Discussion

Authors:
Hinal Bharatbhai Arya
Saeid Rezaei
David Moreno-Bautista

Date: 2024-04-06

Intro

With the exponential growth of data in various domains, efficient data compression techniques are essential for storage, transmission, and processing. Parallel data compression offers the potential for significant speedup and improved throughput by distributing the compression workload across multiple processing units. There's potential for an even bigger speedup improvement by distributing the workload across multiple threads within the processing units. We analyze the performance differences between a serial approach, a parallel-process MPI approach, a parallel-thread OpenMP approach, and a combined MPI plus OpenMP approach.

The Huffman Way of Compression

First, let us define the main compression type we are working with. When it comes to text data, a way to significantly reduce the number of bytes that the file holds, while still maintaining its data, is to encode each character as their Huffman code. Huffman codes are binary character codes specifically calculated from the frequency of each character in the file; a very frequent character is shorter than a less frequent character. This type of encoding is called variable-length encoding.

Huffman created a variable-length encoding algorithm, where characters in a priority queue are extracted in pairs, from the less frequent, to the most frequent. Each pair of characters extracted is connected by a parent node that holds the sum of their frequencies as its value in a tree. The left and right child is either represented by a 1 or a 0 in binary. The nodes are inserted back into the queue, and the process continues until all characters are accounted for. In this way, the most frequent characters are defined by the least number of 1s and 0s, as they are closer to the top, while the least common characters are composed of more 1s and 0s from traversing the tree.

The main process of encoding the entire file is to walk through each character to find the frequencies, perform the Huffman algorithm to build the tree to reference from for the Huffman codes, traverse the tree to calculate each Huffman code for each character, and walk through each character in the text one final time to encode the Huffman code as part of a smaller, compressed binary file.

With this encoding finished, the decoding of the produced binary file can be done by having a serialized form of the Huffman tree outputted from the encoding process; and walking through each of the bits, traversing the deserialized tree to find each of the characters represented by the bit stream, and outputting each of the characters into the decoded text file.

As we can already tell, this whole process has the potential for a speed-up from parallelization, due to the way that different sections of the text can be processed in parallel, since the same algorithm is applied without any dependencies on each different section. Furthermore, the main loops and tree traversals hold the potential for multi-threading due to no dependencies between each iteration.

For more information on Huffman, please refer to the wikipedia page: https://en.wikipedia.org/wiki/Huffman_coding

PART 1 - The Serial Approach

Testing Serial Encode and Serial Decode

- Navigate to the serial folder within the project.
- To test the encode and decode algorithms, submit the test.sh file to the scheduler in mcs2.
- You can see what the output of this script was during our testing in the analysis.out file.

Serial Encode

To compile and run this program, use the following commands:

```
g++ -std=c++11 encode_serial.cpp -o encode_serial
./encode_serial ./input.txt ./output.bin ./huffman_tree.txt

# input.txt -> The input plain text
# output.bin -> The output encoded binary file
# huffman_tree.txt -> The output tree file used in the decode
```

Data Structures and Functions:

- Node structure: Represents a node in the Huffman tree. Each node has two main parts: the character information and its frequency, and the left and right pointers for their children.
- compare function: Compares the frequency of two nodes in a queue, where the less frequent node takes a higher priority in the processing line.
- buildHuffmanTree function: Creates and returns a Huffman tree's root using a map of the frequency of characters.
- serializeHuffmanTree function: Traverses the tree and stores its representation in a file.
- assignHuffmanCodes function: Traverses the tree to determine the corresponding Huffman code for each character and stores them in a map.
- encodeText function: Encodes the input text according to each character's corresponding Huffman code and writes these bits to the binary file.
- main function: Requires three input arguments from the command line to run: the input file name, the compressed output file name, and the serialized Huffman tree file name. The overall step by step process is then followed.

Step by step process:

- The input.txt text file is read in as a stream.
- **LOOP #1**: N characters are processed to map each unique character to its frequency.
- **LOOP #2**: n=26+ characters and their frequencies are processed using the Huffman algorithm to build a Huffman tree in O(n * log(n)) time.
- **TREE TRAVERSAL #1**: The built tree is traversed fully and serialized into the huffman_tree.txt file in O(n) time.
- **TREE TRAVERSAL #2**: The built tree is traversed fully and the Huffman code is calculated for each of the characters in an overall O(n) time complexity.
- **LOOP #3**: N characters are encoded in binary format to the output.bin file according to their Huffman code.

