CS5234: Combinatorial and Graph Algorithms Lecture 11

Tutorial: Week 11

Lecturer: Seth Gilbert *November 2, 2018*

Array Sum

We have already seen in class how to compute the sum of the elements in an array using a fork-join parallel program. The basic idea is to keep dividing the array into two halves and recursively summing the two pieces. (To initiate the following algorithm, call $\text{SUM}(A, 1, n)$).

Algorithm 1: $\text{SUM}(A, begin, end)$

```
1 if (begin = end) then
2     return A[begin]
3 else
4     mid = (begin + end)/2
5     in parallel
6         (1) s_1 = SUM(A, begin, mid)
7         (2) s_2 = SUM(A, mid + 1, end)
8     return s_1 + s_2
```

The algorithm has $O(n)$ work and $O(\log n)$ span.

Prefix-Sum, Version One

The goal of prefix-sums is to compute, for each i, the sum of the prefix, i.e., $A[i] = \sum_{j=1}^{i} A[j]$. Throughout we will assume that $\log n$ is an integer. If not, you have to add appropriate floors and ceilings everywhere!

To solve this problem, we are first going to compute the sum for each subtree. This is essentially identical to the algorithm above for the sum, but now for each node in the recursion tree, we are going to save the sum of the left and right children.

To store this information, we are going to use a $\log n \times n \times 2$ sized 3-dimensional array S. (Yes, this wastes a bunch of space. Later we will see how to use less space.) If we have recursively computed the left and right sums for the range $[begin, end]$ at level l of the recursion, then we store this information at $S[level, end, left]$ and $S[level, end, right]$. That is, we use the cell in S associated with the last entry in the range. (It does not matter which cell in the range we use, as long as we are consistent.)

We initiate the algorithm by calling $\text{PREFIXSUMPARTONE}(A, \log n, 1, n)$.

Algorithm 2: PREFIXSUMPARTONE$(A, level, begin, end)$

```
1  if (begin = end) then
2      return A[begin]
3  else
4      mid = (begin + end)/2
5      in parallel
6          (1) s_1 = PREFIXSUMPARTONE(A, level − 1, begin, mid)
7          (2) s_2 = PREFIXSUMPARTONE(A, level − 1, mid + 1, end)
8      S[level, end, left] = s_1
9      S[level, end, right] = s_2
10     return s_1 + s_2
```

Once we have computed all the sums for all the subtrees, we can now (in parallel) walk the tree from the root to each leaf, accumulating the proper prefix-sum for each leaf. Imagine walking from the root to some leaf node representing $A[j]$. We start with $sum = 0$. Every time we walk left on the recursion tree (i.e., recursing on the left half of the array), we leave the sum unchanged. Every time we walk right on the recursion tree (i.e., recursing on the right half of the array), we add the sum of the left subtree to sum. By the time we get to a leaf, all we need to do is add the leaf value and we are done.

We initiate the algorithm by calling PREFIXSUMPARTTWO$(A, \log n, 0, 1, n)$.

Algorithm 3: PREFIXSUMPARTTWO$(A, level, sum, begin, end)$

```
1  if (begin = end) then
2      A[begin] = sum + A[begin]
3  else
4      mid = (begin + end)/2
5      in parallel
6          (1) PREFIXSUMPARTTWO(A, level − 1, sum, begin, mid)
7          (2) PREFIXSUMPARTTWO(A, level − 1, sum + S[level, end, left], mid + 1, end)
8      return
```

More Space Efficient Prefix Sums

The algorithm above wastes a lot of space. In fact, we solve the problem completely in-place, using only the initial array A. The key observation is that we can always store the sum of a subsequence of an array in the final slot of that subsequence. For example, if we have just computed the sum of $A[1, n/2]$, then we can store the sum in $A[n/2]$. As the recursion unrolls, a given cell in the array may be overwritten many times. When it is done, the value of a cell $A[j]$ in the array is set to the sum for the biggest subtree for which it is a maximal entry. Notice that it is okay that we have overwritten some of the value: we will never need those values! (Why is that? Prove it!)

Algorithm 4: PREFIXSUMPARTONE$(A, level, begin, end)$

```
1 if (begin = end) then
2     return
3 else
4     mid = (begin + end)/2
5     in parallel
6         (1) PREFIXSUMPARTONE(A, level − 1, begin, mid)
7         (2) PREFIXSUMPARTONE(A, level − 1, mid + 1, end)
8     A[end] = A[mid] + A[end]
```

Then, once we have the sums stored properly, we can do as before, walking through the tree and computing the final sums. However, we have to be a little more careful in this case not to double-count. Notice that when we get to a leaf, if the last step is "to the left," then the leaf is storing the real initial value of that array entry and we could safely add it. However, if the last step is "to the right," then the leaf is storing the sum of the array of entries of its subtree. Hence we had better not add that entry. Thus, in the very last step, we do not add the array entry in either case, simply storing the sum up to that point.

As a result, the following computes for each j the following: $\sum_{i=1}^{j-1} A[i]$, i.e. the prefix-sum for $A[j]$ not including the element itself.

Algorithm 5: PREFIXSUMPARTTWO$(A, level, sum, begin, end)$

```
1 if (begin = end) then
2     A[begin] = sum
3 else
4     mid = (begin + end)/2
5     add = A[mid]
6     in parallel
7         (1) PREFIXSUMPARTTWO(A, level − 1, sum, begin, mid)
8         (2) PREFIXSUMPARTTWO(A, level − 1, sum + add, mid + 1, end)
9     return
```

Finally, we need to fix the prefix-sums. We shift everything over one step to the left. (We will also, at the same time, fix the last entry of the array—see below.)

Algorithm 6: SHIFT$(A, right, begin, end)$

```
1 if (begin = end) then
2     A[begin] = right
3 else
4     mid = (begin + end)/2
5     leftRight = A[mid + 1]
6     in parallel
7         (1) SHIFT(A, leftRight, begin, mid)
8         (2) SHIFT(A, right, mid + 1, end)
9     return
```

Now, we can put it all together to get the complete algorithm. First we execution PREFIXSUMPARTONE, calculating all the sums. We then save the total sum of the entire array, which is currently stored in $A[n]$. Finally, we shift everything over. When we execute shift, we pass in the $total$ as the value for $right$, which sets the last value of the

array to be $total$, as it should be.

Algorithm 7: PrefixSum(A)

```
1 PREFIXSUMPARTONE(A, log n, 1, n)
2 total = A[n]
3 PREFIXSUMPARTTWO(A, log n, 0, 1, n)
4 SHIFT(A, total, 1, n) return
```

We will leave as an exercise developing the proper invariants and proving that this is correct (or finding the bugs therein).