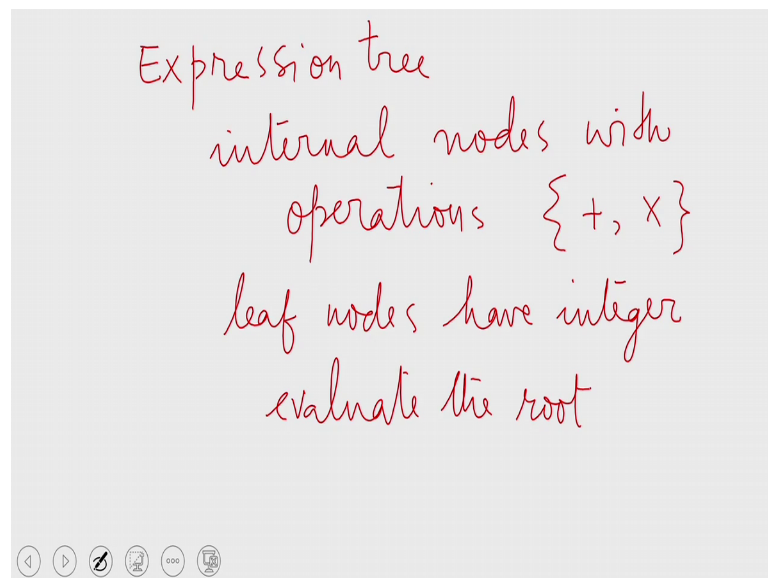
**Parallel Algorithms**
**Prof. Sajith Gopalan**
**Department of Computer Science & Engineering**
**Indian Institute of Technology, Guwahati**

**Lecture – 17**
**Fast optimal merge algorithm**

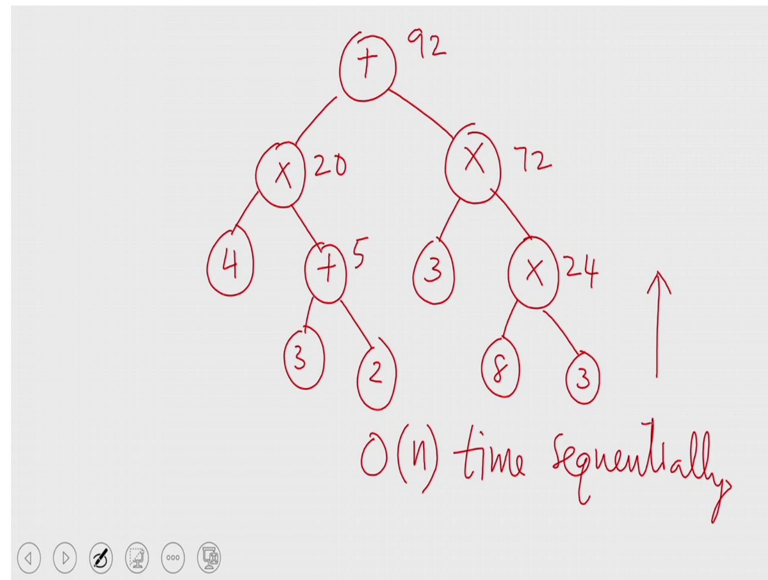
Welcome to the 17th lecture of the MOOC on Parallel Algorithms. In the 16th lecture, we were discussing an algorithm for evaluating an expression tree we suggested an algorithm that uses tree contraction. In tree contraction we use an operation that is called a rake, a rake operation is where you identify a leaf and remove the leaf along with its parent from the tree, while making the sibling of the leaf a child of its grandparent from the same side in which the parent of the vertex was a child of the grandparent. So, using this rake operation repeatedly we can reduce a tree into a tree involving just 3 nodes the root and 2 children.

(Refer Slide Time: 01:25)



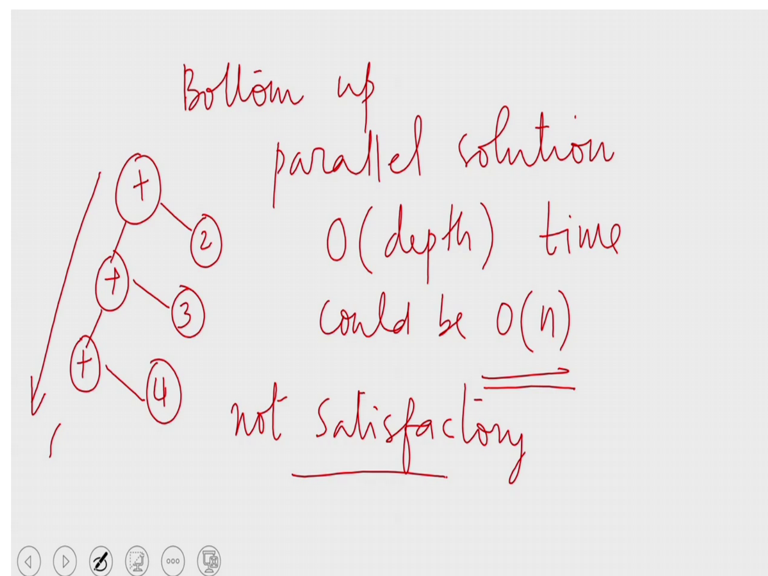

Now, let us see how this tree contraction algorithm can be used for evaluating an expression tree. An expression tree as we saw earlier, consists of internal nodes with operations in them. If we consider only 2 operations; for now which can be easily extended to a richer set of operations, but for now we assume that there are multiplication and addition and no other operation. And then the leaf nodes have integer constants, what we need to do is to evaluate the root.

(Refer Slide Time: 02:23)




For example, if you have a tree of this form, the tree will be evaluated bottom up at this node we will have a value of 24 8 into 3 and then when that is multiplied by 3 we have 72. Here we have 3 plus 2 5 at this node, 5 into 4 20 here and then at the top of the tree we will have the sum of 20 and 72 which is 92. So, the tree can be evaluated in this fashion. So, we said that sequentially solving a tree bottom up requires linear time.

(Refer Slide Time: 03:44)

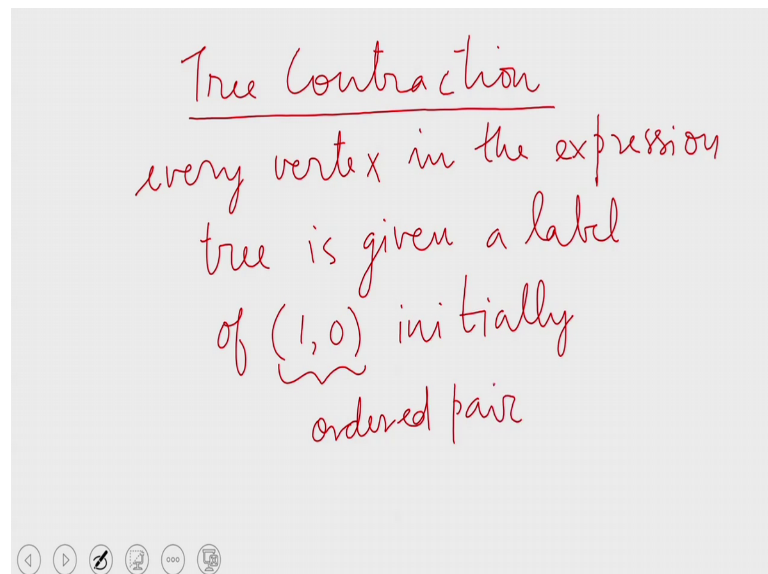



So, the cost of the problem in the sequential setting is order of n time but in parallel, if you go bottom up the algorithm is going to take time that is of the order of the depth of

the tree which could be order of n if the tree happens to be very skewed. For example, if you have a tree of this form we have a skew tree. So in this case the sequence of these nodes the spine of the tree could be of length order of n therefore, the depth of the tree is order of n.

