OpenMP Implementation Adaptive Huffman encoding

1st Darshil Rana
Undergraduate Student
DA-IICT
Gandhinagar, India
201801462@daiict.ac.in

2nd Parthiv Patel
Undergraduate Student
DA-IICT
Gandhinagar, India
201801463@daiict.ac.in

3rd Parth Bhoi
Undergraduate Student
DA-IICT
Gandhinagar, India
201801464@daiict.ac.in

4th Bhargav Patel
Undergraduate Student
DA-IICT
Gandhinagar, India
201801465@daiict.ac.in

***Abstract*—In this paper we have discussed about Adaptive Huffman coding and how it could be parallelized by division of the source symbol set. We have also described the uses and application of adaptive vs. traditional huffman codes, also the time complexity of the serial implementation is discussed here.**

I. Introduction

Data compression is much needed in today's world. With the advent of massive IT industries all over the world, the average data usage rates have increased all over the world. And with the ongoing situation of lockdown the world has seen a rise in usage of technologies like live video and audio sharing and streaming. The transmission of such high quality data may be turn out to be costly if we do not encode or more specifically compress them. There are various techniques and tools which could compress data on a substantial scale, we will be discussing such methods known as Huffman Coding schemes and we will try to parallelize the algorithm of adaptive huffman coding scheme in this paper.

- **Huffman Coding** :
  The basic Huffman coding scheme encodes the message symbols, into bit streams, based on the prior knowledge of frequency of symbol transmission [1]. Each symbol is encoded in such a way that it does not have any common prefix with any other symbol being transmitted, so this gives us the advantage to decode the message at the receiver as soon as the message arrives.
  there are applications of huffman coding scheme in data compression one of the popular is in the fiels of image compression [3]. The main disadvantage of this encoding scheme is that it is static. We need to know the message prior to the message transmission and at what frequency the message will appear.

- **Adaptive Huffman Coding** :
  The main disadvantage of huffman coding scheme is that it is statistical we need to know the statistics of input prior to execute the scheme. But for data transmission such as live video streaming or live voice chatting we will not have such initial information [6]. The adaptive huffman coding scheme is an extended version of the previous algorithm which solves the issues occurred in the original scheme. In this scheme, the frequency of the message symbols are not known at prior to the transmission, we gradually build up the frequency and the encoding tree, corresponding to the symbol's receiving rate. So, adaptive Huffman encoding is dynamic and suitable to adapt to services like live streaming and audio streaming.
  Here, the main focus will be to understand the adaptive huffman coding scheme. Firstly, we will look at its serial algorithm. and then we will try to parallelize it to improve the performance as no such past efforts have been made in parallelization of this algorithm.

II. Implementation

A. Introduction

Huffman coding being static, requires knowledge of the frequencies or probabilities of the source sequence to encode, but when such knowledge is not available, Huffman coding becomes a two-pass procedure, i.e first to collect the statistics of the sequence and then encoding the sequence. Thus, in order to convert the algorithm in one-pass procedure we use *Adaptive Huffman coding* which provides dynamic encoding and decoding at both sender and receiver.

The benefit of one-pass procedure is that the source can be encoded in real time, though it becomes more sensitive to transmission errors, since just a single loss ruins the whole code. [12]

Out of the number of implementation, FKG and Vitter Algorithm provides the best implementation for Adaptive Huffman Coding.

An adaptive huffman tree is a full binary tree where each node has following parameters:

- Node number
- Weight

The node number is unique for each node and follows the property that moving form higher level to lower level the node number decreases, and for the nodes with same level, node number increases from left to right.

And for the weight of the nodes, the weight of each leaf i.e external node is simply the frequency of that symbol, while the weight for each internal node is the sum of the weights of its two child.

Moreover, each node expect the root node, follows the sibling property which is stated as: "All the siblings are arranged in the increasing order of their weights and node numbers".

Also, at any moment for both the transmitter and receiver, the adaptive huffman tree consist of a single node that corresponds to all the symbols that are not yet transmitted and such node is abbreviated as NYT node. Moreover, the weight of NYT node will be zero and also the node number for the NYT node will be smallest amongst all.

As the transmission progresses, the node corresponding to the transmitted symbol will be added to the tree using the *update* procedure. Further, before the beginning of the transmission a fixed code for each symbol is agreed upon between the transmitter and receiver which is For a source with m alphabets/symbols, we need to choose, e and r such that $m = 2^e + r$ and $0 \leq r \leq 2^e$.

B. Serial Implementation

- **Update procedure**

  The update procedure is identical for both transmitter and receiver. The main work for the update procedure is to check and correct all the properties of the Huffman tree i.e preserving the node number and the sibling property.

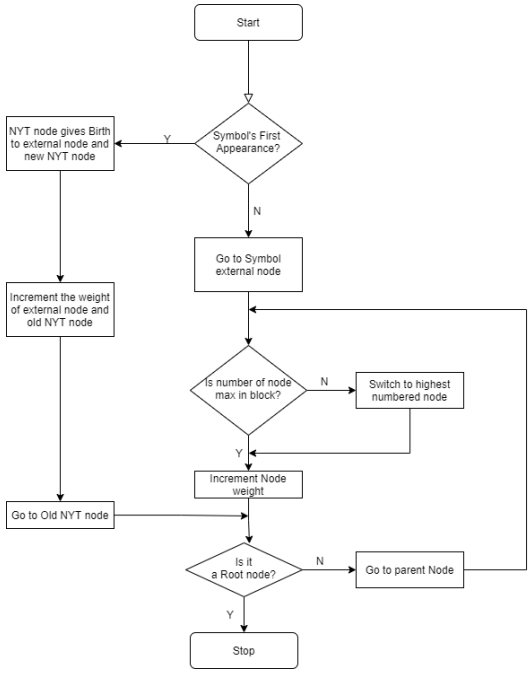

Fig. 1. Update procedure for Adaptive Huffman Coding

Fig. 1 shows the flowchart for the update procedure. Whenever a symbol is encoded or decoded, update procedure is called to update the tree and to preserve the properties of the tree.

