Parallel Data Compression
Tianyang Li

Data Compression: Huffman Coding

The main goal of this project is to implement parallelism on LZ77, Run-length encoding (RLE), and Huffman Coding. LZ77 offers lower potential speedup, but a higher compression ratio. Both Huffman Coding and RLE have higher parallelism potential. After the presentation, I focused on Huffman Coding as it offers better performance in compression ratio as well as parallelism. A simple parallel implementation of Huffman Coding gave us a decent speedup, so I decided to optimize our parallel version of Huffman Coding to get a linear speedup.

Summary

For the parallel implementation of Huffman, I looked at SIMD intrinsics as well as OpenMP on the Intel i7- 12700K CPU. I decided to use Huffman Coding because it has many parallelizable components and higher parallel performance gains. I was able to achieve linear speedup with OpenMP on the processor up to 16 threads. After 16 threads, the rate of speedup increase became slower and stopped at 128 threads. These approaches were looked at in terms of memory bandwidth, CPU utilization and cache behavior, and performance in comparison to sequential implementation.

Background

In the 1980s, as the internet became more popular, many compression algorithms were developed to overcome network and storage bandwidth limitations. There are two main types of compression algorithms: lossy compression algorithms, which are mainly used for image and audio compression, and lossless compression algorithms, which can be used for compression in storage devices where data loss is unacceptable. In our project, I am focusing on parallelizing one of the most widely used compression algorithms, Huffman Coding.

Huffman Coding is a process of finding the best possible prefix code for a collection of source symbols. This algorithm uses entropy encoding, which means that it uses less memory for characters that occur frequently, but more memory for characters that happen infrequently.

Huffman Coding can also be used in conjunction with other compression algorithms, such as LZ77, to provide an even better compression ratio. At present, many compression libraries are implemented sequentially. Therefore, it is very important to parallelize Huffman Coding in order to increase the performance of these compression libraries.

The Huffman Coding consists of 4 parts:

1) Reading a file to create a frequency dictionary that represents the number of occurrences of each character in the file

```
// Count frequency of appearance of each character and store it in a map
    unordered_map<char, int> freq;
    for (char ch : text) {
        freq[ch]++;
    }
```

This simple step is to build a map with character as the key and frequency as the value.

2) Build a Huffman tree with the least frequent character at the bottom and the most frequent character at the top.

First, we need to create a structure for the node

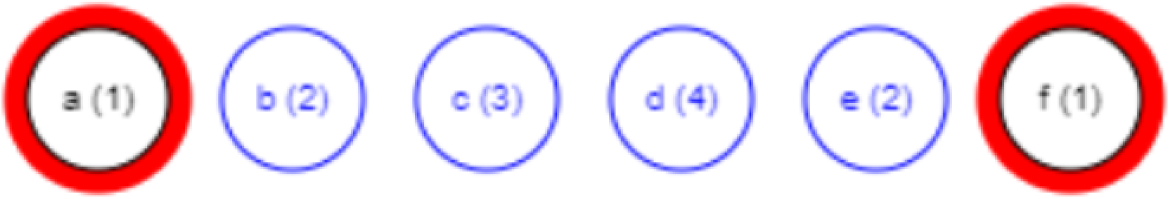
```
struct Node {
    char ch;
    int freq;
    Node *left, *right;

    Node(char ch, int freq, Node *left = nullptr, Node *right = nullptr)
        : ch(ch), freq(freq), left(left), right(right) {}
};
```

The count variable (int freq) returns the number of occurrences of all the characters corresponding to this subtree. Characters are always in the leaves of this subtree. At this point, we have all of the leaves for this subtree. Then, we remove the two nodes with the lowest number of occurrences. We create a parent tree with these two nodes as the "left" and "right" subtrees, and then we put the parent node in the heap.

