UNIVERSITÀ DI PISA

University of Pisa
Master’s in computer science and Networking

Project: Huffman Coding

Student Name : Sagar Kumar
Student ID: 648359

Parallel and distributed systems: paradigms and models Academic year 2023/2024

Table Of Contents

1. Chapter 1 : Introduction

1.1 Huffman Coding

Huffman coding is a compression algorithm used to efficiently represent data by assigning variable-length codes to different symbols based on their frequencies in the input. It was developed by David A. Huffman in 1952.

The algorithm works by constructing a binary tree called the Huffman tree, where each leaf node represents a symbol and the path from the root to each symbol determines its code. Symbols with higher frequencies have shorter codes, leading to overall compression.

Here's how Huffman coding works:

Frequency Analysis: First, the algorithm scans the input data to determine the frequency of each symbol.

Build Huffman Tree: It then constructs a binary tree where each node contains a symbol and its frequency. The two nodes with the lowest frequencies are combined into a new node, with the sum of their frequencies as the frequency of the new node. This process continues until all nodes are connected, forming the Huffman tree.

Assign Codes: Starting from the root, traverse the tree assigning '0' for left branches and '1' for right branches. Each symbol is assigned a unique binary code based on its position in the tree.
Encode Data: Replace each symbol in the input data with its corresponding Huffman code to generate the compressed data.
For example, consider a string "ABBCCCDDDD". After frequency analysis, the tree might look like:

```
           15
          /   \
        7      8
       /  \   / \
      3    4  4   4
     /\  /\
    A  B C  D
```

In this tree, 'A' might get '000', 'B' might get '001', 'C' might get '010', and 'D' might get '011'. 'E' would simply be '1'. Thus, "ABBCCCDDDDEEEEE" could be represented as "000001001010010010011011011011"
Previously it was consuming 80 bits in memory and after compression it consume 30 bits in memory so now 50 bits are free, and the file is easy to store in memory.

1.2 Problem Overview

This project aims to create a parallel application for Huffman coding, a widely utilized method for lossless data compression. The goal is to read an input text file, compute the Huffman code, and generate a compressed version of the original file using this code.

Huffman coding, an optimal prefix code assigning shorter codes to frequently occurring characters, offers efficient compression. The parallel implementation of this algorithm aims to capitalize on parallel computing resources, expediting the compression process.

The task involves designing and developing a parallel application capable of efficiently performing Huffman coding on sizable input files. The application must read an input text file, compute the Huffman code for its characters, and generate a compressed version.

The project entails exploring diverse parallelization approaches, evaluating their performance, and selecting the most suitable one. Challenges encountered during the implementation process will be addressed.
To gauge the parallel application's effectiveness, experiments will be conducted on a remote virtual machine. Performance will be evaluated, and results will be analysed and compared with ideal or predicted figures. The report will not only cover implementation and performance evaluation but also provide instructions for compiling and running the code.

1.3 Algorithm

The algorithm outlined proposes a parallelized application for Huffman coding, aiming to compress data efficiently while considering both input and output file operations. It proceeds through several steps: file reading, character frequency counting, Huffman tree construction, code table generation, Huffman encoding, ASCII encoding, and file writing. Beginning with an ASCII file input, the algorithm undertakes lossless compression by first tallying the frequency of each character in the text. It then constructs a Huffman tree based on these frequencies, enabling the derivation of optimal prefix codes for compression.

Initially, the algorithm initializes a priority queue, Q1, comprising unique characters from the data stream. It organizes these characters in ascending order of their frequencies. For each unique character, a new node is created. The minimum values from Q1 are extracted successively to assign as left and right children of the newNode, with their sum becoming the value of the newNode in the min-heap. This process iterates until the Huffman tree's root is formed, facilitating efficient encoding.

A challenge in Huffman encoding arises post-frequency counting, where binary strings of 0s and 1s are generated. Direct writing of these binary strings would inflate file sizes significantly. To mitigate this, an ASCII encoding strategy is employed, accommodating multiples of 8 bits.

To align with ASCII encoding standards, padding 0s are appended if the binary string's length is not already a multiple of 8. This adjustment ensures proper ASCII encoding, resolving the issue of non-multiples of 8 in binary string lengths.

