Parallel Huffman Coding

Giulio Paparelli (546851) A.Y. 2022/23

Introduction

We here present three implementations of the Huffman Coding algorithm.
The first one is a simple sequential version, the second exploit C++ threads of the STL, and the last one use the parallel programming framework FastFlow.

Problem Structure

The task is to compress a given file using the Huffman Coding.
In the proposed implementations we only exploit data parallelism techniques, as stream parallelism was not part of the assignment.

In all the implementations we used a step-by-step approach to complete the task:

1. **READ**: Read the input string from a file in the disk;
2. **COUNT**: Compute the frequency of each character;
3. **TREE**: Build the Huffman Tree using the frequencies;
4. **MAP**: Build the Huffman Map: associate each character with its Huffman code by traversing the Huffman Tree;
5. **ENCODE**: Encode the input as a binary string, each character is represented with its Huffman Code;
6. **ASCII**: Encode the binary string in ASCII;
7. **WRITE**: Write the produced ASCII string in an output file;

Parallelism Reasoning

Once identified the steps of the problem we considered which data parallel skeleton could have been applied.

It is immediate to see that the **READ** and **WRITE** phases cannot be parallelized as they involve I/O.
The step **TREE** and **MAP** are already $O(1)$ since the working maps are limited to 256, as the all possible ASCII characters.

We then considered the three remaining steps:

- **COUNT**: Split the input string into chunks, use a map pattern to count in parallel the chars of each chunk, and finally use a reduce pattern to merge the frequencies;
- **ENCODE**: As before, we can use map and reduce pattern to divide the input string in chunks and, for each chunk, produce its encoded version, and finally merge them.
  It is worth mentioning that this is possible without any locking because the Huffman Map is accessed in a read-only fashion by each worker;
- **ASCII**: As in the **ENCODE** step, we use map and reduce to encode each binary chunk into its ASCII encoded version and then merge them in the final string that will be written in the output file;

We notice that ideally it would be better to not merge in the **ENCODE** step, so that we could apply the **ASCII** phase directly to each chunk and avoid the onerous task of merging chunks twice.
This is not immediate because of Huffman Codes.
ASCII characters are 8 bits, while Huffman Coding have as objective to use less bits than that.
As a result we have that the binary chunks can not be encoded in ASCII directly: each chunk may have a length not divisible by 8.
To avoid this we merge each binary chunk and then add a tail padding to make the full binary string ASCII encodable.

Implementations

Sequential

The implementation is straightforward and it gives us an idea on which are the more onerous tasks, and which one may benefit the most from parallelism techniques.
Executing the sequential version we obtained the following results:

SIZE	READ	COUNT	TREE	MAP	ENCODE	ASCII	WRITE	COMPLETION
1MB	0.000932006	0.00103855	1.90836e-05	1.10066e-05	0.0143293	0.0477107	0.000980489	0.0650236
5MB	0.00410689	0.0108704	1.89976e-05	1.27638e-05	0.0792452	0.240511	0.0036129	0.338387
10MB	0.0113682	0.0253271	2.5167e-05	1.25558e-05	0.149455	0.481137	0.00715799	0.674495
50MB	0.0740449	0.14613	1.95666e-05	1.28422e-05	0.721238	2.41455	0.0595949	3.4156
100MB	0.159985	0.294986	1.94908e-05	1.2449e-05	1.40508	4.82826	0.118993	6.80735
300MB	0.562113	0.90021	1.94344e-05	1.24732e-05	4.22075	14.4939	0.369905	20.5469

Where the timings are expressed in seconds and averaged between 5 consecutive executions.

As expected, the **ENCODE** and **ASCII** steps are the more expensive phases as they have to merge the partial results by concatenating strings.
The time required to concatenate large strings is slightly mitigated by preserving the amount of space beforehand.