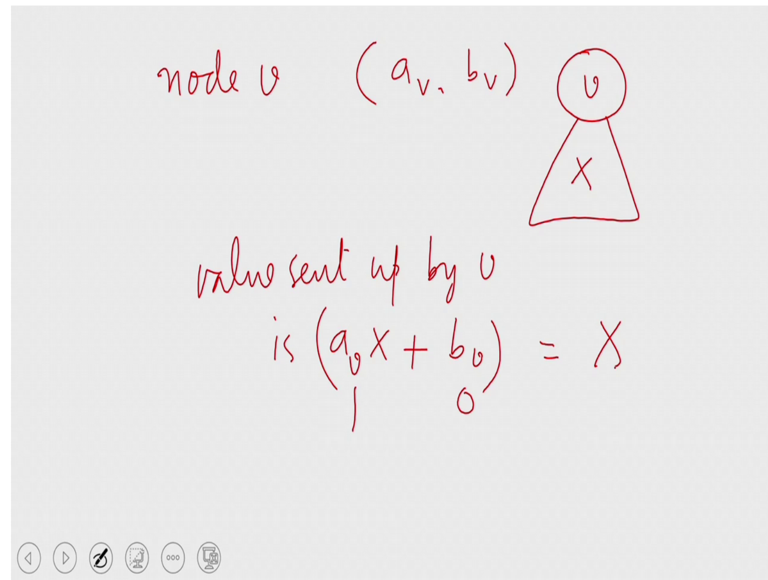
Therefore, evaluating this tree could take order of n time. Therefore a parallel solution which takes order of depth time is not satisfactory. We want an algorithm that runs faster, so according to our conventions an algorithm is efficient precisely when it runs in poly logarithmic time.

(Refer Slide Time: 05:08)



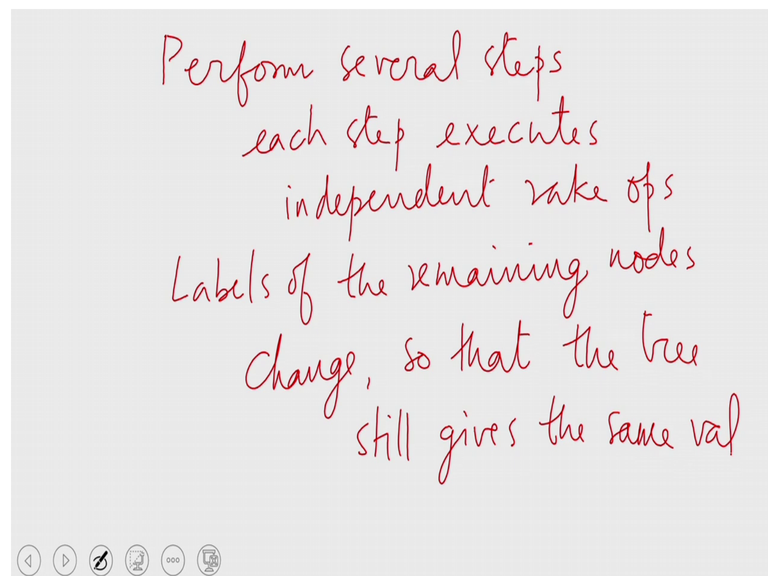

So, we are going to find the solution that uses tree contraction; for this purpose we assume that every vertex in the expression tree is given a label of 1 0 initially. So, every label that we are going to use is an ordered pair of integers, so the label is an ordered pair. And as the algorithm proceeds the labels will keep changing. Our idea is that when a node v with label a v b v evaluates to X that is the sub tree which is rooted at the node evaluates to X let us say.

(Refer Slide Time: 06:05)



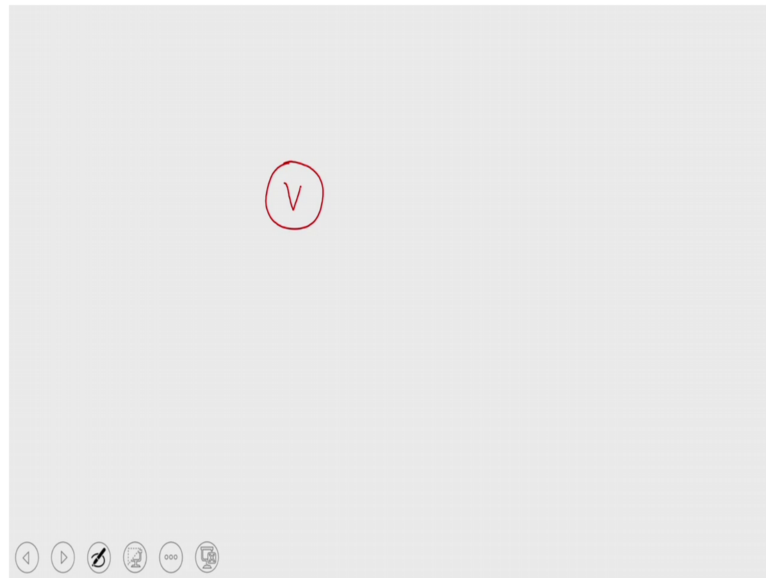

Then the value sent up by the node the value sent up by the node to its parent is a v times X plus b v. If the subtree which is rooted at v evaluates to X then node we will send a value of a v X plus b v upwards to its parent. Initially when a v equal to 1 and b v equal to 0 the value that is being sent up as X; so, if the label of every single node is 1 0 then every node will report its subtrees value truthfully upwards. And therefore, the tree will evaluate to precisely the value that we want which will be the value of the expression.

(Refer Slide Time: 07:15)



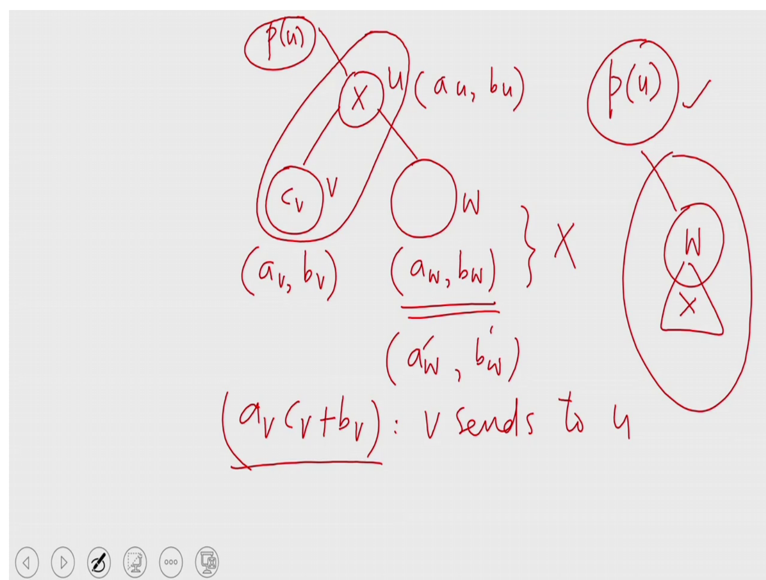
Now our idea is to perform several steps, so that each step executes a number of a independent rake operations. When the rake operations are performed we change the labels of the remaining nodes, but these are changed in such a way that the tree still evaluates to the same value. So, let us see how this is possible.