The algorithm's emphasis on parallelization seeks to enhance efficiency and reduce overall execution time, particularly evident in read and write operations. By parallelizing the encoding process, the algorithm aims to optimize resource utilization and streamline compression tasks effectively, ultimately improving the algorithm's performance and scalability.

2. Chapter 2 : Parallel Implementations

2.1 Parallel Implementation

C++ code implements Huffman encoding in a parallelized manner. It reads input text from a file, constructs Huffman trees, generates Huffman codes, compresses the text, and saves the compressed data to output files. It supports multithreading to improve efficiency, with each thread handling iterations of the compression process. The "workerFunction" is executed by each thread, building Huffman trees, encoding text, and saving compressed data. It employs mutexes for thread safety in time recording. The main function orchestrates thread creation, file reading, and overall execution control. Command-line arguments specify the number of iterations and threads. Finally, it outputs execution times per iteration. Overall, the code efficiently compresses text using Huffman coding while utilizing parallelization for enhanced performance.
The C++ STL implementation can be visualized as figure 2.1.

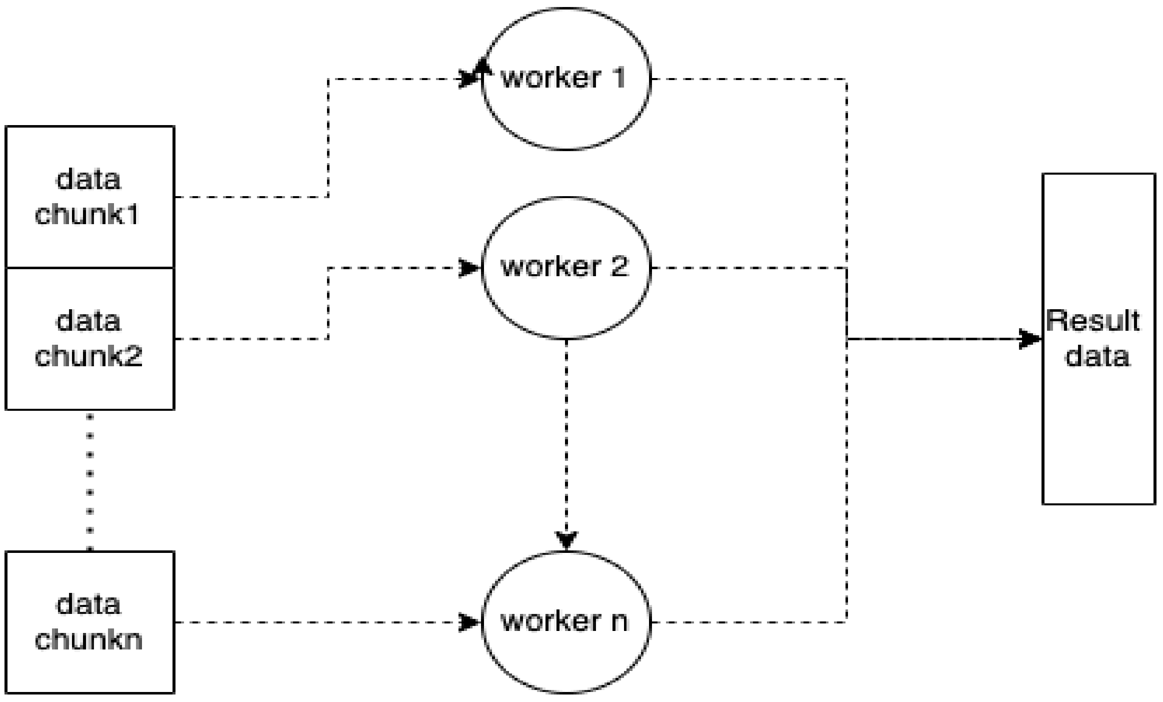


Figure 2.1: Parallel implementation

The code implementation is found under the name Huff_Par.cpp and the parameter to this implementation are the number of iteration and number of threads.

