Exploring Parallel Techniques For Huffman Coding And Data Compression

1st Prayush Dave
DA-IICT
Gandhinagar, India
201801021@daiict.ac.in

2nd Karan Bhatia
DA-IICT
Gandhinagar, India
201801417@daiict.ac.in

3rd Prakhar Maheshwari
DA-IICT
Gandhinagar, India
201801432@daiict.ac.in

4th Aakash Panchal
DA-IICT
Gandhinagar, India
201801459@daiict.ac.in

5th Tushar Patil
DA-IICT
Gandhinagar, India
201801440@daiict.ac.in

***Abstract*—Our work basically revolves around understanding the parallel techniques for Huffman coding and compression. We explore the serial and parallel techniques for Huffman coding and try looking at the possible bottlenecks that affect the performance of the Huffman algorithm. We discuss the necessity for parallelization and also look for novel optimization techniques during the course of this paper.**

I. Introduction

While transmitting data we need to compress it to reduce storage hardware, data transmission time and maximize bandwidth. Huffman coding is an algorithm that uses variable size code words of the minimum average length.
Some of the applications of Huffman algorithm are :

- Cryptography and Data Compression. Applied in algorithms like DEFLATE (used in PKZIP), JPEG, and MP3
- Used in Brotli Compression by Google
- Segmenting, encoding, and decoding DNA sequences based on n-gram statistical language model to handle the vast volumes of DNA sequencing data.

The serial Huffman algorithm has a time complexity $O(NlogN)$. There have been evidences in the paper by J.Leeuwen [4] that the tree building could be approximated in a linear time complexity. The simple Huffman tree has an inefficient memory access pattern which significantly decreases the performance of the algorithm.

After analyzing run time of different parts of code, we got to know that the maximum percentage of time is spent on compression of the input using the hash table of encoded symbol of characters. We can parallelize this to reduce the time. As we can see, first step i.e., building the frequency table or a histogram, is taking second highest time and hence we need to parallelize it as well. Most of the previous efforts evolve around parallelizing these two steps only as these take the maximum time. We would also try to parallelize the third step of traversing the tree in parallel.

A. Related Works

Previous works in performance optimization has been in light in recent years. Parallelizing either of the encoding and decoding techniques are proposed by many researchers through the years. We will briefly look at some of these advancements. Teng [7] was the pioneer in implementing parallel computing to Huffman coding in $O(logn)$ using n^6 processors. Atallah et al. [8] optimized these to some extent, still required high number of processors to be practical. Larmore and Przytycka [9] worked on $O(\sqrt{n}logn)$ time complexity using n processors. A paper by Hashemian [10] shows a speedup in searching for the symbol and reduce both the memory occupied through tree clustering. Some recent works have implemented upon these works and worked on GPU based decoding to optimize the compression techniques. André Weißenberger and Bertil Schmidt presented a fully parallel, GPU based decoderbased on self-synchronizing property, and presented a practical implementation using CUDA based on that property.The decoder works with inputs encoded using Huffman's original method, and, is therefore suited to decode established formats like MP3 and JPEG files. [11]–[13]

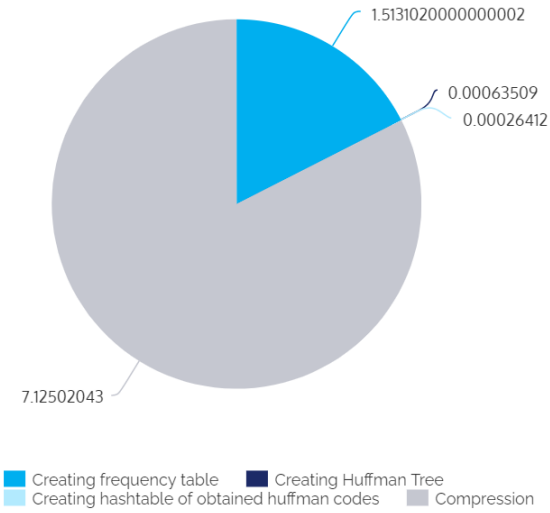


Fig. 1. Time(in seconds) taken by different sections of code on 16MB data set "Bonsai"

II. Serial Implementation of general Huffman Coding algorithm

The general Huffman coding algorithm works on the principle of minimizing the average length of the code by mapping a message to a shorter representation. It takes into account

the variable length code-word instead of using fixed length code-words. The key idea is to map higher frequency symbols with shorter code-word and lower frequency symbols with longer code-words. The total number of bits used to represent a message is hopefully reduced. The characters in the text are stored as fixed 8-bit ASCII code which is not the most efficient way to store and represent data. The general Huffman coding algorithm involves the following steps:

1) Constructing a frequency table sorted in descending order.
2) Building a binary tree. Repeating the steps until completion of the binary tree.
   - Merge the last two items (which have the minimum frequencies) of the frequency table to form a new combined item with a sum frequency of the two.
   - Insert the combined item and update the frequency table.
3) Deriving Huffman tree:
   Starting at the root, trace down to every leaf; mark '0' for a left branch and '1' for a right branch.
4) Generating Huffman code:
   Collecting the 0s and 1s for each path from the root to a leaf and assigning a 0-1 code-word for each symbol.

Any compression algorithm is incomplete without the decompression algorithm. The decompression algorithm is as follows:

- We read the coded message bit by bit. Starting from the root, we follow the bit value to traverse one edge down the the tree.
- If the current bit is 0 we move to the left child, otherwise, to the right child.
- We repeat this process until we reach a leaf. If we reach a leaf, we will decode one character and re-start the traversal from the root.
- Repeat this read-move procedure until the end of the message.

