Parallel Algorithms

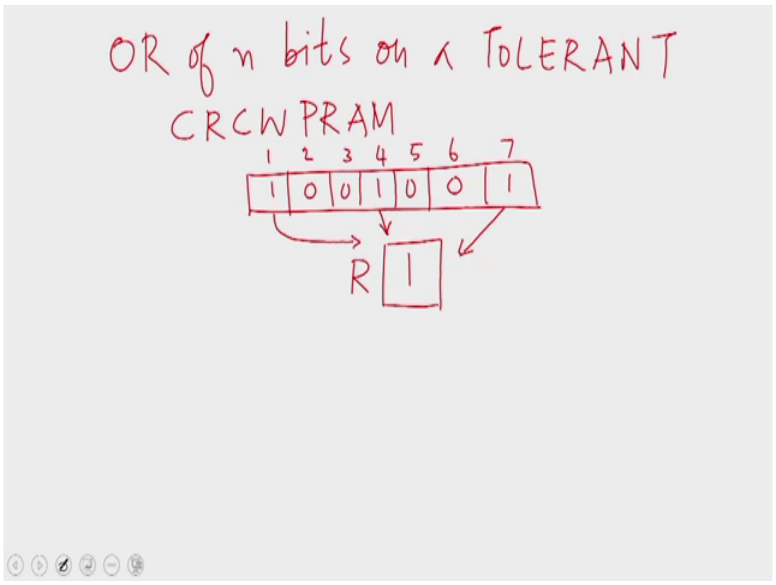
Prof. Sajith Gopalan
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 02
Shared Memory Models – 2

(Refer Slide Time: 00:30)

Welcome to the second lecture of the NPTEL MOOC on Parallel Algorithms. In the first lecture of the MOOC, we familiarize ourselves with the model of computation called parallel random access machine. We also saw an algorithm for finding the OR of n bits on the common CRC WP ram. In particular we had seen the variance of params like exclusive read, exclusive write param, concurrent read, exclusive write param and various categories of concurrent read concurrent write param. So, we will take the discussion further now.

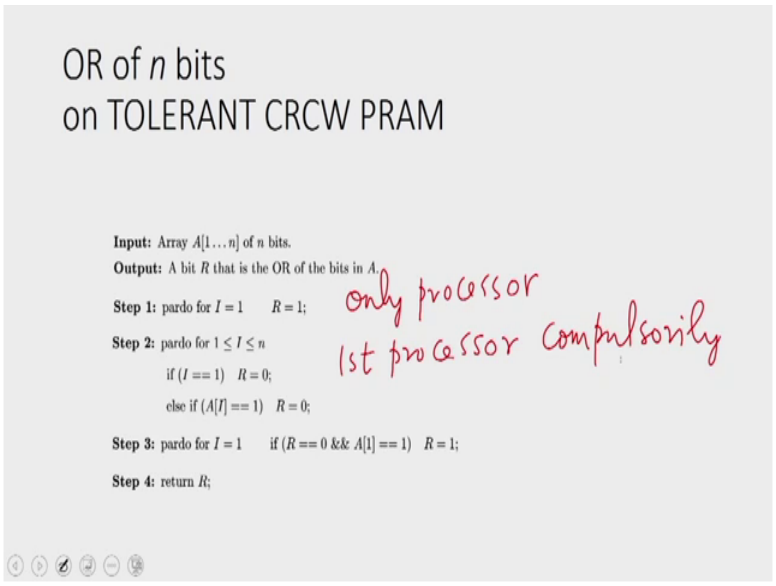
(Refer Slide Time: 01:15)



Let us say we want to find the OR of n bits on a TOLERANT CRCW PRAM. So, let us say we are given an array of size n, in which each location holds a bit. Let us say we need to find the OR of these bits. As we did in the case of the common algorithm, here also we take a location that we named R in which at the end of the algorithm the result is supposed to hold and then we proceed in this fashion. Initially only processor 1, we will initialize this location to contain 1. So, at the end of the first step, only the first processor has come forward and initialized the location to 1.

So, R now contains 1. Then in the second step all the processors that have a 1 will come forward and processor 1 will compulsorily participate in the second step irrespective of whether its bit is 0 or 1 and all of them will simultaneously attempt to write a 0 n R, R already contains 1. Now all of them are attempting to write 0 in location R. Since this is the tolerant model, a concurrent write is merely tolerated. Therefore, if there is a write conflict the content of the location will not change. So, in this case the content of the location does not change. And then finally, in the third step, the first processor looks at this location and finds that the location contains 1 and its own bit as 1. Therefore, there is no further action taken the answer of the problem is 1. So, what exactly are the processes doing here.

(Refer Slide Time: 03:34)



OR of *n* bits on TOLERANT CRCW PRAM

Input: Array $A[1 \dots n]$ of n bits.
Output: A bit R that is the OR of the bits in A.

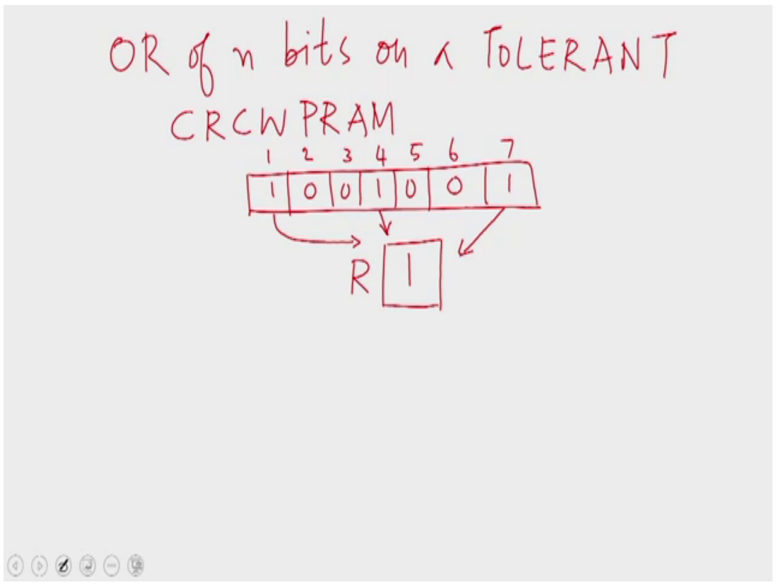
```
Step 1: pardo for $I = 1$        $R = 1$;
Step 2: pardo for $1 \leq I \leq n$
            if ($I == 1$)   $R = 0$;
            else if ($A[I] == 1$)   $R = 0$;
Step 3: pardo for $I = 1$        if ($R == 0$ && $A[1] == 1$)   $R = 1$;
Step 4: return $R$;
```

only processor
1st processor compulsorily

So, let us look at the code of the algorithm. The algorithm proceeds in this fashion. In the first step, all the processors do not participate. This is a tolerant model, this is a variant from the common algorithm. In the common algorithm, all the processes together initialize the variable R to 0 whereas, here only the first processor is initializing the variable to 1. This is because in a tolerant model whenever there is a write conflict, the content of the location will not change. So, for the initialization to work exactly one processor should initialize. Therefore, here we have only the first processor. So, this is at variance with the common algorithm. In the common algorithm all the processes initialized the variable together whereas, here exactly one processor is scheduled to initialize the variable.