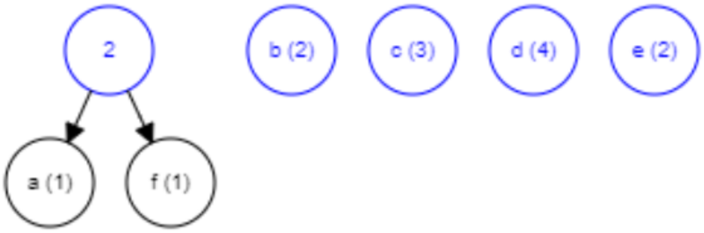
Visualization

Huffman Tree representation for: abbcccddddeef

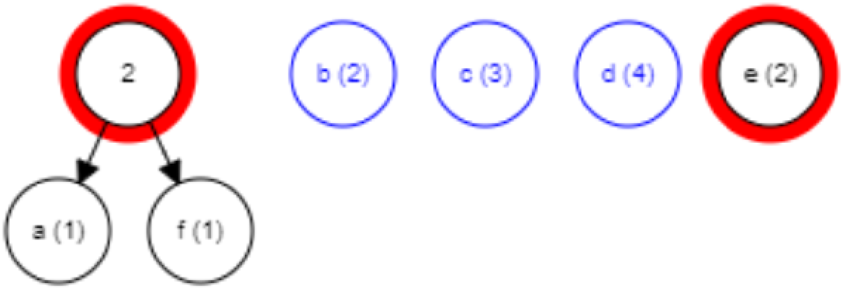
Count the frequency of characters

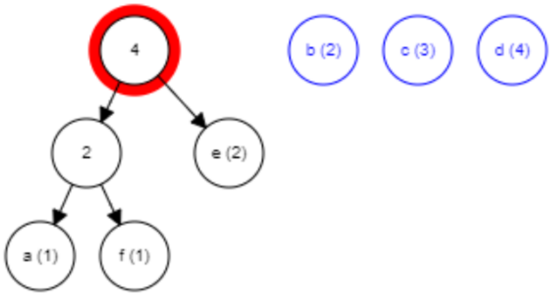


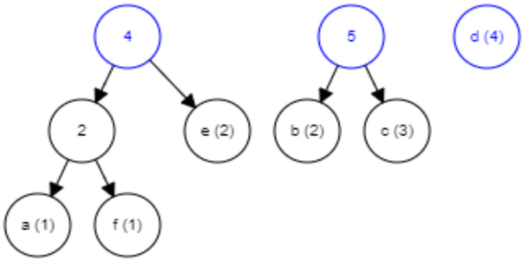
Pick out the 2 least frequent characters and add them as 2 parallel nodes, the parent node will be the frequency as 2 child node characters combined



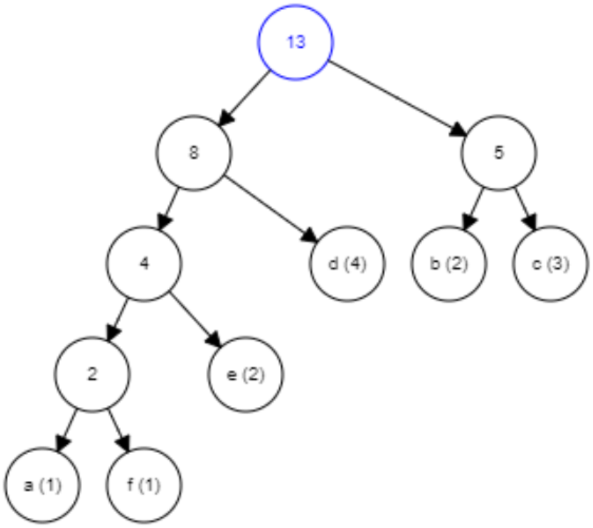
We repeat this step and select 2 new least frequent characters (or the frequency node)







And we get a complete Huffman tree for the string



3) Traversing the Huffman Tree to build an encoding format

Each time take a left write 0, take a right, write 1
First letter a: 0000
Second letter b: 10
Third letter b: 10
Fourth letter c: 11

So far, the encoding is 0000101011
As we can see the character with higher frequency such as b and c only requires 2 bits to be encoded, while the lower frequency such as a and f 4 bits. This saves resources as we will need to visit b and c more often than a and f.

4) The input file is encoded using the following encoding format and the output is written to the file

Then, we find the output file size, allocate an output field of that size, write each character to its encoded output, and load the encoded values to the output field.