This code can be compiled with the command pasted below:
g++ -O3 Huff_Par.cpp -o <name of compiled code>
and then it will be executed by command pasted below
./<name of compiled code><name of text file> <number of iterations> <number of threads>
Name of compiled code will be given by user while compiling the code e.g. Parallel.
Name of text file is the file which will be provided as an input.
Number of Iteration is the number of times to run the code.
Number of threads are the number of threads counts which will be run concurrently to make the execution faster. Double the number of threads will decrease the execution time by half.

2.2 Fast Flow

C++ code implements Huffman coding for text compression using parallelization with the Fast Flow library. It reads text from an input file, builds Huffman trees, generates Huffman codes, compresses the text, and writes the compressed data to output files. The main function orchestrates the parallel execution by spawning multiple threads, each responsible for compressing a portion of the text across multiple iterations. The program accepts command-line arguments for the number of iterations and threads to employ. It utilizes the Fast Flow library's ParallelFor construct to distribute the workload efficiently among threads. After compression, the program calculates and prints the execution time for each iteration. Overall, the code demonstrates a parallelized approach to Huffman coding, aiming to improve compression performance through concurrent processing.

The code implementation is found under the name Huff_FF.cpp and the parameter to this implementation are the number of iteration and number of threads.

This code can be compiled with the command pasted below:
g++ -O3 Huff_FF.cpp -o <name of compiled code>
and then it will be executed by command pasted below
./<name of compiled code><name of text file> <number of iterations> <number of threads>

Name of compiled code will be given by user while compiling the code e.g. FastFlow.
Name of text file is the file which will be provided as an input.
Number of Iteration is the number of times to run the code.
Number of threads are the number of threads counts which will be run concurrently to make the execution faster. Double the number of threads will decrease the execution time by half.

2.3 Difference between Parallel and Fast Flow Implementations

Huffman coding parallel implementation and Huffman coding FastFlow implementation differ primarily in their underlying parallelization frameworks and their approaches to concurrency.

Parallel Implementation:

Huffman coding parallel implementation typically involves manual threading or using threading libraries like std::thread in C++ to parallelize different stages of Huffman coding, such as tree construction, code generation, and text compression.
Developers have more control over thread management, synchronization, and data sharing but may require additional effort to handle these aspects correctly.

Fast Flow Implementation:

Fast Flow is a parallel programming framework specifically designed for high-performance computing tasks, emphasizing efficient communication and workload distribution.
In Huffman coding Fast Flow implementation, developers utilize Fast Flow's constructs, such as ParallelFor, to parallelize computations across multiple cores or processors.
Fast Flow abstracts away low-level threading details, providing a higher-level interface for expressing parallelism, which can simplify development and improve scalability.

3. Chapter 3: Results

For performance evaluation, I conducted experiments on a remote system featuring a dual-socket AMD EPYC 7301 processor. Each socket hosts 16 cores, totalling 32 cores with 2-way hyper-threading, thus providing 64 hardware threads. The system is equipped with 256GB of main memory and runs on Linux Ubuntu 22.04 with g++ version 11.3.0. The code was executed three times for each dataset, and the average was taken to provide more accurate measurements. I utilized the Chrono Library for time measurements. The experiments covered data sizes of 3MB and 10MB, and execution times were averaged over three iterations for each dataset, varying the number of workers (nw) from 1 to 64.

3.1 Data

To assess the algorithm's performance and evaluate its effectiveness, I employed two datasets: " Anna Karenina.txt" of size 10 MB and " Magyar Bancorp, Inc.txt" of size 3 MB located within the directory.

These datasets offer insights into the algorithm's behaviour across varying data sizes, enabling comprehensive analysis of its capabilities with both smaller and larger datasets.

In this chapter we will be discus the result of Sequential, Parallel and FastFlow Implementation with different number of threads.
We will use two datasets: " Anna Karenina.txt" of size 10 MB and " Magyar Bancorp, Inc..txt" of size 3 MB located within the directory.