Multithread

For each of the three parallelized phase we stayed consistent with the techniques applied, more specifically:

- **COUNT, ENCODE, ASCII**:
  - **static balancing:** The complexity of the input is constant, there is no chunk that may take more time to complete than others. Hence, we divided the input among threads, giving to each of them a section to work on.
  - **partial results:** Each worker could have its own independent partial results structure and update the shared one by gaining the lock on it. The synchronization, either by locking the whole shared structure or by locking the single element to be updated, would take too much time and would make the code more complex to read and debug, hence we always decided to go with a shared partial results data structure.
- **COUNT**
  - **counting structure:** Instead of a `unordered_map<char, int>` we decided to use a `vector<int>` of length 256, and use the int representation of each char as index to increment the value at that index, which represent the char frequency.

- **counting stream:** It would be possible and probably more efficient to count the frequency of chars while reading from the file. We decided to not use this technique as it would have altered the clean step-by-step structure of the project.

FastFlow

The considerations for the multi-threaded implementation are also applied here.

More in detail, the **COUNT** and **ENCODE** phases are implemented using a `ParallelForReduce` object, with a call to the function `parallel_reduce_static()`, in which the input data is divided into equal-sized chunks, and each chunk is processed independently by a separate task.
The partial results obtained from each task are then combined to produce the final result.
This allowed to keep the code lean and at the same time have to implement the same idea presented in the previous section.

As previously said, the **ASCII** step requires chunks of size multiple of 8, which is not possible with object `ParallelForReduce`.
We then used the less restrictive `ParallelFor`, calling the function `parallel_for_static` and performed the merging phase sequentially.

Project Structure and Usage

Project Structure

The project has the following structure:

- **app:** folder of the file `main.cpp`, which produce the executable application
- **data:** folder of the input and output files
  - **input:** folder where are stored the input files that the code uses
    - `ascii_at_random.py`: a python script that creates a file of given size of randomly chosen ASCII character
    - `input_files_generator.sh`: a bash script that calls `generator.py` 6 times, creating in the input folder 6 files of different dimensions (1, 5, 10, 50, 100, 300 Megabytes) which are used for measuring the performance
  - **output:** folder where the encoded strings are written into output files
- **source:** folder of implementations
  - **headers:** folder of headers
    - **steps:** headers for the declaration of the various steps of the project
    - `huffman.hpp` is the header for the implementations of the project
  - **steps:** implementations of the steps of the project
  - `huffman.cpp` is the implementation of the different versions of the algorithm
- **results:** folder where are stored the performance measurements
- `compile.sh` is a simple bash script that compiles the code and produce an executable `main`

Usage

We can compile the project by running `./compile.sh` in the project root folder.

Then we can choose to proceed in two different ways:

- production of measurements:
  - go to the folder `data/input` and run `./input_files_generator.sh`
    - this creates in the input folder the files `1MB.txt`, ..., `300MB.txt`
  - go to the folder `data` and create the folder `output` if not present
  - go to the project root folder and run `./main -a`, which produces the `.csv` files in the folder results
- interactive usage:
  - optional: go the folder `input` and run `python3 generator X --unit Y` where `X` is an integer representing the size and `Y` is either `KB` or `MB` to specify the unit file size to create a file of randomly chosen char of custom size and unit. The file will be named `XY.txt`
  - go to the folder `data` and create the folder `output` if not present
  - go to the project root folder, run `./main -i` and follow the process.

