Department of Sociology and Social Research

University of Trento, Italy

High Performance Computing for Data Science

A.Y. 2022/2023

Parallel Huffman Encoding and Decoding

Final Report

12-02-2023

Shaker Mahmud Khandaker

shaker.khandaker@studenti.unitn.it

Matricola: 229221

GitHub link : github.com/khandakerrahin/parallel-huffman-coding-MPI

Index

1 Introduction

In today's data-driven world, data compression has become a critical aspect of many applications, ranging from storage and retrieval to data transmission and analysis. Huffman coding is one of the most widely used data compression algorithms, thanks to its simplicity and efficiency. The algorithm is based on the principle of entropy encoding, which assigns shorter codewords to more frequently occurring symbols, and longer codewords to less frequently occurring symbols. To improve the efficiency of Huffman coding, many researchers have proposed parallel implementations that leverage the computing power of multi-core processors and GPUs. In this project, we have implemented a parallel version of Huffman coding, which divides the input data into smaller chunks and processes each chunk in parallel, thus reducing the overall time required for compression and decompression. Our implementation is based on the MPI (Message Passing Interface) library, which provides a standardized and portable interface for parallel programming.

We have evaluated the performance of our parallel Huffman coding implementation on several randomly created text datasets of different sizes, and compared the results with those of existing sequential and parallel implementations. Our results show that our implementation provides substantial speedup compared to the sequential version, and is competitive with other parallel implementations in terms of runtime. Additionally, we have also analyzed the scalability of our implementation, and observed that it scales well with an increasing number of cores, thus demonstrating its suitability for high-performance computing systems.

Overall, our implementation provides a promising solution for efficient data compression, and can serve as a valuable resource for researchers, practitioners, and students who are interested in parallel computing and data compression. The source code and results of our project are available online, and we hope that they will provide a useful reference for further research and development in this area.

2 Background

The field of data compression has long been a crucial area of study and development, given the increasing amount of digital information being generated and stored. One of the most well-known and widely used methods of data compression is Huffman coding. Developed by David A. Huffman in 1952, this technique is a variable-length entropy encoding algorithm that uses a binary tree to represent the frequency of occurrence of each character in a given input data. The most frequently occurring characters are assigned the shortest codewords, while the least frequent characters are assigned the longest codewords. This results in an average codeword length that is much shorter than the fixed-length encoding schemes, which are based on simple binary representations of the characters.

Huffman coding has been found to be particularly effective for compressing text and other sources of data that contain redundant information. It is commonly used in a wide range of applications, including data transmission, data storage, and image and audio compression. Despite its widespread use, however, Huffman coding is a computationally intensive process, which has led to the development of several parallel and distributed algorithms designed to speed up the encoding and decoding process. These algorithms have been found to be highly effective in reducing the time and computational resources required to perform Huffman coding, making it a viable option for even larger and more complex data sets.

3 Design and Implementation

The project was developed using C programming language with MPI bindings. At first, we developed the serial implementation of parallel Huffman encoding and decoding. Once we were done with the sequential coding and decoding, we parallelized the solution. We will discuss each step in detail.

3.1 Implementation of Huffman Encoding

The first step was to implement the sequential Huffman encoding algorithm. This involved creating a frequency table of all the characters in the input text and using it to build the Huffman tree. Each character was then assigned a unique code based on its position in the tree.

Here is the pseudocode for the algorithm:

```
C/C++
START
 Initialize start and end timestamps
  Set distinctCharacterCount to 0

  Open the input file for reading
  Get the length of the input file

  Find frequency for each symbol in inputFileData:
    frequency[symbol]++

  Close the input file

  Loop over the frequency values
    If the frequency is greater than 0
      Set the count, letter, and NULL left and right values of the huffmanTreeNode
      Increment the distinctCharacterCount

  Build the huffman tree
    Loop over the distinctCharacterCount - 1
      Set combinedHuffmanNodes to 2 * i
      Sort the huffman tree
      Build the huffman tree

  Build the huffman dictionary
    Call buildHuffmanDictionary with head_huffmanTreeNode, an empty bit sequence, and a
bit sequence length of 0

  Compress input date
    for each symbol in inputFileData:
      for each bit in huffmanDictionary[symbol].bitSequence:
        writeBit = writeBit << 1 | bit
        bitsFilled++
        if bitsFilled == 8:
          compressedData[compressedFileLength] = writeBit
          bitsFilled = 0
          writeBit = 0
          compressedFileLength++
```

```
        if bitsFilled != 0:
          for i in 0 to 8 - bitsFilled:
            writeBit = writeBit << 1
          compressedData[compressedFileLength] = writeBit
          compressedFileLength++

  Write the compressed data to a file
  End timing and calculate the CPU time used

  PRINT CPU time used
END
```

3.2 Parallelization of Huffman Encoding

To make the algorithm run in parallel, the input text was divided into equal-sized chunks and assigned to different processes. Each process was responsible for calculating the frequency table for its chunk of the input text and building a local Huffman tree. The local trees were then combined to form a global tree, which was used to assign codes to all the characters.

