Advanced Parallel Systems - Project

CHALOYARD Lucas

January 10, 2022

1 Introduction

This report is written for the Advanced Parallel Systems project related to the use of MPI (Message Passing Interface).
I chose to do my project on the Huffman Algorithm because I find this algorithm pretty interesting, it allows me to have visible results and I thought that it was possible to improve it with MPI parallelization.

Huffman compression algorithm is designed to generate a new encoding for each characters of a file given in input.
With this new encoding will rewrite the file, switching each character by its new encoding, it should and will in most of the cases, reduce the total size of the initial file.
Let's do an example.

We'll assume that our initial file contains this :

```
Hello World!
Hello France!
Hello Grenoble!
```

The main idea of the Huffman Algorithm is to compute the total occureces of each character in the file in order to give them a new encoding which size is related to the total number of occurences.
For exemple in our case, we see that the character "G" is appearing only one time, contrary to the character "l" which is appaearing eight times. So, the new encoding for the character "G" will be longer than "l" encoding.
We'll discuss more in detail how those new encoding are generated in the next section.

So, what's the aim of my project ? I'll first implement the Huffman Algorithm using basic C++ in order to be able to compare its performances with the new version I'll implement, the parallelize one (using MPI).

Now, let's see what an Huffman execution usually produce.
After studying the frequency of each character, the encoding is generated.
Here is the encoding generated for our example with my version of Huffman algorithm :

```
b:00000000
G:00000001
c:00000010
a:00000011
F:0000010
d:0000011
W:000010
n:000011

:00010
!:00011
r:0010
 :0011
H:010
```

```
o:011
e:10
l:11
```

Like I said earlier, the "l" does have one of the smallest new encoding and the "G" have one of the longest one.
What about the output ? It wouldn't be useful to write the weird characters produced, but let's talk about the size.

The initial file has a size of 43 bytes, and the compressed file has a size of 186 bytes. So what's the matter here ?
In this case it is because of the metadata. In order for our Huffman algorithm to work we'll need to put some metadata in the compressed file, which in our case, have a size of 163 bytes.
I decided to make the metadata pretty verbose, it allows use easier debugging but it could be reduced a lot, and it should be if the algorithm needs to be used in real case scenarios.
So we can still conclude something, if we're only talking about of the interesting data of the compressed file (not the metadata), we can see that they have a size of 23 bytes, which is almost twice as less as the initial file.

2 Algorithm main ideas and specifics

2.1 Functionnality

Now let's talk a bit about a few of its specifics and how the algorithm is working.
Like I said previously, the main idea of this algorithm is to use the frequency of each character in order to generate a new encoding which will give a shorter size to the most frequent character.
But how does it trully work ?

I will not describe the entire algorithm, I will simply introduce it with the following image extracted from Wikipedia :

