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 Theoretical analysis

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. 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. The asymptotic sequential complexity of this part is $\theta(m)$ where m is the number of characters in the file.

This operation has to deal with the disk. Under the following assumptions:

- the reading of a chunk from the disk is slower than the counting operation of that chunk;
- the readings are sequential;

it seems useless to parallelize this phase, since the disk is the bottleneck of the operation. The minimum completion time could be achived with two workers in pipeline, one reading and one counting.

Nevertheless, the tests that i did mapping the file in main memory and reading it with multiple threads, showed that the parallelization actually improves the performance. This is probably due to how the SSD works and the caching syestems. Furthermore, the counting operation seems to be slower than the disk speed. Quantitative results are shown in section 4 .

Reduce The *reduce* operation can again be executed in parallel, this time each reducer takes a subset of the alphabet and sums the occurences of each character in that subset. The sequential asymptotic complexity is constant since depends only of he number of different character. However, from a parallel point of view, this operation has a complexity that depeds on the number of chunks the file was divided into in the *map* phase (that can be different from the number of mapper workers), because the reducers have to sum the assigned characters along all the chunks. So if c is the number of chunks and A the number of different symbols (128), the complexity of the *reduce* phase is $\theta(c \times A)$.

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.

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. 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. The sequential complexity of this stage is $\theta(m)$ where m is the number of characters in the file.

We can make a similar reasoning as the one made for the *map* phase of the counting stage. The disk could be the bottleneck of the operation. If we divide the file in chunks and we encode in parallel each of them, we end up with a set of encoded chunks that have to be sorted and summed to get the total lenght before writing (if we want to write them in parallel). This operation has a complexity of $\theta(c \times log(c))$ where c is the number of chunks.

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 Results