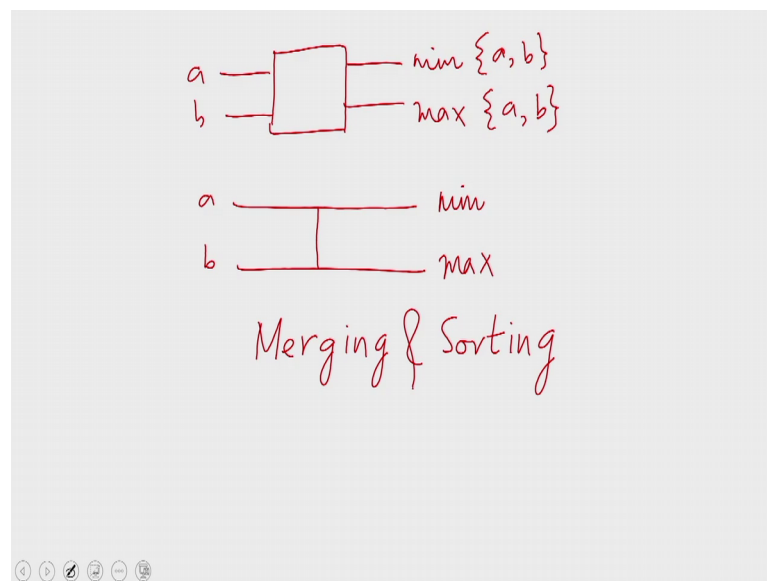


**Parallel Algorithms**  
**Prof. Sajith Gopalan**  
**Department of Computer Science & Engineering**  
**Indian Institute of Technology, Guwahati**

**Lecture – 10**  
**Odd Even Merge Sort (OEMS)**

Welcome to the 10th lecture of the NPTEL MOOC on Parallel Algorithms. In the previous few lectures we have seen several algorithm design techniques. Today we shall start studying some algorithms designs for the comparator networks. A comparator network is made up of several comparator units. A comparator unit is a hardware that has 2 inputs and 2 outputs. On the first output the comparator produces the smaller of the 2 inputs and on the other output it will produce the other one which is the larger of the 2 outputs.

(Refer Slide Time: 01:08)



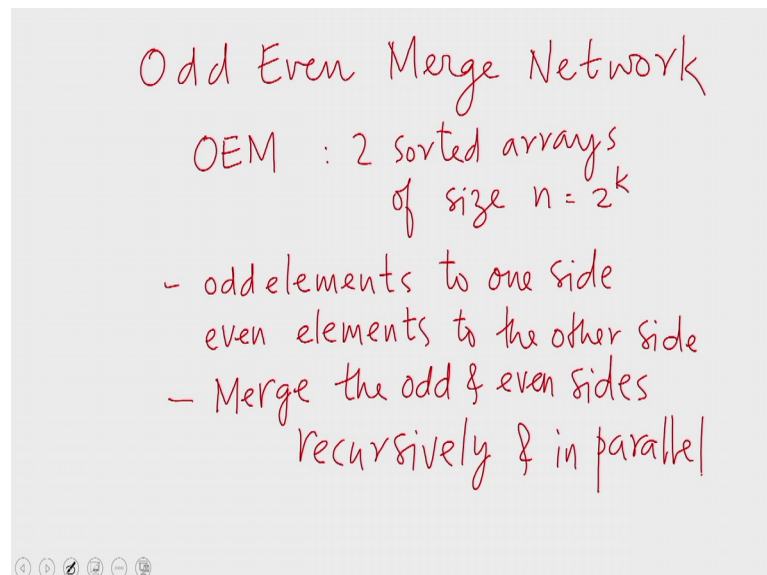
But comparator unit therefore, consists of 2 inputs a and b and it has 2 outputs. On the first outputs it produces the smaller of a and b, on the other output it produces the larger of a and b. For easiness of representation, we would represent a comparator using a line in the sense. A line connecting a vertical line connecting 2 horizontal lines, we will represent a comparator that is inserted between the 2 lines.

So, if these 2 lines are labeled a and b; the bottom figure and the top figure will mean exactly the same thing. So, the upper input will be the smaller and the lower input output

will be the larger. The upper output will be the smaller and the lower output will be the larger. So, this is how we will denote a comparator. A network made up of comparators is called a comparator network.

So, we shall assume that an output of a comparator will drive only one input of another comparator. So, that is the fan out constraint that we have. So, with this constraint we want to develop comparator networks for solving certain problems. In particular we are going to consider today a merging network and a sorting network. First let us see how we can perform merging using comparators.

(Refer Slide Time: 02:58)

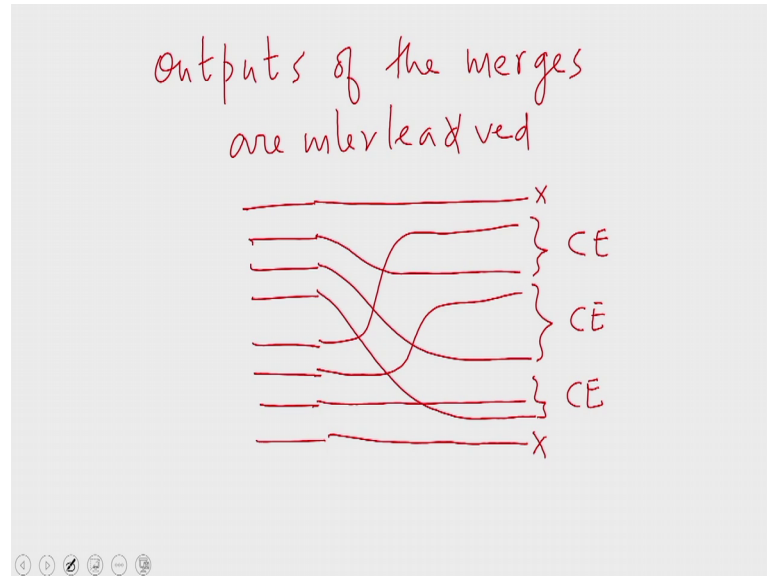


So, the algorithm that we are going to design is called the odd even merge network. The odd even merge network or OEM network for short let us say applies on 2 sorted arrays of size  $n$  each where  $n$  is an exact power of 2. So, the odd even merge algorithm works in this fashion. We extract the odd elements from both the arrays to one side. We are given 2 sorted arrays. From these 2 sorted arrays we would extract the odd elements to one side and the even in even elements to the other side and then we would merge the odd side and the even side recursively and in parallel these 2 solutions will happen simultaneously.

