15-853 Course Project

Generate Huffman Codes in parallel

Yichen Ruan (yichenr)

For code see GitHub: https://github.com/ycruan/ParallelHuffman

1. Introduction

In this project, we implement a parallel algorithm to generate Huffman codes for a set of symbols. The algorithm is based on the work proposed by Ostadzadeh et al. (2006).

In section 2, we describe the algorithm itself together with its performance. In section 3, we present some technique difficulties while implementing the algorithm and the corresponding solutions. The experiment setting and results are discussed in section 4. We summarize the project and address possible future work in section 5.

2. Algorithm

The algorithm can be divided into three phases: 1) CLGeneration: generate the length of codeword for each symbol; 2) Length Count: given a list of codeword length, find the indices of elements that have a different value compared to the next element; 3) CWGeneration: generate the final codewords for each symbol. An outline of the algorithm is shown in Fig. 1.

The algorithm requires O(n) work, where n is the number of symbols. The runtime is O(n) for the worst case, while for most of time it can be as low as O(log(logn - 1)!).

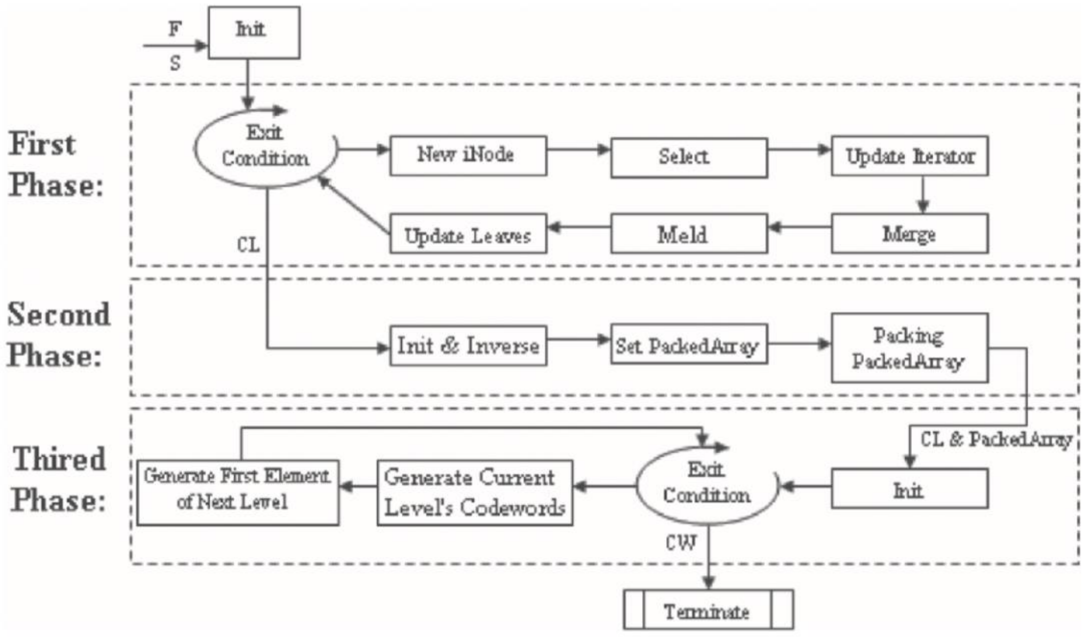

Fig 1. Algorithm outline

2.1. CLGeneration

- Input: an array of sorted frequencies
- Output: an array of codeword length for each symbol

This is the key part of this algorithm. As is shown in Fig. 1, this phase keeps repeating 6 steps until all the codeword length is written to the output array:

- New iNodes: pick 2 nodes (can be leave nodes or internal nodes) with the smallest frequencies, and merge them to create a new internal node (called iNode). The frequency of the new node is set as the sum of the 2 picked nodes;
- Select: find all the leave nodes whose frequency are less than that of the new iNode created in the previous step, copy them to a temporary array;
- Update Iterators and Merge: the leave nodes selected in the last step, together with the iNodes that have not yet participated in merging before, are merged to form a new array. However, the array has to have even number of elements. So before doing the merge, we need to compare these nodes and keep only an even number of them with the smallest frequencies;
- Meld: This step is to merge pair-wisely all consecutive nodes in the array generated in the "merge" step;
- Update Leaves: If some node merged in the "meld" step have children, all the leaves rooted at it are increased by one level. Thus, we need to update their codeword lengths by 1.

In the implementation, we do not maintain an explicit tree structure. Instead, we will keep 5 arrays: output array for codeword length, internal nodes, leave nodes, selected leaves in step "select", and merged leave and internal nodes in step "meld". Each time two nodes are merged, we need to update these arrays accordingly.

