UNIVERSITÀ DI PISA

Parallel and Distributed Systems: Paradigms and Models

Academic Year 2022/2023

Huffman Coding

Luca Santarella

Contents

1 Problem Evaluation **2**

2 Theoretical Solution **4**

2.1 Counting 4

2.2 Huffman Coding 5

2.3 ASCII encoding 5

2.4 Alternative solutions 6

3 Experiments and Results **7**

4 How to run the project **10**

Chapter 1

Problem Evaluation

The problem of the project is to implement the Huffman coding in parallel using native threads and FastFlow. The Huffman coding is a kind of encoding algorithm with variable-length codes that can be used to compress data such as text file, in our case the input is an ASCII file. The algorithm can be divided into these major steps:

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

One of the steps worth parallelizing are the **counting of the frequencies of the characters** inside the file, this step will take in input the string containing the contents of the input file and it will produce in output a vector of integers containing the frequency of every character. In this case we exploit the fact that we are receiving in input only ASCII files, so the data structure chosen is a vector of integers of dimension 128, in this way each position of the vector will contain the frequency for the character with the corresponding decimal

representation, e.g. in ASCII the character 'a' corresponds to the decimal value 97, so in position 97 we will have the frequency of the character 'a'. Another step worth parallelizing is the actual **Huffman coding** which will take in input the string of the contents of the input file, it scans the string character by character and using the code table produced by the previous step it will produce in output another string of zeros and ones which represents the original string encoded with the Huffman coding. The final step which is parallelized is the **ASCII encoding**, in this case the string of zeros and ones needs to be written on file so to do this we simply convert the binary string into a string of characters using the ASCII encoding and then write it in a text file.

In all of the three steps mentioned before, the weight of the computation is highly correlated with the dimension of the input file, thus, a bigger size of the file will make the computation of the encoding slower since the three steps will have to scan a longer string. In figure 1.1 are represented the execution times (taken using 'utimer.hpp') of the various steps for a 10 MB text file and a 1 MB text file. The other steps (except for the reading and writing) are independent of the input size, so they are constant in the execution time. One of the main challenges in the parallelization of the algorithm is the fact that the Huffman coding uses variable length codes, this means that the input and the output size will be different. Another aspect to keep in mind is the fact that the ASCII encoding requires that the input binary string must be a multiple of 8, so after the Huffman encoding a padding of zeros is set. The number of zeros which were used for the padding must be provided to whoever wishes to decompress the text file.

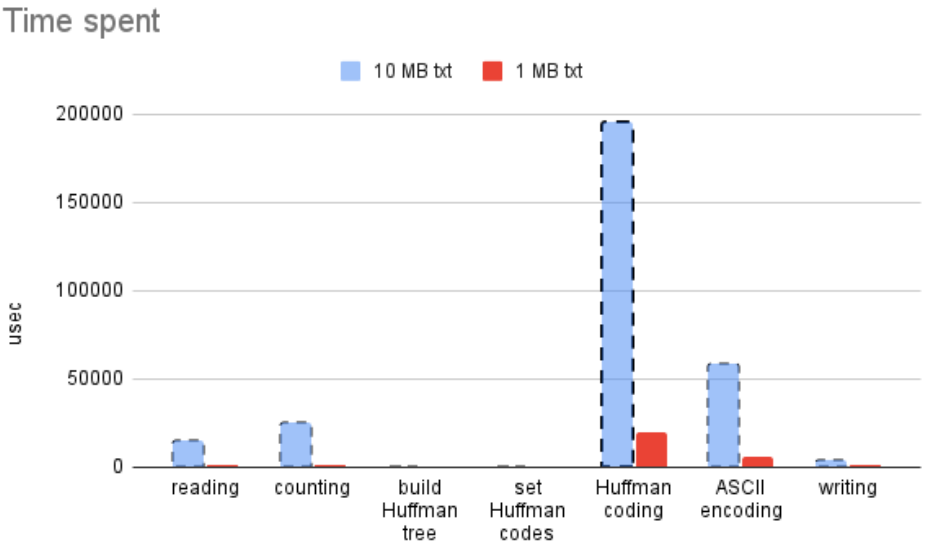


Figure 1.1: Time spent during the steps of the algorithm

Chapter 2

Theoretical Solution

After analyzing the problem by understanding the weights and providing a sequential version of the code, the idea was to use the data parallel pattern map to deal with these heavy computing steps of the algorithm.