Serial Decode

To compile and run this program, use the following commands:

```
g++ -std=c++11 decode_serial.cpp -o decode_serial
./decode_serial ./output.bin ./huffman_tree.txt plain.txt

# output.bin -> The input encoded binary file
# huffman_tree.txt -> The input tree file generated by encode
# plain.txt -> The output decoded plain text file that should equal the original
```

Data Structures and Functions:

- Node structure: Represents a node in the Huffman tree. Each node has two main parts: the character information and the left and right pointers for their children.
- deserializeHuffmanTree function: Parses the serialized Huffman tree file and traverses the tree while building the structure used in the program. Returns the root of the tree.
- decodeText function: Decodes the input binary string according to each bits's corresponding Huffman code and returns the final string.
- decodeBinaryData: Converts the binary file into a string of 1s and 0s, calls the decodeText function, and returns its result.
- main function: Requires three input arguments from the command line to run: the input encoded file name, the input serialized Huffman tree file name, and the output decompressed text file name. The overall step by step process is then followed.

Step by step process:

- The input tree file is read in as a stream.
- **TREE TRAVERSAL #1**: The huffman_tree.txt tree file is read in and deserialized into the Huffman tree in O(n) time, where n is 26+.
- The input output.bin encoded file is read in as a stream.
- **LOOP #1**: N character codes are processed and converted from bytes to strings.
- **LOOP #2 + TREE TRAVERSAL #2**: N character code strings are processed and converted from code to their character representation in an overall O(N*n) time complexity.
- The decoded text is written to the plain.txt file as a stream.

Analysis

- After submitting this program to the scheduler in the mcs2 server with various values of N using the test.sh script, we came to the conclusion that it takes around 90,346,176 characters for the input file to slow down the serial encode algorithm to beyond 30 seconds. The resultant 53,627,564 byte encoded file takes around 15 seconds to decode.
- The following is the precise results:

```
>>>> Serial Algorithm Time Analysis >>>>
Compression completed successfully.
N = 86201856 ./input.txt -> Encode seconds run = 29
Decoding completed successfully. Decoded text saved to: plain.txt
N = 51167584 ./output.bin -> Decode seconds run = 14
--
Compression completed successfully.
```

```
N = 90346176 ./input.txt -> Encode seconds run = 31
Decoding completed successfully. Decoded text saved to: plain.txt
N = 53627564 ./output.bin -> Decode seconds run = 15
--
Compression completed successfully.
N = 94490496 ./input.txt -> Encode seconds run = 32
Decoding completed successfully. Decoded text saved to: plain.txt
N = 56087544 ./output.bin -> Decode seconds run = 16
--
Compression completed successfully.
N = 98634816 ./input.txt -> Encode seconds run = 33
Decoding completed successfully. Decoded text saved to: plain.txt
N = 58547524 ./output.bin -> Decode seconds run = 17
--
Compression completed successfully.
N = 102779136 ./input.txt -> Encode seconds run = 35
Decoding completed successfully. Decoded text saved to: plain.txt
N = 61007504 ./output.bin -> Decode seconds run = 17
--
Compression completed successfully.
N = 106923456 ./input.txt -> Encode seconds run = 36
Decoding completed successfully. Decoded text saved to: plain.txt
N = 63467484 ./output.bin -> Decode seconds run = 18
--
```

PART 2 - The MPI Approach

Testing MPI Encode and MPI Decode

- Navigate to the mpi folder within the project.
- To test the encode and decode algorithms, submit the test.sh file to the scheduler in mcs2.
- You can see what the output of this script was during our testing in the analysis.out file.

MPI Encode

To compile and run this program, use the following commands:

```
mpic++ -std=c++11 encode_mpi.cpp -o encode_mpi
mpirun -np 40 ./encode_mpi ./input.txt huffman_tree.txt output.bin

# input.txt -> The input plain text
# output.bin -> The output encoded binary file
# huffman_tree.txt -> The output tree file used in the decode
```

Data Structures and Functions:

- Node structure: Represents a node in the Huffman tree. Each node has two main parts: the character information and its frequency, and the left and right pointers for their children.
- compare function: Compares the frequency of two nodes in a queue, where the less frequent node takes a higher priority in the processing line.
- buildHuffmanTree function: Creates and returns a Huffman tree's root using a map of the frequency of characters.