Easy to see this algorithm follows a bottom-up scheme. The correctness of this method follows immediately from that of the original Huffman tree algorithm. After each iteration, all nodes at or below the current level (i.e. nodes whose frequencies are no greater than the current maximum frequency) are already up-to-date. Thus, the number of iterations is bounded by the height of the Huffman tree. With precisely n processors, all operations described above can be done in O(1) time – except the "merge" operation (step 4) that have runtime O(loglogM(i)) (Kruskal, 1983), where M(i) is the number of nodes to be merged in iteration i. Throughout the procedure, precisely n-1 nodes are merged, i.e. $\sum_{i=1}^{L} M(i)$. Here L is the height of the underlying Huffman tree, which is O(n) if the tree is unbalanced, and O(logn) otherwise.

As a result, work $W = O(n)$

For one-side tree, $M(i) = O(1)$, depth $T = O(n)$

For balanced tree

$$T = \log\log 2 + \log\log 2^2 + \ldots + \log\log 2^{\log n} = O(\log(\log n - 1)!)$$

2.2. Length Count

- Input: an array of codeword length for each symbol
- Output: reversed codeword length array, sorted array indicating the indices whose codeword length are different with its successor.

As per the outline in Fig. 1, this part is comprised of 3 steps:

- Reverse: the codeword length generated in section 2.1 follows a bottom-top style, however, the final Huffman codes are assigned in a top-bottom fashion. We therefore need to first reverse the codeword length array. During the implementation, the array is reversed in place with n/2 processors in constant time.
- Set and pack output array: we assign a processor to each element of the reversed array, find all the indices where the elements have different value with its successor. Those indices correspond to different levels of the Huffman tree. The array is then packed in O(n) work and O(logn) (Ostadzadeh, 2005) and output to the next phase.

To summary, this phase requires O(n) work, and O(logn) time.

2.3. CWGeneration

- Input: reversed codeword length array, sorted array indicating the indices whose codeword length are different with its successor.
- Output: an array of actual codewords

The algorithm in this phase is based on the following formula (Hashemian. 1995):

$$C_{i+1} = (C_i + 1) \times 2^{cl_{i+1} - cl_i}$$

where C_i is the codeword for symbol i, cl_i is the length of C_i.

As is shown in Fig. 1, this phase iterates over 2 steps until all codewords are filled

- Generate codewords for the current level: given the current codeword (initialized with the first longest codeword), generate all codewords with the same length by adding the distance to that longest codeword in the reversed codeword array.
- Generate first codeword for the next level: find the symbol with one more codeword length (level) as per the indicating array from phase 2. Generate the codeword for this element using Hashemian's formula.

Finally, the codeword array needs to be reversed once again to recover to the original order. All the steps described above can be done in O(n) work and O(1) depth. The number of iteration is again bounded by the height of the Huffman tree.

To conclude, all three phases takes O(n) work. If the underlying Huffman tree is balanced, the depth is O(log(logn - 1)!) for "CLGeneration", O(logn) for "Length Count" and O(logn) for “CWGeneration”, thus is O(log(logn - 1)!) in total. However, for a one-side tree with O(n) height, the depth is O(n) for “CLGeneration”, O(logn) for “Length Count” and O(n) for “CWGeneration”, and O(n) in total.

The algorithm is guaranteed to generate some optimal Huffman codewords, but that may be different from the result of the traditional Huffman algorithm. For instance, given input frequencies [1,2,3,4,5,6,7], Fig. 2 and 3 show the generated Huffman tree for the traditional algorithm and the parallel algorithm used in this project.

