Parallel Implementation of Huffman Coding

Federico Matteoni

A.A. 2022/23

1 The Alternatives

Huffman coding may be implemented in various ways, each with its own approach. At its core, Huffman coding relies on a symbol occurrence table, which captures the frequency of each symbol in the input text. The primary objective is to assign a distinct sequence of bits to each symbol in the most efficient manner possible. This entails ensuring that the encoding for each character possesses a unique prefix, eliminating any ambiguity when decoding. To accomplish this, the characters are organized within a binary tree structure. By assigning a 0 or 1 value to each subtree, the code for each character is determined by traversing the tree from the root to the leaf, concatenating the encountered 0s and 1s.

Greedy One of the first approaches that can be considered is the greedy approach. We begin with a binary tree rooted in a fake node, where each of the n characters is contained in a separate subtree. The goal is to iteratively combine pairs of nodes with the smallest weights, creating a new tree where the weight of the root node is the sum of the weights of the merged nodes. This process loops until we obtain a complete binary tree.
To implement this approach, we first need to sort the characters based on their frequencies. This sorting operation requires $O(n \cdot \log n)$ time, where n is the number of unique characters. Then, we perform $n - 1$ iterations of merging pairs of nodes and updating the overall tree. Therefore, the overall time complexity of this approach is $O(n \cdot \log n)$.

Top-Down Another approach to building the Huffman tree is a top-down approach. This method considers all possible ways of dividing the characters into two subtrees. Once a division is chosen, the process is repeated recursively for each of the two subtrees.
This approach has a significant drawback in terms of time complexity. Since we have a binary decision for each character, the number of possible combinations grows exponentially with the number of characters. Specifically, for n unique characters, the time complexity becomes $O(2^n)$. As a result, this approach is not feasible for larger values of n as it quickly becomes computationally expensive.

Heap The selected implementation of Huffman coding utilizes a heap data structure to store and encode the Huffman tree. This choice allows us to leverage well-known and optimized algorithms for heap operations. Traversing the heap, an operation involved in finding the nodes with the smallest weights and in inserting new nodes, has a time complexity of $O(n \cdot \log n)$.
One significant advantage of using a heap-based approach is that we utilize a single tree throughout the encoding process. In contrast, the greedy approach, as discussed earlier, requires additional steps to merge nodes and update the overall tree, requiring more passages along the tree with each step. These extra operations introduce overhead and can potentially impact the efficiency of the encoding process.
By utilizing the heap data structure, we can efficiently handle node operations and achieve a time complexity of $O(n \cdot \log n)$.

2 Implementation

The initial phase of the project focuses on implementing the sequential program. This approach allows for the development and testing of essential functionalities such as the heap logic, data structures, node insertion logic, encoding generation and file reading/writing.
The sequential approach is beneficial in several ways. First, it helps identifying and debugging any issues in the heap implementation.
Additionally, the sequential implementation allows for the validation of the generated encoding. By comparing the encoded output with the expected results, we can ensure the correctness of the encoding process, and any errors wouldn't be attributed to an incorrect orchetration of the parallel computation.
The sequential implementation phase serves as a crucial step in developing and testing the fundamental components of the project, ensuring their correctness, functionality, and correct memory usage.

Data The remote machine's powerful components provide an advantage in terms of execution speed, even for slower or less efficient implementations. This means that the differences in runtime between the three implementations may not be significant when processing small files. The testing data includes paragraphs of "Lorem Ipsum Dolor" in a few variations. These variations are designed to test different aspects of the implementation, such as handling uppercase-lowercase letters and symbols. Additionally, a final testing file called "`longfile`" has been created by copying and pasting the paragraphs multiple times. This was done until the sequential approach took a few seconds to complete its computation without any compiler optimizations (without the "`-O3`" flag).
The "longfile" serves as a suitable benchmark for performance evaluation. It allows to observe and compare the differences between optimized and non-optimized binaries, as well as the variations between sequential and parallel code execution. By using this file, the impact of different optimization techniques and parallelization can be better appreciated and analyzed.