- serializeHuffmanTree function: Traverses the tree and stores its representation in a file.
- assignHuffmanCodes function: Traverses the tree to determine the corresponding Huffman code for each character and stores them in a map.
- encodeText function: Encodes the input text according to each character's corresponding Huffman code and writes these bits to the binary file.
- *main function*: Requires three input arguments from the command line to run: the input file name, the compressed output file name, and the serialized Huffman tree file name. MPI_Init initializes the MPI environment. Multiple processes are initialized according to the number specified in the command line. MPI_Comm_rank assigns a process number to each process stored in the my_rank variable. MPI_Comm_size retrieves the total number of processes and stores it in the num_procs variable. The overall step by step process is then followed.

Step by step process:

- The input.txt text file is read in as a stream on *process 0*.
- The total size of the text is broadcasted to all processes using *MPI_Bcast*.
- The input text is divided into smaller sections, and each section is sent to each of the processes using *MPI_Scatterv*.
- **LOOP #1**: *N/p characters* are processed to map each unique character to its frequency by each process (where p is the number of processes).
- The final global result of the frequencies is collected by using *MPI_Reduce* to collect each of the individual sums into one.
- The rest is done by *process 0*:
- **LOOP #2**: n=26+ characters and their frequencies are processed using the Huffman algorithm to build a Huffman tree in O(n * log(n)) time.
- **TREE TRAVERSAL #1**: The built tree is traversed fully and serialized into the huffman_tree.txt file in O(n) time.
- **TREE TRAVERSAL #2**: The built tree is traversed fully and the Huffman code is calculated for each of the characters in an overall O(n) time complexity.
- **LOOP #3**: N characters are encoded in binary format to the output.bin file according to their Huffman code.
- *MPI_Finalize* terminates every process.

MPI Decode

To compile and run this program, use the following commands:

```
mpic++ -std=c++11 decode_mpi.cpp -o decode_mpi
mpirun -np 40 ./decode_mpi ./output.bin ./huffman_tree.txt plain.txt

# output.bin -> The input encoded binary file
# huffman_tree.txt -> The input tree file generated by encode
# plain.txt -> The output decoded plain text file that should equal the original
```

Data Structures and Functions:

- Node structure: Represents a node in the Huffman tree. Each node has two main parts: the character information and the left and right pointers for their children.
- deserializeHuffmanTree function: Parses the serialized Huffman tree file and traverses the tree while building the structure used in the program. Returns the root of the tree.
- decodeText function: Decodes the input binary string according to each bits's corresponding Huffman code and returns the final string.

- decodeBinaryData: Converts the binary file into a string of 1s and 0s, calls the decodeText function, and returns its result.
- *main function*: Requires three input arguments from the command line to run: the input encoded file name, the input serialized Huffman tree file name, and the output decompressed text file name. MPI_Init initializes the MPI environment. Multiple processes are initialized according to the number specified in the command line. MPI_Comm_rank assigns a process number to each process stored in the rank variable. MPI_Comm_size retrieves the total number of processes and stores it in the size variable. Time is measured using MPI_Wtime. The overall step by step process is then followed.

Step by step process:

- The input tree file is read in as a stream.
- **TREE TRAVERSAL #1**: The huffman_tree.txt tree file is read in and deserialized into the Huffman tree in O(n) time, where n is 26+.
- The input output.bin encoded file is read in as a stream.
- The total size of the file is gathered by all the processes using *MPI_Gather*.
- The encoded file is divided into smaller sections, and each section is gathered by each of the processes using *MPI_Gatherv*.
- **LOOP #1**: *N/p character codes* are processed and converted from bytes to strings.
- **LOOP #2 + TREE TRAVERSAL #2**: *N/p character code strings* are processed and converted from code to their character representation in an overall *O(N/p*n) time complexity*.
- The decoded text is written to the plain.txt file as a stream by *process 0*.

