Huffman Coding

Francesco Bozzo Michele Yin

December 15, 2022

University of Trento

Introduction

Introduction

Table of contents

- The Huffman Coding Algorithm
- Serial Implementation
- Parallel Implementation
- Performance Evaluation and Benchmarks
- Q&A

The Huffman Coding Algorithm

Huffman Coding is a **lossless data compression** algorithm designed to find a more convenient bit representation to store data through variable-length sequences of bits defined as *alphabet*.

ASCII character	Byte encoding	Huffman encoding
a	01100001	00
b	01100010	010
c	01100011	011
d	01100100	10
e	01100100	11

Table 1: Example of Huffman alphabet for 5 letters

Frequency-based Encoding

The Huffman code for each character is decided by the occurrences of that character in the text using a **greedy** procedure:

- more frequent -> less bits
- less frequent -> more bits

Optimal Prefix Code

Even if the the Huffman algorithm is based on a greedy approach, it is able to generate an **optimal prefix code** in space efficiency.

Serial Version

Serial Version

The Huffman Compression Algorithm is composed by four phases:

1. Count the byte frequencies
2. Build the Huffman tree using the frequencies
3. Generate the Huffman alphabet by visiting the Huffman tree by using a DFS algorithm
4. Data encoding using the Huffman alphabet

