Master Degree in Computer Science

Parallel and Distributed Systems: Paradigms and Models

Parallel Huffman encoder

Antonio Pace

Academic Year: 2022-2023

Contents

1 Introduction

This is an evaluation of an application that implements a parallel Huffman encoder for ASCII files, using both native threads and Fastflow in C++.

1.1 Huffman encoding

Huffman encoding is a prefix-free compression algorithm used for lossless data compression. It works by assigning variable-length codes to input characters, with shorter codes assigned to more frequently occurring characters. The process involves building a binary tree called a Huffman tree. Each leaf node of the tree represents a character from the input data, and the path from the root to a leaf node determines the character's code. Huffman encoding is widely used in applications like file compression and transmission, where reducing data size while preserving original information is crucial.

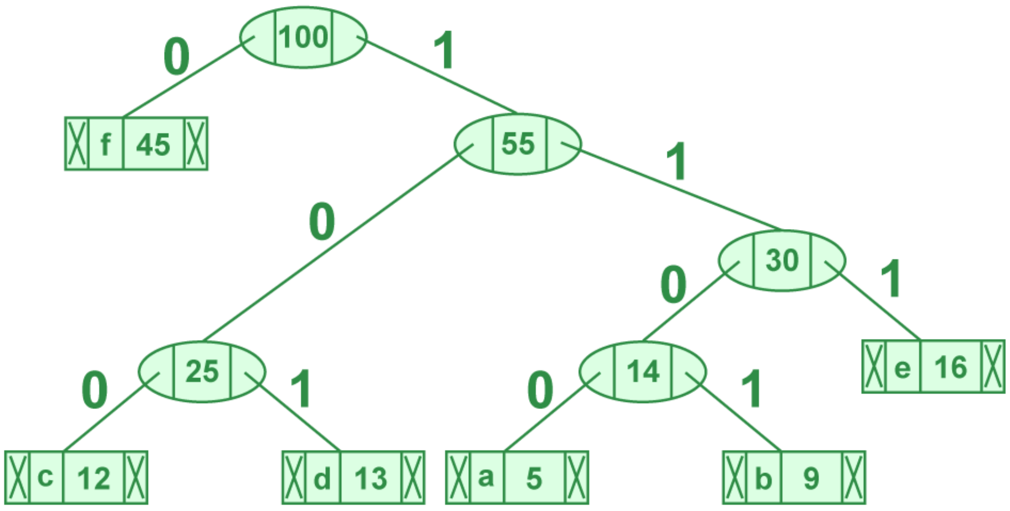

Figure 1: An Huffman tree. The node values represent the frequencies.

The process can be divided in different steps:

- Reading the file.
- Counting the characters frequencies.
- Building the Huffman tree.
- Encoding the text.
- Translating bits in ASCII.
- Writing the encoded file.

2 Sequential implementation

A sequential version has been implemented for the first evaluation of the program, for discover which steps are more complex and for having a comparison for the parallel performances.

After reading the ASCII text as a string, the program just scan it counting the frequencies of each character. Since these are char, a **vector<int>(256)** is used for holding the frequencies, so that each char has its implicit position. This solution appears to be more efficient than using a **map<char,int>**, especially if the characters are scattered.

With the frequencies is possible to compute the Huffman tree: a node of the tree is defined as a struct with data, frequency and two pointers to other nodes. First the leaf nodes are inserted in a priority queue, ordered by increasing frequency, then they are are popped out and merged creating internal nodes, until the queue is empty. The edges of the tree are labeled with "0" or "1" so that the path from the root to a leaf creates a bit sequence. The sequence for each char is held in a **vector<string>** for the same reason as before. For the encoding, the text is again scanned and for each char the corrisponding sequence is appended to a string. To create the encoded file, we need to translate each group of 8 "bit" in a char, so if the encoded sequence is not a multiple of 8, a padding and an header (1 byte) that suggests the padding are needed. The last thing to do is writing in output the final encoded sequence.

