UNIVERSITÀ DI PISA

Huffman Coding parallel implementation

Parallel and distributed systems, paradigms and models course

A.Y. 2022/23

Student:
Luca Miglior - mat. 580671

Lecturers:
Prof. Marco Danelutto
Prof. Patrizio Dazzi

Contents

1 Introduction and theoretical analysis **2**
1.1 The sequential version 2
1.2 Moving towards the parallel version 4

2 Implementation **5**
2.1 C++ Thread version 5
2.1.1 Frequency map generation 5
2.1.2 Encoding sequences 6
2.2 FastFlow implementation 6
2.2.1 Frequency map generation 7
2.2.2 Encoding sequences 7

3 Experiments **7**
3.1 Working on small files 8
3.2 Working on larger files 8

4 Conclusions **11**
4.1 Compilation and Execution 11

1 Introduction and theoretical analysis

Huffman coding is a common compression technique, used in a wide range of applications. This project's goal is to write a parallel application implementing the Huffman compression algorithm for ASCII text sequences.

1.1 The sequential version

The very first operation to take is to design a concise and sequential version of the algorithm, in order to analyze the most salient points in the perspective of identifying relevant parts that could have benefit of a parallelized implementation. The sequential version of the Huffman Algorithm is basically divided into four main functions. The first operation to be performed on the text file is generating the frequency map of the characters inside the corpus. This is well described in Algorithm 1. Since we have to process each byte in the file, time complexity of this phase is $\mathcal{O}(n)$[1].

Algorithm 1 Huffman Frequency Generation

```
1: function GenerateFrequencyMap(corpus)        ▷ Corpus is ASCII string
2:     map ← new HashMap
3:     for all c ∈ corpus do
4:         map[c] ← map[c] + 1
5:     end for
6:     return map
7: end function
```

After having processed the whole text file, we need to generate the Huffman Tree and corresponding codes for each character: given that we have m distinct characters in the file, we can tell that tree-generation procedure time-complecity is $\mathcal{O}(m \log(m))$, and codes generation has a linear complexity of $\mathcal{O}(m)$. This phases are reported in Algorithms 2 and 3. In our particular case these operations can be considered as constant operations, since $m = 256$ given that we are only considering ASCII files.

Algorithm 2 Huffman Tree Generation

```
1: function GenerateHuffmanTree(map)
2:     queue ← new PriorityQueue
3:     for all e ∈ map do
4:         queue.push(HuffmanNode(e.K, e.V, nil, nil))
5:     end for
6:     while q.size ≠ 1 do
7:         l ← q.top
8:         r ← q.top
9:         top ← HuffmanNode(nil, l.freq + r.freq, l, r)
10:    end while
11:    return queue.top
12: end function
```

[1] Provided that HashMap implementation has $\mathcal{O}(1)$ insertion times

Algorithm 3 Huffman Codes Generation

```
1: function GenerateHuffmanCodes(root)
2:     codes ← newHashMap
3:     queue ← newPriorityQueue
4:     q.push(root)
5:     elem ← q.pop()
6:     while q.notEmpty do
7:         if elem.node.c is nil then
8:             codes[elem.node.c] ← BoolSequence(elem.code)
9:         else if elem.node.left is not nil then
10:            elem.code.push(0)
11:            queue.push(elem)
12:        else if elem.node.right is not nil then
13:            elem.code.push(1)
14:            queue.push(elem)
15:        end if
16:    end while
17:    return codes
18: end function
```

Finally, we generate the actual Huffman encoding, by (again) traversing the ASCII string and replacing each character with its own Huffman code. Complexity of this procedure is clearly $\mathcal{O}(n)$, provided constant access to the codes. The complete encoding is obtained by executing this four functions sequentially, one after each other. To these operation, we have to add two more steps, needed for reading and writing the file out to memory. The total sequential time needed for the sequential version of our code is given by equation 1, where t_{hm}, t_{vec} are the hashmap and vector access times.

$$t_{seq} = t_{IO} + t_{hm} \times n + t_{(tree,codes)} + t_{malloc} + n \times (t_{hm} + t_{vec}) + t_{IO} \tag{1}$$

Note that i specified t_{malloc} which is the time necessary for the allocation of the codes vector onto the heap. Even if we are in a shared memory environment, allocating memory is not a cheap operation, an may lead to huge overhead, especially on large chunks of memory. For this reason, i will show an implementation that will take care of parallelising memory allocation time by the usage of nested data structures.