So, we have taken the odd elements from the first sorted array and the odd elements from the second sorted array to one side. When we pick the odd elements in sequence, the subsequence that we get is also a sorted one. So, we have now 2 instances of merge; the

odd side merge instance and the even side merge instance. Both emergence instances will be solved in parallel using recursion.

(Refer Slide Time: 05:23)



Once we have the outputs of the merged algorithms, the outputs of the merges are interleaved that is the outputs from the odd side and the outputs from the even side would be interleaved in this fashion. The first output of the odd side and the first output of the even side are taken first out then the second output of the odd side followed by the second output of the event side followed by the third output of the odd side followed by the third output of the event side and then the fourth output of the odd side followed by the fourth output of the even side; this is how we are going to interleave the outputs.

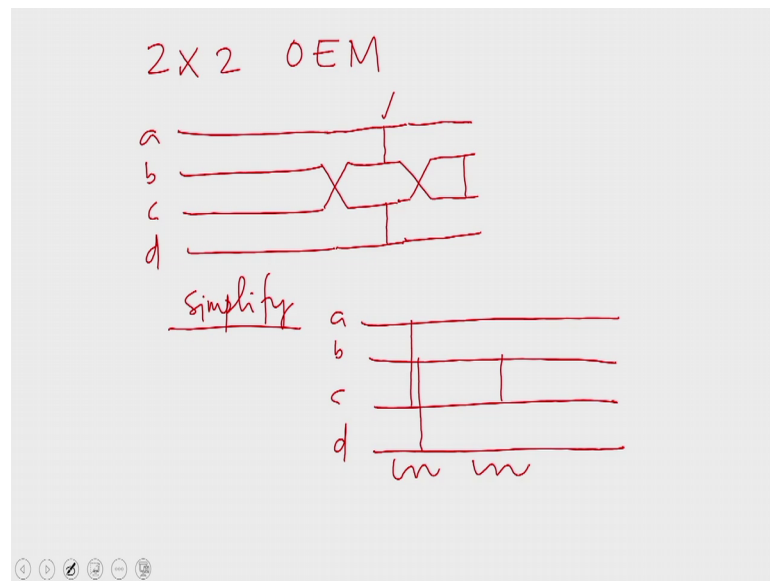
After interleaving the outputs we leave out the first one and the last one and then we pair off the remaining outputs. For each of these pairs we will perform a compare and exchange. Let us we would compare and exchange these elements if necessary. So, that is what the algorithm involves. Given 2 sorted arrays to merge both of the same size  $n$ , where  $n$  is an exact power of 2.

What we do is to take the odd elements to one side from both the arrays and even elements to one side from both the arrays then we have 2 merge instances the odd side and the even side merge instances we solve each instance recursively after that, we interleave the outputs after interleaving the outputs in this fashion with the first output of the odd side coming first and then the first output of the even side coming next an so, on.

After interleaving the outputs in this fashion, we leave out the first output and the last output the first output will be the first output from the odd side and the last output will be the last output from the even side. These 2 are left out and the remaining are paired off in sequence and then they are compared and exchanged then the output would be sorted this is what the algorithm is.

So, let us try to represent odd even merge circuit for various sizes.

(Refer Slide Time: 07:54)



So, this is a recursively defined circuit. So, the basis of the definition is the 1 by 1 odd even merge network which means we have 2 sorted arrays of size 1 each that have to be merged which means we have 2 elements that must be merged. When there are only 2 elements to be merged we can use a single comparator to merge them. So, this is what the 1 by 1 odd even merge circuit. This will form the basis of our networks.

Now, coming to a 2 by 2 odd even merge network, we proceed in this fashion. Let us say we have one 4 inputs that we called a b c d is respectively. So, these represent 2 sorted arrays of size 2 each which means a b is a sorted array and c d is a sorted array. Then what we do is to take the odd elements to one side. So, a and c are taken to one side, b and d are taken to the other side.

So, now we have 2 instances a c and b d. We solve these 2 instances these are instances of size 1 by 1 each. So, we solve these 1 by 1 merge instances recursively, but as we have

just seen 1 by 1 instances can be solved in this fashion using one single comparator. So, one comparison between a and c and one comparison between b and d taking place simultaneously can solve these 2 instances, the instances a c and b d.

After that the outputs are interleaved which means a is taken as it is the first line of the second output is the even side will have to go next which happens to be b and then the second output line of the odd instances odd instance should come which happens to be c and then we have the last instance from the even side now what we do is to leave out the first one and the last one and compare and exchange the remaining elements. So, this is what a 2 by 2 odd even merge networkers.

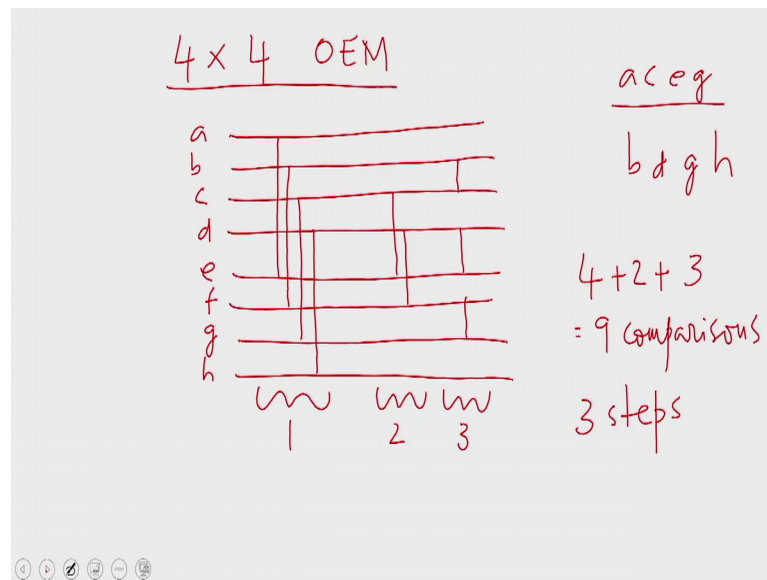
Now, we can simplify this network somewhat. Let us say we do not want the horizontal lines to be twisted. So, here b and c are twisted twice. Instead let us say we keep the input lines and twisted throughout and instead we can extend the comparators. So, at this time instance we have comparisons between a and c and b and d. So, there are simultaneous comparisons between a and c and b and d.