Now, for new symbol a new NYT node and a new external node are added as children to the old NYT node and after that, sibling property and node number property is checked traversing up to the root. While for pre-existing symbol, the weight for the particular external node is incremented and similarly for the parent while traversing upwards.

Now for the symbol at k^{th} position the fixed code for the symbol is decided as:

if $(1 \leq k \leq 2r)$ **then**
  $(e+1)$-bit representation of $k-1$
else
  e-bit representation of $k-r-1$
end if

- **Encoding-Decoding Procedure**

We know that both the transmitter and the receiver starts with a single NYT node, and as the symbols are received the tree is updated using the update procedure. Now, the procedure to encode the symbols to the bits and vice-versa are known as encoding and decoding procedures. Thus, the general algorithms for both encoding and decoding procedure can be given as:

- **Encoder**

```
Initialize_tree();
while (message!=end) do
  encode(c,output);
  update_model(c);
end while
```

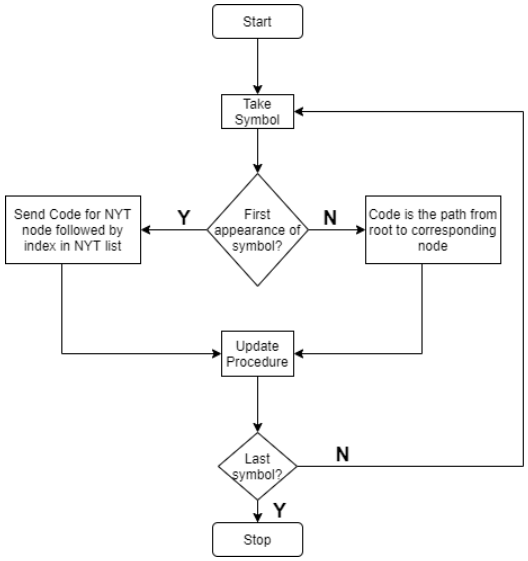

Fig. 2. Encode procedure for Adaptive Huffman Coding

Fig. 2 shows the flow diagram for encoding procedure and as seen, whenever a new symbol is encountered, it is encoded with the NYT-code of the present tree along with the fixed code and then the new symbol will be added to tree using the update procedure, in the same way if pre-existing symbol is encountered, then the huffman code (i.e the code

from root-leaf) corresponding to the symbol is transferred and update procedure is called. Thus, finally our sequence will be encoded and transferred to the receiver.

– **Decoder**

```
Initialize_tree();
while (encoded_message!=end) do
    decode();
    update_model(c);
end while
```

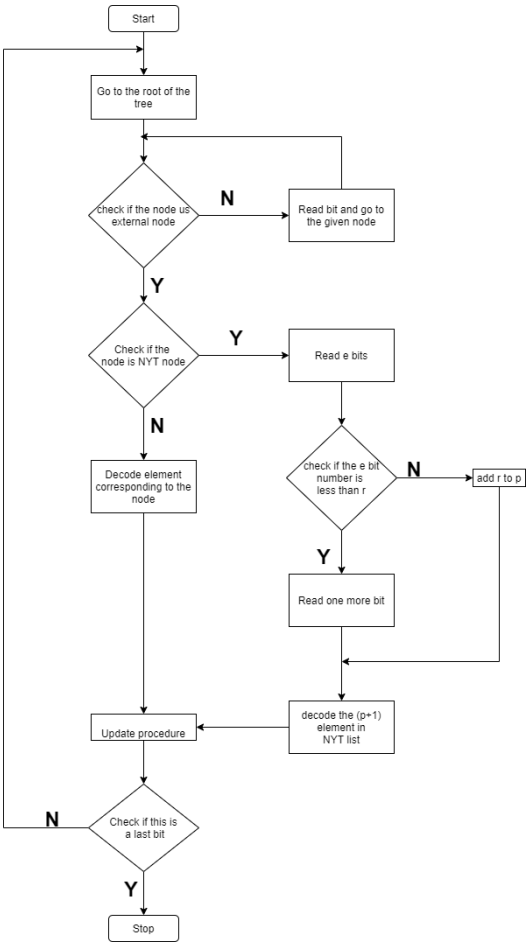

Fig. 3. Decode procedure for Adaptive Huffman Coding

Now for the decoding procedure, as shown in Fig. 3, first we start to traverse the received binary string similar to that done in encoding, and whenever a leaf is encountered the corresponding symbol is decoded. Now, if the leaf encountered happens to be the NYT node, it means that a new symbol is received and to decode that symbol, we fetch out the fixed code from the received stream and decode it using the opposite procedure as done in transmitter side.
Thus, following the complete procedure as above, we can now compress and decompress any messages using Adaptive Huffman Coding.

- **Example**

Fig. 5 shows a sample example for Adaptive Huffman tree at both transmitter and receiver end. Here, the sequence to transmit is [a a r d v a r k] and the symbols are form English alphabets. [11]