Algorithm 4 Huffman Encoding Generation

```
1: function Encode(sequence, codes)
2:     encoded ← new Vector
3:     for all c ∈ sequence do
4:         encoded.append(codes[c])
5:     end for
6:     return encoded
7: end function
```

1.2 Moving towards the parallel version

In the previous subsection, we identified two primary bottlenecks in the Huffman Sequential Encoding process: frequency counting and encoding generation. Both of these steps involve a number of sequential operations proportional to the length of the input string. Given this characteristic, I opted to develop parallel versions exclusively for these two functions. Implementing parallel versions for Algorithms 2 and 3 would entail working on excessively fine-grained data segments. The overhead incurred from forking new processes and the associated synchronization methods would overshadow any potential advantages.

For efficiency gains, I exploited standard data-parallel patterns for both the frequency map generation and the encoding phase. Specifically, I employed a map followed by a reduce operation to parallelize the frequency map generation, while the encoding process was accelerated through a conventional map pattern.

It's worth noting that introducing a parallel reduce phase is not a strict necessity in frequency conuting context. Suppose we have nm mappers generating nm maps, each containing up to 256 entries. Given a relatively small nm, on the order of tens of threads, the granularity of the reduce phase becomes too minute to warrant a parallel approach. In essence, this phase would involve summing up to 256 integers from each mapper, which is a lightweight operation, negligibly impacting the program's sequential fraction.

Taking these considerations into account, we can now derive an estimate for parallel execution times using the following equation, where nm represents the number of mappers, ne stands for the number of encoders, and n denotes the sequence size. As we are operating in a shared memory environment, we can omit, without loss of generality, the time needed for splitting the workload.

$$t_{par} = t_r + t_{fork} + \frac{t_{hm} \times n}{nw} + t_{(tree,codes)} + \frac{1}{ne}(t_{malloc} + n \times (t_{hm} + t_{vec})) + t_{join} + t_w \quad (2)$$

According to above equation, it is possible to estimate the serial fraction f of the program, defined as the fraction of (omitting t_{fork}, t_{join}, t_r and t_w for simplicity).

$$f = \frac{\text{seq. ops.}}{\text{par. ops.}} \approx \frac{m\log(m) + nw \times m}{2n} = \frac{m(log(m) + nw)}{2n} \quad (3)$$

Where m is again the cardinality of the set of symbols to encode, nw is the number of workers (we supposed to have the same number of workers for both the counting and encoding phase) n is the number of symbols in the input file. Thus, for sufficiently large problems (e.g. $n \gg nw, m$) our program should be able to achieve good parallel performances in terms of speedup. In particular, according to Gustaffson law, better performance is expected with large files, for large paralellism degreesz. Note that this estimation does not count operations needed to fork and join the threads nor I/O times. In particular, for larger files, sequential reading and writing is expected to be a bottleneck of this implementation. For this reason, the experimental section will take in consideration both IO and no IO execution times.

2 Implementation

This section will illustrate main implementation details, explaining and motivating the critical point of the application. For this project purposes, two main versions of the Huffman Parallel Encoding application were implemented: the first one, from scratch, with C++ threads, and the second one taking advantage of the FastFlow C++ library. Each implementation comes in different flavors, mainly depending on how memory allocation is managed in parallel and how map-reduce phase had been implemented. Since this program heavily relies on parallel memory allocations, the `jemalloc` library was also tested to perform memory allocations. Despite the use the FastFlow and Jemalloc libraries, all the following implementations were realized using the standard `stdlib` library. No external code was used, nor libraries.

2.1 C++ Thread version

As we discussed in Section 1, the C++ implementation mainly focuses on how the frequency generation function and the encoding phase is implemented. All the code developed for C++ plain thread implementation may be found under the `HuffmanThread.cpp` file. All the shared functions (especially tree and codes generation) may also be found under the `huffman-commons.cpp` source file.

