Lab 7.1 - Parallelizing techniques

December 11, 2018

1 Goal

The goal of this lab is to implement a simple but non-trivial parallel algorithm.

2 Requirement

Given a sequence of n numbers, compute the sums of the first k numbers, for each k between 1 and n. Parallelize the computations, to optimize for low latency on a large number of processors. Use at most 2*n additions, but no more than 2*log(n) additions on each computation path from inputs to an output. Example: if the input sequence is 1 5 2 4, then the output should be 1 6 8 12.

3 Computer Specification

- *CPU:* Intel Core i7-7500U, 2.90GHz
- *RAM:* 8 GB
- *System type:* 64-bit

4 Short Description of the Implementation

Algorithms used in order to check the performance:

- *regular linear algorithm*
- *binary tree algorithm*

4.1 Regular linear algorithm

Complexity: O(n) time
Iterate through the input list, and for each step result[i] = input[i] + result[i-1]

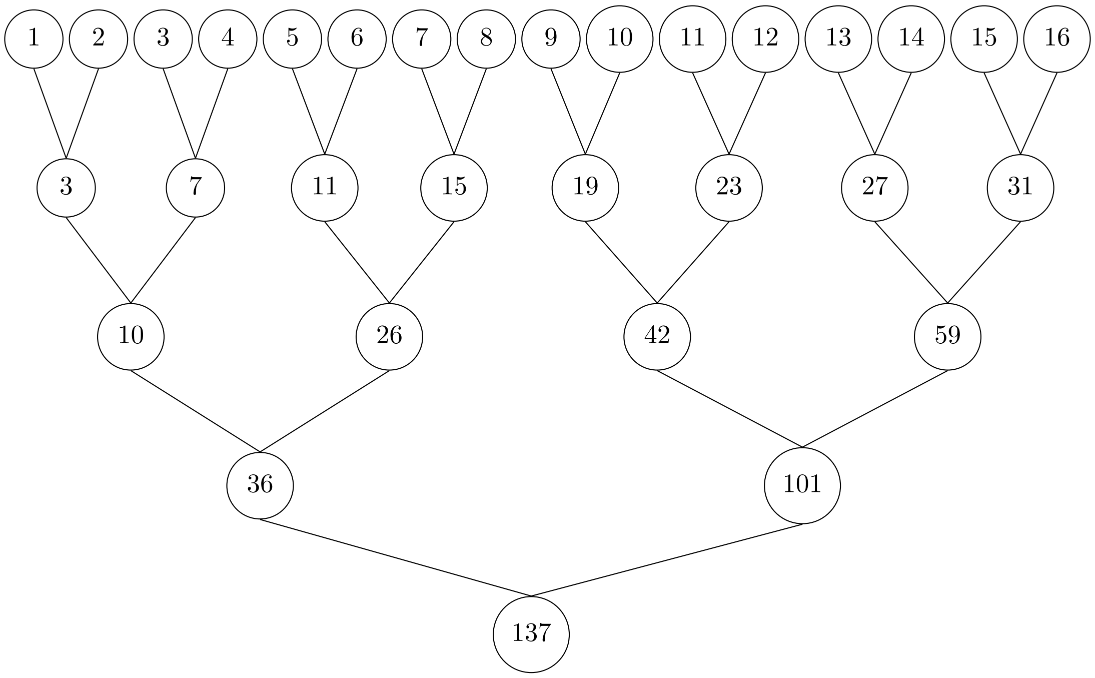
4.2 Binary tree algorithm

Complexity: O(n) time

The algorithm consists in 2 steps, in the first one we generate partial sums of length 2^j, where j is between (1, $log_2 N$) so as $1 < 2^j <= N$. In the second step we perform bottom up calculations of the remaining non-calculated values. We take the maximum j obtained in the first step and bringing it back to 1, on each level finding the elements that are at distance 2^{j-1} from a value that was calculated at step 1.

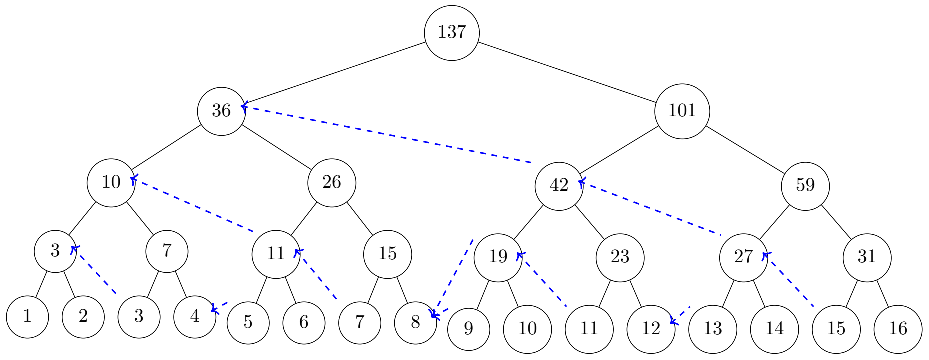
Let's take an example:
Let the sequence be: { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}

4.2.1 Step 1



As we can observe we obtain for each level the sum of the corresponding 2^n number in the sequence in logarithmic time.

4.2.2 Step 2



What happens here is that we go bottom up and we combine the elements that were previously computed with the ones that were only partially computed or not at all. The elements that we want to compute on each level (from 4 to 1) are the ones that are at 2^{j-1} distance from the computed elements. The elements we can be sure after completed at each level j are multiples of 2^j, so k * 2^j, where k is between 1 and whatever number gets us out of range;

For a better understanding of this step let's do it level by level:

- level j = 3, we know that the completed element is only 8 (1 * 2^3), as already (2 * 2^3) goes out of bounds. We look 2^{j-1} positions to his right, there's element 12, with current value of 42, so we combine that 42 with the value in 8, 36, and we get 78, which will replace the current value of element 12.
- At level j = 2, the completed elements are 4, 8, 12 aka (1 * 2^2), (2 * 2^2), (2 * 2^2). We look 2^{j-1} positions at their right and we update their current value as we did in level 3. Beware that the value of element 12 is now 78. We always take the highest value from the branch (which has the lowest position)
- At level j = 1, the completed elements will be all elements on an even position, and now the distance is 1 (1 * 2^0), so we can compute all the elements on odd positions.

5 Performance Tests

note: by level 'x' i am referring that the algorithms were used to generate prefixes for a list of size (100 * x) containing elements from 100 to 900.

Algorithm	Level 1	Level 50	Level 100
regular linear algorithm	0ms	0ms	0ms
binary tree algorithm sequential	13ms	14ms	8ms
binary tree algorithm parallel	5ms	16ms	12ms

Throughout the tests I've put those algorithm through, the results were inconclusive. The binary tree implementation might be optimized to yield better results in the parallel form than in the sequential form, however, in this instance I was not able to get a better performance out of it. The parallel version is on average worse than the sequential version.

6 Conclusion

Although in theory, the binary tree algorithm may be faster than the naive one, due to technicalities of the language and the overhead added by parallelism, this could simply not be worth it.