High Performance Computing for Data Science Report - Huffman Coding -

Andrea Cristiano
Artificial Intelligence Systems
Trento, Italy
`andrea.cristiano@studenti.unitn.it`
`229370`

Gabriele Padovani
Artificial Intelligence Systems
Trento, Italy
`gabriele.padovani@studenti.unitn.it`
`229207`

I. Introduction

Nowadays, the transmission through networks, as well as the storage of large amounts of digital information has become a fundamental part of human activities. Compression and decompression algorithms play an essential role in the formerly described processes. For many applications these must be lossless, as the information has to be reconstructed the exact way it was before. Other approaches, on the other hand, can be lossy, for example in image compression, where the modification of one pixel does not influence the overall result. Huffman coding[1] is likely one of the simplest approaches for lossless data compression.

Parallelization allows software to reach better performance compared to their sequential version. The idea behind this is simple: a complex problem is divided into pieces that are assigned to dedicated processors. These last solve their part and then return it to the master process, which will reassemble the data and will provide the result. Ideally a program, if given more processing power, should have a linear performance improvement, often referred to as Linear Speedup. In practice however, latencies in messages necessary for data sharing, render the effective gain to a much lower value compared to the ideal one.

On one hand, this project aims at developing an application relying on the Huffman coding algorithm to losslessly compress text. On the other hand, exploitation of the parallel paradigm is required to improve the performance of the program itself. For this purpose, MPI[2] library and OpenMP[3] API are used. The application is entirely written in C and the source code is available on GitHub[4].

This document is organized as follows: the next section will briefly introduce the Huffman coding algorithm with all its main characteristics. Following, it will describe the approaches used to parallelize respectively the compression and the decompression phases. Section 3 will introduce the results obtained, by focusing on specific parts of the algorithm, as well as by giving a general comparison between different methods. Section 4 will contain a summary of the difficulties and fundamental issues encountered during the development of this project, and how these were dealt with.

II. Huffman Coding

Huffman coding is one of the most used lossless compression algorithms. It can be used to reduce the size of many kinds of digital information: text, image, video, audio, etc. Its initial version was developed by David A. Huffman, who published his results for the first time in 1952[5]. The main aspect which made the algorithm so popular is its higher compression efficiency, achieved thanks to the variable-length mechanism used to encode each character. The characters with high frequency will receive a short binary code, while less frequent ones will be encoded with a longer sequence.

The following two sections are mainly focused on the specific case of text, but can easily be generalized to other kinds of digital information formats.

A. Compression

The process of compression consists in constructing the so-called Huffman tree and comparing each character against it to encode the text. Fig.1 and Table 1 show an example of the resulting tree and binary code obtained from the compression of the "Hello world" phrase.

Several ways to implement both the compression and decompression algorithm have been proposed over the years. The procedure which was developed in this project consists of the following steps:

1) The source text is equally split between all available processes. Each of them creates a dictionary of tuples `<character, frequency>` by counting the occurrences of each character;
2) All processes send their dictionary to the master, in this case the one with id zero;
3) The master merges its dictionary with all others, obtaining a complete collection;

4) This list is then sorted in ascending order based on frequency, by using the Odd-Even sort algorithm[6];
5) Once the dictionary is sorted, it is used to create the Huffman tree and from this the encoding dictionary;
6) The latter is then shared with all other processes and used to encode text. Every process also creates an array containing the specific dimension of each block[1], which will compose the final encoded text;
7) To conclude, every encoded text and dimension array is sent to the master process, which merges the texts and concatenates the dimension arrays;

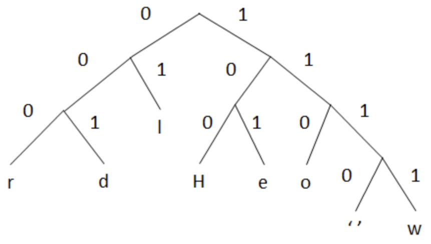

Fig. 1. Hello world Huffman tree representation

100	101	01	01	110	1110	1111	110	000 01	001
H	e	l	l	o	w	o	r	l	d

TABLE I
HELLO WORLD BINARY ENCODING

B. Decompression

A necessary feature of a compression algorithm is the capability to maintain decodability, regardless of the number of processes used for encoding and decoding. Using only the Huffman tree, the algorithm must be able to reconstruct the original text.

