The Report of Huffman Coding Homework

Chung-Yi Chen
311551070

Abstract

In this homework, I implemented the encoding part of basic, adaptive, and extended Huffman coding algorithms in C++. These implementations are highly optimized for speed from the aspects of algorithmic design and parallel execution. Using the given data source, the experiments showed that 1) these three algorithms have almost the same performance, 2) the adaptive Huffman coding is dramatically slower than the basic Huffman coding, and 3) the optimizations on these implementations are effective.

1 Implementations

The programming language used in this homework is C++ for several reasons, but the most important one is that C++ is not only fast but also provides fine-grained control of memory, which is essential while the size of the alphabet set becomes considerably large. Besides, since the basic Huffman coding algorithm is highly parallelizable, I utilized OpenMP[Ope21] to implement a multi-threaded version of it. Most mainstream compilers have their OpenMP implementations.[1] To plot the experiment result, a header-only wrapper [2] for Matplotlib[Hun07], which is a powerful graph plotting library written in Python, is used.

1.1 Huffman Coding

The basic Huffman coding is a two-pass algorithm. The first pass reads the entire sequence of symbols to build a frequency table and constructs the Huffman tree to finally obtain the coding of each appeared symbol. Then the second pass encodes the entire sequence into codewords according to the coding table. Here are four points to be considered.

- Symbol Reading
- Frequency Table
- Tree Construction
- Coding Table Construction

1.1.1 Symbol Reading

Although it is only required to treat the given file as a source of 8-bit and 32-bit symbols, my implementation supports an arbitrary number of bits as the symbol length.

1.1.2 Frequency Table

The data structure of the frequency table does matter. If just simply use an array to store the number of occurrences of a symbol, let n denote the length of a symbol, the size of the array will be 2^n. For example, taking a 32-bit integer to store the occurrence of a symbol results in 16GB memory usage when n is 32.

A practical way is using a hash map instead of an array. This way works because the number of appeared symbols is quite small compared to 2^n when n is large. However, accessing a hash map takes longer time than an array, and thus if the size of the hash map is larger than a specific value, which is $\frac{2^n}{100}$ in use, the frequency table will be transformed to an array for better access speed.

1.1.3 Tree Construction

After the frequency table is built, the construction of the Huffman tree can be started. The Huffman tree is constructed in a bottom-up manner, each time we pick two nodes with the smallest and second to the smallest occurrences and merge them into a new node with a larger occurrence until there is only one node left, so the priority queue is an efficient data structure to achieve such the requirement, with the time complexity $O(v \log v)$, where v is the number of the appeared symbols.

Nevertheless, the priority queue is hard to parallelize by the nature of the binary heap. To get better performance, I instead use another method[3] to construct the Huffman tree.

This method involves two lists. One list stores all leaf nodes of the tree and the other one stores all non-leaf nodes. Call them leaf list and internal list respectively. In the beginning, all appeared symbols are stored in the leaf list and the internal list is empty. Then the leaf list is sorted in $O(v \log v)$

[1] https://www.openmp.org/resources/openmp-compilers-tools/
[2] https://github.com/lava/matplotlib-cpp
[3] This method is described in the compression section. https://en.wikipedia.org/wiki/Huffman_coding#Compression

time. After that, the Huffman tree can be constructed in linear time. In each step, by examining the head of the two lists twice, we get the desired two nodes and generate a new node for them. Then insert this new node into the tail of the internal list. Repeat this procedure until there is only a node left in the internal list, and the Huffman tree is obtained.

Although the two methods have the same time complexity, the latter method can be parallelized well. The reason is that the sorting can be done by using the parallelized merge sort I implemented. Besides, since the two lists are iterated sequentially, this method is more cache-friendly than the priority queue. And thus is superior to the priority queue implementation.

