Malcolm Mashig (mjm2396)
COMS 4995: Parallel Functional Programming
Columbia University, Fall 2021
12/22/2021

Final Report
ParHuff: Parallelizing Huffman Encoding & Decoding

Introduction:

Huffman Coding requires the implementation of three steps:

1) Create a code tree based on character frequency (within some text file).
    a) Generate a min heap with each character as a root node and their frequencies as weights.
    b) 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.
    c) Repeat the previous step until only one non-child node remains. This node is the root of the code tree.
2) Encode each character in the text via traversal of the code tree.
    a) Movement to a left child is noted with a zero and movement to a right child is noted with a one.
    b) Once a leaf node is reached, the character at the node is assigned to the bit sequence (of steps) tracked on the way there.
    c) Notably, no two characters' bit sequences will share a prefix, which is vital for deterministic decoding.
3) Decode the encoded bit sequence via traversal of the code tree.
    a) Bits are read sequentially left-to-right as instructions for traversing the code tree from the root. A zero signifies moving left down the tree and a one signifies moving right.
    b) 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.

The purpose of Huffman Coding is to optimize the compression of a file into a smaller file by giving smaller codes (bit sequences) to the more frequent characters in the file. The success of Huffman Coding is often measured by a compression ratio, which we will define as the number of bits in the full encoding divided by the number of bits required to store the original file.

Parallelizing this entire process seems straightforward, but there is a caveat. We can forego parallelization for the first step – creation of the code tree – for the sake of simplicity. To parallelize step two – encoding – we can split the input text into batches and encode them in parallel. To parallelize step three – decoding – we can split the encoded bit sequence into batches and decode them in parallel. This is where we run into the caveat, and the main challenge to parallelizing the Huffman Coding process.

The Caveat: how do we decide the indices at which to split the encoded bit sequence into batches?

The character codes generated during step one will vary in length. That said, splitting the encoded bit sequence into batches may risk splitting up the bits corresponding to individual characters across batches. If this happens, the decoding step will fail. Next, we discuss our implementation and how we worked around this specific issue.

Implementation:

We derive our implementation largely from Andrew Sackville-West's sequential implementation [1].

First, we read the text from a file. We calculate a map [(char, frequency)] of each character in the file and the number of times (frequency) it appears in the file. Using this map, we build the code (decoding) tree which stores the characters in a tree-like data structure. We use this tree to then build the encoding dictionary [(char, string)] which maps each character to their code (bit sequence) based on their location in the tree. We perform all of the above steps completely sequentially.

```
test <- readFile filepath
let freqlist = charFreq test
let tree = buildDecTree freqlist
let encDict = buildEncDict tree
```

Thereafter, we begin parallelized encoding. First, we split the input text into a specified number of batches, with each batch as a member in a list. Then we map our encoding function onto the list of batches using the `rdeepseq` parallelized evaluation strategy. This results in a list of bit sequences – the encodings of each respective batch of text – and so we concatenate them into one combined bit sequence.

```
let inputBatches = batchInput test n_batches
let encodings = Prelude.map (encode encDict) inputBatches `using` parList rdeepseq
let encoded = concat encodings
```

Next, we focus on saving the encoded text to file with all information required for lossless/deterministic decoding. This requires first obtaining the lengths of each batch's resulting encoded bit sequence. We then create a list of the cumulative lengths. This list will act as the indices at which to split the full encoding (into the specified number of batches) during decoding. These indices resolve the caveat mentioned above. We also extract lists of the individual characters and frequencies (respectively) from the original char-frequency map. This allows us to write a concatenation of all meta-information and the full encoding to file.

```
let lengths = Prelude.map length encodings
let inds = Prelude.map show (Prelude.take (n_batches - 1) (scanl1 (+) lengths))
let chars = freqListToChars freqlist
let freqs = Prelude.map show (freqListToFreqs freqlist)
let out = [(show n_batches)] ++ inds ++ [show (length chars)] ++ freqs ++ [chars] ++ [encoded]
writeParHuff out "test-ParHuff.txt"
```

The resulting file will look something like this:

```
4
838
1659
2474
7
3
2
2
7
1
1
1
 ',.ABH
11000011010011000011100011010100011 1
```

The "4" in the first line represents the number of batches. The three (4 - 1) following numbers represent the indices at which the encoding can be split into batches. The following number (7) represents the number of characters in the original text, which is followed by seven frequencies, which is followed by the seven characters with those respective frequencies. All lines after this represent the actual encoded bit sequence. Realistically, because the purpose of Huffman Coding is to be able to write the encoding to a (smaller) binary file, the meta-information above the actual encoding should ideally be written to a separate file that can be zipped with a binary file

that stores just the encoding. However, I did not focus on the irrelevant, logistical aspects of the process.

Next, we move to step three and we necessarily assume that the decoding is completely independent of the encoding. That is, the person decoding a compressed file might be different than the person who compressed the file, and in that case would not be aware of the code tree (or the batch indices) utilized during parallelized encoding.