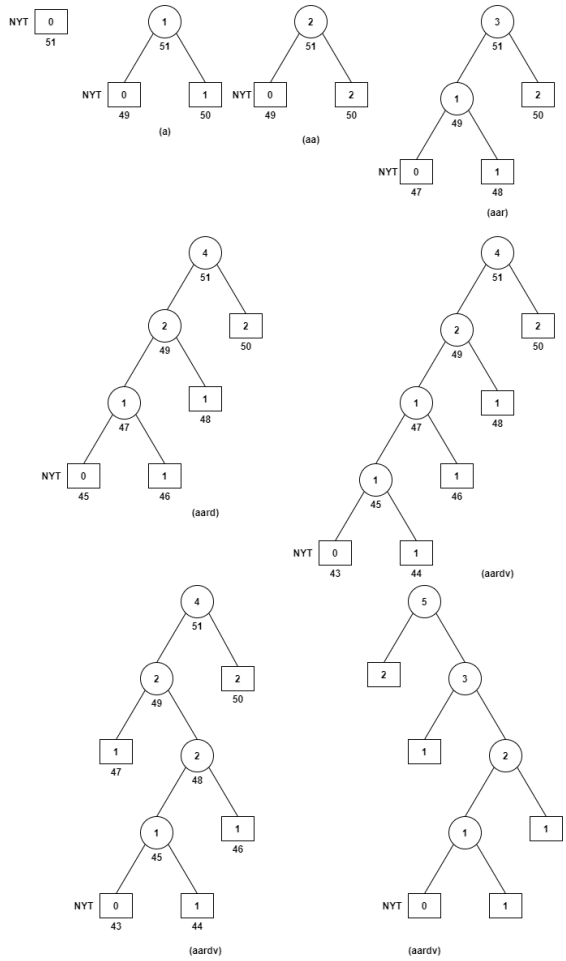

Fig. 4. Sample Example for Adaptive Huffman Coding for sequence [a a r d v a r k]

As $m = 26$, we get $e = 4$ and $r = 10$ and so after encoding, the huffman tree obtained after first few iterations are shown in fig. 5. After performing the encoding the encoded string so obtained will be "00000101000100000110001011010110001010" and similarly after decoding the encoded bits, we get our original sequence i.e aardvark. **e.g.,**

```
code = "aardvark"
The final Code we get is:
00000 1 010001 0000011 0001011 0 10  110001010
  a   a   r      d        v     a  r     k
```

Fig. 5. Decoded message from the encoded string

- **Serial Code Complexity**

  Since we need to encode each symbol one-by-one and for each symbol we need to iterate the huffman tree, the time complexity of the serial implementation will be $O(n * m)$. Here, n is the length of input string and m is the total number of symbols that can be transmitted. Also, the total number of nodes present in the huffman tree at any time will always be less than $2 * m + 1$. Thus, the space complexity of the serial algorithm will be $O(m)$.

C. Parallel Implementation

- **Need for parallelization**

  From the above serial implementation we can see that the encoding takes time of about $O(n * m)$ to transmit the message and by observing the types of application(Live streaming) this encoding scheme is used in, we need to parallelize the serial algorithm so that the symbols are transmitted at a much faster rate and better results are obtained.

- **Parallel Implementation**

  As we saw from the serial implementation that with the increase in the message size, run-time increases, so one-way we can parallelize our algorithm is to divide the complete message among p processors. Thus, doing so each processor at both the transmitter and the receiver will simultaneously work on different messages of length n/p.

  Thus, the implementation of the parallel code will be almost same as that of serial code, the only difference is that worksharing for each processor needs to be scheduled manually as the sequence in the final message matters. To achieve it, we used *pragma omp parallel* clause to create parallelism in our code and for worksharing we divided the original message into p different blocks, and each processors at both transmitter and receiver deals with the associated block. **Since we need to statically determine the worksharing, using scheduling clauses won't be helpful in our implementation.**

- **Example**

  To understand the parallel implementation properly, consider the example below.

  Fig. 6 and 7 shows the huffman tree formed at the final iteration for message string *threadreaper* for both serial and parallel implementation. The encoded string for serial implementation is obtaiend as "1001100011100100011000010011000000011100000011 11011101111111110001111111 0110" with length as 75.

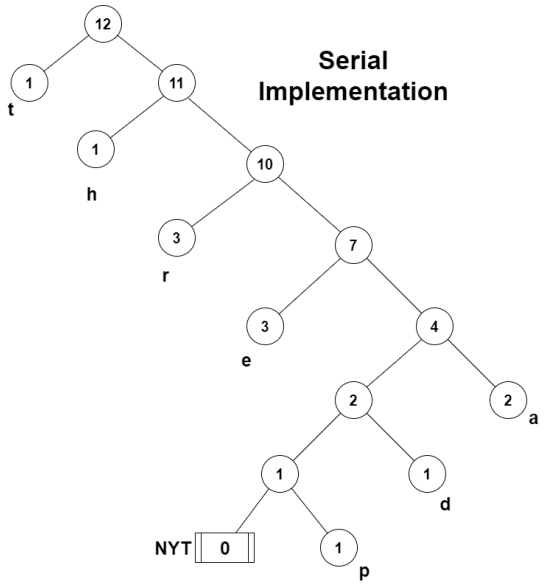

Fig. 6. Serial Implementation, Message String: 'threadreaper'

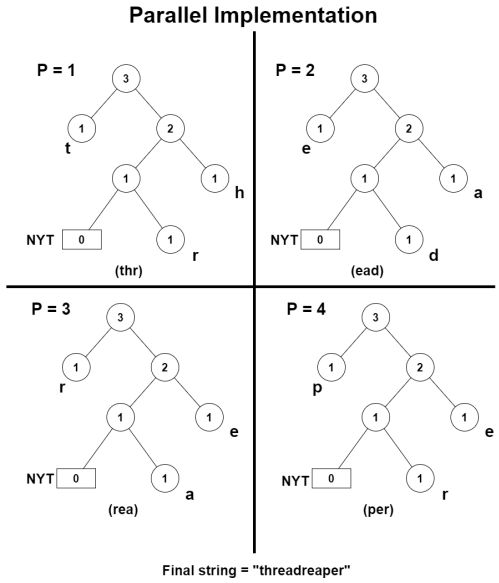

Fig. 7. Parallel implementation in each thread, Message String: 'threadreaper'

Now, for parallel implementation among 4 processors, we divided our message string into 4 sub-strings which are $("thr","ead","rea","per")$ and hence each processor will now encode a single substring and hence 4 different huffman trees will be formed. The encoded strings from each processor obtained are:

- Processor 1: thr-100110001110010001
- Processor 2: ead-0010000000000000011
- Processor 3: rea-100010001000000000
- Processor 4: per-011110001000010001

These 4 different encoded strings are now transmitted to the receiver and hence each processor at the reciever will decode the associated string and hence the message will be decoded. Further, the total length of the encoded string that is now transmitted is $4 * 18$ i.e 72 which is less as that of serial part.
Thus, parallelization results in reduction of run-time as well as more compressed data, while the overheads due to parallelization are p different huffman trees will be formed which results in increase in space complexity and also the division of string among p processors.