2.1 Counting

The first idea for the **counting phase** was a map pattern to split the string among the workers and to keep the frequencies in a single shared vector but soon I realized that the shared vector would have been a huge bottleneck for the computation. The final solution for the counting phase was to equally split the string into chunks among the workers with a simple static load balancing, the reason behind this is that the workers will take more or less the same execution time for the chunks assigned.

Next, the chunks are processed by every worker which will count the frequencies by scanning character by character and store them into a local vector to keep track of the partial result. Once the substring (chunk) has been elaborated, the worker will add the frequencies of its partial result contained in the local vector to the shared vector in mutual exclusion using a lock. The main drawback of this approach is that the final merge of the partial results in mutual exclusion is a overhead which takes some time but it is way less of an overhead than the approach of a single shared vector.

2.2 Huffman Coding

For the Huffman coding the process is fairly similar, the input string is still equally divided among the workers which take their chunks and process sub-results, finally the sub-results are combined into the final result. The worker scans the input string character by character and by using the previously built code table (the shared code table is accessed in read-only) transforms the character into its Huffman code equivalent. The partial results of the workers are stored into a vector of strings of dimension nw, (where nw is the number of the workers), in this way once the worker is done with the encoding it writes the sub-result in position th_id, (where th_id is the logical id given to the thread), of the vector. In this way there is no need for synchronization since every worker writes in its own memory location, the final step, once all the workers are done, is to concatenate the strings of the vector into a single string which represent the final result. Also in this approach the main drawback is the final sequential concatenation of the substring, this step is necessary because the output string will have a different size w.r.t. the input string (Huffman coding uses variable-length codes) so we need to preserve the ordering of the encoding, otherwise there is no guarantee about the finishing order of the working threads.

2.3 ASCII encoding

For the final ASCII encoding which is needed to write the binary string on file, the approach is similar to the Huffman coding with the detail to make the chunk size a multiple of 8, since the ASCII encoding is done 8 bits at the time. The input binary string (which is the final result of the Huffman coding) is divided into chunks and distributed among the workers, each one of them transform 8 bits at the time into the corresponding ASCII character. Once the whole binary substring has been transformed into a string of ASCII character they are copied into a shared vector of string with dimension nw. As in the Huffman coding, each worker has its own position in the vector where to report the sub-result, once all workers are done the final result is obtained by concatenating the strings of the vector, also in this case this constitutes an overhead to the computation.

The solution adopted for the two parallel implementations is the same described above, the only difference is that the FastFlow implementation realizes the map using the ff_Farm, so in this case there are also an emitter and a collector.

