Parallel Algorithms

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

Lecture – 12
BSMS, Optimal List Colouring

Welcome to the 12th lecture of the MOOC on Parallel Algorithms. In the previous lecture we studied about the bitonic sort based merger and the sorter obtained from that this merger dependent on bitonic sorter network. A bitonic sorter is a network which takes a bitonic sequence as an input and produces a sorted permutation of that. So, a bitonic sorter works in this fashion. When a bitonic sequence is given to it is as an input it compares every element of the sequence with its diametrically opposite element simultaneously.

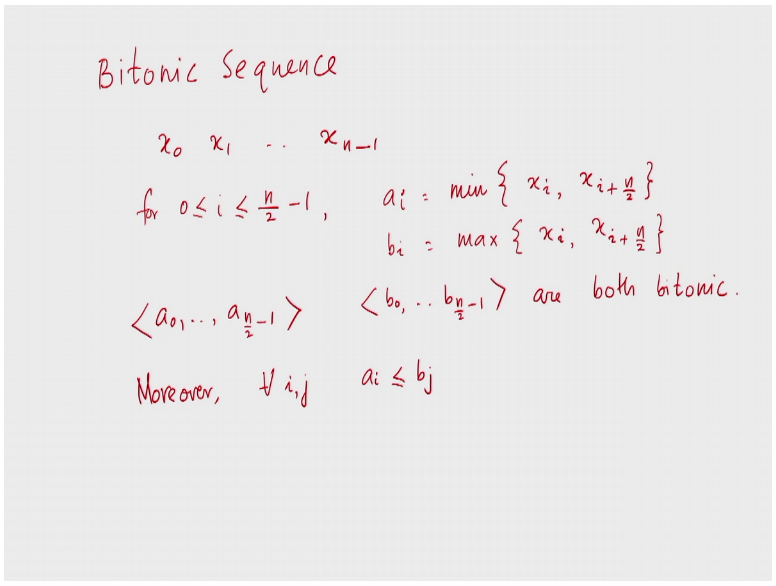
And as a result of this comparison the there is an exchange performed if necessary. So, that the smaller of a pair of elements will always be in the first half and the larger will be in the second half at the diametrically opposite position. Once this is done we claimed the input will divide into 2 bitonic sequences.

So, that the top half every element in the top half is smaller than every element in the bottom half. If this is the case then the algorithm can proceed with the top half and the bottom half independently sorting them in place and the result will be a sorted output. So, on the basis of this assumption we designed a network for sorting bitonic sequences then we went on to design a merger. The merger dependent on the fact that given 2 sorted arrays when the second array is inverted and pasted onto the first array what we get is a bitonic sequence.

And then merging the 2 arrays reduces to the problem of sorting the bitonic sequences and then based on the merger we designed merge sort algorithm which works in the conventional way given a set of elements to sort, we divide it into 2 equal halves we sort each half recursively and then we merge the 2 halves using a bitonic sort merger. So, this was our algorithm, but the correctness of the algorithm crucially dependent on the bitonic sort of working correctly.

So, let us now prove that the bitonic sort works correctly. So, let us consider a bitonic sequence.

(Refer Slide Time: 02:47)



Bitonic Sequence

$x_0 \quad x_1 \quad \ldots \quad x_{n-1}$

for $0 \leq i \leq \frac{n}{2}-1$, $\quad a_i = \min\{x_i, x_{i+\frac{n}{2}}\}$

$\quad b_i = \max\{x_i, x_{i+\frac{n}{2}}\}$

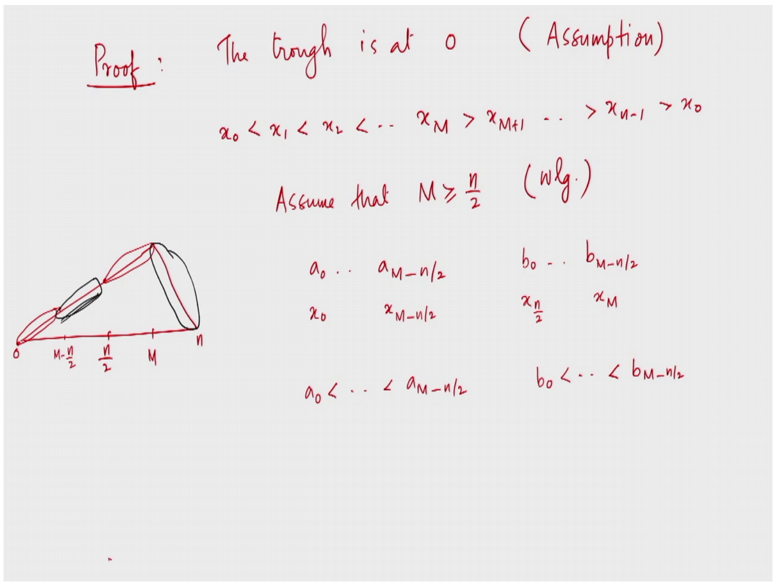
$\langle a_0, \ldots, a_{\frac{n}{2}-1} \rangle \quad \langle b_0, \ldots b_{\frac{n}{2}-1} \rangle$ are both bitonic.

Moreover, $\forall i, j \quad a_i \leq b_j$

This is a bitonic sequence of length n. So, this sequence has a trough and a peak. Let us say for i varying from 0 to n by 2 minus 1 we define a i as the smaller of the 2 elements x i and x i plus n by 2. If we keep the elements circularly then the element which is diametrically opposite to x i is x i plus n by 2. So, the smaller of x i and its diametrically opposite element will form a i, the larger of them will form b i. So, once a is and b i's are defined in this fashion we want to claim that the sequences 0 through a n by 2 minus 1 and b 0 through b n by 2 minus 1 are both bitonic. Moreover, for all i j a i is less than or equal to b j which means every element of the a sequence is smaller than every element of the b sequence is what we want to clime if this claim is true then the circuit that we designed in the last class turns out to be a correct one.

So, let us now attempt to prove this.

