SPM Project Report

Davide Amadei
Computer Science (AI). d.amadei@studenti.unipi.it
Academic Year: 2022/2023

1 Introduction

In this project I implemented three versions of the Huffman encoding algorithm. A sequential implementation that serves as a base, and two parallel implementations, one using native C++ threads and one using FastFlow. The overall approach to parallelization was to mainly consider data parallel patterns, rather than stream parallel ones.

2 How to compile and execute

All the instructions described in this section assume that they are called inside the base directory of the project. To compile everything it is enough to simply call `make all`. To compile the FastFlow version it is obviously required to have the FastFlow library present. It can be placed either in the `/usr/local/include` folder or into a `./lib` directory inside the folder of the project.

The rules `make download_txt` and `make create_large_test` can be used to get some files for testing purposes. The former downloads a copy of the "Divine Comedy"(∼500KB) and "War and Peace"(∼4MB) in .txt format and converts them to ASCII. The latter needs to be run after the first one and creates a larger file of ∼120MB obtained by concatenating "War and Peace" 40 times.

To run one of the versions of the program after compiling the following command has to be used: `./<seq/par/ff>_hc.out -i input_file -o output_file`.

This is the bare minimum needed by any version to run. There are also other optional parameters, as explained in the help message printed by the program which I report here. The `-t` option is only present in the parallel implementations.

```
The program accepts the following arguments:
    -i path: path to the file to be encoded, required.
    -o path: path where the encoded file has to be saved, required.
    -t number: number of threads to use, default 4.
    -v: set verbose.
    -l dir: enable logging to file, output is written to directory dir.
    -d: debug mode, only works if logging is enabled.
```

The debug flag is only there to change some of the program flow for easier tracking of performance in some parts of the code, as the implementation makes it impossible to properly measure them otherwise.

While not required by the project I also implemented a simple sequential version of a decoder to check that files were being encoded properly. It is compiled together with everything else and the instruction to run it is: `./decode_test.out input_file output_file`.

Documentation can also be generated with `make docs`.

3 Sequential implementation

The sequential version is rather straightforward. The file to be encoded is read all at once into a vector of characters, which is then is parsed one character at a time to count how many times each one appears. To store the character frequencies I used a vector of 256 integers. This makes it so that the program actually works with any type of file, not just ASCII. It would in fact be more appropriate to say that the file is parsed per byte, rather than per character, and the frequencies are those of the possible configuration of bits in a single byte. An advantage of using a vector for this purpose is that it can be directly indexed, making it faster to access it compared to more complex data structures. Something to note is that I tested many ways to read the file and the one used is the fastest I could find.

To create the Huffman tree a well known approach with two queues is used. One of the queues will be initialized with the leaves in increasing order of frequency, the other will contain the intermediate nodes. At each step the algorithm takes the two nodes with lowest frequency from the heads of the queues and creates a new parent node for them, which is inserted into the queue. The algorithm ends when both queues are empty. The tree is then explored to find the encoding of each character. Codes are stored as integers and built by shifting bits as appropriate while exploring the tree, so in addition to the code its length needs to be stored as well.

When the tree has been built and the codes extracted the file can then be encoded. For each character, the corresponding encoding is retrieved, which can be done efficiently because codes are stored in a vector indexed with the characters themselves. The encoding of the character is then stored in a 1 byte buffer by manipulating the buffer and the encoding with bitwise operations. When the buffer is full its value is stored in a vector and the buffer gets reset. If the last buffer has some remaining space the significant bits are shifted to the beginning of the byte and the rest are considered padding. An alternative solution I initially considered to store the encoding is using a vector of booleans, but researching it further I found it is generally best to avoid it, as in C++ it is not a standard vector, packing multiple bools inside a single byte, possibly causing issues especially when writing to file.

