Huffman Coding

Final project for the Course of "Parallel and Distributed Systems: Paradigms and Models"

Alberto Dicembre, mat.668377

Introduction

Huffman Coding is an optimal type of prefix method, commonly used for lossless data compression. The purpose of the project is to provide an application for the compression of an ASCII text file using the Huffman Coding method. In particular, three different implementations were provided: a sequential one, and two parallel ones, of which one using native C++ thread mechanisms, and one using FastFlow.

Sequential implementation

1st step: Counting occurrences

Since we need to know the occurrences for every character in the file, the algorithm starts by traversing the input text file and populating a map with its characters and their frequency.

An *std::unordered_map* was used, since we don't need the key-value pairs to be sorted yet.

2nd step: Populating the Priority Queue

We define an std::priority_queue with an std::vector as underlying container and a custom "Compare" class, that allows us to reverse the standard assignment of the priority. In fact, we want the least used characters to be on the top, and not the opposite.

Once done that, we simply iterate over the map and, for every key-value pair, we create an object of type HufNode, which, in addition to the character and its frequency, holds the pointers to the eventual right and left child nodes. The objects are pushed into the queue.

3rd step: Building the Tree

Now that we have a Priority Queue of HufNodes sorted on decreasing frequency, we can begin the construction of the tree, by applying the formerly described algorithm. We will be left with a single node: the Huffman Tree root.

To identify non-leaf nodes, a special character '$' (defined by a Macro in the file "headers/utils.hpp") was used instead of the character value.

4th step: Building the Huffman Codes Table

Once the tree is built, we can construct the table that maps every character to its corresponding code. Starting from the root, the function starts with an empty code string. It calls itself recursively on the left and right child of the current node, appending '0' to the string in the first case, and '1' in the second, until it gets to a leaf node. At that point, the character-string pair is added to a map that will, in the end, contain all the characters and their respective prefix code.

5th step: Encoding the file

It's now time to compress the input file. For every character in the input file, we will retrieve its Huffman Code from the previously built map and write it in a 64-bit buffer (a variable of type uint64_t was used) with a bit-shift operation. Once the buffer is full, we write its value on the output binary file.

Parallel implementation

As we may notice from the algorithm description, the process is not entirely parallelizable. In particular, the Tree construction has to be done in-place and in-order on the priority queue, or the results may be altered. In the same way, writing to the output file has to be done in a sequential way.

On the other hand, though, the tree-building process is not a costly computation: the number of operations depends on the size of the queue (which itself depends on the size of the alphabet), not on the size of the input. If we assume that the size of the alphabet will never be crucially big, it's not even worth to parallelize it (as also supported by the following statistics). The same goes for the Code Table building process.

Figure 1 reports the average execution time of the functions, on a 10MB text file.

What was opted for, in the parallel implementation, was to parallelize the file-reading and consequent occurrences-counting operation, and the (less parallelizable) output file writing process.

This implementation uses a Thread Pool: in this way, we initialize all the threads at the beginning of the program, and we don't need to kill and re-instantiate them continuously during the execution.

Also, we don't use bare collectors anymore, but classes that wrap them with mutexes and condition variables: in this way we guarantee synchronization and integrity of the data.

Even though , throughout the application, threads don't ever return an actual result value, *std::futures* were used, so that we are notified when the threads are finished working and the main execution can continue.

Improving "count_occurrences"

Since the operation on the file is read-only, it can be executed in parallel without issues[1]. Therefore, the first optimization that came to mind was to use a farm for the computation: splitting the file in

[1] Efficacy of parallel reading of a file was only tested on an SSD architecture. On an HDD one, it might perhaps not be as efficient.

