Parallel Huffman

Giulio Piva

February 2, 2024

Contents

1 Problem statement

The Huffman algorithm is a lossless data compression algorithm that employs a variable-length code table for encoding a sequence. This code table is derived from the probability of occurrence of each distinct symbol in the input. In this scheme, the most frequently occurring symbol is encoded with the shortest code, while the least frequent symbol receives the longest code. The Huffman algorithm can be broadly divided into four major steps:

1. Frequency counting: The first operation to be performed is generating the frequency map of the characters in the input sequence. The time complexity of this operation is $O(n)$, where n is the length of the input sequence.

2. Huffman Tree generation: The tree is built by iteratively picking the two nodes with the lowest frequency and connect them to a parent node. In this way, the characters with higher frequency are matched with a node closer to the root of the tree and consequently in the encoding it will result in shorter code. The most appropriate data structure to support this step is a priority queue, which allows to retrieve the nodes with the lowest frequency in $O(1)$ time and to insert a new node in $O(log(m))$ time. Therefore, the time complexity of this step is $O(mlog(m))$, where m is the number of different characters in the input sequence. Since we are dealing with ASCII characters, m is at most 256 and therefore the time complexity is constant.

3. Code generation: This phase saves the huffman codes produced by the previous step in a hashmap. It is done to avoid traversing the Huffman tree for every character in the input sequence and providing fast access to the codes. The time complexity of this step is $O(m)$.

4. Encoding: The last step is the actual encoding of the input sequence. This is done by traversing the input sequence and for every character, appending the corresponding code to the output sequence. Again The time complexity of this step is $O(n)$.

We must also consider the time of the the operations for reading and writing the file. As we can see already from the time complexities, the most expensive operations are the frequency counting and encoding steps. For large input sequences, we can expect them to become primary sources of bottlenecks, together with the I/O operations. In the following sections, I will describe the parallelization strategies adopted to overcome these bottlenecks.

2 Implementation

This project employs the template design pattern. The primary class, **huffman_base**, outlines the algorithm's structure. This base class specifies the

sequence of function executions, where the parallel functions are defined as pure virtual functions. It also incorporates boilerplate code for file reading and writing (binary) files, thereby eliminating code duplication. For each variant of the algorithm, a distinct class has been created: **huffman_sequential**, **huffman_thread**, and **huffman_ff**. This design pattern aligns naturally with the project's requirements, as the same algorithm needs to be implemented in different ways. Furthermore, it provides a straightforward method for implementing new (parallel) versions of the algorithm. The functions to be specialized are:

- `virtual encoded_data* encode_string(`
  `std::unordered_map<char, std::vector<bool>*>& codes,`
  `std::string &text) = 0;`
- `virtual std::unordered_map<char, unsigned int> count_frequency(`
  `std::string &text) = 0;`

2.1 Data structures

2.1.1 Table codes

The codes of the characters are stored in a hashmap of type

```
std::unordered_map<char, std::vector<bool>*>
```

The choice of a vector of bools to represent the codes is dictated by its space efficiency, since it uses a single bit for each element as opposed to a char which uses 8 bits for each one. The purpose of the pointers instead is to avoid unnecessary data copying or movement during the encoding phase.

2.1.2 Encoded sequence

For storing the encoded sequence I used a data structure with the following type

```
typedef std::vector<std::vector<std::vector<bool>*>*> encoded_data;
```

The inner vector represents a pointer to the encoding of a character. The middle vector represents a part of the input sequence encoded by a single worker. Finally, the outer vector holds a collection of these chunks which represent the whole encoded sequence. One advantage of this data structure is to avoid the reduce phase. During the write phase, all the produced codes must be traversed one by one, regardless of them being concatenated in a single vector or split into chunks. In such a manner we avoid the non-negligible overhead of concatenating the subsequences. The sequential version implementation doesn't not require particular adaptions as it will produce a vector composed of a single chunk (the whole sequence encoded). This datastructure provides also advantages in terms of memory access. In a NUMA system (the reference architecture for this project), each processor (or core) has its own local memory, and accessing this

local memory is faster than accessing memory that is local to another processor. Having a small vector also enhances the chances of the data being stored in the some level of cache.