1.1.4 Coding Table Construction

The coding table can be generated by recursion. Since the tree is not modified during recursion, this can be parallelized easily. And because it is only required to calculate the expected codeword length, this step in fact only finds the codeword length of each symbol to speed up the execution and reduce the memory usage at the same time. Despite this, it is easy to modify the source code of this step to generate the real codeword when necessary.

1.2 Adaptive Huffman Coding

Unlike basic Huffman coding, adaptive Huffman coding is hard to optimize in a parallel manner, as suggested in the flowcharts given in the textbook[Say17]. However, if implement the given flowcharts without extra care, the speed performance will be dramatically bad. For example, the flowchart for encoding says that if a symbol appears more than once, then its code is the path from the root node to its corresponding node. If directly starting from the root node to the corresponding node, the average time complexity will be $O(v)$ since all leaf nodes have to be checked. But with the help of an auxiliary hash map that maps a symbol to its corresponding node, the time complexity of traversing from the corresponding node back to the root node is $O(\log v)$, since only the nodes in the path from the root to the corresponding node are visited.

Another similar case is how to find the max node of a block as required in the update flowchart. A naive way is starting from the root node and searching all the nodes with the same occurrence and then comparing all of them to find the max node. This needs $O(v)$ to complete in the worst case and probably the average case I believe. But with the help of the binary heap I implemented, we can maintain a hash map that maps the block number to a binary heap of all nodes having that block number.

By doing so, a query of the max node given the block number only costs $O(1)$. This improvement is crucial for reducing overall time consumption because there are $v \cdot O(\log v) = O(v \log v)$ queries of an update. When the number of updates becomes larger, the disadvantage of $O(v)$ query becomes more obvious. For example, let the length of a symbol be 16 bits, then the number of updates is 122,204,600. It is nearly impossible for the $O(v)$ implementation to be complete in a reasonable time, but it is possible (and actually does) for the $O(1)$ implementation.

1.3 Extended Huffman Coding

Extended Huffman coding follows the same optimizations as described in Huffman Coding.

2 Experiments

The program should be compiled by a C++ compiler supporting C++20 or above standard because it used some features introduced since C++20. It is acceptable if the C++ compiler does not have OpenMP support and the execution environment does not have Python and Matplotlib installation. But in exchange, it will run in serial and will not output figures.

In the following sections, the data and figures are generated by this program. It took 465 minutes to finish on an Intel® Core™ i7-1260P processor with 32GB main memory.

If not specified, the optimized version of each implementation is used and the data size is 233MB, i.e., the given data is not separated into several trunks.

2.1 Huffman Coding

The expected codeword length of 8-bit and 32-bit symbols are 6.98806 and 8.91635 bits respectively. Dividing the given file into several 40MB trunks only gained little benefit, because the expected codeword length of 8-bit and 32-bit symbols are 6.96842 and 8.85179 bits. The reason is that the PMFs of the whole data (233MB) and 40MB trunks are similar, as Figure 1 and Figure 2 show.

Another experiment is the time consumption of the naive implementation and the optimized implementation described in previous sections. Figure 3 shows that the optimized version has a better running speed.

Figure 4 is the performance metrics of different symbol lengths from 1 bit to 127 bits. The compression ratio is given by $\frac{\text{Uncompressed Size}}{\text{Compressed Size}}$. When the symbol length increases, the compression ratio also increases, however, considering the size of

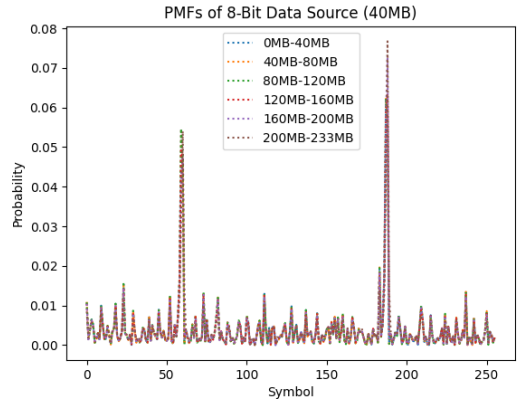

Figure 1: PMFs of 8-bit data source (40MB). PMF of 8-bit data source (233MB) looks similar.