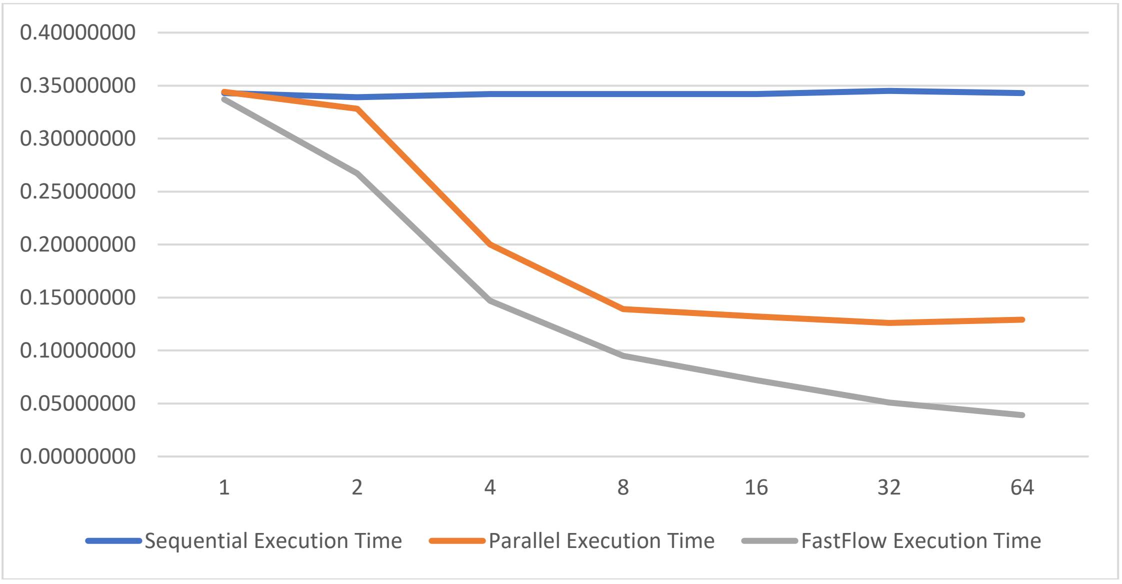
3.2 Dataset Results Sheet

Input Size	Sequential Execution Time	Parallel Execution Time	FastFlow Execution Time	# Workers
Anna_Karenina [9631 KB]	0.34300000	0.34406200	0.33700000	1
	0.33900000	0.32800000	0.26700000	2
	0.34200000	0.20000000	0.14700000	4
	0.34200000	0.13900000	0.09500000	8
	0.34200000	0.13200000	0.07200000	16
	0.34500000	0.12600000	0.05100000	32
	0.34300000	0.12900000	0.03900000	64

Dataset Table 4.1

Here in this table 4.1, we have analysed all three implementations of the Huffman coding with different number of workers. In sequential we have notice that the execution time is almost the same because all the work is done in sequential manner, but we have also observer the changes in parallel and fast flow implementation as the number of workers increase the execution time decreases. This is because the worker divided the data load equally and each worker do their work on their own specific piece of data, so the work is less as compared to sequential implementation.

3.3 Dataset Result Graph



3.4 Conclusion

Each stage of a process possesses its unique characteristics, necessitating the selection of the most efficient approach based on available resources and input size. Huffman coding, in particular, heavily relies on both the input data and the diversity of characters it encompasses. When implementing Huffman coding in C++, various strategies can be employed, each yielding different results. By analysing the outcomes through tables and graphs, we can observe distinct patterns.

The impact of threading becomes apparent when examining performance metrics. Initially, with only one core dedicated to threading, the execution closely mirrors that of the sequential approach. However, as the number of threads increases, performance improves significantly. Nevertheless, there exists a saturation point where adding more threads does not yield the anticipated performance gains.

In evaluating parallelization strategies, it is crucial to consider dependencies and the nature of the input data. Certain parallelization techniques may be more effective depending on the specific characteristics of the problem at hand. Understanding the intricacies of the data and its dependencies allows for informed decision-making when selecting the optimal approach.

In essence, the effectiveness of parallelization hinges on a thorough understanding of the problem domain. It requires careful consideration of various factors, including resource constraints, input characteristics, and dependencies within the algorithm. By conducting comprehensive analyses and testing different approaches, developers can identify the most suitable parallelization strategy for a given task. This iterative process ensures that the chosen approach maximizes performance while minimizing resource utilization.