The way I use to decode the Huffman tree is as follows:

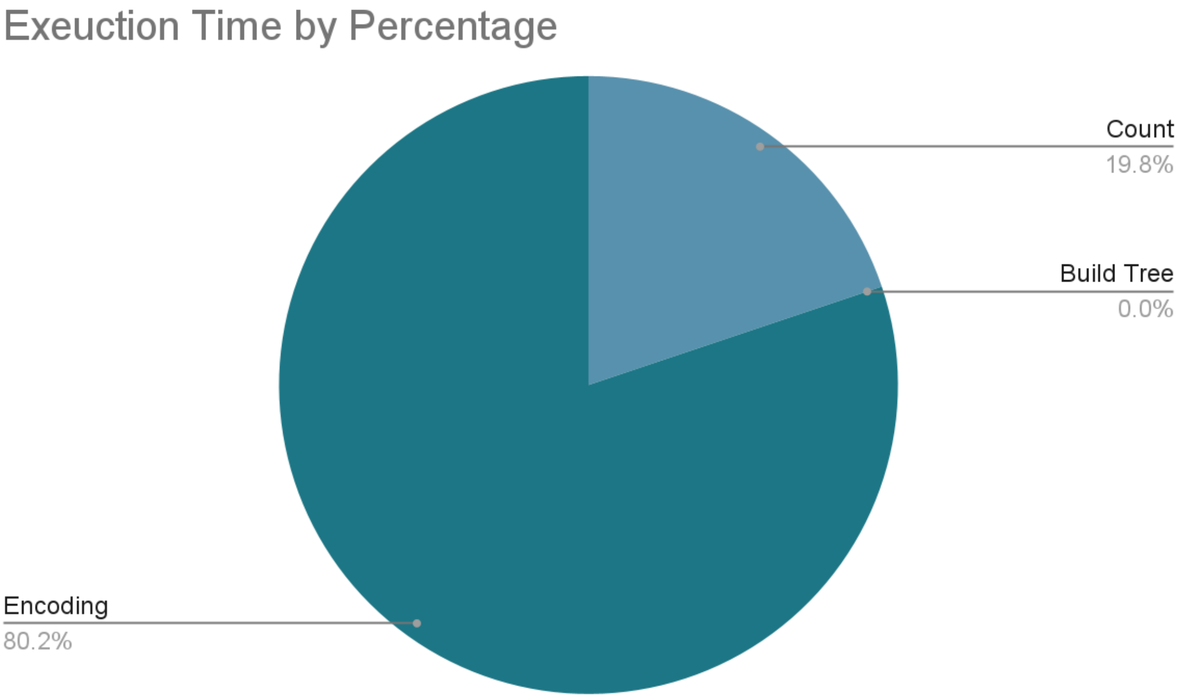
We start from the top level, when reading a 0 make a left. When reading a 1 make a right until we reach the bottom.
After reading 0000, we reach the bottom and we know the first letter is a
After reading 10, we reach the bottom and we know the second letter is b

Context

The command used to generate the text file:

```
tr -dc "A-Za-z 0-9" < /dev/urandom | fold -w100|head -n 100000 > bigfile.txt
```

This generates a text file with 100000 lines, which is about 963 MB



Writing the table part only takes microseconds and it’s not shown in the pie chart

As we can see the build tree and write table part is so negligible that doing parallelization on them would be pointless. So we focus on the counting character frequency and encoding part.

Hypothesis

Data partitioning: We can split input data such as a file or text string size into smaller pieces. The size of the partition can be determined by the number of threads in OpenMP, with the goal that each thread should have about the same amount of data. Once the Huffman tree is constructed, the encoding of the original data chunks can be done in parallel. Each thread encodes its respective data chunk using the Huffman tree.