(Refer Slide Time: 05:03)


0
$M-\frac{n}{2}$
$\frac{n}{2}$
M
n

Proof: The trough is at 0 (Assumption)

$x_0 < x_1 < x_2 < \cdots x_M > x_{M+1} \cdots > x_{n-1} > x_0$

Assume that $M \geq \frac{n}{2}$ (wlg.)

$a_0 \cdots a_{M-n/2}$ $b_0 \cdots b_{M-n/2}$

x_0 $x_{M-n/2}$ $x_{\frac{n}{2}}$ x_M

$a_0 < \cdots < a_{M-n/2}$ $b_0 < \cdots < b_{M-n/2}$

So, to prove this we assume that the trough is at 0. This is only a temporary assumption later on we will generalize this. So, for now we will assume that the trough happens to be at 0, then we have inequalities of the sort the peak happens at capital M. In other words if you look at the sequence it will look like this. It will be either this or the mirror image of this

So, let us make that assumption 2. Let us assume that M is greater than or equal to n by 2 that is the peak happens to the right of the midpoint. But this assumption is without loss of generality because if this is not the case all that, we have to do is to take the mirror image of our arguments just look at the argument from right the right to let the left. So, let us consider the elements ranging from a 0 through a M minus n by 2 and b 0 through b M minus n by 2; a 0 happens to be the smaller of the 2 elements x 0 and x n by 2.

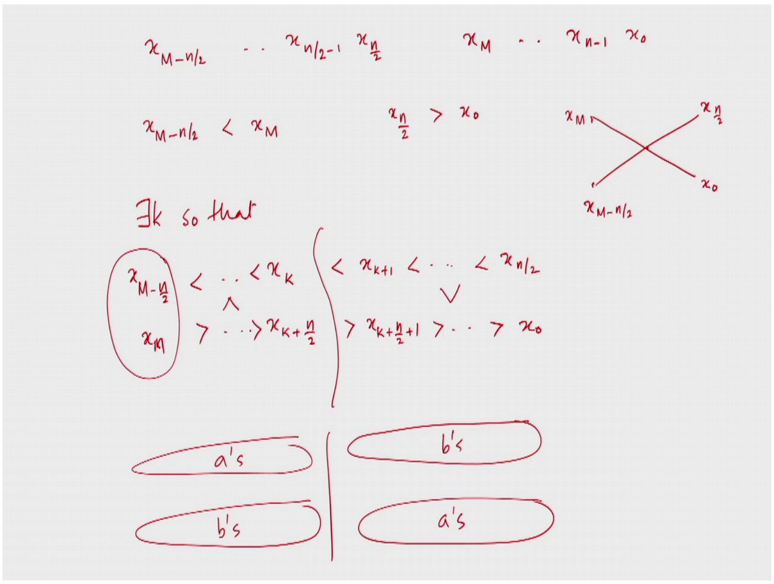
So, x 0 happens here and x n by 2 happens here. Clearly x 0 is smaller than x n by 2 therefore, a 0 is x 0 and if you look at the right element in this range, the right element is happens to be x M minus n by 2 and its diametrically opposite element is x M. Since, x M is the peak of the sequence x M is greater than x M minus n by 2.

Therefore the smaller of these 2 would be x M minus n by 2 which happens to be a M minus n by 2. Then b 0 is x n by 2 and b M minus n by 2 is x M. If you mark them on this diagram I have to find a point n by 2 positions away from M then these elements are a 0 through a M minus n by 2 whereas, these elements are b 0 through b M minus n by 2.

So, clearly we can see that the a 0 through a M minus n by 2 elements are all smaller than b 0 through b M minus n by 2 elements. In particular we also know that e 0 is less than a 1 which is less than a 2 which is less than and so on; this is how the inequalities go. From the diagram it is also clear that b 0 is the smallest of the other sequence.

So, we have established these inequalities. Mind you our goal is to show that the a's and the b's form bitonic sequences which means we have to look at the a's and b's and establish that there are exactly 2 tones in them. So, we have done part of the job that is we have shown that the sequence from 0 to M minus n by 2 in both cases have exactly 1 tone.

(Refer Slide Time: 09:23)



$x_{M-n/2} \cdots x_{n/2-1}\ x_{\frac{n}{2}}$ $x_M \cdots x_{n-1}\ x_0$
$x_{M-n/2} < x_M$
$x_{\frac{n}{2}} > x_0$
x_M
$x_{\frac{n}{2}}$
x_0
$x_{M-n/2}$
$\exists k$ so that
$x_{M-\frac{n}{2}} < \cdots < x_k < x_{k+1} < \cdots < x_{n/2}$
$x_M > \cdots > x_{k+\frac{n}{2}} > x_{k+\frac{n}{2}+1} > \cdots > x_0$
a's
b's
b's
a's

Then continuing further let us compare elements x M minus n by 2 to x n by 2 on the one hand and x M through x n minus 1 through x 0 on the other hand. So, for symmetry let me write like this. The diametrically the diametrically opposite position for x n by 2 is x naught and the diametrically opposite element for x n by 2 minus 1 is x n minus 1.