2.1.1 Frequency map generation

Due to the potential high computational cost, especially when operating on large volumes of data, the efficiency of this function is crucial. To achieve this, we have adopted a standard map-reduce pattern. Specifically, we've implemented two variations: a parallel map phase followed by a single-threaded reduce phase. Although not strictly necessary as per the theoretical analysis, a parallel reduce was tested.

The common starting point for both implementations is the Map phase. After reading and storing the sequence within a string, a user-defined number of threads and an array of partial results (one for each worker) are created. Each thread is responsible for determining the appropriate index to operate on[2]. Subsequently, each mapper processes the sequence character by character, incrementing a counter in the associated HashMap according to its ID.

When all worker threads complete their tasks, the partial results are collected and merged sequentially by summing the individual maps. The commutative and associative nature of summation makes it a suitable reducing function for this purpose.

In the case of the parallel reduce version, the process becomes more intricate. Each mapper computes a partial result hashmap (taking advantege of the map-fusion rule). However, in this scenario, each element is sent to a queue associated with a reducer based on a hash function computed from the key. Consequently, the utilization of locks and synchronization mechanisms becomes necessary to access shared queues and signal reducers about the availability of new data. The mapper threads

[2]It is worth noting that while in a shared memory environment, the necessity of determining the correct index may be less strict, its inclusion aids in reducing the serial bottleneck within the program.

conclude their tasks by sending a pair `<NULL, NULL>` to the reducers. The final step involves merging the results and returning them to the main thread. Notably, this implementation exclusively employs standard C++ data structures. The results are stored in the form of `std::unordered_map` for efficiency, and in the parallel reduce implementation, `std::queue` is utilized to store the reducer queues.

2.1.2 Encoding sequences

Upon generating Huffman Codes sequentially for the given sequence, the results are stored within a `std::unordered_map<char, std::vector<bool>*>`. The choice of using vectors of bools for the Huffman codes is based on their space efficiency. It's worth noting that during the encoding phase, the operations are performed on pointers to codes. This approach ensures high-speed processing, as it primarily involves moving pointers, eliminating unnecessary data copying or movement.

For managing the encoding process efficiently, we utilize a data structure of the type: `vector<vector<vector<bool>* >*>`. Here, the inner vector represents a pointer to the Huffman code for a specific character. The middle vector, serving as a pointer to a chunk, represents the codes that a single encoder worker is handling. Finally, the outer vector holds a collection of these chunks, with the size matching the number of encoders. This structure effectively mirrors a proper Huffman Encoding of the original sequence.

By adopting this approach, memory allocation occurs only as needed and in a parallel manner. The task of reserving memory with `vector.reserve(chunk_size)` is distributed among all workers. Memory management is done manually, ensuring deallocation when no longer required. The program underwent testing with the Valgrind tool, confirming its freedom from memory leaks.

Transitioning to the encoding phase, it adheres to a standard map pattern. In line with the description for the reduce phase, a vector of chunk pointers with a length of `n_encoders` is generated. Each worker determines its index and proceeds to compute the encoding for every character within its segment of the sequence. This is achieved by adding the corresponding character's pointer to the designated chunk. Subsequently, the worker returns this encoding to the calling thread.

This encoding approach avoids the need for result merging during encoding. The encoding data isn't consolidated into a single chunk; instead, it's traversed once during the writing phase. This *lazy* design choice doesn't impact the number of operations required for the writing phase, as the element count remains equivalent even when operating with an "unfolded" encoding structure.

2.2 FastFlow implementation

The FastFlow (FF) implementation is somewhat similar to the previous one, and implements the same patterns described before, but in an high level fashion. To grant full compatibility with the shared functions, and to make a fair comparison among the different versions, the same data structures and considerations illustrated before were used.

