Queensland University of Technology

CAB401 - High Performance and Parallel Computing
Semester 2, 2018

Parallel Huffman Encoding Report

Spencer Huston
n10144226

Table of Contents

Motivation

After some careful deliberation and searching for a project to parallelize I looked to previous programs I had written for classes in the past. Immediately I came upon my implementation of Huffman Encoding, and having thoroughly enjoyed the original project I thought it best to attempt to parallelize it to the best of my ability. Being a compression program that is able to be used on data of any size, it easily qualifies as computationally intensive. I developed my project on my home university's computer labs remotely.

Specifications:

Model:	Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz
CPU(s):	32
Thread(s) per core:	2
Architecture:	x86_64

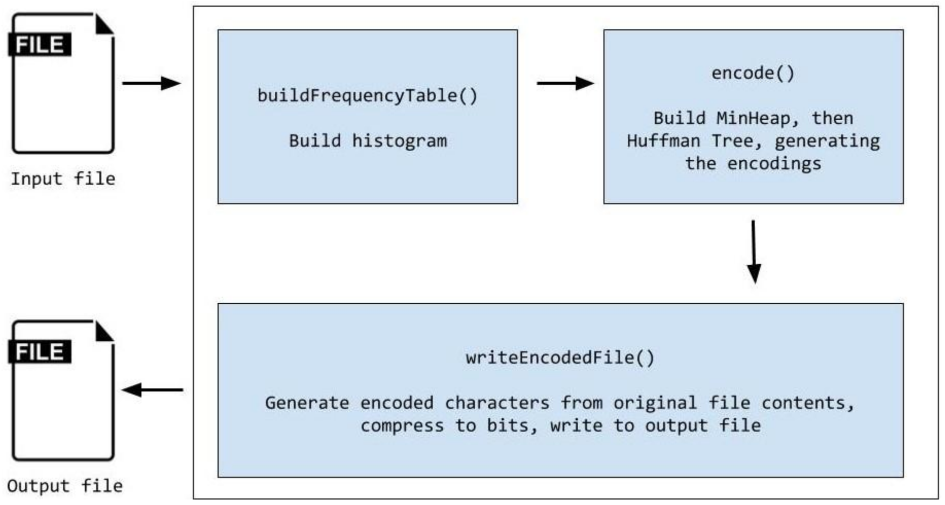
Huffman Encoding Overview

Huffman Encoding is a well established lossless compression algorithm. The premise of the algorithm is that characters (raw bytes in this instance) with a higher frequency of occurrence will receive smaller codes, and vice versa with less frequent characters. The special property about Huffman encoding, or more specifically a Huffman Tree, is that no code will ever be a prefix of another, which makes for efficient decoding.

The general architecture of the application itself goes as follows:

1. Construct a histogram of frequencies of the bytes of the given file
2. Construct a MinHeap data structure using the histogram
3. Construct a Huffman Tree using the MinHeap
4. Generate the output for the new file by replacing each character with its corresponding Huffman code
5. Write this output to the new file

Given that we want to compress the data as much as possible, there are some finer details that are more programming specific. The first of these requires the encoded data to be “padded” with extra 0’s on the end so as to make the length of the data to be modulo 8, which is necessary for the next step. Second, is the need to convert the 0’s and 1’s from characters to raw bits. These last few steps are usually left out of the algorithm since they are more related to the application itself.


FILE
Input file
buildFrequencyTable()
Build histogram
encode()
Build MinHeap, then Huffman Tree, generating the encodings
writeEncodedFile()
Generate encoded characters from original file contents, compress to bits, write to output file
FILE
Output file

Data flow of the main component of the program, `Encoder.cpp`

Analysis

Upon inspection of my original implementation, it was not entirely clear where to start. Most of the loops in the program were `while` loops and of the `for` loops in it, they had quick computations in them and all were bound by 256 except for the last one. I did some further timing on each section which revealed major bottlenecks in the overall execution time. These included file I/O, reassigning the original text with its Huffman codes, and "compressing" the Huffman code text into a binary form. On top of this, I noticed many inefficiencies regarding data that could be saved and moved in between class methods.

