Multi-threaded File Compression

CS F422: Parallel Computing

Assignment 2

Guntaas Singh (2018A7PS0269P)
Harshit Garg (2018A7PS0218P)
K Vignesh (2018A7PS0183P)

1 Introduction

Huffman coding is commonly used for lossless data compression. It assigns variable-length prefix codes to characters depending on their relative frequency. No codeword is a prefix of any other codeword. Any prefix code can be visualized as a binary tree with the encoded characters stored at the leaves. In case of Huffman coding, such a tree is known as a Huffman tree.

The performance for encoding and decoding can be analysed in terms of n - the size of the original text file in bytes, and c - the number of unique characters present in the file. In our case, we assume the file to be ASCII-encoded, hence $c \leq 128$, and can be assumed to be constant. ($c = \Theta(1)$). The length of a codeword (in bits) can lie between 1 and $c - 1$, and shall be considered as constant l_1 for the analysis.

1.1 Encoding

1. Measure frequency of all characters in the given text file.
2. Construct a Huffman tree using the standard greedy algorithm.
3. Traverse the Huffman tree to build a dictionary storing codewords for each character.
4. Encode header information.
5. Encode the contents of the text file by concatenating codewords corresponding to each character, obtained by referring to the dictionary.

The granularity of memory operations for step 5 is a constant x_1. (For our implementation, $x = 32$.) In order to allow decoding of the encoded file, the encoding scheme followed must be written to the output file, typically as header information, preceding the encoded file contents. The time complexity for each of these steps can be computed as follows:

$$T_1 = n \qquad T_2 = O(c \log c) = h_1 \qquad T_3 = \Theta(c) = h_2 \qquad T_4 = cl_1 + 2 \qquad T_5 = n\frac{l_1}{x_1}$$

$$T_{\text{encoding}} = n + n\frac{l_1}{x_1} + h_1 + h_2 + cl_1 + 2 = \Theta(n) \tag{1}$$

1.2 Decoding

1. Decode header information.
2. Reconstruct the Huffman tree using encoding scheme supplied in header.
3. Decode the encoded file by reading the contents bit-by-bit and traversing the Huffman tree.

$$T_6 = cl_1 + 2 \qquad T_7 = cl_1 \qquad T_8 = nl_1$$
$$T_{\text{decoding}} = nl_1 + 2cl_1 + 2 = \Theta(n) \tag{2}$$

2 Design and Implementation

2.1 Implementation of Sequential Algorithms

- A min-heap based priority queue is used for an efficient implementation of the Huffman algorithm.
- File I/O is handled through memory mapping. This allows fast and convenient concatenation of codewords for adjacent characters through bit manipulation, without intermediate buffering.
- The character to codeword encoding map is written as header information in the encoded file as a list of 3-tuples - (char, length of codeword, codeword).
- The number of bytes is also supplied in the encoded file to facilitate decoding. This is necessitated by zero-padding at the end of the file which may be required to round up the file size to bytes.
- The Huffman tree facilitates fast decoding by simple traversal of the tree nodes as the encoded bit stream is read. It is reconstructed using the encoding scheme provided in the encoded file.

The profile information for encoding and decoding a 50MB text file using our implementation of the sequential algorithms described above is given below.

```
% time   self    children   called    name
                                      <spontaneous>
100.0    0.00    0.59                 main
         0.08    0.00       1/1           getCharFreq (step 1)
         0.00    0.00       1/1           huffman (step 2)
         0.00    0.00       1/1           populateCodeTable (step 3)
         0.00    0.00       1/1           encodeHeader (step 4)
         0.08    0.43       1/1           encodeFile (step 4)

                                      <spontaneous>
100.0    0.00    1.64                 main [2]
         0.00    0.00       1/1           decodeHeader (step 5)
         0.00    0.00       1/1           reconstructHuffmanTree (step 6)
         0.08    1.56       1/1           decodeFile (step 7)
```

It is evident that the bulk of the execution time is spent in measuring the frequency of characters, encoding file contents, and decoding the encoded file. Hence, we focus on parallelizing these components.

2.2 Design and Implementation of Parallel Algorithms

Measuring Character Frequencies The file contents can be divided into chunks which can be processed independently. The resulting task decomposition has no dependencies. Blocked assignment of equal-sized chunks is used to support potential benefits from spatial locality. Each thread processes n/p characters. The character freqency vectors produced by each thread are added to a shared variable protected using a mutex lock, which takes $O(pc)$ time.

$$T_{1,\text{parallel}} = \frac{n}{p} + pc$$

Encoding File Contents Here too, input data decomposition and blocked assignment are natural choices. However, the resulting tasks are dependent on each other, since concatenation of the output for each task depends on the length of encoded output for tasks corresponding to all preceding chunks. To avoid sequential encoding, the chunks can be parallely encoded by using separate intermediate buffers for each task. However, concatenation of buffer contents would then require bit-shifting for each thread's output which may be expensive. Hence, we do not take this approach.

