Parallel Algorithms
Prof. Sajith Gopalan
Department of Computer Science & Engineering
Indian Institute of Technology Guwahati

Lecture – 08
Basic Techniques 4

Welcome to the 8th lecture of the NPTEL MOOC on Parallel Algorithms.

(Refer Slide Time: 00:39)

In the previous class, we were discussing the algorithm design technique called divide and conquer in which a given problem is divided into several sub problems. The sub problems are solved in turn and then their solutions are combined to get the solution for the given problem. So, we have seen a couple of examples of divide and conquer, we shall see more problems that can be solved using divide and conquer today, but before that one particular problem that we will be using as a subroutine which is the problem of searching.

(Refer Slide Time: 01:13)

Search

Sorted array of size n

element x

if x is present in the array, what is its rank?

Let us say we are given a sorted array of size n and we have an element x. We want to find if the element is present in the array or not and if the element x is present in the array, what is its rank? The rank of an element in a sorted array is the number of elements in the sorted array that are smaller than or equal to the given element. So, we want to find the number of elements less than or equal to x in the given sorted array.

(Refer Slide Time: 02:23)

Search 1

x

i = 3

SEARCH-1

Input: A sorted array A of size n and an item x.

Output: The rank R of x in A. Model: CREW PRAM

```
{
    R = 0;
    pardo for 1 ≤ i ≤ n − 1
        if (A[i] ≤ x  &&  A[i+1] > x)  R = i;
    return R;
}
```

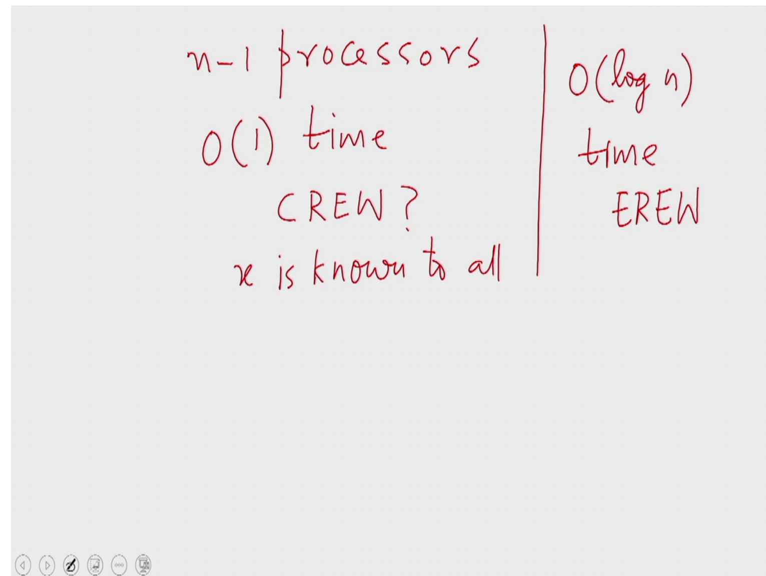
So, we have the sorted array A of size n and an item x, we want to find the rank R of x and A. Let us say the model is CREW PRAM so, we initialize R to 0 and then we do in

parallel for i varying from 1 to n minus 1, that is let us assume that we have one processor for every single element in the array except the last. So, we assume there is a processors stationed at every single element except the last. What these processors do is this the process, each processor is standing on an item i. So, the processor will compare x with Ai as well as Ai plus 1 that is the item on which it is standing and as well as the item to the right.

If x is greater than or equal to the item on which it stands and is less than the item that is to it's right then x will have to fall between these two that is between the item on which the processor stands. And the next item then the processor will report i as the rank of x. So, its like this, we have a number of elements and the processors are standing on each element. This one particular processor finds that x is greater than or equal to this element and less than or equal to the next element which means x falls between the third and the fourth elements in this figure.

In that case this process of the third processor will report i as its index which is 3 as the result. So, what it means is that the element x falls within the third segment, we assume that the sorted array divides the real line into a number of segments. So, the element that we want falls between the third element and the fourth element. If x does not fall between any two of the given elements that is if x falls to the left of the leftmost element then R will retain its initial value which is 0. So, you can see that the algorithm works correctly provided; we have n minus 1 processors with n minus 1 processors. The algorithm runs in order of one time so, here we have assumed that the model is CREW, but where did we use the concurrent read.

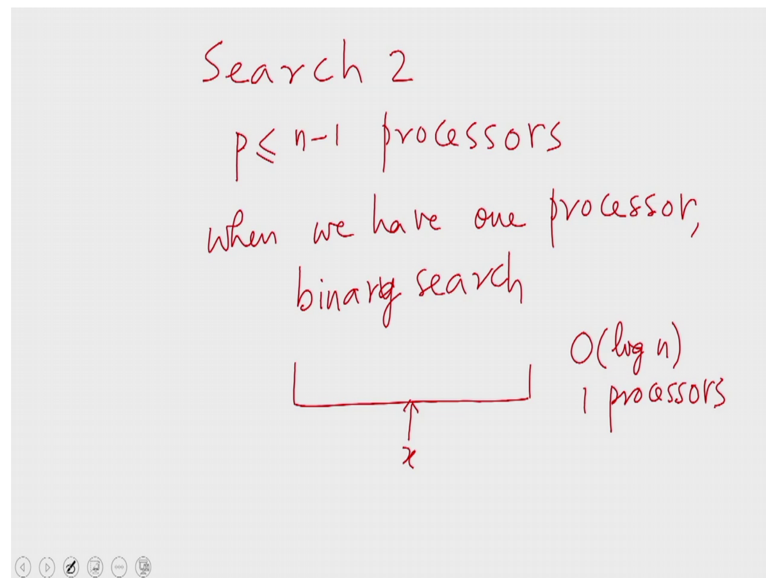
(Refer Slide Time: 04:41)



If you look at the code you would not think that concurrent read is being used anywhere, because the access to Ai and Ai plus 1. Since, they are synchronized will never cause a read conflict when the ith processor is accessing Ai the i plus first processor is accessing Ai plus 1. Later when the i th processor is accessing Ai plus 1, the i plus first processor is at accessing the Ai plus second element i plus second element Therefore, there will be no read conflict, but CREW model is necessary, because x is not known to all initially.

So, if all the processors have to read x simultaneously, we will require concurrent read, but if x is known only to one processor and this has to be broadcast over the other processors then that would require order of log n time on the ER EWP R. Therefore, if this has to be executed on the CREW PRAM, it would take order of log n time if x is not known to every single processor to begin with. FFor now let us continue with the CREW PRAM model, let us say how we would execute search when we have P processors.