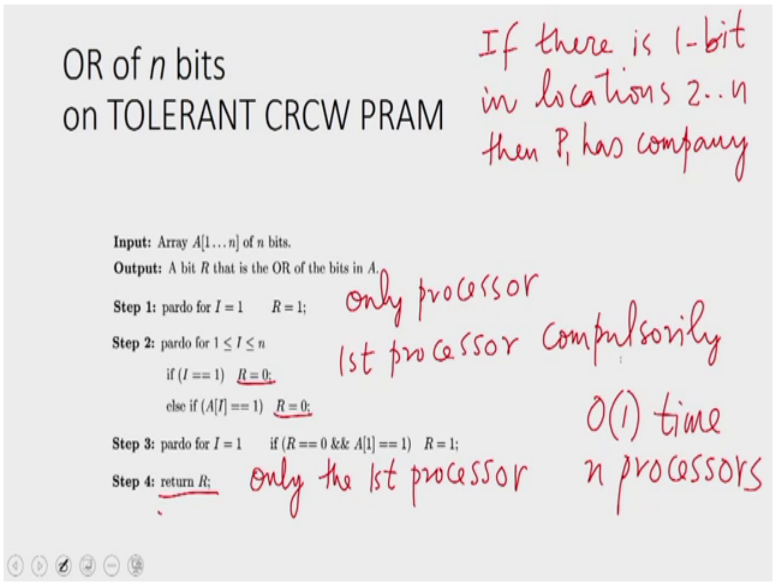
After this, all the processors together in parallel will do this. If I equal to 1, then R is set to 0 which means the first processor is compulsorily participating. The other processors will join the first processor if their bits are 1.

(Refer Slide Time: 05:05)



So, as we saw in the previous example, the processors sitting on locations 4 and 7 which are p 4 and p 7 will participate in the right, that is precisely because their bits are 1. Processes p 2, p 3, p 5 and p 6 will not participate, because their bits are 0.

(Refer Slide Time: 05:21)



OR of n bits on TOLERANT CRCW PRAM

If there is 1-bit in locations 2..n then P_1 has company

Input: Array $A[1 \ldots n]$ of n bits.
Output: A bit R that is the OR of the bits in A.

```
Step 1: pardo for $I = 1$      $R = 1$;
Step 2: pardo for $1 \le I \le n$
            if ($I == 1$)  $R = 0$;
            else if ($A[I] == 1$)  $R = 0$;
Step 3: pardo for $I = 1$      if ($R == 0$ && $A[1] == 1$)  $R = 1$;
Step 4: return $R$;
```

only processor

1st processor compulsorily

only the 1st processor

O(1) time

n processors

Processor 1 is compulsorily participating whereas, the other processors are participating conditionally. They will participate only if their bit is 1. So, R is initialized to 0. So, these two initializations in fact, are synchronous. Processor 1 and the other processes that are operating conditionally will be attempting to change the contents of R to 0 simultaneously. So, what are we achieving by this? We are checking whether there is any processor other than 1 that has a 1 bit. If at least 1 processor other than 1 has a 1, then all those processes will be attempting along with processor 1 to change R to 0.

So, if there is a conflict at all that is, if there is a 1 bit in locations 2 to n, then all those corresponding processors will be attempting to reset R along with processor 1; then all of them will provide company to processor 1. So, this allows us to check whether there is at least 1 1 in the range 2 to n. So, there are multiple cases. Suppose there is no 1 in the range from 2 to n, in this case process of 1 will succeed in changing R to 0. So, R equal to 0 now indicates that the R of the positions 2 to n is 0, then in the third step what we do is this if R equal to 0 and A 1 equal to 1. We are setting R equal to 1, which means if R equal to 0 in any case that is if there is no 1 in locations 2 to n, then the condition which will allow are to be set to 1 in the third step is the first bit being 1.

So, unless the first bit is 1, R will remain 0 which it should because in that case there would be no 1 in the array, to recap if there is not a single 1 in the array, then in particular in positions 2 to n there is no 1. Therefore, there will be no company to p 1 in the second step p 1 will be resetting R alone. Therefore, p 1 will succeed in resetting R and R will end up being 0.

Then in the third step if A 1 also happens to be 0, R will not be changed, R will continue to be 0 and that is what the answer is going to be. The second case is where there is at least 1 1 in positions 2 to n in which case in step 2 processor 1 is going to get company, there will be a write conflict. When there is a write conflict, the content will not change; the content will remain 1 that we had set in the first step.

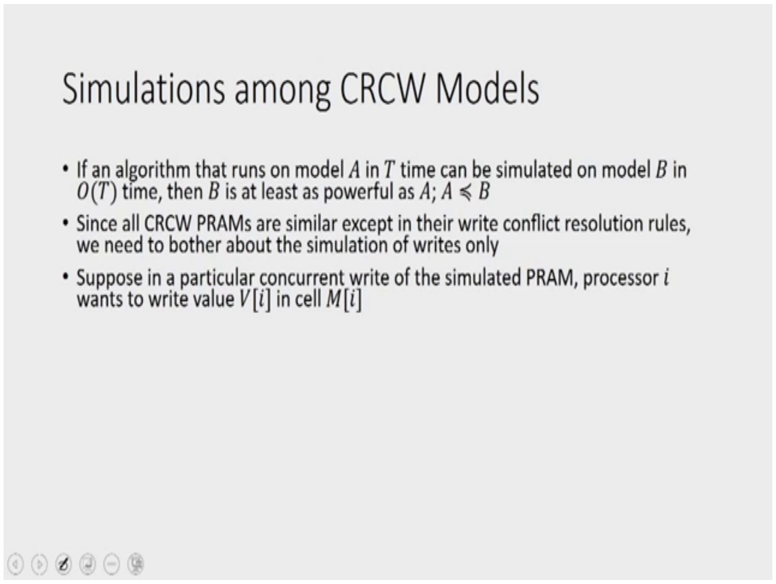
Therefore at the end of the second step R will be 1 indicative of the fact that there is at least 1 1 in positions 2 to n. If R equal to 1, then we do not do anything and we come out that is precisely how it should be there is at least 1 1 in positions 2 to n and the result does 1 indeed.

Now the third cases where there is no 1 in positions 2 to n, but the first bit is 1 in which case at the end of the second step, we will have R equal to 0 and then in the third step we check the condition is R equal to 0 and A 1 equal to 1 that condition is true, then R will be set to 1 which is how it should be.

So, in the 3 cases in all the three cases we find that the algorithm works correctly and the total time taken by the algorithm happens to be order of 1. In step one only process of 1 operates, so, it is an order 1 it is a 1 step operation. In the second step, all the processes are participating, but every processor has only 1 right to do apart from 1 condition check. Therefore the second step also can be executed in order 1 time. And the third step has only the first processor operating and all that is done is a condition check and an assignment. Therefore, the third step also runs in order 1 time. In the fourth step the return that is being performed will have to be performed only by the first processor.

So, here this is wrong, only the first processor should be returning the value or any one of the processors. This has to be conditional. It should be pardo for I equal to 1 return R. So, only the first processor will be returning the value. So, the algorithm works correctly in order of 1 time using n processors. So, on the tolerant model as well we are able to find the R of n bits in order of 1 time exactly as we did in the case of the common algorithm, but as you can see the algorithm is lot more complex. And we shall see that algorithms on tolerant generally tend to be far harder to prove for correctness and to analyze than algorithms for common. Therefore, we will we usually find that common is a lot more used in the literature.

(Refer Slide Time: 11:23)



Simulations among CRCW Models

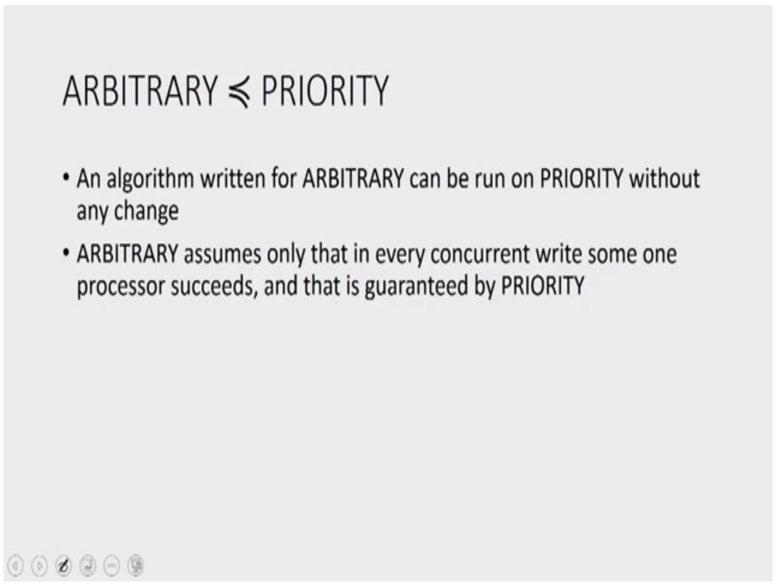
- If an algorithm that runs on model A in T time can be simulated on model B in $O(T)$ time, then B is at least as powerful as A; $A \preccurlyeq B$
- Since all CRCW PRAMs are similar except in their write conflict resolution rules, we need to bother about the simulation of writes only
- Suppose in a particular concurrent write of the simulated PRAM, processor i wants to write value $V[i]$ in cell $M[i]$

So, we have seen several varieties of CRCW PRAM models. How are these models related to each other? Is it possible to simulate these models on each other? That is once we have designed an algorithm for one of these models, is it possible to run that algorithm on 1 of the other models with modifications and if we do that what exactly is the cost of such a simulation? So, that is what we are going to explore now.

