HIGH PERFORMANCE COMPUTING: PARALLELIZING THE HUFFMAN ENCODING LOSSLESS DATA COMPRESSION ALGORITHM

By
Frank Vega-Aguirre
Eleanor Little
Ian Schmidt

SUMMARY
Our goal was to test the performance speedup (if any) yielded by parallelizing the Huffman encoding/decoding algorithm, a lossless data compression algorithm. In order to test for any improvement via parallelization, we wrote a program in C and used the Message Passing Interface (MPI) library. Our program was built around testing the encoding of plain text files, but the algorithm's core concepts can be extended to other data types and formats. At its core, our application did the following: split the source stream up into chunks, assigned a chunk to each processor, encoded the chunks into an intermediate buffer in parallel, concatenated the encoded results from the intermediate buffer into the final output buffer, and finally did the aforementioned steps in reverse to decode the given data. We ran the final program several times remotely on the Blue Gene\Q (BGQ) supercomputer under varied MPI rank and supercomputer core configurations.

BACKGROUND
Compression is the process of encoding information using fewer bits (or other information-bearing units) than an un-encoded representation would use through use of specific encoding schemes. In order for compression to be useful in any sense, compressed items must be able to be retrieved in their original form without data loss (i.e. lossless compression). The process of compressing this data is known as encoding, and its reversal, its "decompressing", is known as decoding. Several lossless compression algorithms are well documented. One of these well-known algorithms is known as the Huffman compression algorithm. The Huffman compression algorithm works on a set of symbols (in this case plain text characters) and some given weights (usually proportional to probabilities). The algorithm then proceeds to find a prefix-free binary code with the minimum expected codeword length (which is often a tree with the minimum weighted path length from the root.)

RESEARCH
There has been many different experiments done on the Huffman encoding over time as the Huffman coding was first published in 1952. Experiments have been done in a variety of different ways, focusing on general performance, power efficiency, performance on specific types of data, and some even on creating hardware for Huffman encoding working with Lempel-Ziv compression algorithm specifically. In most experiments the Huffman encoding is used in sequence with another compression algorithm, where the other data compression algorithm runs first and the output is fed to the Huffman encoding for further processing. In the case of the Parallel Apparatus for High-Speed, Highly Compressed LZ77 Tokenization and Huffman Encoding for Deflate compression, which was the device specifically made for parallel encoding, the system broke what was going to be encoded into pieces and fed the different chunks to the LZ77 compression method and then brought the chunks together into the Huffman encoding portion where it was compressed further (Milne). By first using a different algorithm to simplify the data allows for higher performance in the Huffman encoding. Another interesting phenomenon with Huffman encoding is that after a given length there is a synchronization point when decrypting if done in chunks, as would be in parallel. This is a point where even if the piece of data being decrypted was not started at the correct point so that it will give the correct data the decryption will get itself back on track and begin decrypting on the correct path again (Klein). This is both a blessing and a curse as it will mean that even if the paralyzation is done poorly and the decryption is split in spaces it should not be that some of the correct data will be decrypted. However at the same time it also means that a partial positive will return when the implementation is in fact not correct at all.

ALGORITHM
The Huffman Encoding/Decoding algorithm used was based on a serial implementation of Huffman encoding (Gyaikhom). The source algorithm read in input from a file, calculated the frequency of each character in the input, then generated a Huffman encoding tree that was then used to encode the input. A header file with information about the encoding was created and the encoded input was output to a separate file. The algorithm could then be run again on the encoded file to decode the file, it would read the header file to be able to recreate the Huffman tree that was then used to decode the input and the now decoded input would be output to a file. This source algorithm has been modified heavily to support parallelization and has been optimized for running on the BG/Q. The main focus of the

parallelization was seeing the speed up of the decoding and encoding portions of the algorithm, as such, parts of the original algorithm have been removed to reduce extra steps since the decoding was being run immediately following the encoding. In the interest of only evaluating the effects of parallelization on encoding and decoding input, the writing and reading of header files has been excluded since it is just reading and writing to files in parallel.
The algorithm we implemented reads the input into an array and then uses MPI_Scatter() to send chunks of that array to each MPI rank. Each MPI rank will then calculate the frequency of each character in its subsection of the input and create a Huffman encoding tree based off of those frequencies. That tree will then be used to encode that rank's section of the input. Since the writing and reading of header files has been omitted, the decoding part of the algorithm will then run using the Huffman encoding tree generated during the encoding phase. Each rank will decode their encoded input. Finally, MPI_Gather() is used to collect all the parts of the decoded input, put them in order and return them to rank 0 so that the input can be output to a file.

APPROACH

The algorithm was run remotely on the BGQ via SSH login. We used the BGQ's processor frequency to determine the execution time of the Huffman compression algorithm's two star features: encoding, and decoding. We ran the program under a variety of configurations in order to test any deviations in execution time; these configurations consisted of the following:

- a serialized run (1 compute node and 1 MPI rank)
- 8 runs at varying compute nodes (1 -128) with MPI ranks fixed at 2 MPI ranks per compute node (2-256 total MPI ranks)
- 8 runs at varying compute nodes (1-128) with MPI ranks fixed at 4 per compute node (4-512 total MPI ranks)
- 8 runs at varying compute nodes (1-128) fixed at 8 MPI ranks per compute node (8-1024 total MPI ranks)
- 8 runs at varying compute nodes (1-128) fixed at 16 MPI ranks per compute node (16-2048 total MPI ranks)
- 8 runs at varying compute nodes (1-128) fixed at 32 MPI ranks per compute node (32-4096 total MPI ranks)
- 8 runs at varying compute nodes (1-128) fixed at 64 MPI ranks per compute node (64-8,192 total MPI ranks)
- 7 runs at varying compute nodes (2-128) fixed at 128 MPI ranks per compute node (256-16,384 total MPI ranks)
- 6 runs at varying compute nodes (4-128) fixed at 256 MPI ranks per compute node (1024-32,768 total MPI ranks).