A. Analysis of Huffman Coding

The algorithm of Huffman coding requires to find two minimum frequency items at each iteration hence this can be implemented using Min-Heap. The root of the Min-Heap is the minimum frequency item and to extract it requires $O(1)$ time. Also, to build the Min-Heap again requires $O(logN)$ time. This operation of building and extracting the minimum item is done $2N-1$ times and hence the time complexity of building the Huffman tree is $O(NlogN)$. The first step of sorting the frequency table can be done in $O(NlogN)$ time. Traversing the tree for finding the code for a symbol can be done in $O(height\ of\ tree)$ i.e $O(logN)$ time. So, in encoding the total time complexity is $O(NlogN)$. Hence, the total time complexity of the Huffman Coding algorithm is $O(NlogN)$. Suppose there are K nodes in the Huffman tree then the space to store the Huffman tree is $O(K)$ and the space to store the coded text is $O(N)$.

Data Set	Total Runtime	Compression Ratio
Bonsai	11.572 sec	3.27
Aneurism	12.752 sec	7.51

Fig. 2. Total Run Time(in sec) and compression ratio for two different data sets of same size. Here Total Run-time includes: All four step as in previous figure and Decompression as well.

III. Dataset and Run-times for different datasets

We have obtained two data sets from Open Scientific Visualization. These include two raw images of size 16 Mb each (Aneurism and Bonsai). We compressed these two images using our algorithm and then decompressed it to its original size.

IV. Challenges in parallelizing the algorithm:

We look at different steps of the serial Huffman Coding algorithm and thus try to identify the potential bottlenecks and challenges in parallelizing the Huffman coding algorithm.

1) Creating frequency table: We can parallelize this step by dividing the file into blocks virtually among threads. Now, we have two strategy to choose from:
   1) Create separate frequency table for each thread and combine it with a table of master thread in the end.
   2) Only one frequency table and use critical or atomic or similar instruction inside the loop and use them to avoid race condition.
2) Huffman Tree Construction: As in the standard Huffman compression algorithm, the tree construction algorithm is dependent on the previous steps. Thus, it is not possible to construct the tree in parallel using the standard algorithm.

   One strategy to do so (which definitely fails) is dividing the same frequency symbols among different processors and then creating the Huffman tree in parallel and then use critical section before adding the combined set to the frequency table and then repeating the steps. To understand it, we provide an example: Suppose *"BILL BEATS BEN."* (without quotes) is to be compressed. We create the frequency table and sort the table. "." and "N" can be combined by one processor, "S" and "T" by another, and so on. After all processors complete, the combined symbols are added to the frequency table and method is again applied. But, this fails when we do not have different symbols with same frequency, for example, *"HHHHWWWTTI"*. This string if wanted to compress using the above parallel method, cannot be done since there is a flow dependency and the method fails. Constructing trees in parallel is a challenge.
3) Creating hash table of encoded symbols: To create it, we need to traverse the tree, which can be done similar

to using ”task” directive as on page 50. But as we can see in the first graph, this step does not consume much of the time in the whole algorithm, we need to see whether parallel overhead increases the time of parallel implementation than serial implementation.

4) Compression step: We can see that this step consumes most of the time in compression step. Also, this part is not doing any heavy computations and hence much of the time is spent on memory handling. Therefore, Huffman algorithm is **memory bound**.
   We can parallelize this step by creating virtual blocks and dividing the work among threads, but as Huffman coding generates variable length codes(in encoding step) for each character, it may not be the case that the length of the output data is divisible by 8 (1 byte = 8 bit and 1 byte is the smallest memory data that can be read and write independently). Hence, careful code need to be designed to combine the output of different threads to avoid any discrepancy in the data.
5) Decompression step: Similar to compression, as Huffman coding generates variable length codes(in encoding step) for each character, decompression is also tricky.
   We need to determine indices of such bits in the compressed data where if we are doing it serially, then while traversing the tree(in decompression algorithm) we are starting at the root again on that bit. This helps to divide the work of decompression among threads, as we are starting from the root it can be handled independently by each thread as if doing it serially.
   But, we need to care about not dividing the work unequally among threads. As of now, we are able to see that we need to use some concept similar to prefix sum and we can divide the work by finding such length of data blocks efficiently using a binary search.
   Note that number of data blocks should preferable be number of threads in the core.

V. Parallelization approach:

We have parallelized three main parts of the serial Huffman Coding algorithm namely: creating the frequency table, the compression step, and the decompression step. Other two remaining steps are not taking very less time(less than 0.1ms) as we can see in the graphs.

We have two approaches to parallelization of huffman algorithm. First version is discussed below. Other implementation is discussed in the latest draft.

Version 1

In the first version of our algorithm, the main approach is to divide the file into the number of cores parts, i.e., if the number of cores is 2, then file will be divided into two parts: first half and 2nd half. These parts will be processed by each individual cores and then we combine the results in the end.

The details and results of the 2nd version is discussed in the latest report.

A. Frequency table construction

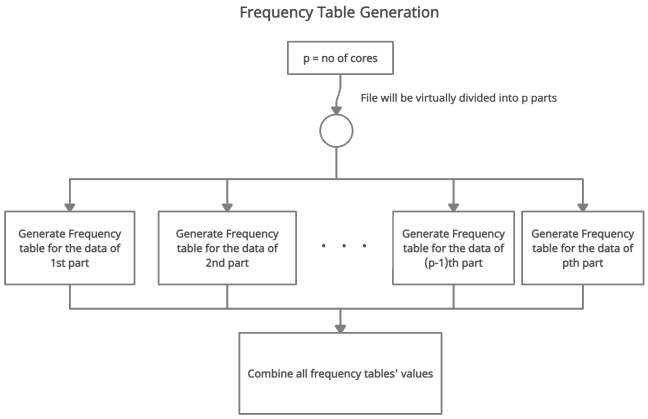


Fig. 3. Schematic for Generating Frequency Table in Parallel

Here, combine all the frequency tables involves below procedure:

Algorithm 1: Combine all Frequency tables step

```
for all p parts
#pragma omp critical
  for i in range(0,255)
    final_frequency_table[i] +=
    freqency_table_of_current_part[i]
```

Similar idea is used in the compression and decompression as well.