- **Parallel Code Complexity**

Since the complete message is now divided among p processors, the run-time of the algorithm will decrease by a factor of p while for each processor a new huffman tree will be formed and so in the worst case, space complexity will increase by a factor of p. Thus, the time complexity of our parallel implementation will be $O(n * m/p)$ and space complexity will be $O(m * p)$.

- **Hardware Details**

```
[201801464@gics0 tmp]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
Stepping:              2
CPU MHz:               2660.718
BogoMIPS:              4804.35
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              15360K
NUMA node0 CPU(s):     0-5,12-17
NUMA node1 CPU(s):     6-11,18-23
[201801464@gics0 tmp]$
```

III. Profiling

- **Gprof Profiling**

First we did the profiling using gprof for naive implementation as seen in Fig. 8 and optimized our codes respective to that.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 30.22      8.45     8.45 74758304     0.00     0.00  add_freq(node*, char)
 24.21     15.22     6.77 74758328     0.00     0.00  build(node*)
 23.80     21.87     6.65 74758328     0.00     0.00  swap(node*)
  7.71     24.03     2.16 37379164     0.00     0.00  find(node*, char, std::string&)
  2.68     24.78     0.75 37379164     0.00     0.00  find_leaf(std::string&, int&, no
  1.32     25.15     0.37        1     0.37    12.24  Decoder(std::string&, int&, int&
  1.31     25.51     0.37       24     0.02     0.02  NYTCode(node*, std::string&, cha
  1.15     25.83     0.32        1     0.32    15.57  Encoder(std::string&, int&, int&
  1.09     26.14     0.31       12     0.03     0.03  code(std::string, int, int, char
  0.82     26.37     0.23 85669119     0.00     0.00  __gnu_cxx::__normal_iterator<cha
  0.75     26.58     0.21 96579888     0.00     0.00  __gnu_cxx::__normal_iterator<cha
  0.61     26.75     0.17 246096590     0.00     0.00  __gnu_cxx::__normal_iterator<ch
  0.59     26.92     0.17 85669119     0.00     0.00  bool __gnu_cxx::operator< <char*
t&)
  0.57     27.08     0.16 37379176     0.00     0.00  void std::__reverse<__gnu_cxx::_
std::string>, std::random_access_iterator_tag)
  0.50     27.22     0.14 48289944     0.00     0.00  __gnu_cxx::__normal_iterator<cha
```

Fig. 8. G-prof of naive implementation

So, for the naive implementation we developed a separate function for each update process and so for every iteration we have to traverse the tree 2-3 times. Thus, we reduced that in our optimized algorithm in such a way that for each iteration we just need to traverse the tree only once and so better results were obtained.
Fig. 9 shows the profiling of our optimized code

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 33.74      2.76     2.76 37379152     0.00     0.00  find(node*, char, std::string&)
 15.12      3.99     1.24 37379164     0.00     0.00  find_leaf(std::string&, int&, node*, node*&)
  4.29      4.34     0.35 246096590     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::st
  4.29      4.69     0.35 74758328     0.00     0.00  std::_Bit_iterator::operator*() const
  4.16      5.03     0.34        1     0.34     6.77  Encoder(std::string&, int&, int&)
  3.37      5.31     0.28 85669119     0.00     0.00  bool __gnu_cxx::operator< <char*, std::strin
t&)
  3.12      5.56     0.26 37379176     0.00     0.00  void std::__reverse<__gnu_cxx::__normal_iter
std::string>, std::random_access_iterator_tag)
  2.76      5.79     0.23 37379164     0.00     0.00  std::_Bit_reference::operator bool() const
  2.76      6.01     0.23 74758328     0.00     0.00  std::_Bit_reference::_Bit_reference(unsigned
  2.45      6.21     0.20 74758329     0.00     0.00  std::_Bit_iterator::_Bit_iterator(unsigned l
  2.45      6.41     0.20 74758328     0.00     0.00  std::vector<bool, std::allocator<bool> >::op
  2.39      6.61     0.20 37379164     0.00     0.00  std::_Bit_reference::operator=(bool)
  2.20      6.79     0.18 37379176     0.00     0.00  void std::reverse<__gnu_cxx::__normal_iterat
d::string>)
```

Fig. 9. G-prof of optimized implementation

Thus, we can see that few function calls are reduced in the optimized implementation and several functions are calls to stopped which we can compare clearly. Further, to stop traversing in the tree every time to check whether the character is already transmitted on not, we use a bool array to store the same, i.e if the character is already transmitted, we mark it as true and so we in the optimized we can just use that array to check the same. This way we optimzed our naive implementation to get better results.

- **Valgrind Profiling**

For memory management of our optimzed code, we used valgrind profiler and the results obtianed are shown below.

```
==16045==
==16045== HEAP SUMMARY:
==16045==     in use at exit: 70,464 bytes in 1,967 blocks
==16045==   total heap usage: 79,669,166 allocs, 79,667,199 frees, 2,843,694,574 bytes allocated
==16045==
==16045== LEAK SUMMARY:
==16045==    definitely lost: 37,760 bytes in 1,180 blocks
==16045==    indirectly lost: 24,576 bytes in 768 blocks
==16045==      possibly lost: 4,560 bytes in 15 blocks
==16045==    still reachable: 3,568 bytes in 4 blocks
==16045==         suppressed: 0 bytes in 0 blocks
==16045== Rerun with --leak-check=full to see details of leaked memory
==16045==
==16045== For lists of detected and suppressed errors, rerun with: -s
==16045== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Fig. 10. Memory management profiling using Valgrind