Build the Huffman Tree

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, z2 = Q.deleteMin(), Q.deleteMin()
    // Create and insert inner tree node into the queue
    z = Tree(z1.f + z2.f, null)
    z.left, z.right = z1, z2
    Q.insert(z.f, z)
// The last element in the queue is the root of the Huffman tree
return Q.deleteMin()
```

Get the Huffman Alphabet from the Tree

Make a DFS visit of the Tree from root to leaves

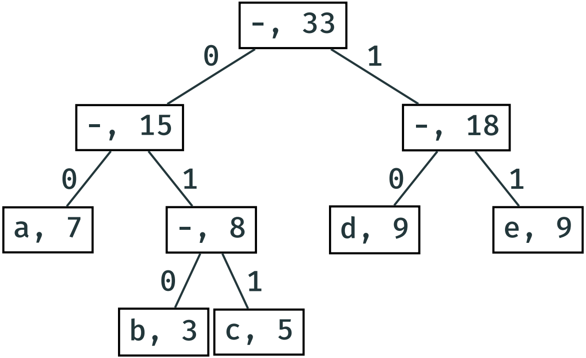

Figure 1: An example of Huffman tree.

Parallelization

~~Serial~~ Parallel Version

The Huffman Compression Algorithm is composed by four phases:

1. **Count** the byte frequencies
2. Build the Huffman tree using the frequencies
3. Generate the Huffman alphabet by visiting the Huffman tree by using a DFS algorithm
4. **Data encoding** using the Huffman alphabet

Step 1 and 4 are the most expensive and easiest to parallelize

Parallelization

- Multiple **processes** should handle separate files.
- Multiple **threads** of the same process should work on different chunks of the same file in parallel.

Reasoning

- In most operating systems a file is a resource that the OS gives to a single process to avoid I/O race conditions.
- Because threads of the same process share the address space, we can avoid the expensive data transfer across processes.

Multithreading

Given *m* threads:

1. A file is divided into *c* *chunks*, usually $m < c$
2. Until all chunks are not processed:
   2.1 A single thread reads *m* chunks and stores them in a shared memory space
   2.2 Each thread works on its own assigned chunk
   2.3 A single thread writes the processed chunks on the disk

