

Parallel computing

Course project

***TOPIC: PARALLEL IMPLEMENTATION OF HUFFMAN’S***

***COMPRESSING ALGORITHM***

AUTHOR TUTOR

Mario Valentinov Dimitrov Hristo Hristov

Computer Science FN 80715 Lead Teaching Assistant, PC

Table of content

1. Abstract
   1. Problem Description
   2. Huffman tree construction
   3. Constructing the alphabet based on the Huffman tree
2. Solution strategies
   1. IO operations influence on performance
   2. Compressing
      1. Single file implementation
      2. File per thread
      3. Final decision
         1. Regarding compression
         2. Regarding performance
   3. Decompressing
3. Implementation notes
   1. Bla bla
4. Test results
   1. Test configurations
      1. Development configuration
      2. Performance measurement configuration
   2. Test cases
      1. Small test – natural language
      2. Big test – generated content
5. Performance results

# Abstract

## Problem description

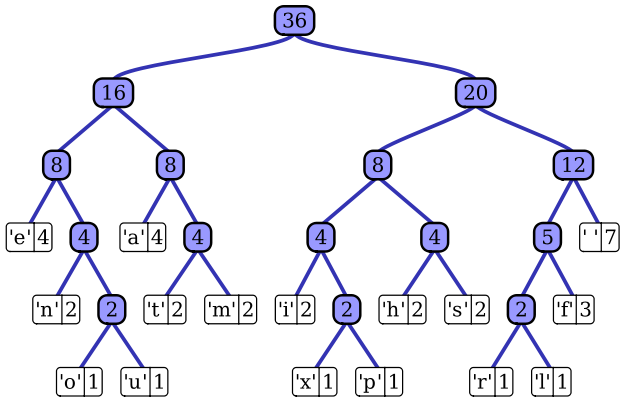
Huffman coding is an entropy encoding algorithm used for lossless data compression. It uses a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol.

The above solution is to the algorithm being used to compress and decompress an ASCII encoded text file, where at most 255 different symbols may appear.

## Huffman tree construction

To determine the compression alphabet, we construct a binary tree in the following steps.

1. Build a frequency map of all characters, appearing in the text
2. May there be n distinct symbols. We construct n trees contining one element each.
3. We join them in pairs (we choose the two with the smallest values) where a new tree is constructed – containing the cumulative count of symbols of its two subtrees.
4. We perform step C until a single tree is formed – the Huffman tree



## Constructing the alphabet based on the Huffman tree

The process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value). Before this can take place, however, the Huffman tree must be reconstructed. In the case character frequencies are stored in the nodes, the tree can be preconstructed (and even statistically adjusted on each compression cycle) and thus reused every time, at the expense of at least some measure of compression efficiency.

# Solution strategies

## IO operations influence on performance

Since IO operations are sequential, it is of little importance to performance whether the file reading operations will be performed by each thread individually or before the splitting. Inspite the fact that reading the whole file before involving the threads would make performance measurement easier and that using a Random Access File introduced overhead due to more system primitive calls, it was chosen that the whole file will be read in the memory prior to the algorithm run.

## Compressing

Given the initial file the implementation must parallelize the frequency map building, then construct the appropriate Huffman tree and produce a compressed version of the initial file, that can be easily decompressed afterwards. Naturally, the more work is parallelized, the higher the performance boost would be when running multiple theads.

### Single file implementation

A single Huffman tree is used to compress the whole file. Each threads builds its own character frequency map and then all maps are joined – either through a Concurrent Hash Map, through a Map – Reduce strategy, or simply sequentially. Then a single Huffman tree is built and every compressing thread receives its own copy to perform the compressing task.

### File per thread

In this option every threads builds its individual frequency map, constructs its own Huffman tree, and compresses its part of the file using that tree.

### Final decision

#### Regarding compression

It is a matter of heuristics to decide which of these strategies would bring a better compression level – in a case where one character dominates in the beginning of the text and another is in the same role at the end, the file per thread solution would bring better results. Storing multiple huffman trees would increase the total size, but for large files this overhead is concidered insignificant.

#### Regarding performance

The synchronization (joining) of multiple frequency maps in the single file implementation would require additional computing resources, while the creation of multiple files in the other solution would not be measured.

All things considered the file per thread solution was preferred, being a bit simpler and more clear.

## Decompressing

The ideas discussed in the compressing part are also valid in the decompressing part, and because the choice there determines this decision, every thread will decompress its given data independently and then all produced files will be merged sequentially.

# Implementation notes

## Console tool

The console tool reads the filepath, max threads count and quiet mode preferences from a predefined list of options default names.

## Compressing

The whole file is read using a Buffered reader into an Array List. Then it is being split between the threads. The last thread consumes the remaining smaller amount of buffers. Every single thread is fed and then builds

## Decompressing

# Test results

## Test configurations

### Development testing configuration

Dell 15 R laptop with Windows 8.1 64 bit, JDK 1.7, Eclipse Kepler, Intel Core i-7 2630 QM 4 cores, 8 threads, 6GB RAM, HDD

### Performance testing configuration

Linux Ubuntu 16 cores

## Test cases

### Small test

### Large test

Random ASCII encoded characters 512 MB - around 512 million