Here is the pseudocode for the algorithm:

```
C/C++
START
    rank = MPI rank of current process
    numProcesses = MPI number of processes

    if rank == 0
        start clock
        open input file
        get length of file
    end

    broadcast file size to all processes

    Get chunk size for each process
    by dividing file size by number of processes

    distinctCharacterCount = 0
    for i = 0 to 255
        if frequency[i] > 0
```

```
            set huffmanTreeNode[distinctCharacterCount].count = frequency[i]
            set huffmanTreeNode[distinctCharacterCount].letter = i
            set huffmanTreeNode[distinctCharacterCount].left = null
            set huffmanTreeNode[distinctCharacterCount].right = null
            increment distinctCharacterCount
        end
    end

    for i = 0 to distinctCharacterCount - 2
        combinedHuffmanNodes = 2 * i
        sortHuffmanTree(i, distinctCharacterCount, combinedHuffmanNodes)
        buildHuffmanTree(i, distinctCharacterCount, combinedHuffmanNodes)
    end

    bitSequence = []
    bitSequenceLength = 0
    buildHuffmanDictionary(head_huffmanTreeNode, bitSequence, bitSequenceLength)

    Compress the data
    Gather compressed data length in an array from all processes

    if rank == 0
        a. Write the compressed data to file
        b. Stop timer and calculate total time
        print rank, total time
end
```

3.3 Implementation of Huffman Decoding

The next step was to implement the sequential Huffman decoding algorithm. This involved reading the encoded text from the file and using the global Huffman tree to decode it. The decoded text was then written to a file.

Here is the pseudocode for the algorithm:

```
C/C++
START
  Initialize start and end timestamps
  Set distinctCharacterCount to 0
```

```
  Open the compressed file for reading
  Get the length of the compressed file
  Read the length of the output file
  Read the number of compressed blocks
  Read the length of each compressed block
  Close the compressed file

  Read the frequency values from the compressed file

  Loop over the frequency values
    If the frequency is greater than 0
      Set the count, letter, and NULL left and right values of the huffmanTreeNode
      Increment the distinctCharacterCount

  Build the huffman tree
    Loop over the distinctCharacterCount - 1
      Set combinedHuffmanNodes to 2 * i
      Sort the huffman tree
      Build the huffman tree

  Build the huffman dictionary
    Call buildHuffmanDictionary with head_huffmanTreeNode, an empty bit sequence, and a
bit sequence length of 0

  Read the compressed data and the output data
  Initialize currentInputBit to 0 and currentInputByte to 0
  Initialize decompBlockLengthCounter to 0

  Loop over the compressed data
    Get the next bit from the currentInputByte
    Set current_huffmanTreeNode to head_huffmanTreeNode

    Loop over the bits in the huffman tree
      If the next bit is 0
        Set current_huffmanTreeNode to the left child
      Else
        Set current_huffmanTreeNode to the right child

    If the current_huffmanTreeNode is a leaf node
      Set the output data at the decompBlockLengthCounter to the letter of the
current_huffmanTreeNode
      Increment the decompBlockLengthCounter

  Write the output data to a file
  End timing and calculate the CPU time used

  PRINT "Total time taken:", CPU time used
END
```

3.4 Parallelization of Huffman Decoding

To parallelize, the encoded text was divided into equal-sized chunks and assigned to different processes. Each process was responsible for decoding its chunk of the encoded text using the global Huffman tree. The decoded chunks were then combined to form the final decoded text.

Here is the pseudocode for the algorithm:

```
C/C++
START
    rank = MPI rank of current process
    numProcesses = MPI number of processes

    if rank == 0
        start clock
        open compressed file
        get length of file
        read outputFileLength, numCompressedBlocks, and compBlockLengthArray from file
        close file
    end

    broadcast outputFileLength and compBlockLengthArray to all processes

    open compressed file for read only access
    seek to compBlockLengthArray[rank]
    read frequency from file
    close file

    distinctCharacterCount = 0
    for i = 0 to 255
        if frequency[i] > 0
            set huffmanTreeNode[distinctCharacterCount].count = frequency[i]
            set huffmanTreeNode[distinctCharacterCount].letter = i
            set huffmanTreeNode[distinctCharacterCount].left = null
            set huffmanTreeNode[distinctCharacterCount].right = null
            increment distinctCharacterCount
        end
    end

    for i = 0 to distinctCharacterCount - 2
        combinedHuffmanNodes = 2 * i
        sortHuffmanTree(i, distinctCharacterCount, combinedHuffmanNodes)
        buildHuffmanTree(i, distinctCharacterCount, combinedHuffmanNodes)
    end
```

```
    bitSequence = []
    bitSequenceLength = 0
    buildHuffmanDictionary(head_huffmanTreeNode, bitSequence, bitSequenceLength)

    compBlockLength = compBlockLengthArray[rank + 1] - compBlockLengthArray[rank] - 1024
    compressedData = allocate array of size compBlockLength
    outputData = allocate array of size outputFileLength / numProcesses
    read compressedData from file
    decompressData(compressedData, compBlockLength, outputData, outputFileLength /
numProcesses)

    if rank == 0
        write outputData to output file
        stop clock
        print rank, total time
end
```