When the file has been completely encoded the results can be written to file, though some metadata has to be stored first to allow decoding. The first thing written is the number of chunks (this is always 1 in the sequential version but is needed for consistency with the parallel implementations, as will be clearer later). Then the number of occurrences for each character is stored, as it is necessary to rebuild the tree when decoding the file. Finally, the size of the chunk in bytes and the number of padding bits are written, followed by the actual encoded data.

4 Parallel implementation

To better explain the reasoning behind the choices made in the parallel implementations, in table 1 I report the execution times of the sequential version run on two files of different sizes. Times in the table are the average over 10 runs.

As expected most of the time is spent encoding the file, which is a highly parallelizable task, as the file can be split in chunks and each one can be encoded independently of each other.

Section of Code	Execution time (∼4MB)	Execution time (∼120MB)
Reading file	1,715.0 μs	71,259.2 μs
Gathering frequency data	3,249.6 μs	363,694.0 μs
Creating Huffman Tree and table of encodings	38.2 μs	55.5 μs
Encoding file	19,795.5 μs	786,787.0 μs
Writing output	997.0 μs	28,960.0 μs
Total	26,166.6 μs	1,257,580.0 μs

Table 1: Execution times for the sequential implementation

Parsing the input file and counting how many times each character appears is also quite slow, especially with larger input files. This operation can also be easily made parallel the same way as the encoding, processing each chunk and putting results back together at the end. On the other hand, building the Huffman Tree does not depend on input size, so the times are basically the same. Additionally the time needed is largely irrelevant to the total execution time, as long as the file is not incredibly small. Thus, this section of the code has no need to be considered for parallelization (note that it would also be rather hard to do because of how the algorithm works). IO operations take a smaller but not irrelevant amount of time, though they cannot be easily parallelized. Nonetheless, there is one way to introduce a small optimization in reading and writing, which will be explained in the following section.

With all this said, the focus in making the program parallel naturally falls to improving performance of the two slowest sections of code. As previously described, making them parallel is quite simple, basically being two map operations. Additionally, it should be noted that in both cases there is never any need to write to data structures shared between workers, meaning that false sharing will not be an issue and that there are no critical sections, thus no need to introduce synchronization mechanisms that would slow down the code. The aforementioned IO optimization does require some synchronization when writing though.

4.1 Native C++ threads

Rather than reading the file all at once, it is read in chunks. When a chunk is read it is passed to a thread which gathers character frequencies of that chunk and stores the results in a vector of partial results. When all threads are finished the results are gathered in the main thread in a single vector which is then used to build the Huffman tree. Joining the partial results adds basically no overhead, as it depends only on the number of threads, and even with 64 threads it only gets up to around an average of 11 μs from the tests I ran. This section is basically a farm pattern, although each worker only processes a single chunk and is started when data is available. Reading the whole file at once and then running a map on the chunks should in theory take the same amount of time on average, but from testing done ends up being slightly slower compared to the implementation I used. The main difference is that in the implemented method the additional time taken after reading the file is given by the last worker to be started,

in the other way it would be the slowest one, given they would all be started together. Given that the chunks are all of the same size load balancing is not an issue, so any variance in execution time of each worker does not depend on the program.

Moving on to encoding, the implementation is very simple, as it just involves each thread encoding a chunk of the file. The encoding process of each chunk is the exact same as for the whole file, and there is no need for synchronization, as each thread works on its own chunk and only reads from the table of encodings. One possible issue for this implementation is load balancing, as chunks might contain a lot of characters with short or long encodings, causing some threads to finish before or after the others. When considering ASCII files though, it is reasonable to assume that characters will have similar distribution throughout the whole text. A possible alternative to reduce the issue of load balancing would have been to consider a stream parallel approach instead of a data parallel one, though this would make it more difficult to handle an issue with putting results back together, which is already present in the current implementation to a much less problematic degree.