To alleviate this issue, each thread precomputes the length of its output using character frequency vectors produced for each chunk in step 1. This can be done in $O(c)$ time in parallel. 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 can be added to obtain 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.

$$T_{4,\text{parallel}} = cl_1 + p + 3$$

$$T_{\text{output-length}} = 2c + p \qquad T_{\text{prefix sum, output-offset}} = p - 1 \qquad T_{\text{encoding batch}} = \frac{n}{p}\frac{l_1}{x_1}$$

$$T_{5,\text{parallel}} = \frac{n}{p}\frac{l_1}{x_1} + 2p + 2c - 1$$

Decoding File Contents Dividing the encoded file into chunks is difficult. Because of variable length encoding, character boundaries cannot be identified easily. To tackle this, the number of threads and their chunk offsets (calculated during encoding) are stored as header information in the encoded file. These can be used to directly decompose the file into independent tasks producing equal-sized character chunks on decoding. The output can be directly written to memory mapped file buffers.

$$T_{6,\text{parallel}} = cl_1 + p + 3 \qquad T_{8,\text{parallel}} = \frac{n}{p} l_1$$

3 Analysis

3.1 Time Complexity

$$\begin{aligned} \text{Serial Runtime} = T_S &= n + nl_1 + n\frac{l_1}{x_1} + h_1 + h_2 + 3cl_1 + 4 \\ &= \Theta(n) \end{aligned} \tag{3}$$

$$\begin{aligned} \text{Parallel Runtime} = T_P &= \frac{n}{p} + \frac{n}{p} l_1 + \frac{n}{p}\frac{l_1}{x_1} + pc + 4p + h_1 + h_2 + 3cl_1 + 2c + 5 \\ &= \Theta(\frac{n}{p} + p) \end{aligned} \tag{4}$$

3.2 Other Performance Measures

$$\text{Speedup} = S = \frac{T_S}{T_P} = \frac{\Theta(n)}{\Theta(\frac{n}{p} + p)} = \Theta(\frac{np}{n + p^2}) \tag{5}$$

$$\text{Efficiency} = \frac{S}{p} = \Theta(\frac{n}{n + p^2}) \tag{6}$$

$$\begin{aligned} \text{Cost} = pT_P &= n + nl_1 + n\frac{l_1}{x_1} + cp^2 + 4p^2 + h_1 p + h_2 p + 3cl_1 p + 2cp + 5p \\ &= \Theta(n + p^2) \end{aligned} \tag{7}$$

If $p^2 = \Theta(n)$ (or equivalently, $n = \Theta(p^2)$), then the cost is asymptotically similar to the serial work. Hence, the parallel system is cost-optimal.

Iso-efficiency Function

$$\begin{aligned} \text{Overhead Function} = T_o &= pT_P - T_S \\ &= cp^2 + 4p^2 + h_1 p + h_2 p + 3cl_1 p + 2cp + 5p - h_1 - h_2 - 3cl_1 - 4 \\ &= k_2 p^2 + k_1 p + k_0 \end{aligned}$$

Considering the largest term, we find that the iso-efficiency function is $O(p^2)$.

Maximum number of processors The maximum number of processing elements that can be used to solve this problem cost-optimally is given by the inverse of the iso-efficiency function.

$$W = n = Kp^2$$
$$p^2 = \frac{n}{K}$$
$$p = \sqrt{\frac{n}{K}} = O(\sqrt{n})$$

3.3 Experimental Results

The following plots and tables present various performance measures for the parallel implementation. It should be noted that the implementation was tested on a machine with an Intel Core i5-8300H - a 4 core multi-processor with support for 8 hardware threads.

p	n = 10 MB	n = 20 MB	n = 50 MB	n = 100 MB	n = 200 MB
1	0.604851	1.572956	3.784581	5.844283	11.644833
2	0.324389	0.586685	1.536980	3.248568	5.937453
3	0.204484	0.448086	1.105888	2.150877	4.353180
4	0.157139	0.345677	0.841089	1.758818	3.370228
5	0.183750	0.369049	0.910336	1.827227	3.520905
6	0.156678	0.316319	0.854290	1.585781	3.252701
7	0.136578	0.278466	0.736190	1.529311	2.994754
8	0.133510	0.264917	0.706996	1.430529	2.821581

Table 1. Execution Time for encoding and decoding a file of size n for different values of p.

