CME 323: Distributed Algorithms and Optimization, Spring 2020
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matriod and Stanford.

Lecture 3, 04/7/2020. Scribed by Robin Brown.

3 All Prefix Sum

Given a list of integers, we want to find the sum of all prefixes of the list, i.e. the running sum. We are given an input array A of size n elements long. Our output is of size $n+1$ elements long, and its first entry is *always* zero. As an example, suppose $A = [3, 5, 3, 1, 6]$, then $R = \texttt{AllPrefixSum}(A) = [0, 3, 8, 11, 12, 18]$.

3.1 Algorithm Design

In sequential world, this is trivial. How can we parallelize this problem? We need a mix of divide and conquer and our first parallel summation algorithm. We design the following algorithm.

Algorithm 1: Prefix Sum

Input: All prefix sum for an array A
if *size of A is 1* **then**
  return *only element of A*
end
Let A' be the sum of adjacent pairs
Compute $R' = \text{AllPrefixSum}(A')$ `// Note:` R' `has every other element of` R
Fill in missing entries of R' using another $\frac{n}{2}$ processors

A note on the size of our (sub)-problems The general idea is that we first take the sums of adjacent pairs of A. So the size of A' is exactly half the size of A. Note that if the size of A not a power of 2, we simply pad it with zeros. Notice that R' has every other element of R, our desired output.

Pairing gets a running sum for even parity indices Specifically, every element in R' corresponds to an element with an index of even parity in A. It's as though as we did a running sum, but we only reported the running sum every two iterates. That is, we have an array A and R

$$A = \begin{bmatrix} a_1 & a_2 & a_3 & \dots & a_n \end{bmatrix}$$
$$R' = \begin{bmatrix} r_2 & r_4 & \dots & r_{n/2} \end{bmatrix}$$

That is, $r_2 = a_1 + a_2$, and $r_4 = a_1 + a_2 + a_3 + a_4$, and in general $r_k = \sum_{i=1}^{k} a_i$.

Filling in the odd-indices To compute the running sum for elements whose index is of odd parity in A, i.e. set

$$r_i = r_{i-1} + a_i$$

for $i = 1, 3, 5, \ldots$, where we by convention let $r_0 = 0$.

3.2 Algorithm Analysis

Our base case requires constant work and depth. Let's consider work and depth for each remaining step of the algorithm.

Pairing entries Notice that step 4, where we let A' be the sum of adjacent pairs, we must perform $n/2$ summations, hence work is $O(n)$. Realize that we may assign each processor a pair of numbers and perform the summations in parallel. Hence depth is $O(1)$.

Recursive call Step 5 is our recursive call, which is fed an input of half the size of A.

Filling in missing entries Notice that step 6, filling in missing entries, we can assign each of the $n/2$ missing entries of R to a processor and compute its corresponding value in constant time. Hence step 6 has work is $n/2$, i.e. $O(n)$, and depth $O(1)$.

Total work and depth Now let's consider the work and depth for the entire algorithm. This is where recurrences come into play. Let $T_1 = W(n)$, and $T_\infty = D(n)$,

$$\begin{aligned} W(n) = W(n/2) + O(n) &\implies W(n) = O(n), \\ D(n) = D(n/2) + O(1) &\implies D(n) = O(\log(n)). \end{aligned}$$

The expression for work follows since we make exactly one recursive call of exactly half the size, and in outside our recursive call we perform $O(n)$ work. By the Master Theorem, $W(n) = O(n)$.[1]

With regard to depth, again realize that we make a recursive call on input size $n/2$, and outside the recursive call we only require constant depth. Again by the Master Theorem, we see that $D(n) = O(\log n)$. For prefix-sum, this is pretty much the best we can hope for. We emphasize that recursion was critical for the parallelization of this algorithm.

A note on recursive algorithms and parallelization

We conclude with the remark that although recursive algorithms are amenable to parallelization, the algorithm designer must do some analytical work to make algorithms efficient in a parallel setting. Often times, we the combine step of a divide-and-conquer algorithm is sequential in nature, and can be the bottleneck of our analysis. To get around this, we must think carefully. We'll see more on this when we talk about MergeSort next lecture.

[1] Again, unrolling our recurrence we yield a geometric series scaled by n, hence work is $O(n)$.

4 Mergesort

Merge-sort is a very simple routine. It was fully parallelized in 1988 by Cole.[1] The algorithm itself has been known for several decades longer.

4.1 The mergesort algorithm

Algorithm 2: Merge Sort

Input : Array A with n elements
Output: Sorted A

```
1 n ← |A|
2 if n is 1 then
3 |   return A
4 end
5 else
  |   // (IN PARALLEL, DO)
6 |   L ← MERGESORT(A[0,...,n/2))        // Indices 0, 1, ..., n/2 − 1
7 |   R ← MERGESORT(A[n/2,...,n))        // Indices n/2, n/2 + 1, ..., n − 1
8 |   return MERGE(L,R)
9 end
```

4.2 Subroutine: merge

It's critical to note how the `merge` sub-routine works, since this is important to our algorithms work and depth. We can think of the process as simply "zipping" together two sorted arrays.