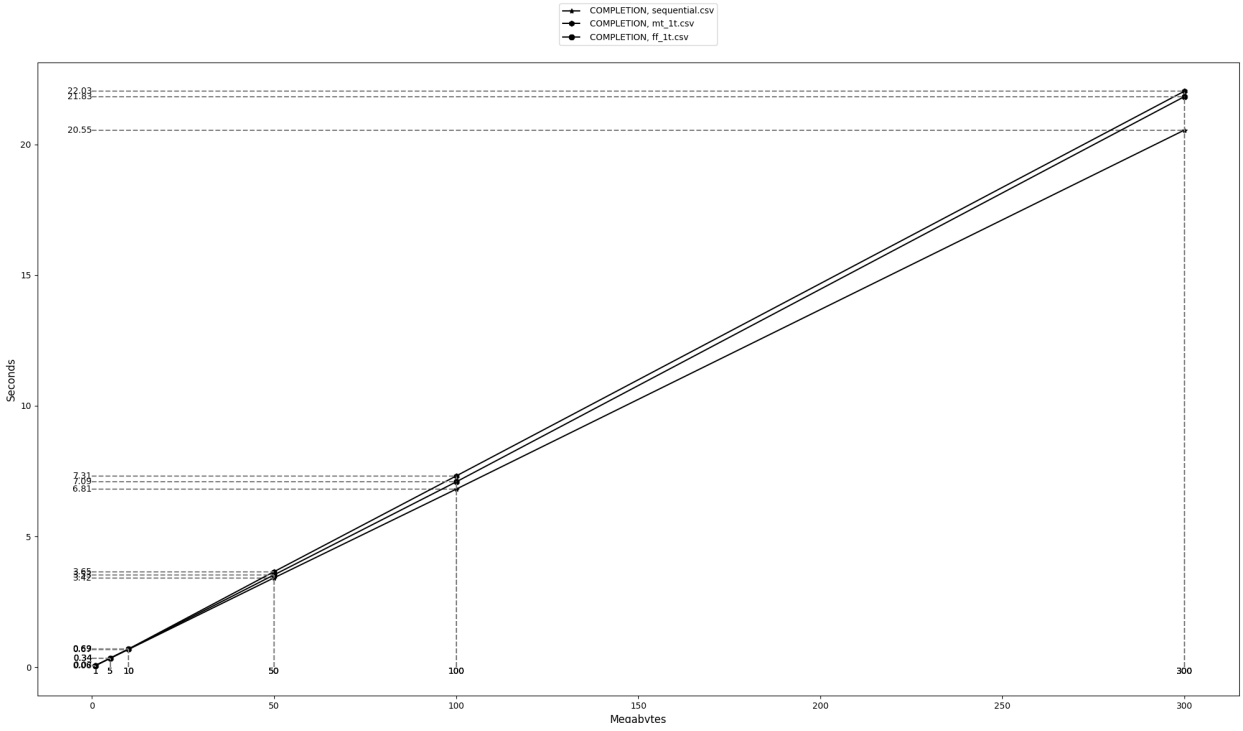
To plot more results the script `plotter.py` is located in the `results` folder.

Performance Analysis

Here we discuss the performances of the various implementations and show some plots to better visualize the results.

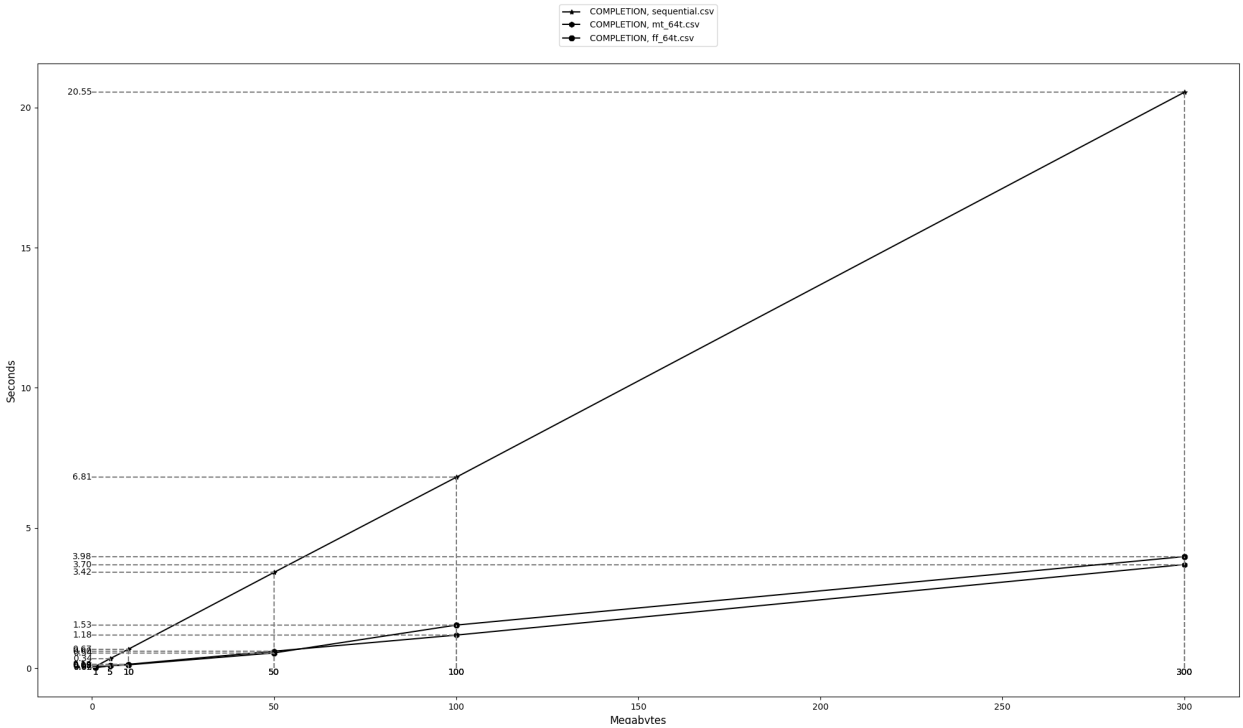
The results are obtained by running them on a dual socket machine with AMD EPYC 7301. Each socket has 16 cores (total 32 cores), 2 way hyper threading for a total of 64 hw threads.
The results are averaged over 5 consecutive runs.