Analysis

- We have reduced the serial frequency-calculation traversal from N to N/p by spreading it between parallel processes in the MPI encode.
- We have also reduced the encoded file traversals from N to N/p by spreading them between parallel processes in the MPI decode.
- The encoding is the bottleneck for the Huffman process, so we focus on its results for this analysis:
- After submitting this program to the scheduler in the mcs2 server with various values of N using the test.sh script, we came to the conclusion that it takes around 98,634,816 characters for the input file to slow down the serial encode algorithm to beyond 30 seconds. The resultant 58,547,524 byte encoded file takes around 1 second to decode.
- The encode result is contrasted to the 90,346,176 characters that it takes the serial encode approach, which means that *8,288,640 more characters can be processed by the MPI encode in the same amount of time as the serial one*. Furthermore, the decode result is contrasted to the 17 seconds that it takes the serial decode approach on a smaller file, which is a *70% speedup by the MPI decode from the serial algorithm*.
- The following is the precise results:

```
>>>> MPI Algorithm Time Analysis >>>>
N = 86201856 ./input.txt -> Encode seconds run = 27
N = 51167584 ./output.bin -> Decode seconds run = 2
--
N = 90346176 ./input.txt -> Encode seconds run = 28
N = 53627564 ./output.bin -> Decode seconds run = 2
--
N = 94490496 ./input.txt -> Encode seconds run = 30
N = 56087544 ./output.bin -> Decode seconds run = 1
--
N = 98634816 ./input.txt -> Encode seconds run = 31
```

```
N = 58547524 ./output.bin -> Decode seconds run = 1
--
N = 102779136 ./input.txt -> Encode seconds run = 32
N = 61007504 ./output.bin -> Decode seconds run = 1
--
N = 106923456 ./input.txt -> Encode seconds run = 34
N = 63467484 ./output.bin -> Decode seconds run = 1
--
N = 111067776 ./input.txt -> Encode seconds run = 35
N = 65927464 ./output.bin -> Decode seconds run = 2
--
N = 115212096 ./input.txt -> Encode seconds run = 36
N = 68387444 ./output.bin -> Decode seconds run = 2
--
N = 119356416 ./input.txt -> Encode seconds run = 37
N = 70847424 ./output.bin -> Decode seconds run = 2
--
```

PART 3 - The OpenMP Approach

Testing OpenMP Encode and OpenMP Decode

- Navigate to the openmp folder within the project.
- To test the encode and decode algorithms, submit the test.sh file to the scheduler in mcs2.
- You can see what the output of this script was during our testing in the analysis.out file.

OpenMP Encode

To compile and run this program, use the following commands:

```
g++ -std=c++11 -fopenmp encode_openmp.cpp -o encode_openmp
./encode_openmp ./input.txt ./output.bin ./huffman_tree.txt

# input.txt -> The input plain text
# output.bin -> The output encoded binary file
# huffman_tree.txt -> The output tree file used in the decode
```

Data Structures and Functions:

- Node structure: Represents a node in the Huffman tree. Each node has two main parts: the character information and its frequency, and the left and right pointers for their children.
- compare function: Compares the frequency of two nodes in a queue, where the less frequent node takes a higher priority in the processing line.
- buildHuffmanTree function: Creates and returns a Huffman tree's root using a map of the frequency of characters.
- serializeHuffmanTree function: Traverses the tree and stores its representation in a file.
- assignHuffmanCodes function: Traverses the tree to determine the corresponding Huffman code for each character and stores them in a map. Using the *OpenMP sections* library, which is used to divide the work between several threads, different parts of the tree are processed simultaneously.
- encodeText function: Encodes the input text according to each character's corresponding Huffman code and writes these bits to the binary file. The text is walked through using an *OpenMP parallel for loop*, which splits up the work between each thread operating simultaneously.

- main function: Requires three input arguments from the command line to run: the input file name, the compressed output file name, and the serialized Huffman tree file name. The overall step by step process is then followed.

Step by step process:

- The input.txt text file is read in as a stream.
- **LOOP #1**: N characters are processed to map each unique character to its frequency.
- **LOOP #2**: n=26+ characters and their frequencies are processed using the Huffman algorithm to build a Huffman tree in O(n * log(n)) time.
- **TREE TRAVERSAL #1**: The built tree is traversed fully and serialized into the huffman_tree.txt file in O(n) time.
- **TREE TRAVERSAL #2**: The built tree is traversed fully and the Huffman code is calculated for each of the characters in an overall *O(log(n)) time complexity* due to the OpenMP sections.
- **LOOP #3**: *N/t characters* are encoded in binary format to the output.bin file according to their Huffman code (where t is the number of threads due to the OpenMP parallel for loop).