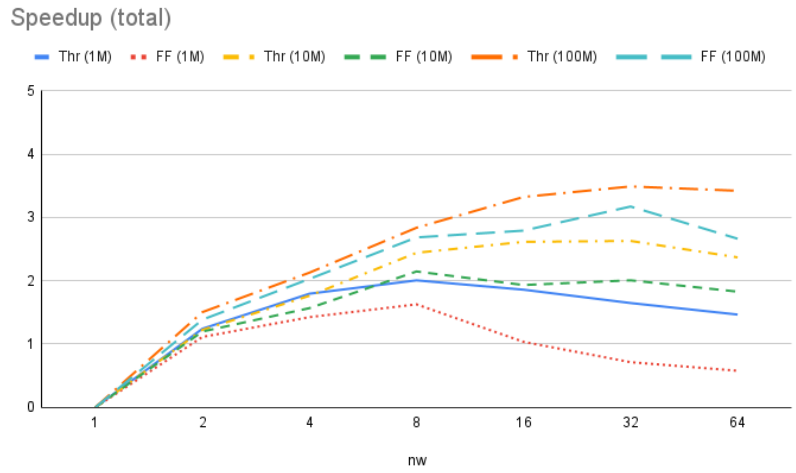
2.4 Alternative solutions

Another possible solution would have been to use a thread pool to prevent wasting time in the spawning and terminating threads for each one of the three steps, this path was not followed as I believed that the overhead incurred was not as significant as to implement a more structured solution. A possible solution to reduce the overhead of the merge would have been to implement a dedicated worker whose task was to receive the sub-results from the workers and concatenate them in parallel preserving the order, also in this case the overhead estimated from a sequential merge was not worth to implement such solution. As seen in the experiments made (see chapter 3), the algorithm is better suited for text files of great size. For this reason a possible way to further optimize the computation would be to have a more dynamic approach consisting in analyzing the size of the input string and based on a (possibly dynamic) threshold choosing if the algorithm should be run in parallel for files with a size greater than the threshold or sequentially otherwise.

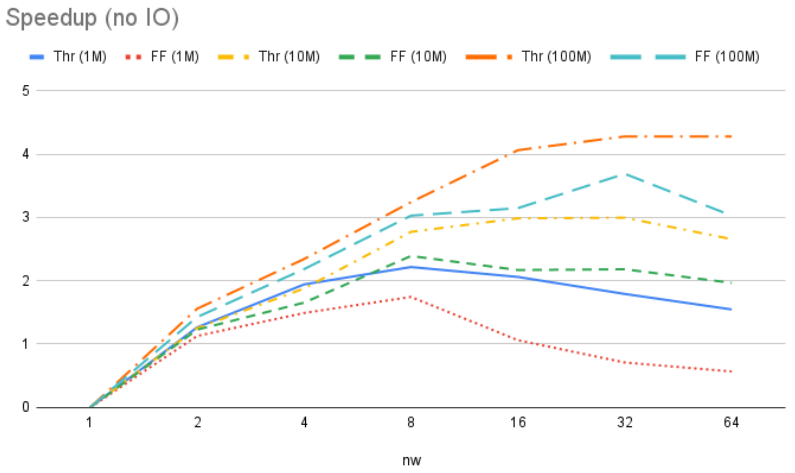
Chapter 3

Experiments and Results

For the Huffman encoding phase the expected performance was to achieve a good speedup for bigger text files whereas smaller files would lead to slower execution times w.r.t. the sequential version. This happens because the overhead generated by the spawning and termination of the threads as well as the overhead of the synchronization and the final merge of the sub-results cancels the benefit gained from the parallelization smaller and smaller substrings. In order to verify this, the experimental tests on the execution times were conducted on three different size of text files of dimension 1 MB, 10 MB and 100 MB. Note that the execution times reported are the average execution times after running the algorithm twenty times, in this way we can avoid inconsistent times and have less variance in our data, the number of workers tested are powers of 2 going from 1 to 64. As we can see on figure 3.1 and figure 3.2 the 100 MB has the best speedup and scalability, the maximum speedup reached it's 4.28 with 32 workers on the thread version and the maximum scalability achieved is 4.61 with 32 workers. It is possible to observe that in fact the performance on the 1 MB text file is worse in terms of speedup w.r.t. the 100 MB file, as expected this is due to the size of the string elaborated from the workers. Both implementations with native threads and in FastFlow have comparable execution times, speedup and scalability with FastFlow falling slightly lower, this is most likely due to the more structured nature of the map which is implemented as a Farm. The execution times for the 100 MB file are reported in table 3.1.