```
==25102==
==25102== I   refs:      51,278,101,449
==25102== I1  misses:             3,423
==25102== LLi misses:             3,379
==25102== I1  miss rate:           0.00%
==25102== LLi miss rate:           0.00%
==25102==
==25102== D   refs:      22,723,165,736  (15,207,373,316 rd   + 7,515,792,420 wr)
==25102== D1  misses:        19,749,860  (    10,486,124 rd   +     9,263,736 wr)
==25102== LLd misses:        15,992,460  (     7,625,682 rd   +     8,366,778 wr)
==25102== D1  miss rate:            0.1% (           0.1%     +           0.1%  )
==25102== LLd miss rate:            0.1% (           0.1%     +           0.1%  )
==25102==
==25102== LL refs:           19,753,283  (    10,489,547 rd   +     9,263,736 wr)
==25102== LL misses:         15,995,839  (     7,629,061 rd   +     8,366,778 wr)
==25102== LL miss rate:             0.0% (           0.0%     +           0.1%  )
```

Fig. 11. Memory management profiling using Valgrind

From Fig. 11 we can see that the miss rate in L1 data cache is 0.1%, that means our implementation is memory efficient.

IV. Results of Open MP implementation

- **Run-time as a function of message size**

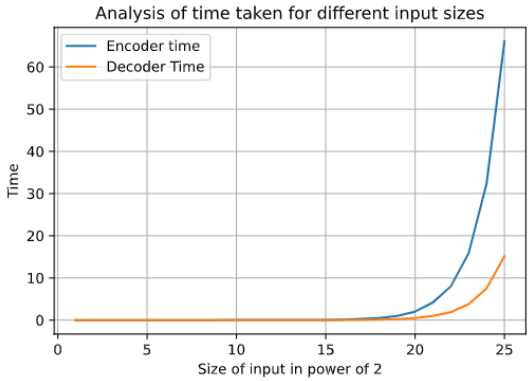

Fig. 12. time taken for parallel implementation on 16 cores

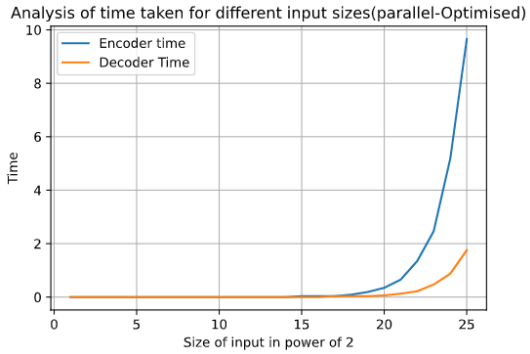

Fig. 13. time taken for parallel implementation on 16 cores

Fig. 12 and 13 shows the run-time for our optimized implementation of serial and parallel code as a function of input size i.e message size.
From the figures it is clear that for very small input size, the run-time is approximately zero, but as the size increases the time taken for execution increases linearly. Also, the same result is seen for the parallel implementation. To mathematically show it, we know that the run-time is directly proportional to the message size, and so the run-time is showing linear relation to message's length(n) as m and p remains constant. Here, the encoder takes more time as compared to the decoder because more computations are required to perform to encode the message than to decode. i.e to calculate the fixed code for each new encountered character and other to reverse the path of each character in the tree to add it to encoded message.

- **Results for different inputs in terms of repetition**

For in-depth analysis we created a custom string in which the probability of repetition of a symbol can be changed. Here, for this case $r = 6, e = 64$. we varied the probability of repetition and observed how it affects compression ratio and encoding time through our serial implementation.

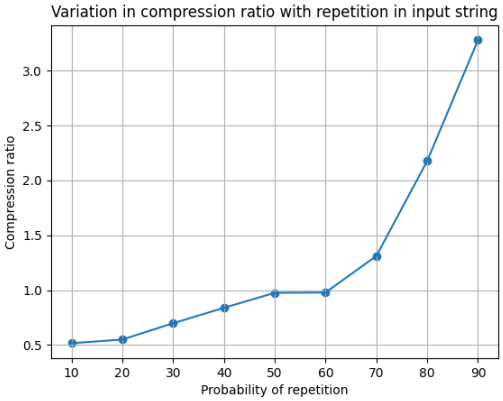

Fig. 14. Compression ratio with for different probability of repetitions

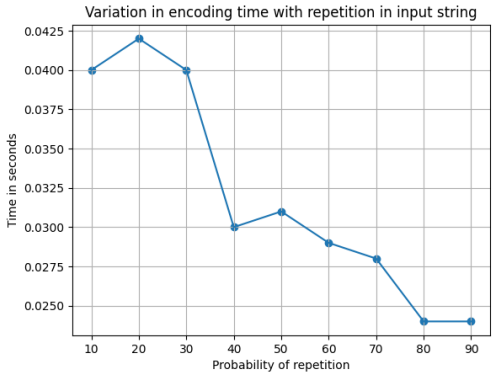

Fig. 15. Encoding time with varying the probability of repetition

- We can observe from Fig. 14 that, compression ratio is increasing as we increase the probability of repetition. This reason for this is simple, as we just need to pass the character's path to the encoded message.
- From fig. 15 we can see that time to encode the string decreases as probability increase. The reason for this is number of extra computations to perform decreases as repetitiveness increases.