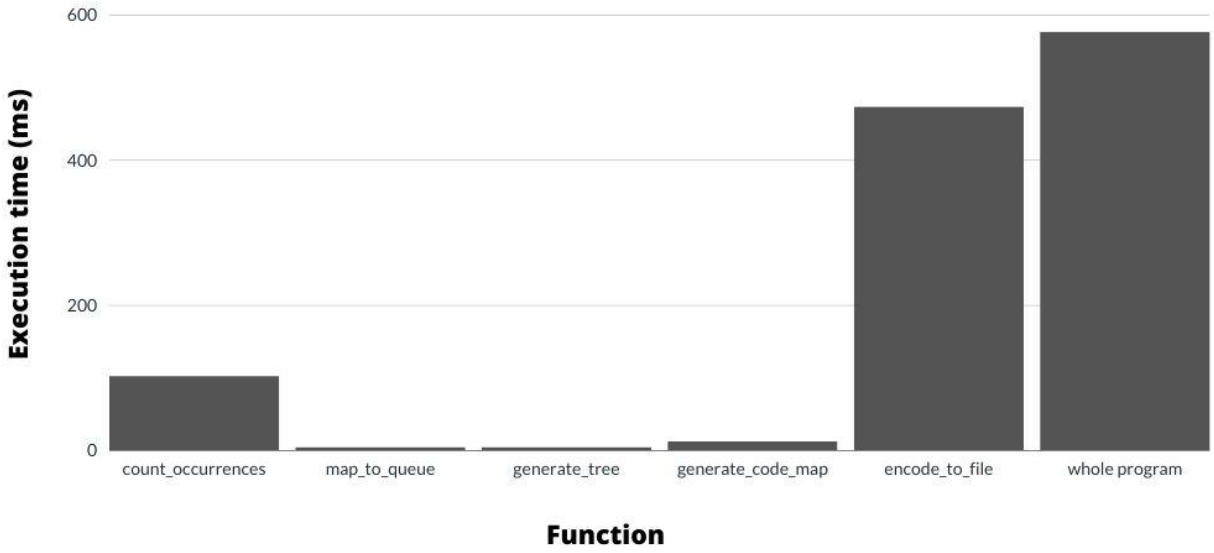


Figure 1: Execution time of the functions over a 10MB file (sequential)

chunks and have different threads processing it, to realize what in fact is a map operation from the file to the occurrence map. Sharing a single destination map, though, would be very slow, since every thread, for every read character, would need to access it. In order to avoid this overhead, every worker now stores the result in a local map. These maps, once the computation has ended, are merged together with a reduction.

The Emitter (actually represented by the main thread) pushes the tasks on the task queue, including: a reference to the file name, a reference to a local occurrence map for the thread to use, the indexes delimiting the chunk of the file to compute.

The workers fill their local maps and, when the computation is completed, the Collector (also an Emitter) pushes the Reduce tasks: we now use the half of the workers and have them compute the sum of two maps. When this is done, the Collector sequentially sums the remaining maps.

Figure 2 shows the diagram of the Map operation. It's an approximate representation, since the Collectors and emitters are actually all the same entity (the main thread).

Improving "encode_to_file"

Differently from reading, writing on a file in parallel is not something doable. Thus, the optimization of this function was achieved throughout a pipeline.

The work was split in three sub-computations:

1. Reading characters from file to a buffer;
2. Translating characters in the buffer to corresponding Huffman Codes;
3. Constructing the bits from the Huffman Codes and writing them into the file.

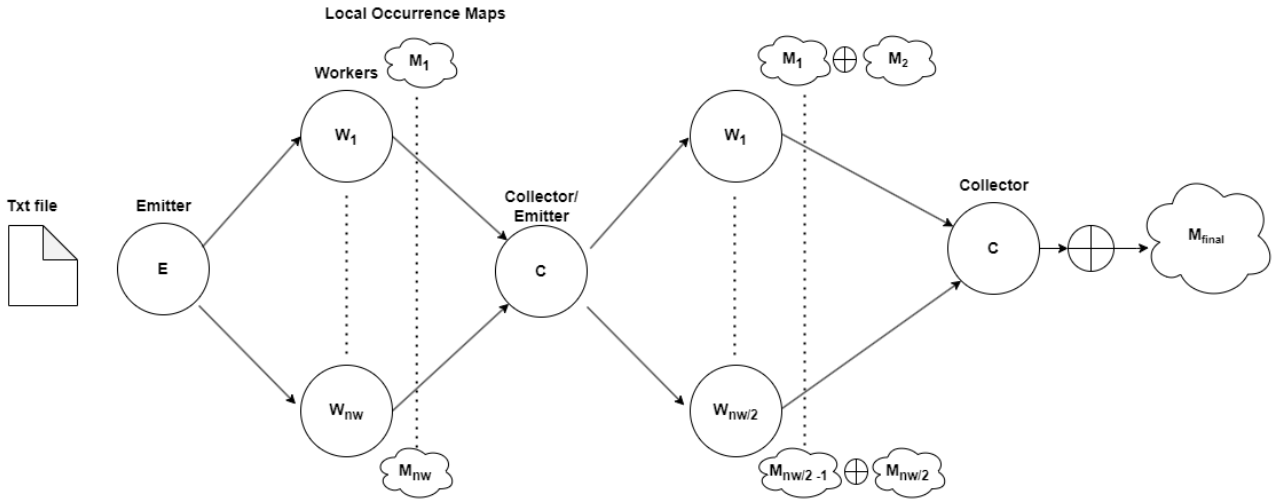


Figure 2: Diagram of the "count_occurrences" farm

In particular:

- The first stage (main thread):
  - Immediately pushes a task for the second stage; the task contains a reference to a shared queue of buffers;
  - reads the file and stores the characters in buffers (of size 4 KB. Changing it to 1/2/8 KB hasn't led to any difference);
  - When a buffer is filled, it is pushed to the buffer queue, and the second stage is notified;
  - When the file ends, an End-Of-Stream character is appended in the buffer and the buffer pushed like the others.
  - Now the thread awaits for the future of the second stage.
- The second stage works like the first one: it uses a queue of buffers to share with the next stage, and, when the computation is ended, it propagates the EOS to the last stage; then, it waits for its future.
- The third stage, when the EOS is received, writes the final byte it was working on (it may not be filled completely, the remaining bits are written as 0s), and then it returns 1. Now the second stage will get the future value, return 1 itself to the main thread and the computation is completed.