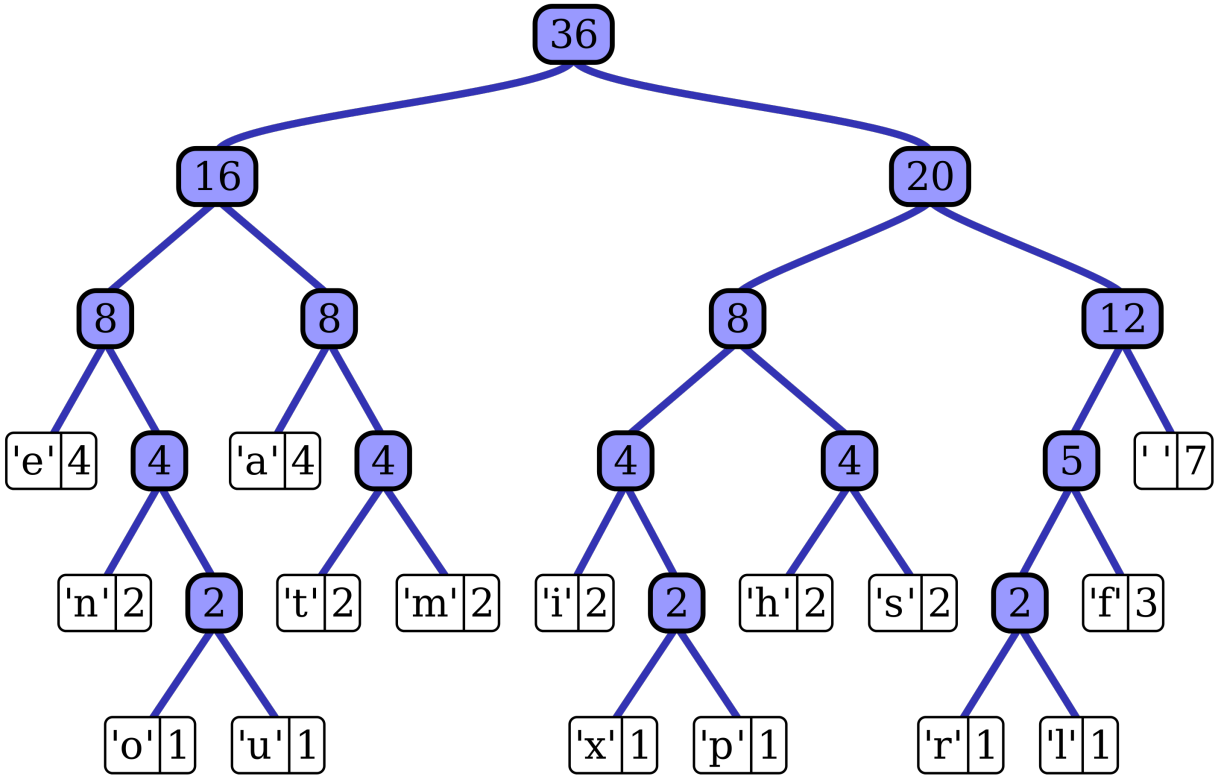

Figure 1: Huffman tree produced from "this is an example of a huffman tree"

So what's happening there ? Here we have a binary which have special leaves.
Those leaves contains two things, a character and a number, which corresponds to the number of occurences of its

associated character.

This tree was build using the frequency of each character, here is the algorithm used :

```
1. We take the two characters appearing the less.
2. We put them in leaves, which we'll make childs of a new node.
3. We compute in this new node the sum of the occurences of its two childs
4. Now we consider this new node has a possible child with the occurences computed previously
5. We repeat this process until there is no more leaves available
```

But, what's the point of having this tree ? I'll show it to you with the following image :