Each individual program run generated an output file that consisted of the encoding execution time (time measured from the start to the end of the encoding section of the Huffman compression algorithm), and the decoding execution time (time measured from the start to the end of the decoding section of the Huffman compression algorithm). These outputs were fed into a python script that leveraged the numpy, panda, and matplotlib libraries to produce plots that included:

- Encode and decode times vs compute nodes for each set of program runs at N, where N is 1-256, fixed MPI ranks (plotted with serial times on graph)
- Encode times vs total MPI ranks for each set of program runs N, where N is 1-256, fixed MPI ranks
- Decode time vs total MPI ranks for each set of program runs N, where N is 1-256, fixed MPI ranks
- Total MPI ranks vs average running time of both encode and decode operations, where the average running time is the calculated mean of all runtimes at N total MPI ranks.

RESULTS

The following figures are the python-generated graphs from the results of our Huffman compression algorithm runs on the BGQ.

Fig. 1

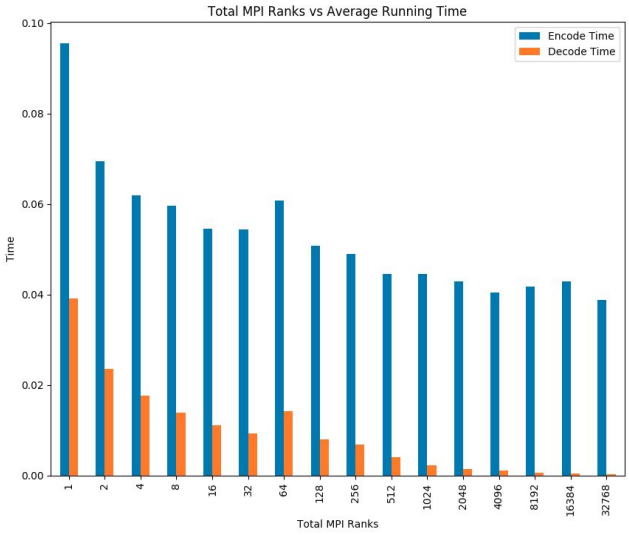

Fig. 1: Plots the total number of MPI ranks on the x-axis vs the average execution time for both encoding and decoding.

Fig. 2

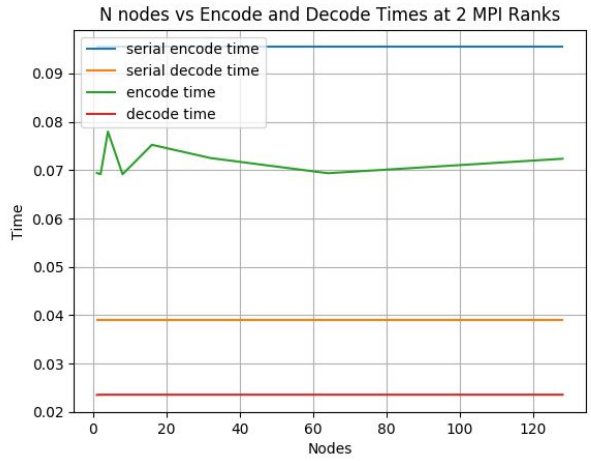

Fig. 2: Plots both the encoding and decoding times for 1-128 compute nodes, each fixed at 2 MPI ranks, alongside the serial encode and decode execution times.

Fig. 3

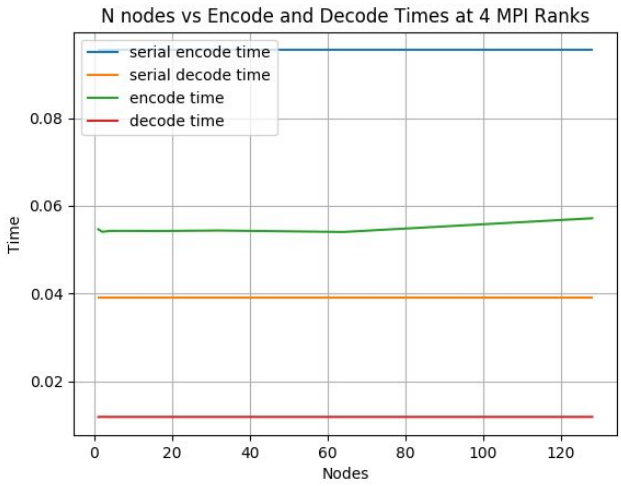

Fig. 3: Plots both the encoding and decoding times for 1-128 compute nodes, each fixed at 4 MPI ranks, alongside the serial encode and decode execution times.