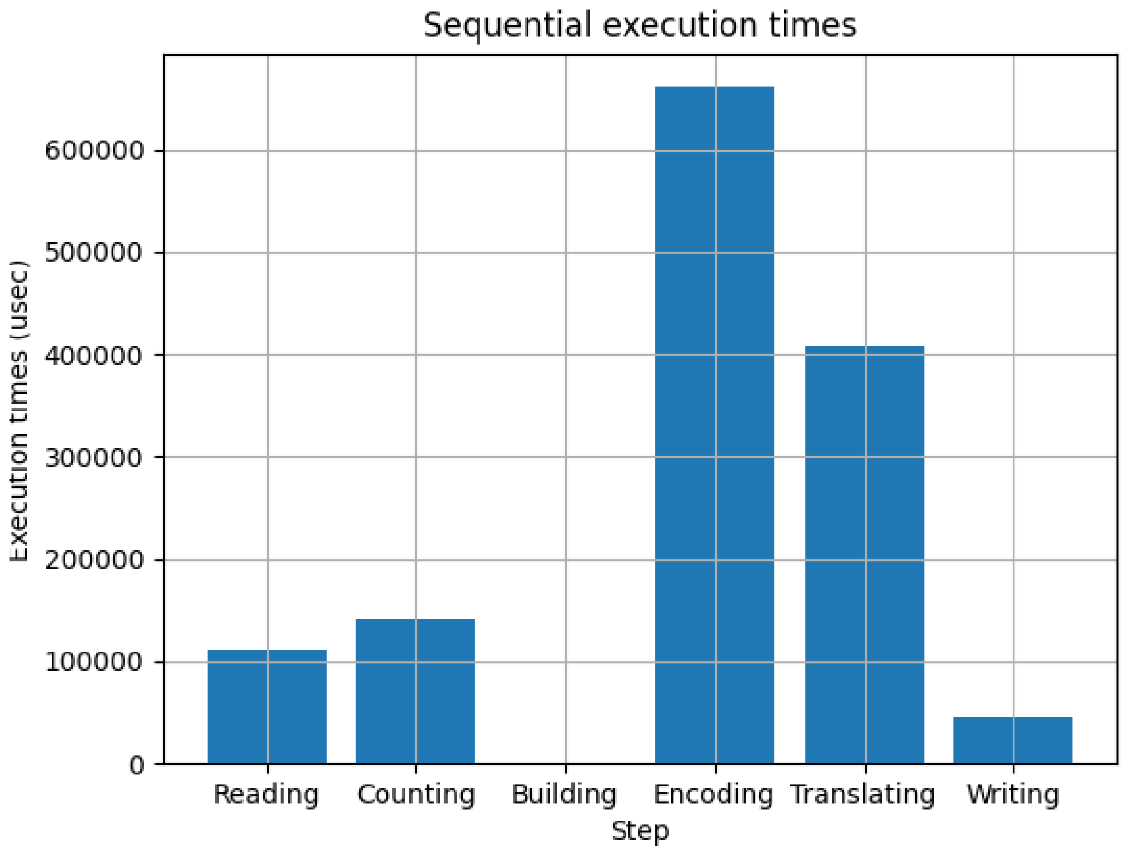

Figure 2: Sequential time for each step (50 MB file)

From Figure 2 it's possible to see the most time consuming steps. The building of the tree is trivial and parallelising I/O operations is pointless, so the choice is to compute count, encode and translation in parallel.

3 Parallel implementation

Two implementations are given: with native C++ threads and with FastFlow library[1]. The skeleton of the code is similar to the sequential version, with some adjustments for parallelizing the three steps that will be commented below. Considering the nature of the problem, the ideal way to proceed is using data parallel patterns.

3.1 Native threads

Since the counting step requires to count the frequency for each character in the input text, it can be implemented with the *MapReduce* pattern. Input text is splitted into chunks, one for each worker, that locally counts the frequencies and, with an hash function, redirects the pairs <*char,freq*> to a worker based on the key. Here starts the Reduce phase, each worker locally aggregates the pairs received and then merges the

[1] *https://github.com/fastflow/fastflow*

results in a synchronized shared vector. Each worker acts first as a mapper and then as a reducer, with a barrier in the middle.