- Thus, from this part we can conclude that Adaptive huffman coding is more efficient when the chances of repetitiveness in the string is more.

- **Results for Bonsai and Aneurism data-sets**

  For encoding the provided standard data-sets, we first converted .raw file to .txt file which contains unsigned integers ranging from $0 - 255$. Using file handling, we took the complete .txt file into a string and considered a string as our message and implemented the same. Further, we took the fixed code for each character as the binary representation of its ASCII value, and so for this case, there will be 128 different symbols which can be transmitted which implies that $e = 6$ and $r = 64$.

  Table below shows the results obtained from both naive and optimized approach for both serial and parallel implementation for bonsai and aneurism data-sets. We have considered 16 processors for parallel implementation.

Data set: Bonsai			
Algorithm	Encoder Time(sec)	Decoder Time(sec)	Compression ratio
Serial (naive approach)	20.4062	11.4375	2.55
Serial (Optimized version)	**11.9375**	**1.4375**	**2.55**
Parallel (naive approach)	3.59375	10.9688	3.02
Parallel (optimized version)	**2.65625**	**0.25**	**3.02**
Data set: Aneurism			
Algorithm	Encoder Time(sec)	Decoder Time(sec)	Compression ratio
Serial (naive approach)	14.1875	9.125	5.12
Serial (Optimized version)	**7.15625**	**0.75**	**5.12**
Parallel (naive approach)	2.3475	8.6875	5.14
Parallel (optimized version)	**1.78125**	**0.15625**	**5.14**

From the results we can see that parallel algorithm shows better results as compared to serial for both bonsai and aneurism data-sets with respect to run-time and compression ratio. Also, optimized approach is better than naive implementation in terms of run-time while encoding will remain same as we have only optimized the code by avoiding some unnecessary calls.

The run-time of aneurism data set is less than that of bonsai is beacuse the length of message for aneurism data-set is less than bonsai. Further the compression ratio for aneurism data set is high, because aneurism data-sets contained more repetitive symbols.

- **Curve Related Analysis for Aneurism and Bonsai datasets**
  - **Aneurism Data:**

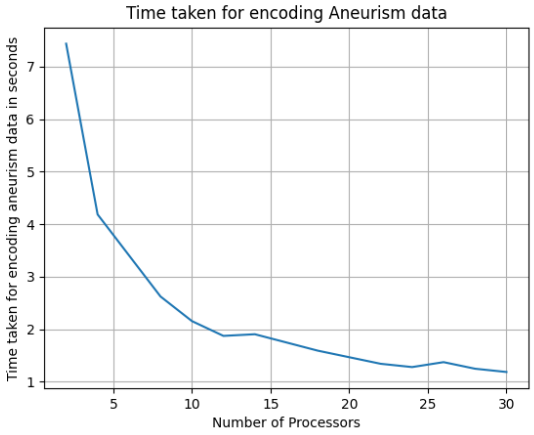

Fig. 16. Time Taken for Encoding the Data, with high repetition (Optimised version)

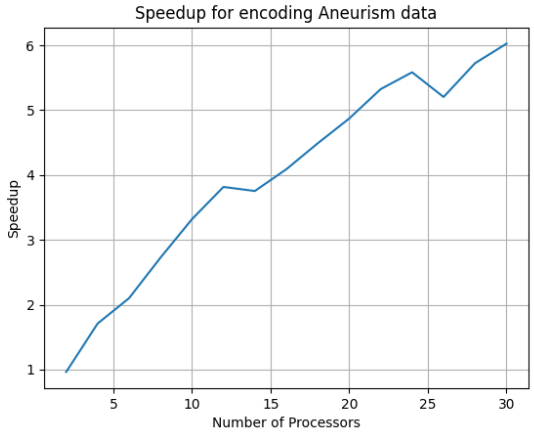

Fig. 17. Speedup for Encoding the Data, with high repetition (Optimised version)

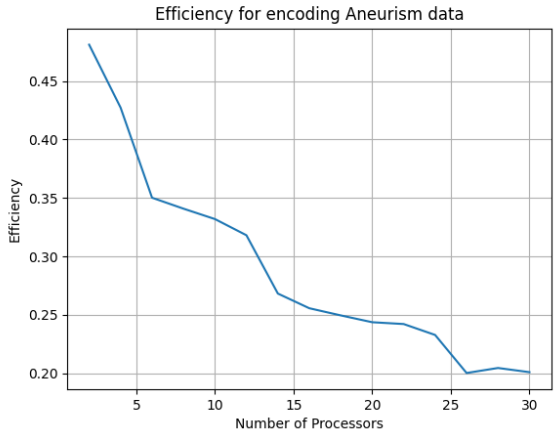

Fig. 18. Efficiency for Encoding the Data, with high repetition (Optimised version)

– **Bonsai Data:**

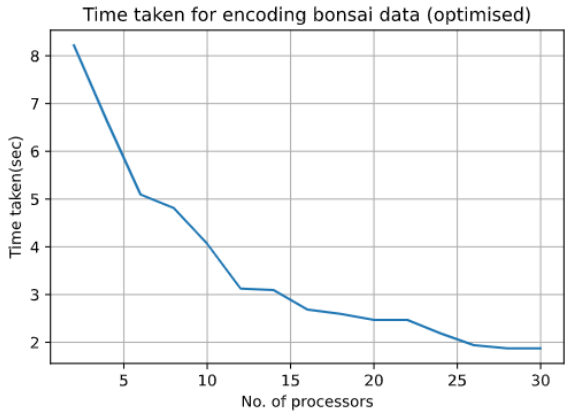

Fig. 19. Time Taken for Encoding the Data, with low repetition (optimised version)

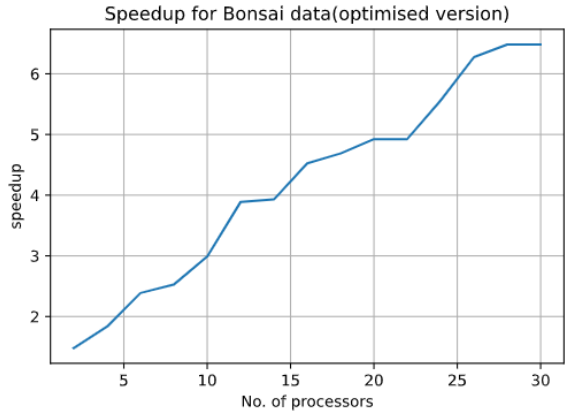

Fig. 20. Speedup for Encoding the Data, with low repetition (Optimised version)

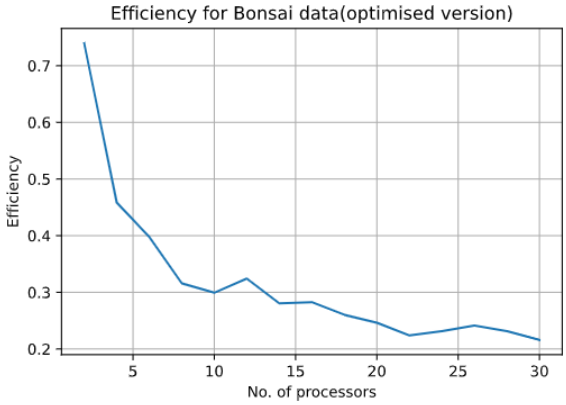

Fig. 21. Efficiency for Encoding the Data, with low repetition (Optimised version)

- **Analysis**
  - As we can observe from Fig. 16 and Fig. 19 that, for both the data sets the time taken for encoding decreases as we increase the numbers of processors. To explain it mathematically, we know that run-time of parallel algorithm is inversely proportional to p. Thus, as we increase p, run-time decreases.