B. Parallel Compression

Algorithm 2: Pseudocode of Parallel Compression step

```
buffer contains the whole file
size_of_parts = file_size / no_of_processors
#pragma omp parallel {
    thread_id = get_thread_num()
    _offset_ = thread_id * size_of_parts
    total_bits = 0

    Apply the standard huffman compression algorithm
     on the data of size size_of_parts, which is
    starting from the index _offset_ in the buffer.

    total_bits stores total number of bits in the
    compressed file which is helpful in the
    decompression step.
}
Combine all these compressed parts.
```

Similar algorithm is used in the decompression step as well.

C. Parallel Decompression

Algorithm 3: Pseudocode of Parallel Compression step

```
buffer will contain the decompressed file
size_of_parts = file_size / no_of_processors
#pragma omp parallel {
    thread_id = get_thread_num()
    _offset_ = thread_id * size_of_parts

    read (thread_id)th compressed part
    Apply the standard Huffman decompression
     algorithm on this part.

    Write the decompressed part in the buffer at the
      offset of _offset_.
}
Finally, the buffer contains the decompressed file
     which is equal to the original file.
```

VI. Hardware Details and Results from Parallel Implemenatation

Details	Configuration
CPU(s):	16
On-line CPU(s) list:	0-15
Thread(s) per core:	1
Core(s) per socket:	8
Socket(s):	2
NUMA node(s):	2
CPU family:	6
Model:	63
Stepping:	2
CPU MHz:	1528.820
BogoMIPS:	5206.19
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	20480K
NUMA node0 CPU(s):	0-7
NUMA node1 CPU(s):	8-15
RAM :	4 GB per CPU

Looking at the performance of the parallel implementation on the basis of the stages as explained earlier:

VII. Results

We have used below datasets for the analysis.

Dataset name	Size
Anuerism	16 MB
Bonsai	16 MB
Pancreas	120 MB
Head Anuerism	256 MB
Magnetic Reconnection Simulation	512 MB

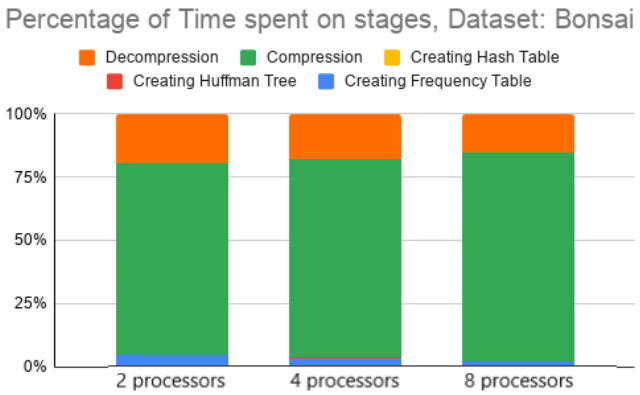


Fig. 4.

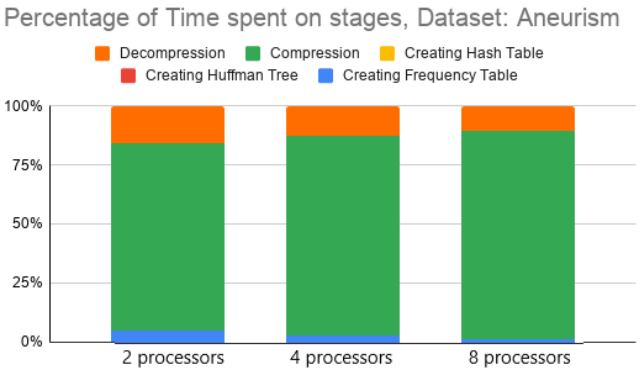


Fig. 5.

As observed in both Fig .4 and .5, the maximum time being taken is in the compression stage and then decompression stage. The reason behind these results according to our observations of timing segments of codes is that the memory access time for these two stages are quite high, due to the number of read-write operations. And the reason behind, why compression stage takes maximum time, is the computational time of the encoding algorithmic implementation. So, one of our upcoming challenges is to workout how to optimize these computations in more efficient manner.

Speedup:

	Bonsai	Aneurism
2 processors	1.082180317	1.019997919
4 processors	1.323756227	1.040216855
8 processors	1.364976602	0.9143334513

Fig. 6.

Fig:XII shows the speedup analysis, where as observed the values are not quite following the linear trend, due to the above mentioned issue in parallel implementation.

Here is the detailed run times of our both data sets arranged by the different sections of codes.

VIII. PERFORMANCE ANALYSIS:

Let's analyse the results with different parallel performance analysis laws and formulae.

A. Analysis using Amdahl's Law:

Amdahl's law defines the maximum theoretical speedup which can be achieved by a parallel algorithm. Let f be the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. The maximum speedup achievable with p cores for this computation is

$$\frac{1}{f + (1-f)/p}$$

. When we consider this law to predict the theoretical maximum speedup of our algorithm we obtain the maximum speedup of 1.9 for 2 cores, 3.4 for 4 cores, 5.9 for 8 cores for the Bonsai database and 1.9 for 2 cores, 3.9 for 4 cores, 7.9 for 8 cores for Aneurism database.These are the maximum theoretical speedup but it does not take parallel overhead into account. The maximum theoretical speedup are calculated for the 3/5th of the Huffman code since only 3 parts out of 5 are parallelized hence the expected maximum theoretical speedup would not be exactly 'p' where 'p' is the number of cores. [14]

B. Analysis using Gustafson-Barsis' Law:

Amdahl's Law states maximum speedup for a problem of a fixed size but it is not effective for larger problems as it under estimates the speed up and thus producing $Amdahl's\ effect$ and making it difficult to analyse for larger problems. Gustafson-Barsis' law calculates the scaled speedup. If p is the number of cores and s is the fraction of the program's total execution time spent executing inherently sequential code then the maximum speedup achieved is

$$p + (1-p) * s$$

. Considering Gustafson-Barsis's Law for theoretical speedup, we get a speed up of 1.9 for 2 cores, 3.9 for 4 cores, 7.9 for 8 cores for the Bonsai database and 1.9 for 2 cores, 3.9 for 4 cores, 7.9 for 8 cores for the Aneurism database. This algorithm also does not include the overhead because of the parallel overhead. [14]