2.2 Thread version

2.2.1 Frequency counting

To parallelize the frequency counting, I adopted a standard map-reduce pattern. Every thread is assigned statically a portion of the input sequence to compute a partial frequency map. When all the threads have finished their task and are joined together, the produced partial frequency maps are merged together to obtain the final frequency map. The reduce phase is not parallelized: In our particular scenario, given that we are working with a maximum of 256 characters (ASCII), this phase entails the summation of up to 256 integers from each mapper. This operation is relatively lightweight and the benefits of parallelization are negligible or might even be detrimental.

2.2.2 Encoding phase

Building upon the previously described data structure, this phase employs a standard map pattern. The input sequence is statically divided into chunks, with each thread independently processing a chunk to produce encoded subsequences. Each worker identifies its index and calculates the encoding for every character within its assigned segment of the sequence. This is accomplished by appending the corresponding character's pointer to the appropriate chunk. Once this process is complete, the worker returns the encoded sequence to the invoking thread.

2.3 FastFlow version

The FastFlow implementation is similar to the previous one and implements the same patterns described before, in an high level manner.

2.3.1 Frequency counting

FastFlow provides naively high level data parallel patterns. It was simple to implement the parallel-reduce function taking advantage of the ff::ParallelForReduce class and the parallel_reduce method. Specifically, two primary lambda functions were defined: one for mapping and another for reducing. The mapping lambda function is responsible for incrementing the count for the character under examination. The reducing function, on the other hand, handles the merging of results.

2.3.2 Encoding phase

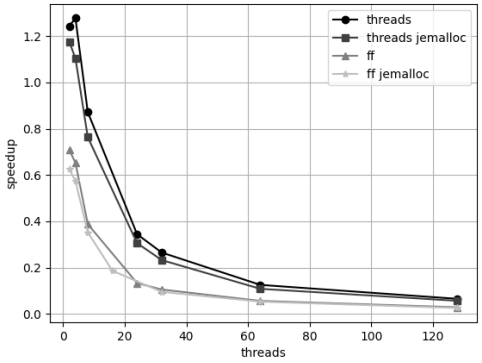
The implementation of this phase utilizes the ParallelFor construct of FastFlow. This approach was chosen both for its simplicity and have a sort of fair comparison with the threaded version. The body of the parallel for is essentially identical to that in the threaded version. An alternative implementation could have utilized the **ff_Farm** pattern. This would have necessitated the definition of an Emitter as an instance of **ff_monode t**. This instance would create an object of type **Task** containing details about the portion of the sequence to be encoded by a worker. Subsequently, A **Collector** object would collect the chunks produced by the workers into the final result. This approach requires a certain amount more of code to be written. Moreover, The ParallelFor construct is based on the farm building block, therefore the performances should be comparable.

3 Benchmark

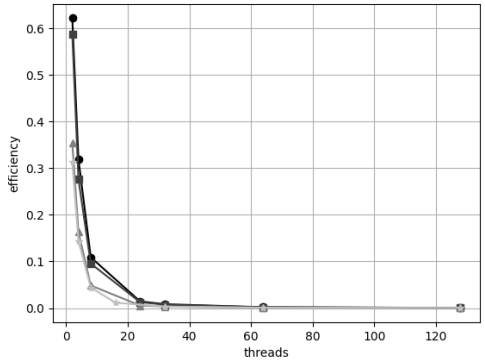
This section presents the results from the conducted tests. The tests were executed on a dual-socket NUMA AMD EPYC 7301 machine, equipped with 16 cores and 32 threads each, totaling 64 hardware threads. For each thread count, the execution time was measured five times, and the average time was calculated. Each version of the Huffman algorithm was tested in conjunction with the **jemalloc** library. The application is compiled with the -O3 flag.