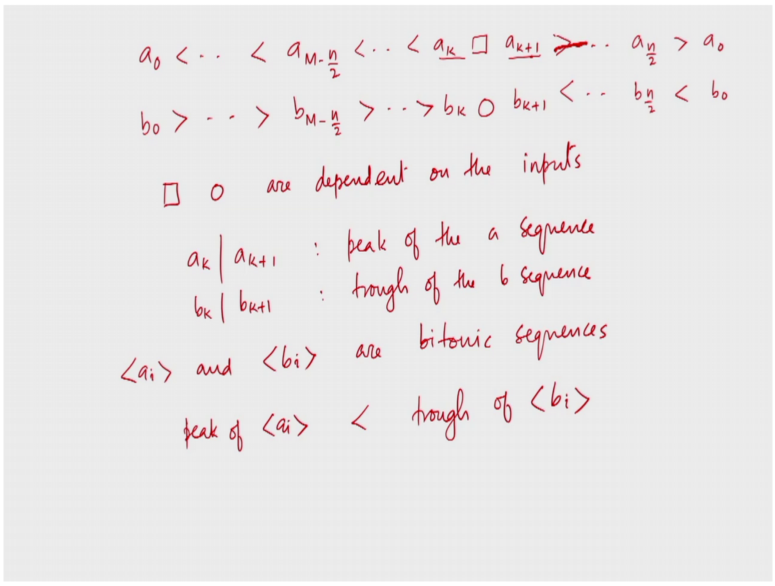
So, let us compare these 2 sub sequences. We find that x M minus n by 2 is less than x M x M is the peak. So, clearly this is the case whereas, x n by 2 is greater than x naught that is because x naught is the trough. Therefore, when you compare these 2 sequences what we find is that they cross over; x M minus n by 2 is smaller than x M whereas, x n by 2 is larger than x naught. So, at some point these 2 sequences must cross over. In this figure we are considering this part and this part; these are the 2 sequences we are comparing now.

So, we can clearly see that these are crossing over because the leftmost point of the left part which is this part, the position at M minus n by 2 is smaller than x M where is the right point which is at x n by 2 that is greater than x naught; n is the same as 0. Therefore, these 2 must be crossing over at some point. Let us say the crossover happens at k. Therefore, what we find is that there x is k, so that x M minus n by 2 increases all the way to x k and then we have x k plus 1 continuing all the way to x n by 2 whereas, on the other hand we have x n which is greater than all these elements yeah.

So, as the diagram shows x M through x 0 is a decreasing sequence and x M minus n by 2 2 x n by 2 is an increasing sequence. But then what we know is that there is a crossover point somewhere between x M minus n by 2 to x n by 2. So, for that value of k a crossover happens so that for every a for the elements to the left of the cross over the upper sequence is smaller whereas, for the elements to the right of the crossover the lower sequence is smaller. So, we have written this sequence on the lower side. So, what we know is that here x M 2 the crossover point is larger. So, I can denote it in this fashion whereas, from here onwards it is the other way round.

Since, the elements that are returned against each other are diametrically opposite elements of each other what it transpires that x M is the b of x M minus n by 2 and x M that is out of these 2 elements. The b happens to be x M and the a happens to be x M minus n by 2 or in other words the elements on the upper side are all a’s and the elements on the lower side are all b’s on the side of the divider. On the other side of the divide it is just the other way around these are all b’s and these are all a’s.

(Refer Slide Time: 14:01)



$a_0 < \cdots < a_{M-\frac{n}{2}} < \cdots < \underline{a_k} \;\square\; \underline{a_{k+1}} > \cdots a_{\frac{n}{2}} > a_0$

$b_0 > \cdots > b_{M-\frac{n}{2}} > \cdots > b_k \;\bigcirc\; b_{k+1} < \cdots b_{\frac{n}{2}} < b_0$

$\square$ $\bigcirc$ are dependent on the inputs

$a_k \mid a_{k+1}$: peak of the a sequence

$b_k \mid b_{k+1}$: trough of the b sequence

$\langle a_i \rangle$ and $\langle b_i \rangle$ are bitonic sequences

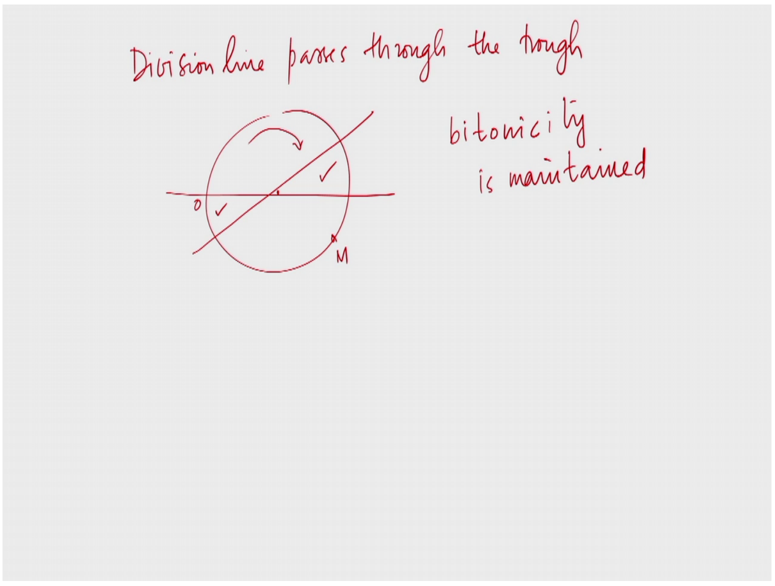
peak of $\langle a_i \rangle$ $<$ trough of $\langle b_i \rangle$

What that means, is that we have these inequalities that we established earlier. Now, further on we can go on to argue that up to a k we have an increasing sequence then we do not know the relationship between a k and a k plus 1. We have these inequalities similarly we also have these inequalities. The square and the circle represent inequalities that we do not know yet. So, as you can see these directly follow from what we concluded in the previous slide, we have as on this side and as on this side. So, the x M minus n by 2 through x case are all increasing values of a's and then x k plus n by 2 plus 1 through x naught or all decreasing values of a's; similarly, for the b sequence.

So, the square and the circle are dependent on the inputs, but then here if we see that the a sequence peaks at either a k or a k plus 1. One of this is the peak of the a sequence. Similarly b k or b k plus 1 is the trough of the b sequence. The peak of the b sequence happens to be b 0 and the trough of the a sequence happens to be a 0 which means both a and b are bitonic sequences which is partly what we wanted to show. In addition to this we also wanted to show that every element of the a sequence is smaller than every element of the b i sequence. To this all we have to show is that the peak of the a i sequence is smaller than the trough of the bi sequence.