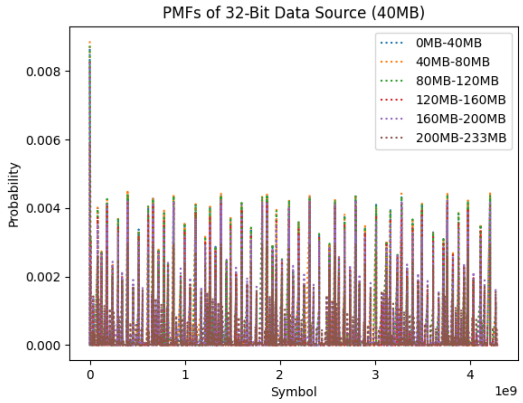

Figure 2: PMFs of 32-bit data source (40MB). PMF of 32-bit data source (233MB) looks similar.

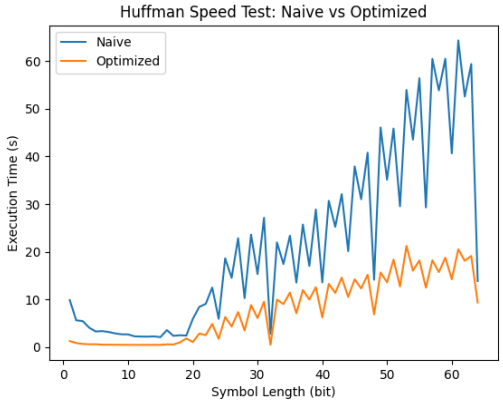

Figure 3: The time consumption of different implementations under different symbol lengths.

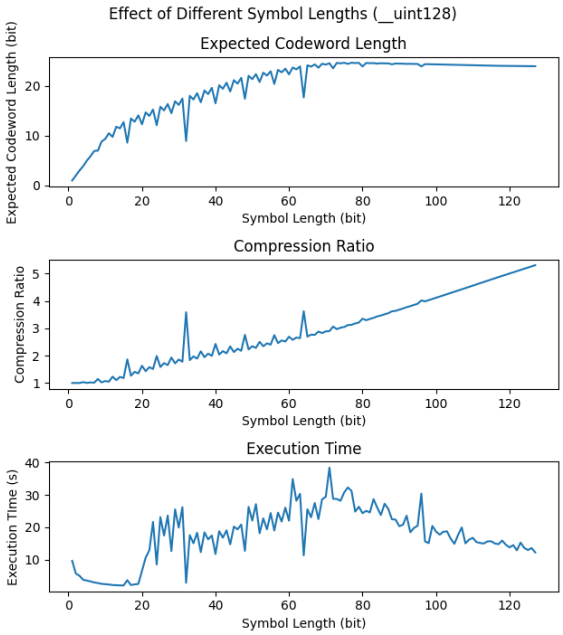

Figure 4: The performance metrics of different symbol lengths. __uint128 is used to present a symbol.

	Symbol Length	
Trunk Size	8 bits	32 bits
233MB	6.9881	8.9204
40MB	6.96854	8.8609

Table 1: Expected codeword length of different trunk sizes and symbol lengths.

the coding table, 32-bit may be the best choice for compressing the given file.

2.2 Adaptive Huffman Coding

Table 1 records the expected codeword length of different trunks sizes and symbol lengths. Although adaptive Huffman coding spent much more time than basic Huffman coding, it did not gain better performance.

Similarly, the optimized implementation (with $O(1)$ query enabled) of adaptive Huffman coding is dramatically faster than the naive version (with $O(1)$ query disabled, it would be too slow if disable all optimizations) when the symbol length is larger than four as shown in Figure 5. The naive version is faster when the symbol length is smaller than five. The reason is that in this case, the tree is not deep enough to benefit from the optimization due to the cost of maintaining a map of binary heaps.