The problem is as follows. When encoding the file in chunks, each one will likely have some padding in the final byte, requiring the encoded chunks to be joined in some way before writing them to file, given that the decoder would be unable to know which are the padding bits to be ignored. Any solution to join the chunks, though, would likely introduce a relevant amount of overhead, considering the need to shift or copy large amount of data. Considering the parallel setting, this process would be further slowed down by the need to retrieve the data to be joined. This is because each chunk will be allocated by different threads, thus by different cores, and will almost surely not be in the local memory of the main thread. The situation gets even worse when running the program on a multiple socket machine.

To avoid this overhead I opted to simply not concatenate the chunks. Rather, at the beginning of the output file the number of chunks is written. Then, for each chunk, its size and the amount of padding bits are written before the actual encoded data. This comes at a slight increase of 5 bytes per chunk in the size of the output file, a very low amount compared to the space needed by the rest of the file. This same solution would have been less appropriate in a stream parallel implementation, as the amount of chunks would be much higher than one per thread, causing a larger overhead in terms of space.

An apparently small detail that turned out to be very relevant in this section is the initialization of the vector storing the encoded data. If no space is reserved for it the insertion of full buffers into the vector will significantly slow down the process, especially with a larger amount of threads. This is because every time the vector is filled it needs to be reallocated in memory and the existing elements need to be copied to the newly allocated memory. To solve this issue it is enough to use the reserve function of C++ vectors. I chose to reserve two thirds of the size of the chunk being encoded because from the tests done it is generally enough to store the encoding of the chunk, at worst only one reallocation will be done if it does not suffice.

The file is written progressively as the results are ready. This is done in the same function, thus in the same thread, as the encoding. This requires a bit of synchronization, as each thread has to wait on a condition variable and checks a global variable to know if it is its turn to write to file, so that the order of chunks can be preserved in the output. An alternative would have been to write everything inside the main thread, but writing inside the threads has an advantage in terms of trying to exploit data locality, possibly reducing the overhead of retrieving the data

from memory. The overhead caused by synchronization should not be particularly relevant on average, as the encoding of a chunk is fully done when the thread starts waiting and only the writing remains to be done. Additionally, this implementation might also be able to offset load balancing issues. For example if a thread finishes encoding its chunk before the others and it can write to file this will be done while other workers are still encoding, instead of having to wait for all of them to be done. The main advantage though remains the better data locality.

4.2 FastFlow implementation

The FastFlow implementation follows the same exact structure as the parallel one. One general difference between the two versions is that, because of how FastFlow works, many of the data structures needed to be stored using pointers to allow the various nodes to access them without creating copies and to directly store results inside them.

To count the characters in the file I used a FastFlow farm pattern with some modifications. I made a custom emitter which reads the file in chunks and stores them in a shared data structure accessible by other threads. When a chunk is read the emitter sends the index of the chunk to one of the workers in the farm so that it can start working on that chunk. The collector has been removed as it was not necessary, leaving the sum of the partial results to the main thread, being a very quick operation and thus not needing to be run in parallel with other tasks.

This section could also have been implemented as a two stage pipeline, with the first stage reading the file and the second being a farm that processes them. This would waste resources though, as the emitter of the farm would only transmit data between the first stage and the farm, taking up a thread, and a core, for no reason.

To implement the encoding process I used the ParallelFor pattern, which is the FastFlow implementation of a map pattern. Each thread encodes and writes a single chunk, in the same way as the other parallel implementation.

The overhead of this implementation should be around the same as the parallel one in terms of time necessary to create the threads. Given that FastFlow starts uses cores more efficiently because of thread pinning and starting them on different corse, performance should in theory be better, as the implementation using C++ threads has neither of these optimizations.

To better explain some of the results I also checked the order in which FastFlow allocates thread on the machine used. It first uses physical cores, alternating between the two sockets, and after all 32 physical cores have been assigned to a thread hyperthreading starts getting used.

5 Results

To measure the performance of the parallel implementations I ran the program with a variable number of threads, between 1 and 64. This maximum number was chosen because of the machine where the tests had to be run which has a 32 core CPU with 2 way hyperthreading, for a total of 64 threads. Adding more threads would have been unlikely to give better performance, and as it will be shown in the results, even going higher than 32 is not particularly beneficial. The reason is very likely that at that point some cores will have more two threads running on it, slowing down both of them.

