Parallel Huffman Encoding

Advanced Parallel Systems

M2 MoSIG

Student:

Abbas Khreiss

Goal

The goal of this project is to parallelize huffman encoding, but instead of trying to parallelize the original huffman encoding, I tried a simple method of running the algorithm more than once ,in parallel, on the file parts instead of trying to parallelize the original algorithm.

Implementation

I implemented the sequential huffman encoding algorithm from scratch using C#. I created a console application that allows us to compress and decompress files either with the sequential version or the parallel one specifying the number of threads to be used.
The code can be found in the Huffman **Project** directory in the project.

- Implemented a min heap myself using a static array to be used in the algorithm.
- Implemented the original sequential Encoding algorithm, it takes one file as input and output one compressed file as output
- Implemented sequential decoding algorithm, it takes one compressed file as input and returns one decoded file as output.
- Implemented a parallel encoding version where I split the file to be compressed into n parts based on the number of threads that I want to use. The parallel version will run the sequential huffman encoding n times in parallel and outputs a directory containing n files.
- Implemented a parallel decoding version where I create n threads, one thread for each part of the encoded algo. Each thread will run the sequential version of the algorithm at the same time and the results will be merged after.

I used the following header for the compressed files:

- The first byte is used to specify the number of entries that we have, we can have max 256 entries
- Three bytes for each entry:

Byte1	Byte2 - Byte3
Byte to be encoded	Binary Encoding + Padding

  I append the binary encoding to the encoded byte then add padding. For example:
  Byte = 00000000 is ecoded as **10**
  The header will be 00000000 | **10** 100000 | 0000000
  The additional byte is needed in case the encoding used 8 bits
- Compressed Data
- Padding (same as table entry padding)

Note: I used windows during development and experimentation. I used Visual Studio 2022 for development and publishing the code.

Experiments

I created some scripts in python for running multiple experiments and plotting graphs.
They can be found in the **Experiments** directory in the project. We tried compressing files with alphanumeric data and some symbols since if we generate bytes uniformly and all the possible 256 bytes are used, Huffman encoding won't work since all the binary encodings will be formed of 8 bits.
Used characters:
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.!? "

gen_data.py

Usage: python gen_data.py <size_in_MB> <file_type> <destination_path>
This scripts generates a file of specific size based on specific method:

- **uniformly_random**: for each byte generated, we will pick one of the allowed characters at random
- **all_same**: One of the allowed character will be picked at random and it will be duplicated
- **ten_chunks**: we pick 10 allowed characters at random and we split the file into 10 parts, each part will be formed of 1 of the characters

gen_all_data.py

This script generates multiple files of different sizes with different methods automatically

evaluate_wrt_threads.py

Usage: python evaluate_wrt_threads.py <huffman_script_path> <max_threads>
This scripts takes the path of the compiled binary version of my c# application in addition to the max number of threads, then it runs experiments (compression and decoding) with the sequential version and the parallel version trying all threads from 1 till max_threads on all files starting with **uniformly_random** in **ToCompress** directory. It outputs logs to stdout which can be redirected to file.

evaluate_compression.py

Usage: python evaluate_compression.py <huffman_script_path> <parallel_threads>
Almost the same as above, but it runs the compression only. It runs the sequential encoding once and the parallel version once with the specified number of threads and output logs to stdout.

