Huffman Compression

CS F422: Parallel Computing
AY 2022-23, II Semester

Dhruv Rawat (2019B3A70537P)

April 30, 2023

1 Huffman Encoding

Huffman encoding is a lossless data compression algorithm that uses variable-length codes to represent characters in a file. The algorithm works by constructing a binary tree of nodes, where each leaf node represents a character and the path to the root node represents the Huffman code for that character. The Huffman code for a character is determined by the frequency of the character in the file being compressed. Characters that appear more frequently are assigned shorter codes, while characters that appear less frequently are assigned longer codes. This results in a more efficient encoding of the file, as the most common characters require fewer bits to encode.

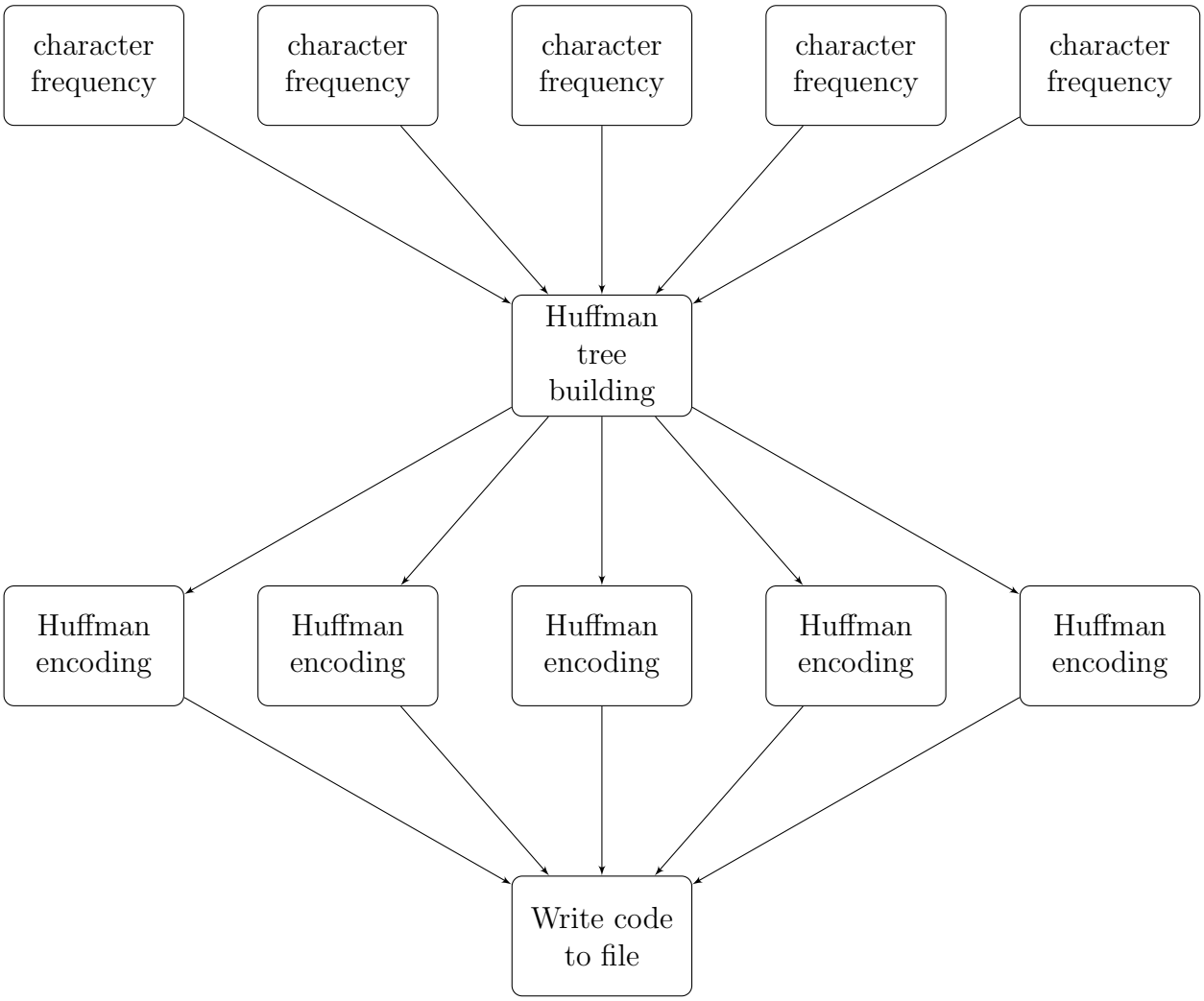
Algorithm 1 Huffman Encoding

```
1:  function ENCODE(s)
2:      f ← compute frequency table for characters in s
3:      pq ← priority queue sorted by frequency
4:      for all c ∈ f do
5:          pq.insert(Node(c, f[c]))
6:      while |pq| > 1 do
7:          n_1 ← pq.pop()
8:          n_2 ← pq.pop()
9:          n ← new node with null character and frequency n_1.freq + n_2.freq
10:         n.left ← n_1
11:         n.right ← n_2
12:         pq.insert(n)
13:     root ← pq.pop()
14:     codes ← compute Huffman codes for each character using root
15:     encoded ← encode s using codes
16:     return encoded
```

Algorithm 2 Huffman Decoding

```
1:  function DECODE(encoded, root)
2:      n ← root
3:      s ← empty string
4:      for all b ∈ encoded do
5:          if b = 0 then
6:              n ← n.left
7:          else
8:              n ← n.right
9:          if n.left = null and n.right = null then
10:             s ← s + n.char
11:             n ← root
12:     return s
```

1.1 Task Dependency Graph



1.2 Opportunities for Parallelism and Concurrency

1.2.1 Identifying Parallelism

- **Data parallelism**: The process of computing the frequency counts for each symbol in the input data can be parallelized across multiple processors by dividing the input data into blocks of equal size, with each processor responsible for processing a block of data. This is a form of data parallelism, where the same operation is performed on different portions of the input data.
- **Functional parallelism**: After the frequency counts have been computed, the construction of the Huffman tree can be parallelized by using multiple processors to build subtrees of the overall tree. This is a form of functional parallelism, where different operations are performed on the same data to achieve a common goal.
- **Pipelining**: Finally, the encoding of the input data can be pipelined, where multiple processors work together to encode different portions of the input data simultaneously. This is a form of pipelining, where different operations are performed on different portions of the input data, and the output of one operation serves as the input to the next operation.