Let us say an algorithm runs on model A in T time and suppose it can be simulated in model B in order of T time, then we say that B is at least as powerful as A and it is denoted in this fashion. It is read as B is at least as powerful as A since all CRCW PRAMs are similar except in their write conflict resolution schemes. We need to bother only about the simulation of the writes that is as we saw in the last class each instruction of a PRAM consists of 3 phases. There is a read phase, there is an execute phase and there is a write phase.

So, consider of an instruction that is to be run on model A and let us say we want to simulate this on model B. The read phase can be executed exactly as it is and the execute phase also can be executed exactly as it is and a simulation needs to be done only for the write cycle. So, for the purpose of the simulation let us assume that in a particular concurrent write of the simulated PRAM, processor i wants to write value V i in location M i. With this assumption, let us see how the models can be simulated on each other. First I shall argue that, the priorities here CRCW PRAM is at least as powerful as the arbitrary CRCW PRAM.

(Refer Slide Time: 13:12)



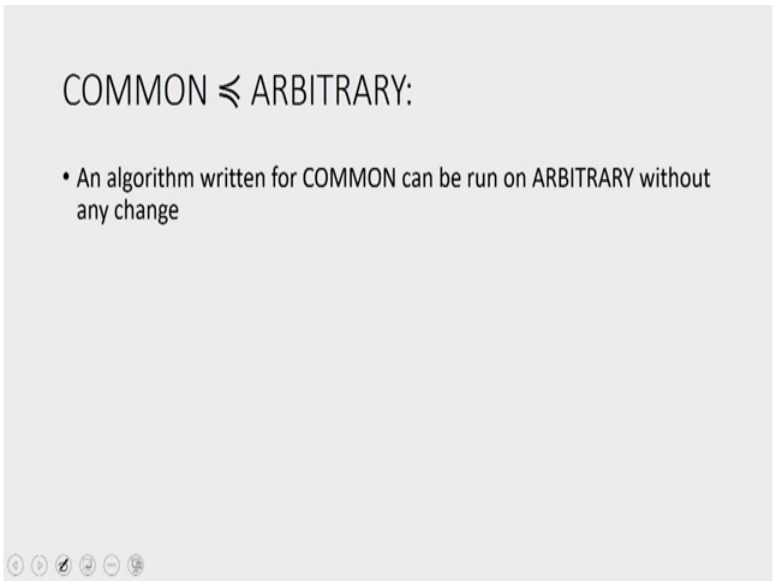
ARBITRARY $\preccurlyeq$ PRIORITY

- An algorithm written for ARBITRARY can be run on PRIORITY without any change
- ARBITRARY assumes only that in every concurrent write some one processor succeeds, and that is guaranteed by PRIORITY

An algorithm that is written for the arbitrary CRCW PRAM model can be run on priority without any change. This is because the arbitrary CRCW PRAM assumes that out of every conflicting set of processors an arbitrary 1 will succeed and the algorithm designer is constrained to assume that the identity of the succeeding processor will not be known a priori. That is at the time of designing the algorithm he or she would not have any idea of which of the conflicting processes is going to succeed. Therefore, the designer is supposed to deal with that uncertainty in the design stage, but in a priority model that uncertainty does not exist. The designer knows that the processor with the highest priority is going to win. Therefore, the processor the designer has a greater freedom on the priority model.

Now here we have an algorithm that is written for the arbitrary model. Therefore, the designer has taken on a greater constraint, the designer has assumed that only one of the processors will succeed, but has not assumed that any one particular model will succeed. Now priority guarantees that, priority ensures that out of every conflicting set of processes, one is guaranteed to succeed and the succeeding processor is going to be the same one in one execution in execution after another. This is what is not guaranteed in arbitrary. But, whatever arbitrary requires is provided by the priority model. Therefore, an algorithm that is written for the arbitrary model can be run on the priority model without any change. Therefore, priority is at least as powerful as arbitrary.

(Refer Slide Time: 15:09)

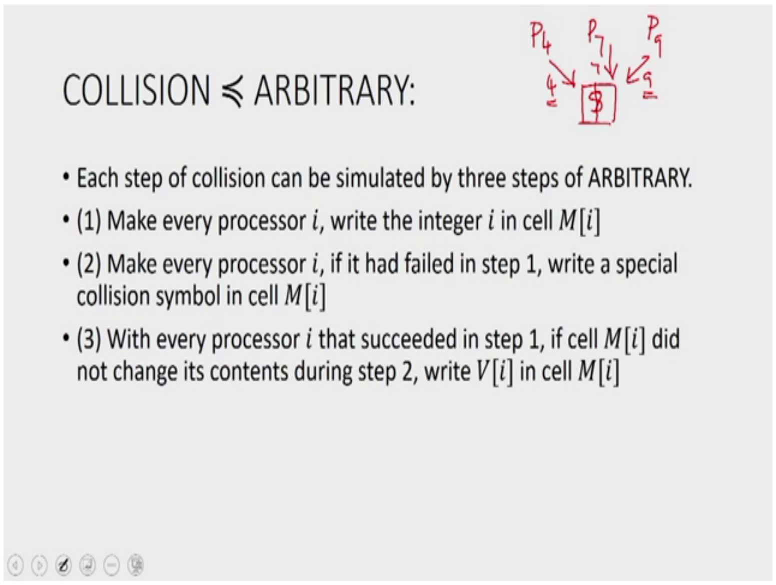


COMMON $\preccurlyeq$ ARBITRARY:

- An algorithm written for COMMON can be run on ARBITRARY without any change

Similarly we can also claim that arbitrary is at least as powerful as common. This is because on the arbitrary model, the designer assumes that, out of every set of conflicting processes an arbitrary 1 will succeed. Common makes a stronger assumption, common assumes that out of every set of conflicting processors that is every processor in a set of conflicting processors is attempting to write the same value. This is a stronger assumption than in the case of arbitrary. When you take an algorithm that is written for the common model and run it exactly as it is on the arbitrary model, the algorithm will work correctly because the now the processors are all attempting to write the same value in every conflict and one of them is going to succeed and the succeeding processor is getting in the value that the other process are attempting to write in any case. Therefore, an algorithm that is written for common can be executed on arbitrary without any change at all.

(Refer Slide Time: 16:12)



COLLISION ≼ ARBITRARY:

- Each step of collision can be simulated by three steps of ARBITRARY.
- (1) Make every processor i, write the integer i in cell $M[i]$
- (2) Make every processor i, if it had failed in step 1, write a special collision symbol in cell $M[i]$
- (3) With every processor i that succeeded in step 1, if cell $M[i]$ did not change its contents during step 2, write $V[i]$ in cell $M[i]$

Now we come to the collision model, I claim that the arbitrary CRCW PRAM is at least as powerful as collision. This is because a step of collision can be simulated on the arbitrary CRCW PRAM in three steps.