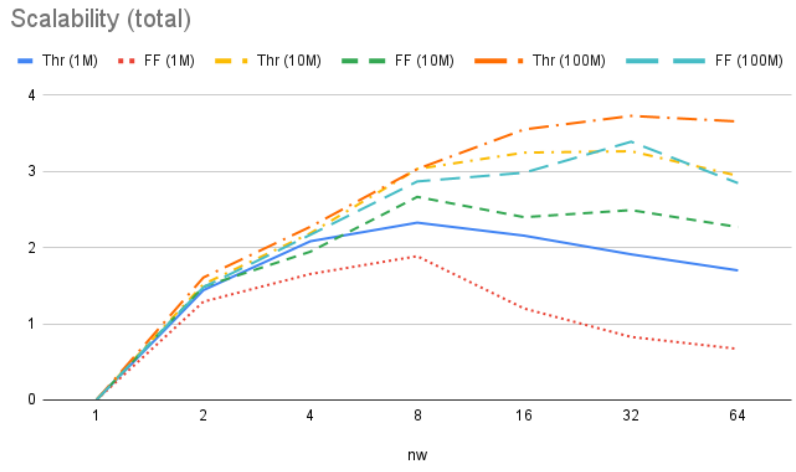


(a) Speedup of the algorithm.

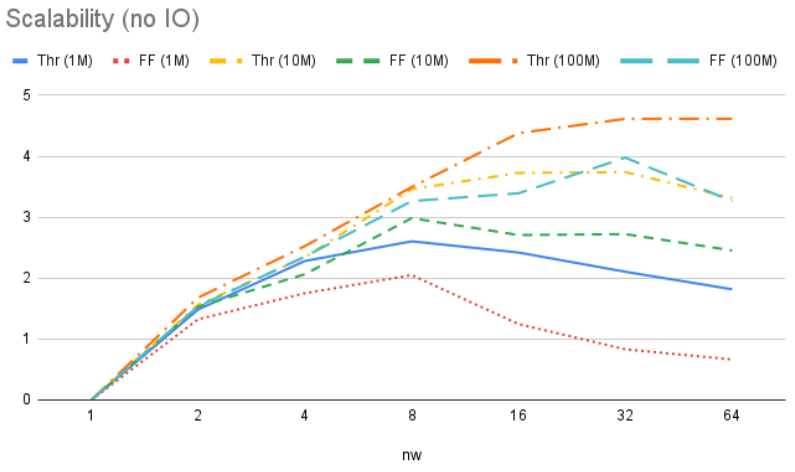


(b) Speedup of the algorithm (no IO).

Figure 3.1: Comparison of the speedup of the algorithm.



(a) Scalability of the algorithm.



(b) Scalability of the algorithm (no IO).

Figure 3.2: Comparison of the scalability of the algorithm.

NW	SEQ	THREADS	FASTFLOW	SEQ (NO IO)	THREADS (NO IO)	FASTFLOW (NO IO)
1	3.49	3.73	4.16	3.21	3.46	3.91
2	-	2.32	2.52	-	2.06	2.25
4	-	1.64	1.72	-	1.37	1.47
8	-	1.23	1.3	-	0.99	1.06
16	-	1.05	1.25	-	0.79	1.02
32	-	1	1.1	-	0.75	0.87
64	-	1.02	1.31	-	0.75	1.06

Table 3.1: Execution times (in secs) of the Huffman coding on the 100 MB file

Chapter 4

How to run the project

The folder contains a Makefile which needs to be run with the command `make`. The text files must be placed into the folder ”txt_files”, to make a file of dimension 10 MB and 100 MB execute the respectively commands

```
for((i=0;i<10;i++)); do cat file_1M.txt >> file_10M.txt; done
for((i=0;i<10;i++)); do cat file_10M.txt >> file_100M.txt; done
```

The project can be tested on the sequential, thread and FastFlow version using the following commands

```
./test_seq_huffman.out file_10M.txt
for((i=1;i<=64;i*=2)); do echo $i; ./test_th_huffman.sh $i file_10M.txt; done
for((i=1;i<=64;i*=2)); do echo $i; ./test_ff_huffman.sh $i file_10M.txt; done
```

The three commands will run twenty times the algorithm using the powers of two as number of workers and print the average execution times for the major steps and the total amount spent with and without the reading and writing phase. The final output file will be saved into the folder ”out_files”.