Parallel implementation of Huffman Code using native C++ threads and FastFlow library

Parallel and Distributed Systems: Paradigms and Models

University of Pisa

Project Report

Matteo Tolloso

m.tolloso@studenti.unipi.it

Roll number: 598067

September 1, 2023

1 Introduction

The Huffman code is an efficient lossless compression code based on the probability of each character.

To build the optimal code for a specific text we have to:

1. count the number of occurences of each character in the text;
2. build the binary tree that represents the code;
3. encode the file.

2 Overview

We are facing a problem that can be divided in three stages. In particular it is a *data parallel* task, since we have all input available at the beginning of the computation.

2.1 Counting the number of occurences

As stated above, the first stage is a counting one. The asymptotic sequential complexity of this part is $\theta(m)$ where m is the number of characters in the file. From a parallel/distributer point of view this is clearly a *map-reduce* operation.

Map The *Map* part can be execute in parallel dividing the file into chunks, the workers count the occurences in a chunk of the file. This operation has to deal with the disk. If we consider the reading of the disk as a sequential operation things became more difficult because it's no longer a data parallel problem but a stream parallel one. In this setting we can describe the process as $\texttt{pipe}(reading, \texttt{farm}(counting, nw))$, the completion time of this process is the time needed to read the file from the disk, under the assumption that the farm has the right number of workers not to be the bottleneck of the operation. This approach is the one that minimize both the completion time and the number of workers but it it cause a lot of comunication overhead, needs some tuning of the chunksize to send and of the scheduler's policy and is in general more complex to implement.

If we instead consider the reading of the disk as a datata parallel operation that consists in moving data from the disk to the main memory, we can use the $Map\ Fusion$ theorem and transform the program in $\texttt{map}(read-count, nw)$. This solution minimize the communication overhead, the completion time and the complexity of the implementation.

Furthermore, the tests that i did mapping the file in main memory and reading it with multiple threads, showed that the parallelization also improves the performance of the read operation. This is probably due to how the SSD works and the caching syestems.

Reduce After the *Map* operiation we end up with a number of counts vectors equal to the number of chunks the file was divided into (that in our case is equal to the number of workers). The *Reduce* operation is again a parallel one, this time each reducer takes a subset of the alphabet and sums the occurences of each character in that subset. It's useless to have a number of workers greater than the number of different characters in the file.

2.2 Building the binary tree

The second stage is the building of the binary tree. This is a more difficult operation to parallelize since most of the operations are sequential. Furthermore, the complexity of this stage is $\theta(A \times log(A))$ where A is the number of differtent symbols (128), so basically it is a constant in our case. Tests showed that the time needed for this stage is competely negligible with respect to the other stages.

2.3 Encoding the file

The last stage is the encoding of the file. This is a *Map* operation, since each character has to be replaced with its code and writed on the disk. We can make a similar reasoning as the one made for the counting stage about the $Map\ Fusion$ theorem.

Unfortunately, the lenght of the final text can only be known after each character has been encoded (because the encoding of each character has a different lenght), so the actual writing needs a step of syncronization. I solved this problem dividing this stage in thread parts:

1. *encoding*: each worker encodes a chunk of the file.
2. *balancing*: the encoded chunks sizes are made multiple of 8 and the the index where the writing should start is computed. This is a sequential syncronization step but the time needed is negligible.
3. *compressing and writing*: each worker takes a chunk of the encoded file and writes it on the disk groping the bits in bytes.

It's fundamental to notice that the *balancing* step makes the encoded chunks independent one from another, so the *compressing and writing* can became a parallel operation.

3 Implementation

3.1 Overheads

False Sharing The false sharing problem is avoided sice each worker writes on a competely different array: the counting arrays and the chunk-encoding arrays are allocated by each worker.