Alternative solution

This was the chosen approach; however, an alternative has been considered: we could have used a farm/map as in the first part of the program, to read from the file in parallel, and have each worker build their own local Huffman vector/string, to then merge them together (in order). The (main) downside of this approach is memory consumption. This because, before the translation in bits happening in the last function, the '0' and '1' digits of the Huffman Codes are actually stored as characters (8 bits). This means that, if the workers are transforming every character of the file in their respective Huffman Code, we are increasing the memory consumption of the program by a factor which is the (average) size of the Huffman Code (which, with the tests made during the development, can be up to 6). This means that we would need memory for up to 6 times the size of the file we're processing.

Other than that, we would have additional tweaks and details that would make the improvements smaller compared to the first farm (the output has to be ordered, huffman codes can have different lengths so we would need to handle padding for the incomplete bits, and the fact that the writing to the file must still be sequential).

FastFlow implementation

Tha FastFlow version of the program is very similar to the native thread implementation; only a few details differ:

- To count the occurrences, a *ParallelForReduce* was used. Therefore, we got rid of the "merge_maps" reduction function. All we needed to do was defining a Lambda that describes how to compute the sum of two maps, and pass it to the *parallel_reduce* method. Since this Parallel For has to be executed only once, the "one-shot" version was used, for a smaller overhead.
    - Note: unfortunately a way to make the workers read directly from the file wasn't found. Because of the nature of the *parallel_reduce* operation, instructing the workers to open the file (once, and outside of the loop) and then execute the computation on their assigned chunk, wasn't possible. For this reason, data is transferred (sequentially) on a vector and then processed from there.
- The third pipeline stage was split in two, therefore leading to a 4-stages pipeline, with better performance. This was done because, looking at the pipeline execution stats, it was clear that the last stage was the one consuming more time. Thus, the file writing operation is performed by a separate stage.
- The buffer size was increased to 8KB, since this led to an improvement in performance.

Figure 3 illustrates the pipeline architecture.

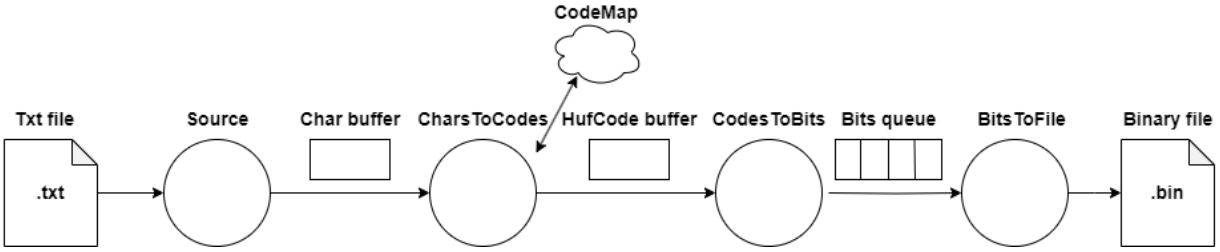


Figure 3: Diagram of the "encode_to_file" pipeline

Performance and Metrics evaluation

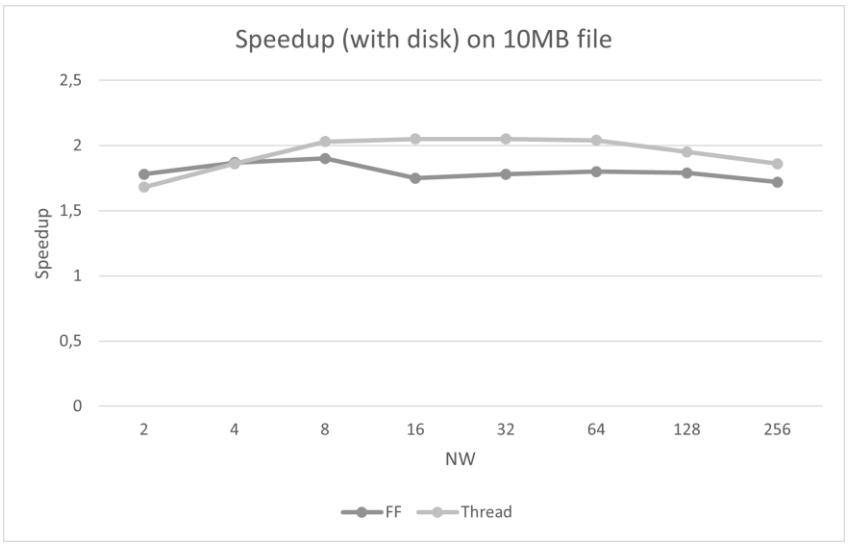
In this section, we will analyze the performance of the algorithm, showcasing the differences between the three implementations. Moreover, every observation will regard two distinct version of the programs: one including read and write operations from disk, and one ignoring it. To achieve this, we created copies of the sequential and thread implementations, which don't read directly