(Refer Slide Time: 08:27)




So, let us say we are identifying a node v for raking let us say we identify a node v for raking.

(Refer Slide Time: 08:37)



Suppose the leaf v contains a constant c v and let us say the ordered pair or corresponding to this leaf is at the moment a v b v. Let w be the sibling of c v; let us say the label of the sibling is a w b w. Let u be the parent of b and let us say u contains the multiplication operator. The label of u is a u b u let us say and of course, above u we have the parent of u. So, we are planning to rake v along with u v and its parent u are being raked. So, when we do this we will have w becoming a child of b of u. So, when we perform a rake operation on v we will be removing both v and its parent from the tree.

We will have w made a child of p of u then we propose to change the label of w so that the value that is send up by node u to its parent still remains the same. So, let us see if we can find new constants a w prime b w prime ensure that the same value will be send up to the node p u. So, first let us evaluate what is the value being sent up to p u in the original tree. The node v evaluates to c v therefore, the value that node v sends up to node u is a v times c v plus b v this is the value that v sends to u.

(Refer Slide Time: 10:38)

w sends a value of $a_w X + b_w$ to u.

$$a_u[(a_v c_v + b_v) * (a_w X + b_w)] + b_u$$

$$= a_u (a_v c_v + b_v) a_w (X) + a_u (a_v c_v + b_v) b_w + b_u \quad \Big\} \; a'_w X + b'_w$$

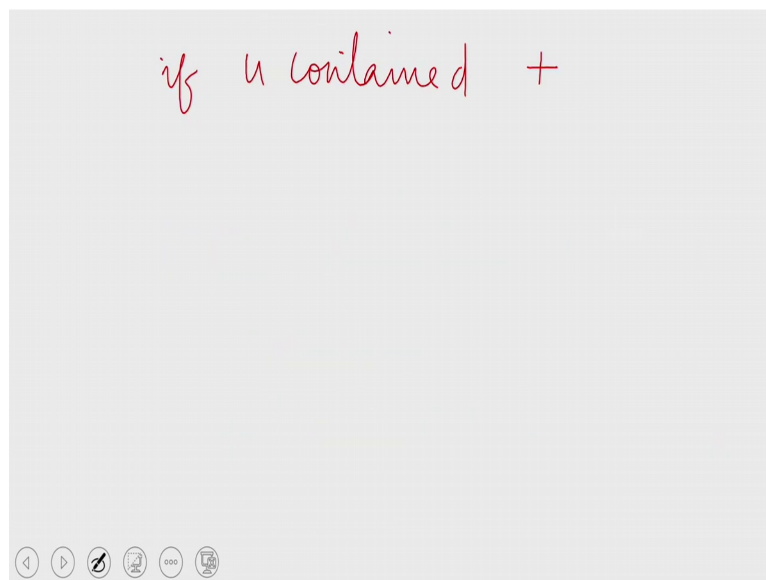
Let us say w sends a value of a w X plus b w to u. What we assume is that the subtree which is rooted at w evaluates to X; the subtree which is rooted at w evaluates to X then w will send a value of aw X plus b w to its parent u. So, u gets 2 values it gets a v c v plus b v from b and aw X plus b w from w and u contains a multiplication operator. So, you will multiply these 2 values. This is the value that the node generates, and then the

value that the node u will send up to its parent p of u would be a u multiplied by this plus bu.

But this value can be written as au times a v c v plus b v times a w times X plus au times a v c v plus b v times bw plus b u which is of the form a w prime X plus b w prime. So, this is the value that u sends up to w. So, now what we propose to do here is to replace u as well as v with w w will be the new right child of p of u in this figure. And we know that the subtree which is rooted in w evaluates to X.

Therefore if we make sure the w sense of the same value that u was sending up to p of u earlier, then p of u will not be aware of any change in this part of the subtree it will still be getting the same value from its right child. In this case had u been on the left side then it would be getting the same value from the left child. Therefore, what we have shown is that, when the operation that u contains is a multiplication then there X is a w prime and b w prime. So, that if the label of w is changed from a w bw to a w prime b w prime. Then w would be sending up exactly the same value that u had been sending up to p of u earlier.

(Refer Slide Time: 14:08)

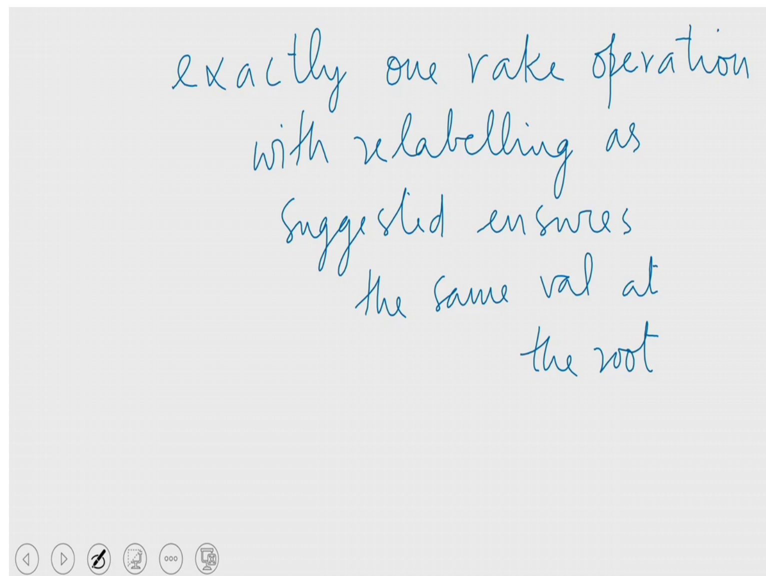



Therefore, as far as the root of the tree is concerned this rake operation will not make any difference to it will still be evaluated to the same value. On the other hand if u contained the addition operation, then we would have a plus here instead of addition at this point

we would be combining the values of b as well as w by addition and then that result would be multiplied by a u and scaled by an and translated by b u before being send up.

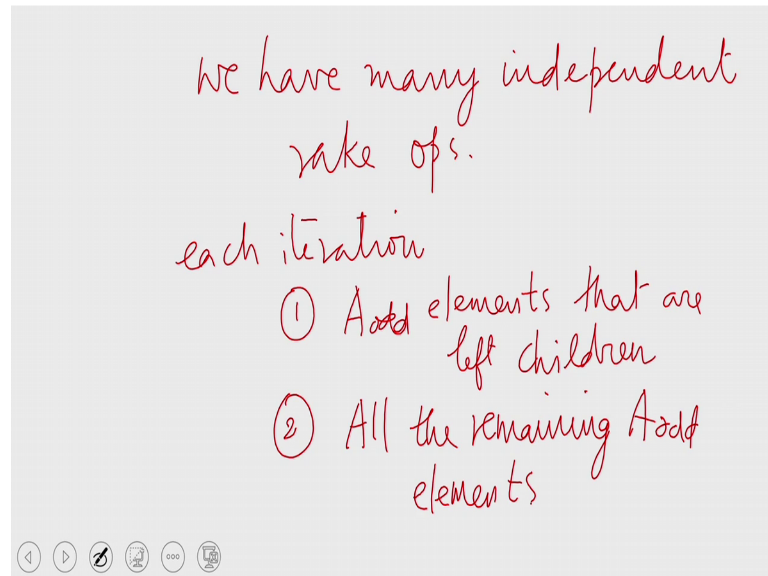
The only difference is that the multiplication here is replaced with addition, and then if you look through the expression you can find that we can still find a w prime and b w prime. So, that a node w can be replaced with the label of node w can be replaced with a w prime B prime so, that the value which reaches p of u is still exactly the same. So, what we have shown is that irrespective of whether u contains multiplication or addition there exist labels a w prime b w prime which upon raking u and v from the tree and useless reliables for w will ensure that the root still evaluates to the same value.