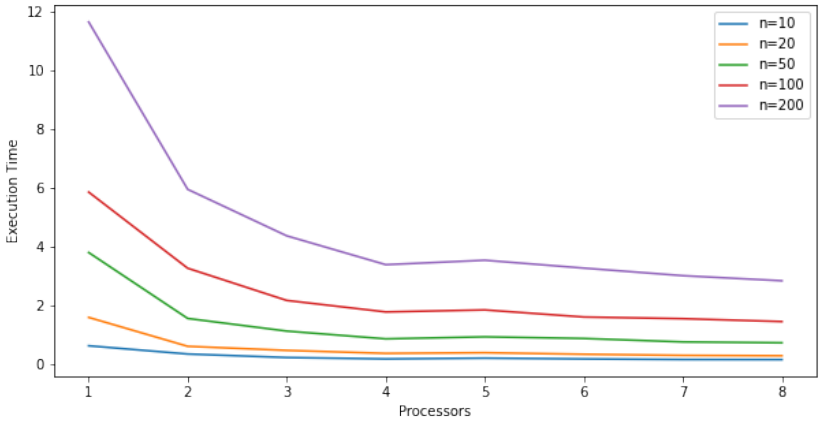

Fig. 1. Execution Time plotted against number of processors for different input sizes.

p	n = 10 MB	n = 20 MB	n = 50 MB	n = 100 MB	n = 200 MB
1	1.000000	1.000000	1.000000	1.000000	1.000000
2	1.864585	2.681091	2.462349	1.799034	1.961251
3	2.957938	3.510389	3.422210	2.717163	2.675018
4	3.849146	4.550363	4.499620	3.322847	3.455206
5	3.291706	4.262187	4.157345	3.198444	3.307341
6	3.860472	4.972689	4.430089	3.685429	3.580050
7	4.428612	5.648647	5.140767	3.821514	3.888411
8	4.530380	5.937543	5.353044	4.085400	4.127060

Table 2. Speedup for encoding and decoding a file of size n for different values of p.

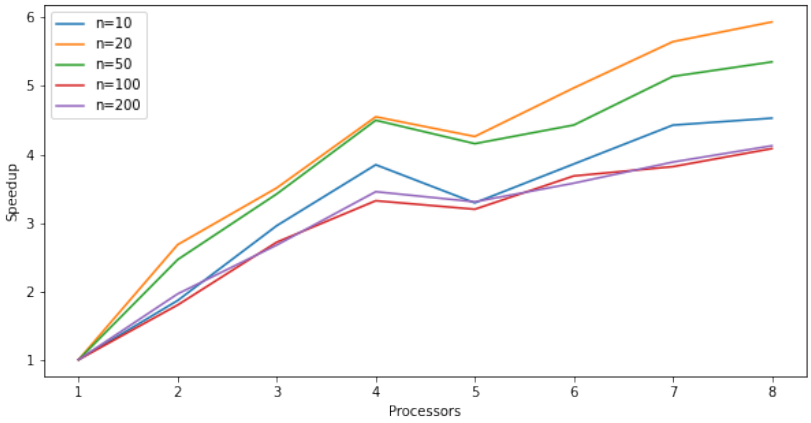

Fig. 2. Speedup plotted against number of processors for different input sizes.

p	n = 10 MB	n = 20 MB	n = 50 MB	n = 100 MB	n = 200 MB
1	1.000000	1.000000	1.000000	1.000000	1.000000
2	0.932293	1.340546	1.231174	0.899517	0.980625
3	0.985979	1.170130	1.140737	0.905721	0.891673
4	0.962287	1.137591	1.124905	0.830712	0.863802
5	0.658341	0.852437	0.831469	0.639689	0.661468
6	0.643412	0.828781	0.738348	0.614238	0.596675
7	0.632659	0.806950	0.734395	0.545931	0.555487
8	0.566297	0.742193	0.669131	0.510675	0.515882

Table 3. Efficiency for encoding and decoding a file of size n for different values of p.