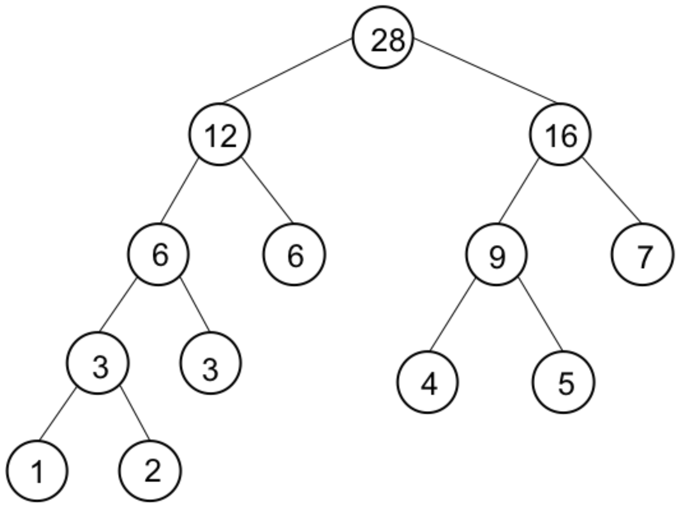

Fig 2. Huffman tree for traditional algorithm

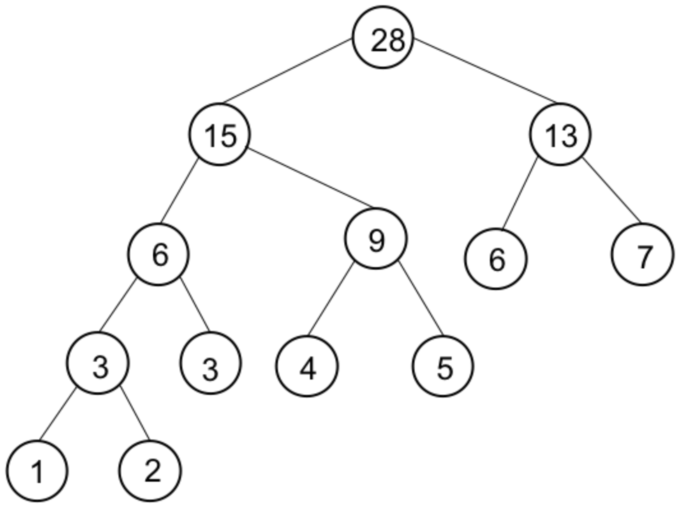

Fig 3. Huffman tree for parallel algorithm

3. Technical details

We implemented the algorithm using Java. Each processor is replaced by a thread object. Note that for all the three phases as shown in Fig. 1, the algorithm is parallelized only inside each sub-step. The main thread, however, should be synchronized. Thus, we need to force the main thread to wait for all sub-threads to finish before we can get into the next step. There are no race conditions – threads typically write to different slots of the array, and no data dependencies exist in the algorithm.

Nevertheless, there are some differences between our implementation and the algorithm described above:

1) String generation. In the CWGeneration phase, the algorithm does not distinguish string and binary integer. However, they are quite different in implementation. For example, the string "00" and "0" both indicate integer 0b0, but they are totally different as Huffman codes (although we know they can not appear at the same time for prefix codes). What is more, it is hard to distinguish the equations "00" + "1" = "01" and "0" + "1" = "1". To fix it, we treat all codewords as integers. And do the arithmetic calculations in section 2.3 using ordinary integer calculation. After all codewords are correctly computed, we convert the integers to binaries with length equals that in the CL array. The additional step can be done by n processors in O(1) work, thus does not affect the total runtime.
2) Pick new nodes: in the "New iNode" step of the first phase, we need to pick out two nodes with the smallest frequencies. We actually need two information: frequencies of the two nodes and whether they are leaves or internal nodes. We could, of course, copy 4 possible nodes (2 from leaves, 2 from internal nodes) into a temporary array, and sort that with respect to frequencies. To do this we will need each node to have an *isleave* filed indicating if it is a leave node. This idea is however inefficient since that filed is only used in this specific step. To fix it, we create two temporary length-2 arrays, one stores frequencies in a non-decreasing order, another indicates identities for each frequency element. For each iteration of the CLGeneration phase, we may need a little bit more O(1) operations to maintain the two arrays, but we are saving a lot of space and time to initialize and copy node structures.

4. Experiment

The experiment is done in the following physical setting:

Processor: 2 GHz Intel Core i5 (4 cores)