C. Analysis using the Karp-Flatt Metric:

The Karp-Flatt Metric provides a means to use empirical data to learn something about the overhead as well as the amount of inherently sequential computation in a parallel program. Also, the other two laws do not take the overhead into account. When we consider The Karp-Flatt Metric the experimentally determined serial fraction f_e of a parallel computation which is given by

$$f_e = \frac{1/s(p) - 1/p}{1 - 1/p}$$

where s(p) is the experimental speedup of p cores, is 0.848121 for 2 cores, 0.673901 for 4 cores, 0.694415 for 8 cores, and 0.8912 for 16 cores for the Bonsai dataset. As one could observe that the value of f_e is increasing with an increase in the cores which suggests that the overhead is not negligible and cannot be overlooked. Also, it suggests a possibility of identification and optimization of the overheads. The experimentally determined serial fraction is high which suggests that the serial computations are prominent in the parallel implementation. [14]

D. Complexity analysis of parallel implementation

Parallel Frequency table construction:
Considering the full size of the file as N we carry out the complexity analysis. In the frequency table construction, we observe that the read file and temporary frequency table creation for each thread is done in parallel. Suppose there are p processors. Hence, the total file is divided among p processors equally and hence the loop is run in parallel in time $O(N/p)$. The critical section is run 256 times since the frequency of all characters must be stored in the main frequency table.

Parallel compression step:
The file is divided among p processors. The loop running to read the main file to the temporary files is run N/p times. encode[a] takes up at worst $O(logN)$ time and str.length() will take up a small amount of time since the code lengths won't be very large. The j loop is running at max len times. The shift operator is not a costly operator and doesn't take too much time. Read and write operations are not considered in the calculation of time complexity generally. Thus at max the time complexity of parallel compression step would be $O(NlogN/p)$.

Parallel decompression step:
The while loop is run until the total_bits become 0. There is a nested while loop that runs to traverse the tree and that takes up at max $O(logN)$ time. The outer loop will run for $O(N/p)$ times since the total number of bits would be equal to the bits present in the file read by the processor. Thus, at max the total time complexity of the decompression step would be $O(NlogN/p)$.

E. CMA analysis

CMA is a very important ration and helps to know the average number of floating point operations per slow memory access and one can hope to do at most CMA ratio operations while the data lies in the fast memory (cache). Higher the CMA higher is the memory efficiency of the program. We

consider two types of memory only and will refer the total cache memory as the fast memory and the main memory as the slow memory. Thus, from the hardware used we have a total of 1568 KB or 1.53 MB of fast memory per CPU and 4GB per CPU of slow memory. Each cache line is of 64 KB in size for the L1 cache.

Creation of frequency table :

For the creation of frequency table, a loop runs for N times where N is the file size and the frequency of each character is set by reading from the buffer array. A file of size 1.53 MB can fit completely in the cache and there won't be any slow memory accesses. Now, if a file has a larger size than that of cache there will be slow memory accesses. Total number of slow memory access will be approximately $N - \frac{1568}{4}$ where N is the number of iterations taken for reading each character from the file. CMA for the frequency table creation is equal to $\frac{N}{N-392}$.

Huffman tree construction:

In the creation of Huffman Tree, the frequency table is traversed and each character with non-zero frequency is considered as a node of the Huffman Tree. At max the complete frequency array will be stored in the cache since its size is just 256×4 bytes and thus it will be accessed completely from the cache with no slow memory accesses. Also, the nodes are then stored in a priority queue for the max heap formation and thus at max 256 of those and hence at max $(2 * 256 - 1) \times 4$ bytes will be used. This can also be stored directly in the cache memory. Thus, all the computations of the Huffman tree construction will be done with the data stored in the cache memory. There would be very few cache misses in this.

Obtaining Huffman Codes from Huffman Tree:

The tree traversal will be done for at max 2*256 - 1 nodes and thus since the tree is stored directly in the cache the chances of cache misses is very low. The final codes are stored in a map and the map can take at most $256 \times O(log(2 * 256 - 1)) \times 4$ bytes where $O(log(2 * 256 - 1))$ is the tree height. This is so because the codes at max will be of the height of the tree and thus this gives a rough estimate of the size of the map. The map of encodes can be stored directly into the fast memory. All computations for obtaining the Huffman codes would be done directly with the data present in the fast memory.

Compression of data :

Now, the file has to be read in the buffer again to encode the file with Huffman codes. There are two loops running - one that reads each character from the file into the buffer and gets the Huffman code. Second that writes the encoded bit string to the compressed file. Here total number of computations done are $N + 6 * O(log(2 * 256 - 1)) * N$. The data accesses to the map of encodes is fetched from the fast memory but that too depends. If the complete fast memory is populated with the file buffer then there wouldn't be any space for the map of encodes to be stored and then it will be fetched from the main memory resulting to a large computation cycle penalty. It becomes very difficult to exactly predict the behaviour of the cache access here since both the situations are very likely that the buffer might populate the cache entirely and the map of encodes has to be fetched from the main memory or the map of encodes is brought to the fast memory wiping some of the data of the buffer then buffer data has to be accessed from the main memory.

Decompression of data:

The compressed file is read from the compressed file buffer and here too a similar situation arises as in the case with compression. The size of compressed file could exceed the total cache memory and thus resulting in some slow memory access of the data. The final output is written to another buffer. There will be write misses in the cache since the entire decoded file might not fit in the entire cache memory. The behaviour in terms of reads would be better for the decompression of data since the compressed file size would be smaller and the spatial locality would be exploited more on average in this case.