Memory Usage During the implementation of all three approaches (sequential, `pthread`-based, and fastflow-based), one significant obstacle that was encountered was memory management. Initially, a traditional C++99 style approach was adopted, which involved heavy usage of `malloc`s and `free`s for variables and arrays declarations. However, this approach proved to be error-prone due to the complexity of both the implementation and the flow of data between modules.
To overcome this challenge, an early decision was made to migrate towards more modern data structures such as `std::vector` and smart pointers (`shared_ptr` and `unique_ptr`). By leveraging the constructors and destructors provided by these data structures, memory management became more streamlined and reliable.
However, it's important to note that there were still some areas that required explicit memory management, specifically related to FastFlow's pointers. To ensure the correctness of memory usage, extensive testing and validation were conducted using the `valgrind` utility with the `--leak-check=full` setting. This allowed for thorough analysis and detection of any memory leaks or issues. usage in the implemented approaches.

Code Reuse One notable aspect of the presented implementations is the emphasis on code reuse. In parallel programming, a key principle is to leverage existing sequential code as much as possible, exploiting the optimizations and ideas that have been developed over the years to solve a particular problem. While this is certainly not the case for this project, as all the code has been written from scratch, an attempt was made to emulate this approach by organizing the code into a small library (the `huffman.cpp` file) and reusing it extensively.
By aggregating functionalities such as file reading/writing, heap building, encoding generation and so on in a single file that was thoroughly implemented and tested, the implementation process became more efficient and time-saving. The only additional code required was an efficient orchestration of the parallel computation, which made use of the pre-existing and validated sequential code.
This approach of code reuse, although challenging, proved to be highly beneficial. It allowed for the utilization of a solid foundation of well-tested code, reducing the need to re-implement and re-validate various components of the system. By focusing on the parallelization aspect and integrating it seamlessly with the existing sequential codebase, the project could efficiently harness the advantages of parallel computation while leveraging the benefits of code reuse.
While the approach of code reuse and leveraging existing sequential code has its benefits, it does come with certain limitations. One of the downsides is that it may hinder the exploration and implementation of specific optimizations that are better suited for parallel computing systems. In the context of this project, it is possible that certain optimizations, such as an extensive rewrite of the encoding-generation algorithm, were not pursued to their full potential due to the focus on code reuse and integrating with the existing sequential code. Parallel computing systems often require a different approach and algorithmic design to fully exploit their capabilities, and this may involve significant changes to the code structure and logic.

Parallel Flow A key aspect in the project was efficiently orchestrating the parallel computation for the Huffman encoding algorithm. This task proved to be particularly difficult due to the limited experience in designing parallel applications. However, the development of this project served as a valuable learning experience and provided an opportunity to acquire skills that will be beneficial in future projects and implementations, especially exploiting modern hardware architectures.
The parallel computation was approached by starting with the sequential implementation and identifying the parts of the algorithm that could be parallelized. This process involved considering various parallel computation patterns introduced during the course and assessing their applicability to the algorithm. The team focused on the patterns that showed the most promise in terms of ease of implementation and potential for significant computational speedup. These selected patterns were then thoroughly analyzed, their implementation designed, and then applied in the project.

3 Evaluation and Discussion

The execution times were computed by averaging the results of 10 runs over a file containing 9,292,657 characters, with 39 unique characters. With a time complexity of $O(n \cdot \log n)$ for the implementation, we can estimate the overall time required for execution to be approximately 206.13τ, where τ represents the time required to perform a single step dominated by the insertion in the heap.
To estimate τ, some early experiments were conducted. These experiments involved adding a number of fake data from a file to the heap and generating the encoding, measuring the time taken averaging over the number of data inserted. Based on these experiments, it was determined that a single step takes approximately 2000 μsecs, considering all the necessary operations from reading to code computation. Therefore, the expected runtime for the sequential implementation is estimated to be around 412,260 μsecs not including file I/O.
We can estimate the speedup and the runtime of both parallel implementations. Considering that the computation does not increase with the number of workers involved, we are in a strong scaling scenario and we can use Amdhal's Law to estimate the speedup. Considering a $T_{seq} = 412260$ and a serial fraction of $f = 0.00055$, corresponding to the portion of the algorithm that has not been parallelized (experimentally obtained by measuring the runtime required by the three portions of the sequential implementation), we can estimate a maximum speedup of 1818. This result has not being achieve by the implementations, but this should not come as a surprise given that the formula ignores all possible sources of overheads and could be seen as reducing the time required for computing the non-serial parts to 0. With the estimated T_{seq}, this means a parallel computation that only takes the time to compute the circa 260 μsecs estimated to be required by the serial part, which is not realistic.
After performing the experiments on the remote machine, the final results are presented in Table 1 and Table 2, which provides information on the measured runtimes. The difference of circa 30,000 μsecs given the 9M characters is, as expected, constant among the implementations.