Of those problems, file I/O and converting to binary were by far the worst perpetrators. The original combination of these two happen at the very end, where I had originally fused them into a single step. Ironically enough, the actual computation regarding construction of the MinHeap and Huffman Tree were relatively insignificant in their percentage of total execution time. On top of this, given the nature of both data structures being trees, any attempted parallelization would be *very* difficult and not at all worth it in my opinion. Another combined step, reading from the file and constructing the histogram, also took up fairly little time when compared to the other bottlenecks present at the end of the program.

A slight restructuring was available with how the data was being handled immediately before and after file I/O, the specifics of which will be discussed in the next section. The algorithm itself was unable to be altered due to its very strict data dependencies as a result of its refined nature. In specific, the MinHeap cannot be constructed until the histogram is completely finished, the Huffman Tree cannot be constructed until the MinHeap is completely finished, and so on.

Parallelization

Constructing the Huffman encoded string

I first took steps towards parallelization by splitting up the process of assigning the original contents with its Huffman encoding. Because I combined reading the file and constructing the histogram together, the file contents were not saved in memory and therefore my original implementation consisted of reopening and rereading the entire input file over again to assign each character its code.

Original code from `Encoder.cpp`, in `writeEncodedFile()`:

```
FILE * input_file = fopen((in_path).c_str(), "r");
...
int c;
string text_data;
while ((c = fgetc(input_file)) != EOF)
{ text_data.append(huff->getCharCode(c)); }
fclose(input_file);
```

I immediately fixed this by saving the contents in a buffer in `buildFrequencyTable()` and then read through that buffer instead of the file itself again. Although this saved some time on file I/O, the bulk of the time was taken up in calling to `getCharCode()` for every single character. This was resolved by parallelization of course. The file contents were divided as evenly as possible between a given number of threads which each constructed its own string of Huffman encodings. Following this, all of these partitions were then appended together in order. A slight issue arose where the length of the contents was not able to be divided perfectly by the number of specified threads. To combat this, the last thread in the pool was to compute its own section and then the remainder as well, computed by *content.length modulo number_of_threads*. This fix worked out fairly well, resulting in a linear speedup up until 32 threads:

Huffman code construction speedup

Threads	1	2	4	8	16	32	64
Time	0.7487	1.1478	1.5209	2.2640	3.0127	2.5287	2.5422

Resulting code from parallel `Encoder.cpp`, the 2nd and 3rd part in `writeEncodedFile()`:

```
void build_text_data(int i)
{
    string local_text_data;
    for (int j = lsections[i]; j < lsections[i] + lsection_sizes[i]; j++)
    { local_text_data.append(huff->getCharCode(buffer[j])); }
    appends[i] = local_text_data;
}
...
bool lremainder_exists = (lsize % NUM_THREADS != 0) ? true : false;
int lsection_size = lsize/NUM_THREADS;
int lremainder = (lremainder_exists) ? lsize - (lsection_size * NUM_THREADS) : 0;
...
for (int i = 0; i < NUM_THREADS; i++)
    threads[i] = thread(build_text_data, i);

for (auto &t: threads)
    t.join();

for (int i = 0; i < appends.size(); i++)
    text_data += appends[i];
```

Bit-shifting and File I/O

While that was nice speedup, it was only a portion of the code responsible for the overall execution time, and the more glaring issue was still at hand concerning the output to the file. As stated, the original code consisted of combining both bit-shifting to pack the bytes and writing out to the file.

Original code from `Encoder.cpp`, at the end of `writeEncodedFile()`:

```
for (int i = 0; i < text_data.size(); i += 8)
{ output_file.put(bitset<8>(text_data.substr(i, 8)).to_ulong()); }
```

At the time of writing the original project time efficiency was not an issue we had to deal with, which resulted in this one loop so *extremely* slow it took up, on average, 79% of execution time. To first go about overcoming this, I had thought to try to perform the loop, as it was, in parallel. I quickly ran into issues regarding setting pointers in the output file correctly for each separate thread to start writing to, and thought it best to instead leave the file I/O as serial access. This turned out to be the far better choice.

I turned towards preprocessing the data some more. I removed the bit-shifting out of the loop to its own loop, which could be parallelized much easier, practically in the same fashion as I did with constructing the Huffman encoding. Just like it, I had to partition the data as evenly as possible per the number of threads. However one issue with this case is the fact that each section *has* to be modulo 8. For example: if I had a string of say 304 0's and 1's split between 8 threads, I could not just process 38 per thread, as 38/8 = 4R6. Because of the necessity for *every* bit to packed into a byte without generating extra 0's to fill the empty space (6 for every 5th byte in this example), I instead split up it up as: (*encoding.length* - *remainder*)/*number_of_threads*. This insures each thread had the same amount of data to work with that could all evenly be packed into bytes with no extra 0's, also therefore retaining the correct encoding. For the remainder, I spawn an extra thread to process that.