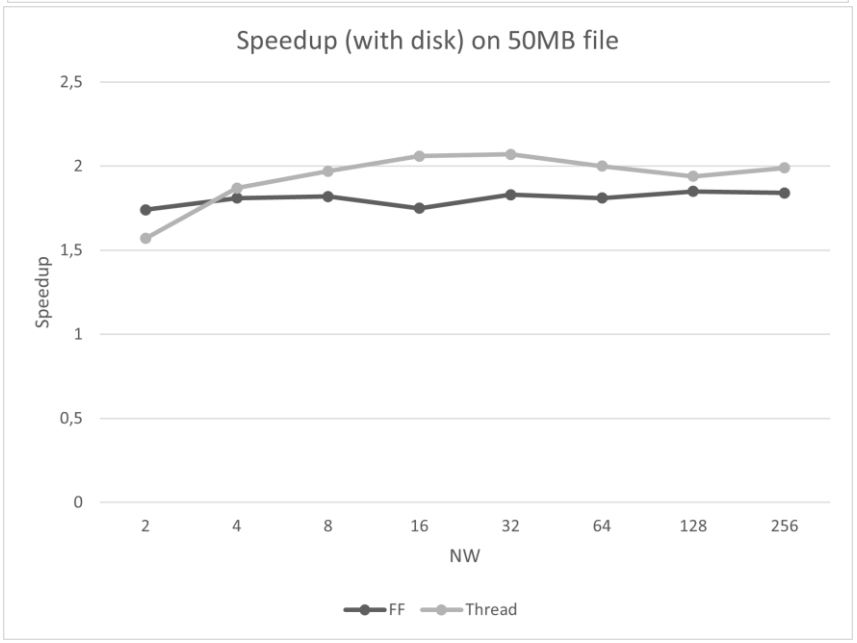
from disk but operate on data contained in a vector (as we already do in the FastFlow one). In this way we are able to analyze the isolated sections.

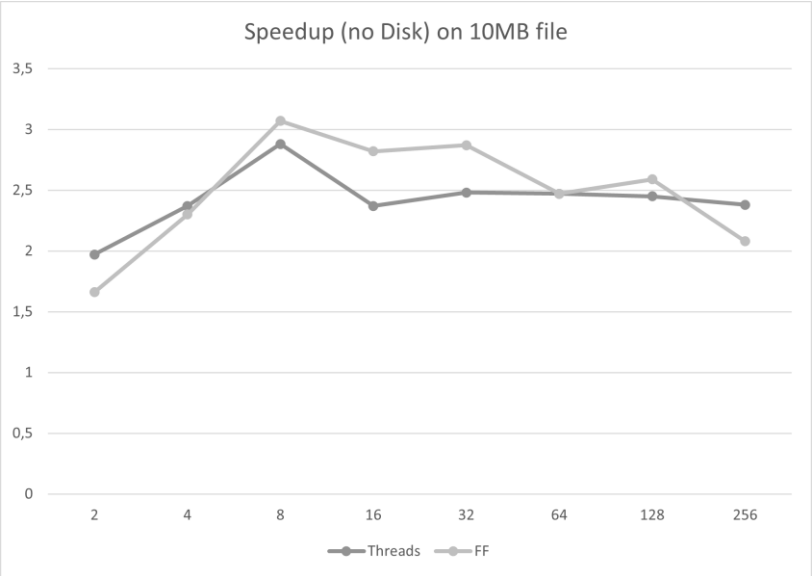
We will use the standard metrics for parallel applications: Speedup, Scalability and Efficiency.

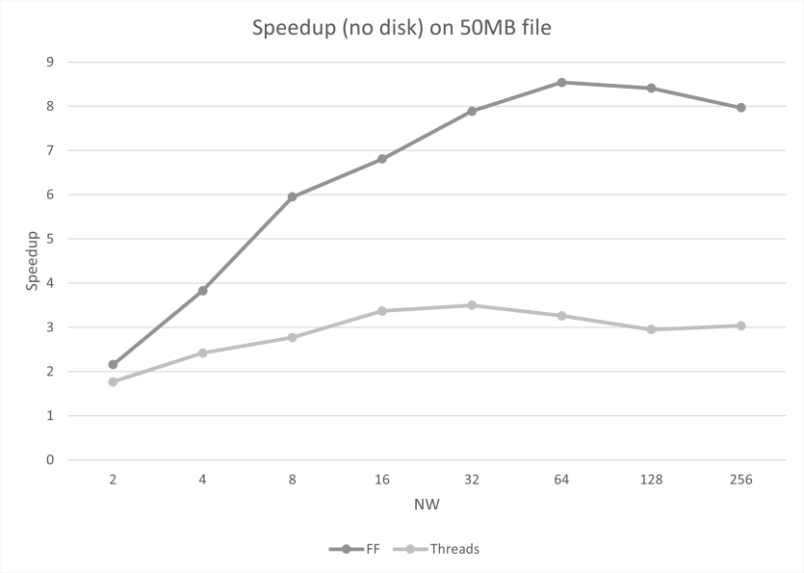
The tests were performed on two different input file sizes: 10MB and 50MB, on the physical San Piero a Grado NUMA machine, averaging between 10 executions.