	Sequential	Threads ($nw = 32$)	FastFlow ($nw = 10$)
Mean (μsec)	763530	409472	485303
Speedup	—	1.865	1.573

Table 1: Results of 10 runs over 9M characters with file read/write

	Sequential	Threads ($nw = 32$)	FastFlow ($nw = 10$)
Mean (μsec)	482671	108944	187231
Speedup	—	4.430	2.578

Table 2: Results of 10 runs over 9M characters without file read/write

All the following results are measured without including file read/write times.

nw	1	10	50	100	150	200
Mean (μsec)	555923	146437	150015	141269	163433	146340
Scalability	—	3.769	3.970	3.935	3.401	3.799

Table 3: Scalability of the `pthread`s implementation on few examples of workers

nw	1	10	50	100	150	200
Mean (μsec)	613299	187231	479001	786374	1012581	1194592
Scalability	—	3.276	1.280	0.779	0.606	0.513

Table 4: Scalability of the FastFlow implementation on few examples of workers

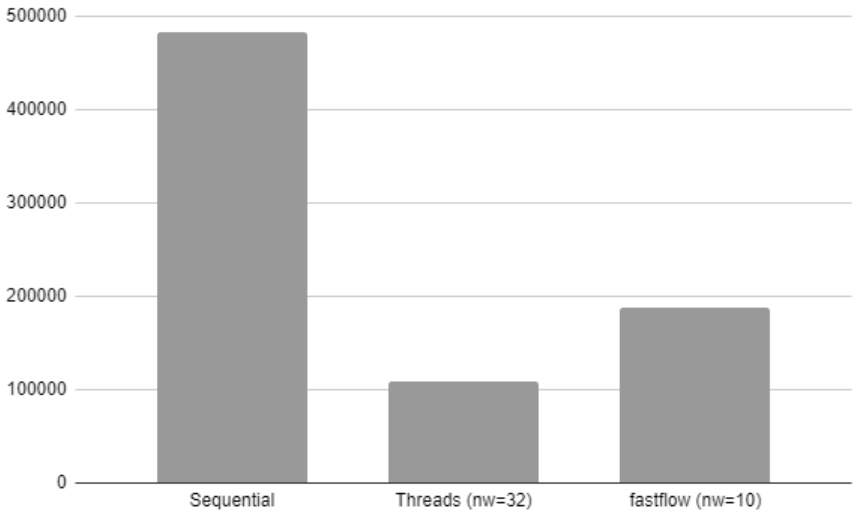

Figure 1: Comparison between the average runtime of the three implementations without file read/write

By generating a plot that showcases the average runtime of an implementation in relation to the number of workers employed, we can obtain significant insights into scalability and efficiency.