Heap pressure The access to the heap is mutual exclusive, so an high number of allocation/reallocation can cause a big overhead. The proble is adressed in two ways:

- Trying to use dynamic memory management only when strictly necessary
- Use an alternative allocation library optimized for multithread applications

Load balancing Let's suppose a static load balacing. During the counting operation the file is equally diveded between the workers i.e. each workers counts the same number of characters. In the reduce phase each worker takes an equal subset of characters and sums the occurences. In both cases could happen that a worker have to deal with bigger number with respect to othes, but there are only *+1* operations thay should not depend on the size of the number. In the encoding phase each worker takes a chunk to encode. This part can be really unbalanced if the orginal file has somewhere a lot of alligned equal character, infact, this character will probability have a short code and the worker that encodes that chunk has to do fewer memory reallocations.

Synchronization In the FastFlow implementation the syncronization is competely managed by the library. One set of threads is spown at the beginning and the runtime support manages the queues and the implicit barriers. In the native threads implementation I had to manually managed the syncronization. The easiest way would have been to spown and join a set of threads for each stage, each time with the assigned funcion and arguments. This approach would have been really simple but it would have caused a lot of overheads since from some tests on the reference machine, the creation and join of a thread takes about $70\mu s$, while the insertion of a task in a shaerd queue takes about $1\mu s$ (and the creation of the shared queue takes $4\mu s$).

4 Tests

The table 1 shows the time of the various stages of the sequential implementation. The great part of the time is spent on the encoding, compressing and writing phases.

In tables 2 and 3 we can see some measures of the FastFlow implementation and the native threads one. The total do not correspont to the sum of the single stages because they didn't take into account the initialization of the memory and the stuctures needed. The stages measureas refers only to the actual computation while the "total" refers to the time from the start of the program to the end.

The "read and count" stage has a great speedup, in particular the application API of FastFlow allows an almost linear speedup in this operation and more in general the speedup of the actual computation is always better with FastFlow than with the native

Stage	**Time**
read and count	25928430 (26 s)
huffman	103 (0.000102 s)
encoding	212385536 (212 s)
compressing and writing	317802196 (317 s)
Total	566897566 (556 s)

Table 1: Sequential times, in usec, for 8GB file of random characters. Averaged over 10 runs.

Stage	**Time**	**Speedup**	**Efficiency**
read and count	839930 (0.8 s)	30.87 x	0.96
huffman	77 (0.000077 s)		
encoding	9725888 (9.7 s)	21.83 x	0.68
balancing	73 (0.000028 s)		
compressing and writing	20835071 (20.8 s)	15.25 x	0.48
Total	36295647 (36.2 s)	15.61 x	0.49

Table 2: Parallel times with FastFlow implementation, in usec, for 8GB file of random characters. 32 physical core machine. Averaged over 10 runs.

threads implementation. If we instead consider the total speedup, threads are slightly better probably due to less overhead in the management. The "compress and write" stage has the worst speedup because it involves writing on disk. The "encoding" phase is independet from the disk and I expected a better speedup. Probably the overhead is caused by the memory reallocation needed to store the encoded characters, ence a competition to access the heap despite the use of the jemalloc library. Even allocate a lot of memory at the beginnig is not a solution because the threads will compete the same. One possible solution could be switch to arena mode immediately but it is ouside my control.

In the figure 1 the completion time of the two parallel implementation is compared. The FastFlow result is quite surprising because it achieves the minimum time with only 16 threads, and starts to worsen the performance increasing the parallel degree. A similar behaviour can be observed with the native thread, wehere is not worth to increase the number of threads over 28. The same results can be seen under a differen prospective in the efficiency plot, figure 2. I suspect that the great efficiency of FastFlow with 16 cores is correlated with the architecture of the machine used for the experiments.

From the figure 2 we can also observe that the efficiency is greater than one. This can happen when the speedup is greater than the parallel degree because the original