4 Benchmarks

To benchmark the performance of the application, we conducted experiments with varying problem sizes and cores. We evaluated the performance of the parallel implementation by measuring the time taken to encode and decode the input text. The results were compared with the sequential implementation to determine the speedup achieved by the parallel implementation.

Then we performed a comparison with existing implementations of Huffman Encoding by the author Taavi Adamson [3] and author Omar Facchini [4].

4.1 Benchmarks on the cluster

For our benchmark, we prepared 7 text files generated with random texts of 1000, 10000, 10000000, 50000000, 100000000, 200000000, 300000000 words respectively. Then we encoded and decoded the datasets with sequential and parallel of 2, 4, 8, 16, 32, 64, 128 cores. For each configuration, we ran the benchmark for 10 times and took the average of the time taken to run.

The evaluation results of speedups and efficiencies for both the encoder and the decoder can be found in Table 1, Table 2, Table 3 and Table 4. We have also plotted the results of the scalability.

Encoding Speedups		Problem size (Words Count)						
		1000	10000	10000000	50000000	100000000	200000000	300000000
Number of processors	1	1	1	1	1	1	1	1
	2	0.42	0.39	1.24	1.21	1.07	1.14	1.06
	4	0.28	0.15	1.85	1.65	1.64	1.74	1.59
	8	0.11	0.14	2.27	3.47	3.27	3.55	3.54
	16	0.09	0.11	3.91	6.29	5.78	6.48	6.77
	32	0.08	0.08	4.89	8.71	7.41	8.17	8.58
	64	0.08	0.09	5.38	7.90	8.52	9.79	9.09
	128	0.02	0.03	1.99	6.28	7.12	9.15	9.55

Table 01: Encoding Speedups across different problem sizes

Encoding Efficiencies		Problem size (Words Count)						
		1000	10000	10000000	50000000	100000000	200000000	300000000
Number of processors	1	1	1	1	1	1	1	1
	2	0.21	0.20	0.62	0.61	0.54	0.57	0.53
	4	0.07	0.04	0.46	0.41	0.41	0.44	0.40
	8	0.01	0.02	0.28	0.43	0.41	0.44	0.44
	16	0.01	0.01	0.24	0.39	0.36	0.40	0.42
	32	0.00	0.00	0.15	0.27	0.23	0.26	0.27
	64	0.00	0.00	0.08	0.12	0.13	0.15	0.14
	128	0.00	0.00	0.02	0.05	0.06	0.07	0.07

Table 02: Encoding Efficiencies across different problem sizes

Decoding Speedups		Problem size (Words Count)						
		1000	10000	10000000	50000000	100000000	200000000	300000000
Number of processors	1	1	1	1	1	1	1	1
	2	0.33	0.39	1.02	1.05	0.91	1.08	1.03
	4	0.31	0.32	1.44	1.61	1.42	1.54	1.73
	8	0.13	0.15	1.65	2.62	2.63	3.22	3.36
	16	0.08	0.10	2.51	4.77	4.27	5.35	5.94
	32	0.06	0.06	3.10	5.87	4.92	6.76	6.69
	64	0.06	0.09	2.41	5.75	7.18	8.81	7.84
	128	0.01	0.03	2.01	5.65	4.41	7.07	7.38

Table 03: Decoding Speedups across different problem sizes

Decoding Efficiencies		Problem size (Words Count)						
		1000	10000	10000000	50000000	100000000	200000000	300000000
Number of processors	1	1	1	1	1	1	1	1
	2	0.16	0.19	0.51	0.53	0.46	0.54	0.51
	4	0.08	0.08	0.36	0.40	0.36	0.38	0.43
	8	0.02	0.02	0.21	0.33	0.33	0.40	0.42
	16	0.01	0.01	0.16	0.30	0.27	0.33	0.37
	32	0.00	0.00	0.10	0.18	0.15	0.21	0.21
	64	0.00	0.00	0.04	0.09	0.11	0.14	0.12
	128	0.00	0.00	0.02	0.04	0.03	0.06	0.06