(Refer Slide Time: 15:27)




So, this is what is happen this is this is what happens when we perform exactly 1 rake operation So, what we have shown is that exactly 1 rake operation with re-labels as suggested ensures that the route evaluates to the same value as before, but then it should be the same even if we perform multiple rake operations

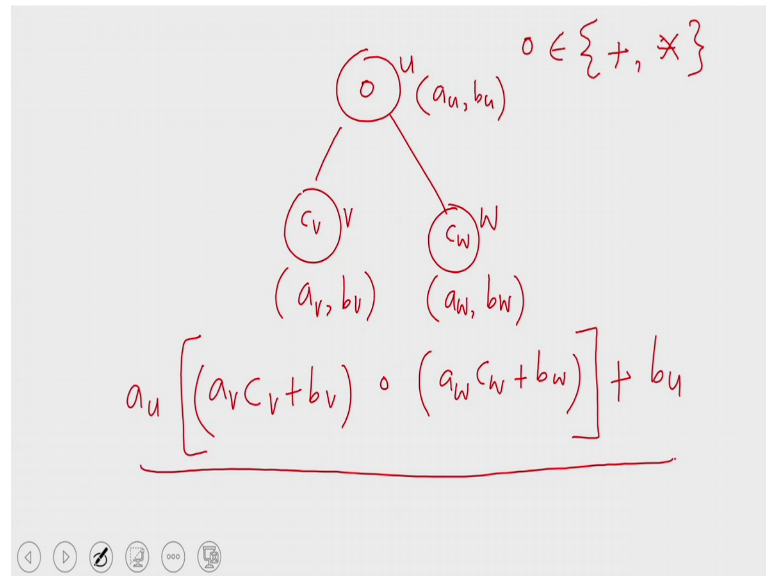
(Refer Slide Time: 16:16)




Let us say we have many independent rake operations. Independent in the sense that these rake operations do not interfere with each other, in the pre-contraction algorithm that we saw in the previous lecture we were scheduling a set of rake operations no 2 of them interfered with each other. So, if we have such a set of rake operations then if we make sure that the rake labels I mean the rake leaves siblings labels are updated in the suggested manner, then the value which is obtained at the root will still remain the same.

Therefore we can extend what we have been talking about to a set of independent rake operations as well. So, in the tree contraction algorithm that we had we scheduled 2 such sets of independent rake operations in each iteration; in each iteration first we had all a odd elements that are left children right. Simultaneously then in the second phase we had all the remaining a odd elements rake, each set as an independent set in the sense that 2 rakes of the set will not interfere with each other. Therefore after each of those parallel rake operations the value of the tree will remain unchanged.

(Refer Slide Time: 18:23)

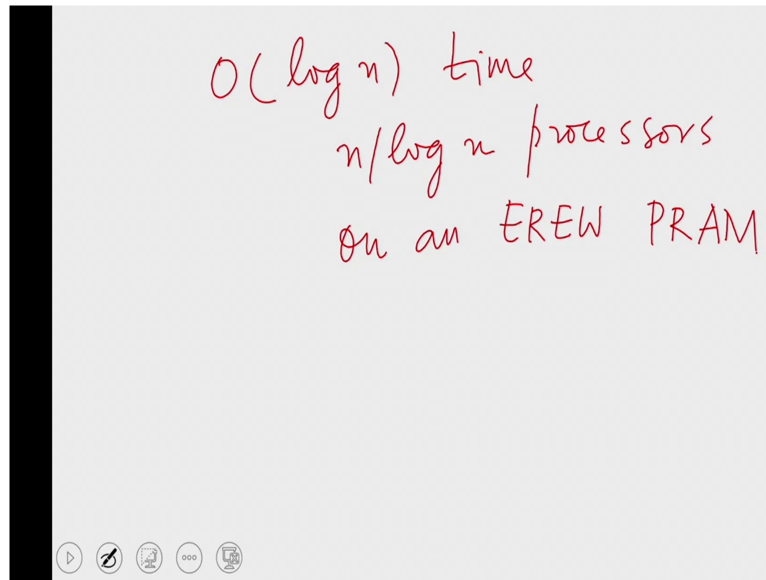



Therefore, if we begin with the tree and perform a series of rake operations, until the tree it reduces to a tree of this form with 2 leaves v and w with the root at u with a v b v as the labels at v and a w b w as the level at w with c v and c w as the values at v and w respectively. And some operation at u the operation at the root is either addition or a multiplication.

So, once the tree has reduced to such a tree through the where the value that the root will provide is guaranteed to be the value that the original tree would have produced. Then that value can now be calculated by evaluating the leaves. So, the leaf v will evaluate to c v and it will read up it will send up a value of a v c v plus b v to its parent and w will send up a value of a w c w plus b w to its parent, and then these 2 will be combined by u using the operation which is present there

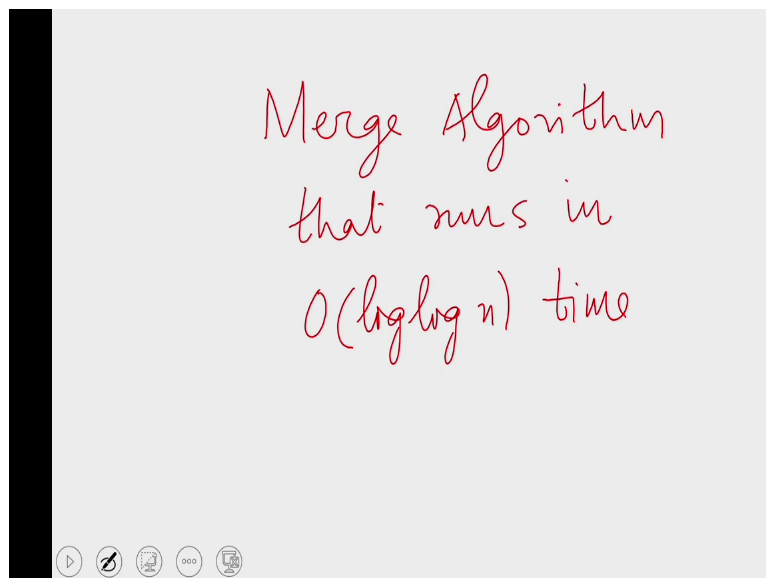
If a u b u is the label of u then the entire value of the tree will be this. So, the algorithm for expression tree evaluation now amounts to this take the tree which is given contract the tree through a series of rake operations, adjusting the labels all the way as we have suggested and then finally, when the tree has reduced to a tree involving just 3 nodes with 1 root and 2 children. At this point evaluate the tree using 1 single expression which can be evaluated in order 1 time.

(Refer Slide Time: 20:31)




Therefore the entire expression can be evaluated in order of log n time, using n by log n processors on an EREW PRAM. So, those were some of the applications of the optimal de strangling algorithm, and in each of the applications we found an algorithm that runs in order of log n time using n by log n processors on an EREW PRAM. So now, we move on to the next topic.