The encoding step is a simple map, each worker encode his chunk of text accumulating the sequence into a local string and then moving it to a shared *vector<string>*. The vector is needed for mantaining the order, in fact the *i-th* worker has the *i-th* chunk of the text and put the encoded string in the *i-th* position of the vector. The full encoded string is recombined sequentially.

The translating step works in the same way, the difference is that the size of each chunk assigned to the vector has to be a multiple of 8. Every load balancing operation in the code is a static block distribution, because the workload of the tasks is the same.

3.2 FastFlow

For the counting step the first thought was to use the *ParallelForPipeReduce*, since should be ideal for cases in which in the reduce phase there are concurrent activity in a shared data structure, but the performance were quite bad. In the end, the chosen solution is to use a simple *ParallelFor* for counting the frequencies of each chunk, followed by a sequential reduce. So it's a bit different with respect to the native thread implementation.

ParallelFor is used also for the two parallel map steps, using the default static distribution, with the same reasoning and arrangements as the other implementation.

4 Overheads and evaluations

The major overhead is given by the two recombination of the string after the encoding and translating steps. This obviously grows with the file size and appears to be a big problem that affects the speedup of the solutions, because it's not present in the sequential implementation. For sure this is the main issue with the parallel solution proposed.

Step	**Time**
Reading	114270 µs
Counting	71277 µs
Building	494 µs
Encoding	236061 µs
First string recombination	352111 µs
Translating	108846 µs
Second string recombination	22233 µs
Writing	52835 µs

Table 1: Steps time, 50 MB file, 4 threads.

Also the threads' join-fork and the synchronization is a (smaller) overhead, so the parallel version is worth only with big text files. In the charts below, it's possible to see how the speedup and the scalability grows with the file size (2,4,8,16,32 and 64 threads). In fact, with the 1 MB file, the performances get worse as the number of threads increases. Considering this, an improvement to the solution could be adapting the number of workers to the file size. It can also be noticed that, also with big files, in the fastflow implementation the performances fall with 64 threads, unlike the native threads one.

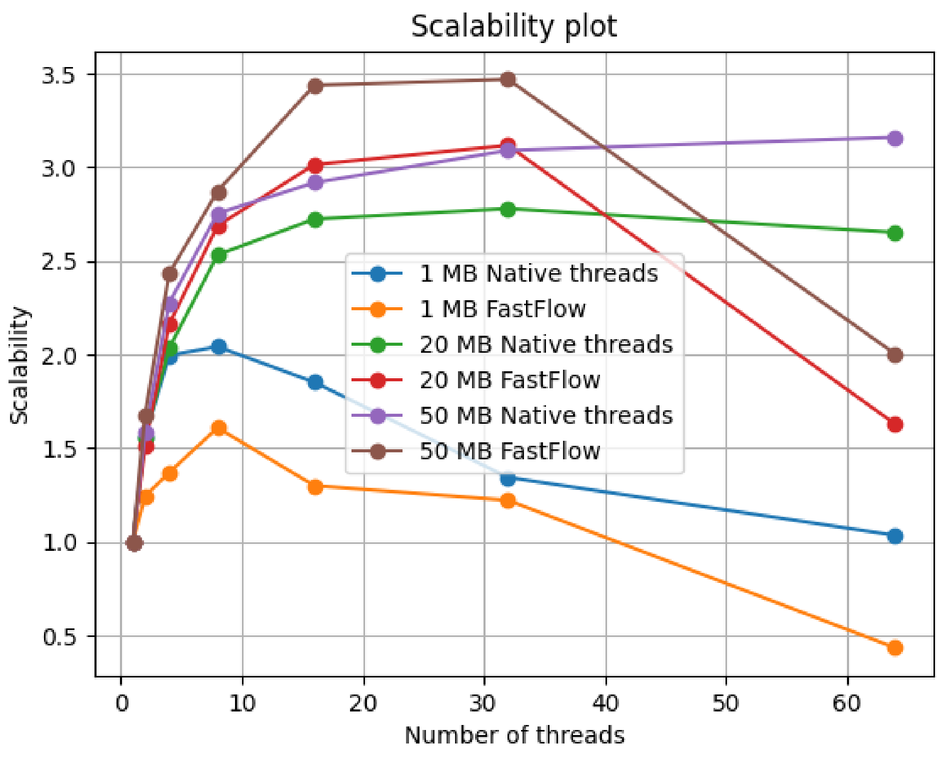

Figure 3: Scalability (no I/O)