Following this, I had a slight revelation where I realized it was not the file I/O that was taking up so much time, but the bit-shifting itself. So by removing that step from the original loop and processing it in parallel, I received a massive speedup. On top of this, the manner I wrote into the file was altered. Instead of using `output_file.put(...)` for every single byte, I allowed for greater throughput by changing it to write the entirety of each section at once.

Huffman bit-shift + file output speedup

Threads	1	2	4	8	16	32	64
Time	0.9647	1.5773	2.1918	4.0204	7.0686	7.3270	10.1182

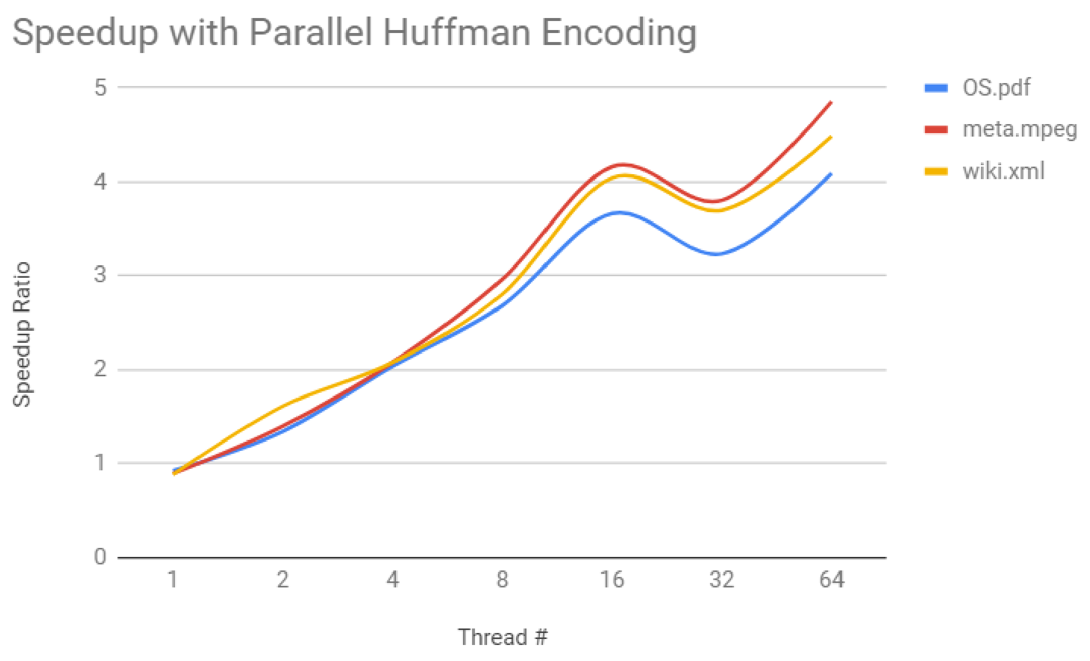
Resulting code from parallel `Encoder.cpp`, 2nd and 3rd parts in `writeEncodedFile()`:

```
vector<long> sections;
vector<long> section_sizes;
vector<string> full_bitset;

void build_bitset(int i)
{
     for (int j = sections[i]; j < sections[i] + section_sizes[i]; j += 8    )
     { full_bitset[i] += (char)(bitset<8>(text_data.substr(j, 8)).to_ulong()); }
}
...
int remainder = text_data.size() % (NUM_THREADS * 8);
long section_size = (text_data.size() - remainder)/NUM_THREADS;

//extra thread that takes care of leftover bits
int thread_size = (remainder != 0) ? NUM_THREADS + 1 : NUM_THREADS;
...
for (int i = 0; i < threads.size(); i++)
        threads[i] = thread(build_bitset, i);

for (auto &t: threads)
        t.join();
```

Speedup

Altogether, my `writeEncodedFile()` looks almost completely different. While my original function is about 44 lines of code, the new version is up to 104, not including an extra 22 lines for the functions that envelop the original `for` loops. Along with this restructuring and parallelizing, some of the variables that were previously local to the function only had to be moved to be global to `Encoder` (`buffer`, `huff`, etc.). The culmination of all these modifications has led to a very nice speedup. As displayed in the tables above, it is definitely linear, with bit-shift + file I/O perhaps even slightly above linear.

