Accelerating Huffman Coding Compression Using MPI

Ramel Cary B. Jamen
CSC175 Parallel and Distributed Computing

June 26, 2023

Contents

Abstract

This proposed project is an implementation of Parallelization in a data compression algorithm called Huffman Coding. Parallelization involves breaking down a sequential process into smaller tasks that can be executed simultaneously, it improves its performance and speed by reducing the file's redundant and irrelevant digital information while preserving the original content when decoded. The main advantages of compression would be reduction of expense in storage hardware and transmission time of data. It will utilize Python's module named MPI4PY; a Message Passing Interface standard for parallelization. The study inspires to identify significant difference for run time efficiency from the Sequential Version and Paralleled Version of Huffman Coding.

1 Introduction

Data compression algorithms refer to techniques used to reduce the size of files while maintaining the same or similar data. This is accomplished by eliminating redundant data or reformatting the data for improved efficiency. There are two main methods of compression: lossy and lossless. Lossy compression permanently removes data, while lossless compression preserves the original data.[1] The choice between these methods depends on the desired level of quality for the compressed files.

Huffman coding is a highly effective technique for compressing data while preserving all of the original information. In the field of computer science, information is represented using binary digits, or bits, which are 1's and 0's. A sequence of bits encodes the information that directs a computer to perform specific tasks. From video games and photographs to movies and more, various types of data are encoded as lengthy sequences of bits in computer systems.

Given that computers can execute billions of instructions in a single second and that a single video game can contain billions of bits of data, it's crucial to have an efficient and clear-cut method of encoding information, which makes Huffman Compression a highly relevant topic in computer science.

In dealing with compression, the input stream will be the size of the file or the sequence of bytes to be compressed, and the assumption is that it is encoded sequentially from every bytes, therefore we grasp the idea of being able to apply parrallelization on that serial region of the algorithm

In this project, we may be able to utilize MPI4PY, a Python bindings for the Message Passing Interface (MPI) standard, that exchanges messages between multiple computer processors running a parallel program across distributed memory

1.1 Background of the Study

This study is mainly accelerating the sequential algorithm of the Huffman Coding that encodes the information in frequency of the variable-length strings to represent symbols in a binary tree. It seeks the answer to the question of differences of run time when it is applied with parallelization and how it is implemented. The Pseudocode of the algorithm is presented below;

Figure 1: Pseudocode of Huffman Coding

```
create a priority queue Q consisting of each unique character.
sort then in ascending order of their frequencies.
for all the unique characters:
    create a newNode
    extract minimum value from Q and assign it to leftChild of newNode
    extract minimum value from Q and assign it to rightChild of newNode
    calculate the sum of these two minimum values and assign it to
        the value of newNode
    insert this newNode into the tree
return rootNode
```

1.1.1 General Objectives

The goal of this project is to be able to present a parallelized algorithm with Message Passing Interface (MPI) in python that can digest a text file with ASCII characters as an input and seeks to answer to these following objectives;

- To be able to assess efficiency in implementing parallelization by differentiating process runtime
- To be able to identify yield performances on different processor workers

1.1.2 Specific Objectives

To be able to attain the General Objectives, there are several requirements to proceed; First is having a serial version of the implementation of Huffman Coding for the basis, Second is formulate a parallelized version of the basis using MPI4PY, third is differentiating their run time using time module of python, and lastly, test with 2 or more logical core workers of the computer unit. On the hardware specification, it requires at least 4 logical cores for executing MPI algorithm

1.2 Scope and Limitation

The scope of this project inputs only ASCII character encoding format, and does not include image and video compression. The difference of hardware capabilities of logical cores of a computer in implementation may affect the run time results, and this study solely sticks to the hardware and software specification of the computer that will be stated in methodology.

Run time may vary as the usage of CPU is shared with some applications that will be used in processing of the algorithm. Getting the average run time of 5 repetitions will be the value of its process time cost.