Fig. 4

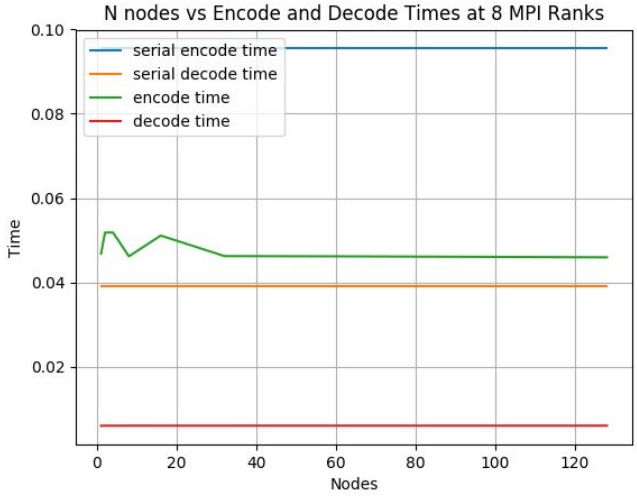

Fig. 4: Plots both the encoding and decoding times for 1-128 compute nodes, each fixed at 8 MPI ranks, alongside the serial encode and decode execution times.

Fig. 5

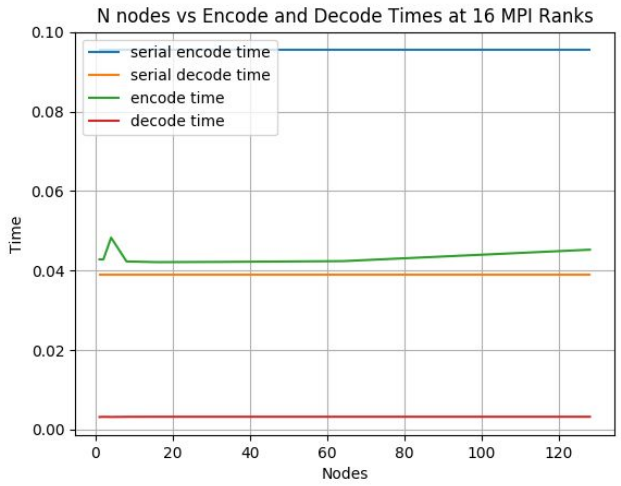

Fig. 5: Plots both the encoding and decoding times for 1-128 compute nodes, each fixed at 16 MPI ranks, alongside the serial encode and decode execution times.

Fig. 6

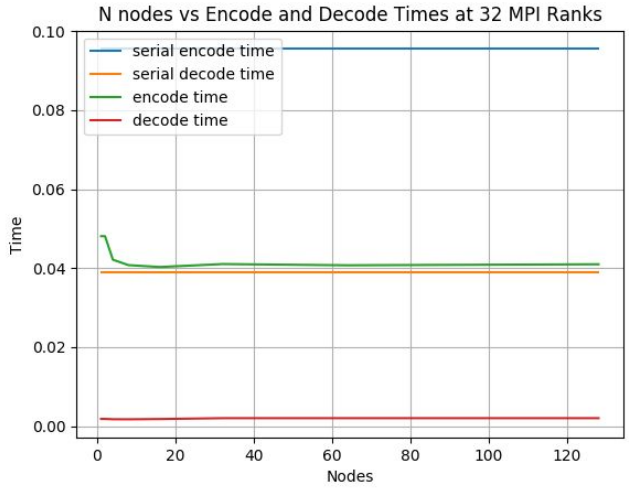

Fig. 6: Plots both the encoding and decoding times for 1-128 compute nodes, each fixed at 32 MPI ranks, alongside the serial encode and decode execution times.

Fig. 7

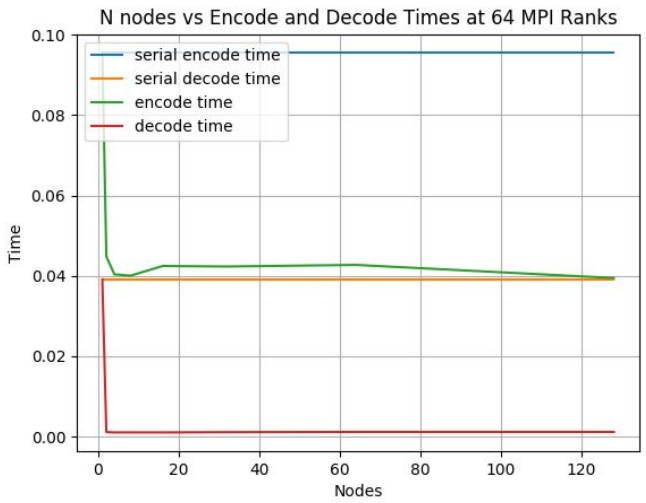

Fig. 7: Plots both the encoding and decoding times for 1-128 compute nodes, each fixed at 64 MPI ranks, alongside the serial encode and decode execution times.

Fig. 8

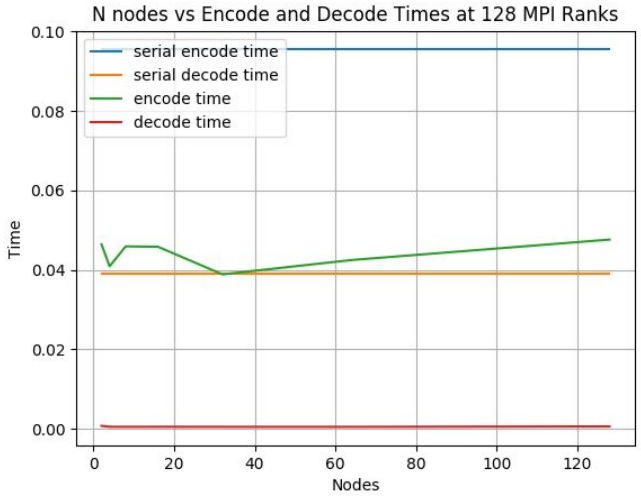

Fig. 8: Plots both the encoding and decoding times for 2-128 compute nodes, each fixed at 128 MPI ranks, alongside the serial encode and decode execution times.