1.2.2 Degree of Concurrency

The degree of concurrency in Huffman compression depends on the number of processors used. Hence we can say:

$$\textbf{Maximum Degree of Concurrency} = n = O(n)$$

1.3 Design in Pthreads

- **Character Frequency** To measure character frequencies, the contents of a file can be segmented into independent chunks for processing. This task decomposition has no interdependencies, and blocked assignment is employed to take advantage of spatial locality. Each thread is assigned to process n/p characters, where n is the size of the input file, and the resulting character frequency vectors are then combined by adding them to a shared variable that is protected by a mutex lock, taking $O(pc)$ time, where c is the number of unique characters in the input file.
- **Encoding Chunks** When encoding file contents, input data can be decomposed and assigned in blocked format. However, these resulting tasks are dependent on each other since the output of each task's encoding depends on the length of the encoded output of the tasks preceding it. To avoid sequential encoding, chunks can be parallelly encoded using separate intermediate buffers for each task. However, concatenating the buffer contents would require bit-shifting for each thread's output, which could be expensive. Hence, this approach is not taken.

  To solve this problem, each thread precomputes the length of its output using the character frequency vectors produced for each chunk in the first step, which can be done in parallel in $O(c)$ time. Each thread then waits on a condition variable until the previous thread computes its offset. Once the previous thread's output offset and length are available, these values can be added to determine the output offset for the current thread. In this way, the prefix sum is computed over output-length to determine the offset at which each thread should

write the encoded output, before actually encoding in parallel. Separate condition variables and mutex locks are used for each thread's entries. The range of elements manipulated by adjacent tasks may overlap at the boundaries; hence, memory accesses to the first and last 32 bits in each task's range are protected using a separate mutex lock.

- **Decoding Chunks** When decoding file contents, dividing the encoded file into chunks is challenging due to the variable length encoding, making it difficult to identify character boundaries. To overcome this challenge, the number of threads and their chunk offsets, which are calculated during encoding, are stored as header information in the encoded file. This information can then be used to decompose the file into independent tasks that produce equal-sized character chunks during decoding. The resulting output can be directly written to memory mapped file buffers.

1.4 Performance Metrics

1.4.1 Time Complexity

To find the time complexity we need to analyze each operation individually.

- `Character Frequency Calculation` - We need to traverse the entire file atleast once to find the frequency of each character. Thus, here, $T_{serial}(n, 1) = O(n)$. In parallel, we can divide this task to p threads working independently and then add a merger overhead of $O(pc)$ to it. Thus, here, $T_{parallel}(n, p) = O(\frac{n}{p} + pc)$

- `Huffman Tree Construction` - Since a greedy approach is used here, this is an inherently serial part of our implementation. Here, since we have c unique characters in a file, thus $T_{serial}(n, 1) = O(c \log c)$. Assuming the file to be ASCII encoded, thus c would be a constant, and hence $O(c \log c) = \theta(1)$. Similarly, $T_{parallel}(n, p) = \theta(1)$

- `Huffman Tree Traversal` - Similar to Huffman tree construction above, this would be dependent on c. Thus here as well, $T_{serial}(n, 1) = T_{parallel}(n, p) = O(c) = \theta(1)$

- `Encoding Header Information` - Since the maximum size of a huffman encoding for a character is fixed (let it be constant l_{max}), the maximum size of header information can be cl_{max}. Thus, here as well $T_{serial}(n, 1) = T_{parallel}(n, p) = O(cl_{max}) = \theta(1)$

- `Encoding Chunks` - There are a total of n words in a file, and each encoding can be of l_{max} size. Thus $T_{serial}(n, 1) = O(nl_{max}) = O(n)$. In case of parallel, this can be distributed over p processors. Also, we add a conditional-wait overhead $O(p)$ for waiting for previous threads for their offset calculation. In addition to this, a pre-computation overhead of $O(c)$ needs to be added for pre-computing the length of thread's own output for which the following thread is waiting. In totality, $T_{parallel}(n, p) = O(\frac{n}{p} l_{max} + p + c)$

- `Decoding Chunks` - Decoding in a serial fashion involves first decoding the header information, then reconstruction of Huffman tree and finally reading the contents bit-by-bit and traversing the Huffman tree. All these operations add up to give $T_{serial}(n, 1) = O(cl_{max} + c + nl_{max}) = \theta(n)$. In parallel, these tasks can be divided onto p processors, Thus $T_{parallel}(n, p) = \theta(\frac{n}{p})$.

$$\boxed{\textbf{Serial Runtime} = T_{serial}(n,1) = 3 * O(n) + 3 * \theta(1) = O(n)} \tag{1}$$

$$\boxed{\textbf{Parallel Runtime} = T_{parallel}(n,p) = O(\frac{n}{p} + p) + O(\frac{n}{p} l_{max}) + \theta(\frac{n}{p}) = O(\frac{n}{p} + p)} \tag{2}$$