The overall speedup graph displays this as well:



OS.pdf, meta.mpeg, and wiki.xml were my main data sets. OS.pdf is 9.2MiB pre-compression with resulting 18% compression, meta.mpeg is 64MiB with 16% compression, and wiki.xml is 103MiB with 36% compression.

As expected, the parallelized version runs slightly slower than the sequential one at just one thread due to the extra processing overhead. Also, as is visible, the speedup seemingly drops significantly below linear upon reaching 32 threads, and then takes off again at 64. This is slightly misleading, as the Huffman encoding construction actually not only drops at 32, but stays there at 64. Vice versa, bit-shift+output also drops at 32, but then continues at a slightly-above-linear pace when reaching 64. Being that the latter takes up significantly more time than the first, the speedup tends more towards the speedup of that individual component, i.e. not plateauing as much as it might seem. While the drop *seems* significant, it is actually along a linear trend line that 16 threads surpassed.

Frameworks and Tools

Given that my project was constructed in C++11 and that is what I am most familiar with I stayed with that to further develop the project rather than rewrite it in some other version. I also normally develop on my Ubuntu Virtual Machine, but because this project necessitated I use my actual machine I instead switched to Cygwin. Also do note I did not do it in Visual Studio or Eclipse because I personally develop much better on the command line than in an IDE. Continuing, I did a portion of the project before finding out that Cygwin actually contains a unfixed bug that does not allow for multicore processing. So I then switched to my home university's lab computers, the specifications of which have already been listed. This was a plus for me because I much prefer Linux over Windows for projects like this.

In terms of frameworks, although C++11 threads do not allow for their affinity to be set manually, I kept developing with it. Affinity in C++11 is abstracted away from the programmer and is handled automatically by the operating system, which although gave me less control, was easier to work with. Tools-wise I used `gdb`, `gmon`, and `std::chrono` for debugging, profiling, and timing respectively. I have used `gdb` many times before, and it is always a must use. However for this project I used it more to see specific amounts of data for different cases to see how to split it among the threads, but eventually shifted away from it and to print statements once I actually implemented the multithreading. I used `gmon` only a few times and honestly did not get that much out of it. I did spot the `bitset<..>` being called frequently but at the time never realized that it was running nearly as long as it actually was, although the profiling in general did point to which functions to look at (`buildFrequencyTable()` and `writeEncodedFile()`). Using `std::chrono`'s `high_resolution_clock` was extremely useful for timing critical sections of the code, as well as revealing that the bit-shifting was taking up majority of the time.

Conclusion

All in all, I believe the linear speedup achieved was much better than I expected. If I had to do it all over again I would definitely research how to best do multiprocessor programming on the environment I will attempt to work on, so as to avoid something like the Cygwin bug again, as well as C++11's lack of manual thread affinity.

With this, I would have liked to have delved a little more into the areas I chose not to parallelize because of how fast they already were: MinHeap, Huffman Tree, file I/O. Specifically I would have liked to work with at least attempting constructing a Huffman Tree in parallel, as I believe that could be a very interesting project. However, I believe that would be quite a large project in and of itself. On top of all of that, I could have probably attempted to parallelize `buildFrequencyTable()` as it was consistently around half the time of the Huffman encoding construction, but again forego to because, relatively, it was insignificant.

Lastly, I do believe I achieved as well a speedup as was possible with this program. Given the nature of Huffman encoding, and how straightforward a process it is, there is not much to directly parallelize within the actual *algorithm* itself, but rather it is better to focus on processing the data before and after encoding to allow for it to be worked on by many threads simultaneously.

Extra Notes

1. The "resulting code" snippets both do not include a large chunk of the preprocessing code for assigning sizes of data to process for the threads.
2. `writeEncodedFile()` is basically the only function drastically altered. `buildFrequencyTable()` was slightly changed but it performs the exact same actions as it did before, only now it retains the file contents in a buffer used for later.
3. A large amount of "fluff" code is still in my source files in the form `std::chrono` used for timing purposes (also kept for anyone using it to see the speedup themselves)
4. Page numbers are slightly off because none of my word editors allowed for starting them on the 3rd page