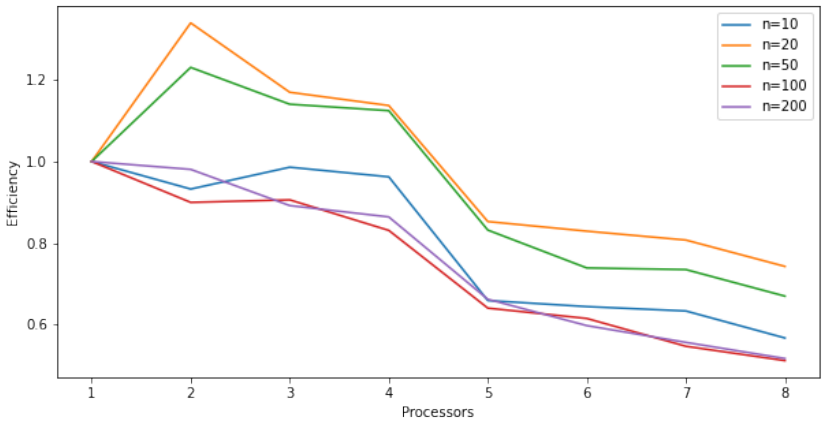

Fig. 3. Efficiency plotted against number of processors for different input sizes.

p	n = 10 MB	n = 20 MB	n = 50 MB	n = 100 MB	n = 200 MB
1	0.604851	1.572956	3.784581	5.844283	11.644833
2	0.648778	1.173370	3.073960	6.497136	11.874906
3	0.613452	1.344258	3.317664	6.452631	13.059540
4	0.628556	1.382708	3.364356	7.035272	13.480912
5	0.918750	1.845245	4.551680	9.136135	17.604525
6	0.940068	1.897914	5.125740	9.514686	19.516206
7	0.956046	1.949262	5.153330	10.705177	20.963278
8	1.068080	2.119336	5.655968	11.444232	22.572648

Table 4. Cost for encoding and decoding a file of size n for different values of p.

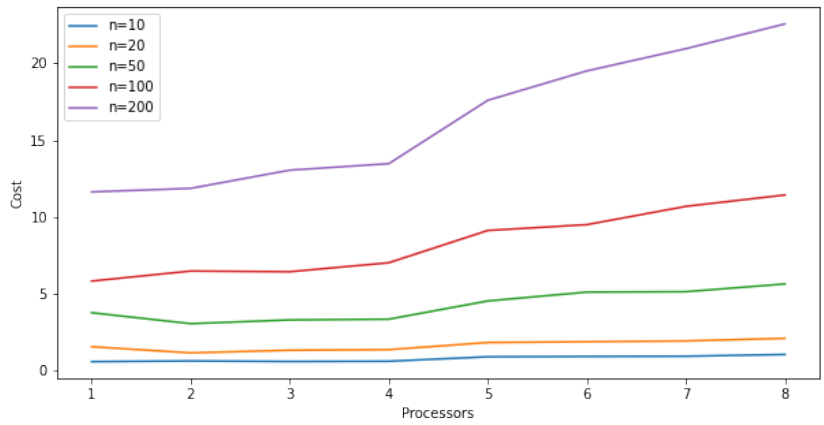

Fig. 4. Cost plotted against number of processors for different input sizes.

3.4 Serial Fraction

p	n = 10 MB	n = 20 MB	n = 50 MB	n = 100 MB	n = 200 MB
1	1.000000	1.000000	1.000000	1.000000	1.000000
2	0.072624	0.000000	0.000000	0.111708	0.019758
3	0.007110	0.000000	0.000000	0.052046	0.060744
4	0.013064	0.000000	0.000000	0.067929	0.052558
5	0.129742	0.043277	0.050673	0.140815	0.127947
6	0.110843	0.041318	0.070875	0.125607	0.135191
7	0.096772	0.039872	0.060277	0.138622	0.133370
8	0.109408	0.049623	0.070640	0.136885	0.134061

Table 5. Estimated Serial Fraction for different values of n and p.

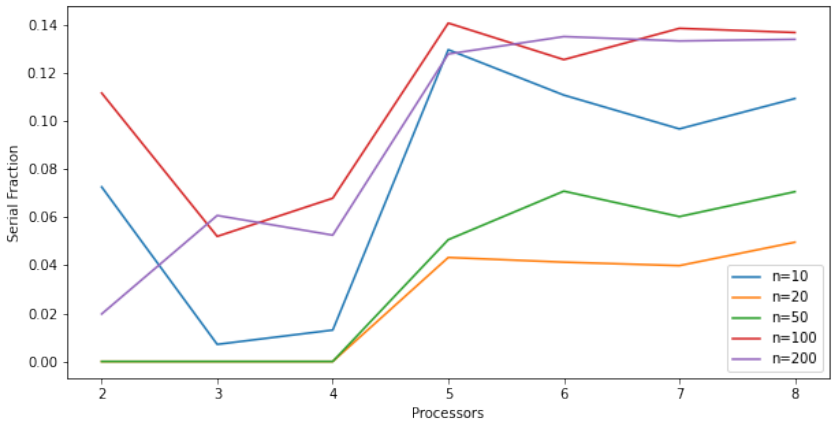

Fig. 5. Estimated Serial Fraction plotted against number of processors for different input sizes.