1.4.2 Speedup

We have just calculated the serial and parallel runtimes above. Therefore

$$\boxed{\textbf{Speedup} = \frac{T_{serial}(n,1)}{T_{parallel}(n,p)} = \frac{n}{\frac{n}{p} + p} = \frac{np}{n + p^2}} \tag{3}$$

1.4.3 Efficiency

$$\boxed{\textbf{Efficiency} = \frac{Speedup}{p} = \frac{n}{n + p^2}} \tag{4}$$

1.4.4 Cost

We know, cost of a parallel algorithm is the product of its time complexity and the no. of processors, p.

$$\boxed{\textbf{Cost} = p * O(\frac{n}{p} + p) \simeq O(n + p^2)} \tag{5}$$

1.4.5 Isoefficiency Metric

T_1 = Time taken by algorithm to execute on single processor = $n = W$
T_p = Time taken by algorithm to execute on p processor = $O(\frac{n}{p} + p)$
T_0 = Total time spent by all processors doing work that is not done by sequential execution
We know

$$p * T_p = T_1 + T_0$$

i.e.

$$\begin{aligned} T_0 &= p * T_p - T_1 \\ &= (n + p^2) - n \\ &= p^2 \end{aligned}$$

Hence, we have $W = Kp^2$ as the isoefficiency function

$$\boxed{\textbf{Isoefficiency Metric} = Kp^2} \tag{6}$$

1.5 Cost Optimality

A **cost optimal** parallel algorithm is an algorithm for which the cost is in the same complexity class as an optimal sequential algorithm. If we assume $p^2 << n$

$$\text{Cost of a sequential Huffman compression algorithm} = O(n)$$

$$\text{Cost of parallel Huffman compression algorithm} = O(n)$$

Hence, **Huffman Compression algorithm is cost-optimal**

1.6 Experimental Results

All experiments were performed on an ®Intel ™Core i7-8550U CPU with 4 cores.

1.6.1 Pthreads

Table 1: Execution time (seconds) for different file sizes

Threads	1MB	2MB	3MB	4MB	5MB
1	0.103297	0.170202	0.282656	0.401438	0.463546
2	0.063969	0.155648	0.155379	0.213634	0.272633
3	0.047666	0.101233	0.129487	0.204572	0.25669
4	0.044786	0.081179	0.117091	0.181664	0.194444
5	0.040669	0.082526	0.103102	0.139666	0.159894
6	0.049809	0.079608	0.121081	0.152392	0.170004
7	0.041795	0.082876	0.124768	0.151514	0.197072
8	0.038372	0.090221	0.121359	0.155116	0.173244
9	0.048096	0.098215	0.107022	0.146688	0.191259
10	0.041552	0.137864	0.168294	0.139579	0.194222

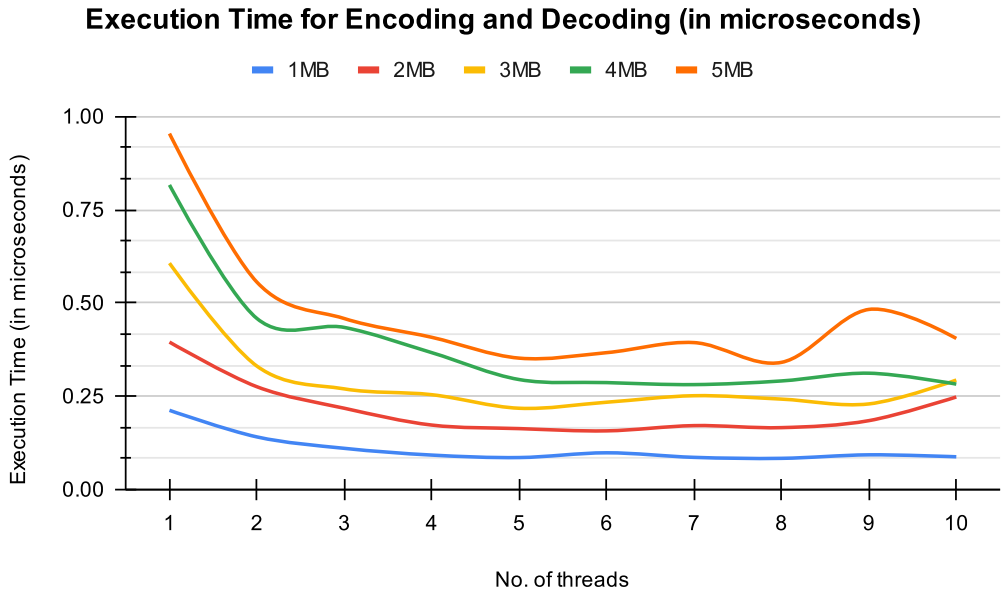


Table 2: Speedup of Huffman Compression for different file sizes

Threads	1MB	2MB	3MB	4MB	5MB
1	1	1	1	1	1
2	1.505373905	1.43468493	1.834241586	1.780638858	1.716323844
3	1.922671544	1.814435876	2.249542209	1.879147307	2.08192626
4	2.29902879	2.287495805	2.387223413	2.225449154	2.338473151
5	2.477001505	2.420818304	2.785331375	2.769851796	2.709143303
6	2.156702623	2.513355418	2.59306514	2.848346374	2.600621929
7	2.462551898	2.307296744	2.413224324	2.903205621	2.42266709
8	2.540208159	2.382646695	2.503963064	2.805183383	2.799209134
9	2.282421988	2.1406415	2.644777498	2.618378649	1.97640889
10	2.4197559	1.59308188	2.067784944	2.889838157	2.356322151

