Parallel Huffman Coding

Francesco Bozzo
DISI, University of Trento
Trento, Italy
francesco.bozzo@studenti.unitn.it
229312

Michele Yin
DISI, University of Trento
Trento, Italy
michele.yin@studenti.unitn.it
229359

***Abstract*—This report aims to explain the design of a parallel encoding and decoding Huffman algorithm. The application is developed with the C99 programming language, using MPI for multiprocessing and OpenMP for multithreading to scale horizontally with increasing hardware resources. Our tool is both able to exploit multiprocessing to process concurrently multiple folders, and to split the Huffman coding of the same file among multiple threads.**

Comparing this tool with other online implementations, we can state that…

***Index Terms*—Huffman, MPI, OpenMP, High Performance Computing**

I. Introduction

Even if the Huffman algorithm is not directly used nowadays, its prefix mechanism is still part of Deflate (PKZIP's algorithm), JPEG, and MP3 compression algorithms

II. The Huffman algorithm

The Huffman algorithm has the objective of finding a more convenient bit representation to store information through lossless compression. Instead of considering groups of eight bits as the way to encode data, the Huffman algorithm uses variable-length sequences of bits with prefixes defined as *alphabet*.

A. Priority Queue

The Huffman algorithm uses a minimum priority queue to build its alphabet efficiently. This specific data structure ensures logarithmic insertion and deletion time with respect to its size. We implemented the minimum priority queue by using a minimum heap. Practically speaking, to implement the min-heap tree, we used a standard C array ensuring that the min-heap property still holds at every insertion and deletion:

$$A[i] \leq A[l(i)], A[i] \leq A[r(i)] \tag{1}$$

where $A[i]$, $A[l(i)]$, $andA[r(i)]$ are respectively a node, its left child, and its right child in a min-heap tree.

To implement the priority queue also a parallel approach has been considered [1], but since it contains only 256 elements, we did not consider it too important in terms of performance.

B. Encoding

The Huffman encoding procedure makes use of a minimum priority queue to build its alphabet efficiently. The idea is to build a tree similar to Figure 1 that defines all the variable-length prefix sequences of bits: these sequences are represented by the path from the tree root to a leaf. Using a greedy approach, the Huffman encoding ensures that the less frequent a byte is in a file, the more probable is to have a longer Huffman representation, which means that his path from the root to its specific leaf is longer.

Once having computed the frequencies for each one of the 256 different bytes in a file, the Huffman algorithm populates a min priority queue with Huffman tree nodes storing the byte value and its frequency. Successively, it removes the least two frequent bytes from the queue, creates a new node with children the two extracted nodes, assigns to it a dummy character and the sum of the frequencies of its two children, and inserts it into the min priority queue. After $n-1$ iterations, the last node is the root of the Huffman tree [2]. Algorithm 1 explains in the detail this procedure.

Algorithm 1: Build the Huffman tree

```
1  // Populate the min priority queue with characters and
     their frequencies
2  for $i = 1$ to $n - 1$ do
3      Q.insert(f[i], Tree(f[i], c[i]))
4  // Repeat until the queue has only a single element left
5  for $i = 1$ to $n - 1$ do
6      // Get the two least frequent nodes
7      z1 = Q.deleteMin()
8      z2 = Q.deleteMin()
9      // Create an inner tree node and insert it into the
         queue
10     z = Tree(z1.f + z2.f, null)
11     z.left = z1
12     z.right = z2
13     Q.insert(z.f, z)
14 // The last element in the queue is the root of the
     Huffman tree
15 return Q.deleteMin()
```

Once built the Huffman tree, it is possible to generate the

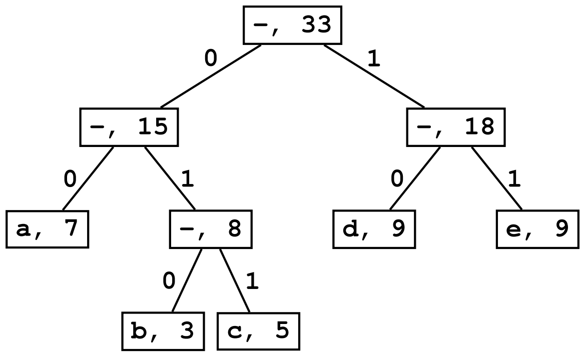

Fig. 1: An example of Huffman tree

Huffman alphabet visiting the tree using a DFS algorithm, assigning 0 to each left-child traverse and 0 for the right one as presented in Algorithm 2. Finally, it is possible to compress the file by creating a stream of bits corresponding to its content using the Huffman alphabet.

Algorithm 2: Encode using Huffman tree

```
1 while not eof() do
2     bit = read()
3     if bit == 0 then
4         encode(node.left)
5     else
6         encode(node.right)
7     if node is leaf then
8         return node.value
```

C. Decoding

Once having the encoded file and the Huffman tree, it is straightforward to decompress the file to its original shape. The prefix alphabet ensures to have that it is possible to visit the tree using a DFS approach and get a unique decoded version of the file using an approach similar to Algorithm 2.

III. Huffman Implementation

To be able to implement the Huffman encoding and decoding algorithm, several things have been taken into account.

- The byte frequencies computed during the encoding process are saved in the compressed file. In this way, the decoding procedure can easily rebuild the Huffman tree.
- Both the encoding and decoding procedures make use of buffers to improve I/O performance: the streams of bytes are first written inside the buffer and then saved on the disk.
- The encoding procedure works with chunks of 4096 bytes. This ensures that the tool can handle even large files when dealing with bit buffers. The decoding procedure deals with chunks of maximum size of $4096 * 32$ bytes, since the maximum length of a Huffman tree traverse from root to leaf is of 256 bits. As explained later, dealing with chunks makes handy the parallel implementation of the encoding and decoding algorithms.
- In the encoded file, the chunk offsets are saved at the end of the file: this ensures that the compressed stream of prefixed sequences of bits is still meaningful.

IV. Parallel Huffman Implementation

The application is developed with the C99 programming language, using MPI for multiprocessing and OpenMP for multithreading to scale horizontally with increasing hardware resources. We choose to use:

- multiple processes to process groups of file and folders separately;
- multiple threads to process different chunk of the same file in parallel.

V. Performance and benchmarking

Performance evaluation.

VI. Conclusions

To conclude.

References

[1] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis, "A parallel priority queue with constant time operations," *Journal of Parallel and Distributed Computing*, vol. 49, no. 1, pp. 4–21, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731598914253

[2] A. Bertossi and A. Montresor, *Algoritmi e strutture di dati*. CittàStudi, 2010. [Online]. Available: https://books.google.es/books?id=WKWLSQAACAAJ