So, the peak of this a sequence is either a k or a k plus 1, we do not know which, but then from what we saw in the previous case if it is a k then a k happens to be less than b k which happens to be either greater or less than b k plus 1 which we do not know, but either way the trough of bi is b k or b k plus 1. So, what we establishes that the peak of the a sequence is certainly less than the trough of the b sequence. So, that establishes the result except in that the division line that we have used passes through the trough.

(Refer Slide Time: 17:49)

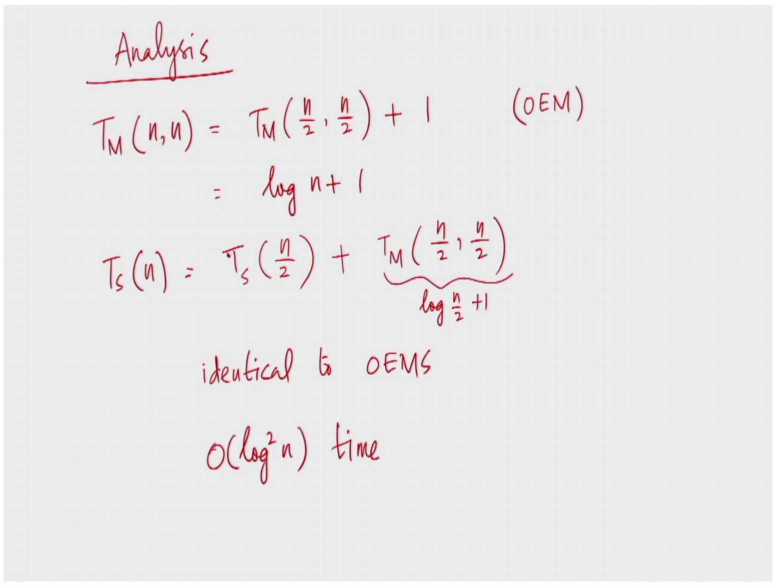


That is we prove the theorem assuming that the circulars divided through the trough that is 0 happens to be the trough is the assumption we made and if we move in this fashion we assumed the peak happens to be somewhere after the midpoint.

Now, if we do not want to make this assumption all that we have to do is to take another line. In that case all that happens is that 2 sectors will swap their positions. They are diametrically opposite to each other. So, every element on this sector will be moving to the other sector and every element on the other sector will be moving on to the sector except for that the sequence continues to be bitonic that is because a cyclic shift of a bitonic sequence is bitonic again.

So, this establishes that bitonicity is maintained even if you change the division line. So, it is not really necessary that the division line passes through the trough even if it does not pass through the trough the bitonicity will be maintained. So, that establishes the correctness of our bitonic sort algorithm, now that is the correctness.

(Refer Slide Time: 19:25)



Analysis

$$T_M(n,n) = T_M\left(\frac{n}{2},\frac{n}{2}\right) + 1 \qquad (\text{OEM})$$

$$= \log n + 1$$

$$T_S(n) = T_S\left(\frac{n}{2}\right) + \underbrace{T_M\left(\frac{n}{2},\frac{n}{2}\right)}_{\log \frac{n}{2} + 1}$$

identical to OEMS

$O(\log^2 n)$ time

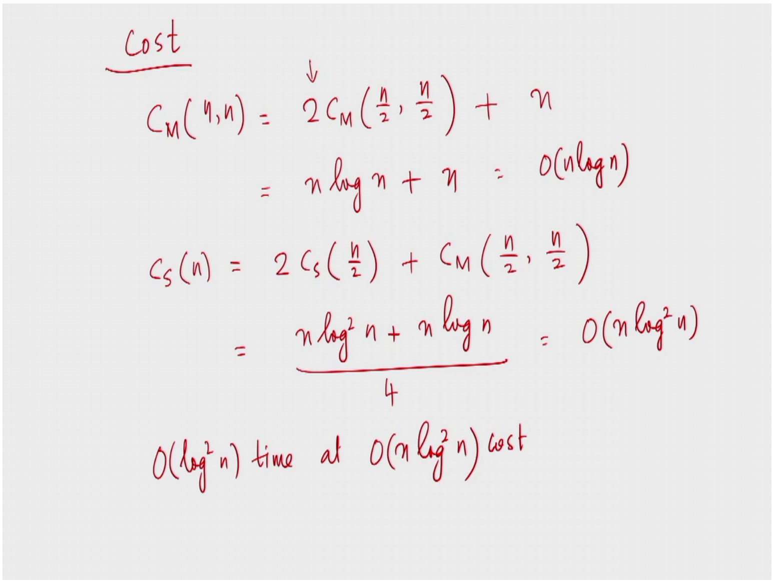
Now, let us go on to do the analysis of the bitonic sort algorithm. First let us consider the cost of merging 2 arrays of size n each. This we find that is governed by this recurrence relation because when we are given 2 arrays to be merged what we do is to twist the lower array pastes them back to back to form a bionic sequence and then we sort the bitonic sequence.

In the bionic sequence what we do is to compare every element with the diametrically opposite element. These comparisons can be performed in order one time, just one single comparison all of them executing simultaneously. So, once these comparisons are performed then what remains is to merge the 2 remaining half that is 2 bitonicity do bitonic sort on the 2 bitonicity sequences that we get which is identical to merging 2 arrays of size n by 2 each. Therefore, the recurrence relation that we get is this which is identical to the recurrence relation that we got for the odd even merge.

Therefore we will have an identical solution. The running time of the bitonic sort merger is also log n plus 1 and the running time of the sorting algorithm again will be identical because we are essentially using the same sorting algorithm; given n items to sort we divide it into 2 halves of n size n by 2 each. We sort each half recursively. Since these recursive invocations are simultaneous we have to count the time only once and then we merge the 2 sequences. This is T of M, but T of M n by 2 n by 2 is log n by 2 plus 1.

Therefore, we will get an identical expression as we got in the case of the odd even merge sort which is order of log squared n. So, bitonic sort merge sort also runs in order of log square n time, but the cost is bound to be different.