IX. Optimizations

1) We are using an array(hash-table) to store the obtained codes after constructing the huffman tree, so as to avoid traversing the tree again and again to see the code of a character. After analyzing our code bit by bit, we realized that we were using strings as a data type to store the encodes of each character which are obtained by traversing the code i.e., as we traverse the tree we will concatenate '0'(left-side) or '1'(right-side) to the string of an code.
   Hence, as we know one character uses 1 byte the size of the array of encodes was increasing and cache miss rate was increased. Due to that we weren't able to achieve proper speedup and good run-time as well.
   Now, we decided to work bit by bit. Hence, as we are traversing the tree we will store these '0' or '1', which were taking 1 byte per character, in a bit-string to take 1 bit per '0' or '1'.
   Internally in our code, this bit-string is generated by using fixed sized 32 bit(an integer) string and using (bit)shift operations ($<<$ or $>>$) to activate the bits as we traverse deeper into the Huffman tree and in the end store how much bits in this 32 bit string were actually used to store the (en)code.
2) Furthermore, rather than reading the original file again while we are generating frequency table and compression, we read the file into a buffer at the very start and then we can reuse it again as per the requirements.

Hence, working bit by bit to construct hash-table of codes and using buffers to read and write the file, lead to the optimized version of our algorithm.

After these optimization, we also implemented another version of parallel code(which was discussed in second draft of our report), which was to virtually divide the data into

many chunks and using **# pragma omp parallel for schedule** in the parallel compression and decompression of chunks individually, to divide the work as equally as possible among the processors.

Version 2

In 2nd version of parallel algorithm, the approach is to virtually divide the file into many small chunks and this chunks will be dynamically processed by the cores i.e., as soon as the processing of a chunk is complete the cores will take next available chunk. This approach is inspired by the variable length codes generated from the Huffman encoding algorithm.

The high level details of the steps in the algorithm are following:

As in the frequency table generation, the work is divided equally among the processors it did not needed any changes.

Compression:

Algorithm 4: Parallel Compression step for version-2

```
Let the size of a chunk is C.
Let the size of the file is F.
number of chunks = F/C.
Virtually divide the data into F/C parts.

#pragma omp parallel for scheduling(dynamic, 1)
for all i in range(1, number of chunks)
  Use standard algorithm to compress this chunk

Combine all the chunks.
```

Similar algorithm is used for decompression as well i.e., all chunks will be decompressed individually and then combined to create the decompressed file which will be equal to the original file.

X. Profiling information

We have used profilers to individually track down the bottleneck functions and issues with the approach. We have used *gprof* and *valgrind's callgrind*, *massif* and *cachegrind* tools for tracking down the time spent, memory growth for the complete program, and cache misses on the 16 MB Aneurism dataset. Below are the complete profiling information for the program. We have used normal function names instead of the function names that appear in the profiling output for better understanding purposes.

Profiling with gprof	
% Time	Name
64.74%	main
24.82%	create freq table
19.69%	decompress
19.26%	compress
0.06%	create huffman tree
0.00%	obtain huffman codes

We observe from the output of *callgrind* that the potential bottlenecks of the program are the *compress* and *decompress* functions and this is what we also observe by measuring time using clock functions in the program. A potential reason for

Profiling with callgrind	
% Time	Name
22.70%	compress
11.24%	decompress
5.39%	create freq table
0.05%	create huffman tree
0.00%	obtain huffman codes

Profiling with massif					
n	time(cycles)	total(B)	useful-heap	extra-heap(B)	stacks (B)
0	0	0	0	0	0
1	4,521,283,750	19,266,712	18,944,367	322,345	0
2	7,040,891,222	939,944	846,249	93,695	0
3	10,931,697,765	16,845,040	16,816,317	28,723	0
4	14,100,148,589	1,780,432	1,518,911	261,521	0

Profiling with cachegrind						
Dr	D1mr	DLmr	Dw	D1mw	DLmw	Name
36.50	36.27	41.97	29.01	0.41	0.09	compress
25.61	10.15	0.27	16.42	74.15	83.93	decompress
15.40	36.07	43.14	8.99	0.00	0.00	create freq table

that is observable from the output of *cachegrind*. Here,

Dr : number of memory reads
$D1mr$: D1 cache read misses
$DLmr$: LL (last level) cache data read misses
Dw: number of memory writes
$D1mw$: D1 cache write misses
$DLmw$: LL (last level) cache data write misses.
It is clearly visible from the table that D1 cache misses and Last level cache read misses in the *compress* are very high and that supposedly could lead to lower speedup. Also, there are write misses in the *decompress* function which are also very large. Main memory is accessed large number of times and this includes large computation cycle penalties and this takes a toll on the speed up.

XI. Results of Open MP Implementation

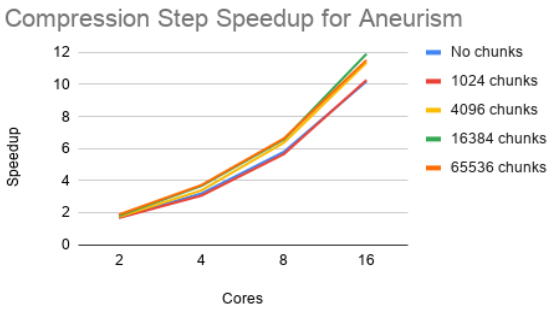
We took different sizes of chunks ranging from 2^8 to 2^{20} and shown the results for some of the chunk sizes.