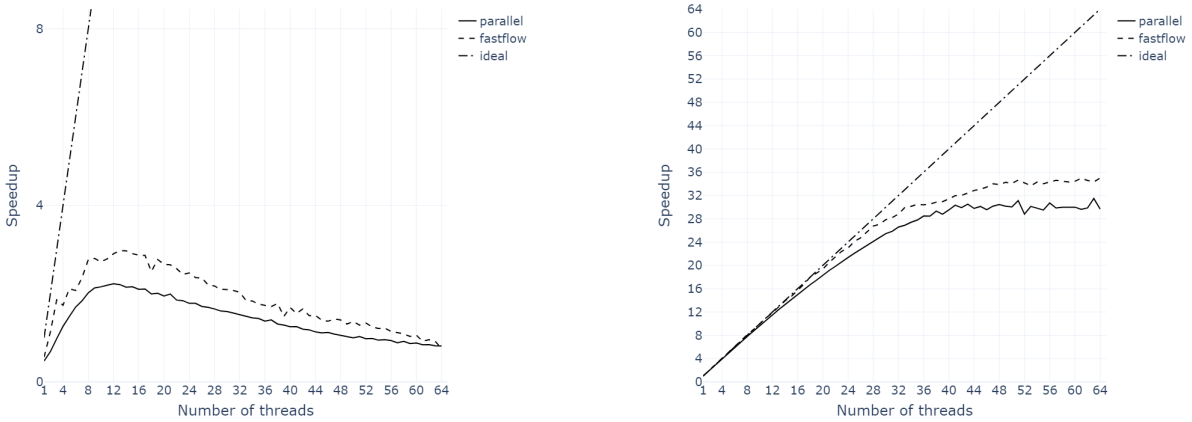
All tests were run on two of the files described in section 2: the 4MB one and the 120 MB one. Additionally, the results are the average over 10 runs. I do not report the results for the sequential version here as they have already been discussed in section 4.

One thing that should be noted is that because of how I implemented reading and writing operations, being run in parallel to counting characters and encoding the file respectively, I needed to add the debug flag that changes the control flow slightly. When enabled the file is read wholly before gathering character frequencies, and the encoded output is not written to file. This allowed me to measure the performance of these operations on their own, without including the time spent reading and writing.

Scalability plots are not included as they are almost identical to the speedup ones.

5.1 Counting characters

The speedup for this part of the program can be seen in figure 1. It differs greatly depending on the size of file being encoded. For the small file the speedup starts increasing more slowly when using more than 8 threads, and it starts decreasing after around 12 threads. The reason for this is mainly that the overhead caused by the thread creation starts being too high compared to the time taken by the actual processing of the chunk. When considering the larger file the speedup is much higher, with FastFlow being close to ideal speedup up to 32 threads. Further increasing the number of threads increases speedup to a much slower rate. In both cases the FastFlow version is faster than the implementation in C++ threads. This was to be expected because of FastFlow using thread pinning and allocates workers on cores in an appropriate order, whereas the other implementation does neither.