3.1 50kb file

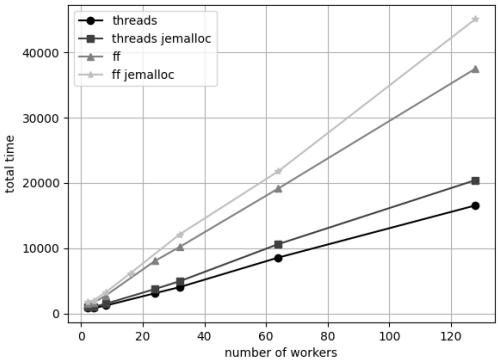
Theoretically, the speedup on such a lightweight file should be very low. The obtained results depicted in figure 1 in fact confirm this hypothesis: The overhead derived from the setup of the parallelization overcomes the sequential execution time. With the thread version, we can observe that only up to 2 threads a small speedup is achieved (≈ 1.2). Then, the execution time increases as the number of threads increases. With FastFlow there is not even a speedup with this file. The introduced overheads for setting up the parallelization and moving data around leads to a performance decrease with respect to the sequential version.



(a) speedup with 50 kb file (no I/O)



(b) efficiency with 50 kb file (no I/O)

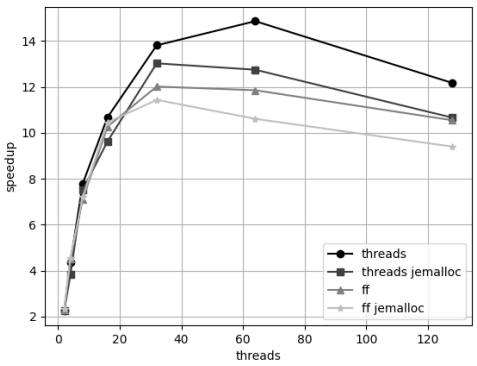


(c) execution time (μs) with 50 kb file (no I/O)

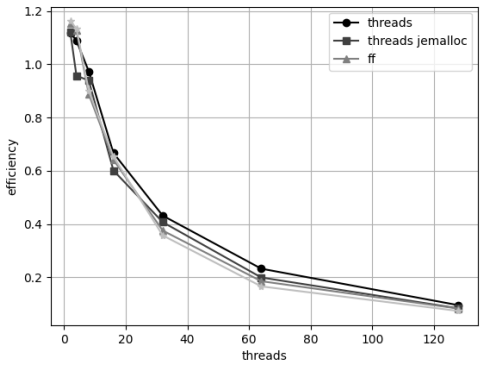
Figure 1: Metrics for 50 kb file

3.2 50MB file

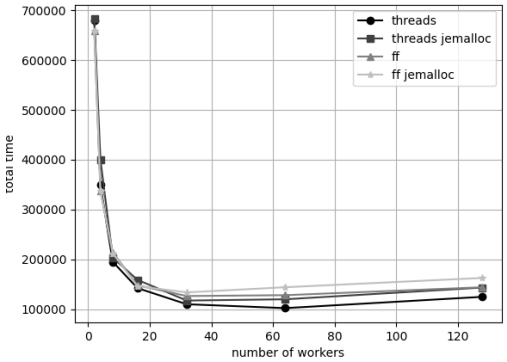
With a reasonable large file size, the benefits provided by the parallelization emerge. We observe in figure 2 that the threads version reaches a peak speedup of ≈ 15 with 64 threads, whereas the FastFlow version obtains a speedup of ≈ 12 with 32 threads. The observed smaller speedup might be a result of FastFlow's communication channels managing increasingly granular computations as the number of workers escalates. Moreover, this graph has been produced with the flag **NO_DEFAULT_MAPPING** compile option, which disables the default thread mapping provided by FastFlow and put the OS in charge to perform it. With this option, it might be the case that the FastFlow nodes are moved from one core to another during the execution and this is a additional overhead since a context switch in envolved. We can observe that the jemalloc library doesn't provide concrete benefits and starting from a certain number of threads



(a) speedup with 50 MB file (no I/O)



(b) efficiency with 50 MB file (no I/O)



(c) execution time (μs) with 50 MB file (no I/O)

Figure 2: Metrics for 50 MB file

it even worsen the performance. As forecasted, the I/O operations turned out to be very expensive and if we consider the total time accounting for them, the speedup drops significantly. Example metrics are shown in table 1.