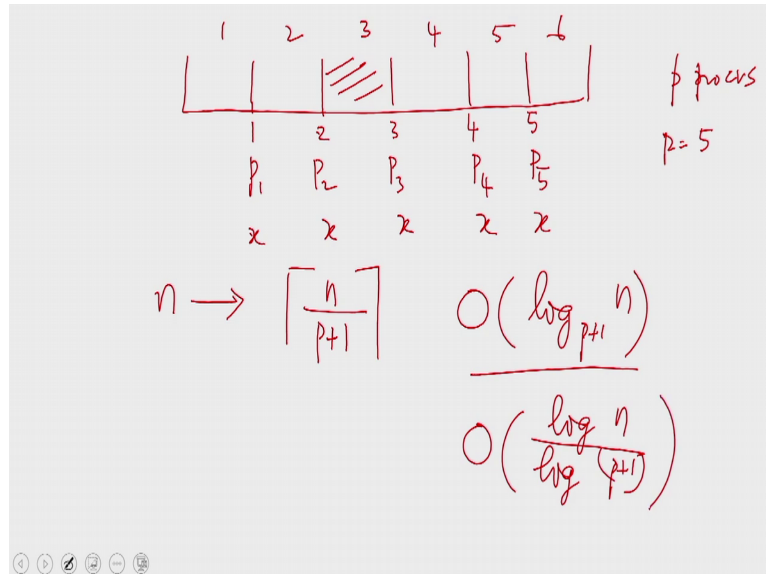
(Refer Slide Time: 06:41)



So, we are going to generalize this algorithm. Let us say we have P less than or equal to n minus one processors in particular when we have only one processor which is the sequential case. We now we would use binary search. In binary search, we look at an array of size n and then first we look at the midpoint x is compared with the element in the middle.

If x is smaller than the midpoint, then we will continue with the left side otherwise, we will continue with the right side. So, after every comparison the search space reduces by a factor of two. Therefore, if we are starting with n elements to begin with in log n steps the search space will reduce to one single element. Therefore, the algorithm will run in order of log n time, this is the sequential algorithm. So, now, we have two algorithms representing the two x streams. If we have 1 processor, we will be able to perform the search in order of log n time. If we have n minus 1 processors we will be able to perform the search in order one time what if we have P processors where P is less than or equal to n minus 1 then; obviously, we should be somewhere between the two x streams.

(Refer Slide Time: 08:43)



So, can we design an algorithm which generalizes, generalizes to the two x streams such an algorithm can be designed like this. Let us say we are given an array of size n and we have p processors. Let us say we divide the array into p plus 1 segment. So, in this case p is 5 and there are 6 segments, 6 segments correspond to 5 sentinels. So, let me call these sentinels they separate the segments.

So, there are 5 segments separation points and in therefore, in this example we assume that p equal to 5 so, we have 5 processors. What we do is this, we depute these 5 processors to these positions, this is a generalization of the binary search. In that in the case of binary search, we have one processor and we send the processor to the middle of the array. Here, we have 5 processors so, what we do is to divide the array into 6 segments and send the processors to the segment separations. There are 5 segment separations so, at these points, we compare the element that is present with x. So, x is compared with all these positions, we assume that every processor knows x. So, our concurrent read has ensured that every process and knows x a priori and then x is compared with all these elements.

Let us say x is found to fall within this segment that is x is greater than or equal to the element with which P 2 compares it, but less than the element with which P 3 compares it. Therefore, we should continue with the search using the segment between 2 and 3 which means the range of search has now reduced from n to n by P plus 1. Initially, we had n elements, we divided these elements n elements into P plus 1 segments. Now we

have reduced the search space to one such segment the size of the segment is n by P plus 1 ceiling.

So, in order one time we have managed to reduce the problem size from n to n by P plus 1, this is analogous to what we do in binary search. In binary search we reduce the search space from a size of n to a size of n by 2 in one single step. So, continuing like this, we find that the search can be executed in order of log n to the base P plus 1. After one step the search space reduces by a factor of P plus 1 after two steps the search space has reduced by a factor of P plus 1 squared and so on.

So, after log of n to the base P plus 1, steps the search space has reduced to a size of 1. So, on the CREW PRAM with P plus 1 processors, we can perform the search in order of log n to the base log of P plus 1 time. This is recalled is the same as log n divided by log of P plus 1 where the base of the logarithm is 2.

(Refer Slide Time: 12:25)

Search 2

SEARCH-2

Input: A sorted array A, an item x, and p the number of processors.

Output: The rank R of x in A. Model: CREW PRAM

{
  if $(n \leq p+1)$ return SEARCH-1(A, x);
  pardo for $1 \leq i \leq p$
    if $(x \geq A[i\lceil \frac{n}{p+1} \rceil])$ $B[i] = 0$;
    else $B[i] = 1$;
  $B[0] = 0$; $B[p+1] = 1$;
  pardo for $0 \leq i \leq p$
    if $(B[i] == 0$ && $B[i+1] == 1)$ $R = i+1$;
  return $(R-1)\lceil \frac{n}{p+1} \rceil +$ SEARCH-2$(A[(R-1)\lceil \frac{n}{p+1} \rceil + 1 \ldots R\lceil \frac{n}{p+1} \rceil], x)$;
}

So, let us look at a formal specification of the algorithm when the input is A sorted array A and we have to search for an item x and p is the number of processors where we want to find the rank R of x in A we proceed as below. When n is less than or equal to p plus 1 that is the number of elements is less than or equal to the number of processors, then we use the first algorithm that is we can perform the search in order one time. Otherwise, we divide the array into p plus 1 segments and then at each segment separation point.

So, i multiplied by ceiling of n by p plus 1 for each i varying from 1 p is 1 such segment separation point. At each segment separation point, we compare x with the point. If x is greater than or equal to that then we set the Bi value of that to 0 otherwise we set the Bi value of that to 1. So, what happens is this; here we have an array of size n, let us say the element falls within this segment then how would the B values be B of 0 is 0 and B of p plus 1 is 1 at all these positions, we will have a 0.

Here we have 1; here we have 1 and here we have 1. So, we find that the segment which contains the element corresponds to the switch from 0 to 1. So, for i varying from 0 to p if in parallel we check this if Bi is 0 and Bi plus 1 is 1, then we can declare the result as i plus 1 and then we have to confine ourselves to this segment that we have found.

So, we recursively invoke search to on this segment the reduced segment which will give us the offset within the segment of x. That offset has to be added to R minus 1 times ceiling of n by p plus 1 which is the total of this length, the sum then would be the rank of x within the array. So, if x has been found to be within this and the offset is somewhere here, then the sum of these will be the rank of x within the array. So, this is recursively specified algorithm outside of the recursion, the time taken is order 1, but from one recursive call to the next the size of the problem reduces by a factor of p plus 1. Therefore, the depth of the recursion is log of n to the base p plus 1. When we start with n elements since at each level of the recursion, we spent order 1 time, the total time taken by the algorithm is log n to the base p plus 1.