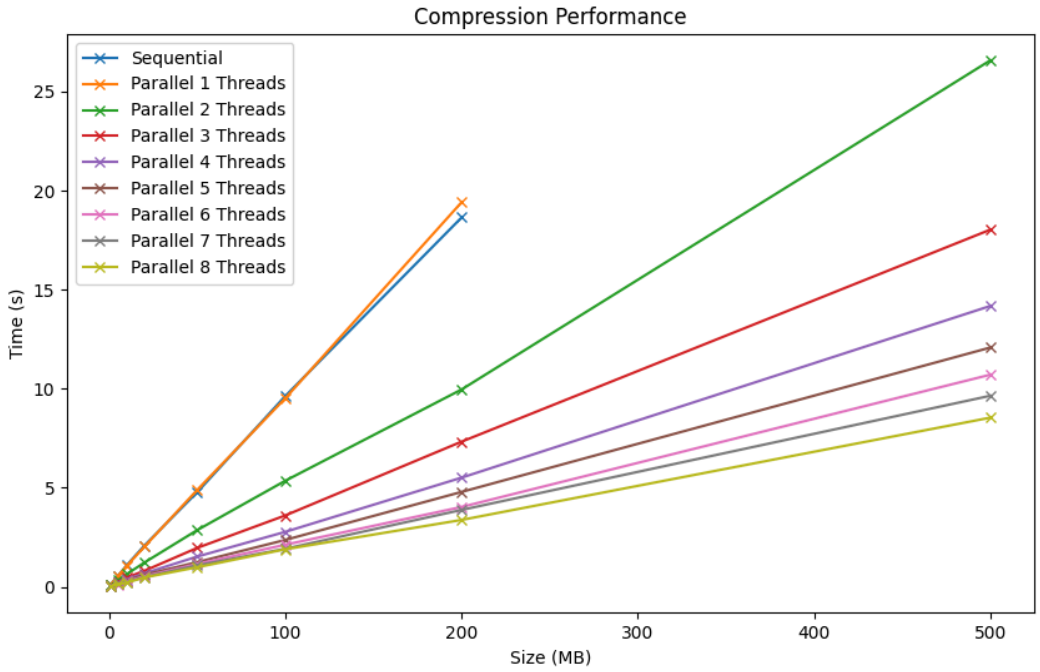
process_data_and_gen_graphs_compression_ratio.py

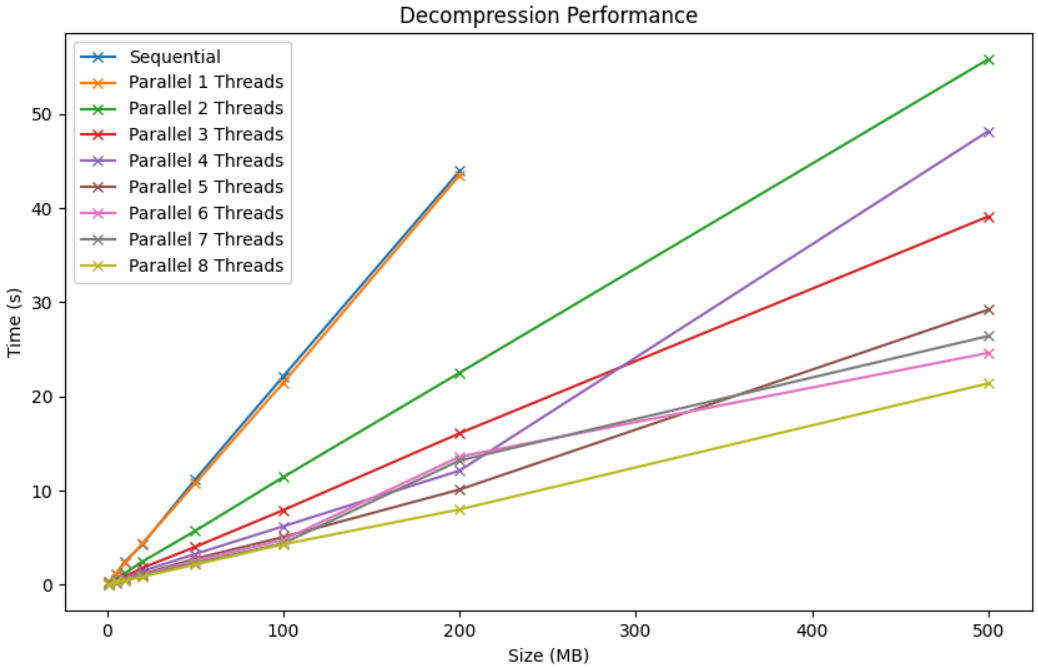
This script takes one file as an argument. The file should contain the logs generated by evaluate_compression.py script, it will process them and generate some of the graphs displayed below.

process_data_and_gen_graphs_wrt_threads.py

This script takes one file as an argument. The file should contain the logs generated by evaluate_wrt_threads.py script, it will process them and generate some of the graphs displayed below.