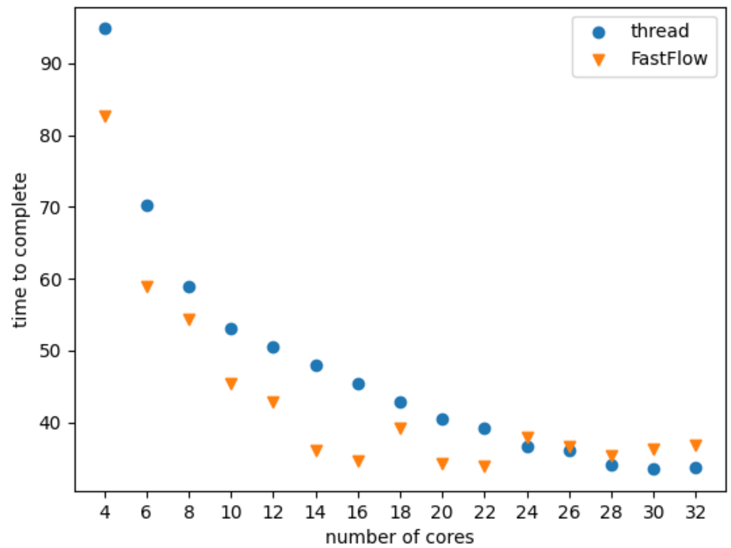

Figure 1: Time for encoding of a 8GB file of random characters. Averaged over 10 runs.

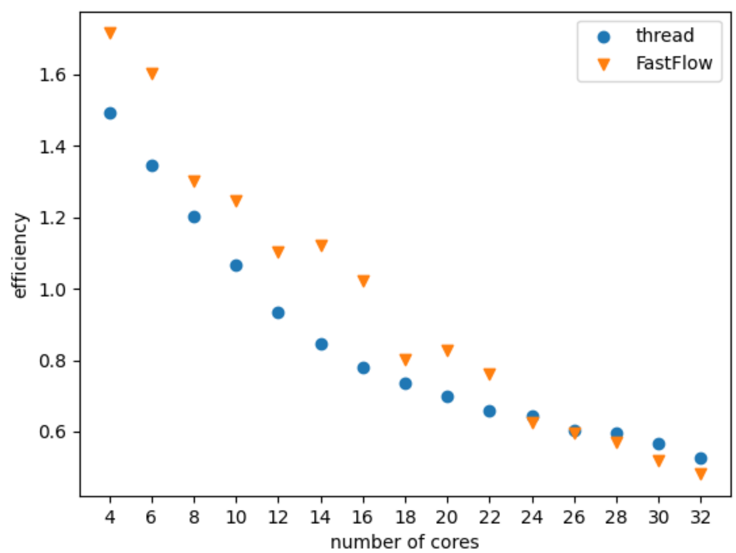

Figure 2: Efficiency in the encoding of a 8GB file of random characters. Averaged over 10 runs.

Stage	Time	Speedup	Efficiency
read and count	894301 (0.9 s)	28.99 x	0.90
huffman	95.7 (0.000095 s)		
encoding	10269624 (10.2 s)	20.68 x	0.65
balancing	27.2 (0.000027 s)		
compressing and writing	21213486 (21.2 s)	13.86 x	5.66
Total	33328868 (33.3 s)	17.00 x	0.53

Table 3: Parallel times with native threads implementation, in usec, for 8GB file of random characters. 32 physical core machine. Averaged over 10 runs.

problem became much easier when it is divided into subproblems. It might be that allocate and work a single 8 GB block is much slower than allocate 32 different 256 MB blocks, especially with the optimizations made by the jemalloc library and FastFlow. For example could happen that the sequential process allocates the memory on the RAM bank attached to the curren core and then is moved to another core or even anoter socket causing a lot of delay in accessing memory; in the parallel process is more likely that each tread allocate the memorory on the nearest RAM bank and then remains pinned to the same core.

Of course these are only hypothesis difficult to verify, however the test reported in table 4 shows how important is the use of the jemalloc library for these implementations, expecially with FastFlow, that suffers by an order of magnitude the unoptimized standart library. This could partially explain the superlinear speedup observed before.

	malloc	**jemalloc**	speedup
threads	169339515 (196 s)	33328868 (33.3 s)	5.08 x
FastFlow	414727131 (415 s)	36295647 (36.2 s)	11.42 x

Table 4: Time to encode a 8GB file of random characters, in usec. Standart malloc library vs jemalloc. 32 physical core machine. Averaged over 10 runs. The term “speedup” here has a different meaning than the usual one: it is the ratio between the time with the standart library and the time with jemalloc.