So, the simulation is carried out in this fashion. First, every processor i writes the integer i in the location in which wants to write. So, here first we are attempting to detect a collision, what we have is an algorithm that is designed for the collision model and this algorithm we want to simulate on the arbitrary model. So, the simulated algorithm is going to run on the arbitrary model and what we do is this in the first step, every processor i writes integer i in cell M i. So, let us say we have a set of conflicting processors, let us say p 4 p 7 and p 9 are all attempting to write into the same memory location.

So, in the first step all these are attempting to write their own indices in this memory location, p 4 will attempt to write 4, p 7 will attempt to write 7 and p 9 will attempt to write 9 all in the same memory location, but this step is being run on an arbitrary model that is because we are simulating the collision model on the arbitrary model. Therefore, exactly one of these processes will succeed.

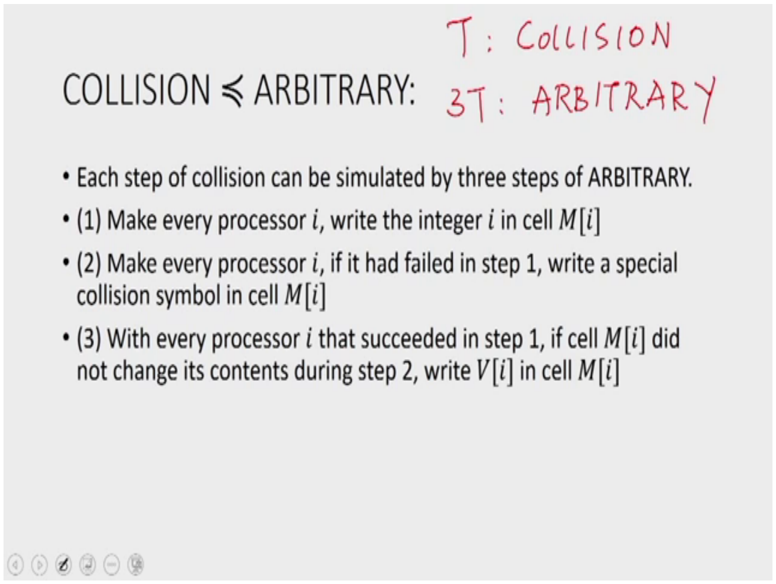
Suppose 7 succeeds here 4 and 9 failed. So, 7 manages to write 7 in this location. Now if these processors, read the value back, then they can know whether they have succeeded or not. So, processor 7 knows that it has succeeded, process of 4 and 9 know that they have failed. Now why did they fail? They failed because they had a conflict. So, when they attempted to attempted to write a value, but find that the value written was something else, then they realize that there was a collision and there was a conflict and some other processes succeeded in the conflict.

So, 4 and 9 are now aware of the conflict, but 7 is still not aware of the conflict. When 7 reads back what it seizes 7 indeed so, it has succeeded in getting its right value. It could have succeeded because it is alone and did not face a conflict or it could have succeeded because it was the lucky one in the conflict. So, except for process of 7 every other processor involved in this conflict knows that they have failed, then in the second step every processor that failed in the first step will write a special collision symbol in M i. So, 4 and 9 now realize that they have failed they have failed because they were in a conflict. Therefore, in the second step both of them will attempt to write the collision symbol in the memory location at least one of them will succeed. It does not matter which one succeeds, but one of them will succeed, since both of them are attempting to write the same value which is the collision symbol, the collision symbol will get inside. So, the location now contains the collision symbol.

Now processor 7 consider the processor 7, processor 7 succeeded in the first step. It could have succeeded because it did not face a conflict or it could have succeeded because it was the lucky one out of a set of conflicting processors. It can find out which of these cases it was now so, for every processor that succeeded in the step one let us read the location so, process 7 now goes through the location and reads it. It knows that it had succeeded in the first step, at the end of the first step it had 7 in the location; it had managed to write 7 in the location. But now, when it looks at the location it finds the collision symbol, then it realizes that the symbol collision symbol appeared in that location because there were some conflicting processes which in turn came in the second step and wrote the collision symbol overwriting the value which it had put in. Therefore processor 7 now realizes that there was a conflict. Therefore, in the third step what every processor does this, every processor that had succeeded in the step in the first step, checks if cell M i did not change its contents during the second step if it did not change its contents, then that could be only because there was no other conflicting processor to change its successful write into the collision symbol.

If there is no other processor that is had there been no 4 and 9 had processor 7 been alone which means had it not been conflicting in this write, then it would have succeeded in the first step and then when it write reads its value back it would not know whether it succeeded because it is alone or because it is the lucky one in the set of conflicting processors. But in after the second step when it finds that the value which it wrote still remains in that location it would realize that there was no other conflicting processor to change its value to the collision symbol. Therefore, in the third step it would go ahead and write the value it wanted to write in location M i.

(Refer Slide Time: 21:24)



COLLISION $\preccurlyeq$ ARBITRARY:

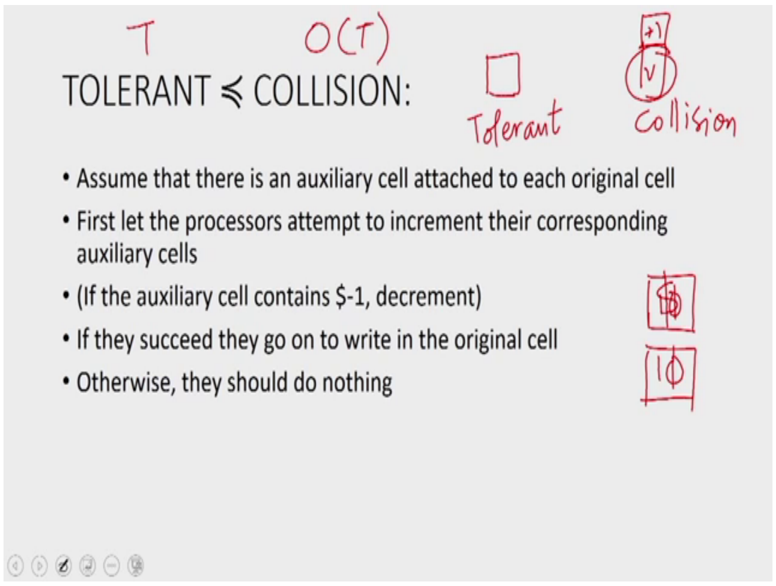
T : COLLISION

3T : ARBITRARY

- Each step of collision can be simulated by three steps of ARBITRARY.
- (1) Make every processor i, write the integer i in cell $M[i]$
- (2) Make every processor i, if it had failed in step 1, write a special collision symbol in cell $M[i]$
- (3) With every processor i that succeeded in step 1, if cell $M[i]$ did not change its contents during step 2, write $V[i]$ in cell $M[i]$

So, in these three steps therefore, a step of the collision algorithm can be simulated on the arbitrary CRCW PRAM, what it means is that an algorithm which runs in T time on a collision CRCW PRAM and algorithm which runs in T time on the collision CRCW PRAM can be simulated in 3T time on the arbitrary CRCW PRAM which is of the same order. Therefore, we say that arbitrary is at least as powerful as collision. And finally, we claim that collisions at least as powerful as tolerant.

(Refer Slide Time: 22:14)



TOLERANT $\leqslant$ COLLISION:

- Assume that there is an auxiliary cell attached to each original cell
- First let the processors attempt to increment their corresponding auxiliary cells
- (If the auxiliary cell contains $-1, decrement)
- If they succeed they go on to write in the original cell
- Otherwise, they should do nothing

T
O(T)
Tolerant
+1
V
Collision
1 0

Given an algorithm on tolerant let us say we want to simulate it on collision. So, let us consider the right cycle of an instruction of the tolerant algorithm. To simulate this on collision we assume we have twice the memory that is the collision model is assumed to have twice the memory that the tolerant model has. Every cell of the tolerant model is assumed to have an auxiliary cell in the collisions in the collision model that is one single cell of the tolerant model is assumed to have an auxiliary cell in the collision model thereby doubling the memory requirement. So, the processors that attempt to write in a memory location will first attempt to increment their corresponding auxiliary cells.