So, let us denote those comparisons in this fashion. This is the comparison between a and c. The start of the line and the end of the line signify between which 2 lines we perform the comparison. So, this is a comparison between lines a and c and we have another comparison between lines b and d. Even though we have drawn them side by side the understanding is that these 2 will take place simultaneously.

A comparison between a and c and the comparison between b and d these are independent because the elements involved are exclusive therefore, these 2 comparisons will be taking place simultaneously. To signify that they are taking place simultaneously, we draw them close together and then the final comparison is between the lines b and c. So, this is the final comparison that will ensure that the output is in sorted order.

So, the 2 by 2 add even merge network runs in 2 steps and involves 3 comparisons. In the first step we have comparisons between a and c and b and d respectively and in the second step we have a single comparison between b and c. So, that is what a 2 by 2 odd even merge networkers.

(Refer Slide Time: 12:37)



Now, let us come to a 4 by 4 odd even merge network. So, in a 4 by 4 network we have 8 lines we have 2 sorted arrays, let me name the lines a b c d. So, a b c d will together form one sorted array e f g h will form, the other sorted array these 2 sorted arrays have to be merged let us say. So, as the algorithm prescribes we have to take the odd elements to one side. So, the odd elements are a c e and g this must be taken to one side and b d g h must be taken to the other side. But then as in the previous example we try to avoid twisting of the lines twisting of the horizontal lines instead we would stretch the comparators.

So, let us keep the horizontal lines as they are and stretch the comparators. So, we have now 2 instances to solve the odd instances a c e g which is a 2 by 2 merge instance. So, we have to first construct a 2 by 2 merge instance between the lines a c e and g. Similarly we have to construct a 2 by 2 odd even merge network amongst the lines b d g h. So, we find that for the instance a c e g we have comparisons between a and e and comparisons between c and g; similarly for the instance b d g h, we have comparisons between b and f and d and h this is from c to g. So, this is how the first 4 comparisons will look like.

These 4 comparisons are taking place simultaneously. They are from a to e and c to g. These 2 comparisons are due to the first step of the recursive called on a c e g, the other 2 comparisons are between b f and d g these 2 comparisons are due to the recursive call invoked on b d g h.

So, as you can see in the previous diagram for 2 by 2 odd even merge this is how the comparisons should take place. So, these comparisons are all taking place simultaneously and then we have comparisons between c e and d g respectively. In the second step we have a comparison between c e and d g, this happens in the second step. So, this is what the recursive calls accomplish. Then the outputs of the recursive calls have to be interleaved, the first one and the last one have to be left out and then we have to compare and exchange the remaining pairs.

So, as you can see comparing and exchange them effectively causes a compare exchange between b and c, even though if you had drawn them with the horizontal lines twisted they would not be appearing in this order, but if you check you will find that the compare exchanges are happening exactly this way; the comparisons are between b and c and d and e and g and f and f and g. So, these 3 comparisons will be taking place simultaneously and will form the third step.

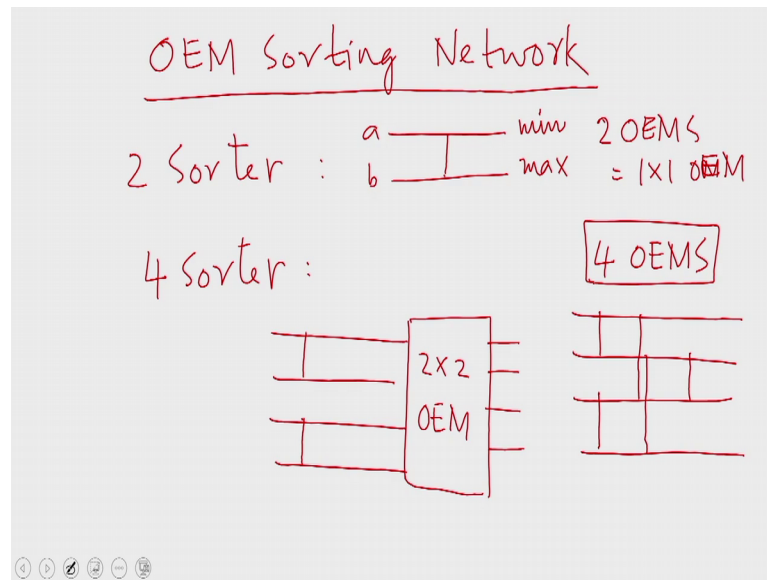
So, here we have a merged algorithm that runs in 3 steps and merges 2 arrays of size 4 each, but the total number of comparisons performed is equal to the number of vertical lines here. In the first step there are 4 comparisons, in the second step there are 2 comparisons and in the third step there are 3 comparisons; for a total of 9 comparisons, the algorithm runs in 3 steps.

So, the time complexity of the algorithm is 3 units and the cost of the algorithm is 9 comparisons. So, this is what a 4 by 4 odd even merge networkers. So, now, you can extend this to an 8 by 8 odd even merger network. In the general algorithm is the same you take the odd elements to one side the even elements to the other side since the given arrays are sorted arrays. The odd side as well as the even side will consist of 2 instances of merges and then solve these instances recursively. After that take the outputs in interleave the outputs.

The first output of the odd side followed by the first output of the even side followed by the second output of the odd side followed by the second output of the even side and so on that is what I mean by interleaving them. After interleaving the outputs we leave out the first output and the last output and then the remaining outputs are paired off and are compared and exchanged that completes the algorithm.

So, this is how you construct a  $2^k$  by  $2^k$  odd even merge network. Of course, I have yet to prove its correctness and establish its time complexity, but all in due course. But let us assume that this is an algorithm which works correctly. So, if it works correctly then I can surely use it to construct a sorting network.

(Refer Slide Time: 18:31)



We can use the odd even merge network to find an odd even merge sorting network. Of course, the sorting network begins with this was again a recursively defined network and the recursive definition requires a basis case which is the 2 sorter. The 2 sorter orders where you have to sort 2 elements given on 2 lines small a and small b these 2 have to be sorted.