Architecture

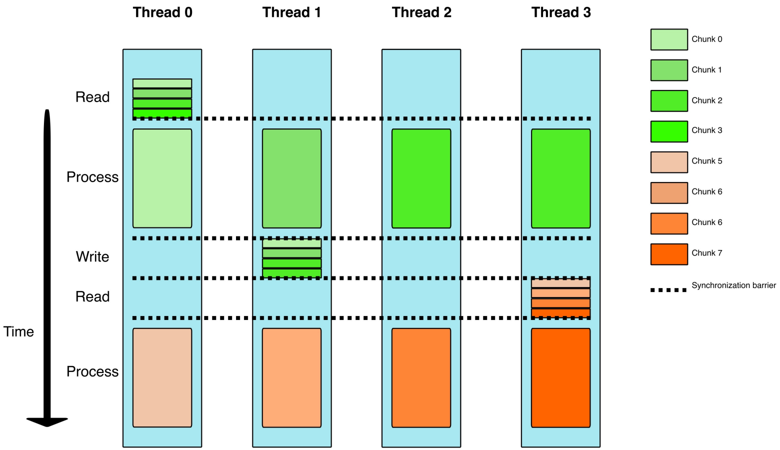

Figure 2: Simple schema for processing a single file with multiple threads.

Multiprocessing

- Rank 0 gathers all files in the input folder
- Reads their size
- Distributes to other processes files, balancing the load using a min priority Q
- Each process work on its own queue of jobs

Multiprocess Architecture

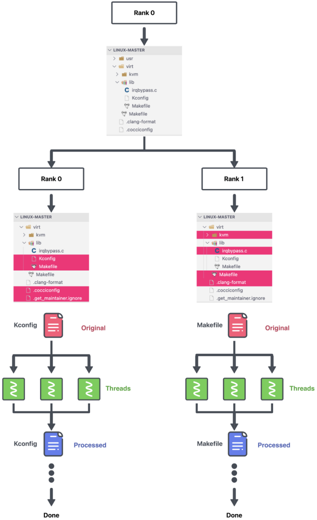

Figure 3: Simple overall schema

Implementation notes

- 1 byte as alphabet. More bytes results in less collisions and therefore less efficiency
- 4096 B as chunk size, because it is the standard linux page size

Alternative Architecture (with Locks)

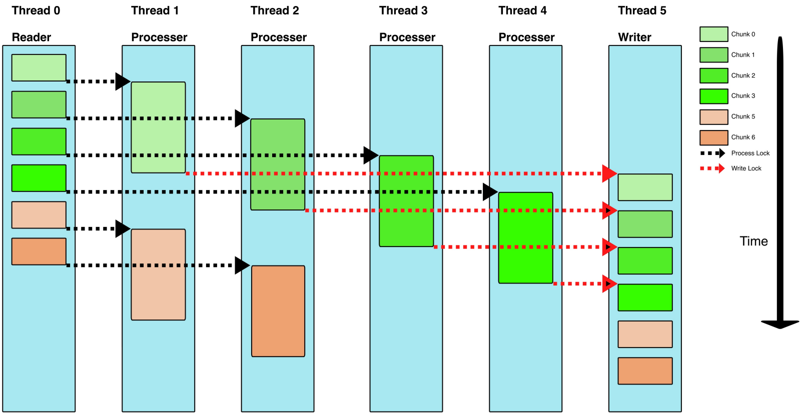

Figure 4: Simple schema for processing a multiple files.

Performance and Results

Results - Encoding

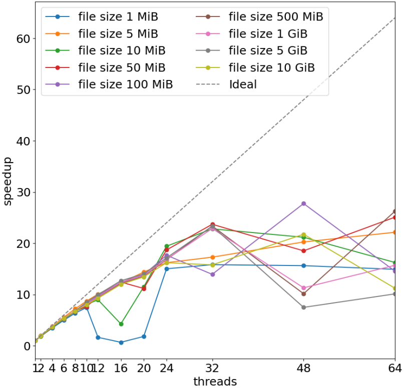

Figure 5: Encoding speedup

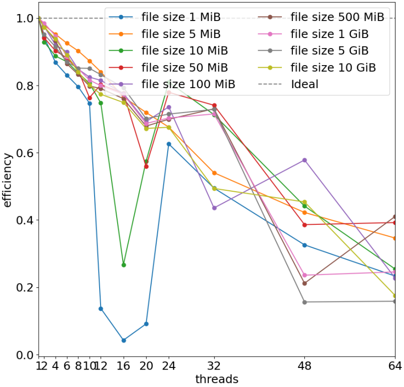

Figure 6: Encoding efficiency

Results - Decoding

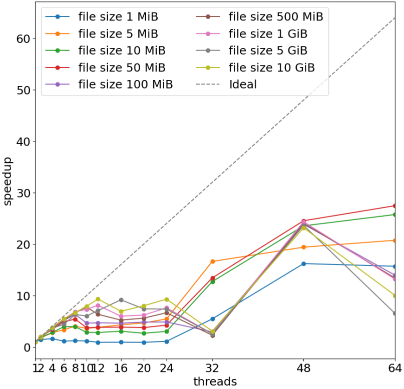

Figure 7: Decoding speedup

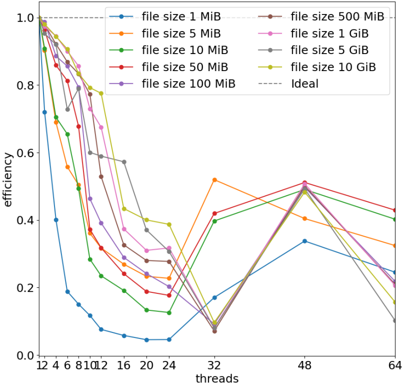

Figure 8: Decoding efficiency

Results - Barrier vs Locks

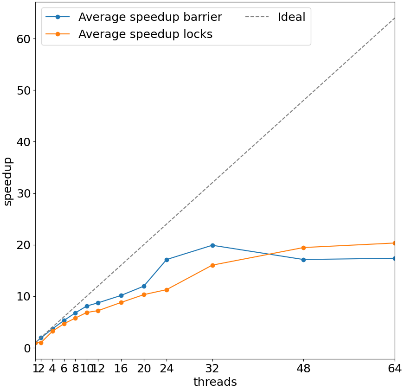

Figure 9: Encoding speedup barrier vs locks

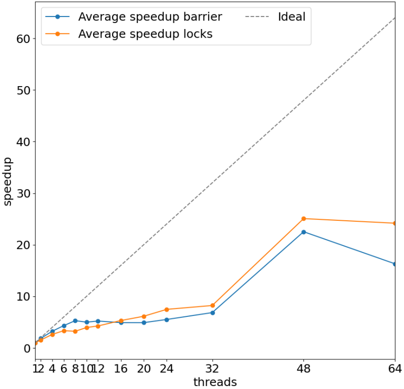

Figure 10: Decoding speedup barrier vs locks

Results - Ours vs Online Implementation

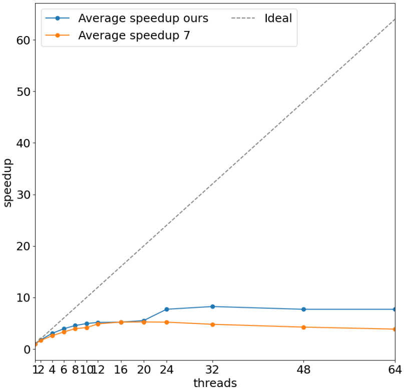

Figure 11: Average encoding speedup ours vs online 7

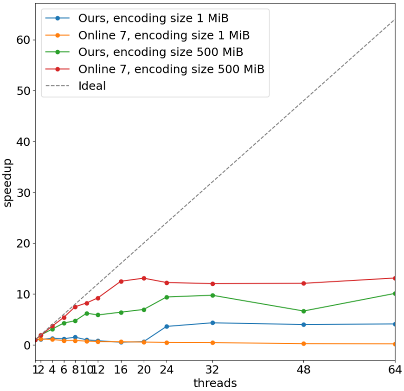

Figure 12: Encoding speedup ours vs online 7

Results - Folders

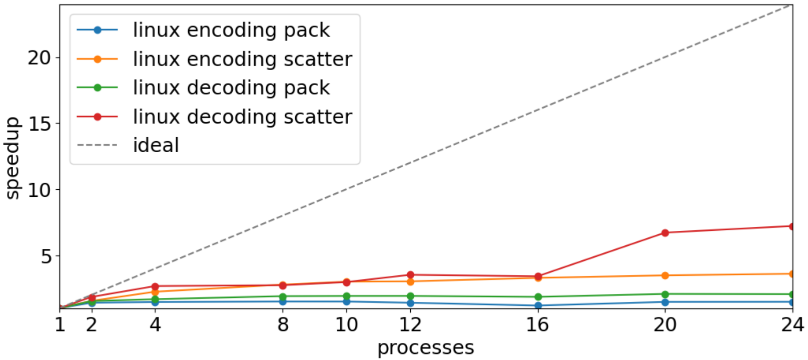

Figure 13: Encoding speedup with Linux