So, to begin with the auxiliary cells will contain some garbage value, it does not matter what the auxiliary cell contains. All the processes that want to write will first read the auxiliary cell of the intended recipient of their rights and then increment these values and write the value back. So, this value is incremented by all the processors. Now this is happening on the collision model. Therefore, if there is in fact, a write conflict the value that a certain will be the collision symbol. So, let us say this auxiliary cell contained 10 to begin with. All the processes that want to write in this location will read the contents which is 10 and change it to 11 or they attempt to change it to 11.

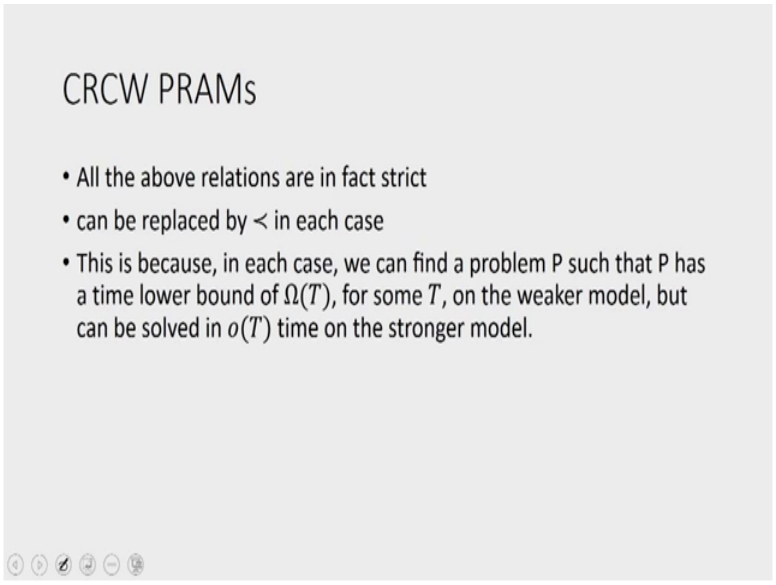
So, if there are multiple processes attempting to write in this location all of them are attempting to write 11 here, but then because of the write conflict they will not succeed in writing 11 here instead they would be writing just the collision symbol that is if there is a genuine conflict. But if there is no conflict, if there is only one processor attempting to write in this memory location, it will read the 10 that was there and write 11 instead. Therefore, when this processor looks at this location it knows that it is not in conflict, but of course, there is a special case we have to consider the case where the auxiliary contains the collision symbol minus 1.

The collision symbol is nothing, but a special integer. So, it is possible that an arbitrary location can contain the collision symbol minus 1 at any point in time. Therefore, if all the processors attempting to write in the memory location increments this value, then the location will end up containing the collision symbol. Therefore, even if there is no collision we might assume that the recirculation, but those can be easily circumvented. A processor which attempts to agreement the location, first checks what value it has written. Here it has read 10 and has attempted to write 11. Similarly if the value that it reads in happens to be the coalition symbol minus 1, then instead of incrementing they will attempt to decrement this value. Therefore, the value that is attempted to be written is indeed different from the coalition symbol.

Therefore the processor knows whether it has succeeded or not. If it finds that the coalition symbol has appeared in the auxiliary memory location, then it knows that it has failed and there is a conflict in which case it would do nothing. The tolerant model is assumed to do nothing whenever there is a conflict. Therefore, the contents of the actual memory location which is this will be left as it is. On the other hand when it finds a coalition symbol when it does not find a collision symbol, then that is a clear signal field for going ahead and writing then the processor know says there is no conflict and the processor will go ahead and write the value that it wanted to write.

So, what this shows is that a step of a tolerant algorithm can be simulated on the collision CRCW PRAM in order of one time. Therefore, an algorithm that runs on the tolerant model in T time can be simulator on the collision model in order of T time. Therefore, collision is at least as powerful as tolerant.

(Refer Slide Time: 26:45)

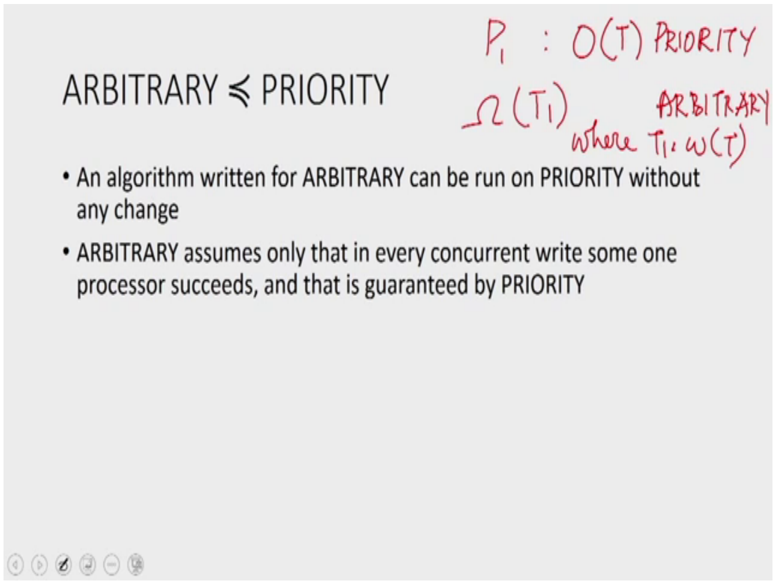


CRCW PRAMs

- All the above relations are in fact strict
- can be replaced by $\prec$ in each case
- This is because, in each case, we can find a problem P such that P has a time lower bound of $\Omega(T)$, for some T, on the weaker model, but can be solved in $o(T)$ time on the stronger model.

So, on the whole we have shown that priorities at least as powerful as arbitrary. Arbitrary is at least as powerful as collision and arbitrary is also at least as powerful as common and collision is at least as powerful as tolerant. In fact, we can show that all of the above relations are in fact strict. It can be replaced with strictly more powerful in each case. In particular you can show that priority is strictly more powerful than arbitrary.

(Refer Slide Time: 27:18)



ARBITRARY $\preccurlyeq$ PRIORITY

$P_1 : O(T)$ PRIORITY

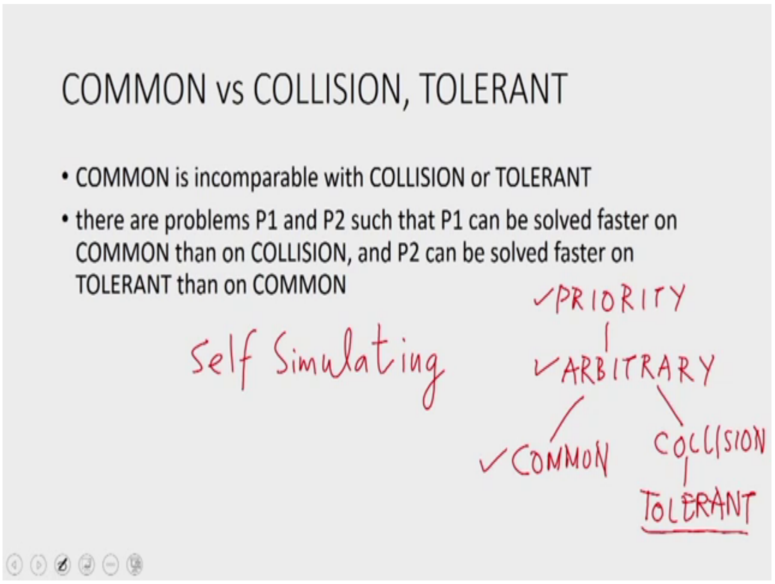
$\Omega(T_1)$ ARBITRARY

where $T_1 : \omega(T)$

- An algorithm written for ARBITRARY can be run on PRIORITY without any change
- ARBITRARY assumes only that in every concurrent write some one processor succeeds, and that is guaranteed by PRIORITY

This can be established by showing that for some problem P 1, there exists an algorithm they turns in order of T time on priority and by showing that for the same problem P 1 on arbitrary there is a low bound of omega of T 1, where T 1 is omega of T its omega of T which means there is a problem P 1 which can be solved faster on priority than on arbitrary. There is a low bound proof which precludes an equally fast solution on arbitrary. Similarly for the next inequality as well we can find such a problem. There exists a problem P 2 which can be solved faster on arbitrary than on common and the rest is a problem which can be solved faster on arbitrary than collision.

