

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

**Lecture – 01**  
**Shared Memory Models – 1**

Welcome to the NPTEL MOOC on Parallel Algorithms. This is the first lecture of the MOOC. I am Sajith Gopalan. I am a professor of Computer Science and Engineering at IIT Guwahati. I shall be your instructor for the course. Parallel algorithms deals with computations done in parallel, that is when you have multiple processors to perform a computation, what you use is a parallel algorithm. You would have done a course on algorithms in your B. Tech program which in which you study sequential algorithms.

A sequential algorithm is an algorithm that is run on a single processor machine whereas; a parallel algorithm is run on a multiple processor machine. In other words there are multiple agents for you to achieve your computation.

(Refer Slide Time: 01:14)

### Inherent sequentialities

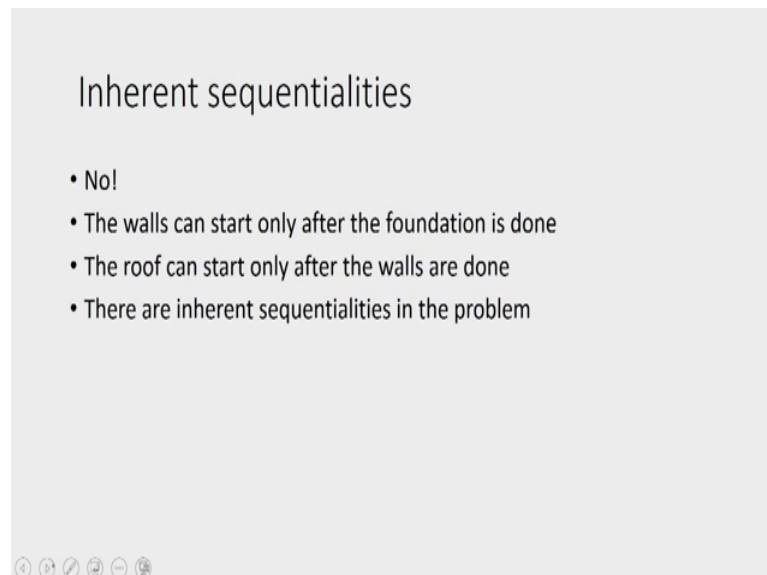
- Suppose a man works for 10000 days to build a house
- That is 10000 man-days of work
- If 50 men work together for 200 days, the house would be built
- If there are 100 men, the house could be built in 100 days
- A speed up of 2
- If we have 10000 men, can the house be built in a day?

Parallel algorithm attempts to find inherent sequentialities, within problems that we want to solve. Every task has some sequentiality. For example, consider the job of build in a house. Suppose, a man works for 10000 days to build a house, working alone he built he spends 10000 days to build the house.

That is 10000 man days of work, instead if we had 50 men working together for 200 days. The house would be built; still we have spent 10000 man days. What if we doubled the number of man? Instead of 50 let us say we have 100 man. Could the house be built in 100 days? Possibly, we achieve a speed up of 2 from the earlier case. When we had 50 man we took 200 days with 100 man we take 100 days a speed up of two.

But, over the 1 man case, we are getting a speed up of 100. What if we have 10000 man? Can the house be built in a day? The answer is an emphatic no that is because there are inherent sequentialities in the task of house building.

(Refer Slide Time: 02:20)



There are certain parts of that task that must be completed, before the other parts could be taken up. For example, the walls can start only after the foundation was done, the roof can start only after the walls are done.

These are the inherent sequentialities in this problem. So, when you consider any task, there will be some inherent sequentialities in the task, if you try to parallelize the task. Even if you have a large number of agents after a certain time, you will not be able to get any further speed up, that is with even a larger number of agents, you would not be able to speed up the process.