First we show how the sequential version compares against the parallel versions that only use one thread:

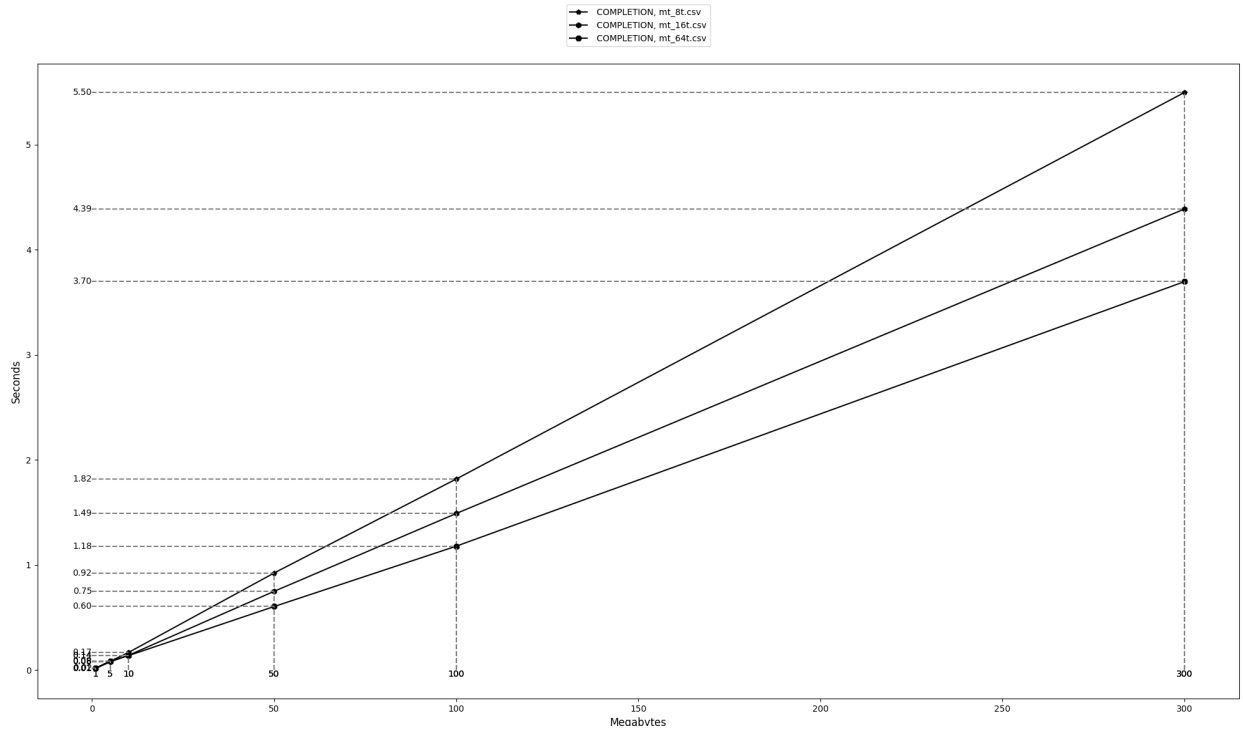


Then we present the comparison between the sequential version and the parallel versions using 64

threads:



Lastly we show how the completion time decreases as the number of threads increase:



Many other information can be derived from the measurements stored in the `/results` folder, and more plots can be produces using the python script `plotter.py`, by running `python3 plotter -h` in the `results` folder that allows to plot single measurements and compare the performance of any number of executions.