(Refer Slide Time: 28:42)



COMMON vs COLLISION, TOLERANT

- COMMON is incomparable with COLLISION or TOLERANT
- there are problems P1 and P2 such that P1 can be solved faster on COMMON than on COLLISION, and P2 can be solved faster on TOLERANT than on COMMON

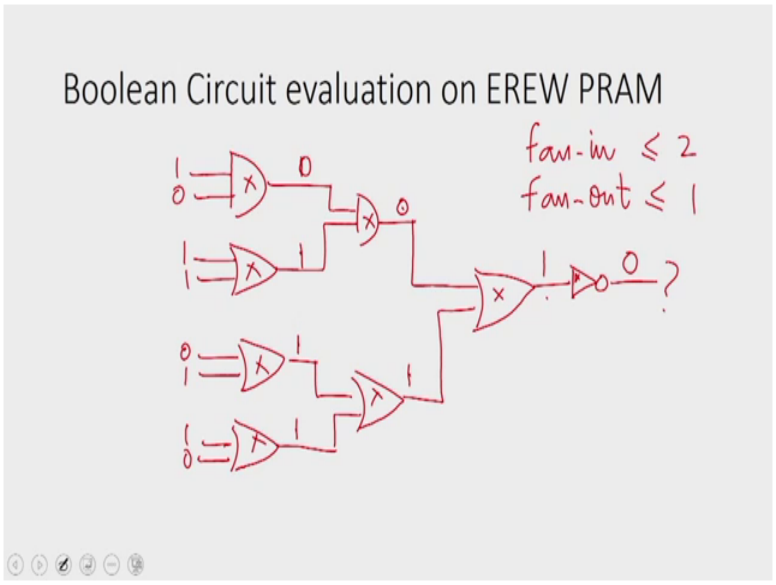
Self Simulating
✓PRIORITY
✓ARBITRARY
✓COMMON
COLLISION
TOLERANT

And there is a problem which can be solved faster on collision than tolerant which means all these inequalities are strict, but then how do collision and collision tolerant compare with common. This question becomes relevant because what we have just shown is that priority is more powerful than arbitrary which is more powerful than common and arbitrary is also more powerful than collision and collision is more powerful than tolerant. But then how do common compares with collision and tolerant? It can be shown that common is incomparable with collision of tolerant. That is because there are problems P 1 and P 2 such that P 1 can be solved faster on common than on collision and P 2 can be solved faster on tolerant than on common.

So, there are two problems P 1 and P 2 that establish that common and collision tolerant are incomparable with each other. In this course, most of our discussions of CRCW

PRAM algorithms would be confined to priority arbitrary and common models. In particular we will be mostly as to the tolerant model because the tolerant model is not known to have a nice property called self simulating. We would like our models to be self simulating. A model is said to be self simulating, if a larger PRAM of that model can be simulated on a smaller PRAM of the same PRAM; the same model for a proportionate slowdown. We shall show that all these models are self simulating except possibly for tolerant and self simulation is a crucial property. Since tolerance is not known to be self simulating, we do not use tolerant much. We have seen a couple of algorithms on the CRCW PRAM s of common and tolerant varieties.

(Refer Slide Time: 31:12)



Boolean Circuit evaluation on EREW PRAM

Let us now see some problems on the EREW PRAM or CRCW PRAM. Let us say we want to evaluate a Boolean circuit. A Boolean circuit is made up of the 3 logic gates AND, OR and NOT. Let us say we are given a Boolean circuit like this and let us say we are given the input values.

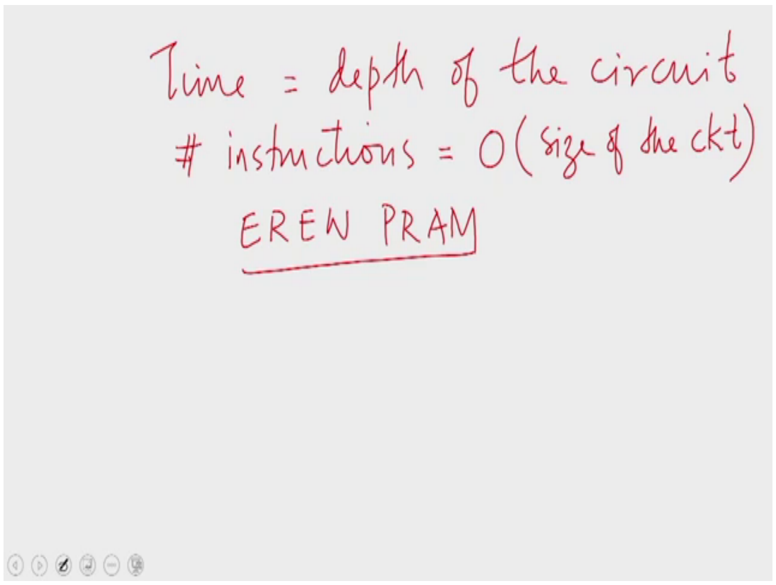
So, given this Boolean circuit and given the input values, we want to evaluate the output value. We know how to do this sequentially, we will evaluate the leftmost gates one by one. So, the outputs of the leftmost gates are solved one by one. Once these values are known, the second level gates can be evaluated. So, the outputs at the second level can now be calculated. Once these values are known, the third level gate can be evaluated and so on. So, this will require evaluating the gates in a certain order in this left right order. A gate can be evaluated if both of its inputs have already been evaluated.

So, sequentially we would require looking at each of the gates in turn and the cost of the algorithm would be linear in the number of gates. So, let us say we want to evaluate this in parallel on an EREW PRAM. We assume that the gates have a fan-in of 2 at the most the OR and AND gates have a fan in of 2 each and the NOT gate has a fan-in of 1. And let us say the fan out of every gate is at most 1, that is an output signal can be used by at most one input of another gate. So, in particular the output is not used at all. So, fan out could be 0 as well.

So, let us say we are given a Boolean circuit of this form which is to be evaluated on an EREW PRAM. So, for the for the purpose of the evaluation, let us say we have as many processors as there are gates. So, let us locate one processor on each of the gates. So, for this example you require 8 processors. So, let us say we have one processor stationed on each of the gates and then the gates can be evaluated in this fashion, consider the leftmost gates first, the inputs to these gates are all known. Therefore, their output values can be calculated simultaneously and in parallel.

So, the processors that are situated on the leftmost column will all be active simultaneously and they would produce their outputs at the same time. So, 0, this is 1, this is 1 and this is also 1. Now once the outputs of the leftmost column of gates is ready, the next set of gates can be activated. So, the gates which are at the second level can now be activated, they would in turn be evaluated in the second step all of them would be evaluated simultaneously. So, here we have a value of 0 and here we have a value of 1. Then the third level of gates will be activated. Therefore, here we will get a value of 1. Finally, the fourth level of the gates will be activated and we will get a value of 0.

