Malcolm Mashig (mjm2396)
COMS 4995: Parallel Functional Programming
Columbia University, Fall 2021

Project Proposal: Parallelizing Huffman Coding (AKA "ParHuff")

Huffman coding is an algorithm that can compress text into bits and conversely decompress bits back into text. This algorithm is deterministic and thus lossless. There are three different components of Huffman's algorithm: (1) creation of a code tree based on character likelihood, (2) the encoding of each character in the text via traversal of the tree, and (3) the decoding of bit sequences via traversal of the tree.

(1) To create the code tree, the empirical frequency or *a priori* probability of occurrence of each character is required as input (or must be computed). With this input, a min heap is generated with each character as a root node and their frequencies as weights. Thereafter, the two characters with the lowest weight are made children of a new node (lowest-weight node as left child) with a weight equal to the sum of their weights. This step is repeated until only one non-child node remains. This node is the root of the code tree.

(2) To encode each character in the text, the tree must be exhaustively traversed, but only once. While traversing the tree, the sequence of steps from the root node is tracked. Movement to a left child is noted with a zero and movement to a right child is noted with a one. Once a leaf node is reached, the character at the node is assigned to the bit sequence (of steps) tracked on the way there. Once all leaf nodes are reached from the root node, a map of each character to a bit sequence is known. Notably, no two characters' bit sequences will share a prefix, which is vital for deterministic decoding. Once the character-to-code map is known, the text can be simply converted to bits.

(3) To decode bits back into text, bits are read sequentially left-to-right as instructions for traversing the code tree from the root. As aforementioned, a zero signifies moving left down the tree and a one signifies moving right. Once a leaf node is reached, a character is obtained, and the next bit becomes the first instruction (restarting at the root) for decoding the next character. This process continues until all bits are converted to characters.

Parallelizing the algorithm *(by modifying Huffman codes):*

There are a few sequential haskell implementations of Huffman coding out there (including this). I may end up referencing and/or adapting this code to get started on the baseline sequential implementation.

Parallelizing (1) the creation of the code tree should be fairly straightforward. Instead of sequentially counting the occurrences of each character in the entire text, we can split up the text into batches and determine character frequencies within each batch, in parallel, then sum the frequencies of each character across the batches. I am unsure of how to parallelize the

iterative process of building a min heap from the character frequencies, or whether doing so will speed up the algorithm, but I will explore this.

Parallelizing (2) the encoding of characters should likewise be fairly straightforward. Traversing all root-to-leaf paths of the tree can be done in parallel by splitting up the tree and producing character-to-code maps for each subtree (prepending the code from root to subtree), then combining the maps. Additionally, like when determining character frequency, we can split up the text into equal-size batches and convert the characters into their respectives bit sequences, in parallel, then concatenate the resultant bits of each batch.

Parallelizing (3) the decoding of bits back into text is the challenge because if we split up the entire sequence of bits into batches, we will not know if the initial bits for a batch (other than the first) belong to a character's bit sequence at the end of the previous batch. For example, if one batch (batch 1) ends with "010" and the following batch (batch 2) begins with "100", it may be that "010" maps to "A" and "100" maps to "B", but perhaps instead "101" (split across the batches) maps to "C." Thus, there is confusion on how to decode the beginning of batch 2 if we have not already decoded the entirety of batch 1. To work around the challenge of parallelizing (3), I am planning on implementing two helper functions.

The first helper function will be responsible for modifying the code tree such that the codes (bit sequences) for all but one character *begin and end with a one*. It will also modify the code for the one character that does not begin and end with a one (let's call it ψ) to be all zeros. More specifically, ψ's code will be set to a sequence of zeros with a length one greater than the max-length internal sequence of consecutive zeros for any other character code. We should be able to determine and store the length of this max-length internal sequence of zeros while we create the original, unmodified code tree. If not, however, we can safely set ψ's code length to be one less than the depth of the modified code tree, but this sacrifices even more "space" and should worsen the compression ratio. The modifications made by this helper function will allow the second helper function (described below) to determine where to partition batches which can then be decoded in parallel.

The second helper function will take partition estimations as input which will essentially be indices of the entire bitstring at which to start looking for ψ's code: a certain number of consecutive zeros (unique to ψ). Once the codes are found throughout the entire bitstring, batches will be partitioned at those locations. This removes the aforementioned confusion on how to decode the beginning of batches, the main obstacle in the way of parallelizing the decoding phase. Once batches are safely partitioned, we can decode the bits back into text, in parallel, then concatenate the resultant characters of each batch.

I am not sure how to best choose ψ but there should be a tradeoff between space and time depending on whether a more frequent character (quicker for the second helper function to find) or less frequent character (fewer bits in the compression) serves as ψ. I am also unsure if the overhead of the helper functions will prevent substantial (or any) speedup from the parallelization on different input/text sizes. I will explore these areas of uncertainty.