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 $A[i]$, $A[l(i)]$, $and A[r(i)]$ 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

```
// Populate the min priority queue with characters and
   their frequencies
for i = 1 to n − 1 do
    Q.insert(f[i], Tree(f[i], c[i]))
// Repeat until the queue has only a single element left
for i = 1 to n − 1 do
    // Get the two least frequent nodes
    z1 = Q.deleteMin()
    z2 = Q.deleteMin()
    // Create an inner tree node and insert it into the
       queue
    z = Tree(z1.f + z2.f, null)
    z.left = z1
    z.right = z2
    Q.insert(z.f, z)
// The last element in the queue is the root of the
   Huffman tree
return Q.deleteMin()
```

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

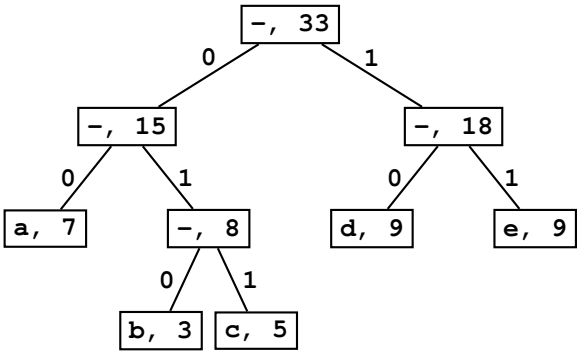

Fig. 1: An example of Huffman tree

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

To parallelize our algorithm over threads and processes, we decided to follow this simple yet effective structure

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

There are a many reasons to our choice. Mainly they are:

- In most operating systems a file is a resource that the OS gives to a single process. We wanted to follow a similar design philosophy.
- Most operating system can allow multiple processes to open the same file in reading mode, but few allow to multiple processes to open the same file in writing mode. This is because that leads to potentially concurrency and data integrity issues. By having only a single process open a single file, we avoid all these issues.
- Because threads of the same process share the address space, we can avoid the expensive data transfer across processes. When data is read or written to file, there is no need to transfer data between threads.

We start by explaining the basic multithreading operations on a single file and then we'll cover multiprocessing with multiple files.

A. Multithreading

The parallel algorithm for a single file follows the same exact procedure as a serial version, but with few key differences to allow multithreading.

A single file is divided in chunks of a fixed size. When that is done, the single process forks and creates the number of threads we intented.

One of them reads a number of chunks equal to the team size. When it's done reading the chunks, all threads can work on the processing in parallel.

Every chunk is processed by a single thread, that reads and writes on the shared memory space of its process. Although all the chunks are in shared memory, each thread is assigned a single chunk and it processes only data in own chunk to avoid any concurrency related problems.

When all threads are done computing either the encoding or decoding of their chunks, a single thread writes the processed chunks to the output file. Again, because all threads share a memory space there is no need to perform data transfer.

The overall procedure is the same in both encoding and decoding, with very small differences.

We also parallelized the counting of occurences of a byte in the input file, that is required in order to create the huffman

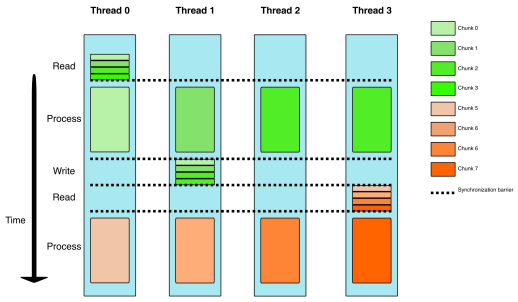

Fig. 2: Simple schema of the threading paradigm.

tree, with a similar architecture. Key difference is that there is no output file to write, but only occurences to count.

In figure 2a schema of the architecture.

B. Multiprocessing

For multiple files we wanted to exploit multiprocessing.

To do this we devised a simple algorithm to distribuite files across multiple processes, so that each process can work on its own list of files, indipendently to others. This way we can minimize communcation and therefore latency and other slowdowns.

First of all the process with rank 0 crawls all the files present in the input directory recursively. Then it opens all the files and reads their size.

Once all files and their respective sizes are known, process 0 creates a min priority queue where each item represents a process and its priority is the size of files that have been assigned to it.

Process 0 can iteratively insert files in the priority queue and updates the priority of each process with the cumulative file size. The main idea is that we can ensure an equal work division among different processes.

When all this is done, process 0 sends the list of files to each process and then each starts to work indipendetly on its own list of files.

C. Implementation details and other notes

- We found that a chunk size of 4096 Byte had the best I/O times. This is probably due to the fact that 4096 B is the size of a page in most Linux based operating systems. The last chunk of a file may be smaller. Reading any other size at the time had resulted in significanlty worse I/O times (both increasing and decreasing the chunk size).
- In general, all threads will finish processing their chunk at around the same time, but in some cases, there are threads that can take longer. This is due to the very unlucky case where a whole chunk contains exclusively very infrequent characters or bytes of data. Because the bytes are very infrequent each is encoded in extremely long sequences, up to 256 bits, resulting in a bigger encoded chunk than compared to the original data. With our alphabet at 1 Byte, the worst case results in a chunk that is 32 times bigger than original.
- We tried to parallelize the I/O, by having each thread read its own chunk. We found that the standard file descriptor provided by C have a lock to guarantee thread safety. This lock slows down the reads significantly. We also tried using multiple file descriptors to circumvent this limitation, however we found no improvement over a sequential read done by one thread for all the threads in it's team. This is likely because the O.S. schedules I/O requests and serves them one at the time, resulting in a impossiblity of having truly parallel I/O.
- We also tested an architecture where we had two dedicated threads to I/O, one for reading and one for writing chunks. The idea is that whenever a chunk is processed, the I/O threads would immediately write the processed chunk to the output file and similarly a new chunk would be read from the input file. Theoretically this would allow to parallelize I/O and computation operations and we avoided concurrency problems by synchronizing the I/O with locks. However we found that this approach was a waste of resources, as the I/O on the cluster is extremely fast ($\sim$ 5 GiB +/s) and it resulted in the I/O threads being idle most of the time, waiting for a chunk to be processed. Although we discarded this architecture, it may be useful in systems or problems where I/O operations take a significant amount of time or at least comparable to processing operations.
- Another noticeable detail we noticed on the implementation with two dedicated threads for I/O, is that if there are not enough cores on the CPU or if the operating system decides to allocate all the threads on a single processor, the encoding and decoding times are greatly affected by the scheduler. This is because it might be that the O.S. gives priority to threads that are waiting (for example the writer cannot write any block until the first has finished, even if all others have finished)

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