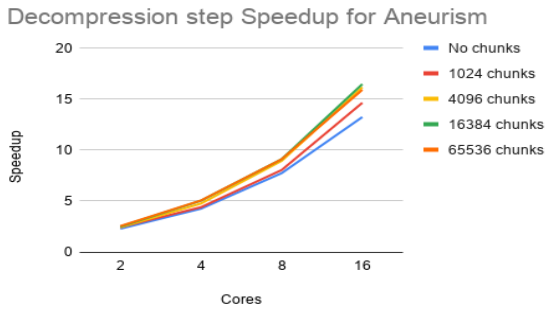
Depending on the data set, the performance of version-2 of our algorithm varies and it sometimes performs worse or equally or better than the previous version. If the amount of data processed by each chunk is almost equal, then this approach of using many smaller chunks will not be effective and overhead due to synchronization and scheduling of these chunks will decrease the speedup.

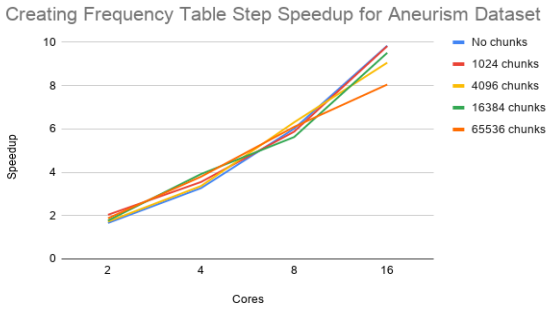
On an average, in general sense, both the versions are showing equivalent results. Hence, for general datasets we can go for version-1 and if we can analyze the general patterns in datasets we are going to work on, then we can choose to go for version-2 and choose an optimal chunk-size.

Note: "No chunks" or "0 chunks" in the graphs here represents the results of version 1 algorithm and "chunks" refer to "chunk-size".

A. Data Set 'Aneurism' 16 MB

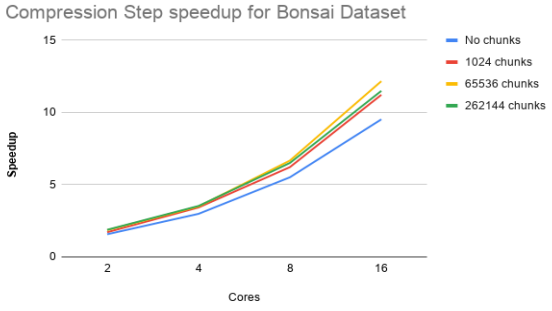


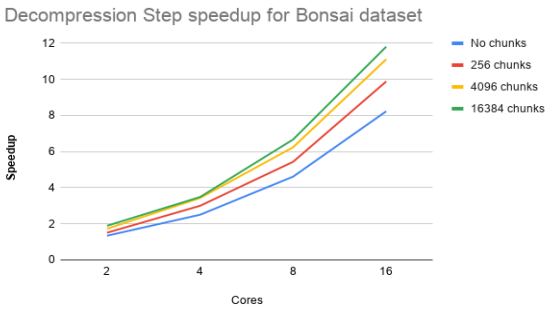


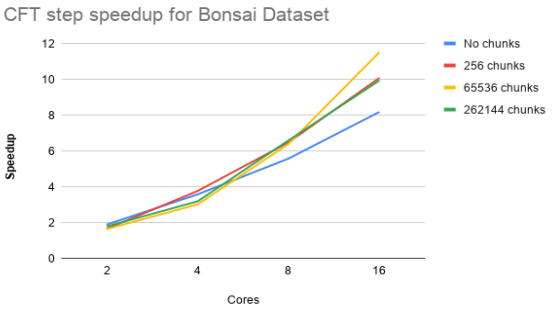


Here we can see that for Aneurism data set as we are increasing the chunk size the speedup for compression step, decompression step and creating frequency table step also increasing.

B. Data Set 'Bonsai' 16 MB

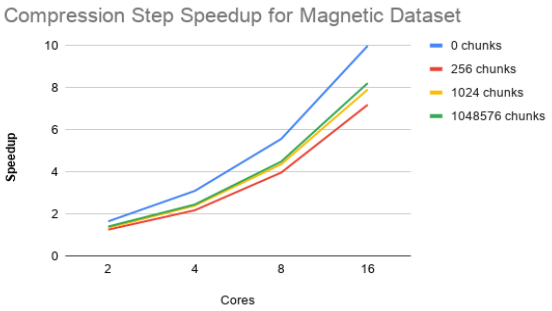


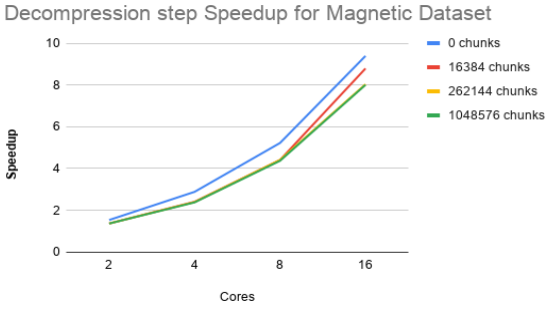


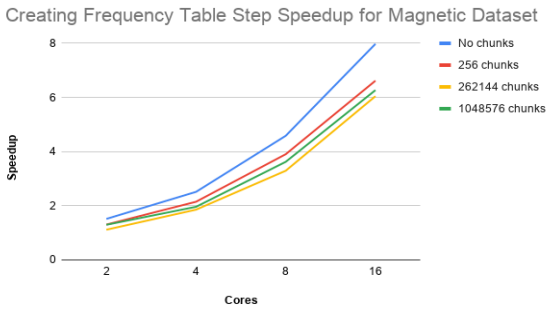


for Bonsai data set with increasing the chunk size the speedup for compression step,decompression step and creating frequency table step increasing as Shown in above graphs.