(Refer Slide Time: 22:15)



Cost

$$C_M(n,n) = \overset{\downarrow}{2}\,C_M\left(\frac{n}{2},\frac{n}{2}\right) + n$$
$$= n\log n + n = O(n\log n)$$
$$C_S(n) = 2C_S\left(\frac{n}{2}\right) + C_M\left(\frac{n}{2},\frac{n}{2}\right)$$
$$= \frac{n\log^2 n + n\log n}{4} = O(n\log^2 n)$$

$O(\log^2 n)$ time at $O(n\log^2 n)$ cost

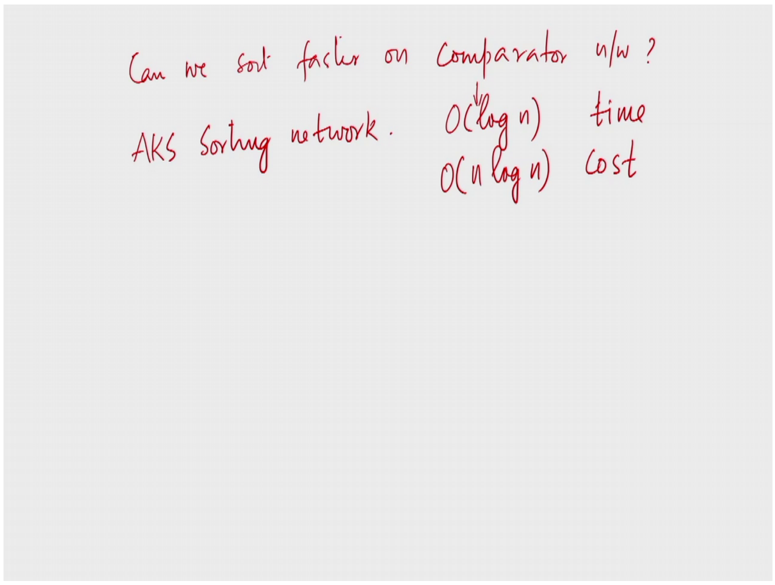
First consider the cost of the merge circuit. The cost of the merge circuit can be written like this. Every element is compared with the diametrically opposite element and exchanged and then after that we are left with 2 bitonic sequences each bitonic sequence has to be sorted. So, the cost of the bitonic sorter will come in the cost of the bitonic sorter of size n is the same as the cost of merging 2 sequences of size n by 2 n by 2 because the 2 are computationally the same.

So, the cost is twice here. We have to count the cost of both the recursive calls and then the additive term in this case happens to be n whereas, in the case of the odd even merge sort it was n minus 1. So, I will leave this recurrence relation for you to solve as an exercise, I will just give you the final solution. You can find that the final I answer happens to be n log n plus n which is more than the cost of the odd even merge sort, odd even merge network. So, this is the cost of merging using the bitonic sort merger. Then the cost of the bionic sort merge sorter can be expressed in this fashion; given an array of size n we divide it into 2 equal halves we sort each half.

So, we have to count the cost of each recursive call that plus the cost of merging the 2 arrays. So, here again I will give you the final expression. I leave the recurrence relation for you to solve. So, this is order of n log square n this is order of n log n. So, what we establishes that the bitonic sort merge sorter runs in order of log square n time at order of n log square and cost exactly as in the case of the odd even merge sorter.

So, both these algorithms have the same asymptotic complexity, both of them run in order of log squared n time at order of n log squared n cost even though both of them use drastically different algorithms.

(Refer Slide Time: 25:05)



Can we sort faster on comparator n/w ?

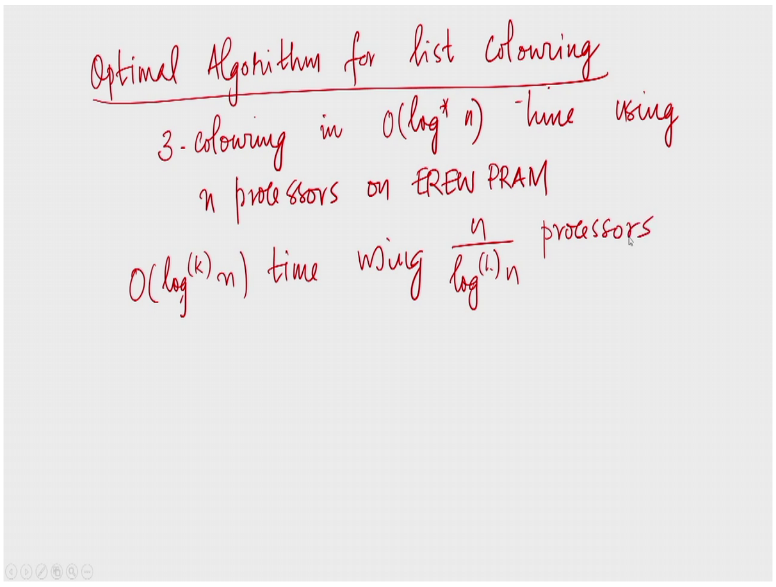
AKS Sorting network. $O(\log n)$ time

$O(n \log n)$ cost

Then of course, naturally you would ask can we achieve a faster algorithm. These 2 networks that you have just seen are very old networks. These algorithms were invented in the 1950’s. Then after nearly 30 years was invented the AKS sorting network which has a complexity of order of log n and a cost of order of n log n which happens to be an optimal algorithm even on an EREW PRAM. But the only downside is that the constant factor hidden here happens to be very large. Therefore, this is not a very practical network, but it is not within the scope of this course to discuss AKS sorting network.

So, that was about sorting on comparator networks. Now we move on to another algorithm. This algorithm we will use is a precursor to an optimally strangling algorithm which we shall study in the subsequent lectures.