Fig. 9

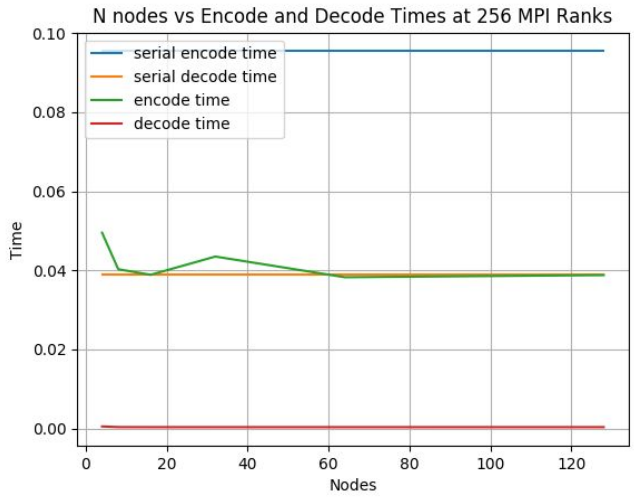

Fig. 8: Plots both the encoding and decoding times for 4-128 compute nodes, each fixed at 256 MPI ranks, alongside the serial encode and decode execution times.

Fig. 10

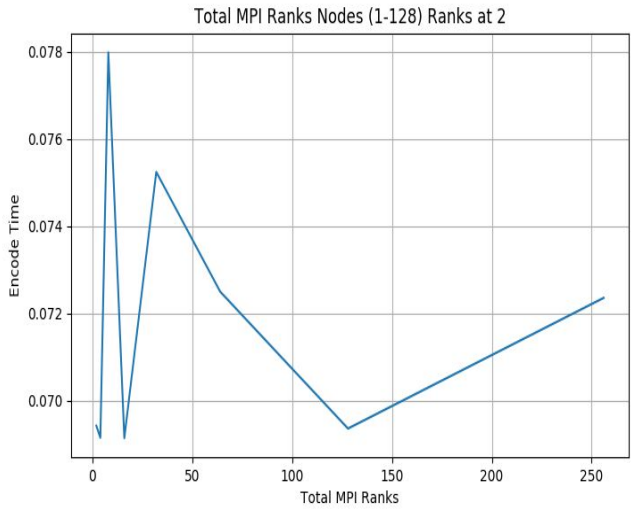

Fig. 10: Plots the encoding time for 1-128 compute nodes each fixed at 2 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 11

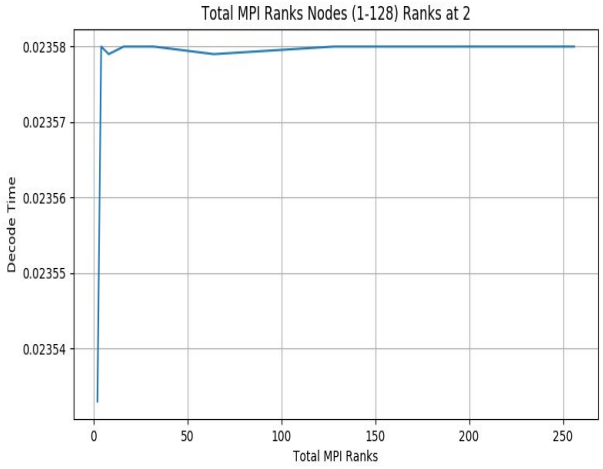

Fig. 11: Plots the decoding time for 1-128 compute nodes each fixed at 2 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 12

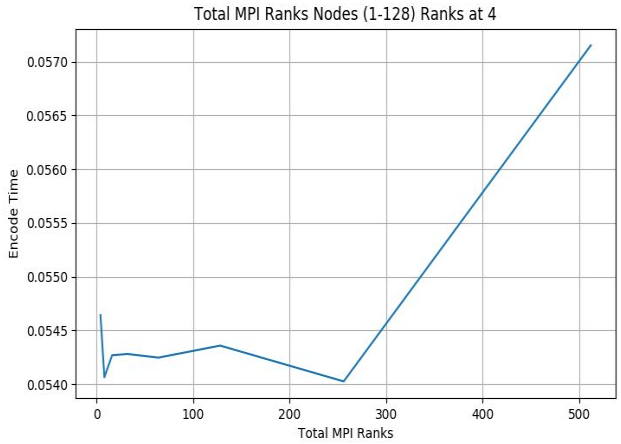

Fig. 12: Plots the encoding time for 1-128 compute nodes each fixed at 4 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 13

