Parallel Implementation of Huffman Coding

Federico Matteoni

A.A. 2022/23

1 The Alternatives

Huffman coding may be implemented in different ways. Huffman uses a table containing the number of occurrences of each symbol in the input text. The goal is to assign a unique sequence of bits to each symbol in an optimal way, meaning that the encoding of each character must have a unique prefix to unambiguously differentiate each character. This is achieved by arranging the characters in a binary tree: by assigning each subtree a 0/1 value, the code for each character is found by traversing the tree from the root to the leaf, concatenating the found 0/1s.
Building such tree is the key of the Huffman Coding algorithm.

Greedy The first approach that comes to mind can be defined as a greedy approach. Given our n characters, the idea is to start with a tree rooted in a fake node with n subtrees, each representing one of the characters. Then, iteratively, we combine two nodes with the smallest weights into a tree, assigning as weight the sum of the two original nodes. This loops until we end up with a binary tree.
This approach involves sorting the characters based on the frequencies, requiring $O(n \cdot \log n)$ for n characters, $O(n-1)$ steps of merging pairs of nodes and updating the overall tree. This results in a $O(n \cdot \log n)$ complexity.

Top-Down Another approach is to build the tree in a top-down fashion. We consider all possible ways of dividing the characters in the two subtrees, and once the decision is taken we recursively do the same for both subtrees.
This algorithm involves a binary decision for each of the characters, thus resulting in an exponential ($O(2^n)$ for n characters) algorithm.

Heap The chosen implementation involves using an heap to store and encode the Huffman tree. This way, using a well-known and understood data structure, we may exploit different kinds of optimizations. The time complexity of traversing the heap, action involved in determining the smallest weights and inserting the nodes, is $O(n \cdot \log n)$. The advantage is that we maintain a single tree, and won't perform additional actions that will add overhead like in the greedy approach.

2 Implementation

3 Evaluation

4 Discussion

5 Running the Code