OpenMP Decode

NOTE: Our implementation for this part is done for demonstrative purposes only; the following theory is not reflective of the actual speed results seen during testing. In order to accommodate for the encoding analysis, the serial decode is used alongside it for the OpenMP analysis to fit the results within the available 5 minutes.

To compile and run this program, use the following commands:

```
g++ -std=c++11 -fopenmp decode_openmp.cpp -o decode_openmp
./decode_openmp ./output.bin ./huffman_tree.txt plain.txt

# output.bin -> The input encoded binary file
# huffman_tree.txt -> The input tree file generated by encode
# plain.txt -> The output decoded plain text file that should equal the original
```

Data Structures and Functions:

- Node structure: Represents a node in the Huffman tree. Each node has two main parts: the character information and the left and right pointers for their children.
- deserializeHuffmanTree function: Parses the serialized Huffman tree file and traverses the tree while building the structure used in the program. Returns the root of the tree.
- decodeText function: Decodes the input binary string according to each bits's corresponding Huffman code and returns the final string.
- decodeBinaryData: Converts the binary file into a string of 1s and 0s using an *OpenMP parallel for loop*, calls the decodeText function, and returns its result.
- main function: Requires three input arguments from the command line to run: the input encoded file name, the input serialized Huffman tree file name, and the output decompressed text file name. The overall step by step process is then followed.

Step by step process:

- The input tree file is read in as a stream.

- **TREE TRAVERSAL #1**: The huffman_tree.txt tree file is read in and deserialized into the Huffman tree in O(n) time, where n is 26+.
- The input output.bin encoded file is read in as a stream.
- **LOOP #1**: *N/t character codes* are processed and converted from bytes to strings.
- **LOOP #2 + TREE TRAVERSAL #2**: N character code strings are processed and converted from code to their character representation in an overall O(N*n) time complexity.
- The decoded text is written to the plain.txt file as a stream.

Analysis

- We have reduced one of the tree traversals in the OpenMP encode from O(n) to O(log(n)) due to parallel OpenMP sections, and we have reduced the final file encoding from N to N/t due to the parallel OpenMP for loop.
- The encoding is the bottleneck for the Huffman process, so we focus on its results for this analysis:
- After submitting this program to the scheduler in the mcs2 server with various values of N using the test.sh script, we came to the conclusion that it takes around 102,779,136 characters for the input file to slow down the serial encode algorithm to beyond 30 seconds.
- The encode result is contrasted to the 98,634,816 characters that it takes the MPI approach. This means that *4,144,320 more characters can be processed by the OpenMP encode in the same amount of time as the MPI*. This might be due in part to the default thread number allocated by OpenMP, which is optimized compared to the 40 processes we chose to use in MPI.
- The encode result is contrasted to the 90,346,176 characters that it takes the serial approach. This means that *12,432,960 more characters can be processed by the OpenMP encode in the same amount of time as the serial approach*.
- The following is the precise results:

```
>>>> OpenMP Algorithm Time Analysis >>>>
Compression completed successfully.
N = 90346176 ./input.txt -> Encode seconds run = 27
Decoding completed successfully. Decoded text saved to: plain.txt
N = 52356624 ./output.bin -> Decode seconds run = 15
--
Compression completed successfully.
N = 94490496 ./input.txt -> Encode seconds run = 28
Decoding completed successfully. Decoded text saved to: plain.txt
N = 56087544 ./output.bin -> Decode seconds run = 16
--
Compression completed successfully.
N = 98634816 ./input.txt -> Encode seconds run = 29
Decoding completed successfully. Decoded text saved to: plain.txt
N = 57141420 ./output.bin -> Decode seconds run = 17
--
Compression completed successfully.
N = 102779136 ./input.txt -> Encode seconds run = 31
Decoding completed successfully. Decoded text saved to: plain.txt
N = 61007504 ./output.bin -> Decode seconds run = 17
--
Compression completed successfully.
N = 106923456 ./input.txt -> Encode seconds run = 31
Decoding completed successfully. Decoded text saved to: plain.txt
N = 58455060 ./output.bin -> Decode seconds run = 17
--
Compression completed successfully.
```

```
N = 111067776 ./input.txt -> Encode seconds run = 33
Decoding completed successfully. Decoded text saved to: plain.txt
N = 65927464 ./output.bin -> Decode seconds run = 19
--
```

PART 4 - The MPI+OpenMP Approach

Testing MPI+OpenMP Encode and MPI+OpenMP Decode

- Navigate to the mpi-openmp folder within the project.
- To test the encode and decode algorithms, submit the test.sh file to the scheduler in mcs2.
- You can see what the output of this script was during our testing in the analysis.out file.