RAM: 8 GB 1867 MHz LPDDR3

OS: MacOS High Sierra 10.13.3

Java version: 1.8.0_111

Java VM version: 25.111-b14

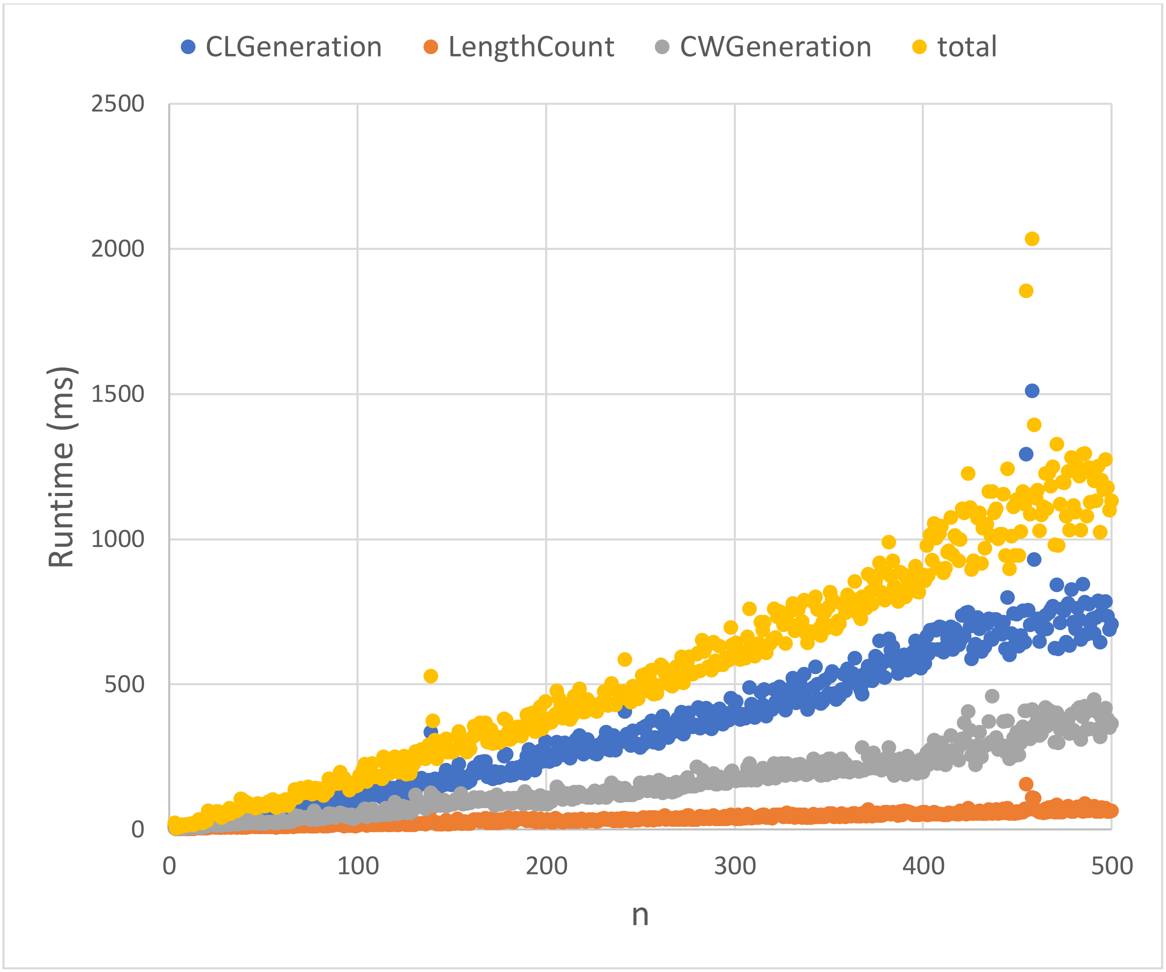

Fig 4. Runtime of the algorithm for different number of symbols

Fig. 4 shows the runtime for frequency set [1,2,3…,n], (n=2,…,500) which should yield a balanced Huffman tree. However, the runtime shown above is nearly linear, contradict with the theoretic result O(log(logn - 1)!). This may be due to the following reasons:

1) Finite physical threads: the algorithm requires n processors to run simultaneously. But we only have 4 physical threads available. What is more, the scheduling is controlled by both the Java VM and the OS, thus is hard to predict the behavior. Context switch can also be time consuming when there are hundreds of threads.
2) Booting latency: in reality, we cannot start all n threads at the same time. We typically have to boot them inside a for loop. Thus, some threads may start later than others.
3) Initialization overhead: Java automatically fills claimed spaces with zero, which is safe, but inefficient for performance purpose.
4) Java GC: which could be quite active considering we have so many thread objects.

5. Summary and Future work

In this project, we implement an algorithm that generate Huffman codewords in parallel. The algorithm accepts a sorted array for frequencies of symbols, and emits an array for the corresponding codewords. The algorithm has complexity of O(log(logn - 1)!) for a balanced Huffman tree, and O(n) in the worst case. It is also efficient in terms of space complexity, as it doesn't explicitly maintain a tree structure. Experiment result shows that the actual runtime is roughly O(n) in Java environment.

A possible future work is try to reduce the number of required processors. Currently the work is the order of O(n), which as discussed in section 4, has potential influence on the actual performance.

Reference

[1] Ostadzadeh, S. Arash, et al. "Parallel Construction of Huffman Codes." Advances in Systems, Computing Sciences and Software Engineering. Springer, Dordrecht, 2006. 17-23.
[2] Kruskal, Clyde P. "Searching, merging, and sorting in parallel computation." IEEE Transactions on Computers 10 (1983): 942-946.
[3] Ostadzadeh, S. Arash, M. Amir Moulavi, and Zeinab Zeinalpour. "Massive concurrent deletion of keys in b*-tree." International Conference on Parallel Processing and Applied Mathematics. Springer, Berlin, Heidelberg, 2005.