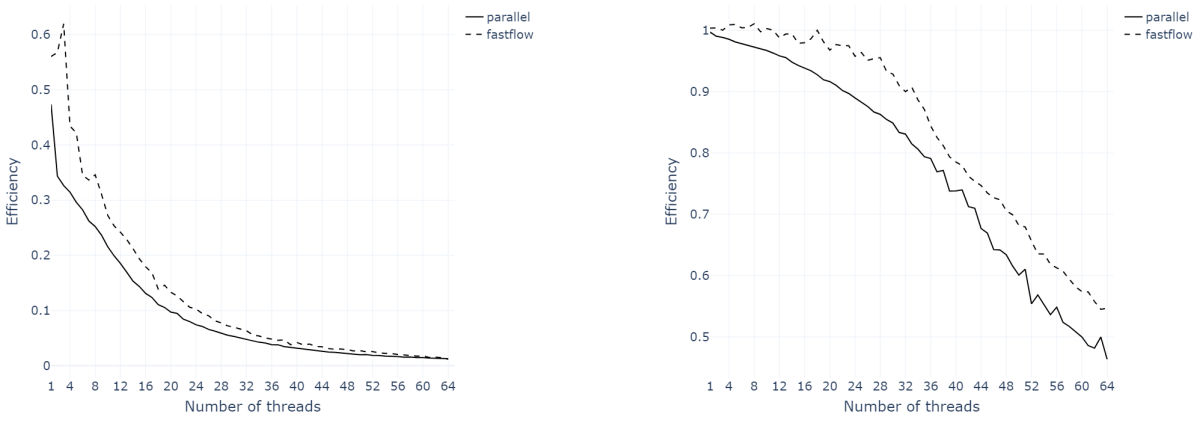
(a) Speedup for the small file

(b) Speedup for the large file

Figure 1: Counting characters speedup

Regarding efficiency, as shown in figure 2, the behaviour is similar to the speedup, albeit in a more extreme way. In fact, small files have quite bad efficiency, being almost always under 0.5. In general the speedup is worse than it should be for small files, as counting the characters

of a single thread is slower than it should be. For example when using 1 thread it takes around twice the time needed in the sequential version, even though the two times should be the same. When considering larger files the efficiency is much higher, being greater than 0.8 up until around 36 threads.



(a) Efficiency for the small file

(b) Efficiency for the large file

Figure 2: Counting characters efficiency

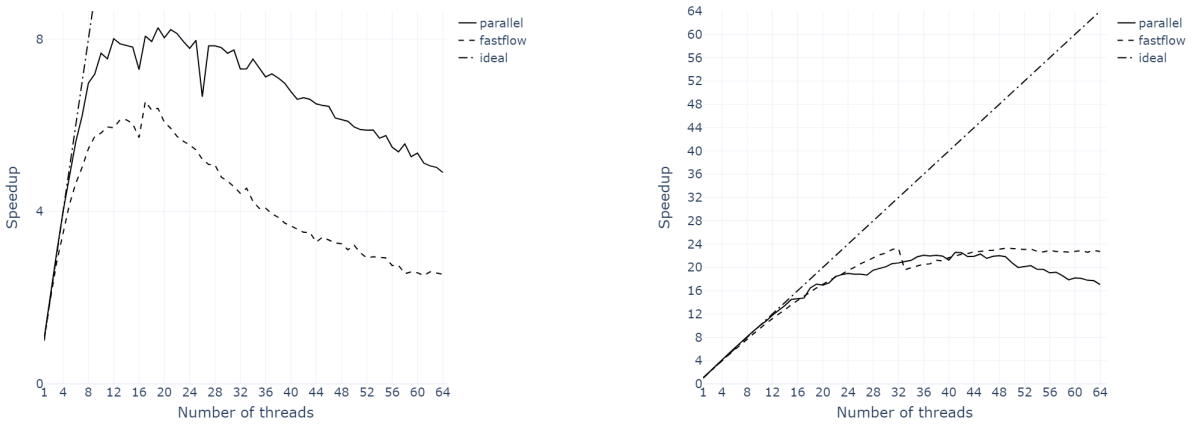
Overall, counting characters sees great benefit from using parallelization, as long as the file is large enough that the time spent on the task is not overtaken by the overhead of thread creation.

5.2 Encoding

As shown in figure 3 the parallel implementation has higher speedups when working on small files compared to counting characters, reaching slightly higher than a factor of 8. Note that the gains with more than 8 threads are much smaller, eventually losing performance with too many threads. Additionally, native threads perform better than FastFlow in this case. When encoding a large file, though, the two implementations are much closer, with FastFlow being slightly better overall. The maximum speedup achieved is around 24 when using 32 threads, which is less than ideal but still quite good. Adding more threads when FastFlow is used causes a steep drop off in performance, which is gradually recovered when further increasing the amount of workers, without ever surpassing the performance at 32 workers. The increase in execution time is likely to be because of the number of threads surpassing the number of physical cores causing the second threads of the CPUs to be used.