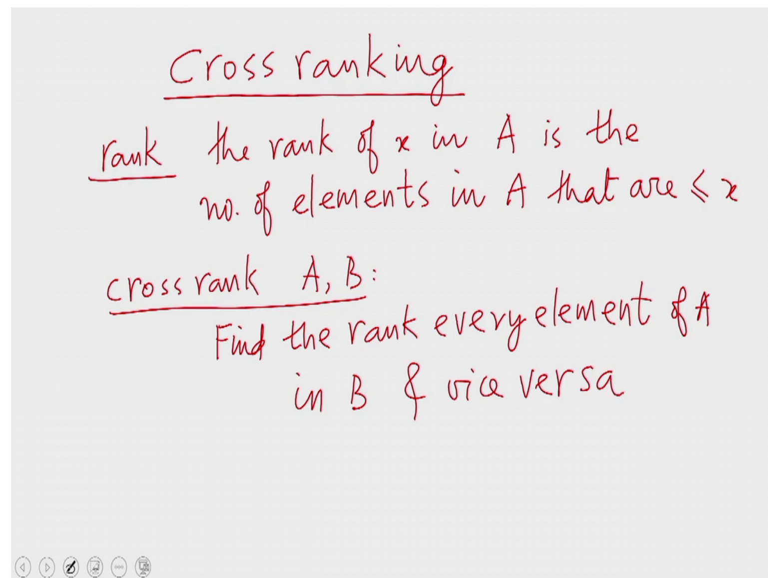
(Refer Slide Time: 15:35)

So, we can see that the two algorithms binary search where we use 1 processor and therefore, the running time is order of log n to the base 1 plus 1. And the n minus 1 processors search which runs in order of 1 time or both generalizations of the algorithm. We have seen where we use p processors and the algorithm runs in order of log n to the base p plus 1 times.

The model used is the CREW PRAM, we require the concurrent read, because every processor should read the item to be searched x simultaneously if we do not assume that every processor has a knowledge of x then we can broadcast x over the processors which would take an additional log of p time with p processors.

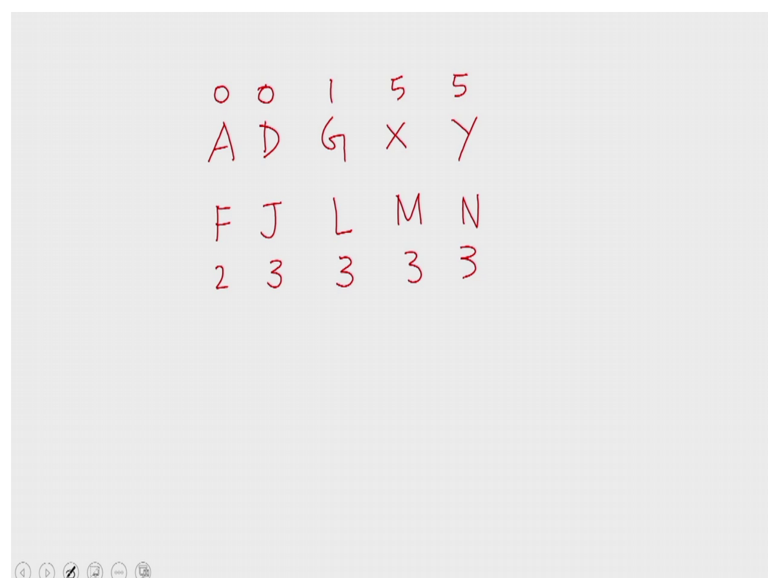
So, on CREW PRAM this algorithm can be simulated in order of log p plus log of n to the base p plus 1 times. Now that we have seen an algorithm for searching, we will try to solve the problem of cross ranking.

(Refer Slide Time: 17:15)



In cross rankings, we are given two sorted arrays and we want to cross rank the two arrays to define the problem first, let me define the notion of a rank. The rank of an element x in a sorted array A is the number of elements in A that are less than or equal to x. So, by cross ranking of two arrays, two sorted arrays A and B what we mean is that find the rank of every element in A and B and vice versa. For example, let us say we are given these two arrays.

(Refer Slide Time: 18:45)



This is the sorted array an array of alphabets, this is another sorted array. Let us say we want to cross rank the two arrays, consider the element A in the first array the number of elements less than or equal to A, in the other array is 0. Similarly, D has no smaller element in the other array therefore, the cross rank of D is also 0 and the cross rank of G is 1, because F is the only smaller element in the other array X and Y have cross ranks of 5 and 5 respectively.

That is because all the elements in the other array are smaller than X and Y on the other side when I consider element F, we find that A and D are smaller than F. So, F has a cross rank of 2, J has a cross rank of 3, because A D G are smaller than J and X and Y are larger, L M N all have cross ranks of 3, because A DG are smaller than all of them. So, these are the cross ranks of the two arrays shown in the diagram. So, let us say we want to find the cross ranks of the two arrays.

(Refer Slide Time: 20:09)

Cross Rank

CROSS-RANK-1

Input: Two sorted arrays $A[1..m]$ and $B[1...n]$; $n \geq m$.

Output: Arrays $\alpha[1...n]$ and $\beta[1...n]$ such that the rank of $A[i]$ in B is $\alpha[i]$ and the rank of $B[i]$ in A is $\beta[i]$. Model: CREW PRAM

```
{
    pardo for 1 ≤ i ≤ n
        α[i] = SEARCH-2(B[1...m], A[i], 1);
    pardo for 1 ≤ i ≤ m
        β[i] = SEARCH-2(A[1...n], B[i], 1);
}
```

$O(\log m)$, $O(\log n)$ } $O(\log n)$

So, this is the algorithm for that we are given two sorted arrays A and B of sizes m and n respectively. Let us say n is greater than or equal to m should be m and we are interested in finding the cross ranks of A and B. So, the cross ranks will be given in arrays alpha and beta alpha will be of size m and beta will be of size n. Let us say the model is CREW PRAM. So, what we do is this first in parallel for every element of A, we search for an element of A within B and then for every element of B in turn we search within A.

So, in the first step, we have assumed that we have one processor for every element of A and in the second step, we assume that we have one processor for every element of B. So, we have to pardo steps, both of them executing in order of log n time. So, here we are in the first step, we are searching within an array of size m.

So, this will take order of log m time in the second step we are searching within an array of size n so, this will take order of log m time log n time. So, together the algorithm takes order of log n time, if we have n plus m processors. So, what it means is that the cross ranks can be found in order of log n time using m plus n processors where m and n are the total number of elements. By the way, here we have assumed that B is of size m and A is of size n. So, cross ranking can be done in order of log n time using or m plus B processors.

(Refer Slide Time: 21:57)

Now, let us see how to perform a merge of two sorted arrays. Let us consider the same two arrays that we have seen, just now for cross ranking that have elements A DG X and Y in one array, FJ L M N in the other array. We had found these cross ranks, element A knows that it is the first element in its own array, element D knows that it is the second element in its own array and so on. So, every element has its own rank, its rank in its own array. Similarly, FJ L M N respectively know that they are at positions 1 2 3 4 5 in their own array.

So, in particular consider the element the 3rd element in the top array which is G, G knows that it's cross rank in the other array is 1 which means 1 element is less than G in the other array. And 3 elements are less than or equal to G in the first array put together there are 4 elements that are less than or equal to G in the two arrays together.

So, if we add the rank of an element in its own array with the cross rank, we will get these values 3 plus 1 equal to 4, 4 plus 5 is 9, 5 plus 5 is 10. And on the other side, we have 3 3 plus 2 5, 3 plus 3 6, 4 plus 4 7 plus 5 8. So, this way we have found the rank of every single element in the output array, the rank of an element in the merged output array. In the output array every element should appear in sorted order. The rank of an element in the output array will be the cross rank added to its own index. Within the array in particular for element G, the cross rank is 1 which is the number of elements less than or equal to G in the other array. And its own rank is 3 which means at most three elements R less than or equal to G in its own array 3 elements less than or equal to G in its own array.