  - In case of speedup also if we observe the Fig. 17 and Fig. 20 speedup increases as number of processor increase which was expected. But, here we can see that we are getting sub-linear speedup which is explained below.
  - **Explanation of Sub linear Speedup**
    - Our implementation of the Encoder has a reverse function, so every code calculated must be reversed before it returns, so it adds a linear time complexity for each iteration.
    - Apart from that our parallel implementation of the code divides the input stream into smaller pieces, so the frequency of characters also splits up among the processors, now a smaller frequency means a larger code for the letter, thus it takes up more time
    - Further, we have a critical section in our parallel code to avoid race-conditions. This also acts as a parallel overhead for our implementation.
    - So the above mentioned points contributes to the sub linear speedup in our graphs.
  - Also from the efficiencies graph i.e Fig. 18 and 21, we can see that efficiency decreases with the increase in processors, which is as expected from the speedup curves.

V. Results following Amdahl's law

For the following results, we plot the efficiency curve for different processors with fixed problem size and the

same for different problem size for fixed processors. The results obtained are as follows:

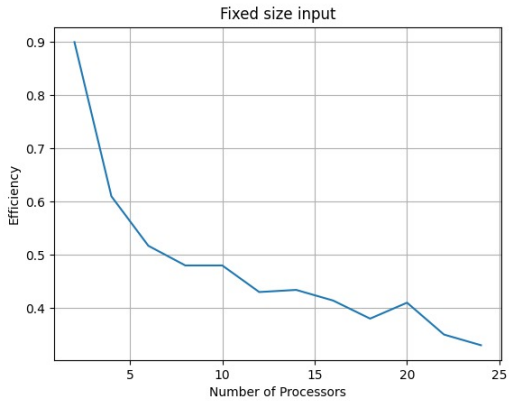

Fig. 22. Efficiency versus processors for fixed problem size of 10^7

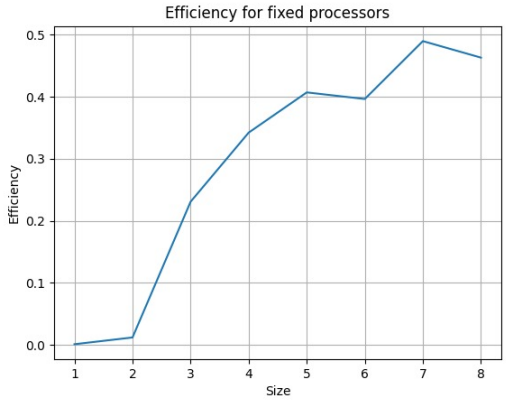

Fig. 23. Efficiency versus problem-size for fixed number of processors i.e 16

Fig. 22 and 23 follows the expected results. i.e it follows Amdahl's law viz. for fixed problem size, as the number of processors increases efficiency decreases and for fixed number of processors, efficiency increases as problem size increases.

VI. MPI Implementation

- We implemented the MPI code for this problem, as we did in the parallel algorithm, the same way we divided the input stream and gave it to different processes.
- The zeroth process is the master process and other processes will send the encoded strings to master process using MPI send
- The master process will receive the strings one by one from the slave processes using MPI receive.
- Here the received string could be of different sizes, so we need to know their sizes beforehand so we could assign the same size to the buffer.
- To achieve that we used the MPI probe function and MPI get count function, the probe function probes the incoming message's status first and the the get count function return the length of the incoming message.
- Finally we receive the message and then sort them according to the processes' number.

VII. Results of MPI Implementation

We obtained the results for a standard database of bonsai. Here, also we approached in a same way as we did in OpenMP implementation that at first we created a txt file and then encoded it through our MPI code.

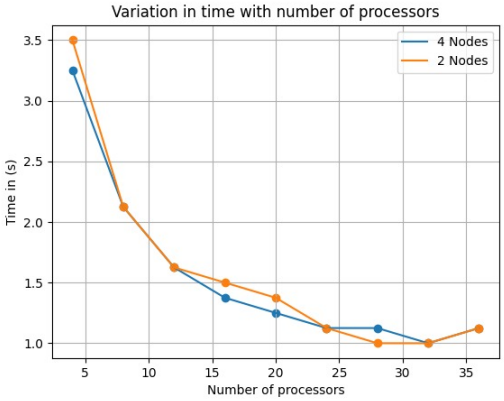

Fig. 24. Encoding time for different number of nodes with varying size of input stream.

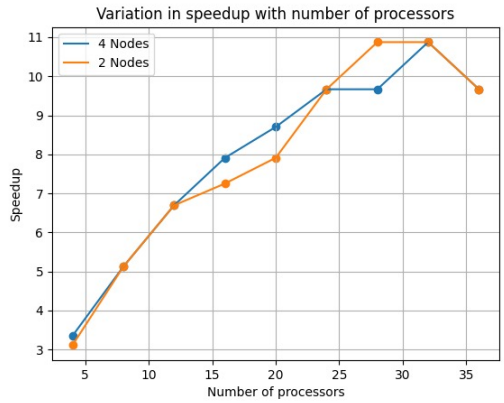

Fig. 25. Speedup for different number of nodes with varying size of input stream.