To sort 2 elements all you require this one comparator. One comparator connected between these 2 lines will ensure that on the upper line we have the smaller and the lower line we have the larger of the 2 elements which means the output is in sorted order.

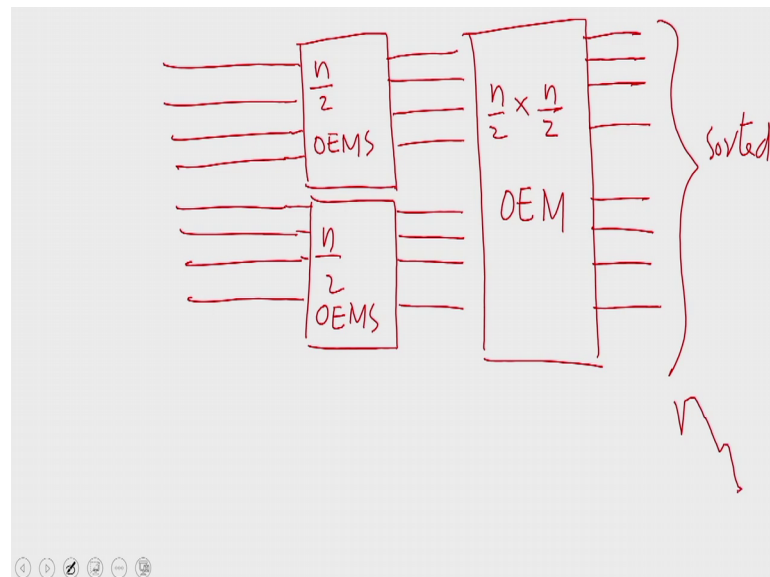
So, a 2 sorter is nothing, but a single comparator. So, as you can see it 2 sorter that is odd even merge sorter of size 2, it is the same as a 1 by 1 odd even merge network. A 1 by 1 odd even merge network is identical to 2 odd even merge sort of. In either case all we require is one single comparator to sort 4 elements when we are given 4 input elements to be sorted. In the classical merge sort way what we do is to divide this into 2 groups of equal size. So, we have 2 elements on the upper side and 2 elements on the lower side we sort each side recursively.



So, first let us sort the upper 2 elements and then sort the lower 2 elements. So, once these 2 sides are sorted we have to merge arrays 2 sorted arrays. These 2 sorted arrays of size 2 each can be merged using a 2 by 2 merger. So, we can feed this into a 2 by 2 merges, but we have just seen a 2 by 2 merges. This will produce a 4 by 4 sorter. So, since we already know what 2 by 2 odd even merge circuit is we can replace the black box with that network. So, these are the recursive sorts and then replacing the black box with the 2 by 2 merger that we have just seen we get this network.

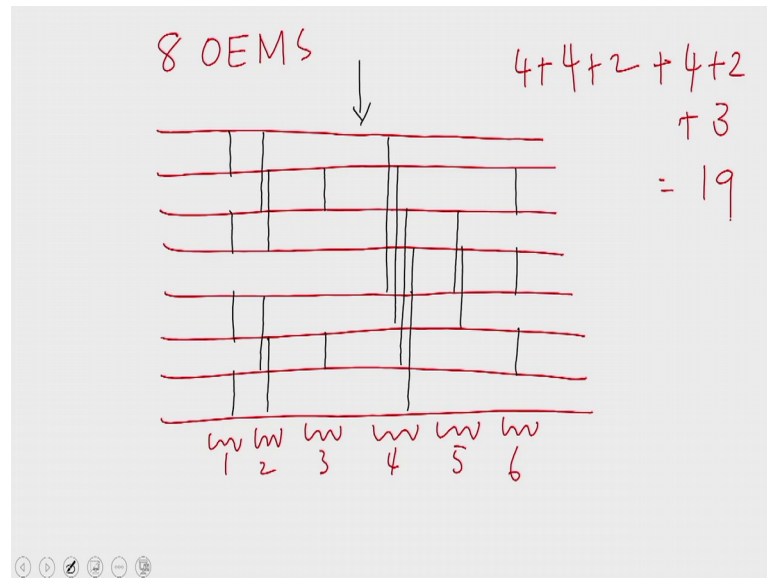
Now, this network is what I call a 4 odd even merge sorter. So, this is how we define an odd even merge sorter.

(Refer Slide Time: 21:57)



Now, we can extend those 2 higher dimensions when we are given  $n$  elements to be sorted, where  $n$  is an exact power of 2. What we do is to divide this into 2 groups. Sort each group recursively. So, I had  $n$  by 2 elements. I had a total of  $n$  elements that is divided into 2 groups. So, I have  $n$  by 2 elements on either side. So, I will invoke  $n$  by 2 odd even merge sorter. Then I get 2 sorted sequences of size  $n$  by 2 each on this I will invoke an  $n$  by 2 by  $n$  by 2 odd even merge and then the output will be sorted, provided that the odd even merge circuit works correctly which we have yet to show. So, assuming that the odd even merge sort works correctly the odd even merge sort network as well works correctly ok.

(Refer Slide Time: 23:23)



So, let us consider what a 8 sorter will look like. So, this involves sorting. The first 4 lines and the bottom 4 lines separately. So, when we sort the first 4 lines this is how we will sort the first 4 lines. On the bottom side we replicate the same pattern. So, in 3 steps we finish the recursive sorting instances.

So, now we have sorted the upper side as well as the lower side. So, now, at this point what we have are 2 sorted arrays of size 4 each then I can merge them using a 4 by 4 odd even merger. So, the 4 by 4 odd even merger as we have seen just like this. So, this is what an 8 sort of using the odd even much technique. So, as you can see here we managed to sort this in this is the first step, this is the second step, this is the third step, this is the fourth step, this is the fifth step and this is the sixth step.

So, we managed to sort 8 elements in 6 steps and the total number of comparisons used would be the total number of vertical lines which would be 4 plus 4; 8 from the first 2 steps then 2 in the third step again 4 in the fourth step 2 in the fifth step and finally, 3 in the final step which is 19.

So, we have 19 comparisons and 6 steps are what are necessary to sort 8 elements in this fashion. So, the odd even merge circuit works in this fashion. So, now, we have to prove the correctness of the algorithm as well as the time complexity of the algorithm for a general case.

