ICS 643: Advanced Parallel Algorithms Fall 2016

Lecture 10 — October 5, 2016

Prof. Nodari Sitchinava *Scribe: Tiffany Eulalio*

1 Overview

In the last lecture, we finished our discussion on segmented prefix-sums. We looked into finding minima on common-CRCW PRAM, in parallel.

In this lecture, we look at merging of sorted arrays in parallel. We present a work-optimal parallel merging solution and then apply this to a parallel sorting algorithm.

2 Parallel Merging of Sorted Arrays

Given two sorted arrays, A and B, each of size n, we want to output a sorted array $C = A \cup B$. For simplicity, we will assume that the arrays contain distinct elements.

Definition 2.1. Given a sorted array A and some number x, a *rank* of x in A, denoted *rank(x, A)*, is equal to the number of elements in A that are smaller than or equal to x. In other words, if we start the indices of A from 1 and rank $(x, A) = i$, then $A[i] \leq x < A[i+1]$.

We can use rank to describe where the elements from our sorted arrays, A and B should be placed in array C. We assume our arrays contain no duplicate elements. Let i be the index in C where an element should be placed, then,

$$\begin{aligned} i &= rank(x, C) \\ &= rank(x, A) + rank(x, B) \end{aligned}$$

We can find $rank(x, A)$ by calling a binary search of x in A. We will assume that the binary search returns returns the index of the largest element in A that is smaller than x and zero if no such element exists. This finds the rank of x in A in $O(\log n)$ time and work. In Algorithm 1 we use the concept of rank, and thus, binary search, to merge two arrays.

We can analyze the runtime and work for this algorithm. Everything is done in parallel and our longest steps are when we call BinarySearch. We do this for both arrays A and B, which have lengths of n. BinarySearch takes $O(\log n)$ time, so we know that our runtime is $O(\log n)$. The **for** loops iterate n times, so our work is $O(n \log n)$. We can see then, that our Merge algorithm is not work-optimal since sequential merging takes linear time. We want to find a work-optimal version of Merge, without increasing the runtime.

We will partition one of our arrays into n' sections, each of size $\log n$. In this example, we'll choose array B to partition. Then, we'll select the elements b_i, such that $b_i = B[i \cdot \log n]$ for $1 \leq i \leq \frac{n}{\log n}$.

Algorithm 1 MERGE($A[1 \dots n]$, $B[1 \dots n]$)

```
in parallel do {
    for i = 1 to n in parallel do
        r_i = BINARYSEARCH(A[i], B)
        C[i + r_i] = A[i]
    for i = 1 to n in parallel do
        r_i = BINARYSEARCH(B[i], A)
        C[i + r_i] = B[i]
    }
return C
```

In other words, we're selecting b_i as the last element from each of the n' sections that resulted from our partition. Then, we'll find the rank of each b_i in the array A. A graphical representation of this process can be seen in Figure 1. The n' sections of the array are labeled B_i, where $1 \leq i \leq n'$. Each index j_i in array A is equal to $rank(b_i, A)$. Then, A is partitioned into $n' + 1$ sections of varying lengths around each index, j_i. Algorithm 2 is this partitioning algorithm.

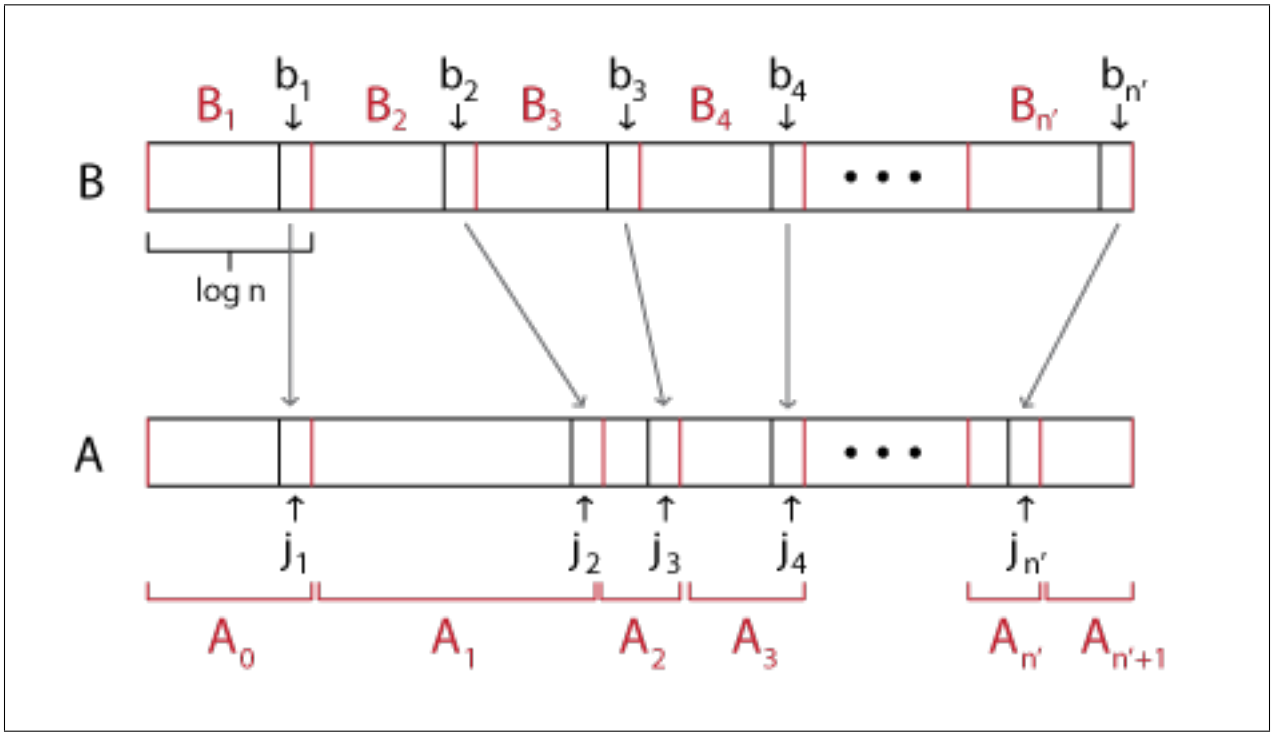

Figure 1: Graphical representation of the PARTITION algorithm on arrays A and B. B is partitioned into n' sections of size, $\log n$, and then rank in A of each element, b_i, is found.