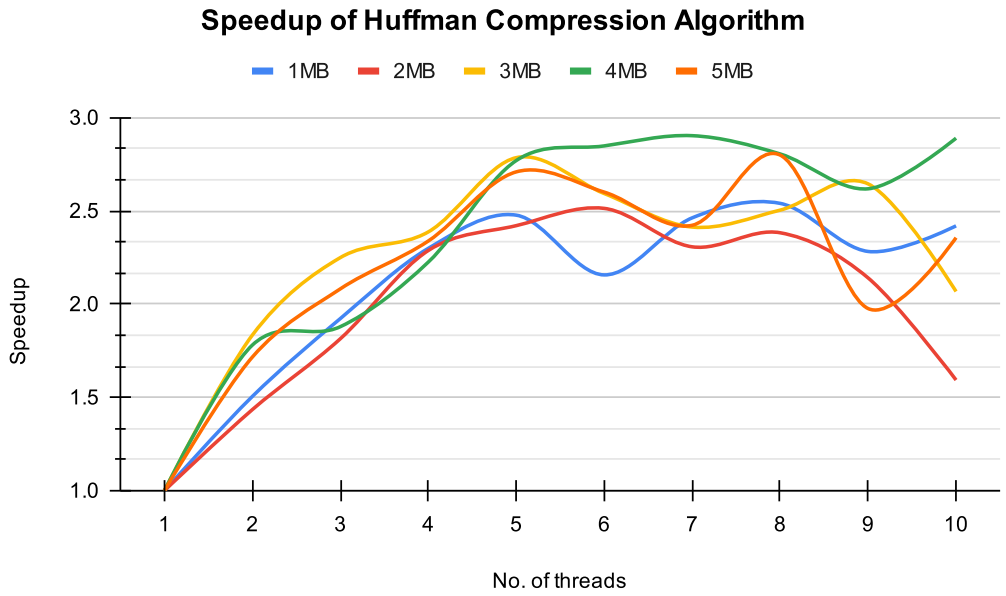
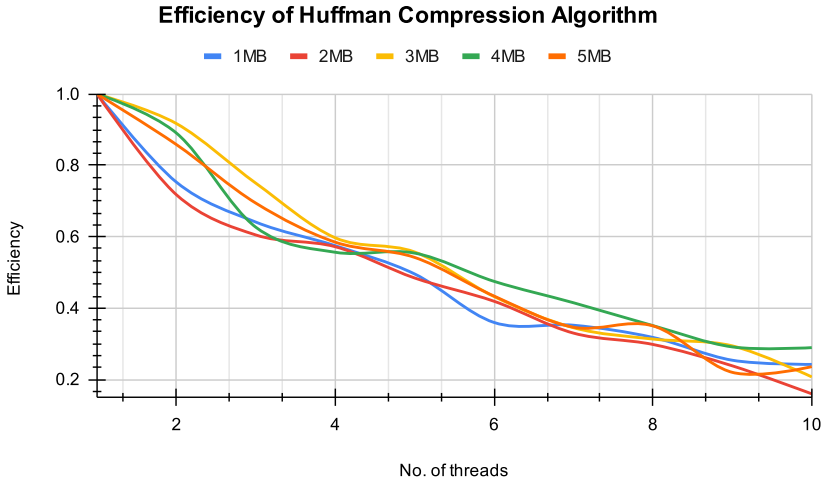


Table 3: Efficiency of Huffman Compression for different file sizes

Threads	1MB	2MB	3MB	4MB	5MB
1	1	1	1	1	1
2	0.752687	0.717342	0.917121	0.890319	0.858162
3	0.640891	0.604812	0.749847	0.626382	0.693975
4	0.574757	0.571874	0.596806	0.556362	0.584618
5	0.495400	0.484164	0.557066	0.553970	0.541829
6	0.359450	0.418893	0.432178	0.474724	0.433437
7	0.351793	0.329614	0.344746	0.414744	0.346095
8	0.317526	0.297831	0.312995	0.350648	0.349901
9	0.253602	0.237849	0.293864	0.290931	0.219601
10	0.241976	0.159308	0.206778	0.288984	0.235632



1.6.2 OpenMP

Table 4: Execution time (seconds) for different file sizes

Threads	1MB	2MB	3MB	4MB	5MB
1	0.471858	0.737402	1.1321	1.446995	1.804224
2	0.24336	0.433357	0.65418	0.825892	1.063042
3	0.204228	0.389486	0.590654	1.080541	0.94501
4	0.334159	0.354794	0.461746	0.620369	1.117996
5	0.36195	0.486296	0.584312	1.003114	0.887163
6	0.571192	0.557393	1.210447	2.415959	0.860662
7	0.341842	2.082772	4.419246	5.482171	4.182622
8	3.386047	5.884788	9.511555	12.585279	15.517958
9	0.274552	0.368591	0.65928	1.629662	1.129485
10	0.245597	0.58475	0.813889	0.748914	1.800193