As to why the speedup is less than the ideal, from some additional tests I ran, I noticed that one cause lies in the time spent by each worker inserting full buffers inside the vector storing them. This time eventually stops decreasing linearly with the number of threads, slowing down the encoding process. This was further amplified before I started reserving memory for this vector, as the calls to insert elements had to reallocate memory many times for each thread.

Another factor is also the communication between cores to retrieve the chunks of file to be encoded. This is because the chunks all get allocated by the main thread, so the workers will not have them in local memory.



(a) Speedup for the small file

(b) Speedup for the large file

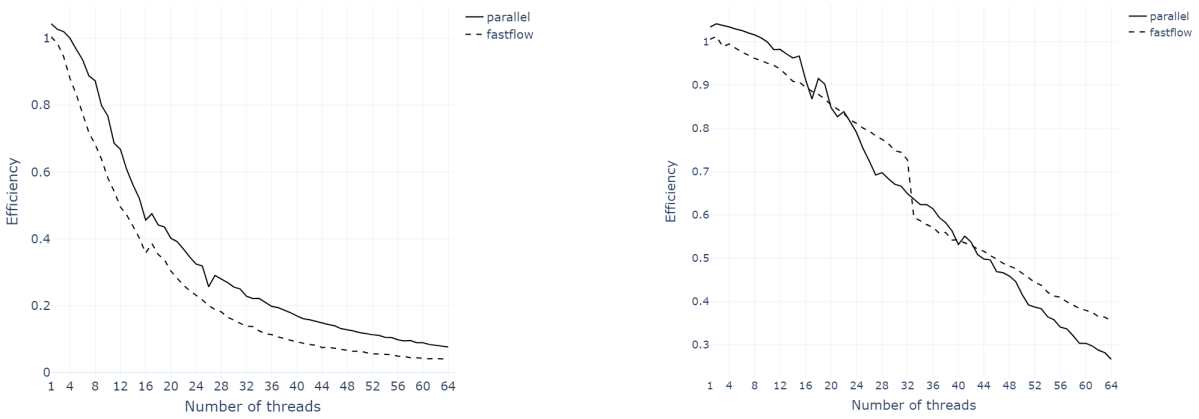
Figure 3: Encoding speedup

As shown in figure 4, efficiency is not too dissimilar from what already seen for the parallelization of counting characters, though small files maintain good enough efficiency for a small number of threads. Large files also show the same drop off as the speedup does after 32 threads. The efficiency is slightly lower here compared to the other parallel section.

5.3 Other

The speedup and efficiency for the whole program run on large files are shown in figures 5 and 6. When considering the time spent on IO operations the speedup achieved is a lot lower compared to the two sections analyzed so far. This is unavoidable as they are the main bottleneck of the program after parallelizing the relevant sections. With enough threads the time taken in the parallel sections eventually becomes similar or smaller than the time spent on IO, meaning that speedup and efficiency will suffer. Overall, looking at both metrics, performance sees the biggest gains until around 8 workers. The maximum speedup lies in the 24-32 range. If the time spent on reading and writing is excluded the speedup is much closer to the two previously analyzed sections, being especially similar to that of the encoding of the file.