Approach

A sequential version of Huffman Coding can be found here, this is an unlicensed project.
Below is my approach for parallelization

Step 1: File reading parallelization

Parallelization of reads does not require synchronization. Each thread reads parts of the file and writes to parts of the input array, which effectively parallelizes all transferred bytes from the file to the input array

```
#pragma omp parallel
    {
      int num_threads = omp_get_num_threads();
      uint64_t chunk_size = get_chunksize(infile_size, num_threads);
      int thread_num = omp_get_thread_num();

      ifstream thread_infile(file_name, ios::in | ios::binary);

      uint64_t thread_start_pos = chunk_size * thread_num;

      if (thread_infile.is_open()) {
        thread_infile.seekg(thread_start_pos, ios::beg);
        thread_infile.read((char *)data + thread_start_pos,
                                  min((thread_num + 1) * chunk_size,
                                      static_cast<uint64_t>(infile_size)) -
                                      thread_start_pos);
                thread_infile.close();
            }
        }
```

I also divided the file into equal sizes so each thread can complete the reading at roughly the same time

```
(num_tasks+num_threads-1)/num_threads;
```

After this step, I will add a barrier to align all strings. Then, each thread will be responsible for integrating a part of the global dictionary. I obtain a linear speedup for dictionary generation.

Step 2: Parallelize Encoding

Since I divided the files into chunks and assigned them to different threads, each thread needs to know the start and end index so I can aggregate them later. So after I split the data, I add a few identifiers to the data. Those are the size of the table, the index of the processor to which the task is assigned, and the start and end index of the data frame.

I use OpenMP to utilize the multi-core CPU. Since I am using the start and end index for each chunk of the file, I avoided the communication between the threads or race conditions.

