1 Parallel Huffman Coding

1.1 Report

This report presents a parallel ASCII file compression algorithm utilizing the Huffman Coding Algorithm. The algorithm exploits the concurrency of contemporary multicore processors to enhance the performance of the traditional algorithm. The primary contributions of this work are as follows:

- A **parallel** implementation of the Huffman Coding Algorithm, employing C++ native threads, which are low-level primitives for creating and managing threads.
- A **parallel** implementation of the Huffman Coding Algorithm, utilizing the **FastFlow C++ library**, which facilitates the development of parallel applications, modeled as structured directed graphs of processing nodes, on multicore platforms.
- A **sequential** implementation of the Huffman Coding Algorithm, serving as a baseline for comparison.
- A series of experiments, written in Python Notebooks, to measure the performance of the three implementations, using varying input sizes and numbers of threads. The results are reported in terms of **execution time**, **speedup**, and **scalability**.
- The implementations of the algorithm.
- Integration tests to verify the correctness of the parallel solutions.

1.2 Algorithm phases

After conducting an analysis, the following steps were identified as necessary to compress files using the Huffman Coding algorithm:

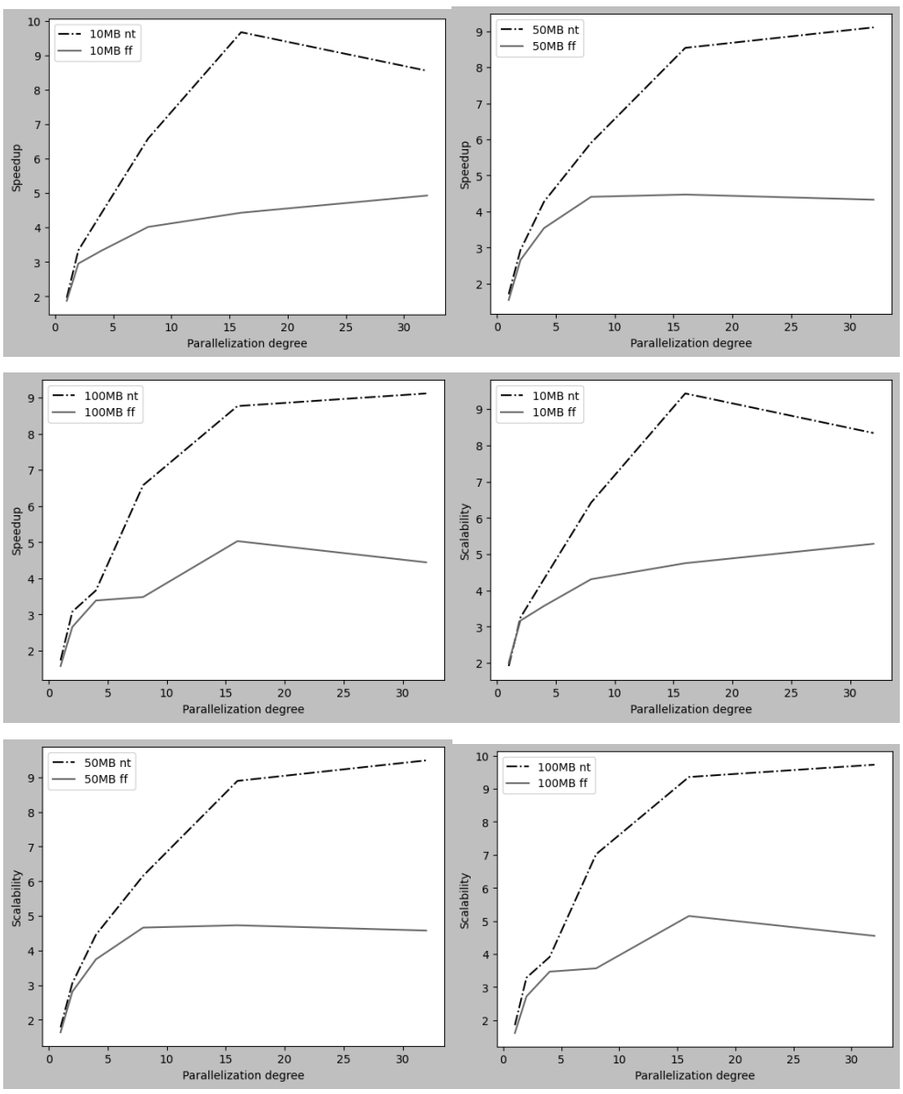
1. **File read**: The file to be compressed is read.
2. **Counting characters**: The frequency of each character in the file is counted.
3. **Build the Huffman tree**: A Huffman tree is built based on the character frequencies.
4. **Build the encoding table**: An encoding table is created, which maps each character to its respective Huffman code taken from the Huffman tree.
5. **Encoding phase**: The ASCII file is encoded using the encoding table.
6. **Compression phase**: The compressed string is grouped into bitset<8> to actually compress the file.
7. **Write the compressed file**.