First, we read the file written during encoding and extract the meta-information – the batch indices, the number of characters, the actual characters, the character frequencies, and the encoding (bit sequence). With this information, we rebuild the code tree. Then we split the encoding into batches according to the indices which ensure that we do not split the bit sequences corresponding to individual characters across batches. Lastly, like we did during encoding, we map our decoding function across the batches using the `rdeepseq` parallel evaluation strategy, and then we concatenate the list of results (the text produced for each batch).

```
dec_instr <- readParHuff "test-ParHuff.txt"
let decInds = getInds dec_instr
let decNChars = getNChars dec_instr
let decFreqs = getFreqs dec_instr decNChars
let decChars = getChars dec_instr decNChars
let decTree = buildDecTree (constructFreqList decChars (getFreqs dec_instr (length decChars)))
let decBits = getBits dec_instr
let decBatches = batchBits decBits decInds
let decodings = Prelude.map (decode decTree) decBatches `using` parList rdeepseq
let decoded = concat decodings
```

At this point, we have concluded the entire Huffman Coding process, and our concatenated decoding matches the original text. Thus, our parallelized compression is lossless.

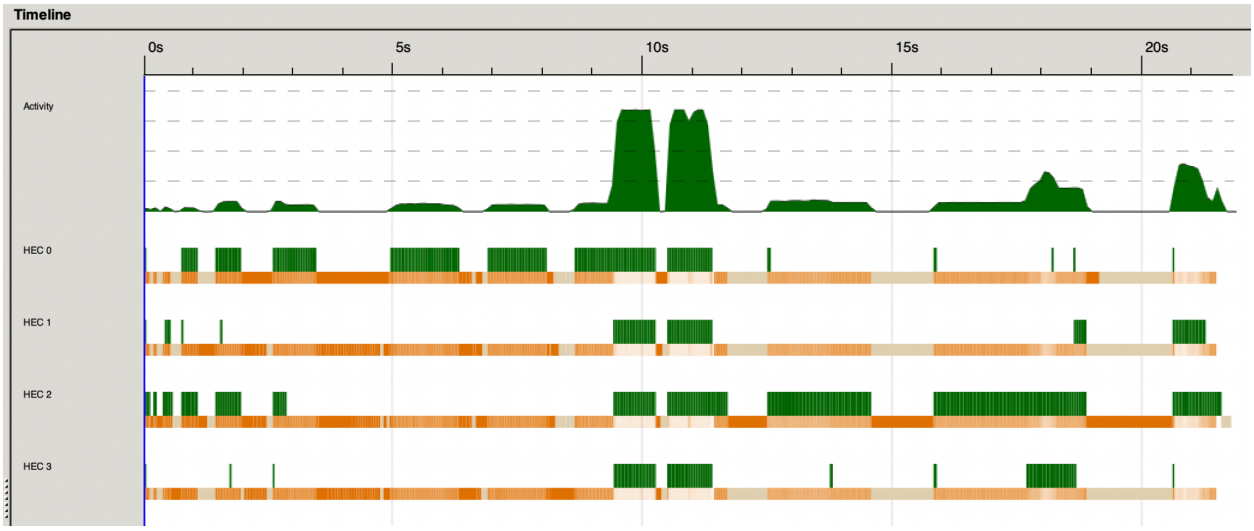
Results:

To test the processing time of our parallelized implementation, we ran all three steps – code tree creation, encoding, and decoding – on The Complete Works of William Shakespeare [2], which is over 5.5 million characters long. We obtain a ~0.60 compression ratio for this text. A summary of the processing times for various combinations of the number of batches and number of cores is provided below, with our best run in bold. The displayed times are the average of three runs for each combination.

Number of Batches	Number of Cores	Time (seconds)
1 (no parallelization)	1	20.789
1 (no parallelization)	2	17.469
1 (no parallelization)	3	19.086

1 (no parallelization)	4	21.626
4	1	21.264
4	**2**	**16.019**
4	3	18.231
4	4	21.690
100	1	26.387
100	2	19.439
100	3	21.968
100	4	25.274
1,000	1	25.781
1,000	2	18.348
1,000	3	22.511
1,000	4	22.001

The runtimes seem to indicate that the overhead of creating batches (once for encoding and once for decoding) for the sake of parallelization counteracts the benefits of parallelizing encoding and decoding. The fact that runtimes increase for large numbers of batches seems to support this. The minimal benefit from parallelization might also indicate that more steps in the Huffman Compression process need to be parallelized – such as the creation of the code tree – to observe higher gains. It is also odd that increasing the number of cores utilized tends to increase runtimes, and this is a sign that the program is not effectively spreading work across the cores/workers. The snapshot of Threadscope after running our program with four batches and four cores is shown below. This snapshot indicates that the program distributes work effectively only for a very brief interval of time, but not for any time outside of this. The program does carry out a number of expensive processes (reading, writing, reading again) without parallelization and so it is understandable that the distribution is not optimal throughout the entirety of the runtime. Better results were certainly expected however.



References

[1] http://lambduh.blogspot.com/2010/08/huffman-coding-in-haskell.html

[2] http://www.gutenberg.org/files/100/100-0.txt