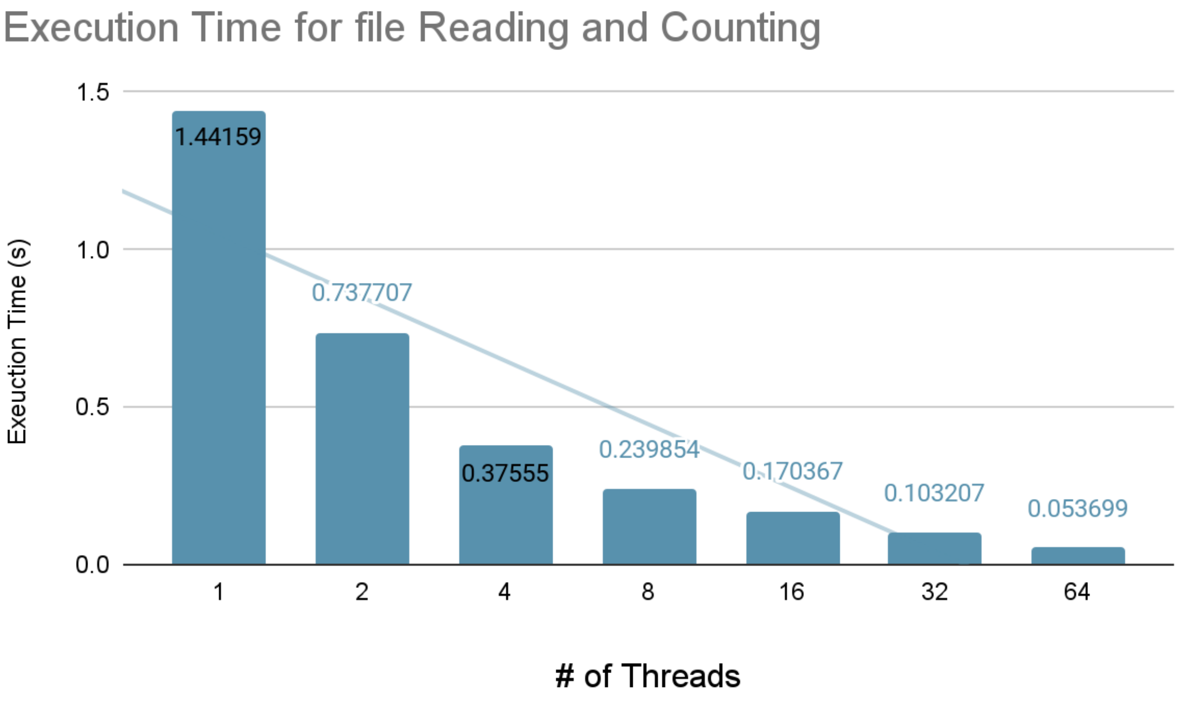
Step 3: Parallelize Building Dictionary

Initially, I implemented a method where each character in the shared frequency dictionary was augmented with a lock. Each thread had its own local frequency dictionary that it updated while iterating through its assigned chunk. At the end, each thread would go through its local frequency dictionary and add the values to the shared frequency dictionary. However, this approach proved to be inefficient as each thread had to acquire 256 locks in order to complete the update process. Additionally, all threads started updating the shared frequency dictionary from the same key, resulting in a significant amount of contention at the beginning of the update. Furthermore, there was a lot of false sharing since consecutive indices in the shared frequency dictionary shared the same cache line. To address these issues, I modified the approach. Instead of acquiring a lock for every key in the local frequency dictionary, it only acquired a lock if the value of the key was greater than 0. This helped reduce the number of locks acquired by each thread. Additionally, to reduce contention at the start of the update phase, each thread started updating the shared frequency dictionary from a different index. This approach proved to be more effective, although there was still some false sharing at higher thread counts. Furthermore, the overhead of constantly acquiring and releasing locks remained relatively high. As a result, I made further improvements. Each thread had its own frequency dictionary, which was also accessible by other threads. The process of building the frequency dictionary was divided into two phases. In the first phase, each thread updated its own frequency dictionary, similar to the first approach. In the second phase, each thread took a portion of the frequency dictionary (based on the number of threads) and added their values across all the frequency dictionaries. The result was then stored in the shared frequency dictionary. This approach completely eliminated false sharing, as threads only accessed the same cache line while reading.

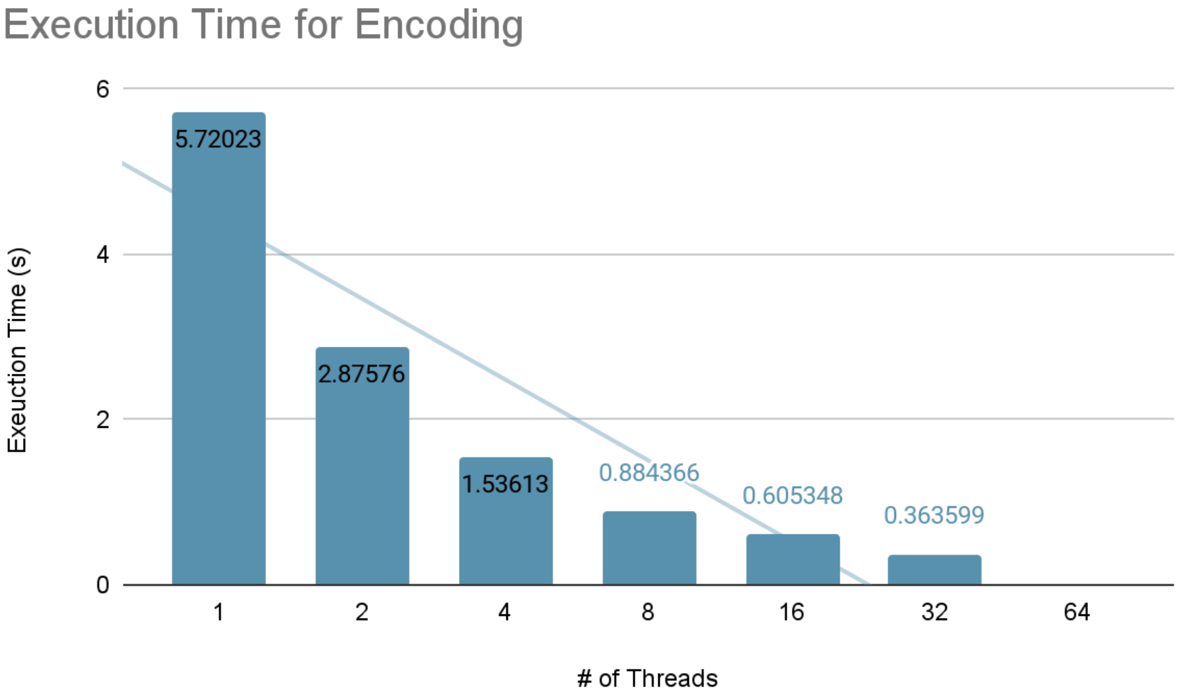
Step 4: Parallelize Decompression
To begin with, the metadata is extracted from the file in order to construct the symbol table in the computer's memory. Subsequently, the file is decoded using the information provided by the symbol table. When it comes to sequential decompression, the majority of the processing load (> 99%) is attributed to file decoding. Initially, I read file offsets, and then each thread can independently decode the corresponding chunk of the file in parallel, without requiring any form of communication. As a result, the speedup obtained during decompression is nearly linear.