So, the total number of elements less than or equal to G in the sorted array is going to be 4. Therefore, G should occupy the 4th position in the sorted array. So, every processor sitting on every single element in these two arrays. Now know which position, it should occupy in the final sorted array? So, likewise so, accordingly if we take 10 positions, if A now, if we take an array of size 10 in which we are going to write the output G knows that it is, it should occupy the 4th position.

Likewise, A knows that it should occupy the 1st position, D knows that it should occupy the 2nd position, X and Y know that they should occupy the last two positions. Then the remaining positions will be occupied by FJ L MN and the array will be sorted. So, cross ranking is essentially equivalent to merging two arrays.

(Refer Slide Time: 25:25)

Merge

$O(\log n)$ time
$n+m$ processors

MERGE-1

Input: Two sorted arrays $A[1 \dots n]$ and $B[1 \dots m]$; $n \geq m$.

Output: Array $C[1 \dots m+n]$, the merge of A and B. Model: CREW PRAM

```
{
    CROSS-RANK-1(A[1...n], B[1...m], α[1...n], β[1...m]);      O(log n)
    pardo for 1 ≤ i ≤ n
        C[i + α[i]] = A[i] ;                                   } O(1) time
    pardo for 1 ≤ i ≤ m
        C[i + β[i]] = B[i] ;
}
```

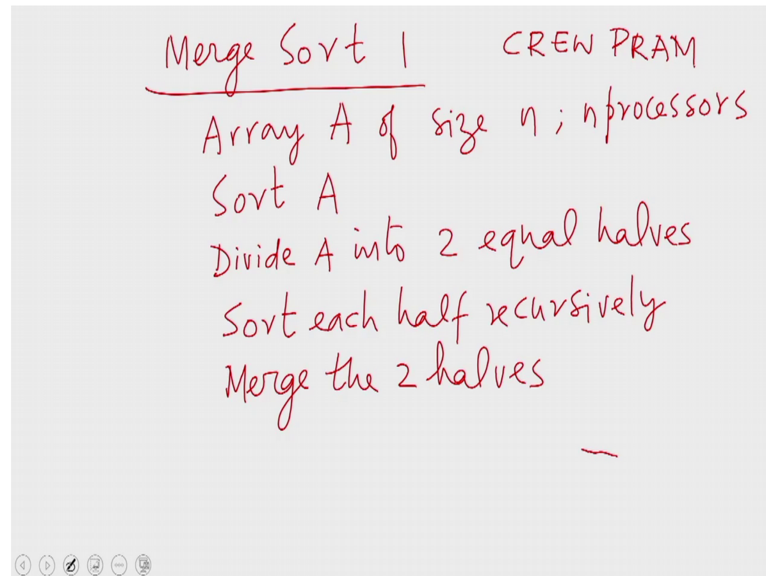
So, now let us look at the code for this, we are given two sorted arrays A and B of sizes n m respectively that let us say n is greater than or equal to m. Our goal is to merge them, the output has to be written in an array C which is of size m plus n and the model is CREW PRAM.

So, in the 1st step we cross rank A and B with the cross ranks stored in arrays alpha and beta, respectively. Then we have two steps, in the first parallel step every element of A will be copied into the responding element of C. The ith element of A knows that it is greater than or equal to i elements in A and it's cross rank is alpha i which means it has at most, it has exactly alpha i elements less than or equal to that.

In the other array namely B therefore, it's position in the final array is going to be i plus alpha i. So, what we do in the first parallel step is to copy every element of A into the corresponding in into its corresponding element in C in particular A i is copied into C of i plus alpha i. Similarly, in the second step the i th element of B will be copied into the i plus beta i th position of C. So, these two parallel steps each takes 1 or 1 unit of time.

So, this part of the algorithm runs in order of 1 time and the cross ranking part. As we have seen runs in order of log n time assuming that n is greater than or equal to m using n plus m processors. So, the overall time complexity of the merge algorithm is order of log n using n plus m processors. As soon as we have an algorithm for merging we can obtain a divide and conquer algorithm for sorting which is called the merge sort.

(Refer Slide Time: 27:33)



We shall see several radians of merge sort this is the first of them. So, given an array A of size n we need to sort A. So, the standard divide and conquer algorithm proceeds like this, divide A into 2, 2 equal halves, sort each half recursively and then merge the 2 halves. So, let us assume that we have n processors and that our model is a CREW PRAM. So, when we divide A into 2 equal halves, we will apportion the processors also. Accordingly half the processors will go to one problem instance and half the processors will go to the other problem instance.

Then we will have two recursive invocations, each will have an array of size n by 2 with n by 2 processors which is identical to the main invocation. Where we had n elements and n processors, the processor advantage is 1, in the recursive call as well. So, after solving the recursive instances these two instances will be solved simultaneously and then the two solutions will be combined by invoking the merge algorithm that we have just seen the merge algorithm runs in order of log n time using n processors.

In this case the two arrays have exactly the same size. Therefore, the time complexity of the result and the sorting algorithm can be calculated using a recurrence relation.

(Refer Slide Time: 29:51)

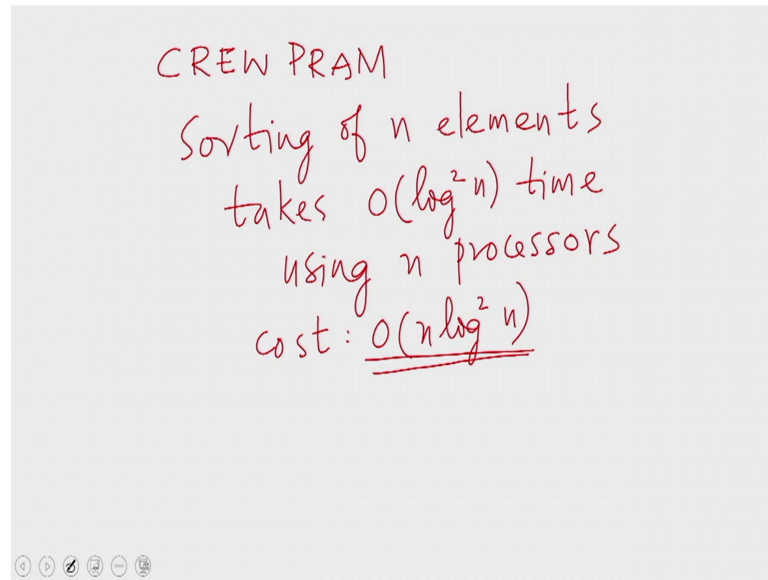
Let T of n denote the time required to sort n elements. So, what we have done is to divide the array into 2 equal halves and invoke the algorithm recursively on each half, but the two halves are running simultaneously. Therefore, the time taken will be T of n by 2 for the recursive calls. Since, the two recursive calls are running simultaneously, we do not have to count their times separately T of n by 2 does not have any multiplicative factor.