2 Review Literature

2.1 Compression

Data Compression is a branch of Information Theory that is often known as bit-rate reduction, it fits into the field because it concerns with redundancy. Information Theory uses the term entropy as a measure of how much information is, entropy fits with data compression in its determination of how many bits of information are actually present in a message.[4]

Data compression involves the process of representing information in a compact form. This is achieved by identifying and leveraging structures inherent in the data. The need for data compression arises from the fact that an increasing amount of information generated and utilized exists in digital form, represented by bytes of data. By compressing this digital information, we can optimize storage space and improve data transmission efficiency.[5]

2.2 Huffman Coding Algorithm

Huffman's algorithm, renowned in the realm of computing, holds a near-legendary status when it comes to calculating prefix-free codes with minimal redundancy. Its remarkable fusion of simplicity and practicality has elevated it to the status of a beloved illustration in algorithms courses, consequently making it one of the most frequently implemented techniques in the field of algorithms.

Huffman's concept, in retrospect, is remarkably elegant in its simplicity. The algorithm begins by assigning the initial weights to a set of leaf nodes, with each node representing one symbol from the input alphabet. A greedy approach is then employed, where the two nodes with the lowest weights are identified and removed from the set. These two nodes are combined to form a new internal node, which is assigned a weight equal to the sum of the weights of its constituent nodes. This newly created node-weight pair is reintroduced to the set, and the process is iteratively repeated.[6]

3 Methodology

In this section, in order to satisfy the objectives of this study, there will be a need of systematic methods or steps for implementation. The following will be;

I Sequential Version of Huffman Coding Compressor - Serves as the basis for identifying significant difference for the formulated parallelized algorithm

II Parallelized Version of Huffman Coding Compressor - This will be formulated after thorough assessment for parallelization

III Identify Findings - To be able to assess efficiency and performance differences for both algorithm

3.1 Materials

In formulating a parallelized Huffman Coding algorithm, there is a need to comply to some of the specification that will define how an application will interact with system hardware, other programs and human users in a wide variety of real-world situations.

3.1.1 Sequential Algorithm Implementation

Figure 2: Sequential Algorithm of Huffman Coding

```
# Huffman Coding in python

contents = open('textfile.txt', 'r')
string = contents.read()

# Creating tree nodes
class NodeTree(object):

    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right

    def children(self):
        return (self.left, self.right)

    def nodes(self):
        return (self.left, self.right)

    def __str__(self):
        return '%s_%s' % (self.left, self.right)


# Main function implementing huffman coding
def huffman_code_tree(node, left=True, binString=''):
    if type(node) is str:
        return {node: binString}
    (l, r) = node.children()
    d = dict()
    d.update(huffman_code_tree(l, True, binString + '0'))
    d.update(huffman_code_tree(r, False, binString + '1'))
    return d


# Calculating frequency
freq = {}
for c in string:
    if c in freq:
        freq[c] += 1
    else:
        freq[c] = 1

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)
```

```
nodes = freq

while len(nodes) > 1:
    (key1, c1) = nodes[-1]
    (key2, c2) = nodes[-2]
    nodes = nodes[:-2]
    node = NodeTree(key1, key2)
    nodes.append((node, c1 + c2))

    nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman_code_tree(nodes[0][0])

print(' Char | Huffman code ')
print('----------------------')
for (char, frequency) in freq:
    print(' %-4r |%12s' % (char, huffmanCode[char]))
```