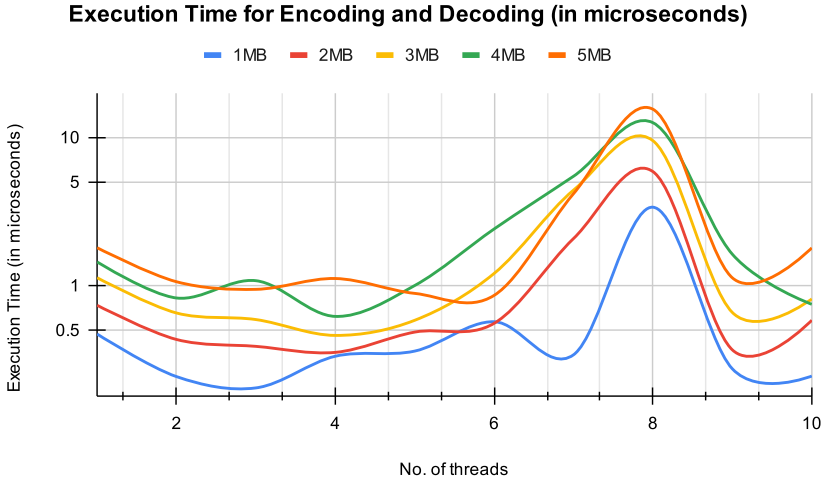


Table 5: Speedup of Huffman Compression for different file sizes

Threads	1MB	2MB	3MB	4MB	5MB
1	1	1	1	1	1
2	1.93892998	1.701603989	1.730563453	1.752039008	1.697227391
3	2.310447147	1.893269591	1.916688958	1.339139376	1.909211543
4	1.412076287	2.078394787	2.451780849	2.332474705	1.613801838
5	1.303655201	1.516364519	1.937492299	1.442503046	2.033700684
6	0.8260935027	1.322948082	0.9352743243	0.5989319355	2.096321204
7	1.380339455	0.3540483548	0.2561749221	0.2639456157	0.4313619543
8	0.1393536475	0.1253064681	0.1190236507	0.1149752024	0.1162668439
9	1.718647105	2.000596868	1.717176314	0.8879111129	1.597386419
10	1.921269397	1.261055152	1.390975919	1.932124383	1.002239204

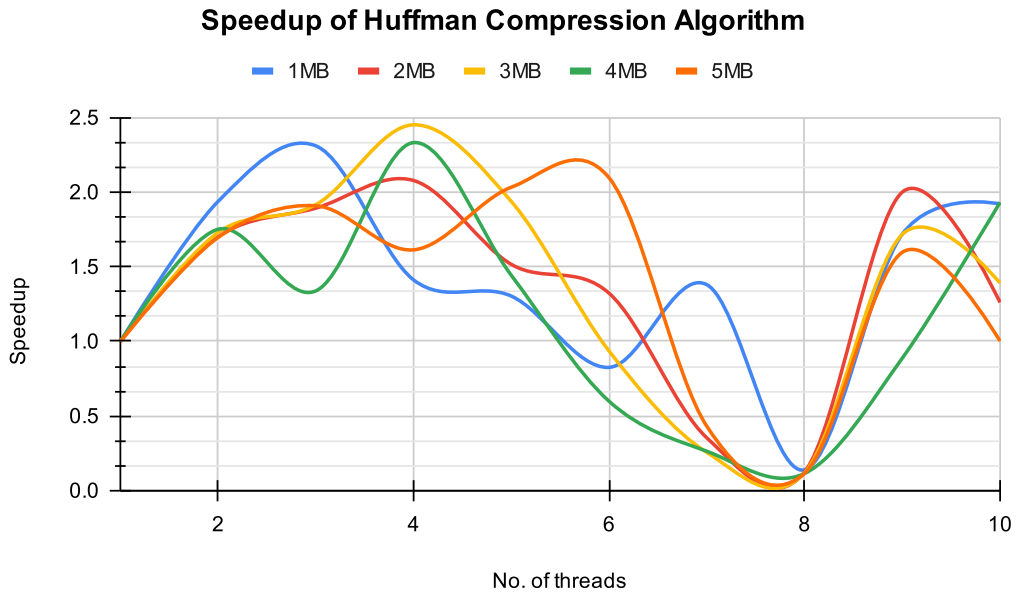
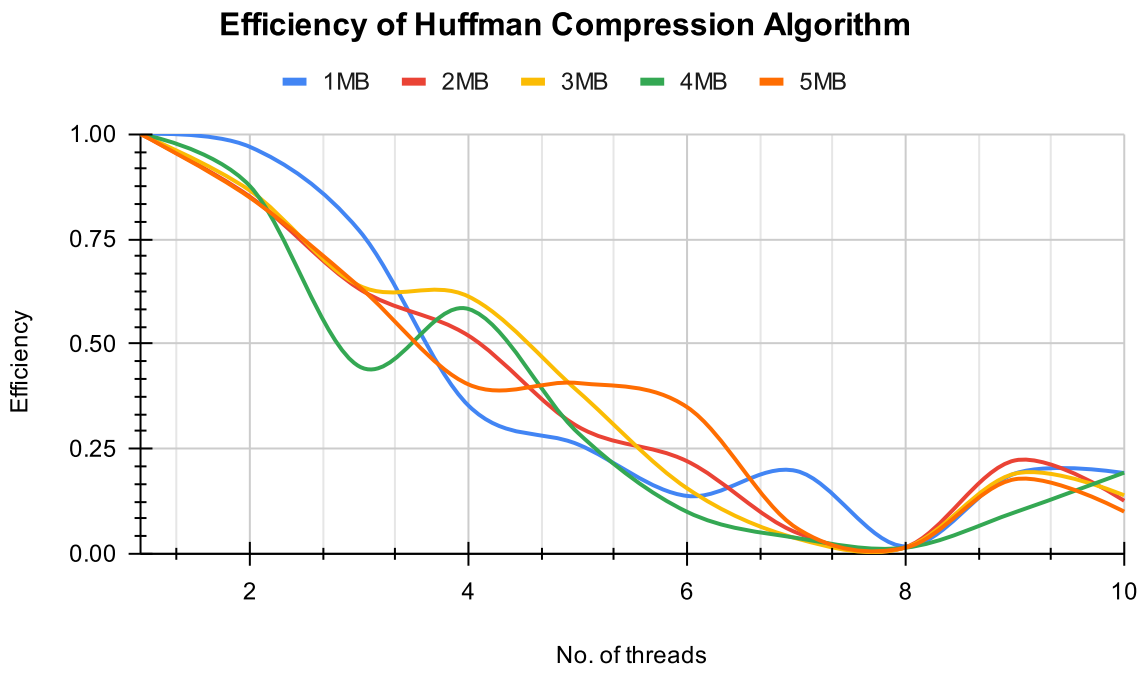
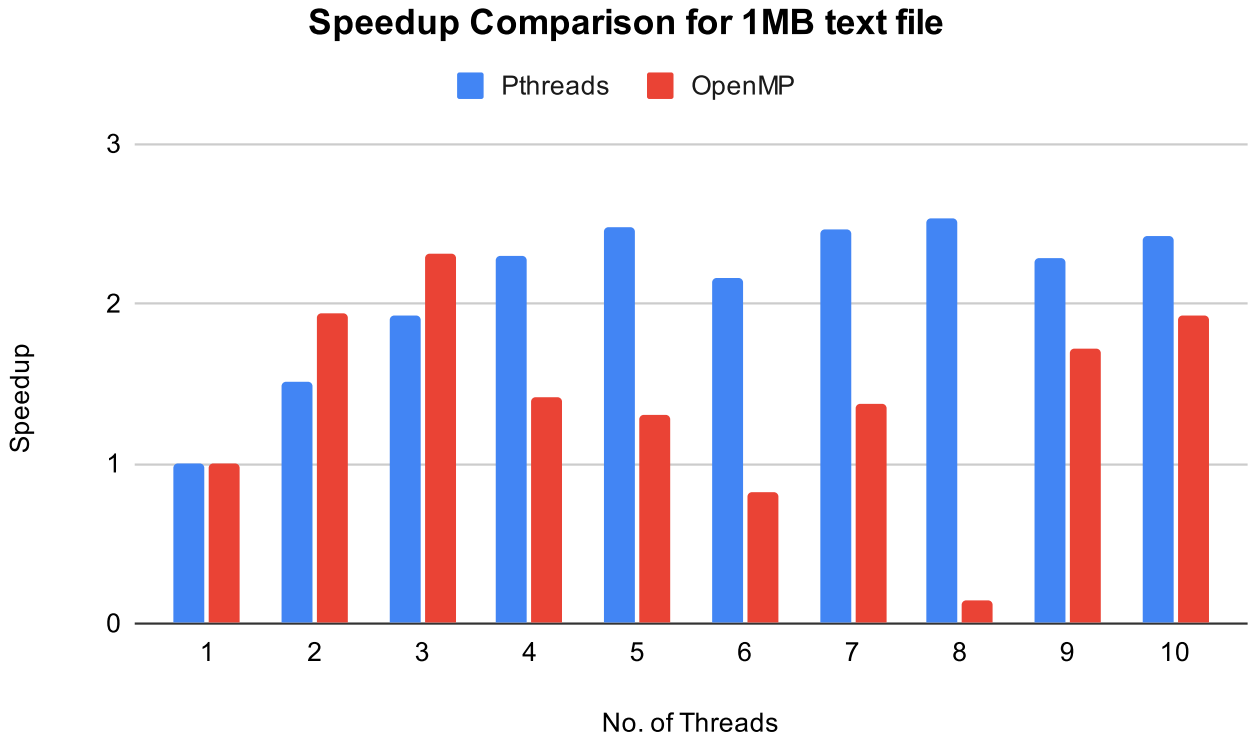