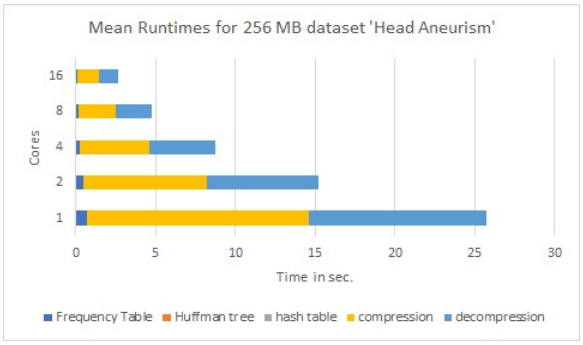
C. Data Set 'Magnetic' 512 MB

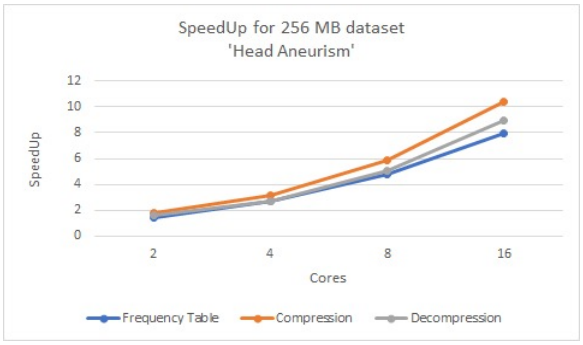




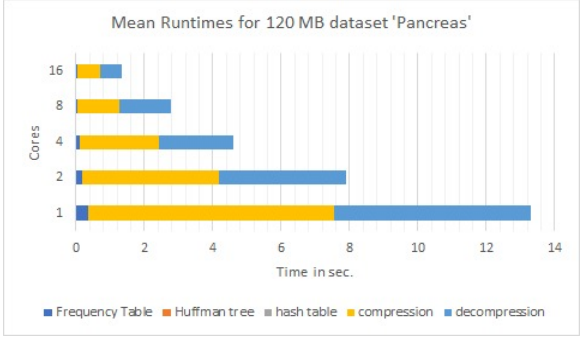


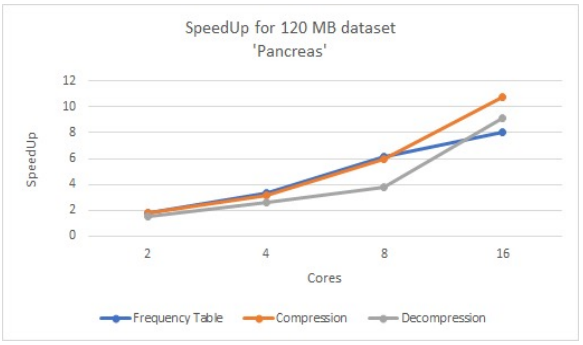
D. Data Set 'Head Aneurism', 256MB





E. Data Set 'Pancreas', 120MB





We have implemented the normal algorithm on all five data sets and the version 2 where we have divided the file into chunks on three data sets, aneurism, bonsai and magnetic. We have plotted the speedups of the the three processes which were taking most time and which we have implemented as parallel algorithm.
Accordingly we can see from the mean time and speedup graphs is that as the number of cores increase speedup increases. In smaller data sets of 16 MB, as we increase the size of file chunks, the speedup increases. However this does not hold true in the compression of 512 MB data set.

XII. MPI IMPLEMENTATION

Algorithm 5: Generate frequency table step

```
p = number of cores
Virtually divide the data into p parts.

for all i in range(2,p)
  MPI_SEND(ith part of data)

Each of the cores will built frequency table of
    their part and send back to master core.

for all i in range(2,p)
  MPI_RECV(frequency table of ith part of data)

combine all frequency tables.
```

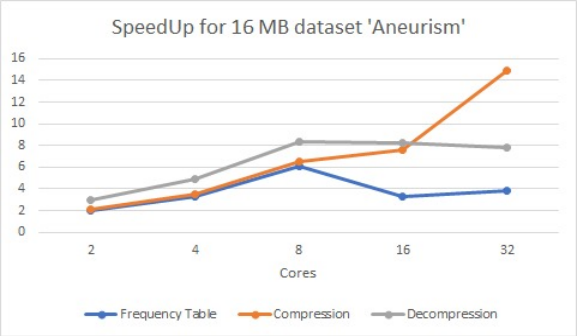
Similar, approach is used for compression and decompression. But note that at the time of compression the size of compressed chunk is variable i.e. not known prior to the step, so we need to send the number of total bits present in the compressed data prior to sending the compressed part; which in turn will also help to allocate the buffer accordingly and in the decompression step as well.

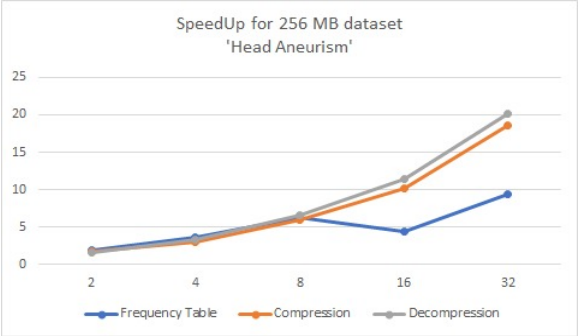
Results and analysis for MPI:

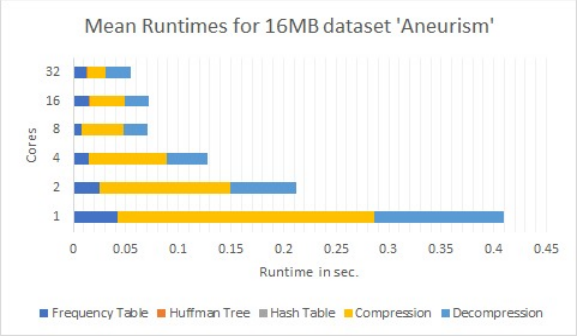
We can observe that speedup is increasing as we increase the number of mpi cores for all three parallelized parts. Moreover, we can see that frequency table having lesser speedups for higher number of cores 16 and 32. The reason is that for relatively smaller sized data (256 * 4 Bytes = 1 KB), the communication overhead is increasing as the number of cores increases; whereas in compression and decompression the

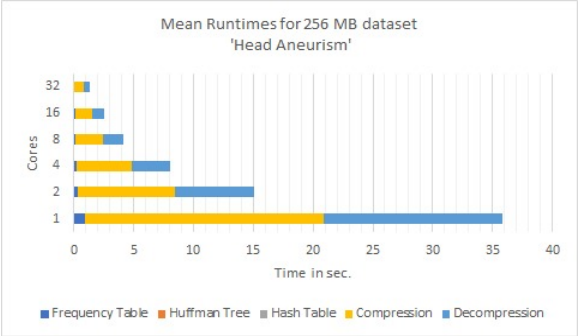
transferred data is larger and hence overhead time will be relatively smaller as compared to the time taken to transfer the parts of the data.