3.1.2 Parallel Algorithm Implementation

Figure 3: Parallel Algorithm of Huffman Coding

```
from mpi4py import MPI

class NodeTree(object):
    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right

    def children(self):
        return (self.left, self.right)

    def nodes(self):
        return (self.left, self.right)

    def __str__(self):
        return '%s_%s' % (self.left, self.right)


# Main function implementing huffman coding
def huffman_code_tree(node, left=True, binString=''):
    if type(node) is str:
        return {node: binString}
```

```
    (l, r) = node.children()
    d = dict()
    d.update(huffman_code_tree(l, True, binString + '0'))
    d.update(huffman_code_tree(r, False, binString + '1'))
    return d

def chunk_file(file_path, chunk_size):
    # Open the file
    with open(file_path, 'r') as file:
        # Read the file contents
        file_data = file.read()

        # Calculate the total number of characters in the file
        file_size = len(file_data)

        # Calculate the number of chunks
        num_chunks = (file_size + chunk_size - 1) // chunk_size

        # Calculate the chunk range for each process
        chunk_start = rank * (num_chunks // size)
        chunk_end = (rank + 1) * (num_chunks // size)
        if rank == size - 1:
            chunk_end = num_chunks

        # Get the starting and ending indices of the chunk
        start_index = chunk_start * chunk_size
        end_index = min((chunk_end * chunk_size), file_size)

        # Extract the chunk of data
        chunk_data = file_data[start_index:end_index]

        # Process the chunk (e.g., analyze, manipulate, etc.)
        process_chunk(chunk_data, chunk_start)

def process_chunk(chunk_data, chunk_index):

    # Calculating frequency
    freq = {}
    for c in chunk_data:
        if c in freq:
            freq[c] += 1
        else:
            freq[c] = 1

    # Gather frequencies from all processes
    freq = comm.gather(freq, root=0)

    # Merge frequencies from all processes
    merged_freq = {}
```

```
    if rank == 0:
        for f in freq:
            for key, value in f.items():
                if key in merged_freq:
                    merged_freq[key] += value
                else:
                    merged_freq[key] = value

    # Sort the frequencies
    freq_items = sorted(merged_freq.items(), key=lambda x: x[1],
                                        reverse=True)
    nodes = freq_items

    while len(nodes) > 1:
        (key1, c1) = nodes[-1]
        (key2, c2) = nodes[-2]
        nodes = nodes[:-2]
        node = NodeTree(key1, key2)
        nodes.append((node, c1 + c2))

        nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

    if rank == 0:
        huffmanCode = huffman_code_tree(nodes[0][0])

        print(' Char | Huffman code ')
        print('----------------------')
        for (char, frequency) in freq_items:
            print(' %-4r |%12s' % (char, huffmanCode[char]))

if __name__ == "__main__":

    # Get the MPI rank and size
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    # Specify the file path and chunk size
    file_path = "textfile.txt"
    chunk_size = size # Adjust the chunk size as per your
                                        requirements

    # Call the chunk_file function
    chunk_file(file_path, chunk_size)
```

3.1.3 Hardware and Software

Hardware Specification

- Laptop or Personal Computer
- Multi-core processors (Preferably 4-16)

Software and Operating System Specification

- Preferably Unix-based Operating System
- Python 3.10.6 or newer version installed
- MPI4PY module installed using pip

3.2 Methods

In making the sample file, there is a combination of Unix commands designed to generate an arbitrarily large, random text file containing one million lines of 100 randomly generated alphanumeric characters each. It is generated by invoking the following code in a terminal;

Figure 4: Command for Input File

```
$ tr -dc "A-Za-z 0-9" < /dev/urandom | fold -w100 | head -n
  1000000 > textfile.txt
```

It begins by using the 'tr' command to filter out characters "A-Z a-z 0-9" from the output of the '/dev/urandom' file, which provides random data. The filtered data is then passed through the 'fold' command to wrap each line at a width of 100 characters. Next, the 'head' command selects the first 1,000,000 lines from the folded output. Finally, the resulting output is redirected to a file named "textfile.txt". As a result, "textfile.txt" contains a sizable amount of random text, comprising 1,000,000 lines, with each line limited to 100 characters in length. [8] Now we have 101.0 MB (101,000,000 bytes)

A parallel algorithm is an algorithm that can execute several instructions simultaneously on different processing device, in formulating Parallelized

Huffman Coding, there are two main points that needs to remark, to identify the serial sections of the algorithm and design a model of computation to operate on a set of instructions for MPI4PY.

In a thorough assessment of the sequential algorithm, the part where it will take time will be the reading of each string from the file and calculating frequency. Given the most time-costly serial sections of the algorithm, communication from different processors must be applied to the program. In application, the large file will be subdivided into chunks of data, and assigning each chunk to the number of processors available.

The parallelized Huffman Coding algorithm adopted a model of computation called Multiple Instruction, Single Data (MISD), it is a parallel computing architecture where a multiple instructions operates on single data element simultaneously. By executing a multiple instruction on single data element, MISD enables faster processing of tasks involving repetitive computations on large data sets. [9]

4 Results and Discussion

To address our third main objective, assessment for efficiency and performance will be handled by the ”time” command in Linux, it is used to measure the execution time of a command or a process. When you precede a command with the ”time” keyword, it will display the amount of real time, user CPU time, and system CPU time taken by that command or simply summarizes system resource usage. It provides information by invoking this command in terminal: ***”$ time [OPTIONS] [COMMAND]”*** where ”file” would be the ***”mpiexec -n [# OF PROCESS] python3 [FILE]”***.