And then merging them will take order of log n time so, that becomes additive term c times log n. So, this is the recurrence relation that governs the sorting algorithm T of n equals to T of n by 2 plus c log n. So, let us see, what is the solution for this recurrence relation? If you unroll the recurrence relation, we find that this is T of n by 4 plus c times log of n by 2 plus c times log of n. If we unroll once more, we get T of n by 8 plus c into log of n by 4 plus log of n by 2 plus log n.

So, if you continue like this for k steps, the unrolled the occurrence relation will look like this. If we put k equals log of log n minus 1 then the first term becomes T of 2 which is 1. Therefore, you can say this is T of 2 plus c times which is order of log squared of n that is because the last term here is log n, the previous term is log n minus 1, then the second term from the right end is log n minus 2 and so on.

So, when we add this sequence log n log n minus 1, log n minus 2, etcetera. All the way down to 2, the sum is going to be order of log squared of n. So, the sorting algorithm runs in order of log squared n time using n processors.

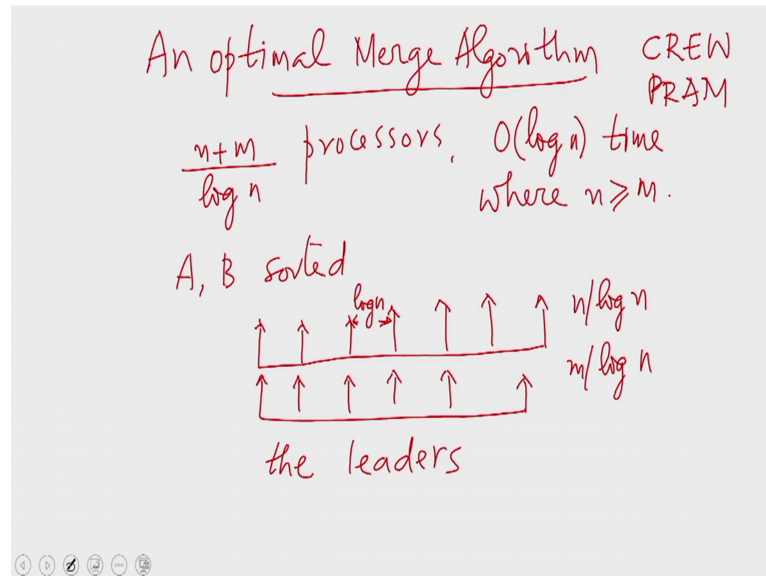
(Refer Slide Time: 33:27)



So, now we have seen that on a CREW PRAM sorting of n elements takes order of log squared n time using n processors. This is not an optimal algorithm, because the cost is order of n log squared n which is a log in factor of from optimality. An algorithm will be optimal only if the cost is order of n log n which matches the pen complexity of the sequential merge sort. We shall see later an algorithm which sorts optimally.

So, we have now seen a sorting algorithm which runs in order of log squared n time which in turn uses a merging algorithm that runs in order of log n time both on CREW PRAM.

(Refer Slide Time: 34:51)



Now, let us see how to design a merge algorithm that is optimal, the model is the same CREW PRAM. We shall use n plus m divided by log n number of processors to merge two sorted arrays of size n and m respectively. In order of log n time where n is greater than or equal to m. Therefore, the cost of the algorithm will be order of n plus m, the number of processors into the time taken therefore, the algorithm will be optimal. So, this will be a divide and conquer algorithm which in turn uses the previous algorithm as a subroutine.

So, given two sorted arrays A and B, what we do is this, the arrays A and B would both be divided into segments of size log n each. Therefore, we will have n by log n segments from one array and m by log n segments from the another array. In the first step, we divide the two sorted arrays in to the segments of size log n each, there will be n by log n segments from one array and m by log n segments from the from the other array. And then from each segment, we pick the first element as a special element, we select this element out of each array, these elements which occupy the first position of a segment are called the leaders.

(Refer Slide Time: 37:29)

A′ : the array of leaders from A

B′ : ——————————— B

crossrank A′ & B′.

$r_1[i]$: rank of $A'[i]$ in B'

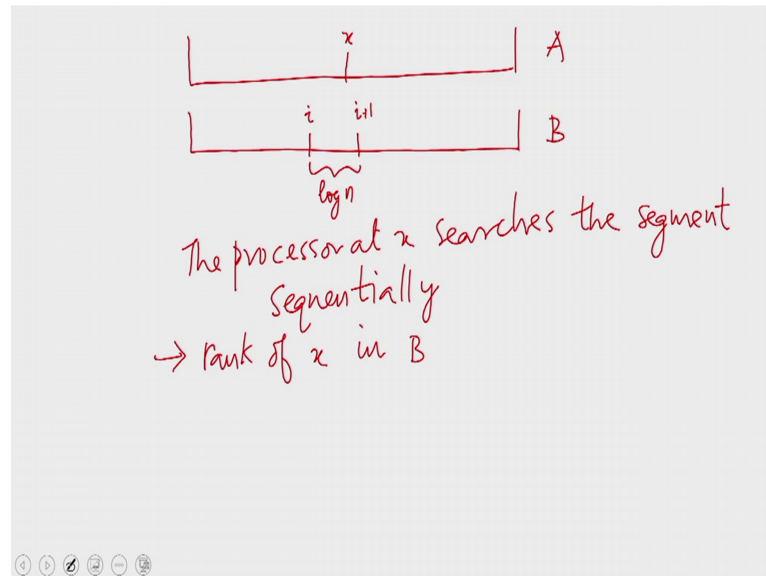
$r_2[i]$: ——— $B'[i]$ in A'

each leader knows the segment on the other side, in which it belongs

So, we have m plus n divided by log n leaders, the number of leaders is exactly equal to the number of processors we have. Let us say A prime is the array of leaders from A. Similarly, B prime as the array of leaders from B, the next step is to cross rank A prime and B prime using the available number of processors. Since, the number of processors is exactly equal to the total size of A prime and B prime, we can use the previous algorithm that we have seen that algorithm works when total number of process is exactly equal to the total size of the arrays. So, using that algorithm, we cross rank A prime and B prime, let us say r 1 of i is the rank of A prime of i, the i th leader of A in B prime similarly, r 2 of i is the rank of B prime of i in A prime.

So, now after the cross ranking is done, each leader of A prime knows two leaders on the other side between which it falls. Similarly, each leader in B prime knows to leaders in A prime between which it falls. In other words each leader on either side knows the segment on the other side in which it belongs.

