Parallel implementation of Huffman coding

Alessandro Cabassa

1 Introduction

Although many different Huffman coding implementations exist, the one chosen for the project exploits a priority queue (max heap) of `Nodes`—each containing a symbol and its frequency—in order to build the classical encoding binary tree, with symbols being leaves and the path from the root to a leaf representing the encoding of the symbol. The overall complexity of this algorithm is $\mathcal{O}(n \times \log n)$.

The programs have been tested on different sizes of *"La Divina Commedia"*, that is, the repeated append of the text to itself for a given amount of times.

The (sequential) program steps can be summarized as follows:

1. The file is read, its text is saved and the frequency of each of its characters is registered;
2. The symbols are stored into an array of symbols, while their frequencies are stored in another array, maintaining the order;
3. Each symbol and its frequency are used to create a `Node` structure, which represents a node of the binary tree using to generate the codes. Each one of this nodes gets pushed into the priority queue, which maintains the nodes with the largest frequency towards its head. Now, until the queue is not emptied, the first two nodes (thus, the ones having the largest frequencies) get extracted, merged into a dummy node containing the sum of their frequencies and pointers to them as children, and enqueued back into the heap.
4. Now that the full tree is created, the codes are recursively generated, saving the string of bits representing the followed path up to each leaf as the code of the character residing in the leaf itself.
5. Next, by iterating over each symbol of the saved text of the file, each character gets replaced by its previously-generated code, ultimately creating a string representing the fully-encoded text.
6. Finally, depending on the user's choice, the last step can be one of the following:
   (a) Write the encoded text to file: in this case, the aforementioned string's bits are written to a new file, finally creating the compressed version of the input.
   (b) Decompress the string: the encoded string gets decoded back and printed to `stderr` in order to check the correctness of the program.

2 Parallel implementations

Both parallel versions—the one using *pthreads* and the one using *FastFlow*—were developed trying to parallelize every step, from the initial file reading to the last file writing. As exceptions, the tree generation, the codes generation, and the optional decompression steps were not parallelized, as ensuring correctness of these tasks is pretty hard and introduces a lot of complexity to the program.

Here, we present both implementations.

2.1 Threads implementation

This version made heavy use of the farm pattern, where nearly every step is parallelized using a farm of threads. The full program structure can be seen in figure 1.

The first task consists in reading a file, saving its text and computing the frequency of each one of its characters. To do so, each thread computes its fraction of the file to be read, positions at the right place and starts reading characters, incrementing the specific character count in a local hashtable on each character occurrence. Furthermore, it appends each read character to a local string. Finally it distributes each tuple *¡character, frequency¿* of the local hashtable to different reducers (stored in an array). At this last step, a synchronization mechanism is employed (since