Table 04: Decoding Efficiencies across different problem sizes

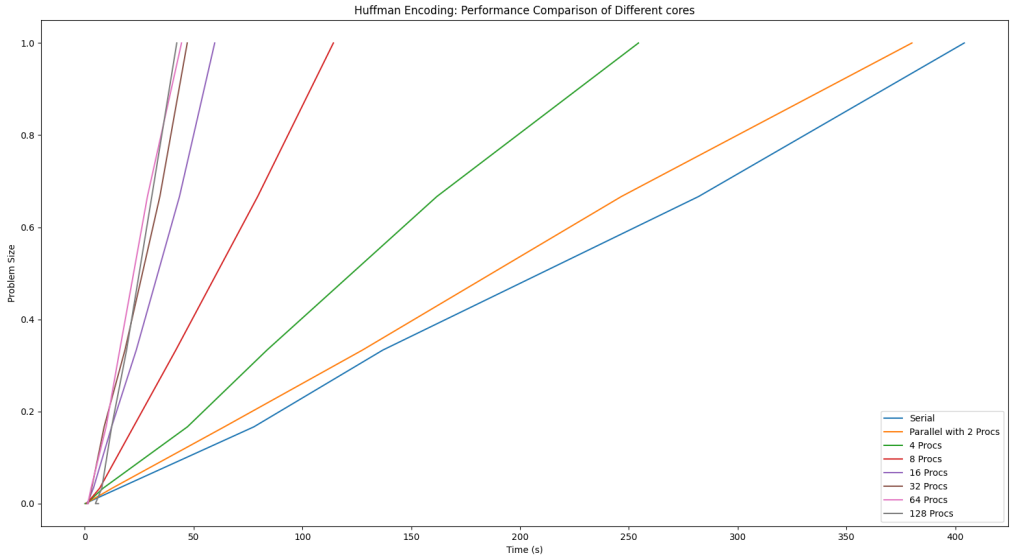


Figure 01: Huffman Encoding performance comparison of different cores

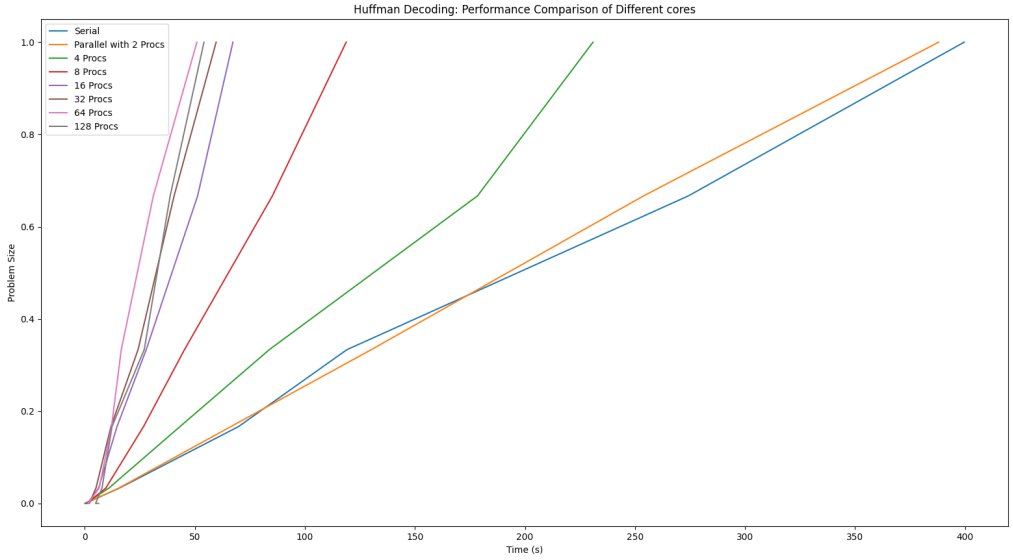


Figure 02: Huffman Decoding performance comparison of different cores

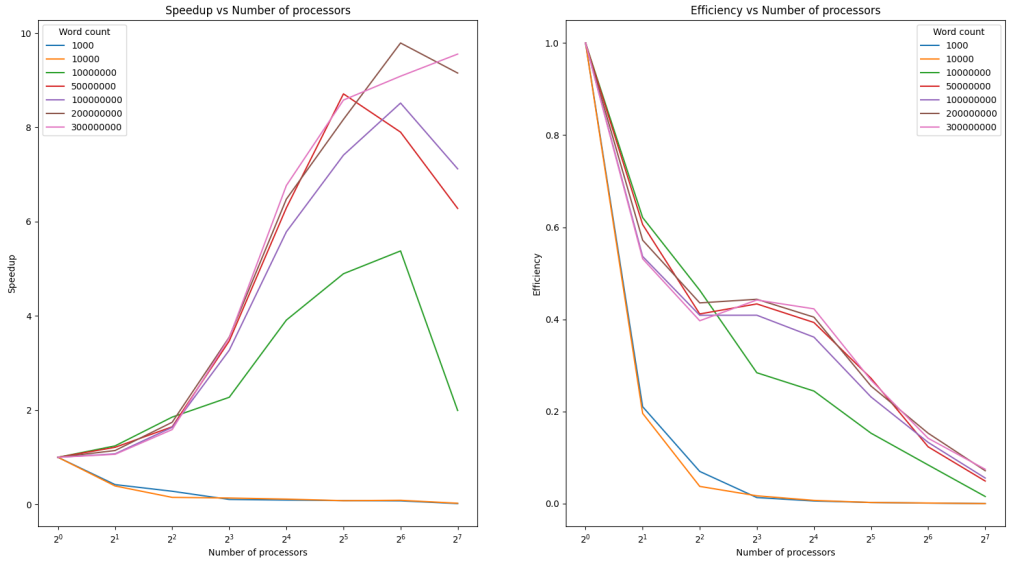


Figure 03: Encoding Speedup and Efficiency