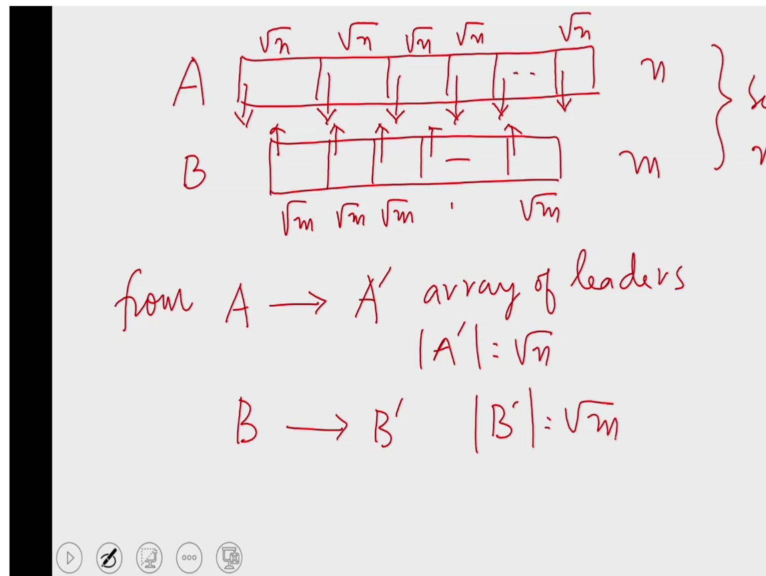
(Refer Slide Time: 21:13)




The next topic is a merge algorithm, that runs in order of log of log n time, this merge algorithm is the 1 that I had promised you at the end of the 8th lecture. In the 8th lecture

we had seen an optimal merge algorithm that runs in order of log n time using n by log in processors for merging 2 arrays of size n and m respectively. Where n is greater than m here we are improving the running time to order of a double log n double log n is a considerable improvement over the log n of that algorithm, we maintain optimality that is the new algorithm is going to be still optimal.

(Refer Slide Time: 22:20)




So, now let us see how we will device this algorithm this algorithm has a structure which is very similar to the order log n time algorithm that we have seen earlier. So, let us say we are given 2 arrays of size n and m respectively. We have an array a of size n and we have an array b of size m both are sorted, and we assume that n is greater than m this is again a divide and conquer algorithm what we do here is to divide the array A into segments of size root n each.

The second array is divided into segments of size root m each. And then from each segment we pick out the first element as a leader all this is exactly as we had seen earlier from every segment we pick out the first element as a leader. So, from A we get a leader array A prime which is an array of leaders from A and the size of A prime is square root of n. Similarly, from array B we get an array of leaders B prime. So, that the size of B prime is square root of n.

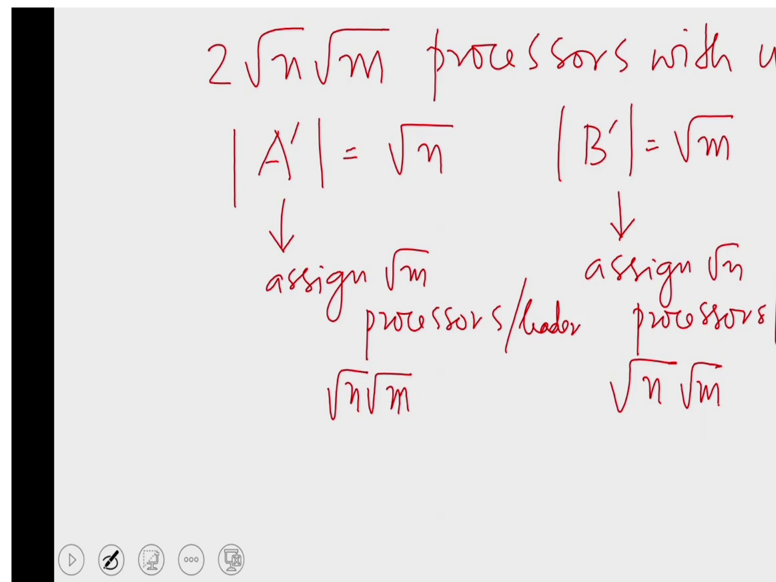
(Refer Slide Time: 24:17)

Here we assume that we have n plus m processors later on we shall see how to reduce the number of processors. So, to begin with we assume that we have n plus m processor, but then recollect that square root of n into square root of m is less than n plus m. That is because 2 n times m is less than n plus m the whole square that is because for this inequality.

The right hand side evaluates to m squared plus 2 n m plus m square, if you cancel 2 nm from both sides you have that 0 less than n squared plus m square which is indeed the case. The right hand side here is a sum of 2 squares therefore, it is indeed greater than 0 now what we have shown is that 2 n m is less than n plus m the whole squared if and only if 0 is less than n squared plus m square which is indeed the case.

But then n m is less than 2 n 2 n m therefore, nm is less than n plus m the whole square. If you take the square root on both sides we have what we want square root of n into square root of m is less than n plus m. So, in particular we have 2 into square root of square square root of n into square root of m processors with us.

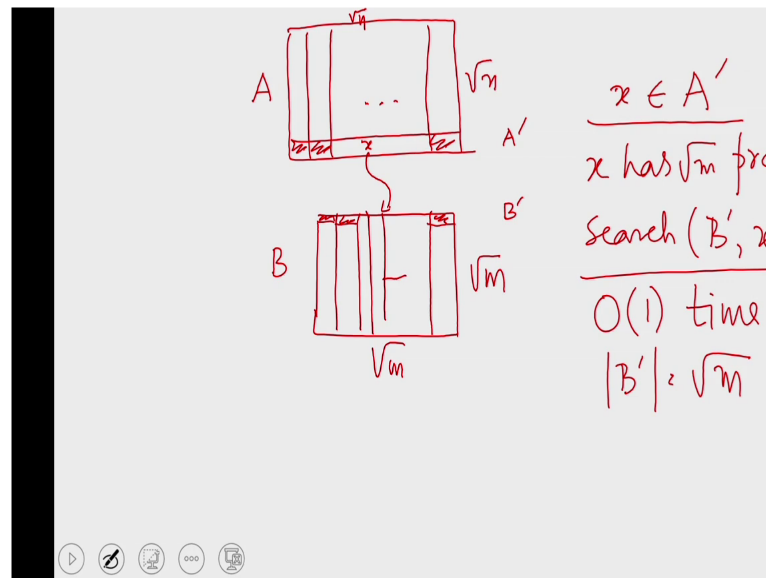
(Refer Slide Time: 25:55)




We have square root of n leaders from A and square root of m leaders from B. Let us say to each leader of A prime we assign root m processors to each leader of B prime we assign root n processors which means on the A prime side we require order of n into square root of n into square root of m processors.

Here also we require square root of n into square root of m processors we require a total of twice square root of n times square root of m processors, which we do indeed have we have n plus m processors which is more than this. So, we have enough processors to allocate them in this manner. So, once we have allocated the processors.