(Refer Slide Time: 02:59)

### Inherent sequentialities

- Suppose a problem  $P$  can be solved in  $T_1(n)$  time with one processor
- Here  $n$  is the problem size
- Suppose with  $p$  processors  $P$  can be solved in  $T_p(n)$  time
- The speed up is  $T_1(n)/T_p(n)$
- We would expect this to be  $p$
- But  $p$  may be impossible to achieve because of the inherent sequentialities of the problem

*$\omega(1)$  speed up*


So, where will that boundary begin, that is one chief field of investigation within parallel algorithms. Consider computational task a problem  $P$ . Suppose, this problem  $P$  can be solved in  $T_1(n)$  time with 1 processor; this is analogous to one man building a house in 10000 days. Here  $n$  is the problem size. So,  $T_1(n)$  is the time with one processor. Suppose, with  $p$  processors, where  $p$  greater than 1 this problem  $P$  can be solved in  $T_p(n)$  time. The speed up achieved is  $T_1(n)$  divided by  $T_p(n)$ .  $T_p(n)$  of course, would be less than  $T_1(n)$  because, if you have more processors you would be able to solve the problem faster. Like, when we had 50 man we were able to solve the problem in 200 days, which is a speed up of 50.

So, usually we would expect the speed up achieved with  $p$  processors to be  $p$ . But, this is not always the case. As we saw when we had 10000 man, we were still not able to, we were not able to build the house in 100 in a single day. So, for a very large  $p$ , it may not be possible to achieve a speed up of  $p$ . But, still it might be possible to obtain an  $\omega(1)$  speed up, a small  $\omega(1)$  speed up still. The small  $\omega(1)$  speed up, would still be achievable with any  $p$ .

(Refer Slide Time: 04:23)

## Parallel Algorithms

- A branch of study where
  - The inherent sequentialities of algorithmic problems are explored
  - We design algorithms that run on multiple processors
  - The design goals are
    - minimize the running time and
    - Minimize the total number of instructions executed




So, parallel algorithms is a branch of study, where we study the inherent sequentialities of algorithmic problems. We design algorithms that run on multiple processors. When we do this, our design goals are for one minimizing the running time, and the other is minimizing the total number of instructions executed. In the conventional algorithms course, you saw problems for which we were attempting to minimize the running time.

(Refer Slide Time: 04:54)

## Multiprocessors

- In a sequential algorithm, most often, we try to minimize the running time
- This is the same as the number of instructions executed
- There is only one processor
- In a parallel algorithm, we assume multiple processors
- The running time is smaller than the number of instructions executed
- Minimizing one does not necessarily minimize the other
- And we want to minimize both!



But, for those problems the running time was identical to the total number of instructions executed. This is not the case with the parallel algorithm, when you have only one

processor, the running time is identical to the number of instructions executed. But, in a parallel algorithm, we assume we have multiple computing agents. All these computing agents are alive all the time; that is throughout the execution of the algorithm, all these computing agents are in operation. Therefore, they are all contributing to the cost of the algorithm; they are all executing instructions simultaneously. Therefore, the total number of instructions executed, would be the sum total of all the instructions executed by all the agents. Whereas, the running time would be the parallel running time like, in the case of 50 man working together to build the house in 200 days, the running time was 200 hundred days.

But, the cost of the operation was still 10000 man days. So, the running time is one thing the cost is another thing in the parallel sitting. In the sequential setting these 2 are identical. The running time is smaller than the number of instruction is executed; usually the running time is less than the cost, when you have multiple agents as I said you would expect to get a small omega of 1 speed up therefore, the running time will be less than the number of instructions executed. Minimizing 1 does not necessarily minimize the other. Therefore, here we have more challenges, than in the sequential setting. In the sequential algorithm setting, we try to minimize one parameter which is the running time. Here we try to minimize 2 parameters the running time as well as the cost. Minimizing 1 does not necessarily minimize the other, and we may want to minimize both, that is why we have greater challenges here.

