Parallel Huffman Compression

Munkhdelger Bayanjargal *

July 14, 2024†

1 Introduction

Data compression techniques, such as Huffman coding, play a pivotal role in optimizing resource utilization, minimizing storage requirements, and enhancing data transmission efficiency across various domains.

Huffman coding, devised by David A. Huffman in 1952 [1], is a fundamental algorithm used for lossless data compression. It operates by assigning variable-length codes to input characters based on their frequencies. Characters with higher frequencies are assigned shorter codes, resulting in efficient data compression. This approach makes Huffman coding particularly useful in scenarios where certain characters occur more frequently than others, such as text data or images. The main stages of Huffman compression are the following:

1. Frequency calculation
2. Huffman tree construction
3. Huffman code generation
4. Data encoding

The process of implementing Huffman coding sequentially involves several key stages. It begins with constructing a frequency table by scanning the input data to determine the frequency of each character. Following this, a Huffman tree is built from the frequency table. The Huffman tree construction ensures that the variable-length codes assigned to the characters are prefix codes, meaning no code is a prefix of another. This prevents ambiguity during decoding and maintains the integrity of the compressed data. Once the Huffman tree is constructed, the codes for each character can be derived by traversing the tree from the root. This traversal involves maintaining an auxiliary array, where a `0` is written when moving to the left child and a `1` is written when moving to the right child. The array is printed when a leaf node is encountered, generating the Huffman codes for each character. Finally, each element of the input stream is replaced by the corresponding Huffman binary codeword, completing the encoding process.

Algorithm 1 Sequential Huffman Coding Pseudocode

Input: Text data T
Output: Huffman tree H and encoded data E
1: Construct a frequency table F for characters in T
2: Initialize a priority queue Q with nodes corresponding to characters and their frequencies
3: While $|Q| > 1$
   1. Extract two nodes with the lowest frequencies n_1 and n_2 from Q
   2. Create a new node n with frequency equal to the sum of frequencies of n_1 and n_2
   3. Set n as the parent of n_1 and n_2
   4. Insert n into Q
4: Assign Huffman codes to characters based on tree H
5: Encode the input data T using Huffman codes
6: **return** Huffman tree H and encoded data E

The time complexity of the sequential Huffman coding algorithm (Algorithm 1) is $O(n \log n)$, where n is the number of input characters. This indicates that the computation time for each phase is highly correlated with the number of input characters. The larger the input size, the slower the encoding since the frequency counting and data encoding part must scan the whole input sequence as illustrated in Figure 1.

Parallelizing Huffman coding leads to significant improvements in performance and scalability. Distributing computation across multiple processors allows parallel algorithms to process larger datasets more efficiently, reducing overall processing time and enabling faster data compression and decompression.

However, parallelizing Huffman coding poses several challenges, including communication overhead, load balancing, and managing data dependencies. Addressing these challenges is crucial for developing

*m.bayanjargal@studenti.unitn.it
†Repository:
https://github.com/Munkh99/Parallel-Huffman-HPC

efficient parallel algorithms utilizing available computing resources. For this project, we researched possibilities to improve the current parallel Huffman algorithm regarding efficiency and compression ratios. The main objectives of this project are:

- Propose parallel Huffman coding algorithm using MPI in C.
- Investigate different parallelization strategies for Huffman coding.
- Evaluate the performance and scalability of the parallel implementation compared to the sequential version.

2 Parallel design

Design strategies for parallel Huffman compression have evolved significantly, focusing on optimizing computational efficiency and scalability. The primary challenge of the Huffman compression algorithm lies in variable-length codewords. Traditional Huffman coding assigns code lengths based on symbol probabilities. This presents a significant obstacle when merging encoded data streams from parallel processing units, as it distupts byte alignment in compressed output.