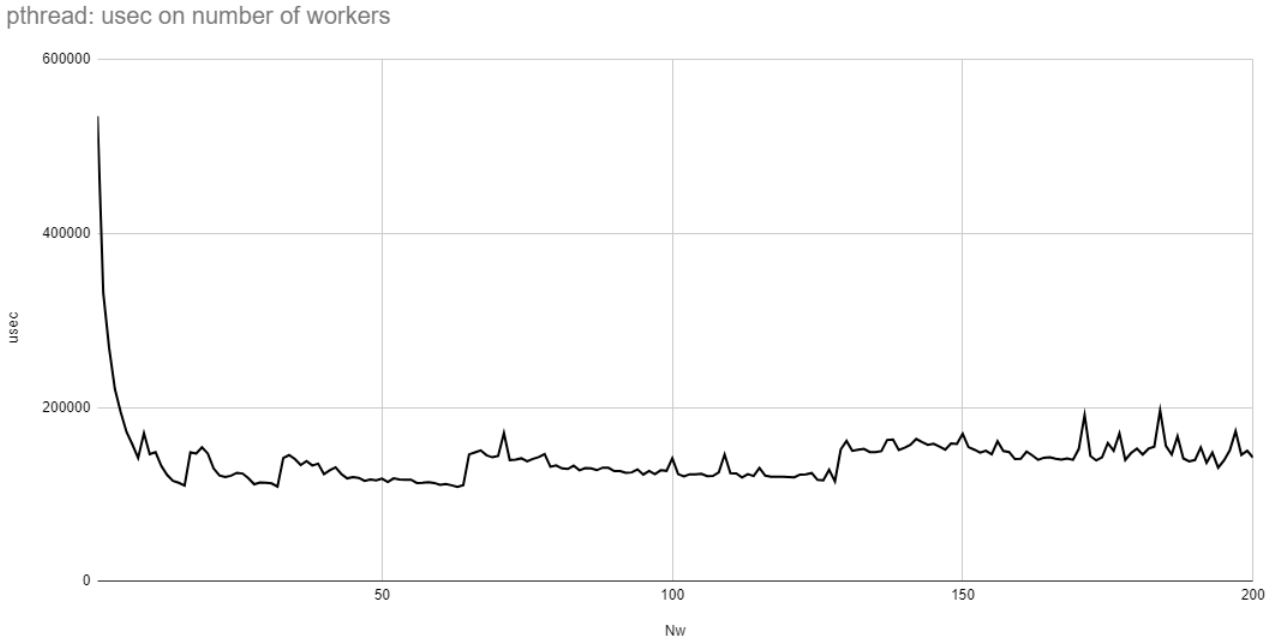

Figure 2: Average execution time of 10 runs over 9M characters per number of workers for the `pthread` implementation.

`pthreads` Analyzing the plot depicted in Figure 2 for the `pthreads` implementation, it becomes apparent that the runtime remains relatively stable once the number of workers surpasses a certain threshold (circa 10 workers). This observation suggests that the overhead introduced by data transfers between workers and between pipeline stages has a minimal impact, thereby indicating efficient handling of data within the system. This is further demonstrated by the Table 3, showing a scalability value that's comparable across the sampled number of workers
The implementation of `pthreads` is straightforward, involving a fixed-size thread pool and logic for distributing data to and collecting data from the workers in the thread pool. Synchronization mechanisms such as mutexes are employed to ensure proper handling of thread access to shared data, specifically the text portions and heap access. However, the scope of each mutex is minimal, and as indicated by the plot, their impact is negligible, at least until the number of workers reaches 200.
The plot reveals a notable pattern characterized by sudden runtime increases occurring every few tens of workers. Although initially appearing random, these jumps consistently occurred at specific worker counts across all experimental runs. The transitions consistently took place from 8 to 9 workers ($2^3 \rightarrow 2^3 + 1$), 16 to 17 workers ($2^4 \rightarrow 2^4 + 1$), 32 to 33 workers ($2^5 \rightarrow 2^5 + 1$), 64 to 65 workers ($2^6 \rightarrow 2^6 + 1$), and 128 to 129 workers ($2^7 \rightarrow 2^7 + 1$). It is reasonable to anticipate a similar jump between 256 and 257 workers and subsequent counts following the same pattern.
Considering that these jumps occur precisely around powers of two (2^n workers) and that we used the compiler optimization flag "`-O3`" enabling various optimization techniques, it suggests that the optimization methods impact how the system manages threads. It appears that the system utilizes threads more efficiently when employing an exact power of two workers, resulting in improved performance. Conversely, when the worker count falls between $2^{n-1} + 1$ and $2^n - 1$, the runtime performance is slightly compromised, albeit decreasing monotonically on average.
Based on the observed characteristics of the implementation, it is advisable to design a system that utilizes the proposed implementation with a number of workers equal to a power of two. This choice aligns with the fact that the runtime exhibits local minima when the number of workers is a power of two, suggesting that the compiler optimizations are most effectively utilized in such scenarios. By leveraging this knowledge, we can achieve a more efficient utilization of resources and potentially enhance the overall performance of the system.