The compression process consists in one pivotal step: the tree is traversed, looking for the leaf node corresponding to a specific character. At each step during the descent, a bit is saved depending on whether the child being searched is the left (bit is zero) or right (bit is one). Once the process reaches the correct leaf, it will also have retrieved the correct bitwise encoding for the character.

However, the variable-length of each encoded symbol makes it difficult to parallelize the algorithm. That's because it is not possible to know a priori where a character ends and where the next one starts. In order to deal with this, during the encoding phase the text has been divided into a fixed number of character blocks, while an array has the task to save the length ni bits of each block.

[1]Blocks are used to guarantee correctness of the decoding operation, even when the number of processes used for compression differs from the one used for decompression.

This approach reduces the compression rate of the text, because in the worst case every block will end with 7 useless bits. However, the encoded phase can easily be parallelized independently of the number of processes used to encode the text.

The procedure developed for the decompression section consists of the following steps:

1) Each process reads the encoded file, extracts the Huffman tree and the dimensions array;
2) The number of blocks that has to be decoded and their dimensions are computed by each process;
3) With all these information the actual decoding phase begins;
4) Finally, the decoded texts are sent to the master process, which joins them in a single string.

III. RESULTS

In the following three sections, the results achieved will be reported and explained. These consist in the calculation of speedup and efficiency factors, also going more in depth into which subsections take more time or does not scale well with the addition of more processes. Several block sizes are also tested, to see which favors better compression.

A. Compression Ratio

The first metric to rate our algorithm with is the compression ratio. It was tested with several block sizes, all shown in Fig. 2, yielding results ranging from 20% to 40%.

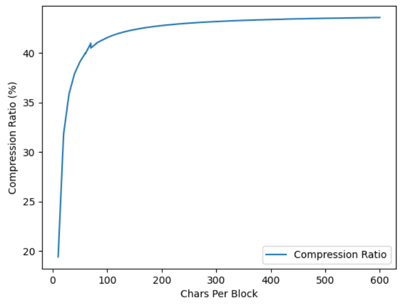

Fig. 2. Compression ratio with regards to the number of characters per block

Overall, a compromise has to be found between algorithm speed, as a higher number of blocks favors better subdivision of work, and compression capability, as again, a lower number of blocks yields less padding and a smaller file header.

B. Compression Performance

For the compression stage, the execution time for the following stages was recorded:

1) Read File: for each process only the sub-portion relevant was read;
2) Calculating the frequencies for each character, this is done independently by each process;

3) Merge Char Frequencies: each frequency dictionary from the several processes are joined into one;
4) Sorting operation of the frequency dictionary;
5) Getting the encoded byte buffer from tree, only the portion relevant to each respective process is calculated;
6) Merging each encoded byte buffer into a single one;
7) Writing the encoded buffer to a text file.

The recorded timings for the main stages are shown in Fig. 3 on the left, while the total time for the whole encoding process is presented on the right. What can be taken from the former image, is that some stages scale well with an increase in the number of processes, such as the frequency calculation and encoding phases. On the other hand, merging stages rightfully take more time as computing capability is scaled.

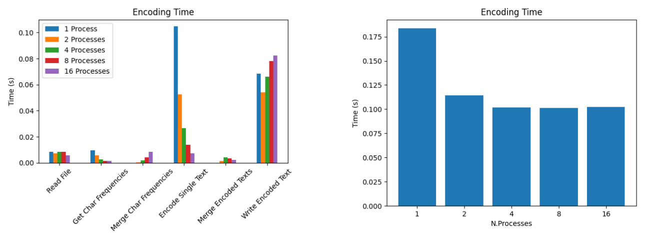

Fig. 3. Left: encoding time for each main section. Right: Total encoding time for the corresponding number of processes.

Although for some operations an increase in processes is not beneficial, the total execution time still yields more competitive results with better parallelization, as shown in Fig. 4.