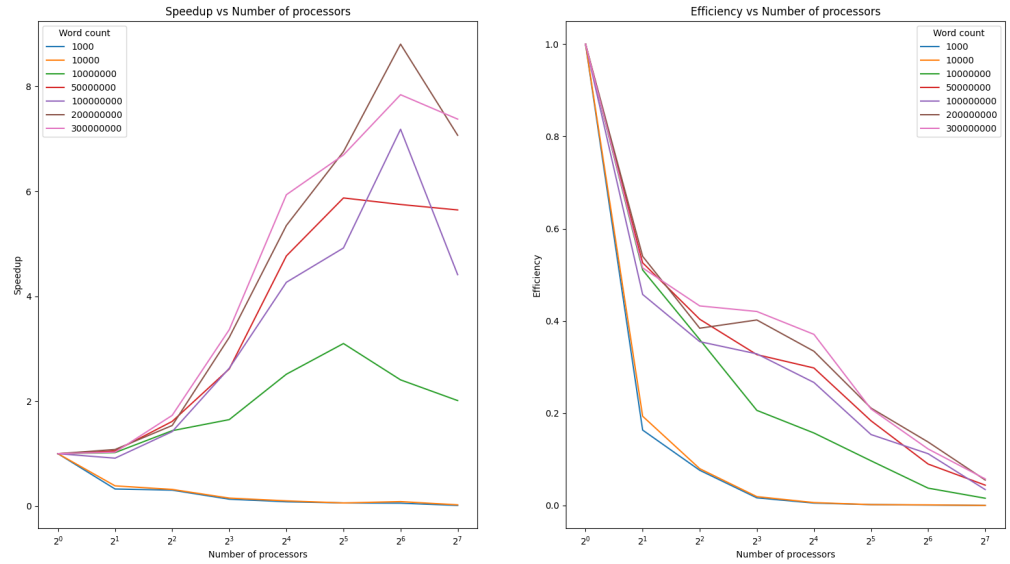


Figure 04: Decoding Speedup and Efficiency

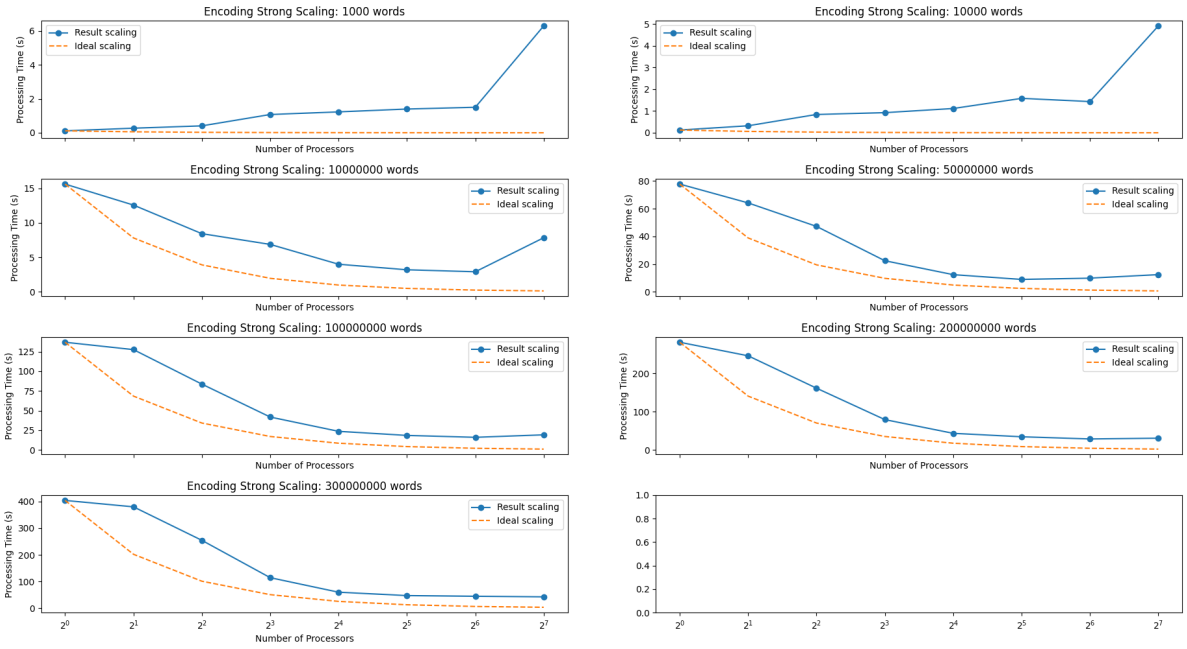


Figure 05: Encoding Strong Scaling for different problem sizes

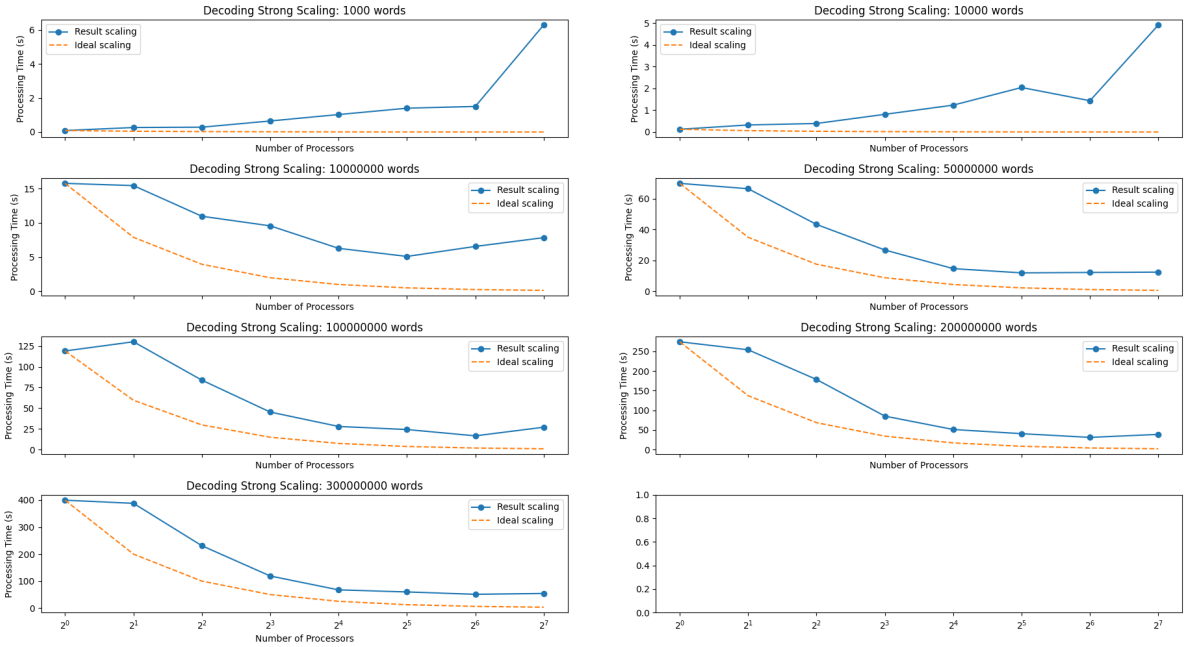


Figure 06: Decoding Strong Scaling for different problem sizes

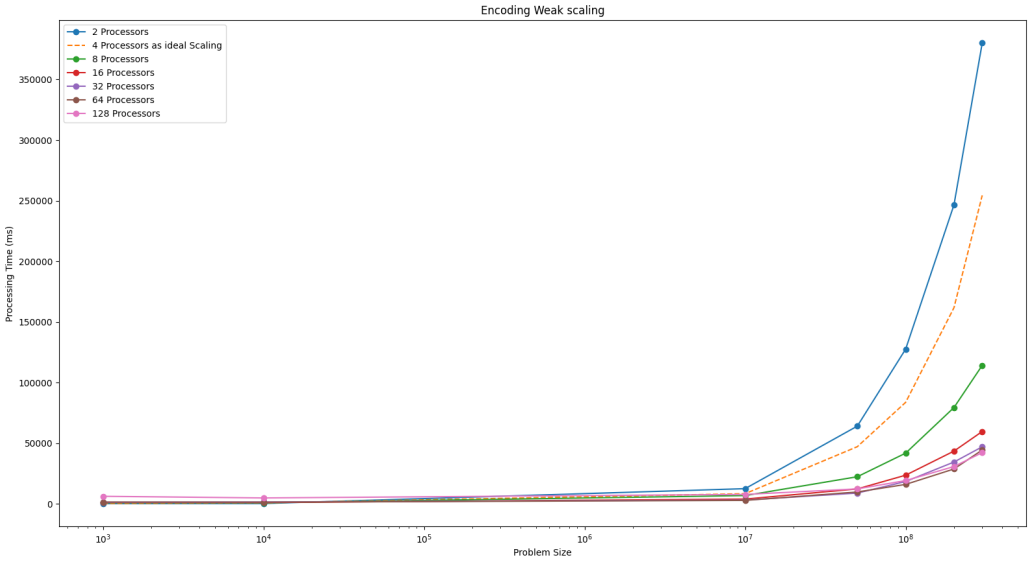