(Refer Slide Time: 39:51)

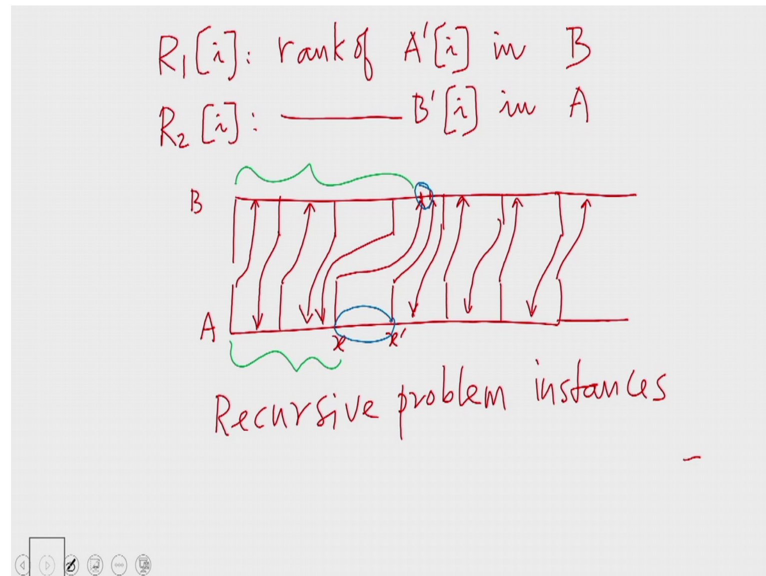


So, let us consider the two arrays A and B, consider a leader in A. Let me call that leader x, the leader x knows the segment in B in which it falls. In particular, if the segment is the one from which the ith leader has been taken that is this is the ith leader, x knows that it is between the ith leader and the ith plus first leader of B, the number of elements between these two leaders in B is log n.

We have assumed that we have exactly as many processors as there are leaders. So, if you depute one processor to x, this processor can go to this segment within B to which x belongs and search the segment sequentially. The processor at x searches the segment sequentially to see where x fits that is in sorted order where we would x fit so, this would give us the rank of x in B. So, this can be computed by every single x in parallel, in order of log n time that is because we have assumed that every leader has one processor.

This processor can go to the segment on the other side to which the leader x belongs and search within the segment sequentially. Since, the length of the segment is log n the sequential search will come to an end in order of log n time even if we use a linear search. Therefore, in log n time we would have ensured that every leader would be ranked in the other array.

(Refer Slide Time: 42:01)



Let me define R 1 of i of the rank of i, rank of A prime of I, the ith leader on the A side in B R 2 of i capital R 2 of i is the rank of B prime of i the ith leader of B in A. Now consider the two arrays, let me draw the two arrays with the leaders facing each other. The figure looks like two combs kept with the teeth facing each other. Now what we have known as that for every single leader, it has found its rank in the other array.

So, let us say the first leader on this side has found its rank like this. The second leader finds its rank here, then naturally the second leader on the other side will have to be ranked between these two, because of the sorted order. What we find this then, because of the sorted order, if we add pointers in this fashion, the pointers will never cross each other. In particular if I consider the leader x on the lower side. So, let the lower side be A and the upper side be B.

If x prime is the next leader in sorted order whether it be from the same side or the other side, then if we consider the reflections of these two leaders. What I mean by reflection is the position at which it would fall in the other array. The rank of these two leaders if I consider the reflections of these two leaders, I can take all the elements falling between these reflections and I can also consider the elements that fall between x and x prime these elements.

We find the elements that are circled in blue, we find are strictly between x and x prime. And these elements are all larger than every element appear in appearing to the left of

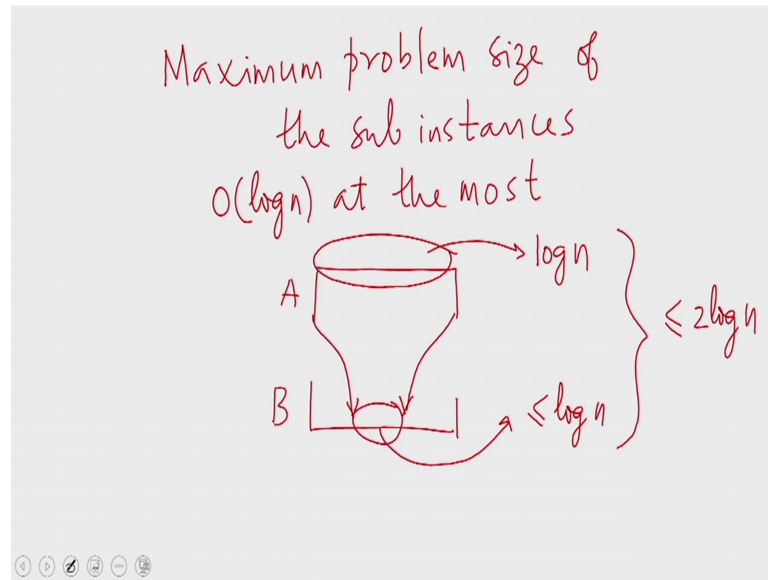
these blue circles and are smaller than every element appearing to the right of these blue circles in the two respective arrays. Therefore, if the elements belonging to these are merged, these two circled arrays are sub arrays of already sorted arrays therefore, they are sorted sub arrays.

So, what we have is a sub instance of merging corresponding to the leader x when we combine these two. When we merge these two sub arrays in sorted order and place them in the exact location where there should be in the final array and this is done in parallel for every single leader, the problem approved would be solved. But then how do we know where these elements should go? The elements obtained by merging the two blue circles will have all these elements less than them.

And these elements would also be less than them, we know the total number of elements marked in green that will be where the result of the merge should go in the output array. So, once every single leader calculates the offset at which its results should go. Then all that we have to do is to solve the individual merge instances in parallel and write the outputs in the exact place in the output array.

So, the original merge problem now breaks down into several recursive problems. Originally, we assume that we have as many processors as there are elements. So, if these processors are distributed one per element then for every recursive instance again we would have exactly as many processors as there are elements in the recursive instance. So, the recursive problem instances can be invoked exactly the same way, this call has been invoked with an equal number of processors to the problem size.

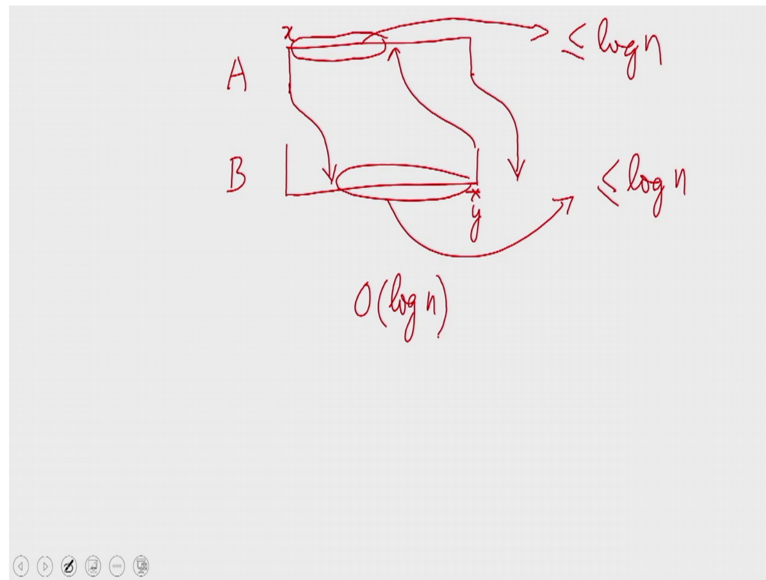
(Refer Slide Time: 47:13)



Now let us see what could be the maximum problem size of the recursive instances. My mistake, these are not recursive instances an algorithm which uses a recursive invocation; we shall see later in this we will not be using recursion. So, the sub instances will have a size of order of log n at the most we can argue those in this fashion. Consider two leaders on the A side, it could be that both the leaders rank into the same segment on the other side.