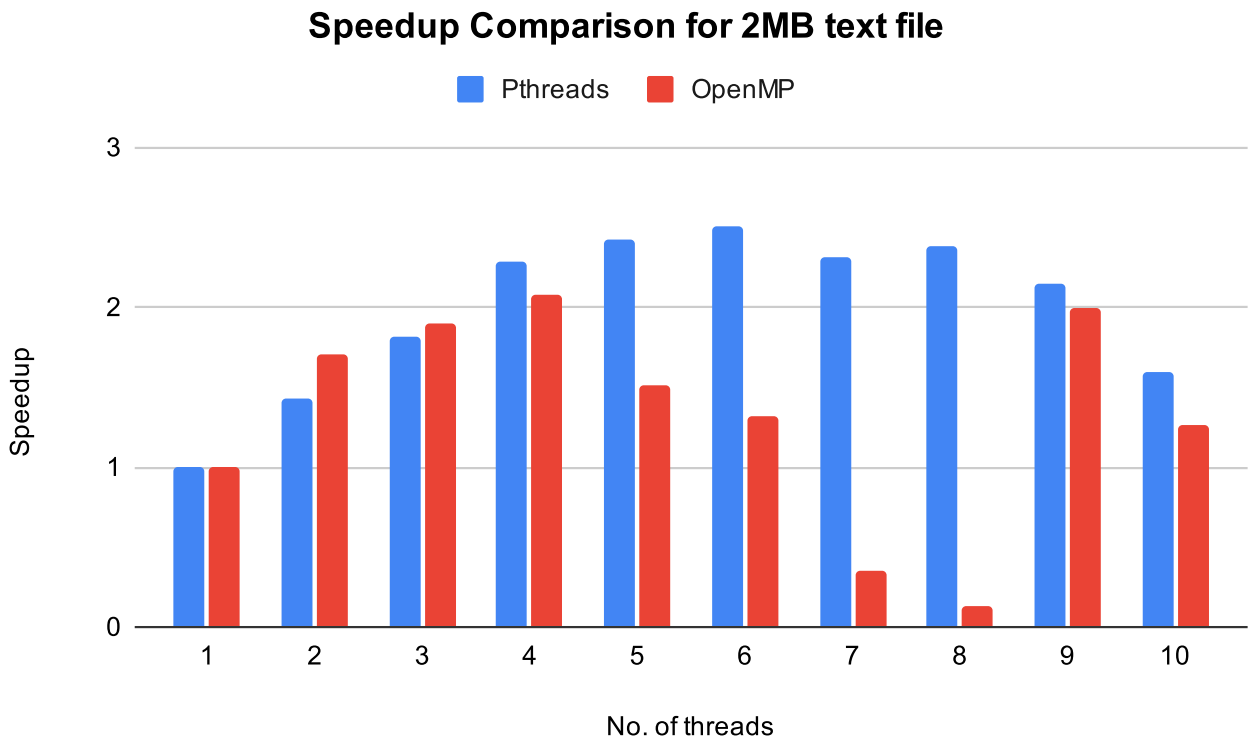
Table 6: Efficiency of Huffman Compression for different file sizes