Results:

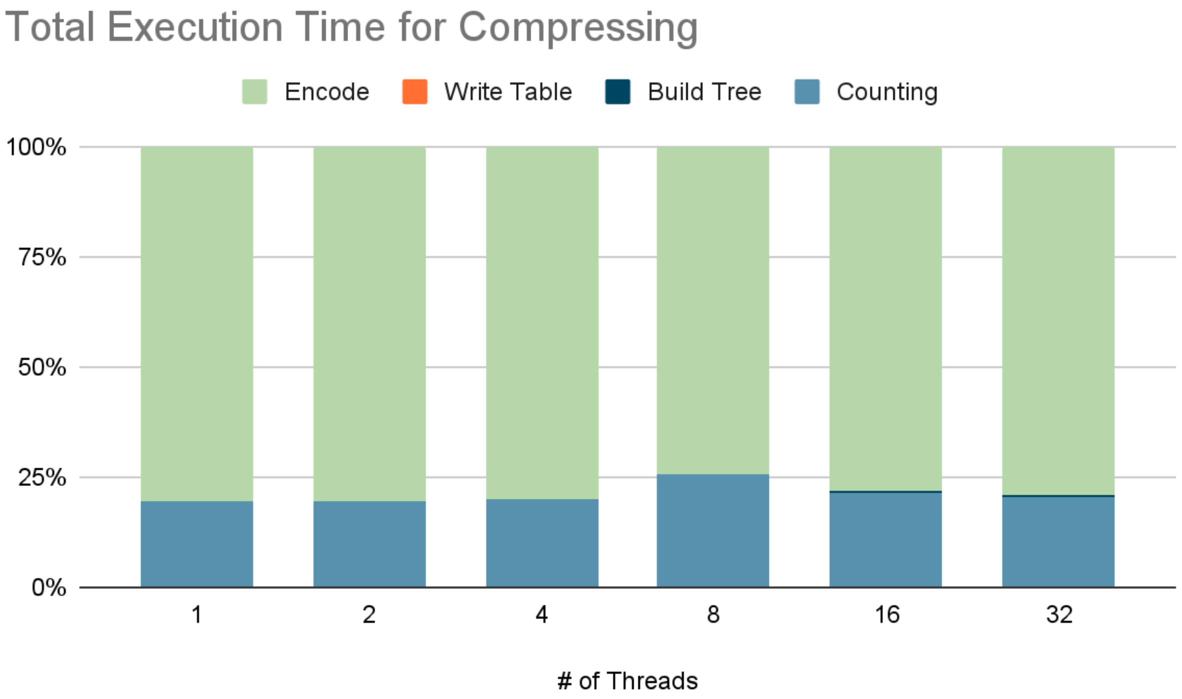
The result for the execution time on reading the file and building the dictionary matches our expectations. I can obtain an almost linear speedup.