(Refer Slide Time: 27:22)

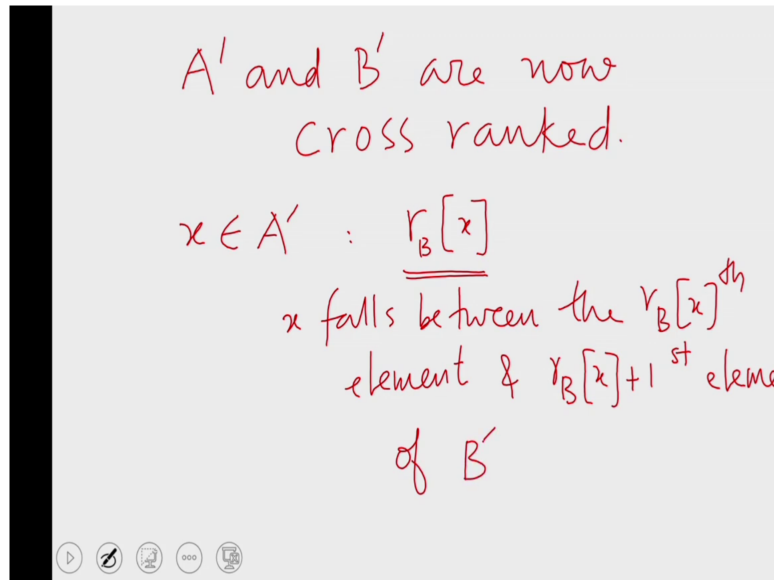



What we do is this let me visualize the 2 arrays A and B as 2 2 dimensional arrays A is a root n by root n 2 dimensional array. B is a root m by root m 2 dimensional array. Let us consider the segments of a each segment has a size of root n and the segments of b have a size of root m each. What we have done is to pick out the first element of each segment as a leader. So, let us assume that the leaders of b are in the top row facing them off are the leaders of A the first element of every segment of A is also picked out as a leader.

So, there are root n leaders on the A side and root n leaders on the B side. So, the first row here will form the array A prime and the top row here will form the array B prime. So, let us say the leaders are facing of each other, the root m leaders on this side have root n processors each with them and the root n leaders on the other side have a root m processors with them. Now consider a leader on the A side we consider X belonging to A prime X has root m processors, with root m processors X can search for itself within B.

Within B prime we look for X this search can be executed in order 1 time that is because B prime has a size of root m and that is exactly the number of processors that X has. So, X has exactly as many processors as the search space sizes therefore, the search can be done in order 1 time. So, the meaning of what I say is that every element in A prime can find its rank in B prime in order 1 time. Similarly, every element in B prime can find its rank in A prime also in order 1 time. So, the leaders have found their ranks in each other

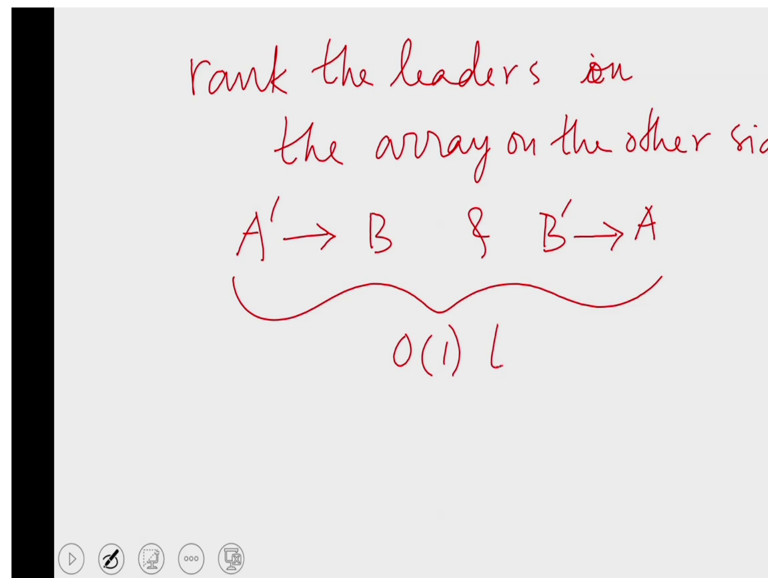
(Refer Slide Time: 30:05)




Then or in other words A prime and B prime are now cross ranked. Once again consider an element x belonging to A prime: let us say the rank of x in B prime is found. So, r of x let us say is the rank of x in B prime for clarity I can write r b of x to signifies that to signify that it is the rank of x in B prime which means these many leaders in B are less than or equal to x or it means x falls between the r B of x th element and the r B of x plus first element of B prime.

In other words x has found the column on the other side in which it falls. So, the leader x that I look I have taken on this side, has found that it falls within a particular column. So, every leader on the A side has found the column in which it belongs on the other side, similarly every element on the B side has found the column on the A side in which it belongs.

(Refer Slide Time: 31:57)

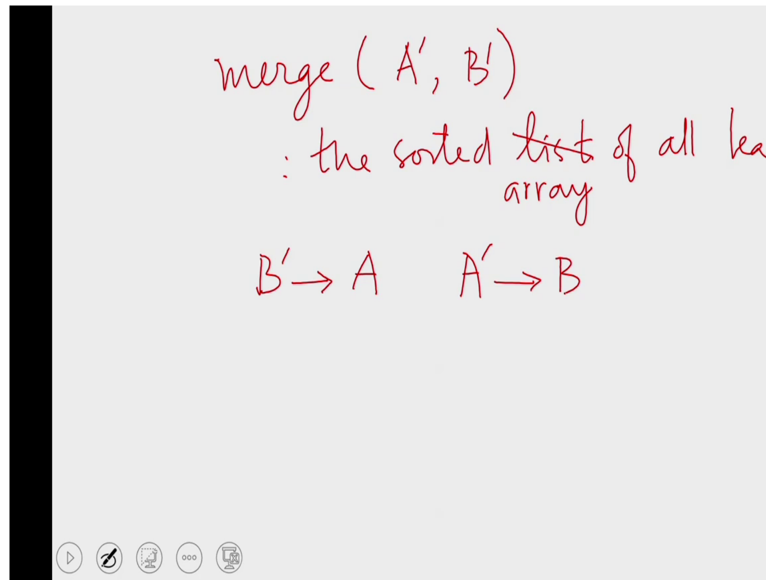



Now, the next step is to rank the leaders in the array on the other side, rank the leaders in the array on the other side which means we want to rank A prime into B and B prime into A. Every leader in B should know its exact rank in A and every leader in A should it should know its exact rank in B.

But this is now easy because every leader on either side has found the column in which it falls on the other side. Once again visualize the arrays as 2 dimensional arrays square arrays every element every leader on 1 side knows the column in which it falls for example, this x knows that it falls within this column. Now what is the size of this column, that is square root of m and how many processors does x have, x has root m processors with root m processors we can search for x within this column, and thats how search again will execute in order of 1 time.

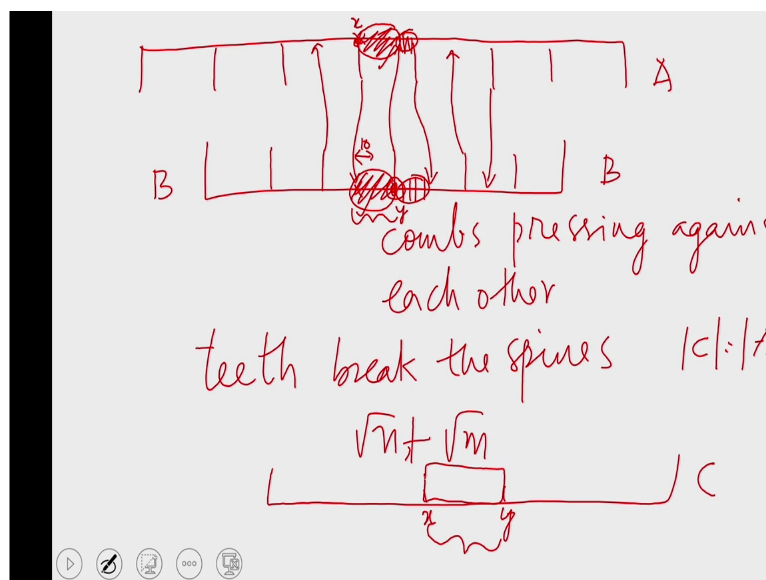
Similarly, every leader on the B side can search for itself within the correct column on the other side in order 1 time, because every column on the A side has a size of root n and every leader on the B side has root n processors at its disposal. Therefore, this search also will finish in order 1 time therefore, these 2 ranks can be found in order of 1 time.