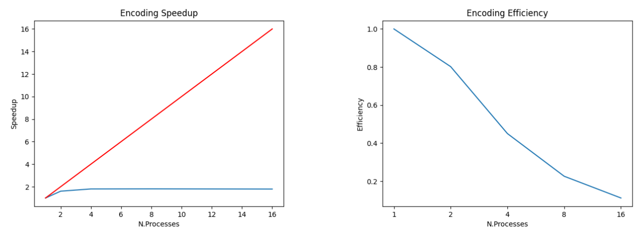

Fig. 4. Left: encoding speedup compared to linear one. Right: Efficiency of encoding algorithm with regards to the number of processes.

The calculated speedup is however far from linear. This is due mostly to the write operation. If the execution time for this phase is omitted, in fact, the speedup factor jumps to a respectable level, as shown in Fig. 5.

C. Decompression Performance

For the decoding stage, execution times were registered for:

1) Parse the file header;
2) Parse the Huffman tree for decoding;
3) Parse block length information, just the one relevant to the current process;

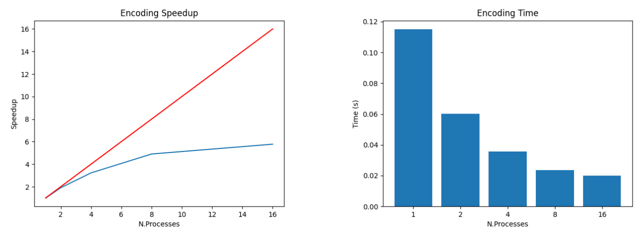

Fig. 5. Left: encoding speedup obtained by removing the costly write operation. Right: Total encoding time for the corresponding number of processes, removing the write phase.

4) Calculate the range of bits which will need to be parsed by the current process;
5) Decode the encoded bits to text;
6) Merge the decoded portions from each process.

In this case, it seems that with the addition of more and more processes, the execution time reaches a minimum value, and then just increases. This is due to the final merging operation, where the master has to wait for each process to finish before joining the string buffers.

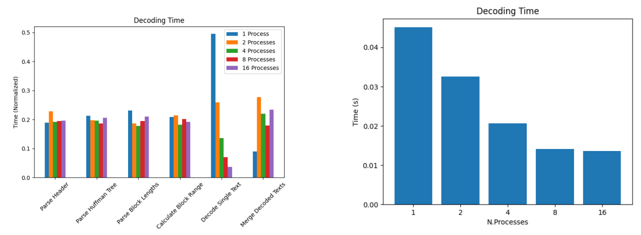

Fig. 6. Left: decoding time for each main section. Right: Total decoding time for the corresponding number of processes.

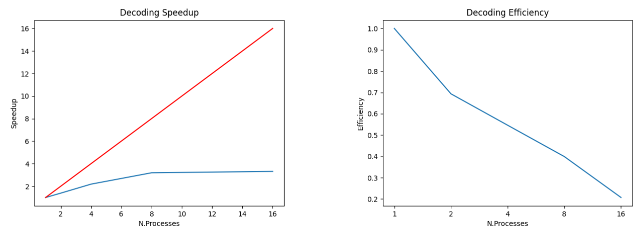

Fig. 7. Left: decoding speedup obtained by removing the costly merge operation. Right: Efficiency of decoding algorithm with the same settings.

D. Multi-Threading for Compression

The OMP library was used to test multi-threading speedup for some operations executed during the compression phase. These include the frequency calculation of characters in the text and the sorting of the frequency dictionary.

Interestingly, the sorting operation seems to be slower with each increase in number of threads. The time taken by this operation, however, is so small that the overhead in the thread creation might be enough to balance out the benefits. It is also important to point out that the increment in time for this phase is not substantial. All the results are shown in Fig. 8.

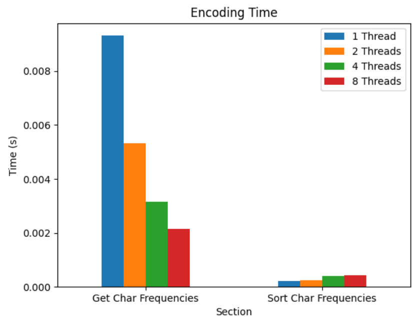

Fig. 8. Encoding time for operations which benefit from multi-threading

IV. Progress and Difficulties

Several problems have been encountered during the development of this project. In the following sections the most relevant ones are reported and explained.