Speedup







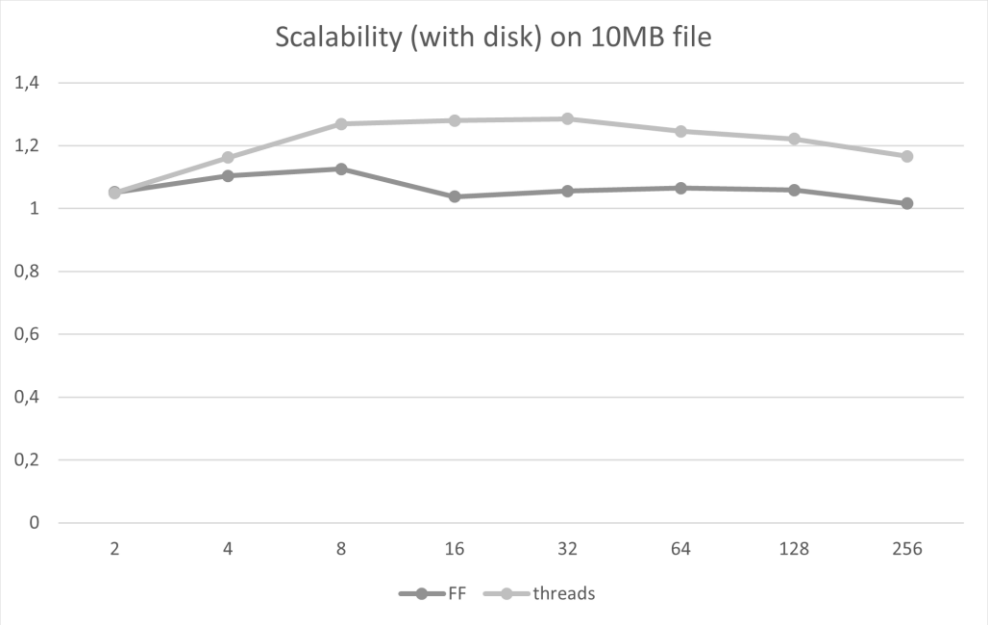


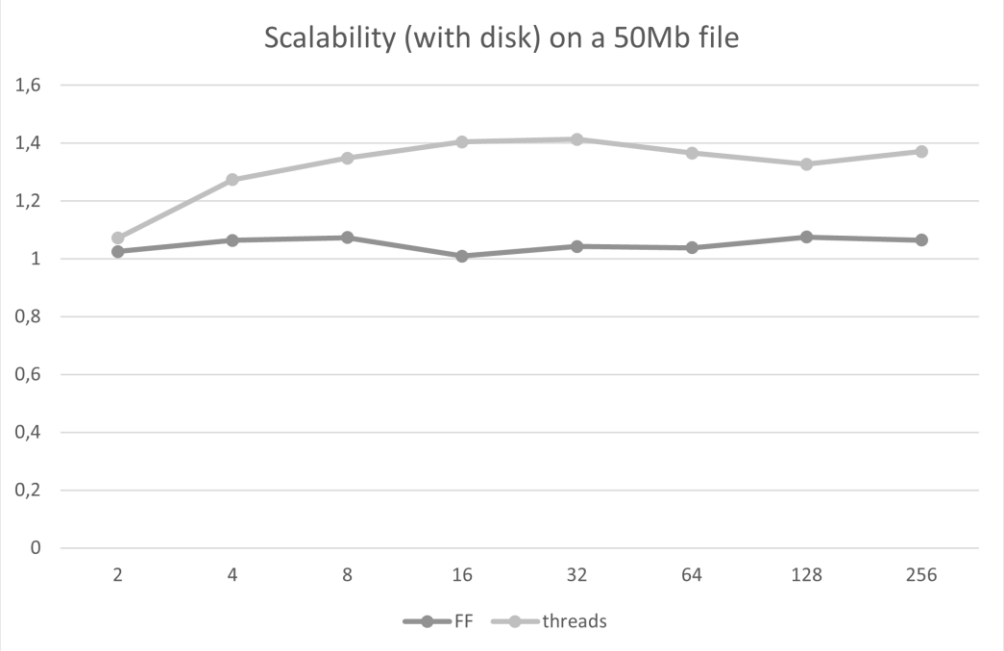
We can notice that a big part of the computation involves reading and writing from the disk. Even though we tried to assess the writing problem with a pipeline, the improvements are small, especially with a file of small size. In fact, when we remove the disk factor from the equation, we can see a considerable improvement.