1.3 Performance tests

Since the goal of this project was to identify areas for improvement in the sequential solution using parallel computing techniques, I measured the execution time of each stage in the sequential solution, enabling the identification of potential candidates for refactoring and enhancement using multithreading. Additionally, I performed performance tests (executed by a machine using a dual socket, 16 cores each, 2 way hyper threading AMD EPYC 7301) with both the parallel implementations (native threads, and FastFlow C++ library).

Stages performance

I decided to first benchmark how much each stages takes in the entire process of compression and here's what emerged by measuring the time taken from each stage (excluding the read/write and the actual compression) sequentially with three different file sizes which are s=10MB, m=50MB, and l=100MB:



(a)

Figure 1: Performance tests

stages performance (%)	counting	huffman tree	encoding
s	57.04	0.80	42.16
m	57.53	0.92	41.55
l	57.00	0.98	42.02

The reason why the Huffman tree phase is much quicker is because the tree has an upper bound equals to 256 which is the dimension of the ASCII table.

Speedup and scalability tests

Recalling that *speedup* (Figure 1) is a measure of the improvement in execution time achieved through parallelization and that $speedup(n) = Tseq/Tpar(n)$ and *scalability* (Figure 1) measures the ability of the system to maintain or improve its performance as the problem size or the number of processing units is scaled up $scalability(n) = Tpar(1)/Tpar(n)$. In the following table there is the speedup table, considering that I took into account only the time spent for the actual work, by leaving apart the time spent for I/O operations (read at the beginning and writing at the end).

Native Threads Speedup

n. workers	2	4	8	16	32	64
10MB	1.96	3.34	4.42	6.58	9.67	8.55
50MB	1.71	2.93	4.27	5.91	8.54	9.11
100MB	1.73	3.08	3.66	6.57	8.76	9.11

FastFlow Speedup

n. workers	2	4	8	16	32	64
10MB	1.87	2.95	3.33	4.02	4.43	4.93
50MB	1.55	2.66	3.54	4.41	4.47	4.33
100MB	1.57	2.65	3.39	3.48	5.03	4.44

Native Threads Scalability

n. workers	2	4	8	16	32	64
10MB	1.91	3.25	4.31	6.41	9.42	8.33
50MB	1.78	3.05	4.45	6.15	8.89	9.49
100MB	1.84	3.28	3.91	7.01	9.35	9.72

FastFlow Scalability

n. workers	2	4	8	16	32	64
10MB	2.00	3.16	3.56	4.30	4.74	5.28
50MB	1.63	2.80	3.74	4.65	4.72	4.57
100MB	1.60	2.71	3.46	3.56	5.15	4.54

For what we can see from the results, the

1.4 Implementation details and choses made

For this project, I organized the code into three distinct classes: **nt solution**, *ff solution*, and **seq solution**. These classes implement the solution using native threads, FastFlow, and a sequential pattern, respectively. In addition to these classes, I also included two files, **huffman** and **utils**, which provide the necessary functionality for building a Huffman tree and various helper functions.