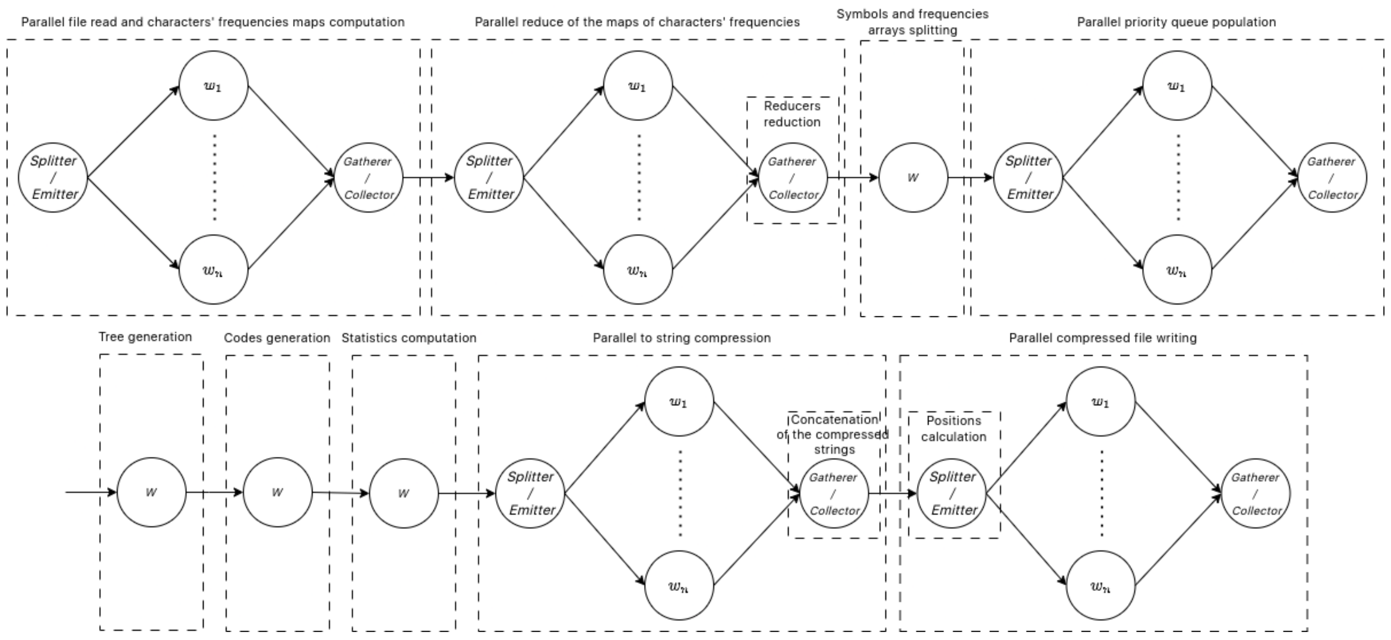
Parallel file read and characters' frequencies maps computation
Parallel reduce of the maps of characters' frequencies
Symbols and frequencies arrays splitting
Parallel priority queue population
Splitter / Emitter
w_1
w_n
Gatherer / Collector
Reducers reduction
W
Tree generation
Codes generation
Statistics computation
Parallel to string compression
Parallel compressed file writing
Concatenation of the compressed strings
Positions calculation

Figure 1: Parallel application structure

a reducer, that is a specific position in the array of reducers, may be concurrently accessed by different threads for pushing their local tuples inside of it).

The following step consists in running in parallel the reducers farm, whose objective is to populate a final hashtable containing the sum of the frequencies of each symbol computed by different threads at the previous step. As previously, this is done by employing a mutual exclusion synchronization mechanism. When the threads of the farm are joined at the gathering phase, the portions of text that have been collected by the workers inside of the first farm are concatenated into a single string which represents the read content of the file.

Following the farm, there is the first step not being parallelized, since it just requires to create an array of symbols and an array of frequencies by, respectively, scanning the keys and the values of the hashtable. Since it is a simple iteration over the `unordered_map` data structure which doesn't contain a huge amount of data (since alphabet symbols are limited), employing threads and the relative synchronization mechanisms would just result in overhead, without many parallelization benefits.

Moving forward, another farm of threads is employed, this time populating the max heap data structure in parallel. This is done by splitting into chunks the just-computed arrays of symbols and frequencies and by using a `mutex` in order to prevent race conditions when pushing into the same data structure.

Two sequential operations follow, respectively the tree generation procedure, and the codes generation procedure. In principle, the latter could've been parallelized by dividing the tree into different portions and assigning each one of them (after a certain level, since the tree is binary) to a different thread. For example, with 16 threads, all of them could fully operate on their sections of the tree only from the 5th level downwards (assuming that in each level, every node has both its child nodes). Although there is this possibility, this phase isn't the particularly expensive computationally-wise, thus the effort to parallelize the section would be too large compared to the gain, which is negligible with respect to the total program execution time.

Going forward, there is another small sequential phase in which the average length of the generated codes is computed. This is not needed by the program to function, but improves the performance when compressing the text into the string.

At this point, the compression of the text into the string of bits to be written to the binary file is made. Again, each thread computes its chunk of the text it will work on, then, for each character in the chunk, it substitutes the character with its encoding. The resulting string of each thread is stored into an array of strings, which are then combined into a single one during the gathering phase.