A. Decoding Synchronization

One of the main problems in Huffman coding parallelization is that, since the encoded file contains bitwise information in the form of variable length characters, each process does not know where one character starts and another one ends. To be able to distinguish this boundary, several approaches have been developed. The two main techniques consist in:

1) Adding padding at the end of blocks, marking where the characters end and the filler information starts;
2) Using special sync characters, which are also encoded, to mark the end of a processor's job [7] [8].

The implementation presented in this report is the former, since it was considered simpler and did not involve increasing the size of all encodings by adding special characters. This technique had several drawbacks as well: it required the use of a header to store information about each block's starting point and it is considered to have worst compression ratio on average.

B. Storing the Tree

Another relevant design decision is the modality with which to store the Huffman tree. To be stored in a file, the tree has to be encoded as an array of bytes. The modality with which this was achieved is to allocate, for each node, space for the contained value, as well as a byte indicating whether the next item is the left or the right child.

As an example, the tree containing the "Hello world" encoding, shown in Fig.1 has been translated to array form in Table 2.

Another important decision regarding the tree is whether to store it in the same file as the encoding, or to keep it in a dedicated one. Keeping separate storage would mean

\$11	\$11	\$11	r00	d00	l00	\$11	\$11
H00	e00	\$11	o00	\$11	...	...	w00

TABLE II
Hello world Huffman tree in array form

significantly reducing the complexity of the reading operation, as well as reducing the amount of padding required around the tree. The chosen solution, however, was to keep it in the same file, as using an additional file would result in an increase in complexity for the user and in a more difficult estimation of the real compression ratio.

C. Sharing information

Another issue with the developed algorithm revolves around the quantity of information which has to be shared among the processes. For the encoding section, each process has to wait for the merging operation of the character frequencies, the sorting of this dictionary, the creation of the Huffman tree and finally the collection of each encoding in an array. Only now the encoding list can be shared back with all processes and can be used to encode each respective subsection.

The more costly operation however, as shown in the previous graphs, is the writing of the encoded text into a single file.

The decoding section, on the other hand, allows for more operations to be parallelized, although at the cost of a very expensive final merge operation. Another issue for the latter section is that all the parallel operations, including parsing necessary information and calculating block ranges, are very quick phases compared to the time necessary for the merge completion.

V. Conclusions

To conclude, the chosen method, although being much simpler compared to several ones found in literature, still achieves competitive results regarding the compression ratio for text files. As reference, the *Divina Commedia*, used as test sample for most of the conducted experiments, was compressed from a starting size of 543kB to a final size of 348kB.

Where our approach encounters difficulties is with the scalability factor, as the large amount of communication required and the small number of operations which benefit from multi-processing, yield a very small improvement with the addition of more computing power.

Another problem encountered is the fact that most of the scalable operations are very quick already compared to the ones which do not scale well. An example of this is the merging of character frequencies during the encoding phase. In this case, two operations which scale well with more computing power are balanced out by a single poorly scalable one.

References

[1] Alistair Moffat. "Huffman Coding". In: *ACM Comput. Surv.* 52.4 (Aug. 2019). ISSN: 0360-0300. DOI: 10.1145/3342555. URL: https://doi.org/10.1145/3342555.

[2] *MPI Site*. URL: https://www.open-mpi.org/doc/current/.

[3] *OpenMP Site*. URL: https://www.openmp.org/.

[4] *HPC4DS Project Repository*. URL: https://github.com/lelepado01/HPC4DS-Project.

[5] *Huffman Coding*. URL: https://en.wikipedia.org/wiki/Huffman_coding.

[6] Peter S. Pacheco. "Chapter 3 - Distributed-Memory Programming with MPI". In: *An Introduction to Parallel Programming*. Ed. by Peter S. Pacheco. Boston: Morgan Kaufmann, 2011, pp. 83–149. ISBN: 978-0-12-374260-5. DOI: https://doi.org/10.1016/B978-0-12-374260-5.00003-8. URL: https://www.sciencedirect.com/science/article/pii/B9780123742605000038.

[7] Gil. Goldshlager. "6.338 Final Paper: Parallel Huffman Encoding and Move to Front Encoding in Julia." In: (2015).

[8] S.T. Klein and Y. Wiseman. "Parallel Huffman decoding". In: (2000), pp. 383–392. DOI: 10.1109/DCC.2000.838178.