Despite the complexities of parallel implementation, researchers have explored various approaches to overcome these issues from different aspects. Berman et al. [2] proposed approximation algorithms for constructing Huffman codes, achieving significant speedup. Noel [3] implemented a method where the dataset is split into chunks and processed by individual processes. Another innovative approach, suggested by Rahmani et al. [4], involves CUDA architecture: after serially constructing the Huffman tree,

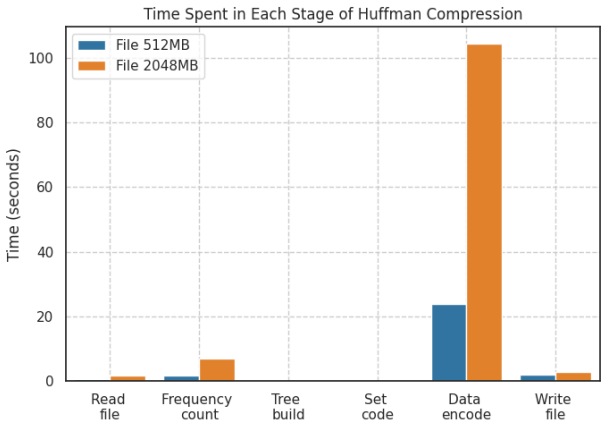

Figure 1: Time spent

Algorithm 2 Parallel Huffman Compression steps

1: File read .....(Serial in Host)
2: Data distribution .....(Parallel)
3: Frequency calculation .....(Parallel)
4: Huffman Tree construction .....(Serial in Host)
5: Huffman code generation .....(Serial in Host)
6: Data encoding .....(Parallel)
7: Encoded data gathering .....(Parallel)
8: File write .....(Serial in Host)

each thread encodes data into a byte stream, with each byte representing a single bit of the compressed output. Consecutively, these bytes are combined in parallel to generate the final stream.

Building upon previous research, our contribution focuses on leveraging CPU architectures to parallelize the stages of the Huffman encoding algorithm to achieve significant performance gains.

The main stages of our parallel algorithm are outlined in Algorithm 2. It begins with reading the text file in the root process. Since our optimization targets the parallel Huffman compression, we primarily emphasize optimizing algorithmic processes rather than parallelizing input/output operations. Then, the input sequence is partitioned and distributed among `n` processes. In the third step, each subset of data computes local frequencies within each process, which are then aggregated at the root process to determine the global frequency. The next two steps are processed serially on the root process since, as shown in Figure 1, runtime insights indicate that these stages consume the least time, nearly zero. The generated Huffman codes are subsequently distributed to other processes. In the sixth step, each process encodes its local subset using the Huffman codes. We adopt a byte-wise representation of the encoded binary sequence to optimize memory allocation rather than storing each bit as a char. For example, the sequence `101010111` is compacted into a single byte instead of eight bytes. Any remaining bits required for byte alignment are padded with zeros, which aids in decoding the compressed file.

Finally, the root process aggregates the encoded byte streams and writes them to a file. Following standard compression practices, metadata such as the frequency table, lengths of encoded sequences from each process, and the number of padding bits appended to each sequence are stored. This metadata is crucial for accurately decoding the compressed data.

3 Implementation

This section details the implementation of our parallel Huffman compression algorithm using MPI. The implementation focuses on efficiently distributing the workload among multiple processes to achieve significant performance gains over the sequential version.

Initialization and Data distribution. The program starts by initializing the `MPI` environment and determining the rank and size. The rank indicates the process's ID, while the size denotes the total number of processes. The root process (rank 0) initially reads the input data file. Once the data is read, its size is determined, and the root process proceeds to partition it among all processes. It's worth noting that most collective communication and point-to-point functions have an integer maximum limit for the amount of data to be sent. Therefore, when the data size exceeds the integer maximum, we consider the multiple data transfers to the same process, each with a size smaller than the integer maximum. To handle these transfers, we employed non-blocking functions, such as `MPI_Isend` and `MPI_Irecv`, to ensure the root process handles multiple operations concurrently.

The input dataset is split into equal-sized partitions (Algorithm 3), considering the load balancing, and sent to the corresponding processes. By broadcasting the original data length to other processes, partition size and offset arrays are computed in all processes. It ensures that each process knows the designated partition, as depicted in the top section of Figure 2. Each process allocates memory for receiving data and retrieves the data into a local data buffer.

Algorithm 3 Partition computation

1: $num\_partitions \leftarrow size$, $num\_send \leftarrow 1$
2: **if** $length/size > INT\_MAX$ **then**
3: $\quad num\_send \leftarrow \lceil (length/size)/INT\_MAX \rceil$
4: $\quad num\_partitions \leftarrow size \times num\_send$
5: **end if**
6: Calculate $partitionSizes$ and $partitionOffsets$

Description: Initially, the data is divided into as many subsets as there are processes. If the size of any subset exceeds the maximum integer value, those subsets are further divided into smaller subsets, each containing at most the maximum integer size.

