EE 382C/EE 371C **Multicore Computing** **Vijay Garg** **Fall 2018**

Lecture on Merging Two Sorted Arrays

1 Introduction

Suppose that we have two sorted arrays A and B, each of size n. We would like to merge them into another array C such that C is sorted. It is easy to design a sequential algorithm that merges them into C. We simply keep two indices i and j in arrays A and B, respectively. At any step of the algorithm, we compare $A[i]$ and $B[j]$. If $A[i]$ is smaller than $B[j]$, then we copy $A[i]$ into the next available slot in C and advance index i. If $B[j]$ is smaller than $A[i]$, then we copy $B[j]$ into the next available slot in C and advance index j. This algorithm takes $O(n)$ time.

```
Sequential Algorithm
i,j,k := 0,0,0;
  while (i < n) and (j < n) do
    if (A[i] < B[j])  {C[k] = A[i]; i++;}
    else  {C[k] = B[i]; j++;}
    k++;

 while (i<n) { C[k] = A[i]; i++;k++;}
 while (j<n) { C[k] = B[j]; j++;k++;}
```

2 A Parallel Algorithm

We now devise the first parallel algorithm. For simplicity, we assume that all elements are unique. This algorithm is based on finding the location in C where each element of A and B will appear. If every element in A and B determines its rank in parallel, it can write that value at the correct location in C.

Define the rank of any element x as the number of elements in A and B that are less than x, i.e., $rank(A[i], C)$ = the number of elements in A less than $A[i]$ + $rank(A[i], B)$. The first number is simply i (for arrays that start with index 0). The second number can be determined using binary search in $O(\log(n))$ time. So the overall time is: $T(n) = O(1) + O(\log(n))$. This algorithm requires concurrent reads but no concurrent writes, so a $CREW$ PRAM is sufficient. However, this algorithm is not work-optimal since it requires n processors doing $O(\log(n))$ work, or $W(n) = O(n * \log(n))$.

3 A Work-Optimal Parallel Algorithm

Can we combine the slow (sequential) but work-optimal algorithm with the fast (parallel) but work-suboptimal to get fast work-optimal algorithm? The idea is to divide arrays A and B into $n/\log n$ groups of size $O(\log n)$. Let us call the least element of these groups as splitters. We have

$O(n/\log n)$ splitters. Using the fast parallel algorithm, we can compute the rank of all splitters in C in $O(\log n)$ time using $O(n/\log n * \log n) = O(n)$ work.

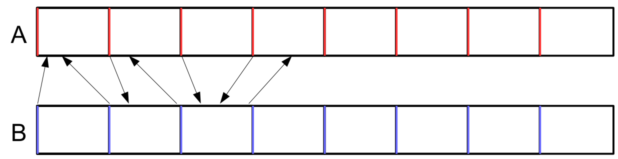

Figure 1: The splitters can go anywhere but they can never cross.

Our next task is to compute the ranks of non-splitters. We will employ the sequential merging algorithm for finding ranks of non-splitters. It is sufficient to find the rank of a non-splitter relative to the rank of the splitter that precedes it. Fig. 1 shows how various splitters divide up the segments. Once all splitters are placed in the target array, we only need to merge small groups appropriately.

Let p and q be any two consecutive splitters. Without loss of generality assume that p is in array A. We have two cases. First consider the case when q is also in array A. In this case, we need to merge the sublist between p and q in A and the sublist $rank(p, B)..rank(q, B) - 1$ in B sequentially to find the ranks of all elements between p and q. Each of these sublists are of size $O(\log n)$. Now consider the case when q is in array B. In this case, we need to merge the sublist between p and $rank(q, A) - 1$ in A and the sublist between $rank(p, B)$ and q in B. Therefore, we can compute the ranks of all nonsplitters between any two splitters in $O(\log n)$ time. Similarly, the rank of nonsplitters after the last splitter can also be computed in $O(\log n)$ time. Since there are $O(n/\log n)$ non-splitters sublists, the total work equals $O(n/\log n * \log n) = O(n)$. Combining the total work for splitters and non-splitters, we get $O(n)$ work which is optimal.