than 30 threads leads to excessive overhead. As for the Speedup, **significant results** are achieved in the implementation that utilizes **Fast Flow**, values exceeding five are achieved.
Conducting some tests on the **5-megabyte file**, considering the implementation with native threads, the overhead to reconstruct the *unordered_ map* in the case of frequency calculation is almost negligible, in the encoding phase the overhead is more significant, however, the **highest overhead** is incurred in the function used to calculate the ASCII encoding, where thread joining incurs a prohibitively high cost. Of course, in both implementations, as the number of threads increases, there will be an increase in overhead. Various times expressed in microseconds from calculation on the **5-megabyte** file on average with 16 workers:

- Time for reading: **9625** usecs.
- Time for Compute the frequency: **4102** usecs, Overhead for frequency: **59** usecs.
- Time for Encoding: **47221** usecs, Overhead for Encoding: **31859** usecs.
- Time for build and traverse the tree: **19** usecs.
- Time for encode ASCII: **153161** usecs, Overhead for ASCII: **41024** usecs.
- Time for write: **2270** usecs.

The overhead times take into consideration the time to reconstruct the result.

7 Instruction for execution

There are three files one for each implementation and an header file that contains the function to construct and traverse the Huffman tree and the function to write the encoded sequence to a file. The three implementations can be executed by specifying the mode as the third parameter, whit '0' the computation time is printed without considering input and output operations. Conversely, with '1', you will see the time including input and output operations and "ASCII trasformation", if is not specify default is 0.
The other "Temp" files display the execution time on the screen, differentiating it for each phase, including the time spent for overhead. Taking as input the text to be translated and the number of workers for the FastFlow files and threads.
In the folder, there is also the file **utimer.hpp** used to measure the time necessary to calculate the results, and the **Test** folder where ASCII files are present to test the program
Each implementation writes the result to the compressed file: **"textOut.bin"**.
Steps to execute the program:

- make all.
- ./HuffmanSeq.out "test.txt".
- ./HuffmanThread.out "test.txt"" numberofthreads".
- ./HuffmanFast.out "test.txt" "numberofthreads".

.