One thing to note is that FastFlow has worse overall performance if IO operations are included, even though they are handled in the same way as the other parallel implementation. One possible explanation for this behaviour is the difference in compilation flags. The sequential and native C++ threads versions use the `-std=c++20` flag, while FastFlow requires `-std=c++17`, meaning there might be differences in the optimization of certain function calls of the standard library. I also noticed a weird quirk with the time needed for IO operations. For some reason the parallel implementations seem to generally take longer than the sequential version. While



(a) Efficiency for the small file

(b) Efficiency for the large file

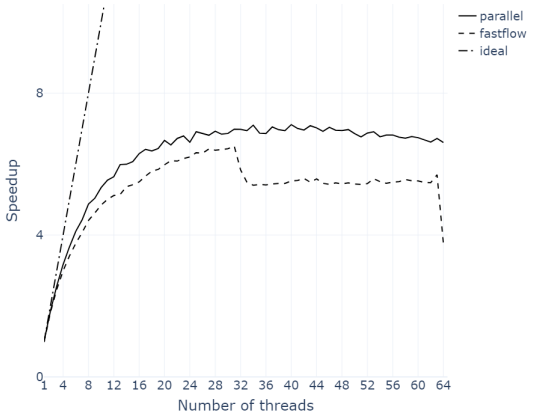
Figure 4: Encoding efficiency

this could be reasonable when increasing the amount of workers, this happens even with a single thread, which means the file is read all at once, same as the sequential version. One possible explanation is the optimizations the compiler is using with the `-O3` flag. In the parallel versions it is possible some of them are disabled because of the presence of threads. This is only a hypothesis though. When considering the total speedup including IO operations there is a noticeable drop in performance when going from 31 threads to 32. This is because of how the farm implemented in FastFlow works. More specifically, when considering a farm with 32 workers the actual amount of threads will be 33 (34 if the collector is not removed) with the extra one being the emitter. This emitter will be allocated on the same core as one of the workers, and the two threads will slow each other down, causing reading in particular to be much slower. One solution would be to use one less worker, so that the emitter does not have to compete for the physical core with another thread. Alternatively the farm approach could be dropped in favour of a map, with the main thread running the chunks and then running a map on them.

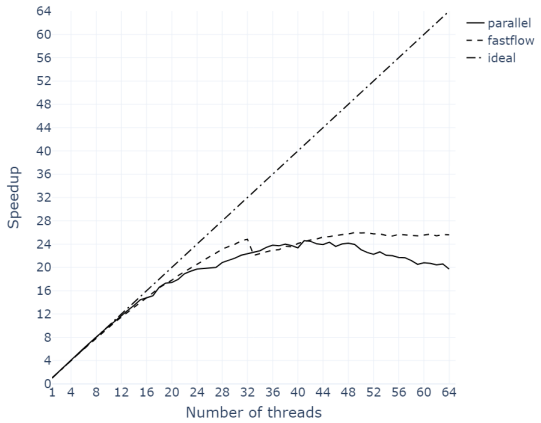
6 Conclusions and possible improvements

Overall, the two parallel sections manage to reach good speedups, although there is still some amount of overhead I could not manage to remove, supposing it is possible in the first place. This holds especially true for the encoding, which has a lower maximum speedup compared to counting the characters in the file. Solving the issues with IO operations being slower in the parallel implementation would also benefit performance greatly.

One more thing that could potentially improve execution times is better exploitation of NUMA architectures with multiple sockets. In the current implementations, the file is all read by the main thread. By having different workers read different chunks, it might be possible to have better data locality in the parallel operations, thus giving better performance.



(a) Speedup for the large file with IO operations



(b) Speedup for the large file without IO operations

Figure 5: Total speedup

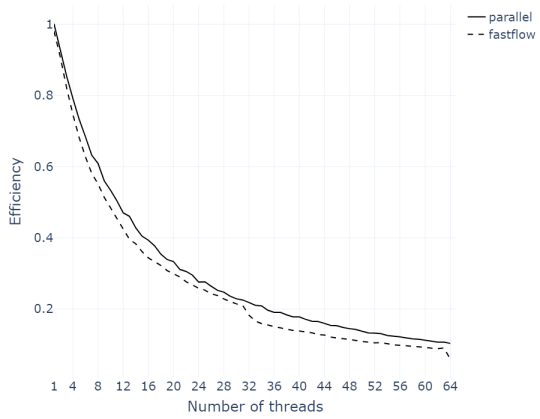


Figure 6: Total efficiency