Algorithm 2 PARTITION(A, B, n)

```
j_0 = 1
for i = 1 to n' = |B| / log n in parallel do
    j_i = rank(B[i · log n], A)
for i = 0 to n' in parallel do
    rangeA[i] = (j_i, j_{i+1})
```

We can analyze the runtime and work for the PARTITION algorithm. The steps are executed in parallel, so we're bounded by the step that takes the longest, which is finding the rank. Rank is found using binary search, so it'll take $O(\log n)$ time. We call $rank()$ on array A, so the runtime is $O(\log |A|)$. Work in the first **for** loop will dominate work in the second, so work will be $\frac{|B|}{\log n} \log |A|$.

After the arrays have been partitioned, we know that $A[j_i] < b_i$. If not, then the rank of b_i, which is j_i, would be larger, by the definition of rank. We also know that b_i is smaller than any element in B_{i+1}, since B is a sorted array. From this, we can say that any element in A_{i-1} is smaller than any element in B_{i+1}. We also know that any element in A_{i-1} is smaller than any element in A_i, since A is sorted. Then, we can also say that any element in B_i is smaller than any element in A_i, because if not, then the rank j_i would have been placed elsewhere. More specifically, j_i would have been placed after the largest element that is smaller than b_i. Thus, we can say that the partitions are separate from one another and so, we can merge each partition independently. So, we can say that, $C = (A_0 \cup B_1), (A_1 \cup B_2), \ldots, (A_{n'-1} \cup B_{n'}), A_{n'}$. We can now merge pairs of $(A_{i-1} \cup B_i)$ in parallel, and if any one of the partitions from A are empty, then we can just copy over the partition from B. Notice that if A_{i-1} and B_i are both, $c \cdot \log n$ in size, for some constant c, then we can merge $A_{i-1} \cup B_i$ in $O(\log n)$ time and work sequentially. Algorithm 4 presents an example of a sequential merge. However, we might encounter the case where the size of A is greater than $c \cdot \log n$. In this case, we repeat the partitioning, except this time, using A_{i-1} as B, and B_i as A. In other words, we'll call Algorithm 2, as PARTITION(B_i, A_{i-1}). We can now apply these concepts to write a work-optimal parallel merge that merges input arrays A and B into array C. We let C_i be equal to the subarray of C that holds $(A_i - 1 \cup B_i)$.

Algorithm 3 PARMERGE(A, B, n)

```
PARTITION(A, B, n)
for i = 1 to n/log n in parallel do
  if |A_{i-1}| ≤ c log n then
    C_{i-1} = SEQUENTIALMERGE(A_{i-1}, B_i)
  else
    C_{i-1} = PARMERGE(B_i, A_{i-1}, n)
Append A_{n+1} to C
```

We can analyze the runtime and work for PARMERGE. The **for** loop runs in parallel, so we need to find the longest step. We have an **if** and an **else** statement so we need to find out the runtime of these separately. The **if** statement calls SEQUENTIALMERGE, which has a runtime of $O(\log n)$ since $|A_{i-1}| \leq c \log n$. The **else** statement has a recursive call, but we know this will happen at most once. We'll need to call PARTITION and SEQUENTIALMERGE in the recursive call. However, we are calling these with an array size of $|B_i|$, thus, the runtime will be $O(\log |B_i|)$. This means that the runtime is $O(\log n)$. To solve for work, we look at the **else** statement since it has more steps than the **if** statement. It has to do the recursive call, and then reach the **if** statement, as well. The work done here is $O(|A_{i-1}|)$, which in the worst-case may be equal to n. Thus, we get, $W(n) = O(n)$, which is equal to the runtime for sequential merging, making this algorithm work-optimal.

Algorithm 4 SequentialMerge($A[1 \dots N], B[1 \dots N]$)

```
i = 1
j = 1
for k = 1 to 2N do
  if i ≥ N then
    C[k] = B[j]
    j++
  else if j ≥ N then
    C[k] = A[i]
    i++
  else if A[i] ≤ B[j] then
    C[k] = A[i]
    i++
  else
    C[k] = B[j]
    j++
return C
```

3 Parallel Sort

Using the parallel merge algorithm that we presented, we can conveniently create a parallel sorting algorithm.

Algorithm 5 ParSort($A[1 \dots n]$)

```
if |A| == 1 then
  return A
else
  in parallel do
      B = ParSort(A[1 ... n/2])
      C = ParSort(A[n/2 + 1 ... n])
return ParMerge(B, C)
```

We can analyze the runtime and work of ParSort. Since it's done in parallel, we have $T(n) = T(\frac{n}{2}) + O(\log n)$. We can use the Master Method to solve this equation. This falls under case 2, with $k = 1$, to give us the solution of $T(n) = O(\log^2 n)$. For the work done, we get $W(n) = 2W(\frac{n}{2}) + O(n)$. We can use the Master Method again, to see that this falls into case 2, with $k = 0$, to give us the solution, $W(n) = O(n \log n)$.