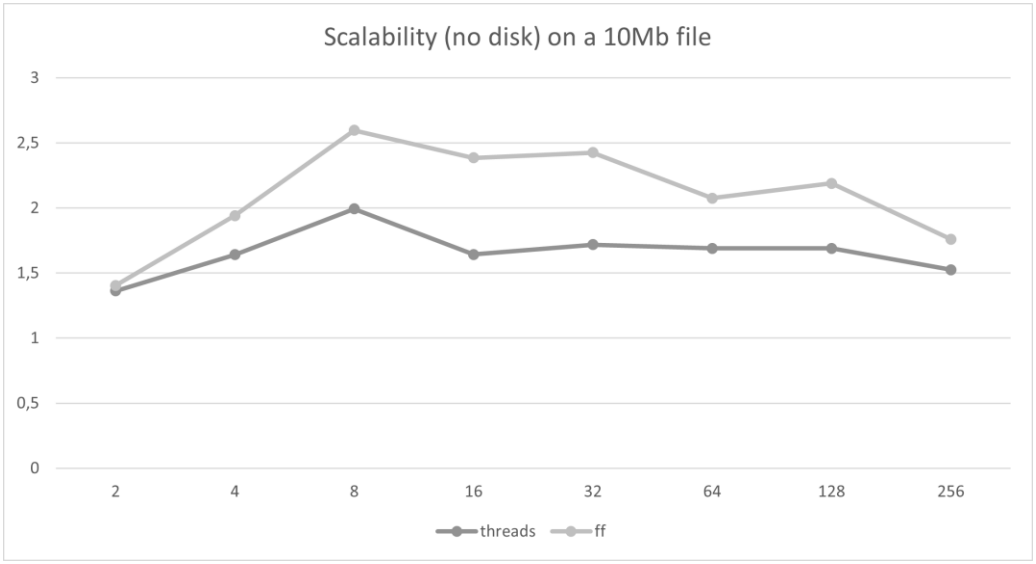
Talking about the performance of the fastflow version against the threadpool implementation, as expected, in the computations involving the disk the fastflow implementation performs worse, because of the fact that we are moving the file content onto memory before starting the computation. When we look at the results of the disk-less programs, though, the threadpool performs poorly. One of the main reasons is surely the thread mapping to the physical cores: on the threadpool implementation it was *not* performed; Fastflow instead does.

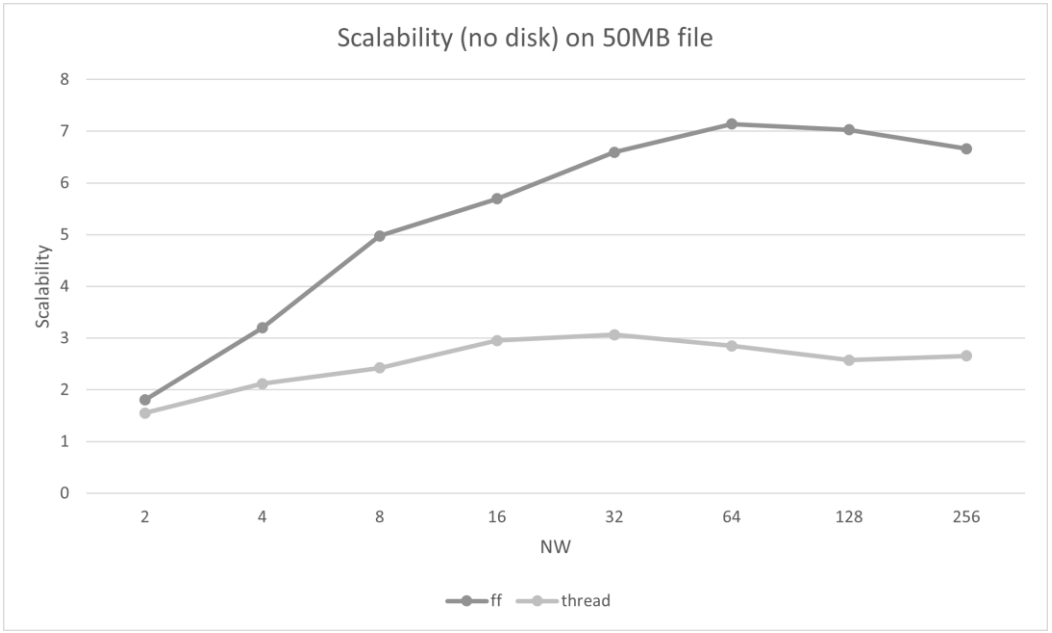
As expected, as well, we don’t notice any improvements when the number of workers matches or becomes bigger than the number of physical cores of the machine; instead we can notice a small decrease in performance due to the overhead.