2.2.1 Frequency map generation

FastFlow provides naively high level data parallel patterns. In particular, it was easy to implement the parallel-reduce function taking advantage of the `ff::ParallelForReduce` class. In particular, two main lambda functions were specified, one for the mapper, another for the reducer. The mapper lambda takes care of increasing the count for the examined character, and the mapper function manages to merge the result. After calling the `pf.parallel_reduce()` method, the usual map is returned to the main thread, then the tree is built and the codes are generated.

2.2.2 Encoding sequences

For this phase, it has been decided to go for the `ff:ff_Farm` pattern. The farm has been modelled as a three-node block, as described below. In this way it is easier[3] to have a finer control on the operations, especially on the memory allocation phase.

- The first node in the farm is the Emitter, an instance of the `ff::ff_monode_t` class. It will create new Tasks and push them to the other stages.
- A set of Workers, each one operating on the tasks generated by the Encoder: they will partition the sequence and traverse the it, pushing pointers to code into their own chunk.
- A single collector, that will push results on the vector of chunks generated by the workers and will return the result.

Tasks are modelled as a C++ structs; They contain fields (mainly pointers) related to the sequence, the number of encoders (needed by the worker to split the sequence and allocate the chunk) and of course a pointer to the map of the Huffman Codes. Tasks are allocated on the heap by the encoder, and it's Collector's responsibility to deallocate them.

A much simpler implementation is provided with a simple `ParallelFor`, and sequential memory allocation, for the sake of completeness, but experimental results will not be reported.

All files for the actual tested version may be found in `HuffmanFarm.cpp`.

3 Experiments

For the experimental results, only mostly significant results are reported. Execution times were taken by exploiting the `std::chrono` library, taking advantage of the RAII pattern. Probe is implemented in file `utimer.cpp`[4]. Each implementation was tested on a small (64KB) ascii files, and on a large (64MB) one, and for each run several metrics are reported, such as execution times, speedup and efficiency. The latter two metrics are reported not including IO times; moreover, parallel-reduce frequency generation times are only reported in charts, since, as predicted by the theoretical analysis, it did not provide significant improvements. All the metrics

[3] With respect to other higher level FF patterns, like `ParallelFor`

[4] This is the same implementation provided by the lecturers during the course.

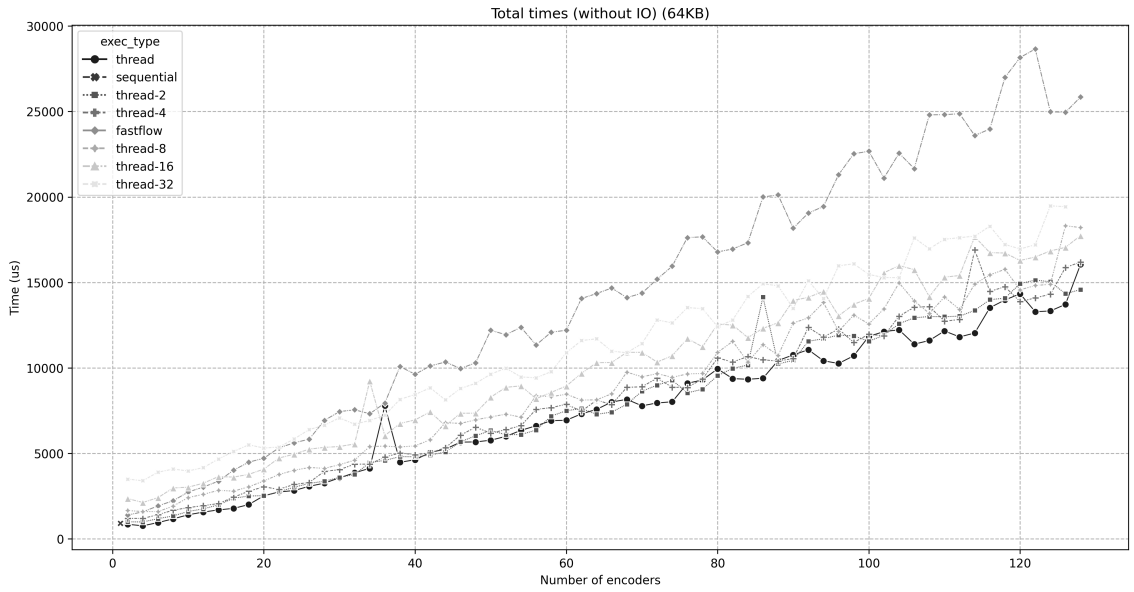

Figure 1: Encoding completion times for 64KB ASCII file.

reported where obtained by compiling an optimized version of the program, using the gcc -O3 flag. Execution times for the non optimized version of the executable file were around 10x times slower. Program was compiled and run on a dual-socket NUMA AMD EPYC 7301 machine, 16 cores/32 threads each, for a total of 64hw threads.