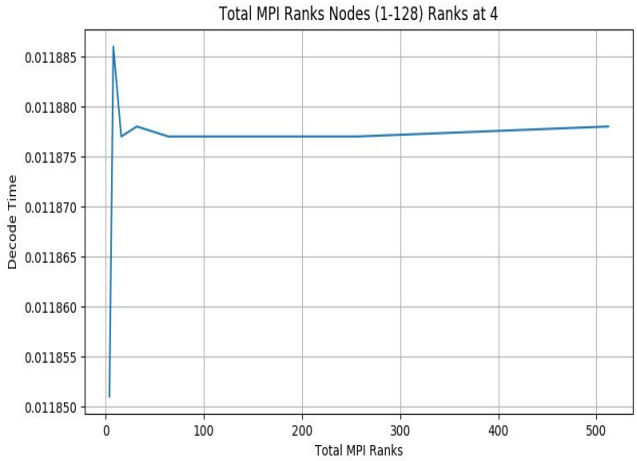

Fig. 13: Plots the decoding time for 1-128 compute nodes each fixed at 4 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 14

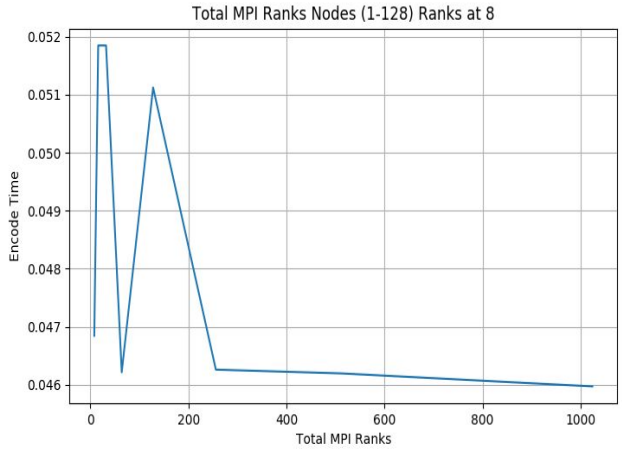

Fig. 14: Plots the encoding time for 1-128 compute nodes each fixed at 8 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 15

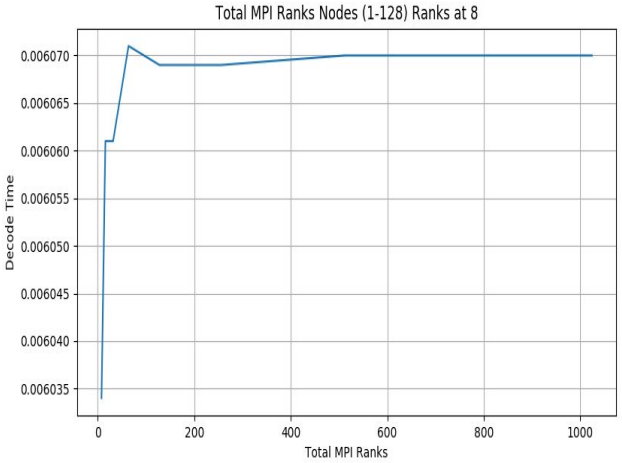

Fig. 15: Plots the decoding time for 1-128 compute nodes each fixed at 8 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 16

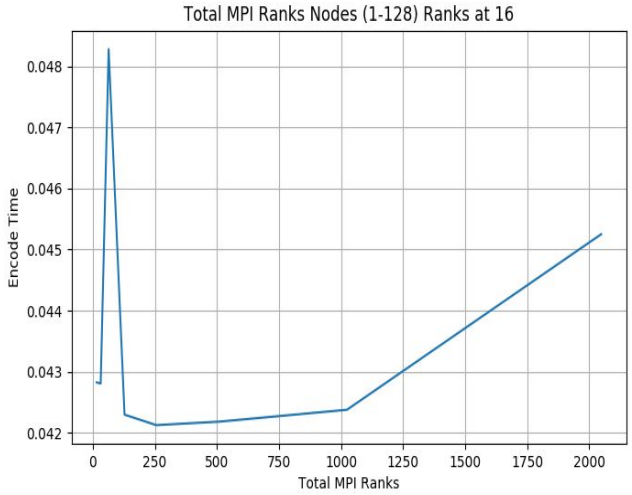

Fig. 16: Plots the encode time for 1-128 compute nodes each fixed at 16 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 17

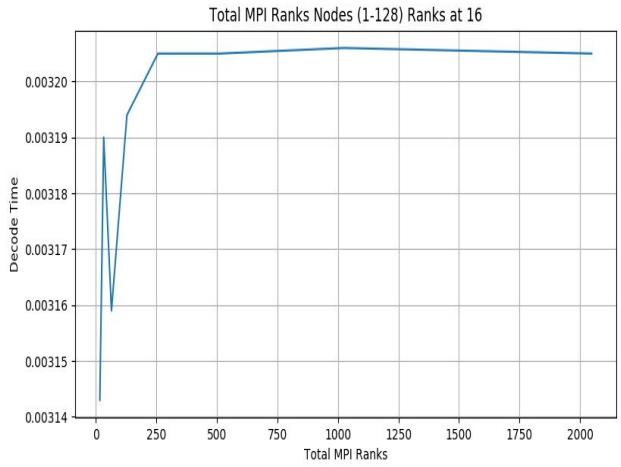

Fig. 17: Plots the decoding time for 1-128 compute nodes each fixed at 16 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 18