(Refer Slide Time: 26:43)

A Comparator N/w  
can be simulated on an  
EREW PRAM  
for the same time &  
cost

Now, as you can readily see a comparator network can be simulated on an EREW PRAM without any increase in the running time or the cost. So, the simulation would be for the same time and the same cost. Therefore, the analysis that we are going to do for the comparator network will work exactly the same way for an EREW PRAM as well and therefore, for all the other pram models that we have seen because an EREW PRAM is the weakest of the PRAM we know.

(Refer Slide Time: 27:48)

Correctness  
OEM Network correctly merges  
two sorted binary sequences  
0000 0111  
 $O^i |^{n-i}$  ,  $O^j |^{n-j}$

So, first let us prove the correctness of the algorithm. We will show that first we will show that the odd even merge network correctly merges 2 sorted binary sequences. So, let us say we are given 2 sorted binary sequences. A binary sequence would look like this a sorted binary sequence in particular will have a number of 0's followed by a number of ones. So, we are given 2 sorted binary sequences.

So, let us say one binary sequences of this form it has  $i$  0's followed by  $n$  minus  $i$  1's. The other sequences of this form it has  $j$  0's followed by  $n$  minus  $j$  1's. So, these are the 2 binary sequences given to us. I want to claim that these 2 will be corrected correctly sorted by our odd even merge network. So, let us see what the odd even merge network will do to these sequences.

So, when these binary sequences are fed to the odd even merge network what it first does is to take the odd elements to one side and the even elements to the other side.

(Refer Slide Time: 29:32)

| <u>odd side</u>                                       | <u>even side</u>                                      |
|---|---|
| $\lceil i/2 \rceil$ 0's                               | $\lfloor i/2 \rfloor$ 0's                             |
| $\lceil j/2 \rceil$ 0's                               | $\lfloor j/2 \rfloor$ 0's                             |
| <hr style="width: 100%; border: 0.5px solid black;"/> | <hr style="width: 100%; border: 0.5px solid black;"/> |
| $\lceil i/2 \rceil + \lceil j/2 \rceil$               | $\lfloor i/2 \rfloor + \lfloor j/2 \rfloor$ 0's       |
| The diff. in no. of 0's : 2, 1, 0                     |   |
| After leaving out — " — : 1, 0, -1                    |   |

So, let us look at the odd side and the even side let us in particular look only at the 0's. We are taking the odd elements to the side and the even elements to this the other side. The 2 sorted sequences have  $i$  0's and  $j$  0's respectively. So, out of the  $i$  0's from the first array the odd side will end up getting ceiling of  $i$  by 2 0's whereas, the even side will end up getting floor of  $i$  by 2 0's. In particular if there are 17 0's in the first array and 18 0's in the second array, we would be getting 9 here and 9 here. So, there would be an advantage to the odd side sorry; on the even side there are 18 0's which means from the

first array we are getting 9 and 8 respectively to the odd side and the even side and from the other array from the 18 elements we are going to get 9 and 9 respectively. So, this is how the 0's will be split.

So, from the first array ceiling of  $i$  by 2 0's come to the outside and floor of  $i$  by 2 0's go to the even side. From the second array similarly ceiling of  $j$  by 2 0's come to the odd side and floor of  $j$  by 2 0's come to the even side. Therefore, if you compare the number of 0's on the odd side and the even side respectively, we find that we have ceiling of  $i$  by 2 plus ceiling of  $j$  by 2 on the odd side and floor of  $i$  by 2 plus floor of  $j$  by 2 0's on the odd even side.

So, when you compare these 2 numbers we find that this could be larger, but if it is larger it could be larger by at most 2 So, if you compare the odd side and the even side, the difference in the number of 0's its either 2 or 1 or 0 that is the number of 0's on the odd side minus the number of 0's on the even side could be 2, 1 or 0's.

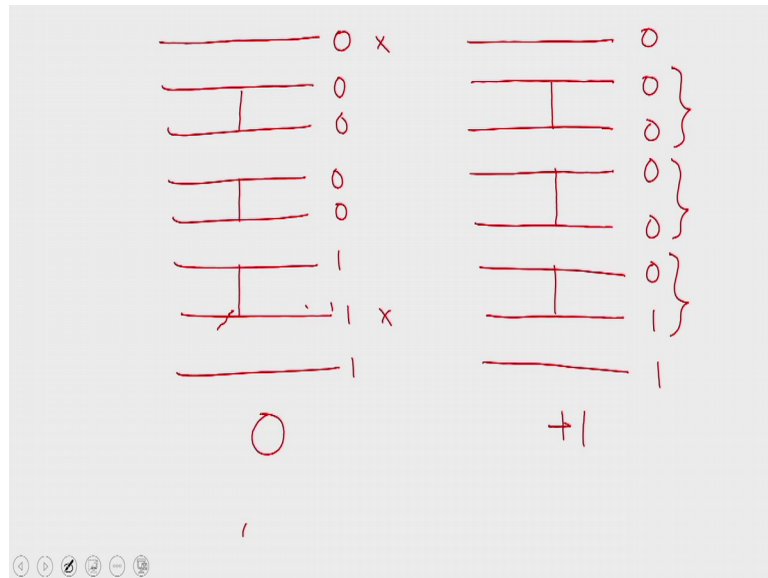
For example, if both  $i$  and  $j$  are odd there would be 2 extra 0's on the odd side. If one is odd and the other is even that is if  $i$  is odd  $j$  is even are vice versa there would be exactly one extra 0 on the outside. If both  $i$  and  $j$  are even then the 0's would be perfectly balanced that is the difference in the number of 0's would be 0.

Now, the algorithm proceeds with the 2 sides and sorts the 2 sides recursively. So, now, let us assume that the odd side and the even side are recursively sorted correctly. After that what we do is to interleave the outputs. So, the 0's and 1's coming in sorted order from the odd side and even side are interleaved. After interleaving them we leave out the first element and the last element.

So, let us separate out the case where every element is of 1. If every element is of 1 then in any case this sequence output sequence will be correctly sorted because no bit is ever changed from 0 to 1 or from 1 to 0. So, if every input is a 1 every output is also a 1, so, the output is trivially sorted. So, let us assume that there is at least one 0. If that is the case after interleaving the outputs the first output that we leave out is going to be a 0. Therefore, after leaving out the first output, the imbalance in the number of 0's between the 2 sides changes.