The **nt solution** class leverages the power of native threads to parallelize the computation and improve performance. The *ff solution* class, on the other hand, utilizes the FastFlow framework to achieve similar results. Finally, the **seq solution** class provides a sequential implementation of the algorithm for comparison purposes.

The **huffman** file contains the code for constructing a Huffman tree, which is a crucial component of the algorithm. The **utils** file, meanwhile, provides various helper functions that are used throughout the project.

1.4.1 Characters counting

- *native threads*: Initially I was thinking to use a google map reduce, but then I realised that the actual job was to increase a number of a certain occurrence, so it was not a demanding work, therefore it was not convenient to implement it because the amount of overhead of spawning/joining mappers and reducers would have outweighted the benefits. As a result, I opted to spawn **num thread** threads, each producing a map with keys representing characters in the text and values representing the number of occurrences of that character in the file chunk read by the thread. Once all threads had completed their maps, we merged them into a single map containing the occurrences for each character in the input file. I also considered using a shared map, but the time spent waiting for locks by each thread was greater than the time required to merge all maps together.

- *fastflow*: The concept is similar to using native threads, but in this case, I utilized a *parallel for static* to enable *static load balancing*. This approach allowed for faster execution of the function, as the tasks were relatively uniform

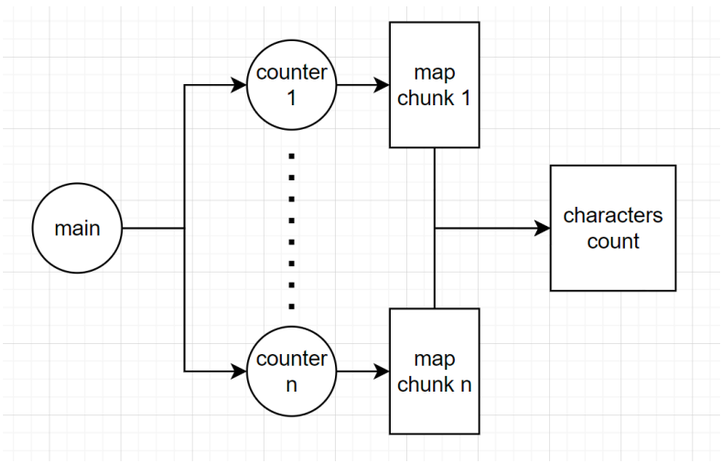


Figure 2: Characters counting phase

in terms of execution time. As a result, using dynamic load balancing would have introduced unnecessary overhead.

1.4.2 Building the Huffman Tree

The process of building the Huffman Tree is standard and straightforward. I implemented all the basic functions that allowed for standard operations such as building the Huffman tree, traversing it, printing it, and creating a new one. // Since the binary tree is filled with all the characters and symbolic nodes, traversing the tree takes only O(1) time. A map was used to store the Huffman encoding for each character, allowing for retrieval of the code for a given character in O(1) time during the encoding phase. The other stages are more expensive, as counting the characters takes O(n) time and encoding the characters also takes O(n) time.