3.1 Working on small files

According to theory, tests on the small ASCII files lead to poor results, while trying to exploit parallelism. In particular, on such small files, it was nearly impossible to achieve any improvement deriving from the parallel implementation. As reported in fig. 1, times increase linearly w.r.t. the number of workers, due to the huge overhead introduced. This behaviour was observed both in the frequency generation phase and in the encoding phase. Table 1 reports schematically the best times obtained.

Implementation	**Time (no IO/IO)**	Workers	**Speedup**	**Efficiency**
C++ Threads	$767/2003\mu s$	2	1.2	0.6
FastFlow	$1397/2001\mu s$	2	0.67	0.3
Sequential	$924/2380\mu s$			

Table 1: Best results on 64KB file.

3.2 Working on larger files

As expected, the efficient utilization of parallelism capabilities became evident when working with a large file. Figure 2 presents the total encoding times for the large file. The reading and writing times remained consistent across all implementations, with mean values of $1.46e5 \pm 1.36e4\mu s$ and $9.29e5 \pm 3.81e4\mu s$ respectively.

Implementation	Workers	Time (no IO) (μs)	Time (IO) (μs)	Speedup
Threads	50	9.78e+4	1.17e+6	13.12
FastFlow	56	1.54e+5	1.22e+6	8.31
Threads (16)	60	9.89e+4	1.22e+6	12.98
Threads (2)	56	9.82e+4	1.13e+6	13.07
Threads (32)	58	9.84e+4	1.13e+6	13.04
Threads (4)	58	9.69e+4	1.17e+6	13.25
Threads (8)	52	1.00e+5	1.18e+6	12.77
Sequential	n.a.	1.28e+6	1.42e+6	n.a.

Table 2: Best results on 64MB file. Notation Threads(xx) refers to the eventual number of parallel reducers used in the encoding phase.

Again, as staded in sec. 1.2 the type of reduction (sequential or parallel) applied during the frequency generation phase demonstrated little to no significant impact on the overall timing. With this implementation, we were able to achieve a peak speedup of approximately **13** using the C++ thread implementation. The speedup results are depicted in Figure 3, which also includes speedups for the `jemalloc` version of the code.

Contrary to our expectations, the `jemalloc` implementation was notably slower than the `malloc` implementation, particularly when dealing with a high degree of parallelism; however, with a smaller number of workers, its behavior was more in line with the standard allocator.

For a concise overview of the best results achieved by each code version, refer to Table 2. It's important to note that the I/O times emerged as a significant bottleneck in this computation, resulting in timings comparable to the sequential version. A more comprehensive examination of efficiency, is presented in Figure 4.