Frequency Calculation. Each process individually calculates the frequency of characters in its local data segment. These local frequency counts represent the occurrences of each character within the portion of data assigned to that process. After calculating these local frequencies, they are then aggregated at the root process using `MPI_Reduce` operation. This aggregation combines the local frequency counts from all processes into a single global frequency table, encapsulating the cumulative character frequencies observed across all distributed data segments.

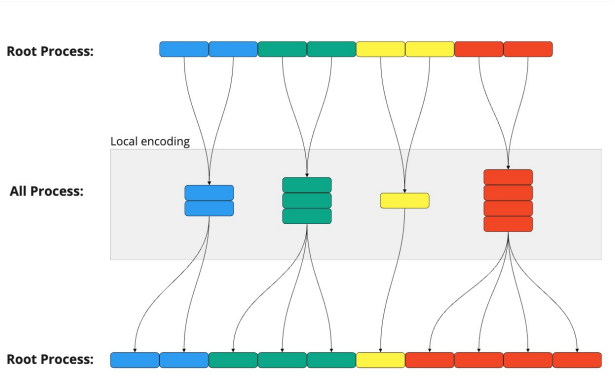

Figure 2: Data Distribution. This figure illustrates the data flow from the root to other processes and from other processes back to the root process using MPI operations. Equally partitioned subsets are distributed to other processes for data encoding. After the encoding, the locally encoded sequence is divided into partitions and sent to the root process. In the root process, received partitions are written in corresponding addresses. The different colors represent processes, and the same colors represent continuous data segments.

Huffman Tree Construction and Code Generation. The root process initiates the construction of the Huffman coding tree using the serial Huffman algorithm. Once constructed, the Huffman tree is serialized into a compact representation for efficient transmission using the `MPI_Bcast` function. After receiving the serialized tree, each process deserializes it to reconstruct the tree and generates a code dictionary. The dictionary maps each character to its corresponding Huffman code, derived from the tree's structure and traversal.

Data Encoding. The encoding process begins with the allocation of memory for the encoded data. The allocation size is set to the size of the local data buffer to ensure enough space to store the encoded data. A lookup table maps characters to their corresponding Huffman codes for quick access during encoding. The function iterates over each character in the local data buffer. For each character, it retrieves the corresponding Huffman code from the lookup table and appends its bits to `current_byte` until it gets 8 bits. Once `current_byte` is full, it is appended to the `encoded_data` array, and `current_byte` is reset to accumulate the next set of bits. This process con-

tinues until all data is encoded. The last byte of the encoded sequence may contain either entirely valid bits or a mix of valid bits and padding bits. This approach significantly reduces the memory required to store the encoded data by packing bits into bytes.

Aggregation and Output. Following encoding, each process contributes its encoded data and padding bit information to the root process. The lower section of Figure 2 illustrates encoded data gathering at the root process. The root process collects the size of encoded data from all processes and allocates sufficient memory for the aggregated data. It then computes potential data splits from each process to determine partition size and offsets, which are crucial for accurately writing data to the correct memory addresses. Simultaneously, each process calculates necessary splits and establishes local partitions and offsets. Utilizing non-blocking communication with `MPI_Isend` and `MPI_Irecv` functions ensures efficient aggregation of partitions into the designated memory locations. `MPI_Waitall` is used to wait for the completion of multiple non-blocking communication requests, ensuring synchronization across all processes. The aggregated data, including encoded content and accompanying metadata such as frequency tables, padding bit counts, and total byte counts per process, is subsequently saved to a file.

3.1 Data dependency

The Huffman coding algorithm presents significant challenges for parallelization due to its inherent sequential nature. In our approach, we mitigate this issue by exploiting equal-sized data partitioning to maintain load balancing and task concurrency while ensuring correct data transfer during communication. Despite dividing our parallel version into stages, sequential dependency remains.