(Refer Slide Time: 26:33)



Optimal Algorithm for list colouring

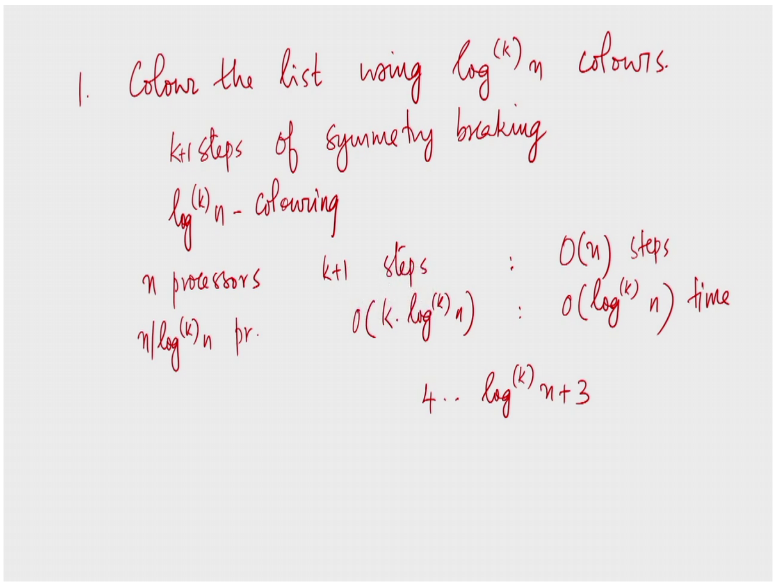
3-colouring in $O(\log^* n)$ -time using n processors on EREW PRAM

$O(\log^{(k)} n)$ time using $\frac{n}{\log^{(k)} n}$ processors

So, today we are going to discuss an optimal algorithm for list colouring. We have already studied an algorithm for list colouring an algorithm that colours a list in order of log star n time using 3 colours, but this algorithm is not optimal that is because it uses n processors, this is the algorithm we have now.

Let us see if we can get an optimal algorithm out of this; log star n is a very slow growing function. So, as we discussed before even for very large values of n log star n is practically a constant. But then still technically speaking this is not an optimal algorithm because the cost of this algorithm is n times log star n as n tends to infinity this grows super linearly with n. Therefore, we want an algorithm with a cost of order n. So, we will device an algorithm that runs in order of log k n time for a constant k using n by log k n processors.

(Refer Slide Time: 28:21)



1. Colour the list using $\log^{(k)} n$ colours.

k+1 steps of symmetry breaking

$\log^{(k)} n$ - colouring

n processors k+1 steps : $O(n)$ steps

$n/\log^{(k)} n$ pr. $O(k \cdot \log^{(k)} n)$: $O(\log^{(k)} n)$ time

$4 \ldots \log^{(k)} n + 3$

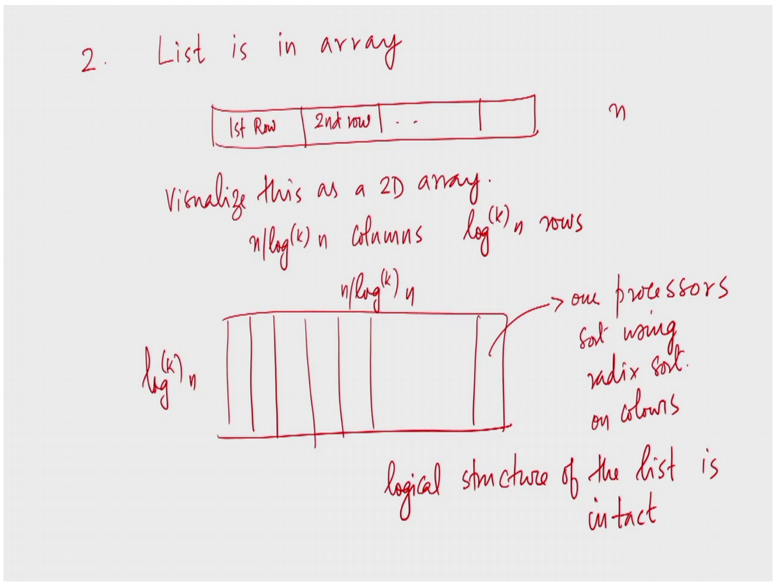
So, the first step of the algorithm is to colour the list using log k n colours. We can achieve this using our symmetry breaking algorithm. In the symmetry breaking algorithm we assume that every vertex has a processor and then the processors looked at the colour of the vertex on which it stands as well as the colour of the vertex neighboring vertex and then using these 2 colors invented new colour for their vertices and this iteration was continued for several steps. So, we established that each of these iterations executes in order of 1 time. So, we will execute some of these iterations.

So, let us say we run this algorithm for k steps, we run the symmetry breaking for k steps. So, after k steps we will be left with a log k n colouring or if you wish we can go for k plus 1 steps because since we add 1 bit at the end of every step the number of steps required to get a log k n colouring is not k, but k plus 1. So, after k plus 1 steps we will have a log k n colouring of the list.

So, we do this using n processors and we have taken k plus 1 steps where k is a constant. So, the total cost is order of n since k is a cost constant. If you have only n by log k n processors, we would take order of k times log k n steps for this. Since, k is a constant this is the same as order of log k n time.

So, in order of log k n time, we managed to find this colouring, but our goal is to finally, 3 colour the list, but right now we have a log k n colouring. So, we have to convert this log k n colouring into a 3 colouring. So, in the second step what we do is this.

(Refer Slide Time: 31:05)


2. List is in array
1st Row
2nd row
. . .
n
Visualize this as a 2D array.
$n/\log^{(k)} n$ columns $\log^{(k)} n$ rows
$n/\log^{(k)} n$
$\log^{(k)} n$
one processors sort using radix sort. on colums
logical structure of the list is intact

The list is given in an array. As we discussed earlier the list has a physical representation, the list is given in an array, but the physical representation does not need to have any relation to the logical representation. The physical representation is the order in which the vertices are given in the array.

So, we have an array of this form let us say. We can visualize this array as a 2 dimensional array. This 2 dimensional array that we are visualizing will have n by log k n columns and log k n rows. So, in this array of size n we can assume that the first n by log k n elements will form the first row then the remaining n by log k n. The next n by log k n elements will form the second row and so on we assume a row major representation.