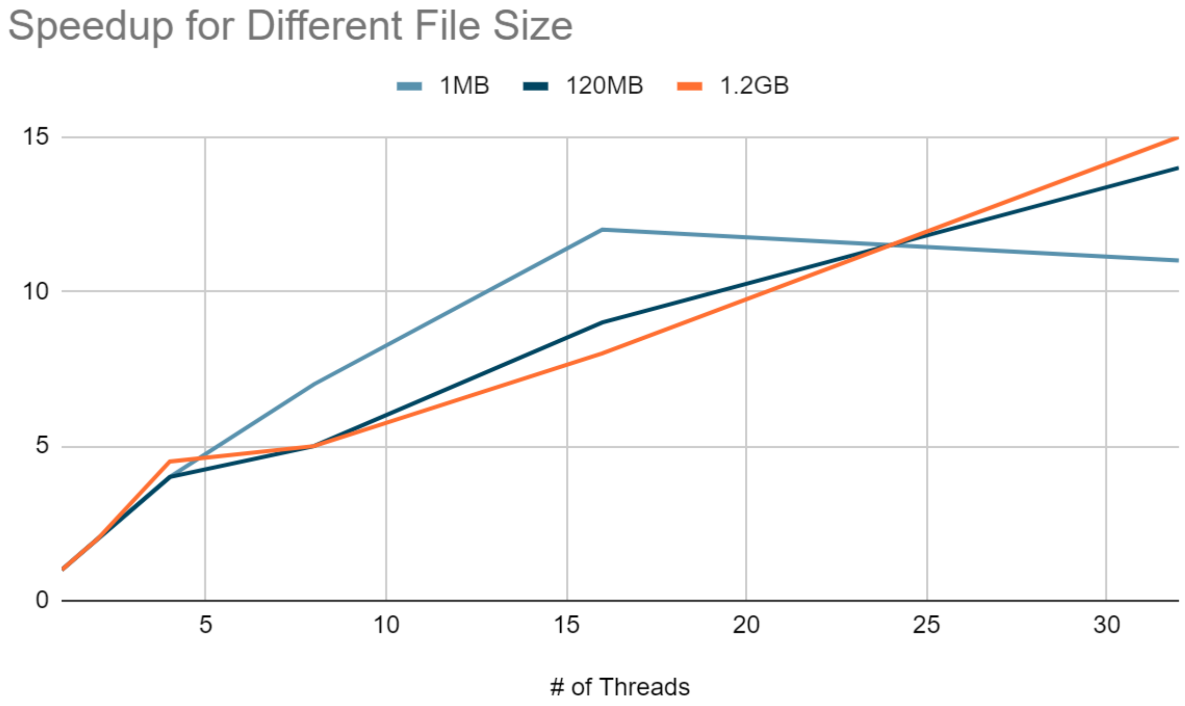
As you can see from the trend line, the result is almost linear.



We can see that the speedup decreases after 8 threads, we think it’s related to Amdahl’s Law. According to Amdahl's Law, even if the encoding phase is perfectly parallelized, the speedup of the entire process is limited by the time taken for the serial part (Huffman tree construction). This means there's an upper limit to the performance improvement that can be achieved with parallelization.



After parallelizing the entire program we can see that there is no bottleneck, the workload has been equally distributed.



The speedup is observed to rise as the number of processors increases, as mentioned earlier. However, when dealing with a small dataset, the rate of speedup begins to decline once 16 threads are utilized. The size of the problem plays a crucial role in this project, as evident from the distinct graphs obtained for small, medium, and large datasets. This is because the construction of the Huffman tree and the creation of the symbol code table (which are sequential tasks) remain unaffected by the problem size. On the other hand, the generation of the frequency dictionary and the encoding of the input depend on the size of the input array, which in turn relies on the size of the input file.

Limitations to Speedup

In order to address memory bottlenecks, I made some adjustments. Firstly, Iloaded both the input array and output array into memory. Additionally, I took special precautions to eliminate false sharing, ensuring that threads primarily accessed the same cache line while reading. This prevented them from writing at offsets from each other. However, as the number of threads increased, I observed an increase in cache misses. This is because the offsets became smaller, resulting in more cache misses. To minimize communication between threads, I introduced preprocessing steps. This ensured that threads already had all the necessary information before entering the parallel section. Although this added a slight overhead, it improved overall performance. Synchronization was also a concern, as the algorithm's four steps needed to be sequential relative to each other. However, it was not the primary reason for the lack of speedup. Based on the data I collected, it appears that the distribution of work across threads is the main factor hindering speedup. This is intuitive, as even though each thread was assigned an equal number of bytes to process, the output length for each character varied. As a result, one thread might end up writing significantly more output bits than another thread, depending on the character distribution in the input file. However, as the number of threads increased, the workload became more evenly distributed due to finer granularity.