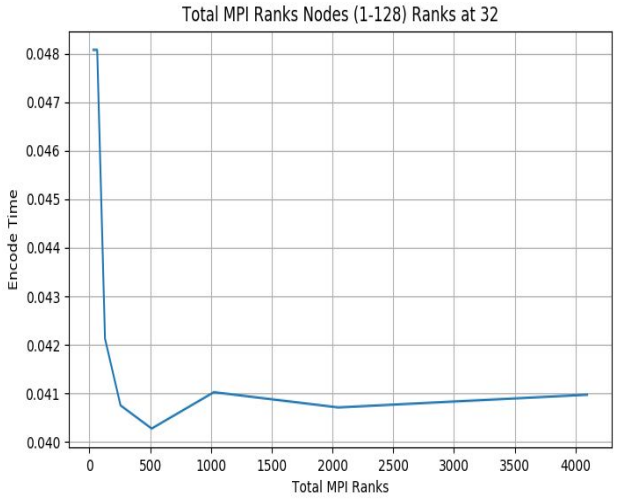

Fig. 18: Plots the encode time for 1-128 compute nodes each fixed at 32 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 19

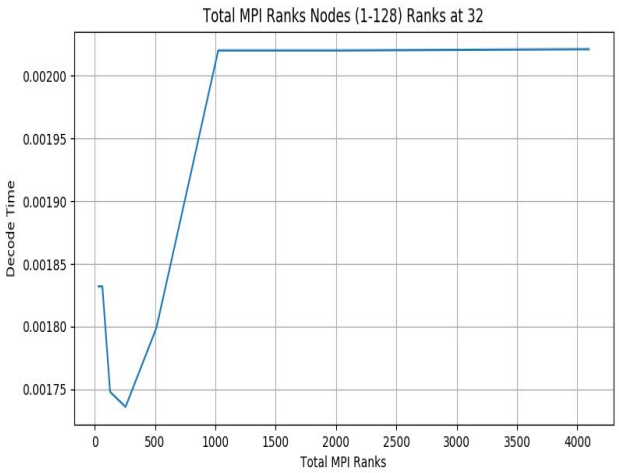

Fig. 19: Plots the decoding time for 1-128 compute nodes each fixed at 32 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 20

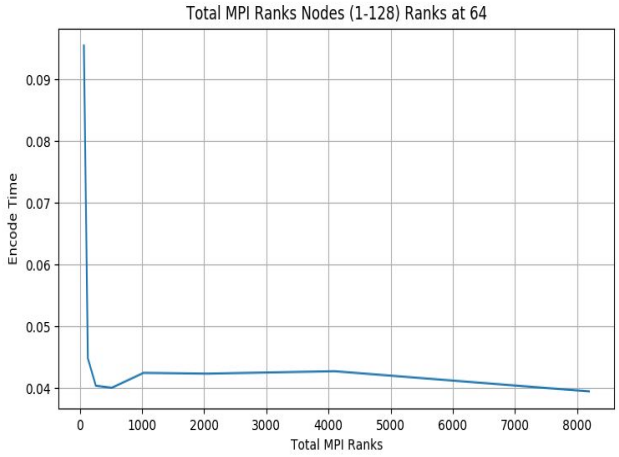

Fig. 20: Plots the encode time for 1-128 compute nodes each fixed at 64 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 21

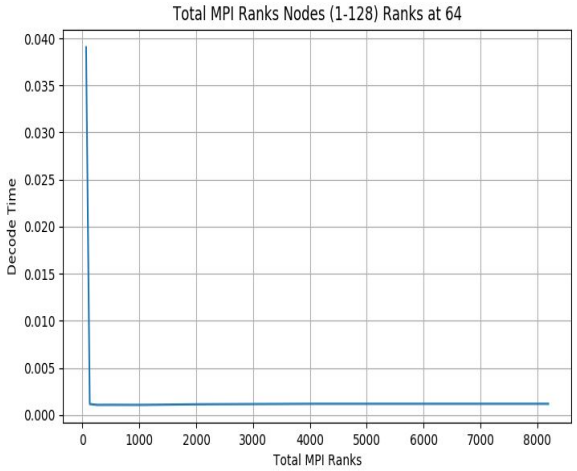

Fig. 21: Plots the decoding time for 1-128 compute nodes each fixed at 64 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 22

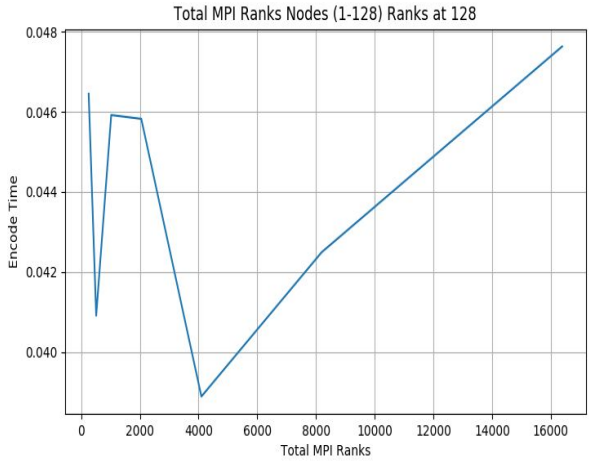

Fig. 22: Plots the encode time for 2-128 compute nodes each fixed at 128 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 23