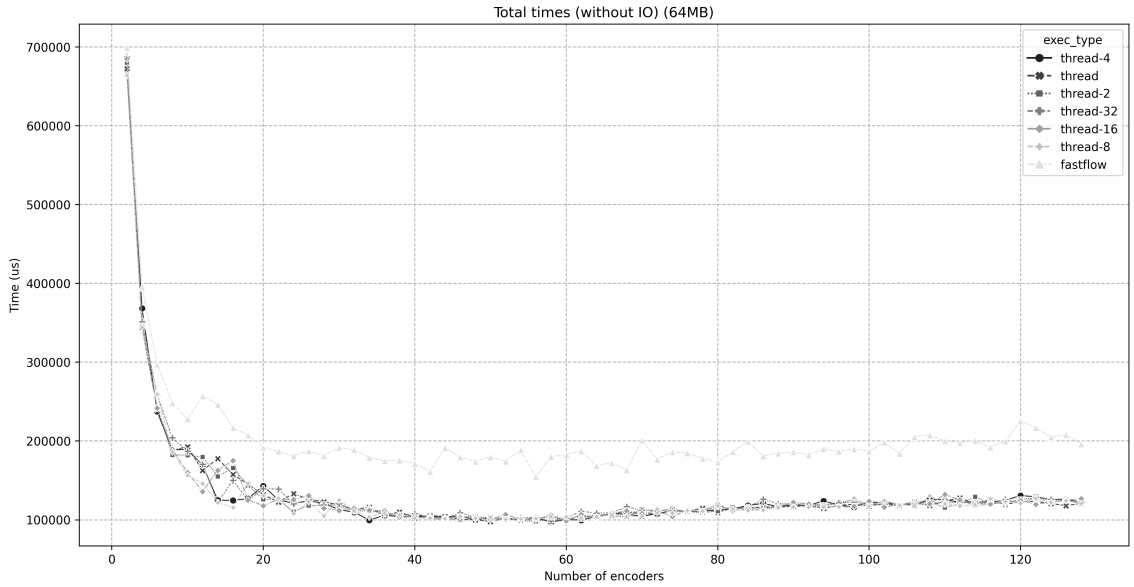

Figure 2: Encoding completion times for 64MB ASCII file, without IO. Notation "thread-xx" refers to the parallelism degree used in the reduce phase.