So, you can say that after leaving out the first element the difference in the number of 0's between the 2 sides would be 1 0 or minus 1. So, that is the scenario that we have after leaving out the first output, we have an imbalance of 1 0 or minus 1 between the 2 sides. So, let us now consider the various possible cases.

(Refer Slide Time: 00:16)

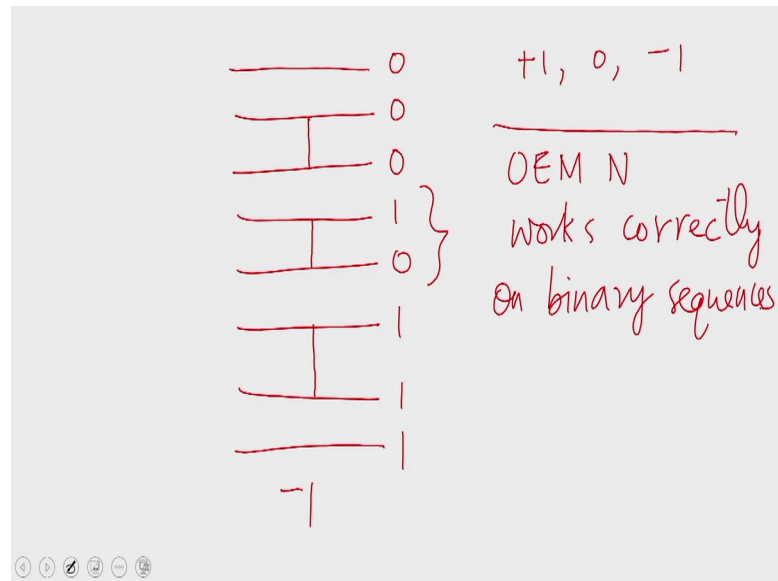


Let us say the first output is 0, since we assume that every output is not 1 the first output is anyway going to be 0. So, what we do is to this is 1 possible scenario, we leave out the first and the last 1's and pair off the remaining and compare exchange within the pairs.

So, if this is an example with 8 outputs. So, we leave out the first output and the last output and pair of the remaining and compare and exchange them. So, in this case after leaving out the first and the last we find that there is an exact balance in the number of 0's. So, this is a 0 balance case. An alternate scenario is where we have after leaving out the first and the last elements we have 1 excess 1 on the odd side.

So, when we pair off in this fashion we have an excess 0 on the odd side. In this case again when we compare and exchange the elements in this fashion the outputs will be sorted. So, this is a case where we have a balance of 0 between the 2 sides.

(Refer Slide Time: 36:09)

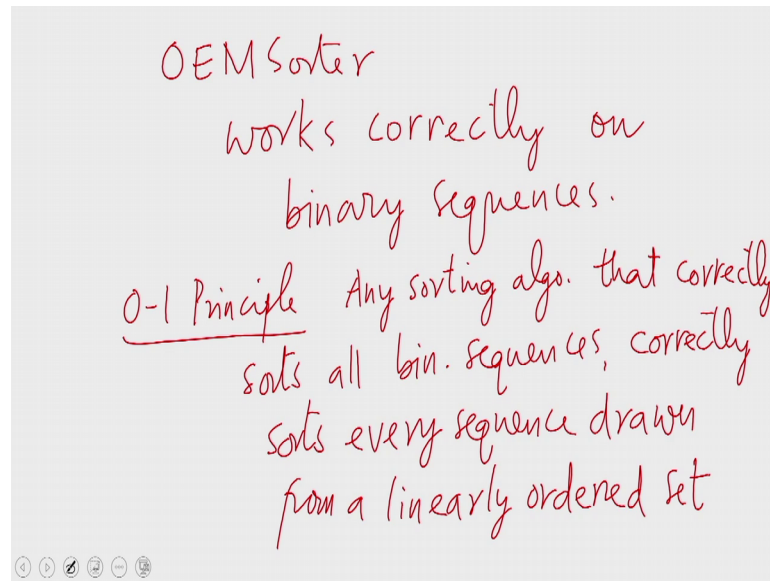


Finally, the third case is where we have a deficiency of 1 on the odd side. In this case when we compare and exchange we find that there is one anomalous pair 1 0 which is fixed by the last column of comparisons. So, this is the case where the odd side has a deficiency.

So, in all the 3 cases when we compare the odd side and the even side we find that the number of 0's between the 2 sides differ by plus 1 0 or minus 1. In all cases we find that exactly 1 comparison will fit fix the anomaly, but the only thing is that we do not know between which pair of elements the anomaly happens. Therefore, what the algorithm does is to compare and exchange every pair of elements after excluding the first element and the last element from the set of interleaved outputs.

This is guaranteed to fix the anomaly wherever it is because we know that there can be at most one anomalous pair as in this case. So, this establishes that odd even merge network works correctly on binary sequences, but then if a merge network works correctly a sorting network constructed using the merge network will work correctly as well. Therefore, we know that the odd even merge sort network or the odd even merge sorter works correctly on binary sequences.

(Refer Slide Time: 38:12)

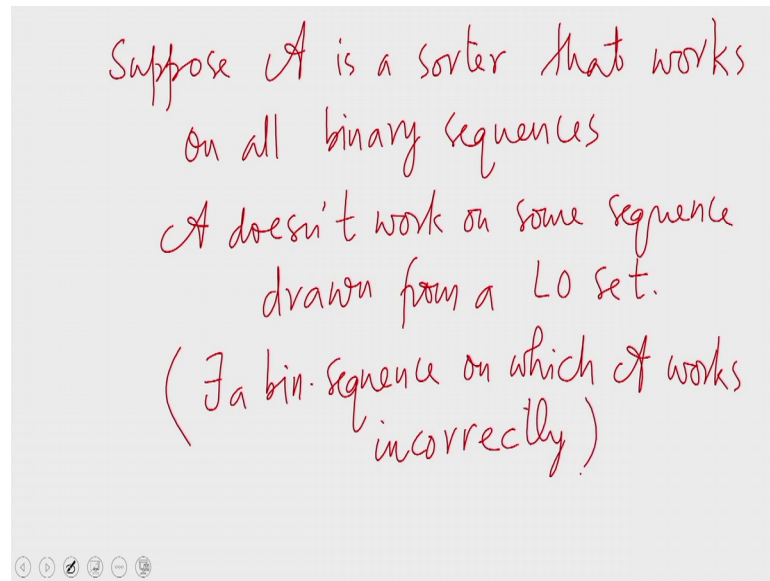


So, indeed our sorting algorithm works correctly on all binary sequences, but then not every sequence is the binary sequence. You would want to sort a sequence drawn from a linearly ordered set. What about such sequences? So, now, we want to say that odd even merge sorter works correctly on all sequences drawn from a linearly ordered set as well. But then we can make that transition by pulling what is called the 0 1 principle. The 0 1 principle asserts that any sorting algorithm that correctly sorts all binary sequences, correctly sorts also correctly sorts every sequence drawn from a linearly ordered set.