Compression Performance





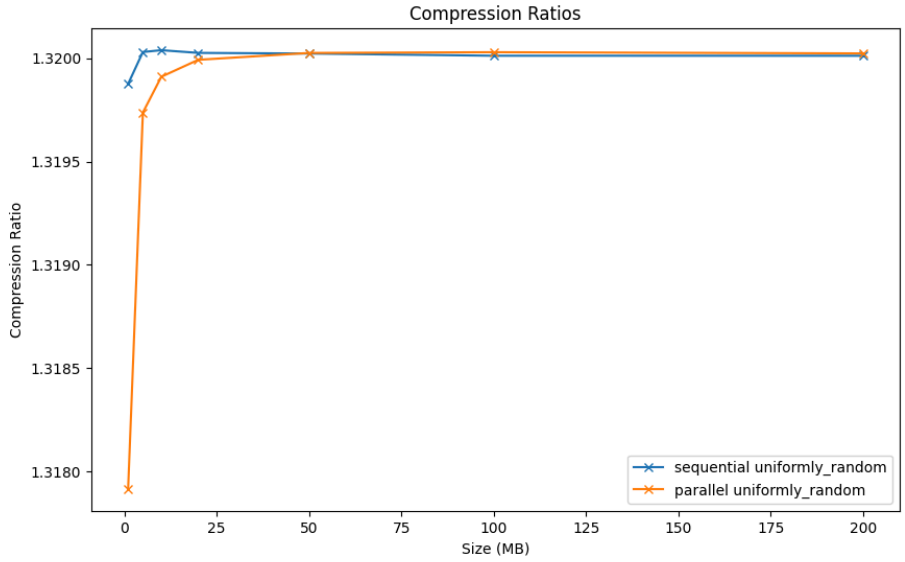
We can see that the parallel version was able to scale very well when using multiple threads. I ran those experiments on my own laptop with 8 cores.
The last point (size = 500MB) is missing for the sequential run because of a limitation in my implementation.

I wanted to run more experiments on Grid5k but I didn't have enough time for that.

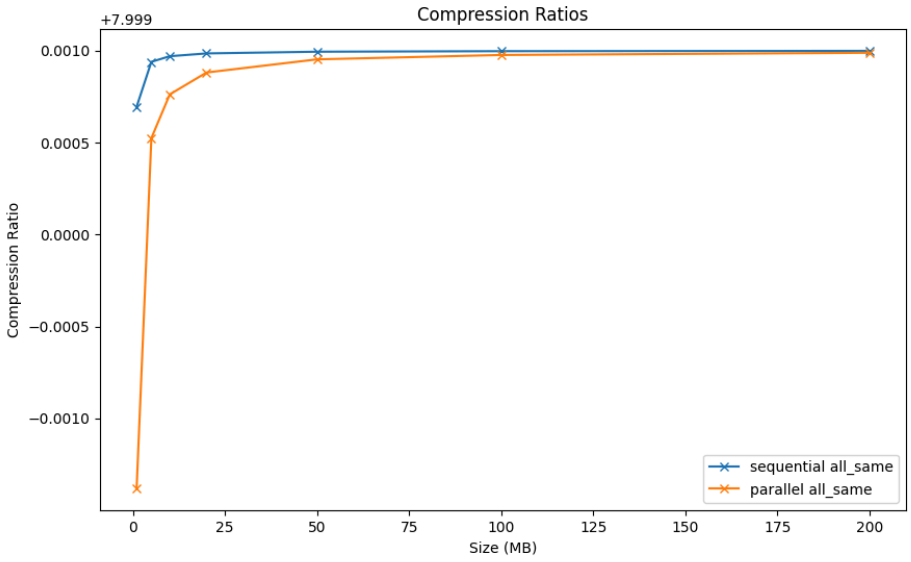
Compression Ratios with different data patterns

I ran the sequential version of the algorithm in addition to the parallel version of the algorithm with 8 threads against multiple file sizes and I obtained the results below.
Compression ratio = original_size/compressed_size