Implementation	Workers	Time (I/0)	Time (no I/O)
Sequential	-	2748534	1645368
Threads	64	1192910	99086
FastFlow	32	1207970	131287

Table 1: Example measurements for 50mb file

3.3 50MB file - default thread mapping

The performance tests has been conducted also using the default thread mapping provided by FastFlow. We observe in figure Figure 3a that the FastFlow implementation experiences a significant decrease in speedup beyond 16 threads. This suggests that the default thread mapping may not be optimally configured for NUMA architecture, leading to inefficient memory access patterns and subsequently to a degradation in performance. NUMA architectures have multiple memory nodes with varying access speeds depending on the proximity to the processing unit. The observed performance drop could be due to thread contention or memory bandwidth saturation when multiple threads access non-local memory simultaneously. Notably, the integration of the jemalloc memory allocator yields a substantial improvement, with the speedup exceeding that of the standard threaded version. The enhanced performance is likely attributable to jemalloc's NUMA-awareness, which enables it to allocate memory closer to the executing threads, reducing latency and increasing memory access speed.

4 Conclusions

This project has provided a comprehensive overview of the various aspects involved in parallel programming. Memory management plays a crucial role in the performance of a parallel application, and understanding the underlying architecture of the machine executing the application is equally important. The results obtained confirm that I/O operations can significantly impact the performance of Huffman encoding. Future work could focus on enhancing the performance of these I/O operations. However, it's important to note that their performance is also largely dependent on the underlying storage system.

5 Compilation

It is possible to compile the project by running the following commands:

```
cmake .
make
```

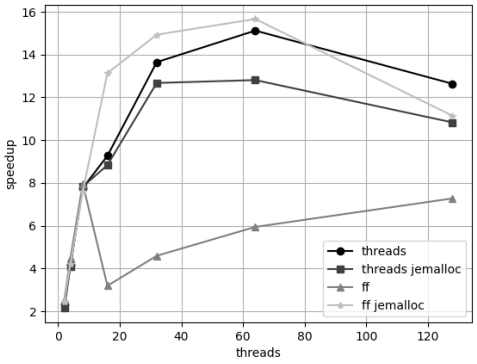
and then the program can be used as follows:

```
./HuffmanProject <input_file> <output_file> <seq|t|ff> <n_threads>
```

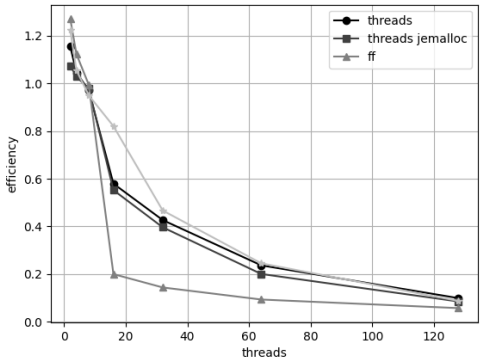
The script for running the benchmarks is **measurements.sh**. It can be executed as follows:

```
./measurements.sh <file> <jemalloc\_path>
```

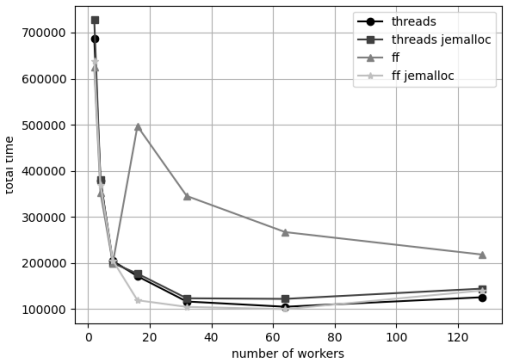
the benchmarks are saved in the **measurements** folder. If you want to test the correctness of the encoding you should uncomment the *add_definitions(-DDECODE)* option in the CMakeLists.txt file and recompile the project. The Plot script is **plot.py** and can be executed as follows:



(a) speedup with 50 MB file (no I/O)



(b) efficiency with 50 MB file (no I/O)



(c) execution time (μs) with 50 MB file (no I/O)

Figure 3: Metrics for 50 MB file with default thread mapping

```
python3 plot.py
```

The resulting figures are saved in the **figures** folder. You should install the matplotlib, numpy and pandas libraries to run the script.