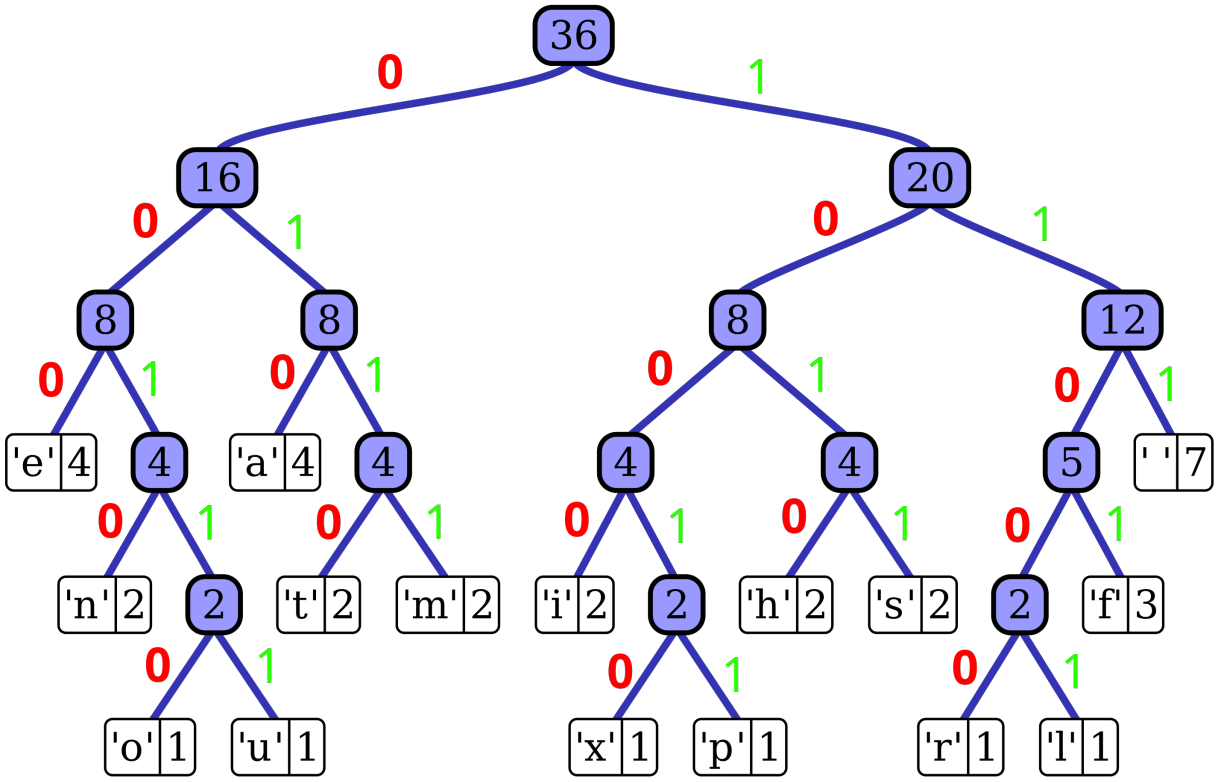

Figure 2: Huffman tree used for the encoding

The next step after building the tree is, generate the encoding.
This is done following a simple principle, the encoding of a character corresponds to its path in the tree. When we have to go through the left child of a node to go to the character leaf, we add a '0' to its encoding, and a '1' if we need to go through the right child.
For example, for the character 'p' we'll have : "10011".
Using this, it is quite easy to generate an encoding for each character, but it also very easy to decode a file.

Let's imagine we're decoding a file contaning these "10011 11000 000 000 1011" (the file will not actually containe these, it will contain the characters encoding by those bits).
In order to decode this, we'll simply have to explore the tree following the direction given by the bit.
For example, for "10011", we'll go right, left, left, right and finally right, and we would end up in the leaf of the character "p". After that we can go back to the root and begin to do that again.

And this is where one property of this encoding is interesting, using this algorithm we generate what we call a "prefix encoding", which is an encoding where each code isn't suffix of any other. This property is fundamental for this algorithm.

2.2 Specific and issues

I have been against several issues, but I will only talk about the most interesting ones.

2.2.1 Padding

Let's imagine we want to encode the following word "press", which is encoded by this "10011 11000 000 000 1011", here I have put spaces in order to make it more clear about which sequence corresponds to which character, but in reality this isn't that simple.
In our file, we're only writting characters, sequence of bytes, so our algorithm would rather write it this way "10011110 00000000 1011". And there is the issue, we can't write half a byte in a file, so our last sequence needs to be completed, to be larger.
We could simply add 4 zeros in order to have a full byte but we would still have an issue, if in our new encoding there is a character encoding by "0", "00", "000", or "0000", we would decode a character that wasn't in the original file.

I found 2 ways to fix this issue :

- Write in the metadata, how much data there is to read.
  Using this, we can tell the decompresser when to stop while reading the compressed data.
  But, this would be an issue for our MPI version, well not quite an issue, but it would complexify a bit the MPI version.
- Write in the metadata, how much data there in excess.
  Using this, we can tell the decompresser that it only have to decode **n** bits in the last byte.

3 How can we improve it with MPI ?

In this section, we'll talk about hashtables (of my personnal design, because it allowed to add some optimization in the hashtables itselves).

3.1 Occurences counting

The counting of the number of occurences for each character is a part where parallisation can be very interesting.
The current design of my MPI version of this part is the following :

- Each process create a personnal hashtable and a final hashtable
- Each process has a part of the file to read in order to count the occurences of each character, which will be stored in the personnal hashtable
- When its done, the secondaries processes will send to the primary process several informations
  - Number of entries in its personnal table
  - For each entry, the character associated and its number of occurences
- The primary process will then one secondary process at the time, gather those informations, and add it in its final table.
- When it's done, it will also add its personnal table information in its final table.
  Now the primary process has a final table which all the information about the characters in the file.
- And finally, it will broadcast its final table to each secondary process

Here is a diagram of the communication when we're using 4 process :

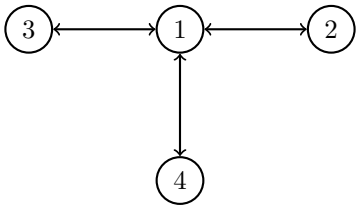

Figure 3: Communication diagram of this part, everything goes through the primary, node "1"

3.2 Compression

The compression part proceed in a different way. A problem of this part is that we have to write the compressed data in the right order, so each process has to wait for the previous to end its writing in order for him to start.
And, we could think that there is no reason to parallelize this part, but there is one job that is done by each process that can be done by all of them during the same time.
Each of them, before writing in the output file will produce a buffer containing the compressed data.
This job include walking through the hashtable several times and some computation. This isn't a truly expensive job so the gain from this parallelization isn't as big as expected but it is still an interesting algorithm.