The root process uses non-blocking communications to handle multiple send and receive operations concurrently, ensuring efficient and balanced distribution across processes. This is an example of task-level dependency where the data distribution must be completed before local computations begin.

During the frequency calculation stage, each process independently calculates the frequency of characters in its local data, which is a non-loop dependency. However, aggregating these local frequency counts at the root process using the `MPI_Reduce` operation introduces a synchronization point, creating a dependency across processes due to its blocking nature.

The construction of the Huffman tree depends on the availability of the global frequency table at the root process, representing a task dependency. Once constructed, the serialized tree is shared using `MPI_Bcast`, ensuring all processes receive the tree needed to encode the data, thus synchronizing the dependency on the tree.

Each process encodes its local data based on the Huffman codes. This is another example of a non-loop dependency. Here, the encoding of each character in the local data buffer can be performed independently by each process.

After encoding, each process sends its encoded data to the root process. This step involves a task-level dependency, where the root process collects the encoded data from all processes. Using non-blocking communication, the root process efficiently gathers the data, ensuring that all processes complete their encoding before the final aggregation.

Our parallel Huffman coding algorithm ensures correct synchronization and efficient workload distribution by effectively managing both loop-level and task-level data dependencies at each stage.

4 Performance and scalability analysis

The performance and scalability of the proposed parallel Huffman compression algorithm were evaluated through a series of experiments across different configurations. The datasets used in the experiment were synthetic, and we generated datasets of various sizes using the 10MB text file. This section analyzes the results obtained from these experiments, including runtime performance, speedup, efficiency, scalability, and compression ratio.

comm_sz	Data Size				
	1024	2048	4096	8192	16384
1	50.23	86.94	170.13*	338.26*	704.81*
2	26.15	45.24	87.21	201.03*	341.87*
4	14.18	25.31	44.59	94.95	188.63*
8	8.19	15.35	24.40	55.00	105.58
16	5.21	9.94	14.26	32.06	63.37
32	3.76	7.34	8.81	19.96	40.75
64	3.27	6.48	7.03	16.82	34.51

Table 1: Runtime seconds (without I/O). Asterisks (*) indicate multiple sends due to integer constraints in MPI.

Table 1 shows the runtime in seconds (without I/O)

for different numbers of processes (comm_sz) and dataset sizes (in MB). Asterisks (*) indicate multiple sends due to integer constraints, meaning runtime is influenced by multiple communications. The runtime decreases significantly as the processes increase, indicating effective parallelization. For instance, at the largest dataset size of 16384 MB, the runtime decreases from 704.81 seconds with 1 process to 34.51 seconds with 64 processes. However, the reduction in runtime becomes less significant as the number of processes continues to increase. We suppose this is because of the overhead time in the parallel process, which typically comes from communication.

The speedup measures the relative performance improvement gained by using multiple processes compared to a single process. Table 2 depicts that the speedup generally increases with the number of processes, indicating effective parallelization and scalability. The efficiency in Table 3, measures how efficiently the parallel algorithm utilizes additional processes. Efficiency decreases as the number of processes increases due to overheads and the non-parallelizable portion of the computation, also known as Amdahl's Law [5]. Nevertheless, even at a higher process counts, the efficiency remains relatively high, especially up to 16 processes.

comm_sz	Data Size				
	1024	2048	4096	8192	16384
1	1	1	1*	1*	1*
2	1.92	1.92	1.95	1.75*	2*
4	3.54	3.43	3.82	3.7	3.74*
8	6.13	5.66	6.97	6.4	6.68
16	9.64	8.75	11.93	10.97	11.12
32	13.37	11.84	19.31	17.62	17.3
64	15.36	13.41	24.2	20.91	20.42

Table 2: Speedup

The compression ratio of our Huffman algorithm is 1.872. This means that the compressed data size is approximately 53.5% of the original data size, indicating the effectiveness of the parallel implementation in reducing data size. This ratio highly depends on the content of the data to be compressed.

