UNIVERSITÀ DI PISA

PARALLEL AND DISTRIBUTED SYSTEMS - REPORT

Parallel Huffman Coding

Antonio Strippoli (625044)

A.Y. 2022/2023

1 Introduction

In this report we analyze and present 2 parallel implementations of the **Huffman coding** algorithm: one implemented with the **C++ threads** of the *Standard Template Library (STL)* and one using the **FastFlow** parallel programming framework.

2 Problem analysis

The problem requires to compress, using the Huffman coding, a given input string.

No stream parallelism will be exploited in the solution, as it was not part of the request: we will analyze one input string at a time. Furthermore, the Huffman tree will be constructed **char-based** (i.e. based on the characters of the input, rather than its words).

In our analysis, the processing of each input can be subdivided into the composition of **7 stages**:

1. **READ**: read the input file on the disk and load it in memory as a string object;
2. **COUNT**: compute the frequency of each character in the string;
3. **TREE**: build the *Huffman tree* using the characters frequencies as weights;
4. **MAP**: build the *Huffman map* (mapping characters to *Huffman codes*) by traversing the *Huffman Tree*;
5. **ENCODE**: encode the input as a *binary string* (i.e. a string of only 0s and 1s) by mapping the characters of the input through the Huffman map;
6. **ASCII**: encode the binary string as a *ASCII string*;
7. **WRITE**: write the encoded input into a plain text file on the disk;

Note that the presented stages cannot be executed in parallel (on the same input), due to the intrinsic sequential nature of the Huffman coding.

2.1 Skeletons analysis

After analyzing the theoretical properties of the problem, we went through each stage and considered the **data parallel skeletons** that we could have employed before actually tackling and coding the parallel implementation of the problem.

We immediately identified 3 stages that can be parallelized:

- **COUNT**: we could **split the string into chunks** and employ a **map pattern** counting in parallel the characters of the chunks-divided input, eventually employing a **reduce pattern** to merge the partial countings.
- **ENCODE**: similarly to the **COUNT** stage, **map** and **reduce** patterns can be employed, with the only difference being that now the partial results are `strings` (as a result of the encoding using the Huffman map, which can be accessed concurrently as it is only read by the workers).
- **ASCII**: similarly to the previous stages, once again **map** and **reduce** patterns came up immediately. In particular, partial results would be `strings` as in the **ENCODE** stage, but particular attention will need to be put into the input chunking: they need to be multiple of 8, as an ASCII character corresponds to 8 bits.

Note that the **READ** and **WRITE** stages cannot be parallelized (as they involve I/O), and that the **TREE** and **MAP** stages are already $O(1)$ (with a very small constant as we are working with an ASCII table of 256 possible characters).

In the 3 mentioned stages, a **divide&conquer** pattern could be employed as well, but it typically requires more communication and synchronization and it would lead to a more complex analysis (as the merge steps typically require different times between each of them) thus we don't think it is worth to implement it for any stage.

A **prefix** pattern was also considered instead of the **reduce** one, but we don't think it would be straightforward to implement it and we are not interested in keeping prefix sums in this problem, thus we thought it would cause **more overhead** compared to the reduce.

Finally, only out of theoretical interest, we also thought about possibly upgrading the throughput of the parallel solution by employing a **pipeline** stream pattern, as we could easily split the work into stages and intuitively some could use higher parallelism degrees than others, but we will not discuss it further as it is not part of the request.

2.2 Performance analysis

Given our problem formalization in stages, the performance model of the final algorithm is trivially given by the following equation:

$$T_S = T_{read} + T_{count} + T_{tree} + T_{map} + T_{encode} + T_{ascii} + T_{write}$$

The theoretical best time of the 3 stages we will parallelize is given by $\frac{T_{stage\ sequential}}{n}$. However, parallelization always introduces overhead - which will explain why we will not be able to reach it - and will be analyzed and discussed later on.