Scalability





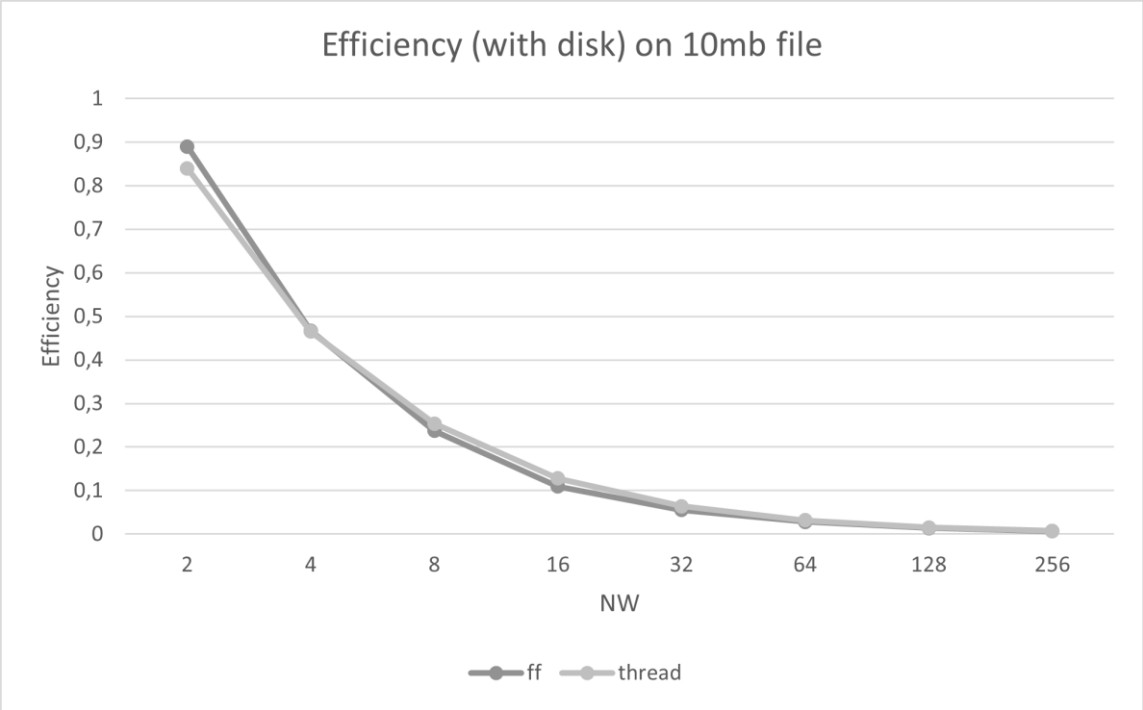


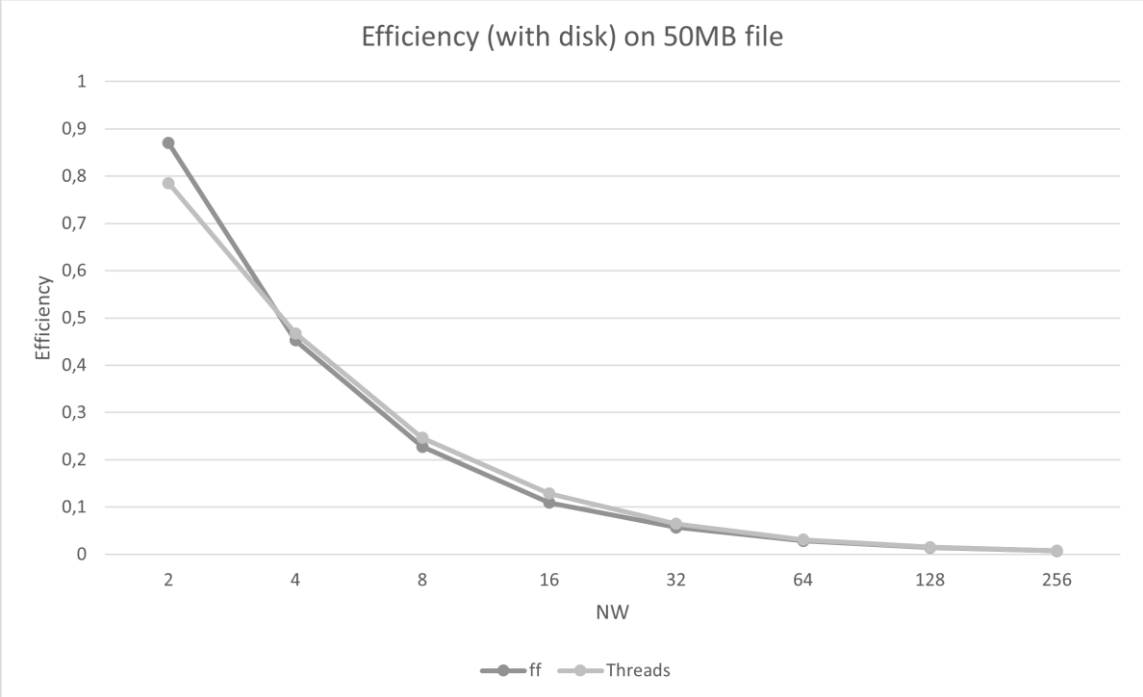


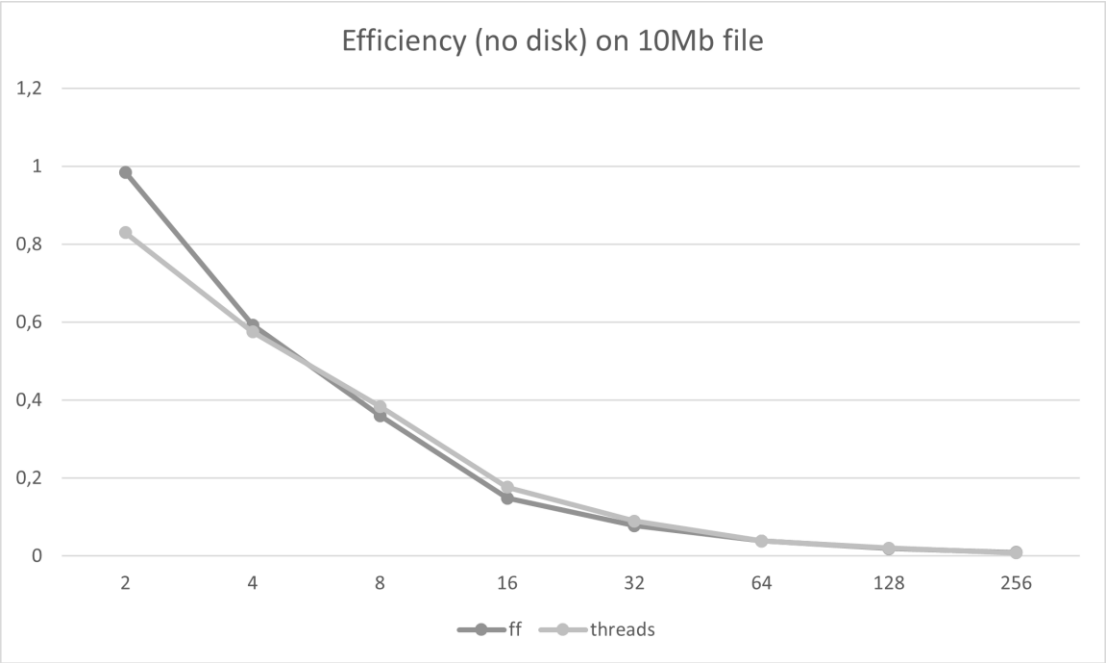
As to confirm the speedup results, scalability improves when :

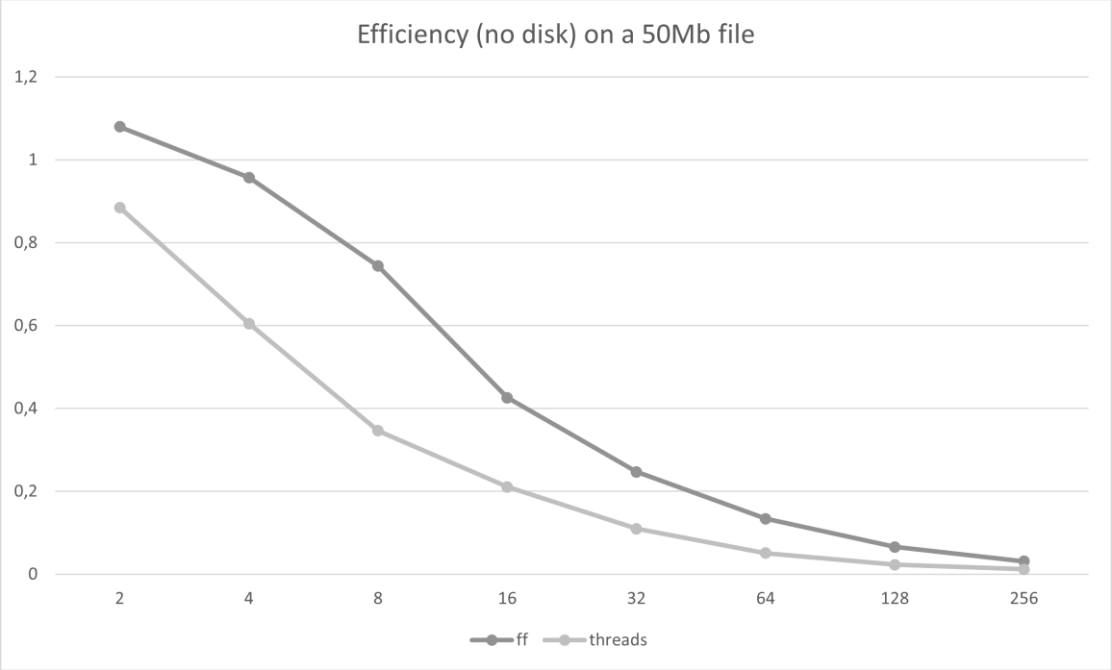
- The file is bigger;
- We are not calculating the disk operations;
- The Fastflow version is used.

Efficiency


Efficiency (with disk) on 10mb file
Efficiency
1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
2
4
8
16
32
64
128
256
NW
ff
thread


Efficiency (with disk) on 50MB file
Efficiency
1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
2
4
8
16
32
64
128
256
NW
ff
Threads





As in both the Speedup and the Scalability, Efficiency results are coherent to what expected as well.