1.4.3 Encoding

In this phase, the goal was to translate each character in the input into an encoded string. To accomplish this, multiple threads were used to take chunks of the input string and compute the encoding of each character in the chunk. The final step was to merge the chunks from the numthreads into a single string, which is the final encoding. To encode each character, the map obtained as the output of the previous phase was used, avoiding the need to traverse the tree each time. However, some doubts arose that led to future improvements: it may have been better to use a vector with 256 positions allocated on it, where each position represents a specific character's occurrence, rather than a map. This could have improved performance by exploiting data locality, as accessing an element with a hashmap could trigger a cache miss and therefore waste time. By using a vector instead, performance could have been improved by frequently hitting the cache and obtaining the encoding of characters in a faster way. Beside this, about *fastflow*, I decided to use a parallel for static iterating from 0 to parallelization degree so that I could allocate chunks to encode in parallel and after that just merging all the results, again the number chunks to merge is equal to the parallelization degree. Here I want to spend a few words regarding the use of parallel for static: the library is written so that developer should be agnostic with the respect to the threads and data parallel management, so probably the expected way to use fastflow in this case was to use parallel for such that the loop iterates over the all characters directly without worring about chunks division. I've also tried using parallel for reduce, which achieved cleaner code but suffered from slight performance limitations, therefore at the end I opted for the parallel for static with the chunks division approach. At the end of the encoding process, if the number of bits representing the input string is not divisible evenly by 8 (byte size), I append padding to the end. This approach enables treating the encoded string as a collection of $bitset < 8 >$, facilitating file compression. If we were to directly write the string, where each character occupies 1 byte, the file size would increase instead of being compressed.

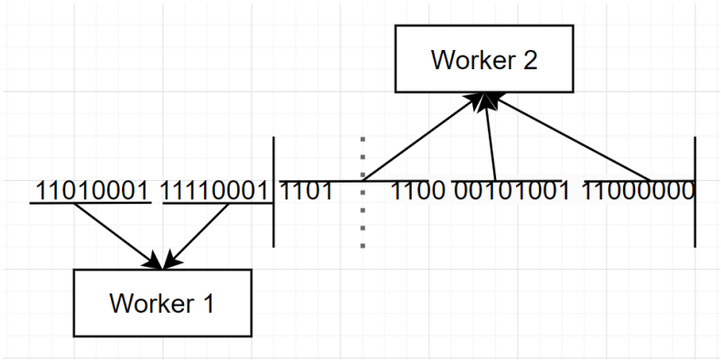


Figure 3: Compressing phase Example

1.4.4 Compression

The strategy I used for this phase was pretty much the same: divide into as many chunks as the parallelization degree. But here's happening something different: due to the fact that we want to acutally compress and declare that each bit in the string is actually a bit, we need to first create groups of bitset¡8¿ and then convert it to a char so that we can compress the sequence of bytes representing the encoding bits in actual bits. To accomplish that I consider the chunk size as the number of characters in the encoded string divided by the parallelization degree minus the remainder of the division by 8, therefore the last worker is going to work until the end of the encoded string, and we're sure that the last worker has exactly a multiple of 8 of characters to emplace as a bitset, since we're assuming that the input has as length a number that is a multiple of 8 (recalling that the encoding phase added paddings on it). Figure 3 shows an example, assuming we have 40 bits of encoded string, and a parallelization degree of 2:

1.5 How to run it

To compile the project you need to:

- set your fastflow path inside the *CMakeLists.txt* and in particular inside the parameter *CMAKE_CXX_FLAGS*
- run *cmake .*
- run *make*

After compiling the project you can run it by following this template:

```
./output <test> <mode> <par-degree> <input-file>
```

- test: if it's "test" then it will execute the tests, no tests would be executed otherwise
- mode: could be 0 (sequential), 1(native threads), or 2 (fastflow)
- parallelization degree
- input file name

The output filename is hardcoded in *./outputs/compressed* file.

To make the solution performance tests, I have created random ASCII files I explain in this example:

To create a random 10MB ASCII file, just run:

```
base64 /dev/urandom | head -c 10000000 > 10M
```

in the inputs file:

```
total 153M

-rw-rw-r-- 1 m.marino38 m.marino38 96M Jul 5 19:09 100M
```

```
-rw-rw-r-- 1 m.marino38 m.marino38 10M Jul 5 19:04 10M

-rw-rw-r-- 1 m.marino38 m.marino38 48M Jul 5 19:09 50M
```

To execute the native threads solution using 2 as the degree of parallelization on a 10MB file:

```
./output n 1 2 ./inputs/10M
```