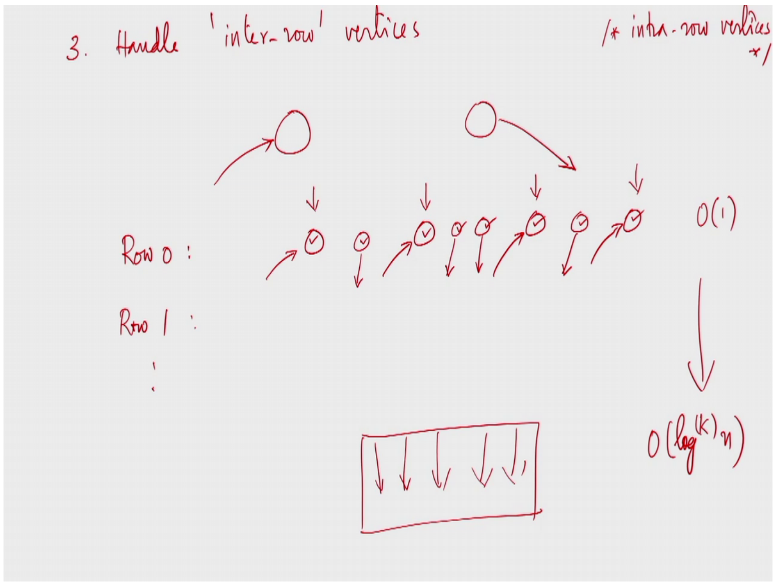
So, we visualize a 2 dimensional array in this manner. So, henceforth let us look at the array as a 2 dimensional array. The 2 dimensional array has log k n rows and n by log k n columns. So, this is a fat array with only a small number of rows. So, this is the fat short array and then what we do is this. Let us depute one processor to each column. Each column has the size of log k n we depute one processor to each column. We can do this because we have enough processors we have n by log k n processors. What these processors do is to start the vertices in the column using radix sort. We have one processor per column and the number of elements in the column is log k n. So, if we use radix sort we can use radix sort because we are sorting on the colours and the colours are integers in the range 0 to log k n minus 1.

So, we can use radix sort to sort the vertices on their colours. So, that the vertices with the smallest colour will appear at the top and the vertices with the larger colours will appear at the bottom. While sorting we make sure that the logical structure of the list is intact. The logical structure of the list is defined by the pointers. Therefore, we have to redefine the pointers appropriately. So, for i node the logical connections are the incoming pointer as well as the outgoing pointer. So, what we do is this every vertex remembers its current position and then once we have sorted the columns every vertex will also know its new position within the same column, a vertex may move up or down in the column.

So, now it knows its new position within the same column. Then it goes back to its original position and there it checks the new positions of its existing neighbors. So, the neighbors will have found their new positions within their respective columns now. So, all that the vertex has to do is to look at the neighbors and find their new positions. So, now, the vertex knows the new positions of its neighbors. So, now, the vertex will go and occupy its new position.

So, now everybody is in place in their new positions with the knowledge of the new positions of their neighbors which means, we have now sorted the vertices while keeping the logical structure of the list in duct and then in the third step we handle what we called the inter row vertices.

(Refer Slide Time: 35:25)



We say that a vertex is an inter row vertex if it either has an incoming pointer from another row or an outgoing pointer into another rho. In these 2 cases we term the vertices inter row vertices, any vertex which is not an inter row vertex is an inter row vertices, we will handle these later. So, at the moment we are going to handle the inter row vertices.

Let us assume that the colours that we used in the first step ranged from 4 to log k n plus 3. Our final goal is to colour the list with colours 1 2 3. We do not want the colour ranges to interfere with each other. Therefore, we will assume that the colours that we used in the first step are in the range 4 to log k n plus 3. So, now, we consider the inter row vertices of row 0 which is the topmost 0 topmost row. So, we consider the inter row vertices of row 0. First let me consider all the inter row vertices in row 0 that has an incoming edge from another row.

Now, no 2 of these can be adjacent to each other that is because their incoming edges are from another row. So, they will form an independent set. So, we have one processor in each column and since we are considering row 0, let us assume that we have one processor stationed on all these vertices. So, out of the row 0 vertices we are making up the vertices that has an incoming edge from another row these vertices will form an independent set.

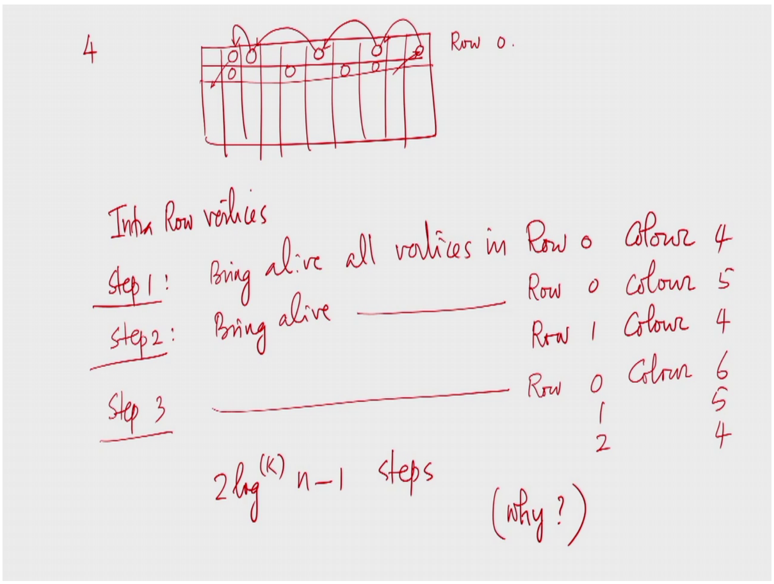
So, let us say these vertices are woken up, we have a processor sitting on them. What the processor does is this it looks at the neighborhood of these vertices and if the and will adopt for this vertex the least colour that is not in the neighborhood. At the moment these vertices are coloured in the then in the third step we handle what we called the inter row vertices using colours in the range 4 to log k n plus 3. So, none of this legally coloured because a legal colour is 1 2 or 3.