3 Implementation

3.1 Versions

3.1.1 Sequential

Seeking support to the ideas we had, we proceeded to implement the sequential version of the algorithm. By implementing it, we gained further insights on the stages requiring more time:

Step	File Size		
	1MB	10MB	100MB
INPUT	750	14,669	180,934
COUNT	2,783	29,309	301,596
TREE+MAP	145	145	137
ENCODE	12,515	178,588	1,560,912
ASCII	48,483	496,087	4,978,617
OUTPUT	1,512	13,039	126,132

Table 3.1: *Times in us of each step (10-steps averaged)*

The **ENCODE** and **ASCII** stages consistently exhibit the highest average times, and they will surely benefit from parallelization. The **COUNT** stage is already fast, and in a stream scenario (when receiving multiple inputs) we could think about putting more working power in reducing the other 2 stages. Since this is not our case, the overall service time will still benefit from parallelizing it, so we will provide a solution for this stage as well.

3.1.2 Thread

There are different technical ways to implement the stages. We have taken into account several considerations for each stage:

COUNT stage

- **Static vs dynamic load balancing**: as there are no areas in the input that could require more time, we decided to go for a static solution. Thus, the input has been divided across the workers in chunks of equal size (apart from possibly the last one);
- **Counting data structure**: in order to store the characters frequency, we first thought about using an `unordered_map<char, int>`. However, as the characters in the input can only span across the 256 possibilities of the 8-bit ASCII table, we went for a `vector<int>` of fixed size, as it better exploit cache locality and generally yields faster results;
- **Partial results collection and reduction**: workers could manage an independent counting data structure in their own stack memory and then synchronizing through a `mutex` to update a *shared* counting data structure yielding the final result. The synchronization could be implemented by either locking the entire structure or by locking at element-level (allowing concurrent writing). We thought that the time to synchronize would take too much

time, so we decided to instead go for a *shared* `vector<vector<int>>` containing the partial results of the workers, which will then be reduced by the emitter as soon as each thread finishes, with the attention to preserve the order. This decision has also been supported by experimental evidence (see Section 4);

- **Counting while reading**: technically, it is possible to **COUNT** as soon as a chunk has been read from the file. We decided to go against it since it would have complicated the logic (i.e. **READ** and **COUNT** would have needed to be implemented in a single function);

ENCODE stage

- **Static vs dynamic load balancing**: we went for a static one, as the same arguments as in the **COUNT stage** applies;
- **Partial results collection and reduction**: we kept a *shared* `vector<string>` instead of letting each worker manage an independent string, as the same arguments of the **COUNT** step apply. The main difference is in the **reduction** step: in C++ there are several ways to concatenate a sequence of strings (e.g. `+ operator`, `strcat()`, `stringstreams`...). We thought about simply going for the + operator, as the overhead could be highly manageable and it makes the code much more readable. Unfortunately, it isn't possible to **pre-allocate space** for the string, as Huffman encoding uses **variable-length** codes;

ASCII stage

- **Static vs dynamic load balancing**: we went for a static one, as the same arguments as in the **COUNT stage** applies. However, we had to be particularly careful in choosing the chunks size: we made it a multiple of 8, as the ASCII encoding works on **8 bit at a time**;
- **Partial results collection and reduction**: the same observations as for the **ENCODE** stage applied;
- **Writing while encoding**: it would be possible to start writing `chunk[i]` as soon as `chunk[i-1]` has been written, saving space and possibly some time. We decided to not go for this approach, as the same arguments as in the **COUNT** stage applied.

We also thought to manage a **thread pool** to limit the **overhead** caused by the thread spawning/termination, but we decided to go against as we thought that this overhead would not be as significant as the cons of implementing a more structured solution.

3.1.3 FastFlow

All the considerations for the thread-based approach applied for FastFlow as well.

More specifically, the **COUNT** and **ENCODE** stages have been implemented using a `ParallelForReduce` object, with a call at `parallel_reduce_static`: this allowed to preserve the high-level control of the library, making the code very minimal while implementing the same ideas described in Section 3.1.2. We also tested a dynamic scheduling approach by calling instead `parallel_reduce`, but we obtained way poorer performances (see Section 4).