The speedup (blue line in 3) closely follows the ideal trend (yellow line in Figure 3) up to 32 processes, indicating good scalability. Beyond that, the speedup may flatten due to increased overheads or limitations in parallel efficiency.

comm_sz	Data Size				
	1024	2048	4096	8192	16384
1	1	1	1*	1*	1*
2	0.96	0.96	0.98	0.87*	1*
4	0.89	0.86	0.95	0.93	0.93*
8	0.77	0.71	0.87	0.8	0.83
16	0.6	0.55	0.75	0.69	0.7
32	0.42	0.37	0.6	0.55	0.54
64	0.24	0.21	0.38	0.33	0.32

Table 3: Efficiency

Figure 4 shows scaled speedup for increasing dataset sizes while maintaining a balanced workload across processes. The observed scaled speedup values remain close to the ideal linear trend up to a certain point, indicating that the algorithm scales effectively with dataset size.

5 Final discussion

In this project, we developed and evaluated a parallel Huffman compression algorithm using MPI in C language, aiming to exploit parallelism for improved performance in data compression tasks. Our implementation successfully achieved substantial reductions in runtime across various dataset sizes and numbers of processes, leading to notable speedup and efficiency gains. However, we observed that speedup gains diminished beyond a certain number of processes, aligning with the limitations outlined by Amdahl's Law. Communication overhead also emerged as a critical

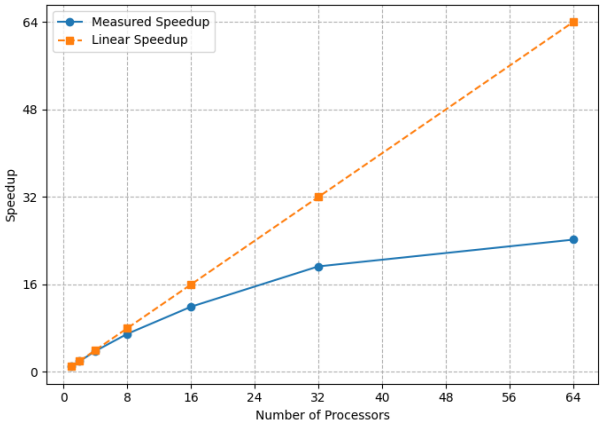

Figure 3: Strong Scalability

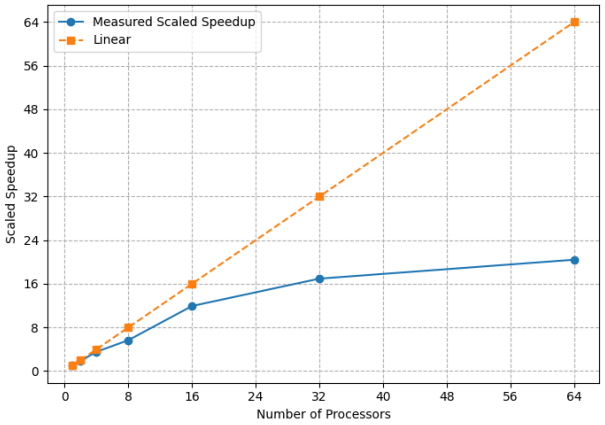

Figure 4: Weak Scalability

factor negatively impacting performance. In addition, the results of strong and weak scaling tests provide reasonable indications for the best match between job size and the amount of resources that should be requested for a particular job. The compressed data occupied about 53.5% of the original size. This underscores the effectiveness of data reduction for storage and transmission.

Looking ahead, future research could explore advanced hybrid parallelization strategies combining MPI with other parallel paradigms, such as CUDA or OpenMP, to mitigate overheads and further enhance scalability. Optimizing I/O operations could also streamline data handling, particularly for larger datasets. These avenues promise to advance the performance and scalability of parallel Huffman compression algorithms, making them more practical and efficient for modern computing applications.

References

[1] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[2] Piotr Berman, Marek Karpinski, and Yakov Nekrich. Approximating huffman codes in parallel. In *Automata, Languages and Programming: 29th International Colloquium, ICALP 2002 Málaga, Spain, July 8–13, 2002 Proceedings 29*, pages 845–855. Springer, 2002.

[3] David Noel, Elizabeth Graham, and Liyuan Liu. Parallel data compression techniques. *arXiv preprint arXiv:2306.11070*, 2023.

[4] Habibelahi Rahmani, Cihan Topal, and Cuneyt Akinlar. A parallel huffman coder on the cuda architecture. In *2014 IEEE Visual Communications and Image Processing Conference*, pages 311–314. IEEE, 2014.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.