So, if we prove the 0 1 principle then we would be done because we can invoke the 0 1 principle and say that OEM sort of therefore, correctly works correctly on all sequences run from a linearly ordered set, since it works correctly on all binary sequences. So, the proof is a general one, it will apply to every correct sorting algorithm for binary sequences.



(Refer Slide Time: 40:32)

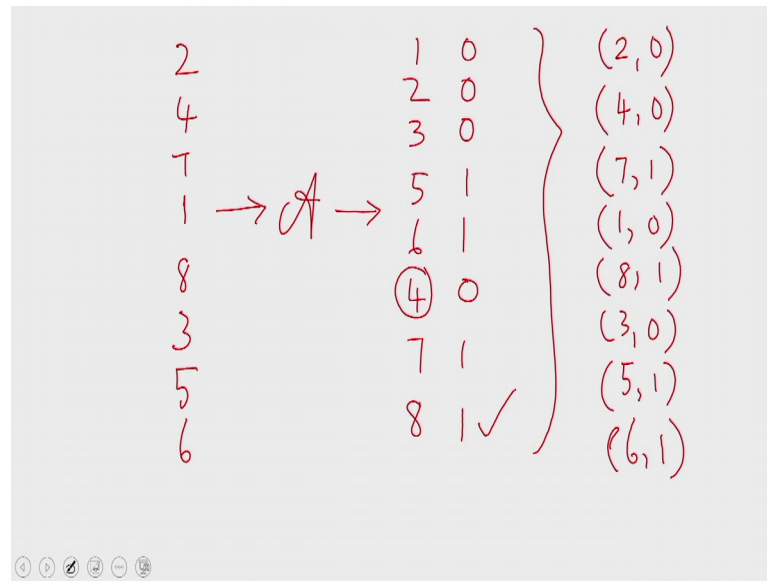


Suppose  $A$  is a sorter that works  
on all binary sequences  
 $A$  doesn't work on some sequence  
drawn from a LO set.  
( $\exists$  a bin. sequence on which  $A$  works  
incorrectly)

So, suppose  $A$  is some sorting algorithm that works correctly on all binary sequences, we are trying to prove the 0 to 1 principle here. So, we have assumed that we have some algorithm  $A$  which works correctly on all binary sequences. Now to contradict the 0 1 principle let us assume that it does not work correctly on some sequence drawn from a linearly ordered set.

So, let us say  $A$  does not work on some sequence drawn from a linearly ordered set. Now we will derive a contradiction by showing that in this case there must exist a binary sequence on which  $A$  must work incorrectly. So, this is what we are going to show there exists a binary sequence on which  $A$  works incorrectly. So, how do we prove this? So, let us take the linearly ordered sequence on which the algorithm does not work correctly and we are going to fabricate a binary sequence on which the algorithm works incorrectly as well.

(Refer Slide Time: 42:29)



So, let us assume that we have some sequence of this form on which the algorithm works incorrectly. So, when we pass it through A, the output is let us say out of order. Of course, I am taking some concrete example it does not have to be this particular sequence. The argument is that for A there exists some input sequence on which A works incorrectly, take that input sequence and feed it through A. A produces the output; the output will not be sorted.

Then from this unsorted output we look for the smallest element that is appearing out of order. Here we find that 1 is in place 2 is in place 3 is in place, but 4 is not in place. We find that 4 is not in place, 4 is the smallest element that is not in place.

And then this element and all the smaller elements we are going to tag 0. So, 4 gets a tag of 0 1 gets a tag of 0 2 gets a tag of 0 and 3 also gets a tag of 0; all the remaining elements will get tags of 1. So, we form a binary sequence in this fashion by tagging the elements in this way. The smallest element that is appearing out of order is given a tag of 0 and all elements smaller than that are also given tags of 0 and then the remaining elements are all given tags of 1.

Now, looking at the tags in this fashion I will form the input sequence, I will reframe the input sequence in this fashion, I will modify every input element by adding a tag. 2 gets a tag of 0 4 also gets a tag of 0 7 gets a tag of 1 1 gets a tag of 0 8 gets a tag of 1 3 gets a tag of 0 5 gets a tag of 1 and 6 gets the tag of 1. So, the input elements are now modified

into ordered pairs in this fashion in particular the input element 8 becomes the ordered pair 8 1 that is because the element 8 has been assigned a tag of 1 here every element greater than 4 is given a tag of 1 here in this example.

So, in particular what we have done is this. Since algorithm A is postulated to work incorrectly on some sequence drawn from a linearly ordered set. We take that input and pass it through the algorithm. The output sequence that is produced by the algorithm is not in sorted order because this is the input on which the algorithm works incorrectly. Then examining the output sequence, I find the smallest element that is appearing out of order.

This element as well as all the smaller elements are given tags of 0 and every larger element is given a tag of 1. Then I modify the original input by adding the tags as a second component. So, every element in the original sequence is converted into an ordered pair where the second component is the tag. After this set of ordered pairs will be once again passed through the algorithm.

So, now let us say the algorithm is performing comparisons on the first components as before. So, every comparison is exactly as before. Every single comparison in the algorithm is exactly as before. For example, if the algorithm had compared 2 and 8 previously, now it would be comparing 2 0 with 8 1 with exactly the same result since 2 is less than 8 it would take exactly the same decision as before and therefore, the output produced would be exactly the same that is the output produced would be 1 0 2 0 3 0 5 1 6 1 4 0 7 1 and 8 1 as ordered pairs, so, nothing has changed.