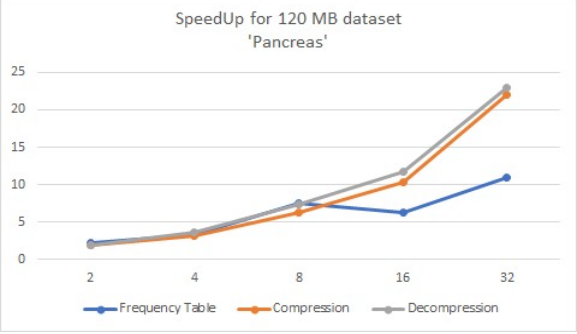
One more observation is that as compared to OpenMP, in MPI we have communication overhead and hence in general the speedup is little smaller as compared to the OpenMP version. Note that in OpenMP we had access to 16 cores but in MPI 32 cores are achieved, and hence the highest achieved speedup is higher in MPI.

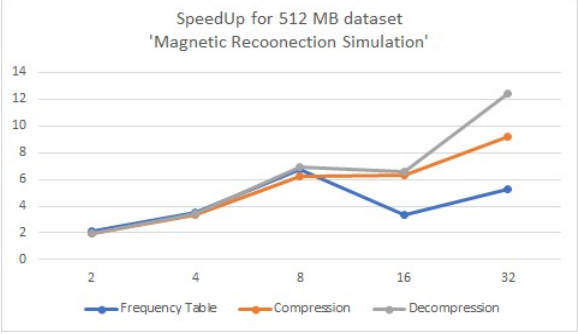


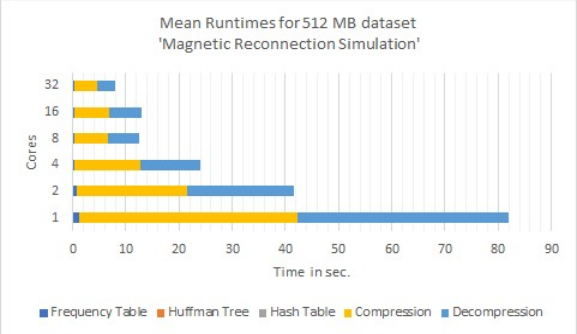












Conditions for all our codes to work:

1) Our code can handle a dataset if big enough storage available for processing and storing compressed and decompressed file sizes and the size is less than 512 MB.
2) The height of huffman tree should be less than or equal 32 or in other words the length of the code obtained by huffman algorithm should be less than or equal to 32. This condition is satisfied in the many of the real world datasets.
3) The file size should be divisible by the no. of cores we are using. We can pad some extra bytes(interestingly extra bytes will not be more than the no. of cores) to achieve this purpose.
4) In the 2nd version, file size should be divisible by the size of chunks as well.

We can certainly extend our code to handle all the above constraints by using some other data types like bitsets, long long, big integers, but the time was limited.

REFERENCES

[1] Scalability of a Parallel JPEG Encoder on Shared Memory Architectures, September 2010, DOI: 10.1109/ICPP.2010.58 SourceDBLP,Conference: 39th International Conference on Parallel Processing, ICPP 2010, San Diego, California, USA, 13-16 September 2010.

[2] Revisiting Huffman Coding: Toward Extreme Performance on Modern GPU Architectures, Jiannan Tian, Cody Rivera, Sheng Di.University of Illinois at Urbana-Champaign, IL, USA, 1 Mar 2021.

[3] University of London *Data compression I. Pu CO0325 2004, University of London 2004, reprinted October 2005*

[4] J.Van Leeuwen, Edinburgh University. July 1976. On the construction of Huffman Trees.

[5] Final Paper: Parallel Huffman Encoding and Move to Front Encoding in Julia, Gil Goldshlager December 2015

[6] Patel, R. A., Zhang, Y., Mak, J., Davidson, A., Owens, J. D. (2012). Parallel lossless data compression on the GPU . IEEE.

[7] Association Computing Machinery, Publication:ACM SIGACT NewsJuly 1987 https://doi.org/10.1145/36068.36071

[8] Publication:SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architecturesMarch 1989 Pages 421–431https://doi.org/10.1145/72935.72980

[9] society of industrial and applied mathematics, SIAM J. Comput., 24(6), 1163–1169. (7 pages) Constructing Huffman Trees in Parallel

[10] Memory efficient and high-speed search Huffman coding Published in: IEEE Transactions on Communications (Volume: 43, Issue: 10, Oct 1995)

[11] Parallel Construction of Huffman code.Islamic Aazad University of Mashhad. Ostad Yousefi St. Gashem abad.2006 Springer.

[12] Parallel Huffman Decoding with Applications to JPEG Files S. T. Klein, Y. Wiseman The Computer Journal, Volume 46, Issue 5, 2003, Pages 487–497, https://doi.org/10.1093/comjnl/46.5.487 Published: 01 January 2003

[13] Massively Parallel Huffman Decoding on GPUs Share on Publication:ICPP 2018: Proceedings of the 47th International Conference on Parallel ProcessingAugust 2018 Article No.: 27 Pages 1–10https://doi.org/10.1145/3225058.3225076

[14] CSci 493.65 Parallel Computing Chapter 6 Performance Analysis, Prof. Stewart Weiss