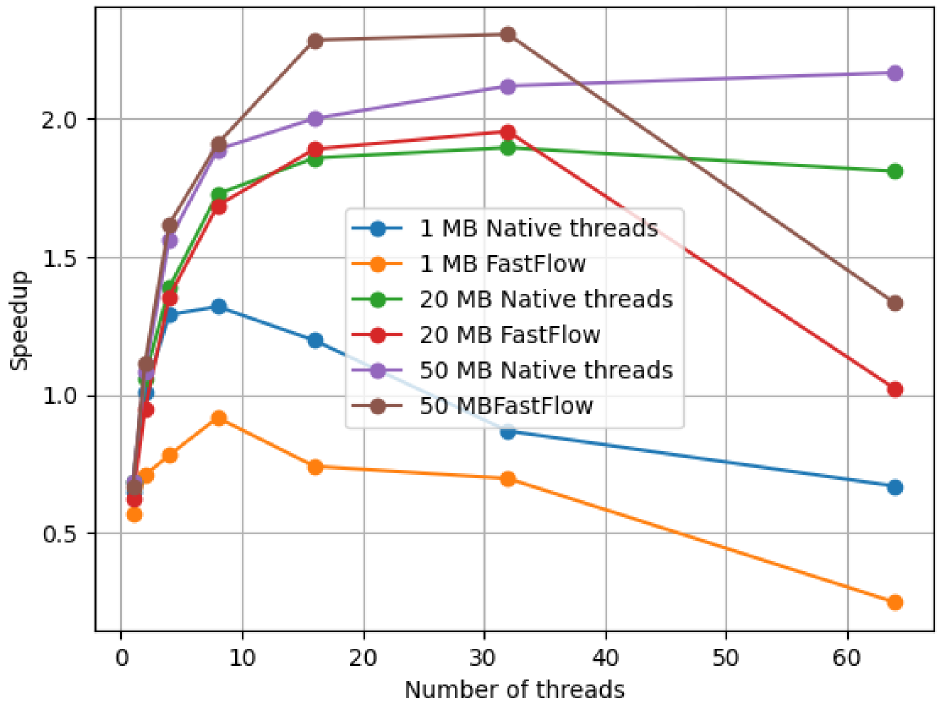

Figure 4: Speedup (no I/O)

In Table 2 it's possible to see the execution times of both implementations, for a 50 MB file. These times are taken executing 15 times for each measure, removing outliers and taking the average. For the sequential version the execution times are **1370 ms** with I/O and **1214 ms** without I/O.

Threads	**Nat. with I/O**	**Nat. no I/O**	**FF with I/O**	**FF no I/O**
1	1937 ms	1771 ms	1990 ms	1827 ms
2	1277 ms	1120 ms	1247 ms	1089 ms
4	942 ms	778 ms	912 ms	751 ms
8	801 ms	643 ms	794 ms	636 ms
16	753 ms	607 ms	682 ms	531 ms
32	718 ms	573 ms	683 ms	527 ms
64	710 ms	560 ms	1073 ms	912 ms

Table 2: Execution times of the two implementations, 50 MB file.

5 Run the code

First of all the text files should be in *txt/*.

For running the sequential version:

```
make sequential_encoder
./sequential_encoder "file.txt"
```

For the parallel version:

```
make encoder
./encoder "file.txt" "num_threads" "mode (0:native threads, 1: FF)"
```

The output will be in the form of *file_ encoded.bin* inside *bin/*.

There is also a sequential decoder:

```
make decoder
./decoder "file_encoded.bin"
```

The output will be in the form of *file_ decoded.txt* inside *dec/*.