But now, let us look at the execution of the algorithm from the perspective of the tags. The tags find that suppose the tags are unaware that the algorithm is. In fact, comparing the original first components. The tags are under the impression that the algorithm works by comparing the tags.

In that case the tags find that everywhere the result of every single comparison is consistent. For example, if the algorithm compared 2 and 4, it is in fact, comparing ordered pairs to 0 and 4 0. In particular it is comparing the first components 2 and 4 and it decides that 2 0 is smaller than 4 0, but then the tags are both 0's. Therefore, what they the tags thing does that?

The tags are compared and since they are identical one of them gets placed on the minimum and the other gets placed on the maximum. Therefore, the tags do not have anything to complain. The tags involved in this particular comparison do not have anything to complain. When 8 and 7 are compared for example, the original algorithm compared them correctly and placed 7 on the smaller output.

In this case the tags are 1 and 1 each and the tags again think that the outputs are consistent with the tag values. 1 and 1 are compared and 1 of the 1's is placed on the smaller output. And if an ordered pair with 0 in the second component is compared with an ordered pair and with first 1 in the second component again the results are consistent with the tags. For example, if we compare 3 0 with 7 1, the actual comparison happens between 3 and 7, 3 is smaller and 3 gets placed on the smaller output, but then the as the tags see 0 and 1 and are being compared and 0 is being placed on the smaller output. Therefore, the tags still think that the comparisons the individual comparisons work correctly.

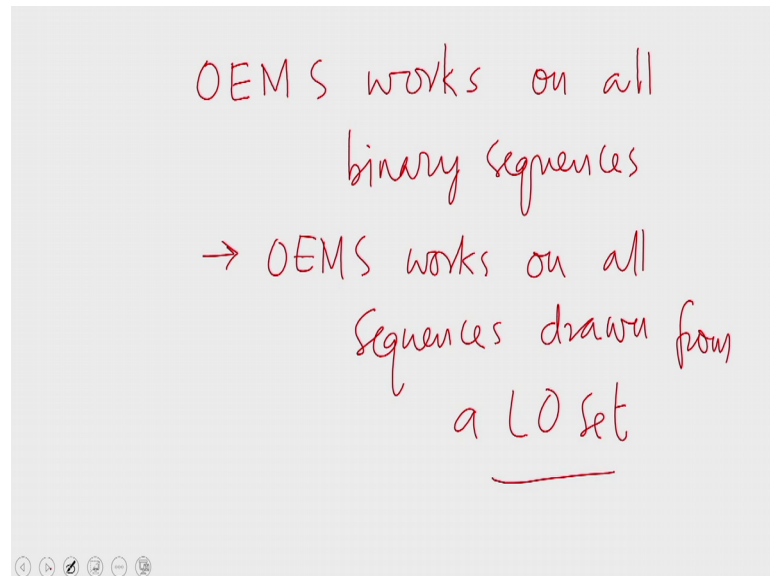
Therefore if you look at the execution of the algorithm A and focus only on the tags, we find that every single comparison within the execution of A works correctly from the perspective of the tags. Therefore, the output ought to be in sorted order of the tags. So, if the tags alone were fed into the algorithm, since the algorithm works correctly on all binary sequences they would perform exactly the same comparisons and would produce the same results, but then what we find is that the output produced is not sorted on the binary sequences that is the tags are not appearing in sorted order still we have the same anomaly that we saw earlier. There is a 0 coming after a 1. Therefore, this is a binary sequence on which the algorithm A does not work correctly.

So, the just of the argument is that if there is at least one sequence drawn from a linearly ordered set on which the algorithm does not work correctly. Then from this sequence I can construct a binary sequence on which the algorithm will work incorrectly. The binary sequence is obtained in this fashion. Past the linearly ordered sequence through the array the sequence on which the algorithm works incorrectly. Look at the output. The output is not in sorted order pick out the smallest element that appears out of order. Tag this element and every smaller element as a 0 and every other element is tagged as 1 then take the tags alone and feed them through the network, the tags will appear in exactly the

same order that we got which is an incorrect order which means this binary sequence is a binary sequence on which the algorithm works incorrectly.

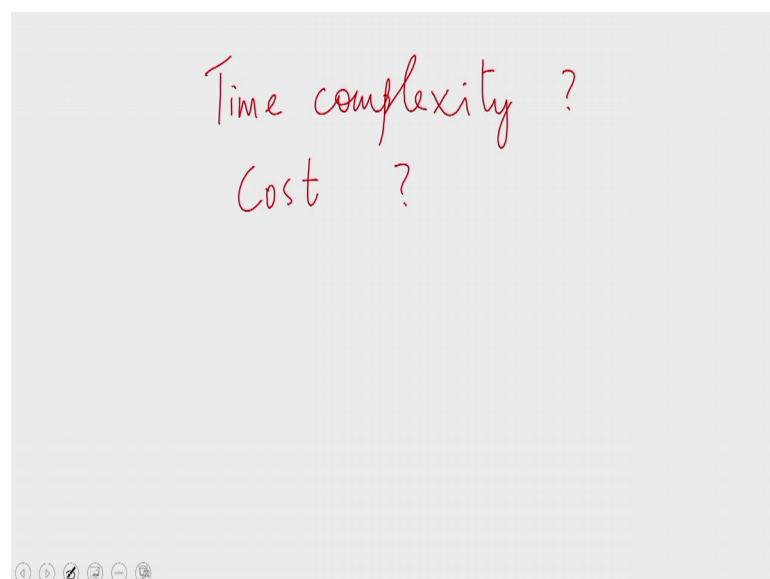
So, that establishes the 0 1 principle.

(Refer Slide Time: 50:47)



Therefore just opposing the 0 1 principle with what we had seen earlier namely that odd even merge sorter works correctly on all binary sequences we get that odd even merge sort merge sorter works on all sequences drawn from a linearly ordered set which means the odd even merge sorter works correctly indeed.

(Refer Slide Time: 51:37)



Now, it remains to show the time complexity and the instruction complexity; namely the cost of this algorithm. The analysis for the time complexity and the cost of the algorithm we shall see in the next lecture. So, that is it from this lecture hope to see you in the next lecture.

Thank you.