MPI+OpenMP Encode

To compile and run this program, use the following commands:

```
mpic++ -fopenmp -std=c++11 encode_mpi_openmp.cpp -o encode_mpi_openmp
mpirun -np 40 ./encode_mpi_openmp ./input.txt huffman_tree.txt output.bin

# input.txt -> The input plain text
# huffman_tree.txt -> The output tree file used in the decode
# output.bin -> The output encoded binary file
```

Data Structures and Functions:

- Node structure: Represents a node in the Huffman tree. Each node has two main parts: the character information and its frequency, and the left and right pointers for their children.
- compare function: Compares the frequency of two nodes in a queue, where the less frequent node takes a higher priority in the processing line.
- buildHuffmanTree function: Creates and returns a Huffman tree's root using a map of the frequency of characters.
- serializeHuffmanTree function: Traverses the tree and stores its representation in a file.
- assignHuffmanCodes function: Traverses the tree to determine the corresponding Huffman code for each character and stores them in a map. Using the *OpenMP sections* library, which is used to divide the work between several threads, different parts of the tree are processed simultaneously.
- encodeText function: Encodes the input text according to each character's corresponding Huffman code and writes these bits to the binary file. The text is walked through using an *OpenMP parallel for loop*, which splits up the work between each thread operating simultaneously.
- *main function*: Requires three input arguments from the command line to run: the input file name, the serialized Huffman tree file name, and the compressed output file name,. MPI_Init initializes the MPI environment. Multiple processes are initialized according to the number specified in the command line. MPI_Comm_rank assigns a process number to each process stored in the my_rank variable. MPI_Comm_size retrieves the total number of processes and stores it in the num_procs variable. The overall step by step process is then followed.

Step by step process:

- The input.txt text file is read in as a stream on *process 0*.
- The total size of the text is broadcasted to all processes using *MPI_Bcast*.

- The input text is divided into smaller sections, and each section is sent to each of the processes using *MPI_Scatterv*.
- **LOOP #1**: *N/p characters* are processed to map each unique character to its frequency by each process (where p is the number of processes).
- The final global result of the frequencies is collected by using *MPI_Reduce* to collect each of the individual sums into one.
- The rest is done by *process 0*:
- **LOOP #2**: n=26+ characters and their frequencies are processed using the Huffman algorithm to build a Huffman tree in O(n * log(n)) time.
- **TREE TRAVERSAL #1**: The built tree is traversed fully and serialized into the huffman_tree.txt file in O(n) time.
- **TREE TRAVERSAL #2**: The built tree is traversed fully and the Huffman code is calculated for each of the characters in an overall *O(log(n)) time complexity* due to the OpenMP sections.
- **LOOP #3**: *N/t characters* are encoded in binary format to the output.bin file according to their Huffman code (where t is the number of threads due to the OpenMP parallel for loop).
- *MPI_Finalize* terminates every process.

MPI+OpenMP Decode

NOTE: Our implementation for this part is done for demonstrative purposes only; the following theory is not reflective of the actual speed results seen during testing. In order to accommodate for the encoding analysis, the serial decode is used alongside it for the MPI+OpenMP analysis to fit the results within the available 5 minutes.

To compile and run this program, use the following commands:

```
mpic++ -fopenmp -std=c++11 decode_mpi_openmp.cpp -o decode_mpi_openmp
mpirun -np 40 ./decode_mpi_openmp ./output.bin ./huffman_tree.txt plain.txt

# output.bin -> The input encoded binary file
# huffman_tree.txt -> The input tree file generated by encode
# plain.txt -> The output decoded plain text file that should equal the original
```

Data Structures and Functions:

- Node structure: Represents a node in the Huffman tree. Each node has two main parts: the character information and the left and right pointers for their children.
- deserializeHuffmanTree function: Parses the serialized Huffman tree file and traverses the tree while building the structure used in the program. Returns the root of the tree.
- decodeText function: Decodes the input binary string according to each bits's corresponding Huffman code and returns the final string.
- decodeBinaryData: Converts the binary file into a string of 1s and 0s using an *OpenMP parallel for loop*, calls the decodeText function, and returns its result.
- *main function*: Requires three input arguments from the command line to run: the input encoded file name, the input serialized Huffman tree file name, and the output decompressed text file name. MPI_Init initializes the MPI environment. Multiple processes are initialized according to the number specified in the command line. MPI_Comm_rank assigns a process number to each process stored in the rank variable. MPI_Comm_size retrieves the total number of processes and stores it in the size variable. Time is measured using MPI_Wtime. The overall step by step process is then followed.