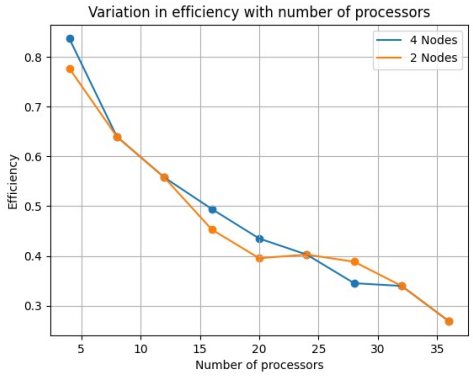

Fig. 26. Efficiency for different number of nodes with varying size of input stream.

– Fig. 24, 25 and 26 shows the average execution time,speedup and efficiency as a function of number of processors, we can see that the results are same as that as obtained for Open MP implementation. Further, comparing the plots we can see that MPI gives better results as it is also a memory-bound problem.

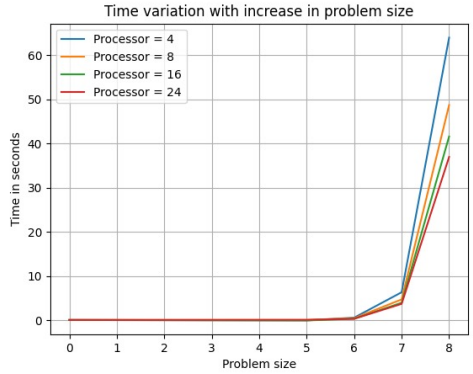

Fig. 27. Execution time versus problem size for varying process

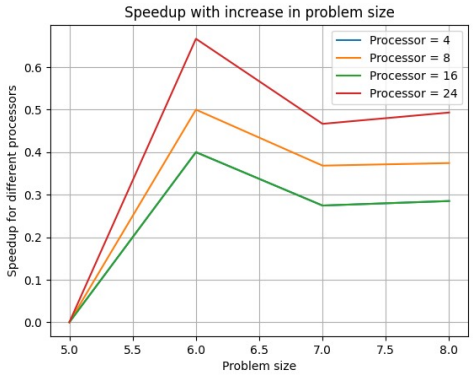

Fig. 28. Speedup versus problem size for varying process

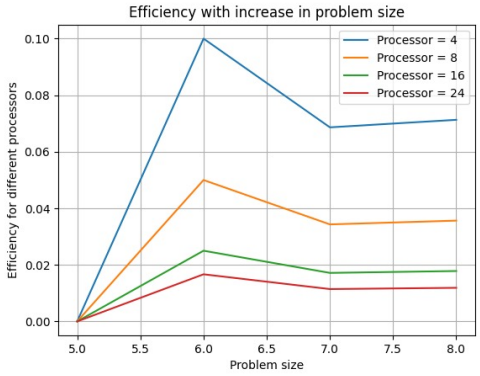

Fig. 29. Efficiency versus problem size for varying process

– Fig. 27, 28 and 29 shows the average execution time,speedup and efficiency as a function of number of problem size. As seen as the problem size increases, the mean execution time also increases, but it follows Amdahl's law i.e for fixed problem size we can see that time decreases as processors increases.
– While from the speedup and efficiency curve, we can see that as problem size increases efficiency increases which follows Amdahl's law.

VIII. Quantitative CMA analysis

For CMA analysis, considering the worst case, for each iteration we need to traverse the huffman tree and for computation we calculate the huffman path which can be worst as the depth of the tree.
Thus, for m different symbols, the huffman tree will be of $2 * m + 1$ nodes while the maximum depth will be m. Thus,

$$CMA_{worst-case} = \frac{m}{2m+1}$$

Hence, $CMA < 1$, implies our problem is memory-bound.

References

[1] Huffman Coding

texttthttps://www.researchgate.net/publication/304395425_Huffman_coding

[2] Introduction to Data Compression (Fifth Edition) - Khalid Sayood, Pages: 67-75

[3] A Fast and Improved Image Compression Technique U sing Huffman Coding

texttthttps://www.researchgate.net/publication/303313377_A_fast_and_Improved_Image_Compression_Technique_using_Huffman_Coding

[4] Open Scientific Visualization Datasets
https://klacansky.com/open-scivis-datasets/

[5] Jagadeesh B, Ankitha Rao, 2013, An approach for Image Compression Using Adaptive Huffman Coding, INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY (IJERT) Volume 02, Issue 12 (December 2013),
https://www.ijert.org/?s=IJERTV2IS121242

[6] Dave Marshall, Adaptive Huffman Coding
https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node212.html

[7] A Survey of Data Compression Algorithms and their Applications, Hosseini, Mohammad
https://doi.org/10.13140/2.1.4360.9924

[8] Duke Computer Science, Jonathan Low
https://www2.cs.duke.edu/csed/curious/compression/adaptivehuff.html

[9] A Memory-Efficient Adaptive Huffman Coding Algorithm for Very Large Sets of Symbols Revisited, Steven Pigeon and Yoshua Bengio
http://www.iro.umontreal.ca/~lisa/pointeurs/TechRep_AdaptativeHuffman2.pdf

[10] Data Compression, Published by University of London
https://london.ac.uk/sites/default/files/study-guides/data-compression.pdf

[11] Adaptive Huffman Coding - Geeksforgeeks
https://www.geeksforgeeks.org/adaptive-huffman-coding-and-decoding/

[12] Adaptive Huffman Coding - Wikipedia
https://en.wikipedia.org/wiki/Adaptive_Huffman_coding