The obtained speedup on a file of 300MB with the multi-threaded implementation is summarized by the following table:

SIZE	SEQUENTIAL	N-THREAD	MT	SPEEDUP
300	20.5469	8	5.49697	3.73785
300	20.5469	32	4.35355	4.71957
300	20.5469	64	3.69631	5.55875

Considered Alternatives

Considering that the speedup we obtained is very low, maybe not even enough to justify the effort of parallelizing the code, we have considered, but ultimately discarded, the following ideas.

Vector of Vector of Pointers

In this scenario we wanted to avoid the onerous concatenation of strings that happens in the **ENCODE** phase, avoiding to rewrite strings that were already on memory.

To do so we modified the Huffman Map to be a map $\text{character} -> \text{string}$* that goes from character to string pointers

```
map<char, string*> huffman_map;
// ...
// the node is a leaf, store its character and code in the map.
if (node->left == nullptr && node->right == nullptr)
        huffman_map.insert(make_pair(node->character, new string(code)));
```

Then we modified the sequential implementation of **ENCODE** so it would produce a vector of pointers to strings.
Each element i of the vector would represent the $i-th$ char of the input string, and would point to the corresponding Huffman code of that character

```
vector<string*> huffman_encodings::sequential_string_to_binary(...){
        // ...
        vector<string*> s_encoded;
        // every char corresponds to a string pointer that points to the h code
        for(int i = 0; i < end; i++)
                s_encoded.push_back(huffman_map.at(s[i]));

        // ...
        return s_encoded;
}
```

This lead to the new multi-threading implementation of the **ENCODE** phase, which produces a vector of vector of strings*:

```
vector<vector<string*>> huffman_encodings::multithread_string_to_binary(...){
        // ...
        vector<vector<string*>> c_enc(num_threads);
```

```
        threads.push_back(thread(
                [&chunks_encoded, i, &s, start, end, huffman_map]() {
                        // fake name for readability
                        c_enc[i] = seq_str_2_bin(s, start, end, huffman_map);
                })
        );
        // ...
        // instead of merging the chunks
        return chunks_encoded;
}
```

This path was discarded for many reasons:

1. Building the vector of vector was slower than concatenating the string.
   When the vector of strings* is sequentially built in this implementation we have as many insertions as the length of the input string.
   Thus with large input the vector would be reallocated very often, which is a slow operation.
   The multi-threading version bares the same problem: the chunks are so large that the reallocation of the "vector chunks" makes the whole process not efficient.
   This is also not fixable by preallocating the vector using `preserve`, for similar reasons: the total memory required makes the program unusable for larger files.
2. The overall code readability and complexity would increase notably.

Padding Avoidance

With the goal of non concatenating the partial results neither in the **ENCODE** and in the **ASCII** steps, we tried to find a way to fix the problem of Huffman encoded chars that are less then 8 bits.
If we do not concatenate the partial results in the **ENCODE** step we remain with a vector of binary strings, the Huffman encoding of chunks of the input string.
To properly encode each binary chunk c_i in **ASCII** it would be necessary that $c_i \bmod 8 = 0$.

To do so we tried to add to a chunk c_i a tail padding by taking the missing bits from the chunk c_{i+1}.

```
for(int i = 0; i < chunks.size() -1; i++){
        string current = chunks[i];
        string next = chunks[i+1];

        int current_length = current.length();
        int missing_bits = 8 - (current_length % 8);

        // the current chunk is not a multiple of 8
        if(missing_bits != 0){
                pad = next.substr(0, missing_bits);
                current += pad;
                // remove the "stealed" bi from next
        }
```

While this solution could be usable, we decided to not fully explore it as it would greatly increase the complexity of the code, without a clear promise of performance gain.