(Refer Slide Time: 35:17)



Time = depth of the circuit

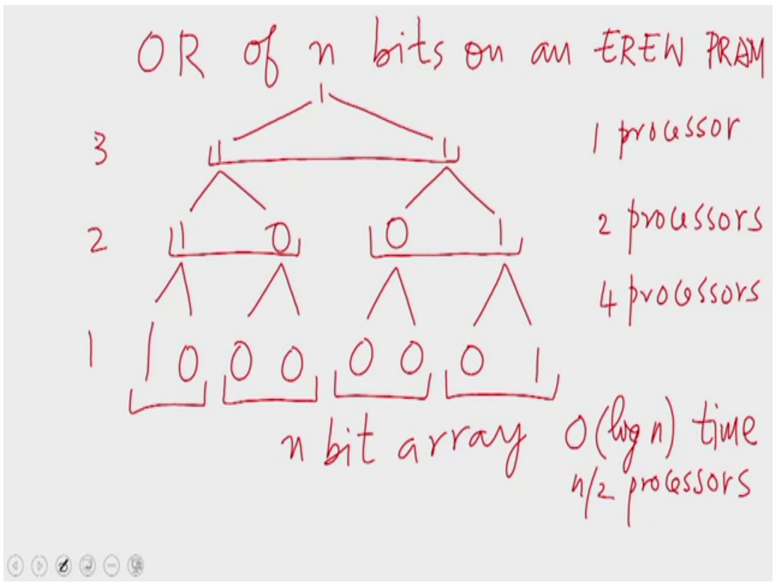
instructions = O(size of the ckt)

EREW PRAM

So, in 4 steps we can find that the circuit evaluates to 0. So, in general given a Boolean circuit on a EREW PRAM, we can evaluate the circuit in time that is equal to the depth of the circuit, but the total number of instructions executed is equal to the size of the circuit which is the number of gates in the circuit. So, as you can see in the example we saw, the size of the circuit is much more than the depth of the circuit. The running time of our simulation is equal to the depth of the circuit whereas, the total number of instructions executed is of the order of the size of the circuit.

The model is an EREW PRAM, this is because on none of the steps have we used concurrent reads and concurrent writes. Concurrent reads are not necessary because the fan out is at most one therefore, an output that is produced will not be read by more than one gate. Therefore, there is no need for concurrent reads and concurrent writes are also necessary not necessary because all the values that are written are on exclusively exclusive lines. Therefore, the modulus and EREW PRAM and the running time is equal to the depth of the circuit.

(Refer Slide Time: 36:53)



Another problem that we can study at this stage is that of finding the OR of n bits on an EREW PRAM. Let us say we are given an array of bits and we want to find the OR of them. We can divide the bits into odd even pairs, we consider the first bit and the second bit together, they form the first odd even pair, then with these form the second odd even pair, this will form the third odd even pair, this will form the fourth odd even pair. Then let us say we have 4 processors using these 4 processors we find the OR of these pairs. So, here we find that the OR is 1, here we find that the OR is 0, here again we find that OR is 0 and here we find that the OR is 1.

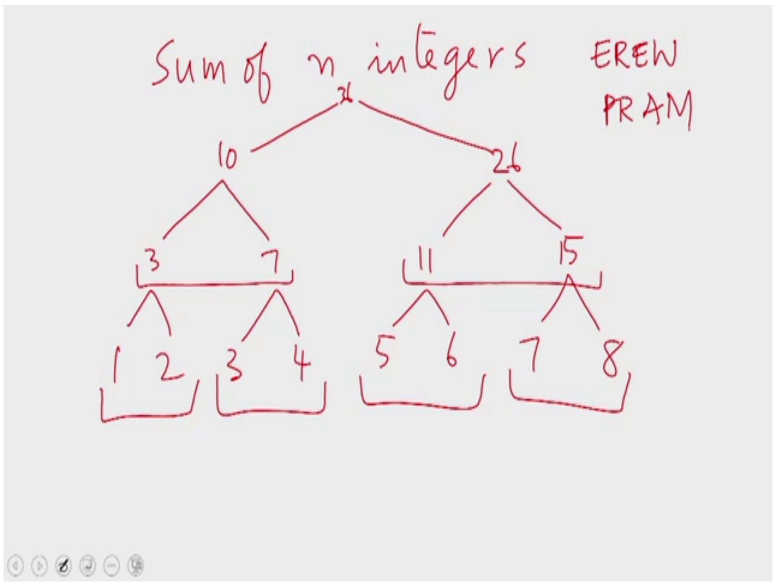
So, we have started with 8 bits now, we are reduced to 4 bits. If we find the OR of these 4 bits, we would have the OR of the original 8 bits. So, the size of the problem has been reduced by a factor of 2 using 4 processors. We depute at one processor for each odd even pair. Now these bits are again paired off, this is the first odd even pair this is the second odd even pair and we depute one processor to each the OR is found in this fashion. The problem size is now reduced to 2, but we have used 2 processors here. Now there are only 2 remaining bits, they will form one single odd even pair with one single processor; I can find the OR of them in one step.

So, here I have used one processor. The total time taken by the algorithm is 3. In the first step, I have reduced the number of bits from 8 to 4, in the second step I have reduce the number of bits from 4 to 2, then in the third step I have reduced the number of bits from

2 to 1. So, the algorithm runs in 3 steps and three comes from the total number of bits that we had we had 8 bits and 3 is the logarithm of that to the base of 2. In general if I had an n bit array, I could have found the logarithm of OR of these n bits in order of log n time and how many processes would I need to use? In the first step I would need n by 2 processors, in the second step I would require n by 4 processors and so on.

So, it would seem that I would require n by 2 processors to achieve running time of log n. So, when we have n by 2 processors running for order of log n time, the total cost of the algorithm is order of n log n which seems exorbitant considering that the or of n bits can be found in order of n time sequentially. We shall see that a scheduling technique called the brain scheduling technique will enable us to reduce the cost of this algorithm from order of n log into order of n later. We shall talk about brain scheduling principle in one of the future lectures.

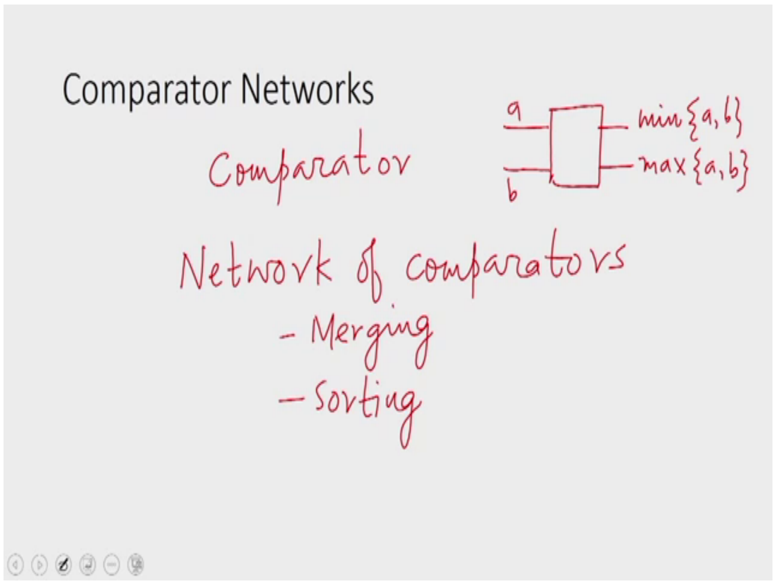
(Refer Slide Time: 41:04)



And analogous problem is that of finding the sum of n integers which again can be solved on a EREW PRAM in the same way. Suppose we are given 8 integers of which we want to find the sum. Again we can form odd even pairs, the sum of the first pair is 3, the sum of the second pair is 7, the sum of the third pair is 11, the sum of the fourth pair is 15; then we have odd even pairs of 3, 7 and 11, 15. Here we have 10 and here we have 26 and finally, the sum of all the numbers is 36.

So, in this fashion for the same cost that is order of log n time and n by 2 processors we can find the sum of n integers in on EREW PRAM. The model that we have used to EREW PRAM because there is no concurrent read anywhere and there is no concurrent write anywhere.