Algorithm 3: Merge

Input : Two sorted arrays A, B each of length n
Output: Merged array C, consisting of elements of A and B in sorted order

```
a ← pointer to head of array A (i.e. pointer to smallest element in A)
b ← pointer to head of array B (i.e. pointer to smallest element in B)
while a, b are not null do
    Compare the value of the element at a with the value of the element at b
    if value(a) < value(b) then
        add value of a to output C
        increment pointer a to next element in A
    end
    else
        add value of b to output C
        increment pointer b to next element in B
    end
end
if elements remaining in either a or (exclusive) b then
    Append these sorted elements to our sorted output C
end
return C
```

Since we iterate over each of the elements exactly one time, and each time we make a constant time comparison, we require $\Theta(n)$ operations. Hence the `merge` routine on a single machine takes $O(n)$ work.

4.3 Naive parallelization

Suppose we parallelize the algorithm via the obvious divide-and-conquer approach, i.e. by delegating the recursive calls to individual processors. The work done is then

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) \\ &= O(n \log n) \end{aligned}$$

by case 2 of the Master Theorem.

As you'll recall from earlier algorithms classes, the canonical implementation of the `merge` routine involves simultaneously iterating over L and R: starting at the first index of each, we merge them by placing the smaller of the currently pointed-to elements of L and R at the back of a new list and advance the pointer in the list that the just-placed element belonged to, and continue until we reach beyond the end of one list. Crucially, `merge` has depth $O(n)$. The depth is then

$$D(n) = D(n/2) + O(n)$$
$$= O(n)$$

again by the Master Theorem.

Using Brent's theorem, we have that

$$T_p \leq O(n \log n)/p + O(n)$$

Therefore $W(n) = O(n \log n)$ and $D(n) = O(n)$.

The bottleneck is in sequential merge subroutine Note that the bottleneck lies in `merge`, which takes $O(n)$ time. That is, even though we have an infinitude of processors, the time it takes to merge two sorted arrays of size $n/2$ on the first call to `mergeSort` dominates the time it takes to complete the recursive calls.

4.4 Improved parallelization

How do we merge L and R in parallel? The `merge` routine we have used is written in a way that is inherently sequential; it is not immediately obvious how to interleave the elements of L and R together even with an infinitude of processors.

Using binary search to find the rank of an element Let us call the output of our algorithm M. For an element x in R, let us define $\text{rank}_M(x)$ to be the index of element x in output M. For any such element $x \in R$, we know how many elements (say a) in R come before x since we have sorted R. But we don't immediately know the rank of an element x in M.

If we know how many elements (say b) in L are less than x, then we know we should place x in the $(a+b)^{th}$ position in the merged array M. It remains to find b. We can find b by performing a binary search over L. We perform the symmetric procedure for each $l \in L$ (i.e. we find how many elements in R are less than it), so for a call to `merge` on an input of size n, we perform n binary searches, each of which takes $O(\log n/2) = O(\log n)$ time.

$$\text{rank}_M(x) = \text{rank}_L(x) + \text{rank}_R(x)$$

4.5 Parallel merge

Algorithm 4: Parallel Merge

Input : Two sorted arrays A, B each of length n

Output: Merged array C, consisting of elements of A and B in sorted order

1 **for** *each* $a \in A$ **do**
2 | Do a binary search to find where a would be added into B,
3 | The final rank of a given by $\text{rank}_M(a) = \text{rank}_A(a) + \text{rank}_B(a)$.
4 **end**

Analysis of parallel merge To find the rank of an element $x \in A$ in another sorted B requires $O(\log n)$ work using a sequential processor. Notice, however, that each of the n iterations of the for loop in our algorithm is independent of the previous, hence our binary searches may be performed in parallel. That is, we can use n processors and assign each a single element from A. Each processor then performs a binary search with $O(\log n)$ work. Hence in total, this parallel merge routine requires $O(n \log n)$ work and $O(\log n)$ depth.

Hence when we use `parallelMerge` in our `mergeSort` algorithm, we realize the following work and depth, by the master theorem:

$$\begin{aligned} W(n) = 2W(n/2) + O(n \log n) &\implies W(n) = O(n \log^2 n), \\ D(n) = D(n/2) + \log n &\implies D(n) = O(\log^2 n). \end{aligned}$$

By Brent's Theorem, we get

$$T_p \leq O(n \log^2 n)/p + O(\log^2 n)$$

so for large p we significantly outperform the naive implementation! The best known implementation (work $O(n \log n)$, depth $O(\log n)$) was found by Richard Cole[1].

4.6 Motivating Cole's mergesort

We notice that we use many binary searches in our recently defined parallel merge routine. Can we do better? Yes. Let L_m denote the median index of array L. We then find the corresponding index in R using binary search with logarithmic work. We then observe that all of the elements in L at or below L_m and all of the elements in R at $\text{rank}_R(\texttt{value}(L_m))$ are at most the value of L's median element. Hence if we were to recursively merge-sort the first L_m elements in L along with the first $\text{rank}_R(\texttt{value}(L_m))$ elements in R, and correspondingly for the upper parts of L and R, we may simply append the results together to maintain sorted order. This leads us to Richard Cole (1988).[1] He works out all the intricate details in this approach nicely to achieve

$$\begin{aligned} W(n) &= O(n \log n) \\ D(n) &= O(\log n) \end{aligned}$$

References

[1] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.