In which case the number of elements between A and B in its own array would be log n, because that is the size of a segment, but the number of elements between A and B from the other array would be at most log n. Therefore, the total problem size can never exceed 2 log n. In this case that is this two consecutive leaders on the same side rank within the same segment on the other side.

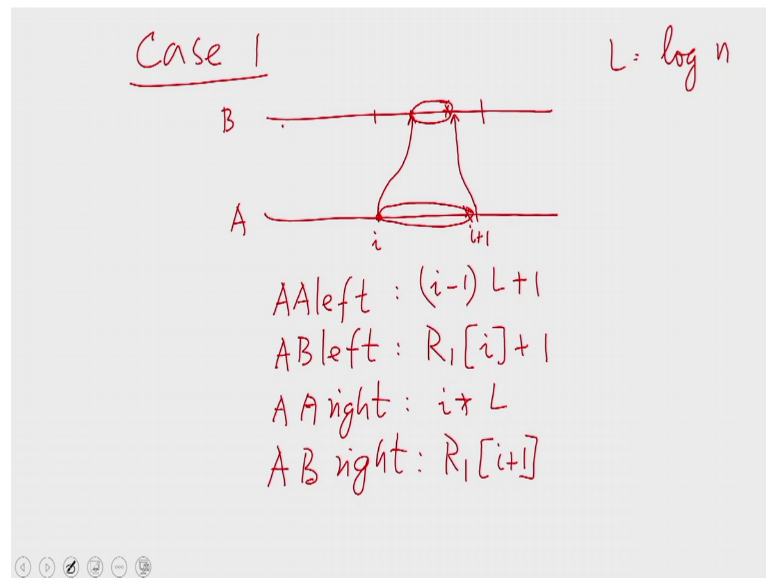
(Refer Slide Time: 48:57)



On the other hand, if two consecutive leaders on the A side are not ranking into the same segment on the other side. Let us say the first leader ranks here, but the second leader ranks in another segment. Therefore, this leader the right leader on the B side will have to be ranked between the two leaders on the A side. Therefore, the two consecutive leaders, if this is x the nearest leader of x would be this one y.

Therefore, the number of elements between x and y would be the total number here plus the total number here. So, as you can see this is at most log n and this is also at most log n. Therefore, the total number of elements in the problem instance is going to be order log n again which means all the sub instances can be solved in order of log n time. And then when they are copied into the exact place the entire array will be sorted as well.

(Refer Slide Time: 50:13)



So, what would be the boundaries of the sub instances? We will have two cases. Consider the ith leader and the i plus first leader on the A side. Suppose both of them rank into the same segment on the other side, then for the problem instance which is circled here the left and right boundaries can be written like this by A A left. Let me name the first element on the A side going into the sub instance.

Since the first element of every segment is picked out as a leader. The rank of this element within A would be i minus 1 into L plus 1 where L is log n, the length of a segment. This is because this element is the ith leader of A which means it is the first element of the ith segment. Therefore, prior to this element, there are i minus 1 full segments within A, they will contribute a total of i minus 1 into L elements. And then comes this element therefore, this is the i minus 1 into L plus first element within A. And the problem instance of which this element is in charge this leader is in charge is going to start from this element on the A side.

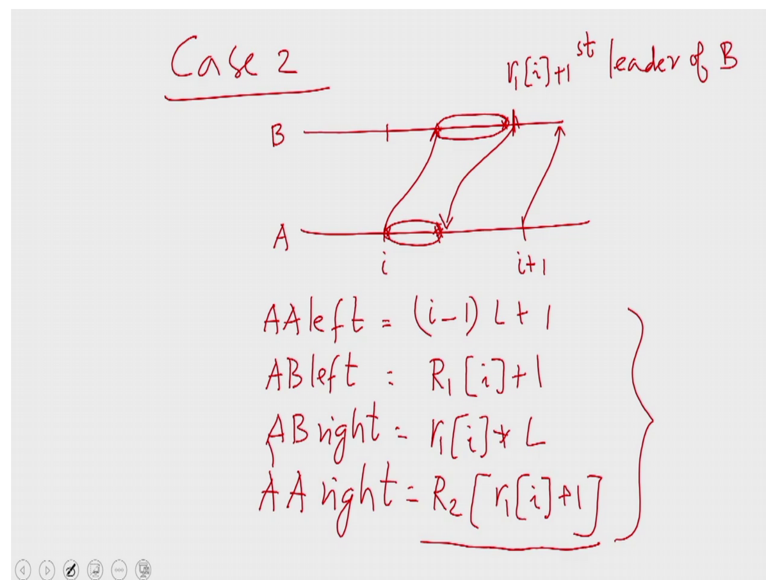
Therefore, we call this A A left the first A is because we are considering a leader on the A side. Then what I call A B left, If the rank of the leftmost element of the circle part. On the other side that is the B side that would then be R 1 of i plus 1 by the definition of R 1, we know that R 1 of i is the rank of i in B that is A number of elements less than or equal to i in B. For now, we assume that all elements are distinct. So, if all elements are distinct

then the leftmost element within the circle part in B would be of index R 1 of i plus 1 on the B side.

Similarly, A A right would be the index of the rightmost element within the circle part on the A side, this would be i into L. That is because in this case it happens to be the i plus first leader on the A side, it is it is an element just before the i plus first leader on the A side. Therefore, its rank is going to be i into L, the last element within the circle part on the right side is going to have a rank of i into L on the A side.

Similarly, on the B side, the last element within the circle part is the last element that is less than or equal to the i plus first leader of A. Therefore, it is going to have a rank of R 1 of i plus 1 so, that was the first case.

(Refer Slide Time: 54:11)



In the 2nd case again, we consider two consecutive leaders on the A side the i th leader and i plus first leader. Let us say both of them do not fall within the same segment on the other side then there is a leader between them on the other side. Consider the leftmost such leader and consider the reflection of that leader on the A side.

So, the elements that the ith leader of A are in charge of would be the circled elements. So, let us consider the boundaries of the circle elements. Let me define A A left and A B left as before, A A left you find this as before i minus 1 into L plus 1, because the

leftmost element is a leader in the circle part of A. Similarly, A B right is R 1 of i plus 1 but pardon me, A B left that was A B left, A B right.

In this case is different from the previous case. In this case, we find that this is R 1 of i into L A B, right is the rightmost element. On the B side within the circle part this is going to be an element just before a leader, but then which leader would that be, we have to consider the number of leaders before the i th leader of A, the i th leader of A has small R 1 of i leaders less than or equal to that on the B side. Now we are considering the next leader here and the reflection of that is going to be the right boundary.

So, the element immediately before that will be the R 1 of i into Lth element that will be the rank of the rightmost element within the circle part on the B side. And finally, we can calculate A A right in this fashion, A A right would be R 2 of R 1 of i plus 1. We want to calculate the index of this element within A the leader this leader is the R 1 of i plus first leader on the B side. The rank of that within A is given by R 2 invoked on that which is what the rank of the rightmost element within the circled part in A is going to be. So, this will define the boundaries of the problem in the second case. So, putting all together, we can now look at the code of the algorithm.