(Refer Slide Time: 42:33)



Comparator Networks

Comparator

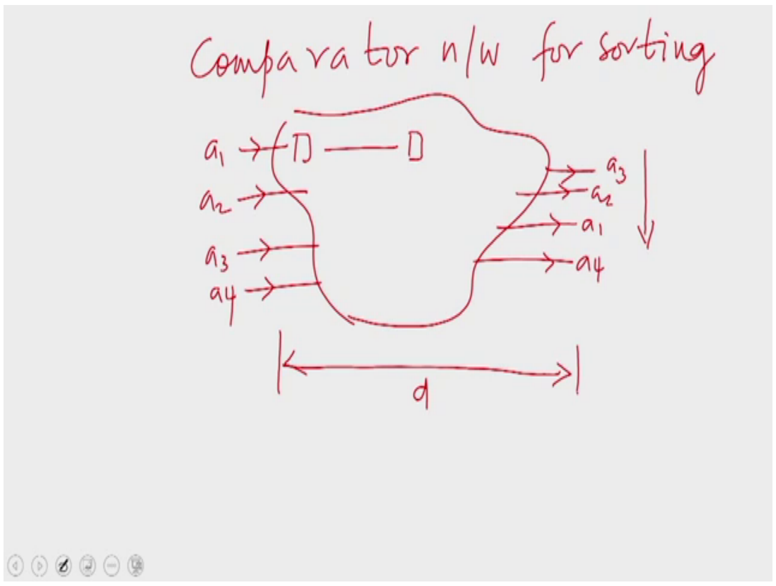
a, b → $\min\{a,b\}$, $\max\{a,b\}$

Network of comparators

- Merging
- Sorting

Another model of computation that we shall be using is what is called a comparator? A comparator is an operators that takes 2 values a and b and then produces the smaller of the 2 values on its upper output and the larger of the 2 values on its lower output. A network that is made up of comparators can be used for problems like merging or sorting.

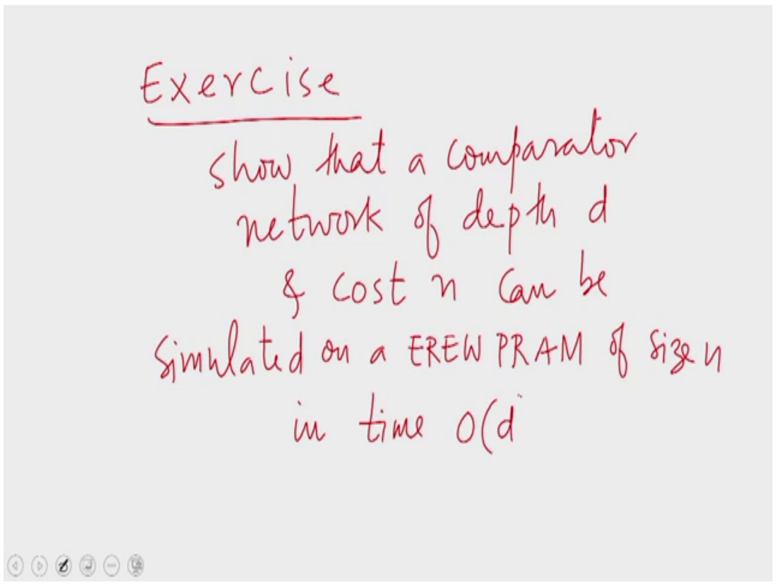
(Refer Slide Time: 43:43)



Let us say we have a comparative network for sorting. We shall study a couple of them later in this course. So, at this moment I am only asking you to visualize a comparative network. So, this network has several comparators interconnected in various ways. We have, let us say a sequence of inputs flowing in and a sequence of outputs flowing out. So, if we provide a set of integers here for example, these will be permuted in the output. So, that in the output they will be in increasing sorted order. So, a 3, a 2, a 1, a 4 will be the increasing sorted order of these elements.

So, imagine such a comparator network for sorting. The depth of this comparator network that is the length of the longest path in this comparator network will be the running time of the comparator network that is then once the inputs are fed in the inputs have to travel through the comparators and appear at the output. When they appear at the output we would have them in sorted order. So, let us say d is the depth of this comparator network which happens to be the length of the longest path in the comparator network, then d would be the time taken by the comparator network to sort a sequence.

(Refer Slide Time: 45:26)

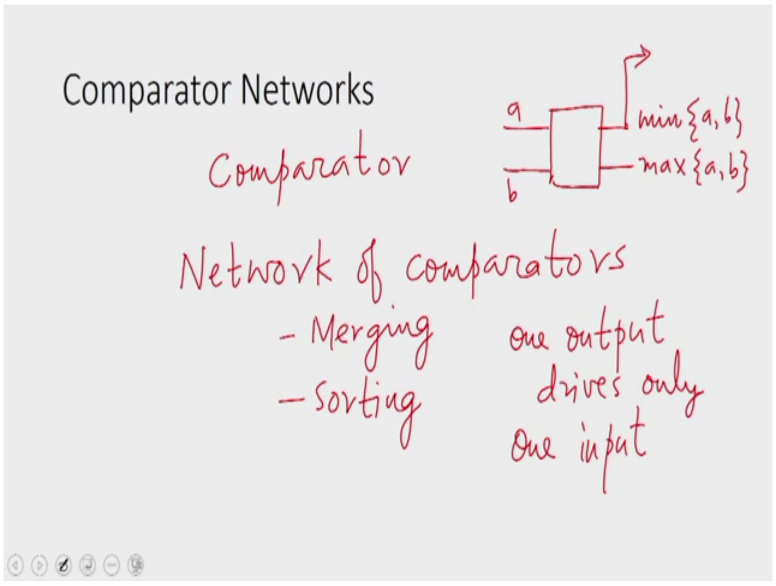


Exercise

Show that a comparator network of depth d & cost n can be simulated on a EREW PRAM of size n in time O(d

I shall give an exercise for you. Show that comparator network of depth d and cost n, the cost of a comparator network is the total number of comparator units you used in it, can be simulated on EREW PRAM of size n in time order of d. Of course, you can assume that one comparator output will be used only at one input this assumption is crucial.

(Refer Slide Time: 46:37)



Comparator Networks

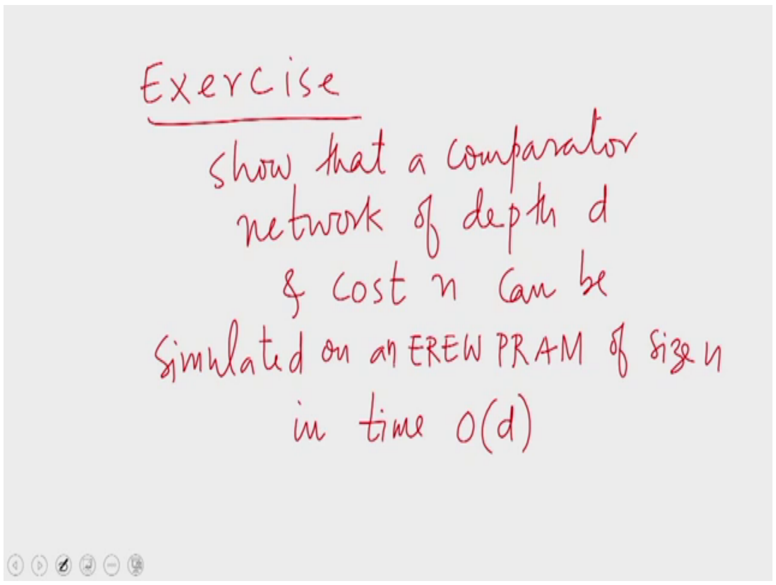
Comparator

Network of comparators

- Merging
- Sorting

One output drives only one input

(Refer Slide Time: 47:00)



Exercise

Show that a comparator network of depth d & cost n can be simulated on an EREW PRAM of size n in time O(d)

But with this assumption, what you need to show is that a comparator network of depth d and cost n can be simulated on EREW PRAM of size n in time order of t. So, with that we come to the end of this lecture, hope to see you in the next lecture.

Thank you.