Step by step process:

- The input tree file is read in as a stream.
- **TREE TRAVERSAL #1**: The huffman_tree.txt tree file is read in and deserialized into the Huffman tree in O(n) time, where n is 26+.
- The input output.bin encoded file is read in as a stream.
- The total size of the file is gathered by all the processes using *MPI_Gather*.
- The encoded file is divided into smaller sections, and each section is gathered by each of the processes using *MPI_Gatherv*.
- **LOOP #1**: *N/p/t character codes* are processed and converted from bytes to strings.
- **LOOP #2 + TREE TRAVERSAL #2**: *N/p character code strings* are processed and converted from code to their character representation in an overall *O(N/p*n) time complexity*.
- The decoded text is written to the plain.txt file as a stream by *process 0*.

Analysis

- We have reduced the serial frequency-calculation traversal from N to N/p by spreading it between parallel processes in the MPI encode.
- Furthermore, we have reduced one of the tree traversals in the OpenMP encode from O(n) to O(log(n)) due to parallel OpenMP sections, and we have reduced the final file encoding from N to N/t due to the parallel OpenMP for loop.
- The encoding is the bottleneck for the Huffman process, so we focus on its results for this analysis:
- After submitting this program to the scheduler in the mcs2 server with various values of N using the test.sh script, we came to the conclusion that it takes around 119,356,416 characters for the input file to slow down the serial encode algorithm to beyond 30 seconds.
- The encode result is contrasted to the 102,779,136 characters that it takes the OpenMP approach. This means that *16,577,280 more characters can be processed by the MPI+OpenMP encode in the same amount of time as the OpenMP*.
- The encode result is contrasted to the 98,634,816 characters that it takes the MPI approach. This means that *20,721,600 more characters can be processed by the MPI+OpenMP approach in the same amount of time as the MPI*.
- The encode result is contrasted to the 90,346,176 characters that it takes the serial approach. This means that *29,010,240 more characters can be processed by the MPI+OpenMP approach in the same amount of time as the serial approach*.
- The following is the precise results:

```
>>>> MPI+OpenMP Algorithm Time Analysis >>>>
N = 102779136 ./input.txt -> Encode seconds run = 26
Decoding completed successfully. Decoded text saved to: plain.txt
N = 61007504 ./output.bin -> Decode seconds run = 18
--
N = 106923456 ./input.txt -> Encode seconds run = 28
Decoding completed successfully. Decoded text saved to: plain.txt
N = 63467484 ./output.bin -> Decode seconds run = 18
--
N = 111067776 ./input.txt -> Encode seconds run = 29
Decoding completed successfully. Decoded text saved to: plain.txt
N = 65804184 ./output.bin -> Decode seconds run = 19
--
N = 115212096 ./input.txt -> Encode seconds run = 30
Decoding completed successfully. Decoded text saved to: plain.txt
N = 68387444 ./output.bin -> Decode seconds run = 20
--
```

```
N = 119356416 ./input.txt -> Encode seconds run = 32
Decoding completed successfully. Decoded text saved to: plain.txt
N = 67022784 ./output.bin -> Decode seconds run = 19
--
N = 123500736 ./input.txt -> Encode seconds run = 32
Decoding completed successfully. Decoded text saved to: plain.txt
N = 73307404 ./output.bin -> Decode seconds run = 21
--
```

Final Comparison

> Encode:

Approach	Encode in 30+ Seconds (Chars)	Speedup from Serial (Percent)
Serial	90,346,176	0.00 %
MPI	98,634,816	9.17 %
OpenMP	102,779,136	13.76 %
MPI + OpenMP	119,356,416	32.11 %

> Decode:

Approach	Compressed Size (Bytes)	Decode Speed (Secs)
Serial	53,627,564	15
MPI	58,547,524	1
OpenMP	61,007,504	[To be explored...]
MPI + OpenMP	67,022,784	[To be explored...]