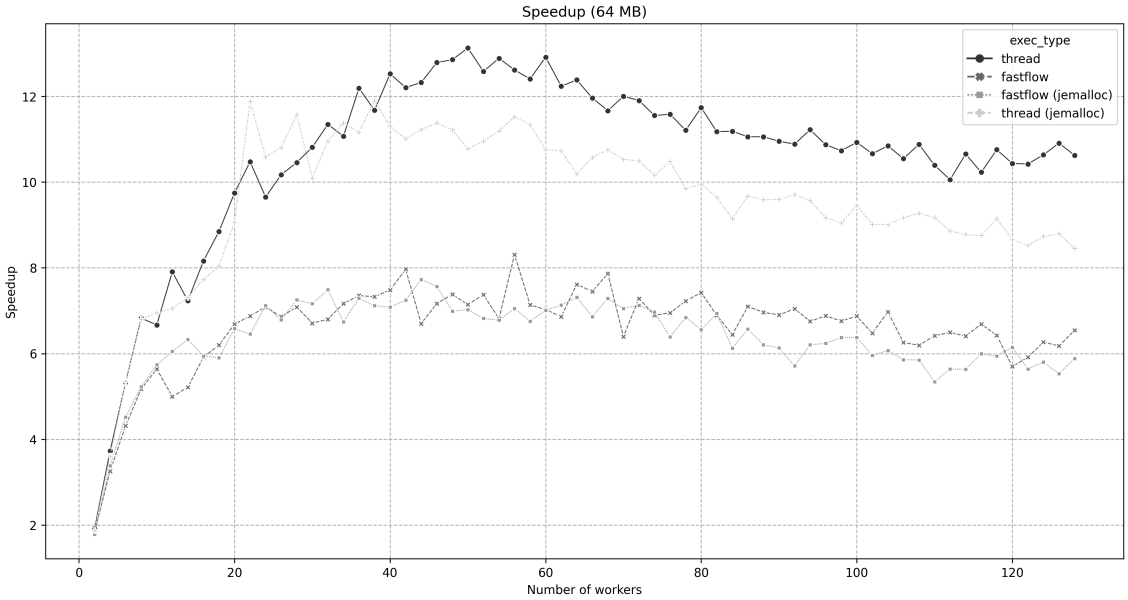

Figure 3: File encoding speedup for 64MB ASCII file, without IO

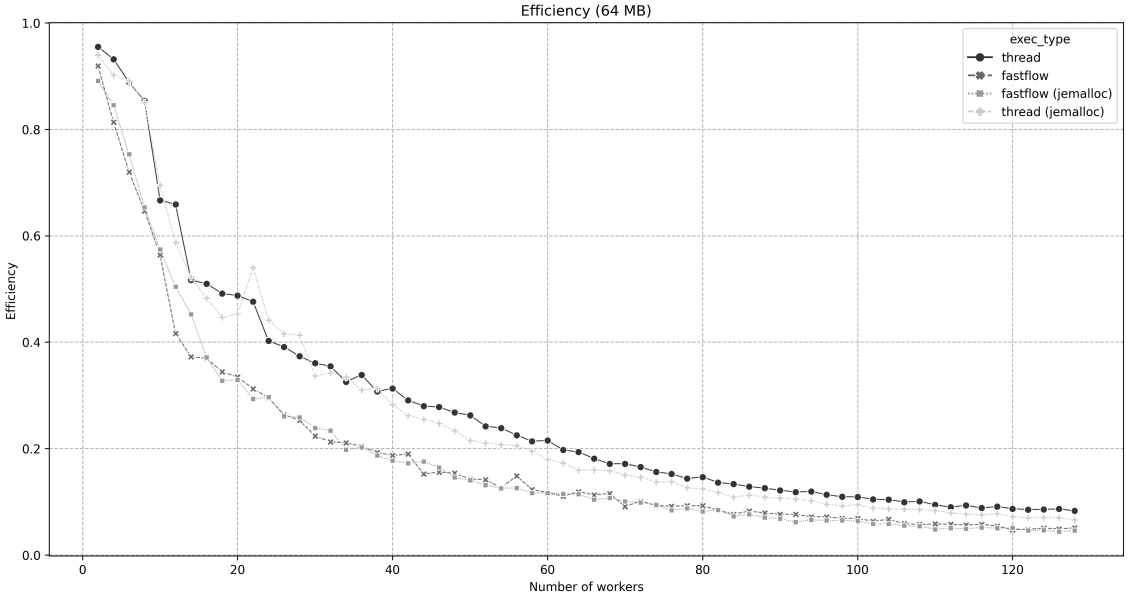

Figure 4: Efficiency of the FastFlow vs thread implementation

4 Conclusions

The experimental phase confirmed the alignment between the anticipated results and the predictions from the theoretical analysis. It is evident that the algorithm's potential for parallelization is constrained by IO operations. Moreover, in this particular case, FastFlow (FF) seemed to perform slightly worse than the thread implementation, especially with a high degree of parallelism. This observed overhead is likely caused by FastFlow communication channels having to deal with finer and finer computations as the number of worker increases. If a word-level implementation were considered, FastFlow's performance would likely reach or exceed the speeds of the thread implementation.

Furthermore, code realized with the FastFlow library is much more concise and expressive than the threaded one, which is a significant advantage in terms of code readability and maintainability and eventual extensivity.

Future advancements could explore the usage of parallel reading and writing functions; however, this largely depends on the underlying operating system (especially system calls and IO buffering) and hardware capabilities. Such alternatives may space from a simple parallel fseek, to more low level techniques such as `mmap` operations. Another avenue for future work lies in creating a parallel version of the decoding phase, which was solely implemented in a sequential manner for experimental purposes and not elaborated on in this report.

4.1 Compilation and Execution

To compile the project, simply execute the `./bld.sh` script. This will trigger CMake to generate Makefiles and build the entire project. For Apple architecture, or cases requiring a custom library path, you may need to specify the fastflow or jemalloc path. If you want to perform encoding correctness check, you may add the `add_definitions(-DCHKFILE)` to the CMakeLists.txt. Running the project is straightforward. Use the following command in the shell:

```
./build/spm_project ./file.txt nm nr ne [seq|map|ff]
```

If you opt for the map running flavor and $nr \neq 0$, the reduce phase will be parallelized based on the specified number of reducers. The command `ff` will execute the Farm version of the fastflow. [5] You can also benchmark the application by running the `./benchmark.sh` script.

[5] The "ParallelFor" version is intended as proof of concept in order to show how is it possible to parallelize with few lines of code with FF. If you want to test it, source files and headers in `main.cpp` have to be replaced to point the parfor implementation, as well as the CMakeLists.txt, but it has not to be intended as an actual part of this project.