4.1 Results

Figure 5: User Run Time

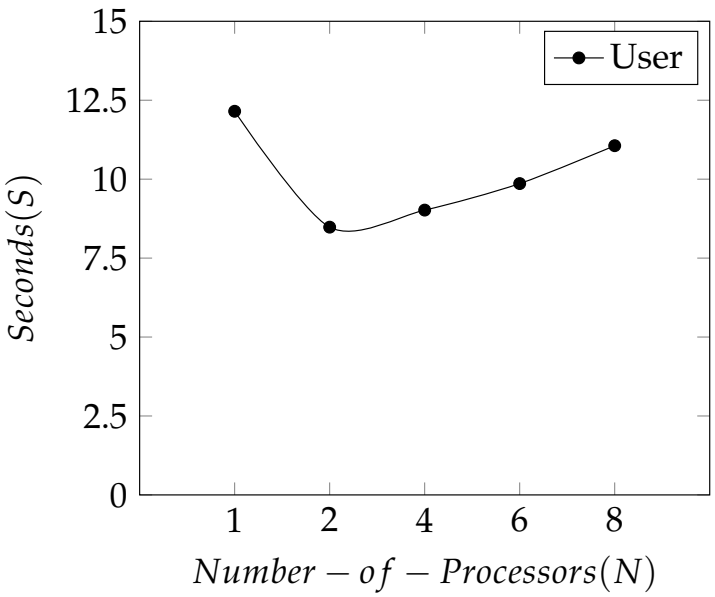


Figure 6: System Run Time

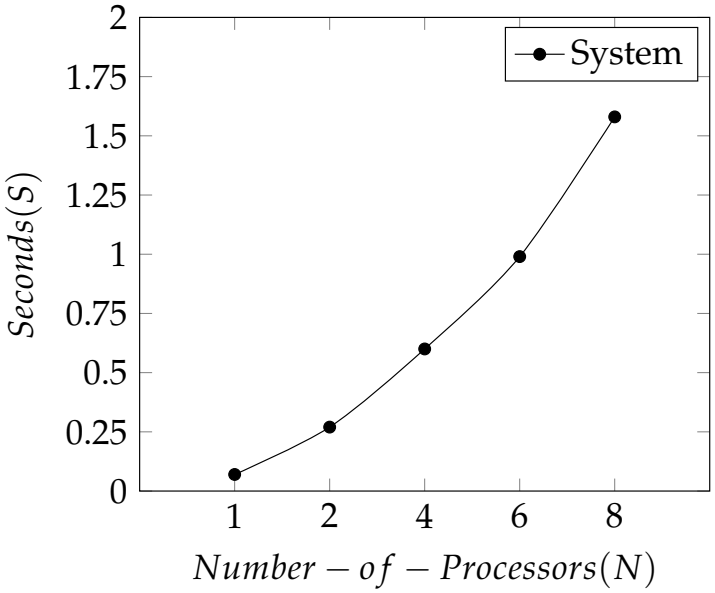


Figure 7: CPU Usage

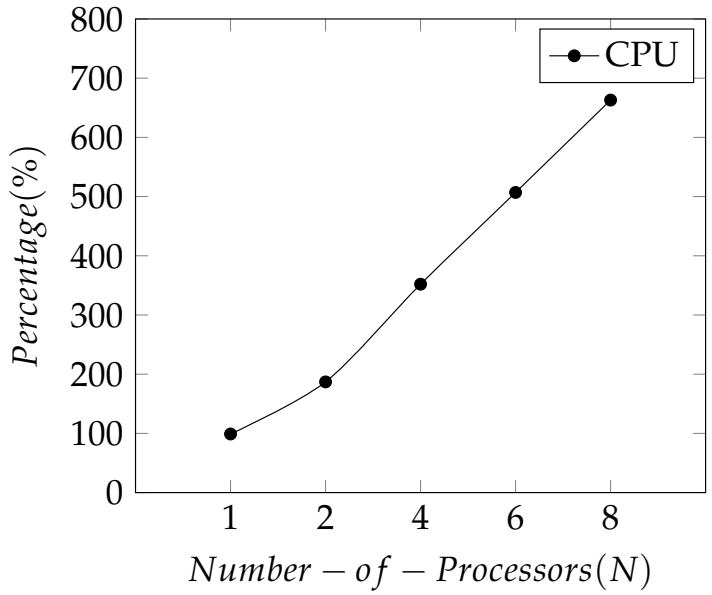
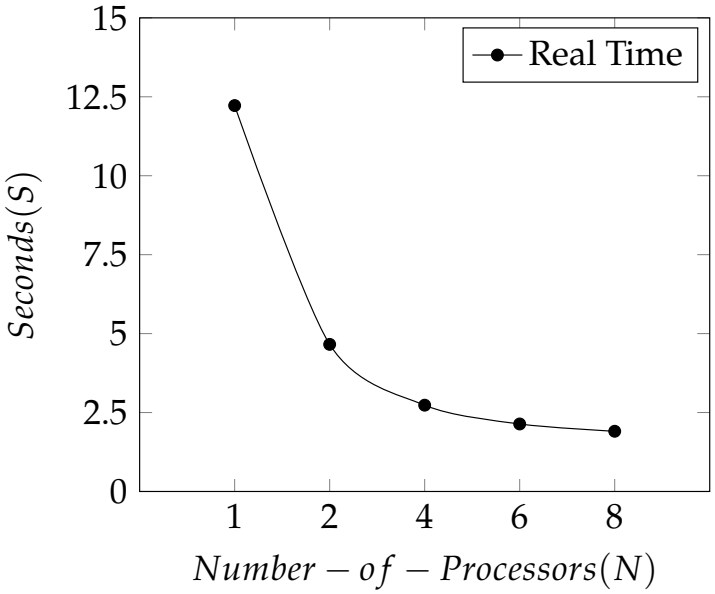


Figure 8: Real Run Time



4.2 Discussion

In figure 5, the user time is the total number of CPU-seconds that the process used directly (in user mode), it is expressed in seconds. This is only actual CPU time used in executing the process.

In figure 6, it is total number of CPU-seconds used by the system on behalf of the process (in kernel mode), it is expressed in seconds. This means executing CPU time spent in system calls within the kernel, as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process. Graph shows an increase of system run time as the number of processor increases.

In figure 7, percentage of the CPU that this job got. This is just user + system times divided by the total running time. Graph shows an increase of CPU usage as the number of processor increases.

In figure 8, Elapsed real (wall clock) time used by the process, in seconds. This is all elapsed time including time slices used by other processes and time the process spends blocked. Graph shows a drastic decrease of time-spent from single processor to double processor used. However, from 2 to 8 processors, it shows a minimal effectivity.

In general performance of the algorithm, computation optimization is clearly visible as it decreases the time for the Huffman Code to arrive a solution. However, we can also see the limits of parallelization in figure 7 as it decreases the effectivity of optimization, this is answered by Gene Ahmdahl, stating that "the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used".

5 Conclusion

