Huffman code

University of Pisa

Parallel and Distributed Systems: Paradigms and Models

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 of each character 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.

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 (since 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.