As previously discussed, the **ASCII** requires chunks size of multiple of 8, which didn't appear to be possible using the `ParallelForReduce` object. Since we required more control, we employed the less restrictive `ParallelFor` object, with a call at `parallel_for_static` and the merging phase performed sequentially (unfortunately out of the library control).

3.2 Usage

You can start by **retrieving** the project:

```
$ git clone https://github.com/CoffeeStraw/parallel-distributed-systems
$ cd parallel-distributed-systems
```

Next, you will need to generate **3 text files** named *"1MB.txt"*, *"10MB.txt"*, *"100MB.txt"* under `tests/inputs/`, since they are required by the benchmarks. We provided a simple `python3` utility to generate them, but you may use any other method you wish:

```
$ cd tests/inputs/
$ python3 input_gen.py 1
$ python3 input_gen.py 10
$ python3 input_gen.py 100
$ cd ../..
```

We provided a `tests/` directory containing all the mains created to **benchmark** the versions:

- `test_0_sequential_steps.cpp`: measurements of the time required by the steps of the sequential version;
- `test_1_sequential.cpp`: measurements of the time required by the sequential version;
- `test_2_thread.cpp`: measurements of the time required by the multi-threaded version;
- `test_3_fastflow.cpp`: measurements of the time required by the FastFlow version;

We also conducted tests to try out alternatives and check for output consistency between the versions of the algorithm:

- `test_chars_frequency_alternatives.cpp`: benchmark the discussed alternatives to the coded versions of the **COUNT** stage;
- `test_chars_frequency_consistency.cpp`: verify that the coded versions of the **COUNT** stage are consistent;
- `test_huffman_encode_consistency.cpp`: verify that the coded versions of the **ENCODE** and **ASCII** stages are consistent;
- `test_chars_frequency_overhead.cpp`: compute the overhead of the **COUNT** stage;
- `test_huffman_encode_overhead.cpp`: compute the overhead of the **ENCODE** and **ASCII** stage;

You can then proceed to compile one of the previously mentioned, and execute the project. The compilation step will be the same for every other test. It is required for *FastFlow (3.0.1)* to be installed (note that in this example, FastFlow has been put under `/usr/local/include`):

```
$ # NOTE: substitute FILE_NAME with the filename of the desidered test
$ g++ -O3 -I /usr/local/include -std=c++17 src/*.cpp tests/FILE_NAME -o main
$ # NOTE: follow the instructions appearing on the screen
$ ./main
```

4 Results

In this section we report the measurements obtained with the parallel versions and discuss them. These results were obtained by running them on a AMD EPYC 7301 CPU machine with **32 physical cores** (2 way hyper threading), averaged over **10 runs**. We present **completion time**, **speedup** and **scalability** of the algorithm running on 3 text files of different dimension (1MB, 10MB, 100MB) containing random characters.