So, for all these vertices the processors will adopt the least colour not in their neighborhoods. And then we will consider all row 0 vertices with point as going to other rows they will again from an independent set no 2 of them can be adjacent because they are outgoing pointers are all 2 different rows.

Now, these nodes are brought up alive and the processes sitting on them will recolour them with valid colours which establishes that every endure row vertex of row 0 has been validly coloured. Then we move on to row 1 where we do the same thing and so on. So, at each row we will be spending order of one time. Therefore, in a total of order of log k n time, we will manage to validly colour every inter row vertex in the list when do we have a 2 dimensional array and we are going down the columns using one processor per column.

So, once the processes hit the bottom we would have properly coloured every inter row vertex. Now we bring all the processors back up and station them in the topmost row again.

(Refer Slide Time: 39:45)



Intra Row vertices

Step 1: Bring alive all vertices in Row 0 Colour 4
Row 0 colour 5
Step 2: Bring alive ________ Row 1 Colour 4
Step 3 ________ Row 0 Colour 6
1 5
2 4

$2\log^{(K)} n - 1$ steps

(why?)

Now, the processors are back here they are all in the topmost row which is row 0. Now we have to handle the intra row vertices. Several intra row vertices could be adjacent to each other which means it could be that we have connections of the sort. We could have several intra row vertices of row 0 connected up like this. What we do is this. In the first step, we bring alive all vertices in row 0 that are of colour 4; color 4 is the least invalid color. So, all vertices of colour 4 and row 0 are brought alive. They will form an independent set because no 2 vertices of colour 4 can be adjacent to each other.

Therefore, all these vertices can be given a valid colour in order one time. All that the processors sitting on them have to do is to look around in the neighbors in neighborhood. There are 2 neighbors at the most and adopt for the vertex the least valid colour not in the neighborhood there is always 1 such least colour. So, every colour 4 vertex in row 0 is now coloured, validly coloured. Then in the second step we bring alive all vertices in row 0 with colour 5 and row 1 with colour 4 which means in step 2 the sum of the row number and the colour should be 5 whereas, in step 1 the sum of the row number and the colour should be 4. So, here we have brought up all these vertices alive.

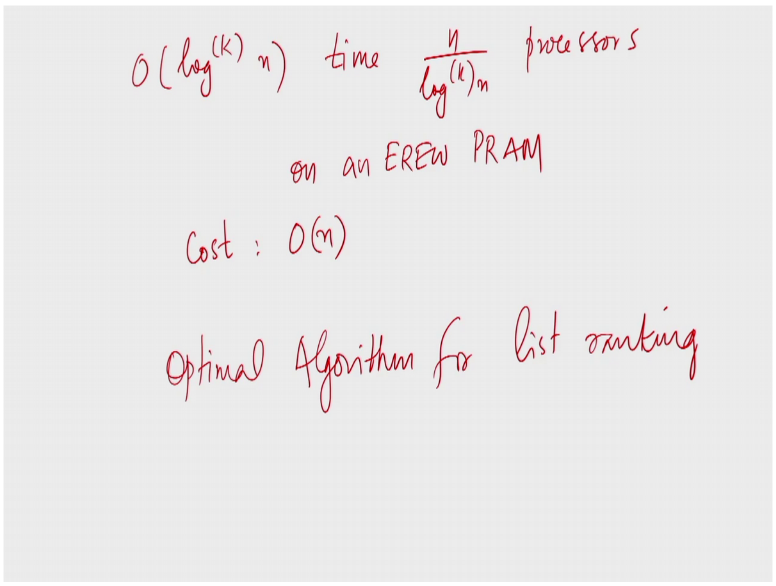
So, here we have some vertices of colour 5 in row 0 and we also consider some vertices of colour 4 in row 1, but then these are all intra row vertices which means for all these vertices the neighbors are from the same row. Therefore, there cannot be a conflict between a row 0 vertex and a row 1 vertex and the row 0 what vertices are all of colour 5. So, they will form an independent set and the row 1 vertices are all of colour 4.

And they will also form an independent set and we have a guarantee that the 2 sets will not conflict with each other. Therefore, together they will form an independent set and we can recolour all of them with valid colours. Then in step 3 we will bring alive all vertices in row 0 with colour 6, in row 1 with colour 5, in row 2 with colour 4 and so on. So, if you continue like this you will require 2 times log k n minus 1 steps before we finish all the vertices.

So, I pose a question to you, how do you show that the total number of steps required is twice log k n minus 1? So, figure it out yourself. But the logic used is this in the first step we consider all vertices where the row number plus the colour is 4. In the second step we consider all vertices where the row plus colour value is 5; in the third step we consider all vertices where the row plus colour values 6 and so on. And this can be at most how much? The maximum row number which is log k n minus 1 and the maximum colour value which is log k n plus 3. So, that is the maximum row plus color value. So, if you imagine where we started from then you can easily figure out that the total number of steps required is twice log k n minus 1 steps.

So, by the end of the, these steps the list would be properly 3 colour.

(Refer Slide Time: 44:17)



$O(\log^{(k)} n)$ time $\frac{n}{\log^{(k)} n}$ processors

on an EREW PRAM

Cost : $O(n)$

Optimal Algorithm for list ranking

So, what we have found is that we can 3 color link list in order of log k n time using n by log k n processors on an EREW PRAM. This is an optimal algorithm because the cost of this algorithm is order of n which is exactly the cost of a sequential algorithm for colouring a list. In fact, sequentially you can 2 color a link list in order of n cost. In the next lecture we will find an optimal algorithm for list ranking which has some similarities with the algorithm that we have just seen.

So, I presented this algorithm as a precursor to the list ranking algorithm that we shall see in the next class. So, that is it from this lecture hope to see you in the next lecture.

Thank you.