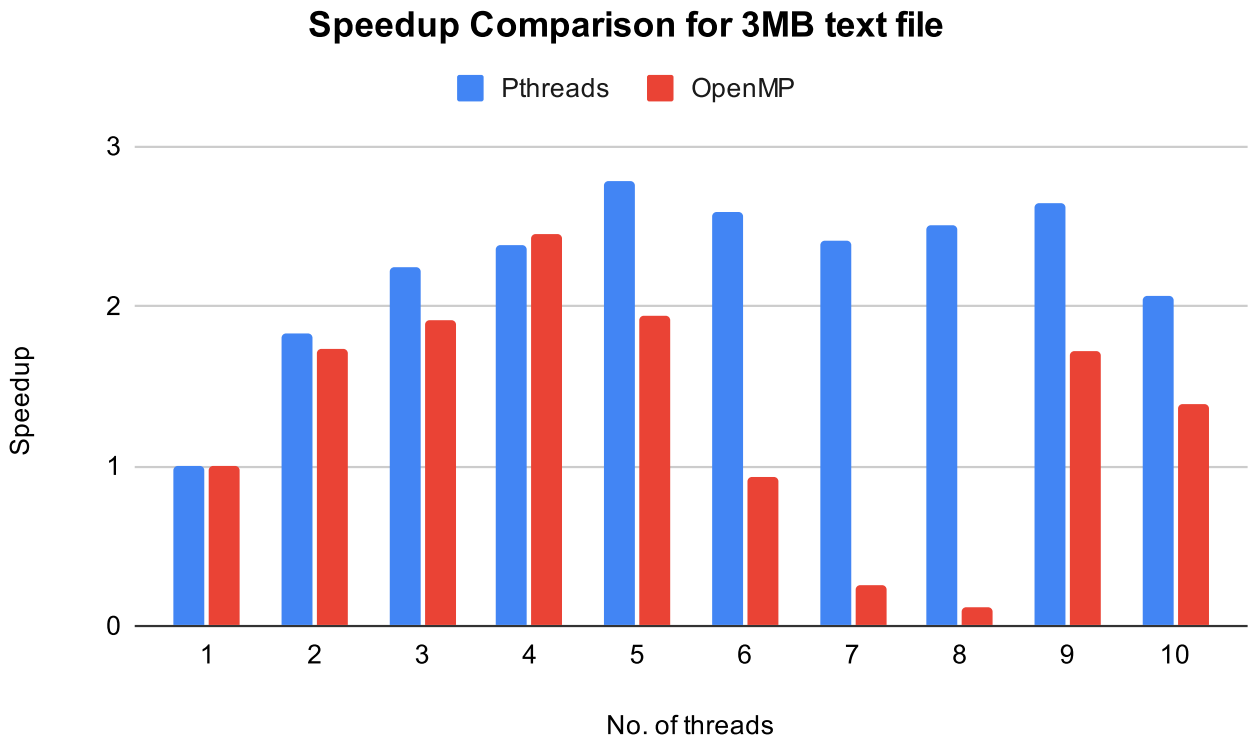
Efficiency	1MB	2MB	3MB	4MB	5MB
1	1	1	1	1	1
2	0.969465	0.850802	0.865282	0.87602	0.848614
3	0.770149	0.63109	0.638896	0.44638	0.636404
4	0.353019	0.519599	0.612945	0.583119	0.40345
5	0.260731	0.303273	0.387498	0.288501	0.40674
6	0.137682	0.220491	0.155879	0.099822	0.349387
7	0.197191	0.050578	0.036596	0.037707	0.061623
8	0.017419	0.015663	0.014878	0.014372	0.014533
9	0.190961	0.222289	0.190797	0.098657	0.177487
10	0.192127	0.126106	0.139098	0.193212	0.100224