Specifically we identified some challenges that this techniques carries:

- It is true that we can always "steal" from the successive chunk?
- This process can be done in parallel but it would require locking and synchronization techniques as threads could try to add to their last binary string the bits from the first binary string of another thread chunk

Bitwise Concatenation

Another way to increase the performance of the **ENCODE** and **ASCII** would be to see Huffman Codes not as `strings` but as `unsigned char`, which is represented with exactly 8 bits.

It would be possible to modify the Huffman Map to map each character c with a struct made of the `unsigned char` representing the code, and a number `size` used to store the "length" of the actual code.

In this way it would be possible to have a `vector<char>` representing the binary string encoded as an ASCII string ready to be written in the output file.

The algorithm is constructed with an idea inspired from the problem Sliding Window Maximum.
We start with an empty `char`, which is our window/buffer `buff`.
Then we start going through the file, checking each character c_i.

- if c_i has Huffman Code of length 8, i.e. `hc.size` = 8, c_i is a complete window, we add it to the vector as it is ready to be written in the output file
- if c_i has Huffman Code ($h(c_i)$) of length < 8 and $h(c_i) + b_{size} <= 8$ we concatenate $h(c_i)$ and w with bit-wise operations.
  - if $h(c_i) + b_{size} = 8$ we push the window in the vector of chars ready to be written, and reset the window
- if $b_{size} + h(c_i) > 8$ we take as many bit from $h(c_i)$ that can fit the window, we save the amount of bit of $h(c_i)$ are still meaningful and push the window.
  - after the push we reset the window and concatenate it with the remaining bits of $h(c_i)$

The following snippet is the fully implemented solution in cpp:

```
void encode_ascii_write() {
        // chars ready to be written in the output file
        vector<unsigned char> encoded;
        // current buffer, the "window"
        unsigned char buff = 0;
        // buffer size
        unsigned int b_size = 0;
        // bits that can't be inserted in the current buffer,
        // we will insert them in the next one
        unsigned int bits_leftovers = 0;
        // bits that can be written inside the buffer
        unsigned int buffer_free;
        // if there are leftovers this is true
```

```
	bool pending_bits = false;

	for (auto i = 0; i < file_content.size(); i++) {
		unsigned char current_char = file_content[i];

		// huffman code: unsigned char (8 bits), #significant bits
		HuffCode hc = huff_map.at(current_char);

		// easy case: we can fit the current char inside the current buffer
		if(b_size + hc.size <= 8) {
			// shift the buffer and insert the code
			buff |= (hc.code >> b_size);
			// increase the buffer size with the #inserted bits
			b_size += hc.size;
		// the current buffer + the current huffman code are more than 8 bits
		} else if(b_size + hc.size > 8) {
			// how many bits there are available in the buffer
			buffer_free = 8 - b_size;
			// number of bits that will not be written in the buffer
			bits_leftovers = hc.size - buffer_free;
			// insert the available bits in the buffer
			buff |= (hc.code >> b_size);
			// the buffer is now full
			b_size = 8;
	}
	// if the buffer is full we push it as this can be encoded as an ascii
	if(b_size == 8) {
		encoded.push_back(buff);
		buff = 0;
		b_size = 0;
		// if there are also leftovers from the current
		// char we insert them in the new buffer
		if(bits_leftovers) {
			buff |= (hc.code << buffer_free);
			b_size += bits_leftovers;
			bits_leftovers = 0;
		}
	}
	// if this char is the last one we insert the
	// buffer even if it is not a full buffer
	else if(i == (file_content.size() - 1))
		encoded.push_back(buff);

	string final;
	for (char c: encoded) {
		// write c in the output file
		bitset<8> rr(c);
		final += rr.to_string() + " ";
```

```
        }
        // cout << final << endl;
}
```

This solution has the best performance when executed sequentially as it basically merge the phases **ENCODE**, **ASCII** and **WRITE** in one, and the concatenation is much less onerous.
This is also the most promising way we have found to drastically increment the speedup in the parallel versions.
However, we decided to not purse further this solution as it would be very challenging to parallelize and it would make the code more complex to read and fix.