At this point, assuming the user did not select the option to verify the compression correctness, the bits represented by the compressed string have to be written to file, actually creating the compressed output. To do so, right before the threads start executing their jobs, a new file of the size of the compressed string's bits (that is, the length of the string divided by eight, since each bit is represented as a char, thus occupies a byte) is created and filled with dummy values. This will allow threads to write on different positions of the file. Now, instead of letting each thread compute its chunk of the compressed string to work on, an array of tuples *¡from, to¿* (representing the actual chunk for each thread) is computed. This is due to the fact that the extremes of the chunk have to be byte-alligned for the writing phase to work correctly, and this can't be done by the threads themselves since it would require passing information inter-thread such as the positions at which it finished writing the byte. Similarly, an array of starting positions in the file is created . Now, each thread positions at the right place in the file and starts analysing its portion of the compressed string, writing the bytes.

If, instead, the user selected to verify the program's correctness, then, instead of the last farm, a sequential stage is employed, reversing the map of characters and their codes to a map of codes and their characters. Next, the function iterates over the compressed string, accumulating into a local string the bits and matching it to the reversed map as soon as possible. This can be done since the codes are all prefix-free, that is, no code is a prefix of another one.

2.2 FastFlow implementation

This implementation maintains the same steps as the previous, with the difference of clearly defining the emitting and collecting workers, as *FastFlow* requires. All the stages are `ff_Farm` objects, with one emitter and one collector each, all put into a single `ff_pipeline` object.

This version makes heavy use of structures grouping data inside, used as messages to communicate between a farm's components or even between the collector and the emitter of adjacent farms.

Each emitter is of `ff_monode_t` type, allowing the choice of the scheduling to the node itself. Every emitter node creates a number of tasks (message structures) equal to the number of user-specified workers, sending one to each of them.

The operations executed in the worker phases are the same of those specified in the thread implementation, with the only difference of working on pointers to data structures instead of data structures themselves, adding a level of indirection, possibly resulting in worse code readability.

Each worker waits to receive the *EOS* message from the emitter node in order to start sending out the task on which it performed some sort of work.

Similarly, each collector node waits to receive the *EOS* messages coming from workers before starting its computations. In particular, it waits until an amount of *EOS* messages matching the amount of workers hasn't been received, translating into a *de facto* barrier. The peculiarity of this specific synchronization mechanism is that, due to the library-defined input queues used as mailboxes for messages by each node, it doesn't require an atomic variable as normal barriers would, since it won't be updated concurrently but once per message reception.

3 Measurements and analysis

As previously mentioned, the three implementations have been tested on texts of different size, with the largest corpus being the concatenation of 200 "La Divina Commedia" containing more than 111 million characters in total. Here, we report the results relative to this last document in order to mark in a more clear way the computational-time differences between the implementations.

Furthermore, for each implementation, two different time measures were taken: one without considering the file writing phase, i.e. the last farm, and one considering the compression to file.

Each measurement is an average of 10 measurements on the same input number of workers, nonetheless, since the test machine can be accessed by many users at the same time, the values might fluctuate up or down based on the available resources and the current workload of the machine.

As it can be seen from table 1, the writing phase results in the most expensive operation in the sequential implementation, more than doubling the total execution time.

Without file write (in μs)	3105600
With file write (in μs)	7678400

Table 1: Computational times of the sequential implementation without file writing (top) and with file writing (bottom)

Considering now the version using pthreads, it can be easily seen by comparing table 1 and table 2 that, up to four threads, there is no benefit in using parallel workers if we are not writing the compressed file. If, instead, we consider the file writing phase, then the advantages of using threads become evident even since the employment of two of them.

Furthermore, the first column of table 2 shows that the threaded version of the program results in a big amount of overhead with respect to the sequential one, without taking in consideration the file writing step.

Lastly, it can be seen from the measurements that although incrementing the number of workers up to 8 nearly halves the computation time, from 8 threads onward the execution time doesn't drop as sharply at each step. Presumably, this happens due to the fact that the overhead tied to the computation of the chunks and to the splittings and mergings of data structures becomes more and more accentuated, making the gain of using many workers negligible.

Furthermore, as the number of workers grows, also does the time spent on synchronizing the threads in some specific areas of the code.