Figure 6 shows the performance metrics of different symbol lengths from 1 bit to 24 bits. It had a similar but not better performance to basic Huffman coding with a much longer time. This fact suggests that adaptive Huffman coding is worse if

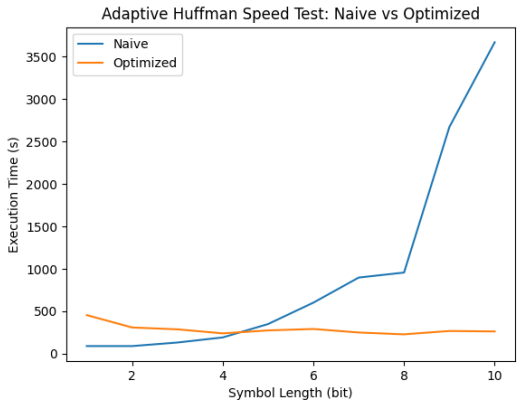

Figure 5: The time consumption of different implementations under different symbol lengths of adaptive Huffman coding.

	Symbol Length		
Extended Size	8 bits	16 bits	32 bits
1	6.98806	8.59072	8.91635
2	13.9513	17.1592	17.8148
3	20.9125	-	-

Table 2: Expected codeword length of different extended sizes and symbol lengths.

the appearance of a symbol follows the same probability distribution when compared to basic Huffman coding.

2.3 Extended Huffman Coding

Extended Huffman coding also cannot gain a better performance as shown in Table 2 and Table 3. The extended size indicates the number of times of symbol repetition. Since the number of appeared symbol grows exponentially, 3× extended size was only tested on 8-bit symbol length. When the extended size is 1, it is the same as basic Huffman coding.

3 Conclusion

In this homework, I implemented three Huffman coding algorithms. Since basic Huffman coding is highly parallelizable, I utilized OpenMP to enable

	Symbol Length		
Extended Size	8 bits	16 bits	32 bits
1	1.14481	1.86248	3.58891
2	1.14684	1.86489	3.59252
3	1.14764	-	-

Table 3: Compression ratio of different extended sizes and symbol lengths.

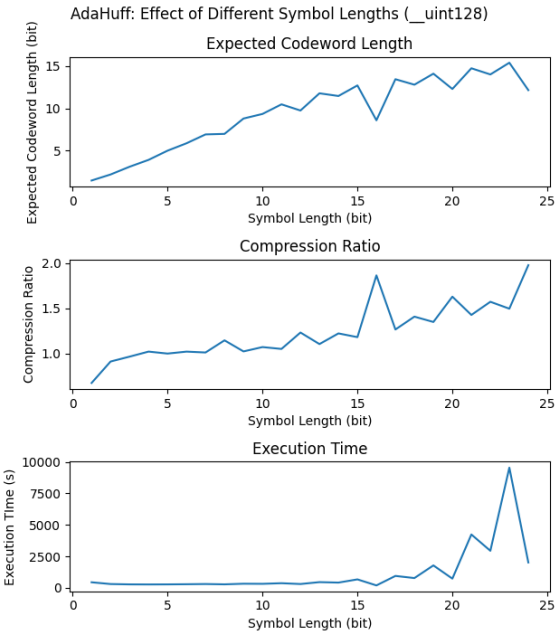

Figure 6: The time consumption of different symbol lengths of adaptive Huffman coding.

the multi-threading ability. For adaptive Huffman coding, the algorithmic improvement shows its superiority over the naive version. The experiments showed that basic Huffman coding has the best performance, this may be because the given data source follows the same probability distribution.

References

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[Ope21] OpenMP Architecture Review Board. OpenMP application program interface version 5.2, November 2021.

[Say17] Khalid Sayood. *Introduction to Data Compression, Fifth Edition*, pages 67–73. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2017.