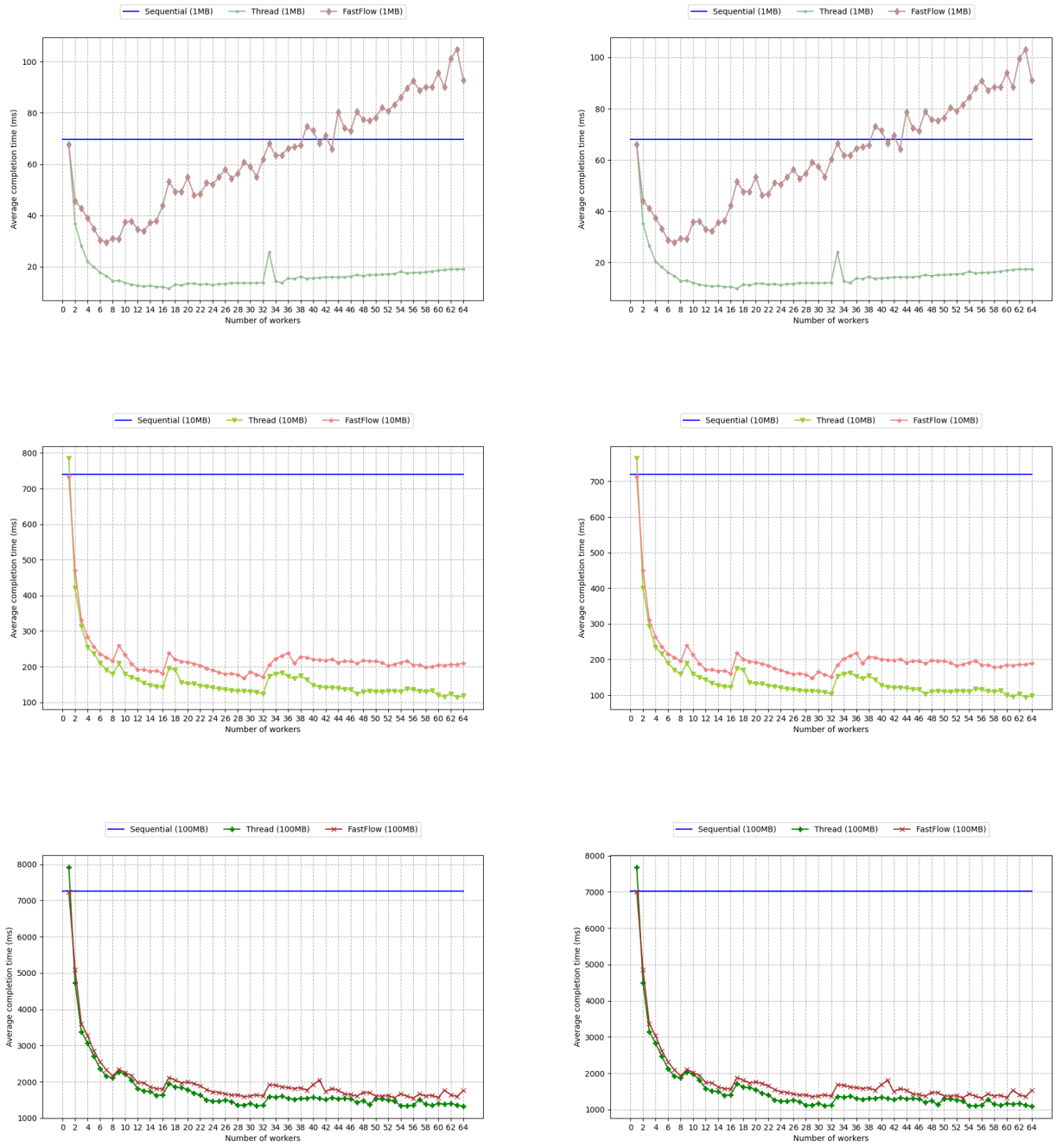

Figure 4.1: *Average completion time (in ms) of parallel versions compared to sequential. Right plots do not consider I/O operations.*

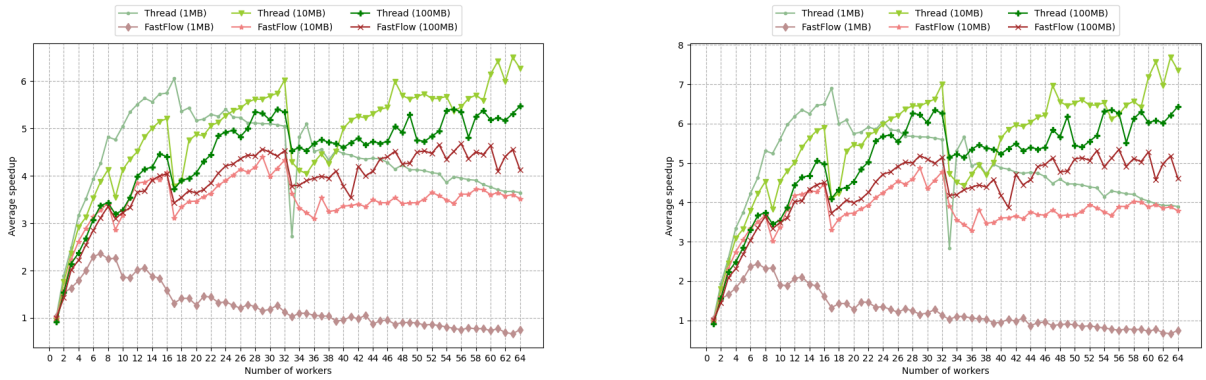

Figure 4.2: *Average speedup of parallel versions. Right plots do not consider I/O operations.*