Figure 07: Encoding Weak Scaling

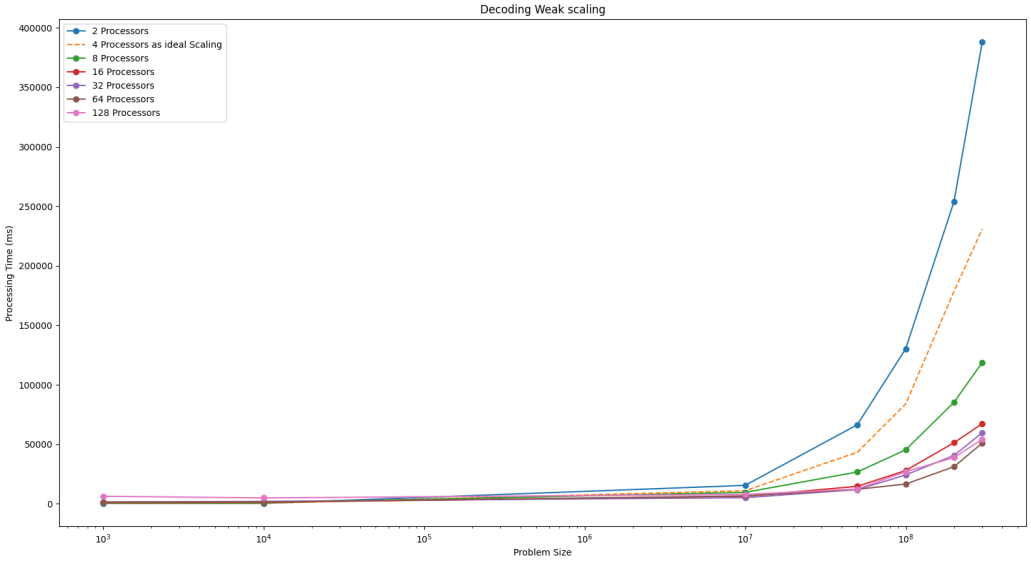
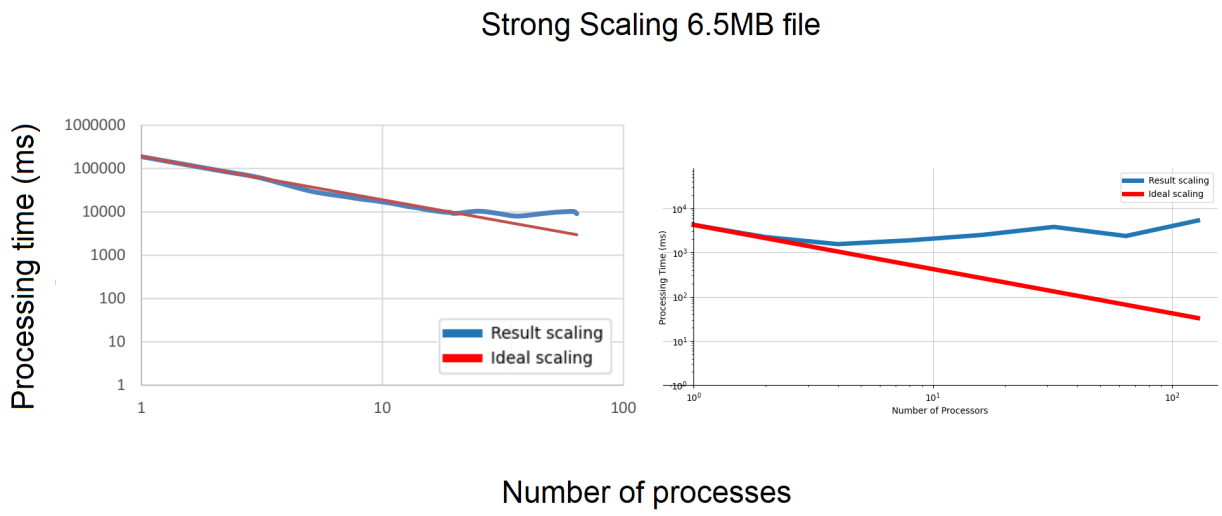


Figure 08: Decoding Weak Scaling

4.1 Benchmarks comparison with existing work

For our first comparison, we are considering the Parallel Huffman Encoder proposed by author Taavi Adamson[3]. They ran the encoder to compress a fixed problem size which is a 6.5 MB text file with an increasing number of processors. Figure 09 (a) shows the strong scaling results they achieved. Their program scales decently which is very similar to the ideal scaling up to process counts of 30 or more. After that the processing time does not change much and starts to increase due to communication overheads.