Number of workers	1	2	4	8	16	32
Without file write (in μs)	6105260	3341260	1840610	1100770	746988	584733
With file write (in μs)	8409000	4453320	2440960	1416060	972385	723687

Table 2: Computational times of the threads implementation without file writing (top) and with file writing (bottom)

Speaking now about the FastFlow-based implementation, it can be seen from table 3 that the first part (up to 16 threads) is comparable with the measurements relative to the pthreads version, although it is a bit slower. Additionally, it can be seen that increasing the number of workers from 16 to 32 yields a deterioration in the execution time. Both these facts may be explained by the fact that this version of the program is based on a big number of allocations and deallocations of structures needed by each step, as well as a great amount of message passing between nodes, thus, increasing the number of workers over a certain threshold results in an overhead being greater than the gained speedup. Moreover, the overall structure of the program is more complex and, since FastFlow is an implementation sitting on top of pthreads, it works at a higher level of abstraction, inevitably resulting in some overhead in order to ease the application programmer's job.

Number of workers	1	2	4	8	16	32
Without file write (in μs)	7097060	3745870	2143380	1313470	1153760	1513060
With file write (in μs)	9175150	4862490	2870090	1588990	1337390	1747140

Table 3: Computational times of the FastFlow implementation without file writing (top) and with file writing (bottom)

In both versions, the sections which have been maintained sequential affect the execution time in a negligible way, each requiring just a bunch of (μs) (up to some hundreds for the codes generation phase).

Following, we report the computed speedup and scalability measures of both implementations; the first table doesn't consider the file writing time, whereas the second does.

In 2 we can see the scalability measures of both versions plotted in a graph, which points out the fact that the second implementation is a little bit slower than the first and that, after crossing 16 threads, its performance drops back close to the 8 workers one.

Number of workers	1	2	4	8	16	32
Speedup (threads)	0,5087	0,9295	1,6873	2,8213	4,1575	5,3111
Speedup (FastFlow)	0,4376	0,8291	1,4489	2,3644	2,6917	2,0525
Scalability (threads)	—	1,8272	3,3170	5,5463	8,1732	10,4411
Scalability (FastFlow)	—	1,8946	3,3115	5,4033	6,1512	4,6905

Table 4: Speedup and scalability measures for both parallel implementations, without file writing

Number of workers	1	2	4	8	16	32
Speedup (threads)	0,9131	1,7242	3,1456	5,4224	7,8965	10,6101
Speedup (FastFlow)	0,8369	1,5791	2,6753	4,8322	5,7413	3,3948
Scalability (threads)	—	1,8882	3,4449	5,9383	8,6478	11,6197
Scalability (FastFlow)	—	1,8869	3,1968	5,7742	6,8605	5,2515

Table 5: Speedup and scalability measures for both parallel implementations, with file writing

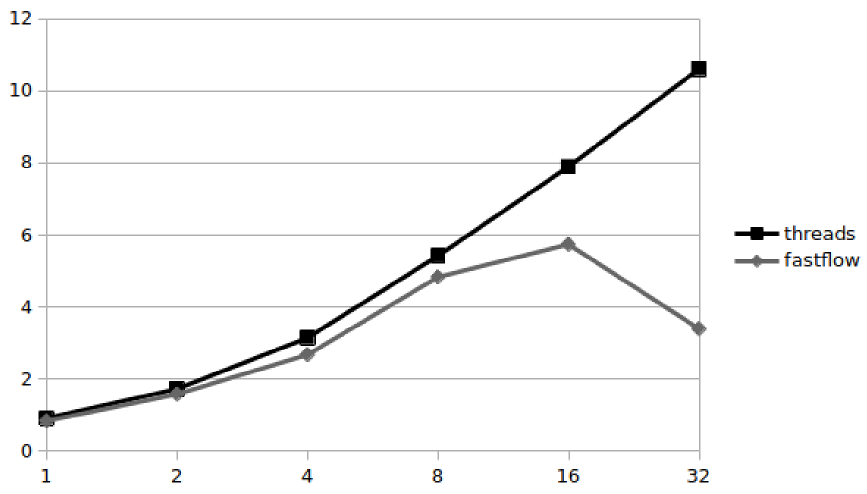

Figure 2: Charted speedup measures (also considering file writing)