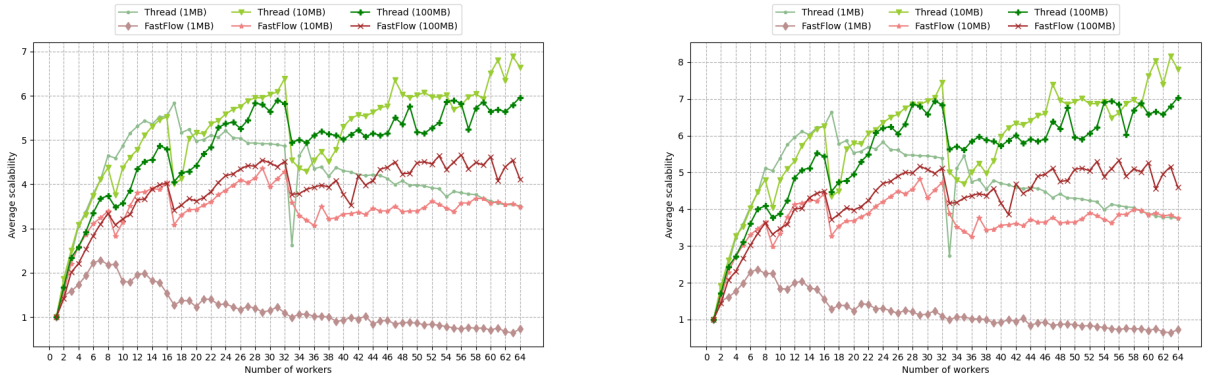

Figure 4.3: *Average scalability of parallel versions. Right plots do not consider I/O operations.*

We notice how the **best results** were provided by the **threads** version, even if it constitutes the most complex implementation. At the same time, the **FastFlow** version provided quite satisfactory and comparable performance results to the threads versions, with the FastFlow framework having the advantage of being an high-level customizable library and requiring less effort to be implemented. This notwithstanding, both the **threads** and **FastFlow** versions provide sufficiently abstract parallel mechanisms that allowed us to focus on the main program logic.

The peak generally happened at **32 workers**, which can be easily explained by the architecture of the machine. Moreover, being **2 way hyper threading**, we can also see minor speedup/scalability improvements when reaching 64 workers.

We noticed that both the **threads** and the **FastFlow** implementation provided some speedup even for the *1MB* file: the overhead of our implementation cancels the benefit gained from the parallelization.

Indeed, we measured the **overhead** of the multi-threaded version for the 3 stages. The overhead is given majorly by the **merging of the partial results**, followed by the **thread spawning/termination** and **creation of the shared vectors** hosting partial results. We tested with 32 workers, on the 100MB files (averaged over 10 iterations):

- **COUNT**: **1,819** us out of **11,853** us;
- **ENCODE**: **609,166** us out of **832,673** us;
- **ASCII**: **71,080** us out of **301,976** us;

Finally, we tested the the **alternatives** discussed in Section 3.1.2 on the *100MB* file, for the **COUNT** stage, 32 workers:

- Thread final: 11,990 us;
- Thread with lock on vector: 13,406 us;
- Thread with lock on each element of the vector: 12,599 us;
- FastFlow final: 27,676 us;
- FastFlow but performing a manual reduce: 30,331 us;
- FastFlow but with dynamic scheduling: 32,054 us;

As we can see, the final chosen approaches not only are the easiest to manage (we don't have to manage anything beside a shared vector of partial results), but also proved to be the most performant (even if slightly).