Due to the problem that I spoke earlier, each process will have to write sequentially and in order.
Also, our process are writing sequence of bits, so when it is done writing, it might not have wrote everything is supposed to.
For example, if the process *i* was about to write a new byte containing the sequence "0010" but has nothing else to put to complete the byte, it will have to transmit what it was going to write ("0010") to the next process **i+1** in order for it to complete the byte with bits that it is supposed to write.
This mechanism create an one-to-one communication between each process **i** to a process **i+1**.
Here is a diagram of the communication when we're using 4 process :

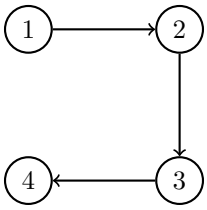

Figure 4: Communication diagram of this part, everything goes through the primary, node "1"

3.3 Decompression, what's the problem ?

The decompression have some of the issues of the compression. We can't have several processes writing in the same output file in the same time.

We could try to parallelize the decompression main part, which is reading from the input file and convert sequence of bits into characters.
But one of the issue with this is that, what if a process **i** has reach the end of the input file, but hasn't a complete sequence of bits (one corresponding to one character). This means that the remaining part of the sequence will be in the part of the input file of the next process **i+1**.
So in order for the process **i** to complete its job, it would have to communicate with the process **i+1** in order to have the missing bits. And the process **i+1** will have to do the same with the process **i+2**, etc...

And in order for that to work, a process **i+1** would have to transmit the missing bits to the previous process **i**. But how can it determine how much bits are missing ?
For now, I don't have a solution to this question. If I found one, I would be able to try to parallelize the decompression, but I'm not sure that it would have a huge impact on the performance.

4 Results

It is hard for me to study the performance of my algorithm for 2 reasons :

- It seems like when the file get too big, the program will crash due to a memory allocation too big to be handled.
- Also, I wasn't able to use Grid'5000, launching my MPI version on Grid'5000 crash for an unknown reason.

But I was still able to have a glance to the gains offered by my version.

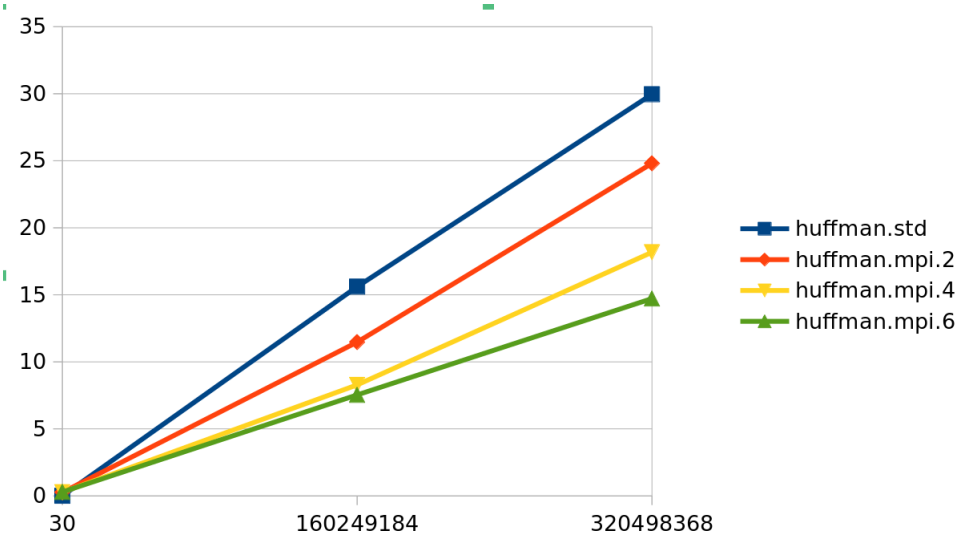

Figure 5: Chart produced using the simple "time" unix tool

Each segment correspond to a different type of execution :

- huffman.std : Execution done by my standard version of the Huffman Algorithm
- huffman.mpi.**n** : Execution done by my MPI version of the Huffman Algorithm, using **n** process

5 Conclusion

It is indeed possible to upgrade a Huffman algorithm using parallelization but the upgrade has to be study well before being implemented, because this algorithm is fundamantally executed sequentially.
In my study I have only done upgrades on the compression part, and for now, I don't have any idea for an upgrade of the decompression part.