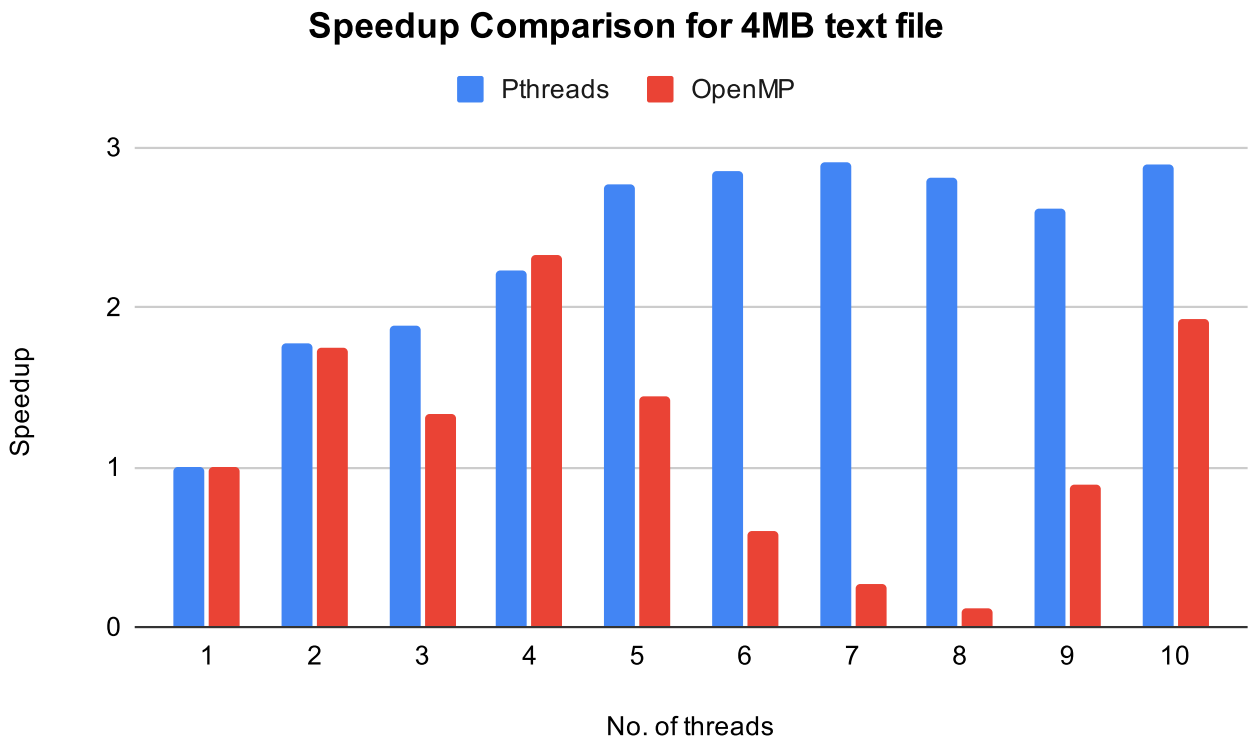


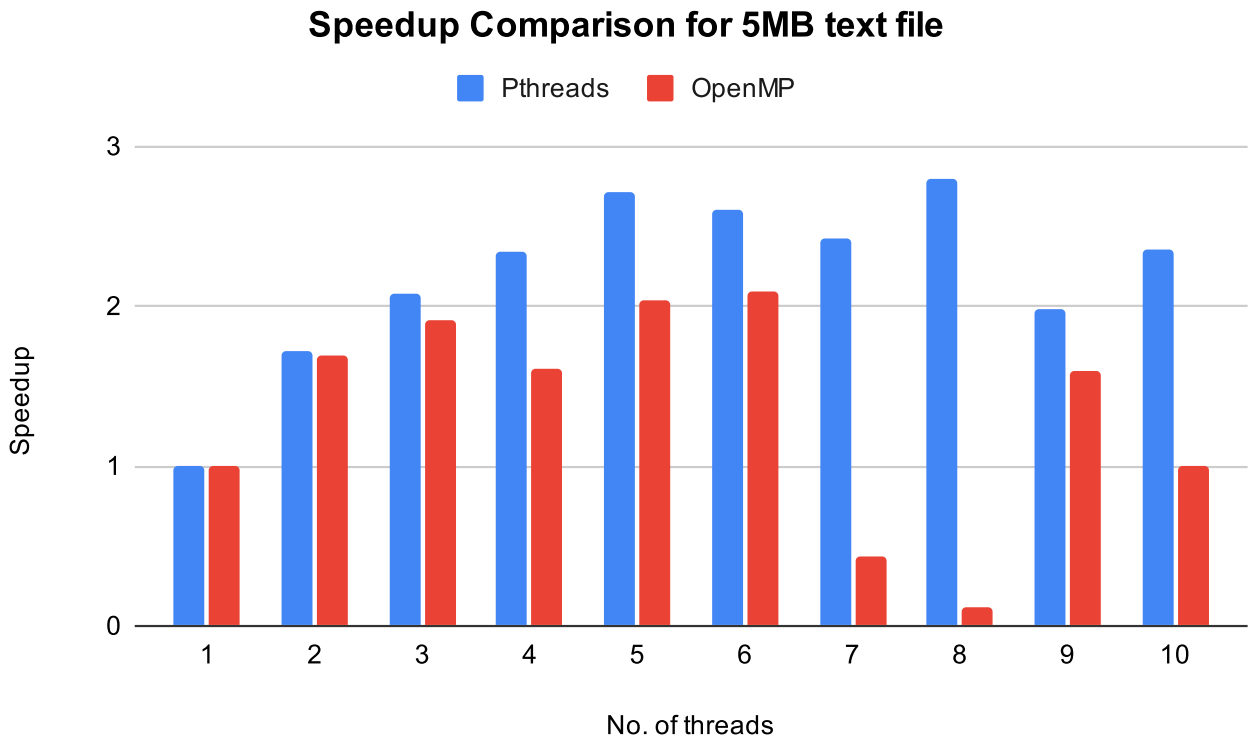
1.7 Speedup Comparison in Pthreads and OpenMP




Speedup Comparison for 2MB text file
Pthreads
OpenMP
3
2
1
0
Speedup
1
2
3
4
5
6
7
8
9
10
No. of threads


Speedup Comparison for 3MB text file
Pthreads
OpenMP
3
2
1
0
Speedup
1
2
3
4
5
6
7
8
9
10
No. of threads


Speedup Comparison for 4MB text file
Pthreads
OpenMP
3
2
1
0
Speedup
1
2
3
4
5
6
7
8
9
10
No. of threads


Speedup Comparison for 5MB text file
Pthreads
OpenMP
3
2
1
0
Speedup
1
2
3
4
5
6
7
8
9
10
No. of threads

References

[1] *Huffman Coding*, available at `hhttps://en.wikipedia.org/wiki/Huffman_coding`.

[2] Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9), 1098-1101.

[3] Sayood, K. (2017). Introduction to Data Compression (5th ed.). Morgan Kaufmann Publishers.