In conclusion, parallelization offers significant benefits in improving computational efficiency and system performance by executing tasks simultaneously on multiple processing units. It is particularly effective for tasks that can be easily divided into independent subtasks. However, parallelization has limitations when tasks have dependencies or require excessive synchronization and communication. Resource availability and scalability also impact its effectiveness. To maximize the benefits of parallelization, careful consideration of task characteristics and appropriate techniques is crucial. Overall, parallelization holds great potential for enhancing performance while acknowledging its limitations.

6 Recommendation

For future journalist who aspires to work on a similar interest, encouraging them to explore the following areas; inclusion of decoding algorithm, different parallelization strategies for Huffman coding. Task decomposition approaches and data partitioning techniques. Identification of challenges and limitations in parallel Huffman coding. Areas for future research and development in optimizing parallelization.

List of Figures

References

[1] Smith, S. W. (n.d.). The scientist and engineer's Guide to Digital Signal Processing.

[2] Srivastava, V. (2020, October 31). 15 most popular data compression algorithms. Geeky Humans.

[3] Data compression. Barracuda Networks. (2022, October 21).

[4] Nelson, M., Gailly, J. (n.d.). The Data Compression Book 2nd edition.

[5] Sayood, K. (n.d.). Introduction to Data Compression.

[6] MOFFAT, A. (n.d.). Huffman coding - university of Melbourne.

[7] Huffman coding. Programiz. (n.d.).

[8] Codemonkey. Looking for large text files for testing compression in all sizes. Stack Overflow.

[9] Wikimedia Foundation. (2023, March 27). Multiple instruction, single data. Wikipedia.