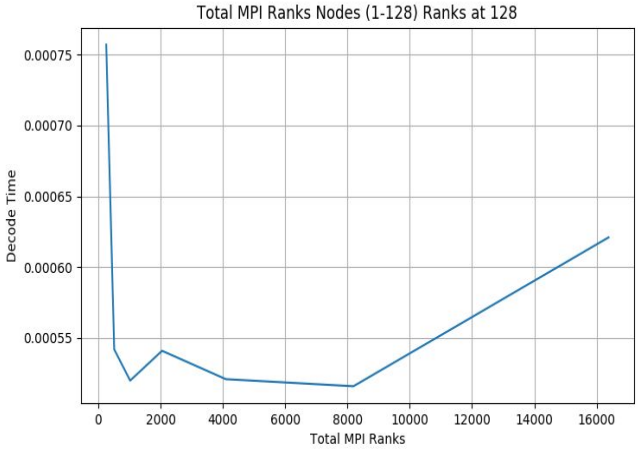

Fig. 23: Plots the decoding time for 2-128 compute nodes each fixed at 128 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 24

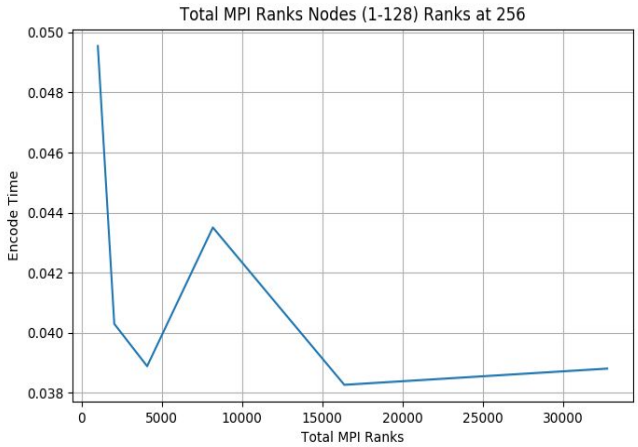

Fig. 24: Plots the encode time for 4-128 compute nodes each fixed at 256 MPI ranks vs the total MPI Ranks at each compute node.

Fig. 25

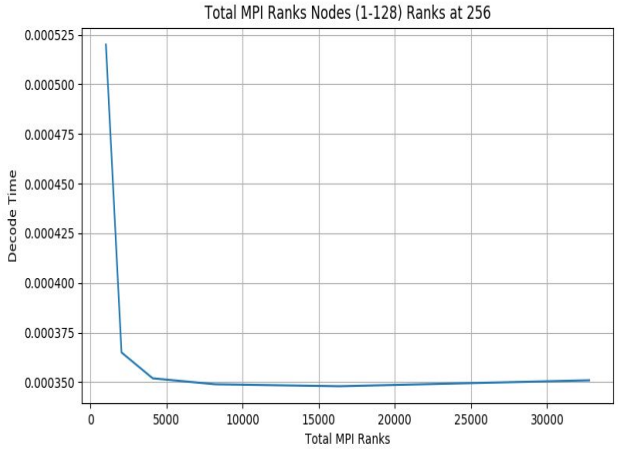

Fig. 25: Plots the decoding time for 4-128 compute nodes each fixed at 256 MPI ranks vs the total MPI Ranks at each compute node.

ANALYSiS

Figure 1 best illustrates that both the encode and decode times generally improve as the total number of MPI ranks increase; however, there is a small deviation in this trend at some points. For example, at 64 total MPI ranks the execution time for both encode and decode operations decrease by about 10 and 35 percent respectively:

$$Speedup = \frac{T0}{T1}$$
$$Se = \frac{0.054450}{0.060796} = 0.895619$$
$$Sd = \frac{0.009308}{0.014262} = 0.652680$$

Furthermore, Figure 1 also demonstrates the highest average execution time speedup between the serial version of the program and the parallelized version. The parallelized program executes the encoding and decoding operations about 2.5x and 112x faster respectively at 32,768 total MPI Ranks (or at 128 compute nodes and 256 MPI Ranks per compute node):

$$Se = \frac{0.095453}{0.038809} = 2.459558$$
$$Sd = \frac{0.039051}{0.000351} = 111.256410$$

On talk of compute performance speedup, we look again to Figure 1. The maximum encode speedup in relation to a pair of consecutive total MPI ranks occurs between total MPI ranks at 1 (or the serial run) and total MPI ranks at 2 (the parallelized run with the lowest total MPI ranks). The speedup here is a factor of about 1.4:

$$Se = \frac{0.095453}{0.069429} = 1.374829$$

Even numbered figures (Fig. 10-24) back this encode speedup claim, but said figures also highlight other trends; for example, Fig. 10 reinforces that the maximum encode speedup occurs between the program's serial run and it's first parallelized run, but from the same figure, we take note that another significant speedup occurs between 64 total MPI ranks and 128 total MPI ranks. Additionally a significant speedup is noted from Fig. 12 between 128 total MPI ranks and 256 (fixed at 2 MPI ranks per compute node) total MPI ranks, but Fig. 12 then shows that the parallelized Huffman algorithm then experiences a significant slowdown, in terms of encode speed, from 256-512 (fixed at 4 MPI ranks per compute node) total MPI ranks. This slowdown trend continues until we increase the fixed MPI ranks per compute node to 8. Summed up the results are mixed:

- Fig. 10: Fixed at 2 MPI ranks: encode execution time tends to increase as total MPI ranks increase
- Fig. 12: Fixed at 4 MPI ranks: encode execution time tends to increase as total MPI ranks increase
- Fig. 14: Fixed at 8 MPI ranks: encode execution time tends to decrease as total MPI ranks increase
- Fig. 16: Fixed at 16 MPI ranks: encode execution time tends to increase as total MPI ranks increase
- Fig. 18: Fixed at 32 MPI ranks: encode execution time tends to level off as total MPI ranks increase
- Fig. 20: Fixed at 64 MPI ranks: encode execution time tends to decrease as total MPI ranks increase
- Fig. 22: Fixed at 128 MPI ranks: encode execution time tends to significantly increase as total MPI ranks increase
- Fig. 24: Fixed at 256 MPI ranks: encode execution time tends to increase as total MPI level of (slight increase) as total MPI ranks decrease