(Refer Slide Time: 33:54)




So, now we have accomplished 2 things one is that, the merge of A prime and B prime: which is the sorted list of all leaders is obtained. This we have achieved in order 1 time the moment we cross ranked the leaders we have merged them. So, we have the sorted list of all leaders. So, with 1 processor per leader we can form this array easily. So, this is in fact an array not a linked list. So, to be precise I should say this is a sorted array of our leaders, similarly we also have found these ranks. Every leader on the A side has been ranked into the B side and every leader on the B side has been ranked into the A side.

(Refer Slide Time: 34:58)

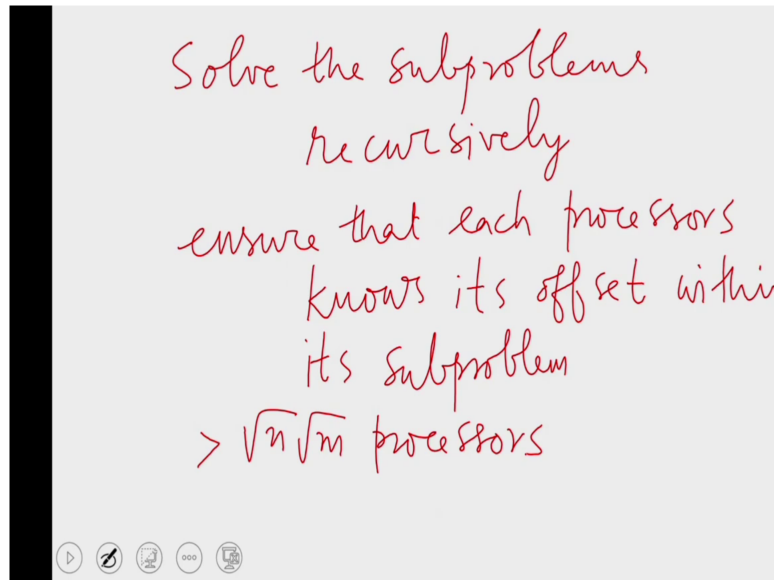


So, all this we have achieved in order 1 time. So, now let us see where this will take us. So, we have the 2 arrays A and B the leaders know their exact positions on the other side. So, this is analogous to 2 combs pressing against each other. Let us see the teeth break the spine on the other side, therefore, if you consider 2 consecutive leaders in sorted order. So, this leader and this leader are consecutive in sorted order. If you consider the 2 segments formed by these 2 leaders on either side, this is 1 segment and this is the other segment.

All the elements belonging to these circled segments will be strictly between these 2 leaders. So, you can leave all of these elements in charge of the smaller leader. So, what it means is that every single leader is now getting several elements in its charge. But then what is the maximum this could be analyzing this exactly as we did in the previous algorithm we find that the maximum size of these segments would be root n on 1 side and root m on the other side. So, you can have at most root n plus root m elements in charge of every leader. More over what we know is that the elements falling, here that is in charge of a particular leader are strictly less than every element that is falling in charge of a smaller or the in charge of a larger leader.

For example, this leader is getting all these elements, these elements are strictly larger than these elements, in a more precise I should say this elements that is the vertically hashed elements are strictly larger than the slanderdly hashed elements. So, this ensures that the merging problem is now decomposed into several sub problems, which are all independent of each other then all we have to do is to sort the to solve these sub problems and place them in their exact place in the output.

(Refer Slide Time: 38:01)



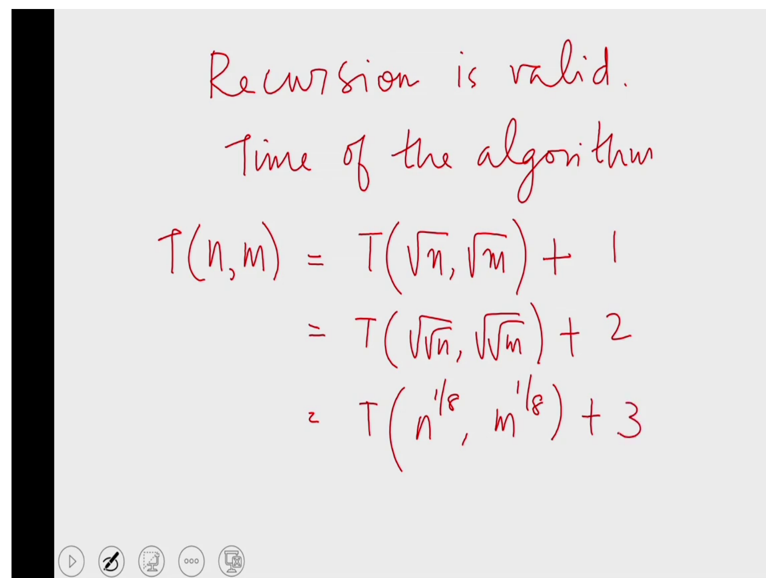

But then how do we solve these sub problems. In this case we solve the sub problems, recursively this is the departure of this algorithm from the previous algorithm here the sub problems are solved recursively. But, to solve this a problems recursively how many processors will we need for the global instance we assume that we have exactly as many processors as the total size of the problem. Therefore, for every sub problem also we should provide exactly as many processors as there are elements in it.

But then if we make sure that the processors are distributed 1 per element, then when the sub-problems are formed each sub problem will have exactly as many processors as there are elements in it, therefore that problem is easy to solve. The next bottleneck is to ensure that, each processor knows its offset, but then it is a problem what I mean is this in the global instance we have these 2 arrays A and B and we have 1 processor per element.

So, the ith processor on the A side is sitting on the ith element of a and this processors aware that it is the ith processor on the A side. Similarly every processor sitting on the B side it is also aware of its offset within B for the recursion to hold good the same level of awareness should be there for every processors deputed to the sub-problems. But, how do we ensure that this is easy to ensure because, we have more than square root of n times square root of m processors.

So, every leader on the A side every leader on the B side can be given root m processors these root m processors can go to these elements on the other side, which are in charge of this leader and inform those elements that they are left in charge of this leader on the A side. Then all these leaders will also get to know they are offsets. For example, and an element which is the 10th from here will be informed by the 10th processor here that it is the 10th on the other side. And the maximum offset on the other side is going to be utmost root m and exactly root m processors are available with the leader here.

(Refer Slide Time: 41:18)




Therefore, we have enough processors to perform this in informing. Similarly, we will have the leaders on the B side in forming the elements on the A side as well. Therefore, we have the perfect setting for recursion. Therefore recursion is valid. And after the problem is recursively solved, that is after the recursive merges are performed the results have to be written in the in an array C. So, that the size of C is the sum of the sizes of A and B. Now the question is how do we form C? Every leader knows its rank in its own array and as and the rank in the other array.

Therefore, it knows the total number of elements less than or equal to this which means each leader here consider leader x here, this leader knows that it belongs to some particular location in C. And it also knows the location of its next leader suppose the next leader is y. So, it knows that y occurs here. So, leader x knows that, all the elements in its

charge should be returned between x and y in the final array. And every element in its charge has been informed of its offset on its own side.

Therefore once the finals merge solution is done, for leader x all that these processors have to do is to write the results within this array. That is in the recursive invocation the argument corresponding to C will now be the argument it will now be this particular sub array the sub array between x and y in C therefore, we conclude that the recursion is valid.