4 Tests for improvement

The first idea for improvement was to also try to parallelize the decompression task, which is executed only if the user asks so and replaces the file writing phase. The result is remarkable, obtaining a great speedup, but the result of the decompression is wrong. In particular, since the compressed string contains a sequence of bits represented as chars, it is impossible for a thread to know the chunk on which it has to work on, since using the usual static distribution mechanism (computed as the total amount of elements divided by the number of workers, with each worker computing its start and end positions in the sequence) often (almost always) results in a thread starting its computation from a bit belonging to the encoding of a character whose analysis has been started by the thread responsible to the preceding chunk. Thus, after many attempts to parallelize this section, we conclude that it is not parallelizable without introducing additional overhead such as an array of starting and ending positions of the codes.

The second tested improvement was to implement a better load balancing mechanism which partitioned the work in such a way that the last worker would not be assigned a longer block than the preceding ones. Unexpectedly, this resulted in worse performance, increasing the execution time of the program of about $3000\mu s$. The assumption is that, with the given text corpus and with the given amounts of threads, this theoretically better load balancing results in a worse cache utilization, probably exposing less spatial locality. This, however, should vary with different input files and numbers of workers, thus we conclude that this doesn't prove that this optimization is pointless.

In the FastFlow program, the static optimizations in the pipeline have been tested as well, providing no increase in performance, as expected. This can be explained by the fact that each emitter and each collector of every farm have been custom-defined and that the pipe is made of a sequence of farms, with the same structure as showed in 1.

Again, in the FastFlow version of the program, adding the `-DNO_DEFAULT_MAPPING` compilation flag provides a minor improvement with 32 threads, leading the execution time closer to the one of 16 workers, considering also the file writing phase. This might happen because, for some reason, the working set doesn't fit the last-level cache (which is an L3 in the provided machine), and by not pinning threads to sibling cores this might result in a more efficient execution.

For both implementations, the `jemalloc` library has been tested, even though only on personal machines as the remote couldn't install and configure the library. This did not yield any performance improvement.

5 Possible improvements

For what regards the native threads version of the program, a possible improvement would be to use a thread pool which, through some policy, asks for a task and computes it, instead of waiting for a splitting process made by the main, sequential thread to spawn other threads and the starting their work. Although this would probably be more efficient, the threads would need to wait while one of them performs the "synchronization steps", such as merging the partial results computed in a previous phase and so on.

Together with the pool of threads asking tasks to compute from a central repository comes a better load balancing implementation, even though this should not provide great speedups since the amount and the type of work is the same for each thread. Threads will never work on different jobs, since, as explained, each program phase depends on the results provided in the previous one, so the only way to parallelize the program is to split the work of each phase between multiple workers, thus resulting in the same type of job per thread.

In the FastFlow version, the thread creation for each farm can be parallelized, but this would yield a minor and negligible performance improvement, since each creation phase takes from $7\mu s$ to $14\mu s$.

Furthermore, again considering the FastFlow implementation, the number of messages (tasks) sent between emitters, workers, and collectors could be lowered, by creating just a few of them (ideally one) with many more shared structures inside, with the majority of them being useless for many program phases. This would decrease the amount of allocation, pointer copies, and deallocation, at the expense of the readability of the program.

Finally, some efforts to allow vectorization can be made, especially in the first phases working on vectors with simple operations, probably resulting in a moderate speedup.

6 Execution

The programs can be compiled with the *make* tool, issuing the commands `make seq`, `make par`, and `make ff` to respectively compile the sequential program, the one using pthreads, and the one using fastflow.

Each one of the versions accepts a *v* as last parameter in order to check the correctness of the code. In this case, no output file will be written, but the decompressed text will be printed on `stderr`. Tests on the correctness were made by redirecting the output to a new file and comparing it with the original file with the `diff` tool.

The sequential program can be run as `./seq commedia200.txt` or `./seq commedia200.txt v`, where *commedia200.txt* is the to-be-compressed file.

The pthreads program can be run as `./par commedia200.txt 16` or `./par commedia200.txt 16 v`, with 16 being the number of workers.

The FastFlow program can be run as `./ff commedia200.txt 16` or `./ff commedia200.txt 16 v`, similarly to the native-threads-based program.