In regards to decode compute speedup. We measure the greatest average decode speedup between 512 and 1024 total MPI ranks. Here we measure a decode speedup of 1.8:
1.833508

$$Sd = \frac{0.004093}{0.002232} = 1.833508$$

The odd number figures above (Fig. 11 -25) give further insight on why the decode speedup here

is the greatest. In each graph the most negative straight-line curve occurs at the transition between 512 and 1024 total MPI ranks. Figure 23 also demonstrates significant decode speedup from about 2000 to about 4000 total MPI ranks, and from 4000 to about 8000 total MPI ranks. But, like the encode times mentioned above, decode time also varies significantly.

We now turn attention to Fig. 2- 9. These figures illustrate the execution times for both encode and decode operations at fixed ranks (2-256) and varying compute nodes (1-128). From these figures we notice:

- Fig. 2: Fixed at 2 MPI ranks has the fastest encode execution time at 16 and 64 compute nodes
- Fig. 3: Fixed at 4 MPI ranks has the fastest encode execution time at 64 compute nodes
- Fig. 4: Fixed at 8 MPI ranks has the fastest encode execution time at 120 compute nodes
- Fig. 5: Fixed at 16 MPI ranks has the fastest encode execution time at around 64 compute nodes
- Fig. 6: Fixed at 32 MPI ranks has the fastest encode execution time at around 16 compute nodes
- Fig. 7: Fixed at 64 MPI ranks has the fastest encode execution time at around 120 compute nodes
- Fig. 8: Fixed at 128 MPI ranks has the fastest encode execution time at 32 compute nodes
- Fig. 9 Fixed at 256 MPI has the fastest encode execution time at 16 and 64 compute nodes

Decode times more or less remain constant as compute nodes increase.

CONCLUSION

The Huffman lossless data compression algorithm does benefit from a parallelized computational approach. In general, both encode and decode times experience a significant speedup as total MPI ranks increase. In our testing, the fastest execution times for both encode and decode operations occurred at 128 compute nodes and 256 MPI ranks per compute node (for a total of 32,678 total MPI ranks). Additionally, our algorithm experience an encode speed up by a factor of 2.5 and a decode speedup of a factor of 111. Decode gets a significantly higher speedup since it does not include the overhead of reading in the plain text format and constructing the huffman encoding trees. Furthermore, the best configuration for optimal speed in regards to both encode and decode operations occurs with 64 compute nodes and 256 MPI ranks per compute node. At this configuration encode times experience a speedup of 2.49, while decode times experience a speedup of 112.22.

FUTURE WORK

Future work can include testing the parallelized Huffman algorithm on different sized inputs. Another interesting area to explore would be parallelization of other lossless data compression algorithms (such as the Lempel-Ziv compression algorithm). And with exploration of other compression algorithms, comes comparing said algorithms to our Huffman program.

WORKS CITED

Gyaikhom. “Gyaikhom/Huffman.” *GitHub*, 14 May 2016, github.com/gyaikhom/huffman.

Kavousianos, Xrysovalantis, et al. “Optimal Selective Huffman Coding for Test-Data Compression.” *IEEE Transactions on Computers*, vol. 56, no. 8, 2 July 2007, pp. 1146–1152., doi:10.1109/tc.2007.1057.

Klein, S. T. “Parallel Huffman Decoding with Applications to JPEG Files.” *The Computer Journal*, vol. 46, no. 5, 11 Feb. 2003, pp. 487–497., doi:10.1093/comjnl/46.5.487.

Knuth, Donald E. “Dynamic Huffman Coding.” *Journal of Algorithms*, vol. 6, no. 2, 2 Dec. 1985, pp. 163–180., doi:10.1016/0196-6774(85)90036-7.

Milne, Andrew, et al. *PARALLEL APPARATUS FOR HIGH-SPEED, HIGHILY COMPRESSED LZ77 TOKENIZATION AND HUFFMAN ENCODING FOR DEFLATE COMPRESSION*. 1 July 2014.

Nourani, Mehrdad, and Mohammad H. Tehranipour. “RL-Huffman Encoding for Test Compression and Power Reduction in Scan Applications.” *ACM Transactions on Design*

Automation of Electronic Systems, vol. 10, no. 1, 1 Jan. 2005, pp. 91–115., doi:10.1145/1044111.1044117.

Sharma, Mamta. “Compression Using Huffman Coding.” *International Journal of Computer Science and Network Security*, vol. 10, no. 5, 5 May 2010.

Stabno, Michał, and Robert Wrembel. “RLH: Bitmap Compression Technique Based on Run-Length and Huffman Encoding.” *Information Systems*, vol. 34, no. 4-5, 24 Dec. 2008, pp. 400–414., doi:10.1016/j.is.2008.11.002.

CONTRIBUTIONS

The researching group consisted of the following:

- Eleanor Little, head Programmer
- Frank Vega-Aguirre, head Tester
- Ian Schmidt, head Researcher.

The aforementioned roles are generalized, and each member had additional contributions outside of said roles.