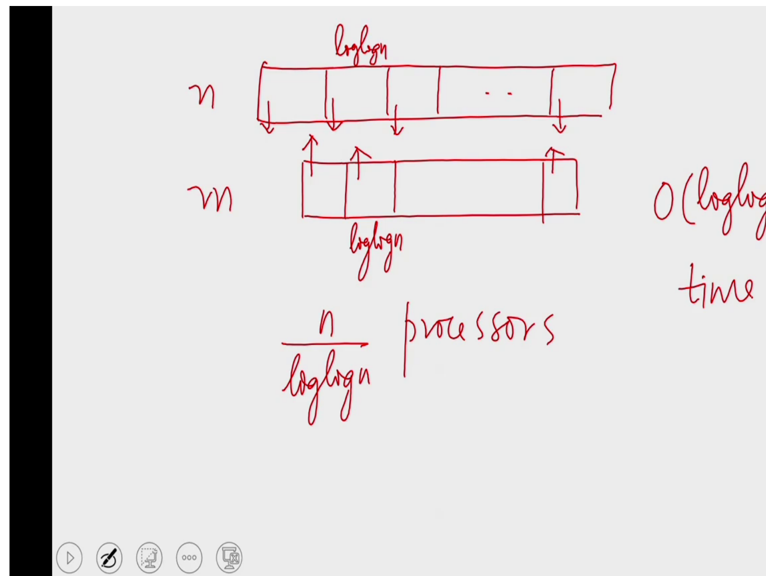
So, now let us see what is the running time of the algorithm, let t of n m denote the running time of the merge algorithm where A has a size of n and B has a size of m. We find that, we have a recursive calls of sizes square root of n and square root of m at the most. And these recursive calls all have been formed in order 1 time. So, ignoring the constant of proportionality, I can write the recurrence relation in this fashion T of n m equals T of square root of n square root of m plus 1.If you unroll this, you find that this is which is the same as T of n power 1 by 8 m power 1 by 8 plus 3.

(Refer Slide Time: 44:24)

So, if you continue like this we get T of n power 1 by 2 power k m power 1 by 2 power k plus k. So, it is easy to see that k is order of log of log n. So, this is an algorithm that runs in order of log of log n time, and the model used is the CREW PRAM because the search algorithm that we are using. Require concurrent reads, so the algorithm runs in order of log of log n time using n plus m processors.

This is a significant improvement over the order of log n time algorithm that we saw in the 8th lecture. But then this algorithm is not optimal, but then if you would recall that in the 8th lecture also first we designed a suboptimal order of log n time algorithm, that ran with n plus m processors and then later on improved that to an optimal algorithm that runs in exactly the same amount of time asymptotically.

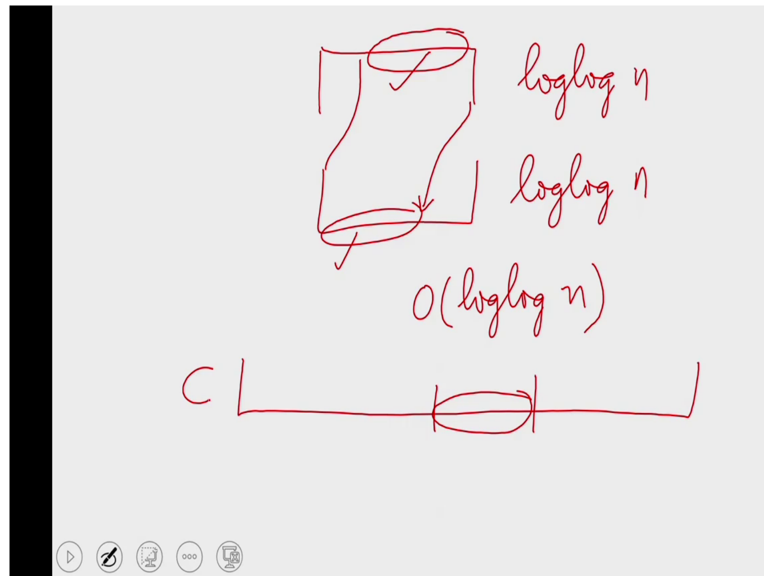
(Refer Slide Time: 45:44)



So, here also we can use exactly the same technique given 2 arrays of size n and m we will divide the arrays into segments of size double log n each, assume that we have n by double log n processors. From each segment pick out the leaders, but when we pick out the leaders we find that the number of leaders is exactly equal to the number of processors we have. Therefore, the merging of the leaders can be done using the algorithm that we have seen just now. Therefore, the cross ranking of the leaders is done in order of double log n time.

Now, every leader on either side knows the segment on the other side, in which it belongs, but the size of the segment is only double log n. So, if you have 1 processor per leader, the leader can search for itself sequentially in the segment in which it belongs on the other side, and it can find its location in double log n time. Therefore, in order of double log n time we managed to find the exact rank of every leader on the other side. Now, again exactly as before the problem breaks into several sub problems, but each sub problem now has a size of double log n each.

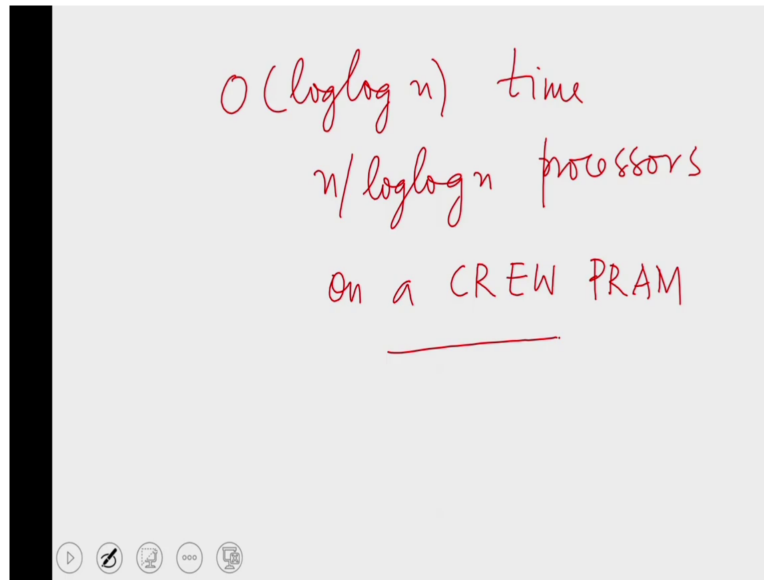
(Refer Slide Time: 47:30)




The segments on either side or double log n sized, and once the leaders find their ranks in exact ranks on the other side, the sub problems that we create will be of size double log n each on either side. These sub problems can be solved in order of log n order of double log n time using a single processor. That is because these sub problems have a size of double log n time.

So, sequentially they can be solved in double log n time and once they are solved all you need to do is to write them into the appropriate offset within the target array C which also can be done in order of double log n time with 1 processor.

(Refer Slide Time: 48:32)




Therefore, what we find is that we can merge 2 arrays of size n and m respectively where n is greater than m in order of log of log n time using n by log of log n processors, on a CREW PRAM. This is a significant improvement over the algorithms that we saw in the eighth lecture.

Now, in the next lecture we will use this algorithm to find an optimal algorithm for sorting which would run in order of log n double log n time. And would have a cost of order of n log n there by being optimal, and then later we will show how to find an optimal sorting algorithm that runs in order of log n time that is the famous Coles merge sort. So, that is our agenda for the next lecture. So, that is all from this lecture hope to see you in the next lecture.

Thank you.