Uniformly Random

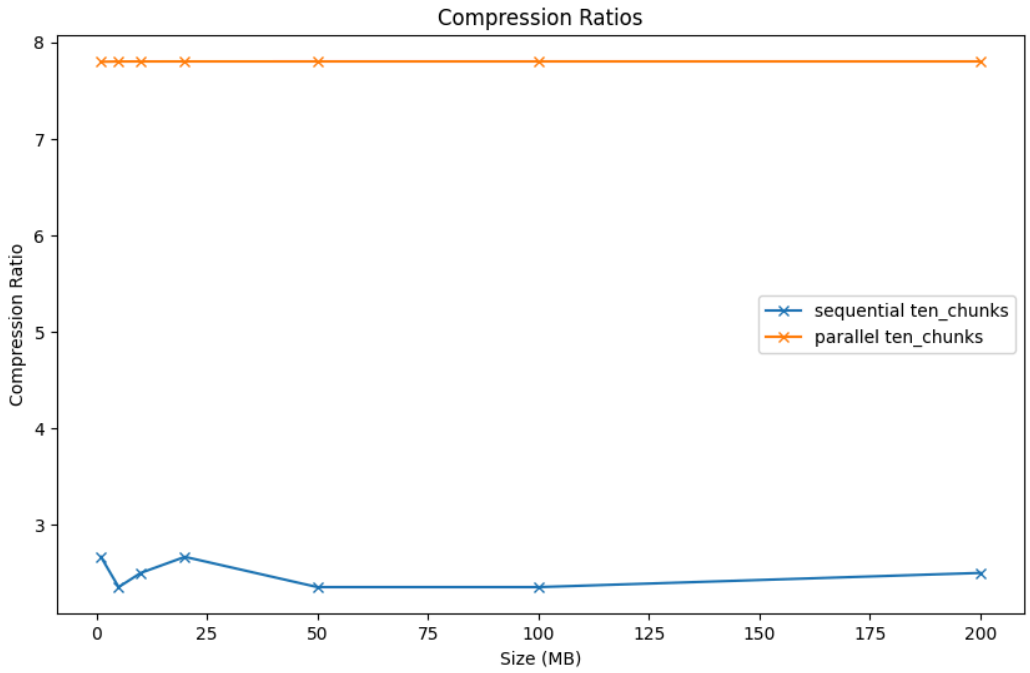


All Same



As we can see from the above graphs, when we have small files compression ratios of the parallel version are a little bit less than the sequential version because of the additional headers for each part. However for big sizes, they are almost the same. Even in the uniformly random pattern, with big files, parallel compression ratio was slightly better.

Ten Chunks



For this data pattern, where we have contiguous chunks of similar data, the parallel version compression ratios were much better than the sequential one because we will have less entries in the encoding table and even the binary encoding will be much shorter.

Alternating bytes

I didn't have time to experiment with it but assuming that we have alternating data pattern as: ABCDEF…ABCDE…
In this case, I believe that the ratios of the sequential version will be much better for small sizes. However, they should be similar for big files.

Conclusion

The provided parallel version of huffman encoding could be great for parallelizing the original huffman algorithm when working with big files since it can scale very well. We should always remember what our goal is from the algorithm that we are using, it is not necessary to keep the same logic, we can tweak it to achieve the same goals that we already wanted.

Possible Extensions

- Using Grid5k for running more experiments with more powerful machines
- Experimenting with performance on small files, we should use the sequential version for files under specific threshold
- **I thought about an interesting idea, I am not sure if it is already studied or if it is even useful in practice. Usually we try scaling computational power horizontally and vertically for improving performance. I was thinking that for big files, writing the compressed files to disk could create a bottleneck on the critical path. I could experiment with scaling disks horizontally instead of vertically. So, when a big file is compressed, each one of its parts being compressed in parallel is stored on a separate disk. This could create some complexities handling the files but it could improve the performance much since we will be doing IO in parallel. I am not sure if it was studied before or even if it is useful but it was an idea.**