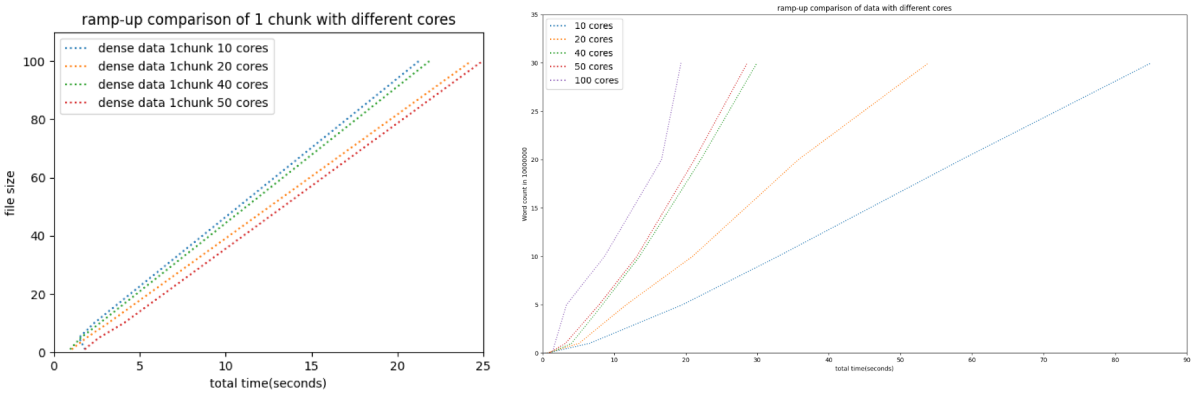
We tried to recreate the similar conditions as their experiment. Figure 09 (b) shows the strong scaling results for our algorithm. The program scales decently up to process count of 8 or more and then the processing time increases. Therefore, our algorithm does not scale well compared to theirs. One point to be noted is that we achieved a lower initial time compared to them, which indicates that the performance of our sequential algorithm is seemingly better. Of course there are other parameters such as creation of 6.5MB files and cluster configurations which can cause the difference in performance.



(a) Encoder by Taavi Adamson (b) Our Encoder

Figure 09: Comparison of Encoding Strong Scaling for 6.5MB file

For our second comparison, we are considering the Parallel Huffman Encoder proposed by author Omar Facchini[4]. They ran the experiments on similar sized datasets as ours to compress. Figure 10 (a) shows how their algorithm performs compared to ours on Figure 10 (b) on similar configurations. We can see that their performance was decreasing as they were increasing the process count from 10. On the other hand, our program performed well as we were increasing the cores.



(a) Encoder by Omar Facchini (b) Our Encoder

Figure 10: Comparison of Encoding ramp up performance

5 Discussion

The conversion of a serial code to a parallel MPI version has been demonstrated to lead to a meaningful speed up. However, MPI comes with significant overheads, such as message passing, which can negate the benefits of parallelism in the case of small datasets. In the case of large datasets, the speed up is still significant and only slightly sublinear in the number of cores. From the data, we can say the performance of the MPI parallel huffman encoder and decoder is mixed. While some speedups are achieved as the number of processors increases, these speedups are not consistent across all problem sizes and sometimes even decrease as the number of processors increases. It is possible that there are limitations in the implementation of the MPI parallel huffman coder that are affecting its performance. Factors such as load balancing, communication overhead, and memory access patterns could be affecting the performance. Further investigation into these factors may help to improve the performance of the encoder.

In conclusion, while the MPI parallel huffman coder does show some improvement in speedup as the number of processors increases, the overall performance is not consistent and may require further

optimization. To further improve performance, the use of OpenMP may be considered as a future direction for this work.

6 References

[1]https://www.intel.com/content/www/us/en/developer/articles/technical/fast-computation-of-huffman-codes.html

[2] https://toledy.github.io/ParallelRayleighBenardConvection/benchmark.html

[3] https://courses.cs.ut.ee/MTAT.08.020/2016_fall/uploads/Main/Huffman.pdf

[4] https://github.com/OmarFacchini/HPC-Parallel-Huffman-Coding-Decoding

[5] https://www.topcoder.com/thrive/articles/huffman-coding-and-decoding-algorithm