(Refer Slide Time: 57:57)

Merge

MERGE-2

Input: Two sorted arrays $A[1 \dots N]$ and $B[1 \dots M]$; $N \geq M$.

Output: Array $C[1 \dots M+N]$, the merge of A and B. Model: CREW PRAM

{

$L = \lceil \log MN \rceil$; $n = \lceil N/L \rceil$; $m = \lceil M/L \rceil$;

pardo for $1 \leq i \leq n$ $A'[i] = A[(i-1) * L + 1]$;

pardo for $1 \leq i \leq m$ $B'[i] = B[(i-1) * L + 1]$;

} O(1) time

So, here we have an algorithm for merging two arrays, we have two sorted arrays of sizes N and M respectively, capital N and capital M. Let us assume that capital N is greater than or equal to capital M. Then we define L as log of M into N, mind you logarithm of

M into N is the same as log M plus log in which is going to be order of log N when N is greater than or equal to M. So, we are planning to divide the array into segments of size L. Let small n be the number of segments on the A side and let capital small m be the number of segments on the B side.

Then in parallel from the two arrays, we pick out the leaders and populate A prime and B prime. A prime will consist of the first elements from every segment of A and B prime will consist of the first elements from every segment of B and then we cross rank the leaders. Let R 1 and R 2 be the cross ranks.

(Refer Slide Time: 58:49)

```
CROSS-RANK-1(A'[1...n], B'[1...m], r1[1...n], r2[1...m]);        O(log n)
pardo for 1 ≤ i ≤ n
    R1[i] = (r1[i] - 1) * L+ SEARCH-2(B[(r1[i] - 1) * L + 1 ... r1[i] * L], A'[i], 1);   } O(log n)
pardo for 1 ≤ i ≤ m
    R2[i] = (r2[i] - 1) * L+ SEARCH-2(A[(r2[i] - 1) * L + 1 ... r2[i] * L], B'[i], 1);
pardo for 1 ≤ i ≤ n
{
    AAleft[i] = (i - 1) * L + 1;   ABleft[i] = R1[i] + 1 ;                               } O(1)
    if (r1[i] == r1[i + 1]) { AAright[i] = i * L;   ABright[i] = R1[i + 1] ; }
    else { ABright[i] = r1[i] * L;   AAright[i] = R2[r1[i] + 1] ; }
}
pardo for 1 ≤ i ≤ m
{
    BBleft[i] = (i - 1) * L + 1;   BAleft[i] = R2[i] + 1 ;                               } O(1)
    if (r2[i] == r2[i + 1]) { BBright[i] = i * L;   BAright[i] = R2[i + 1] ; }
    else { BAright[i] = r2[i] * L;   BBright[i] = R1[r2[i] + 1] ; }
}
```

Now every leader on the A side knows its segment within the other side. Now in parallel for every leader on the A side, we search for that leader on the other side. So, A prime of i is searched for within the corresponding segment in B and R 1 of i minus 1 into L is the offset which is to be added to the result of the search that will give us the rank of A prime of i in B that is, is going to be R 1 of i in and exactly analogous manner, we can also calculate R 2 of i.

Now, what remains is to calculate the problem boundaries we calculate A A left A B left A A, right and A BB, right exactly the way, we have mentioned. Now the other four values B B left B A left B B, right and B A, right for the elements of B will be calculated analogously, for the leaders of B will be calculated analogously.

(Refer Slide Time: 59:57)

Merge (Cont'd)

$O(\log n)$

pardo for $1 \leq i \leq n$

MERGE-SEQ($A[AAleft[i] \ldots AAright[i]], B[ABleft[i] \ldots ABright[i]]$,

$C[AAleft[i] + ABleft[i] - 1, \ldots AAright[i] + ABright[i]]$);

pardo for $1 \leq i \leq$ ~~nBL~~ m

MERGE-SEQ($A[BAleft[i] \ldots BAright[i]], B[BBleft[i] \ldots BBright[i]]$,

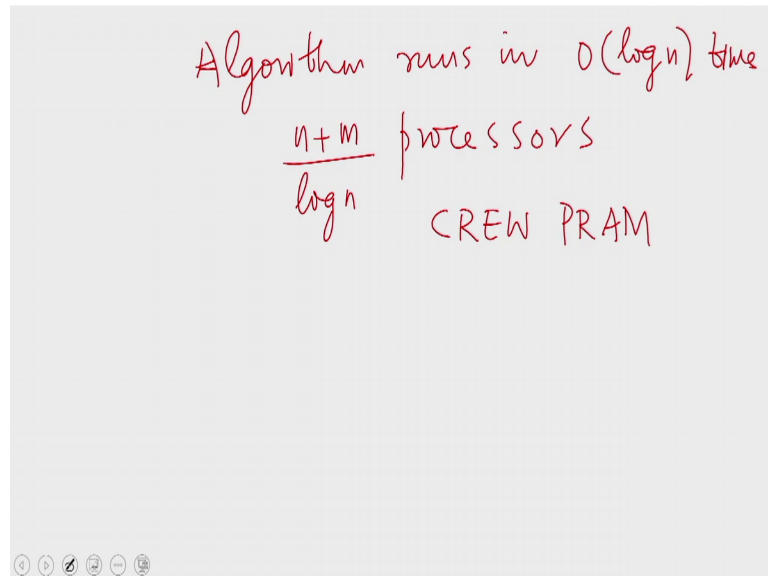
$C[BAleft[i] + BBleft[i] - 1, \ldots BAright[i] + BBright[i]]$); $O(\log n)$

}

So, once we have the boundaries of the problems we can invoke the sequential merge algorithm for the sub problems that is we merge the elements that fall between A A left of i and A A right of i. On the A side and A B left of i and A B right. If i on the B side and put those elements in their appropriate places within C which will be A A left of i plus A B left of i minus 1 to A A right A phi plus A B, right of i that is where these elements should go.

So, from the A A left, A A right, A A left, A B right values we can calculate the exact places where the elements should go within the array C. Similarly, we do this for every single element of B as well so, combining all of them what we find is that the algorithm runs in order of log n time.

(Refer Slide Time: 60:47)



Let us do a recap to see if the algorithm indeed runs in order of log n time picking out of the leaders will take only order one time assuming that we have exactly as many processors as there are leaders. So, these two steps will run in order of one time each cross ranking will take order of log n time.

Since, we have exactly as many processors as there are elements in the arrays and then these searches, these are sequential searches, this would take order of log n time. Again, these are sequential searches, because we are using one processor for these searches. And then calculating of the boundaries will take order one time. All these calculations are done in parallel for every single processor and then finally, we merge the sub instances sequentially each will take order of log n time. Here we do in parallel for every single element on the A side every single leader on the A side.

Similarly, here we do this for every single leader on the B side which would again take order of log n time. Since, every leader is operating in parallel the time taken is order of log N. And concurrent read has been used in multiple places, but concurrent write has not been used anyway. So, the overall time complexity of the algorithm is order of log n and we have used n plus m by log n processors and the model used is the CREW PRAM.

In the next lecture we shall see how to improve the time complexity of the algorithm from order of log n to order of double log n which is a substantial improvement you on

the same model that will again be another divide and conquer algorithm. So, that is hope to see you in the next class.