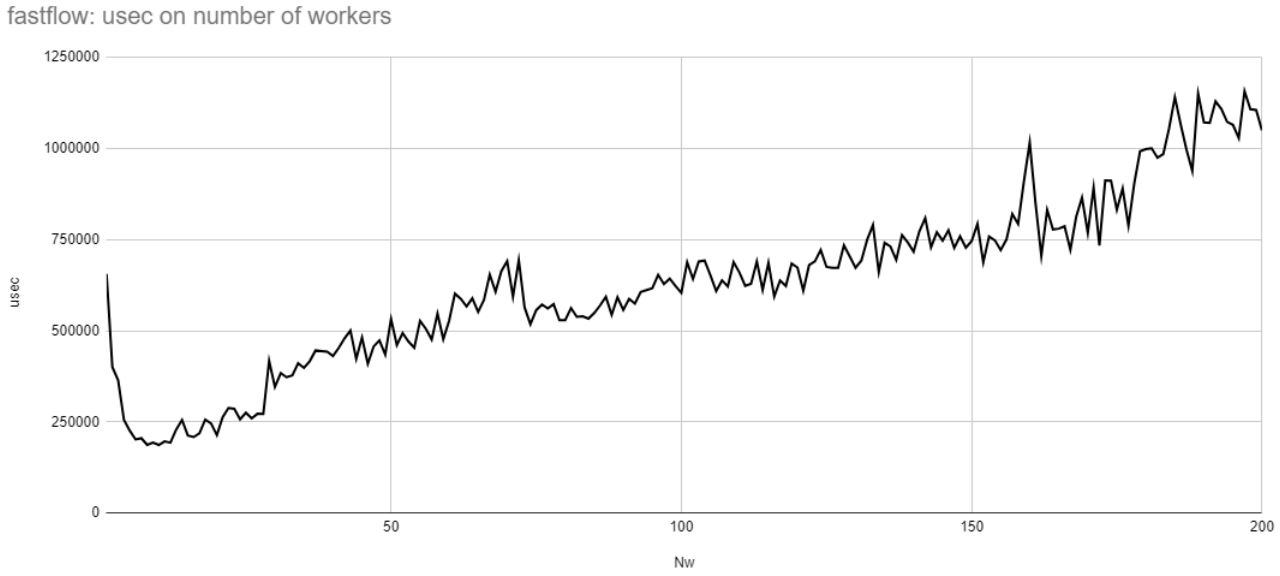

Figure 3: Average execution time of 10 runs over 9M characters per number of workers for the FastFlow implementation.

FastFlow The plot in Figure 3 clearly demonstrates the key difference between the two implementations: the increasing in runtime as the number of workers increases. In the case of the `pthreads` implementation, the runtime remains nearly constant, disregarding the previously mentioned pattern of sharp jumps. On the other hand, the FastFlow implementation exhibits a linear increasing in runtime proportional to the number of workers involved.
The observed linear increase in runtime in the FastFlow implementation can be attributed to the communication overhead between the nodes of the FastFlow farm and pipeline, as well as between the queues and the nodes themselves. These communication processes contribute to the overall runtime and result in the linear increase in runtime.
To further illustrate this behavior, Table 4 provides a numerical representation of the runtime for sampled numbers of workers, with the average runtime recorded. A scalability value less than 1 indicates a degradation in performance,

where the runtime exceeds that of running the same algorithm on a single worker. It is important to consider these scalability characteristics when choosing between the `pthread`s and FastFlow implementations, as they can have a significant impact on the overall performance and efficiency of the system.

Differences between `pthread`s and FastFlow The main distinction between the FastFlow and `pthread` implementations lies in how they handle shared data. FastFlow shares data by passing pointers across queues, enabling data to be transferred between threads with very low overhead. On the other hand, the `pthread` implementation employs a different approach, utilizing `std::vector` containers containing custom data structures. With this design, a single source of data is maintained, and each thread is assigned a specific data structure to write to. As a result, the only synchronization required is for accessing the limited shared variables. Additionally, there is no need to transmit the final results, as the data structures are already accessible within the scope of the other threads and the main thread. In short, while FastFlow needs to copy and transfer the pointers across many different stages, the `pthread` implementation just maintains the original pointers and synchronizes the accesses to the data structures. This approach significantly reduces communication overhead between threads and eliminates the need for data delivery. The linear increase in runtime observed in the FastFlow plot as the number of workers increases can be attributed to this pointer-delivery overhead.

Possible Improvements One potential area for improvement, as mentioned earlier, is to reimplement the logic for generating the character encodings. This would involve developing a custom algorithm specifically designed to fully leverage the parallel architecture, which would necessitate refactoring the existing library of sequential code.
Another opportunity for improvement lies in parallelizing the `for` loops responsible for creating the thread pools in the `pthread` implementation and instantiating the workers in the FastFlow implementation. By introducing parallelization at this stage, the overhead involved in creating and assigning threads could be reduced.

4 Running the Code

The submitted archive contains a `MakeFile` that handles compilation. Make commands for parallel implementations support a parameter named `NW` used to specify the number of workers.

Sequential program: `make seq && ./huffman_seq filename`

`pthread`s implementation: `make threads NW=32 && ./huffman_threads filename`

FastFlow implementation: `make fastflow NW=10 && ./huffman_fastflow filename`