(Refer Slide Time: 06:38)

### A Reality Check

- In reality, multiprocessors tend to have only a small number of processing elements
- We shall see algorithms that run in  $O(\log n)$  time using  $n$  processors, for example, where  $n$  is the problem size
- When  $n$  is large, is there a disconnect from the reality? Not quite
- For one, we can simulate the algorithm on a real machine
- For another, letting the number of processors grow with the problem size allows us to explore the inherent sequentialities of the problem
  - Design techniques
  - Classification of problems

$O(1)$  time  
 $O(n^2)$  processors

But, then it is good to have a reality check, we are designing algorithms for multiprocessors, and in reality multiprocessors tend to have only a small number of processing elements. Because, of various hardware constraints, we have not been able to design multi processors with very large number of processors. But, in this course, we will be talking about algorithms, that use a large number of processors. For example, we shall see algorithms that run in order of  $\log n$  time, using  $n$  processors, where  $n$  is the problem size.

So, if  $n$  tends to say infinity the number of processors tends to infinity, that is found very large instances, we assume a large number of processors. We might also see algorithms that run in order 1 time, using order of  $n^2$  processors. For example, which means the number of processors quadratic in problem size, is there a disconnect from reality here? Not quite, there is a purpose in what we are doing. We can always simulate the algorithm that we design on a real machine. What I mean by a simulation is this? Let us say, we design an algorithm, assuming  $n^2$  processors, for a problem size of  $n$ . If  $n$  runs into thousands then the number of processors required would run into millions.

But, let us say we have only hundreds of processors. Using these hundreds of processors, we can execute every single step of the million processor machine. But, then in the million processor machine we assume that every step is executed by a million processors. But, if we in reality have only 100 processors, these steps will have to be simulated, that is we will take 100 processors, let them pretend to be the first 100 processors initially. After completing the work of the first 100 processors, these same 100 processors will pretend to be the second set of processors, and so on.

So, the million processors that we have will be measured in units of 100. That is in one parallel step we had 10 one million processors executing simultaneously, instead now we have only 100 process executing simultaneously. (Refer Time: 09:00) So, these 100 processors will pretend to be the first 100 processors, it in the first round of simulation, than in the second round of simulation they will pretend to be this second set of 100 processors and. so on.

So, these 10 processors can simulate, the 1 million processors one after the other. All that happens is that, what would have executed in 1 single step, would now take several steps. But, that will be 1 million times 1 million divided by 100 that will be the amount

of time you will have to spend on that one step. Then the cost does not change, when the number of processors has reduced from 1 million to 100 the speed up is lost. The time required goes up proportionately. Therefore, we are actually not losing in cost, the algorithm that we have designed using a large number of processors, can be simulated on a machine with a small number of processors, for the same cost.

Provided the model satisfies certain conditions, which we shall see, later by and by. Therefore, our algorithms are not going waste, the algorithms that we design, can in fact be simulated on actual machines with a small number of processors. For another, letting the number of processors grow with the problem size allows us to explore the inherent sequentialities on of the problem. It is thus that exposed the inherent sequentialities in the house building project, when we had only 1 man the house building project finished in 10000 days, when we increased the work force to 50 man, we had a speed up of 50, when we increase to the 100, we had a speed up of 100.

But, when we increase it, increase the number of men beyond 100; we will not get a proportionate speed up, the speed up starts slowing down. That is because, the inherent sequentialities of the problem begin to show. So, when we consider an algorithmic problem and attempted to solve with a large number of processors to attain the maximum parallelism that we can achieve, the inherent sequentialities so the problem will begin to show. Once we classify the problems based on their inherent sequentialities, we can decide which problem is hard and which problem is easy, we can classify problems accordingly and for different classes of problems we can evolve different design techniques.

So, the work that we are going to do is indeed has a connection with reality, it is not disconnected with reality as some would assume. In any discourse on algorithms, the participants of the discourse must first agree on the model of computation. The model of the computation, the model of computation is the more the machine on which the participants would agree to design algorithms, if a consensus is not evolved, then the participants will be talking about algorithms for different machines and they would not be on the same platform and there could be confusions.

(Refer Slide Time: 12:09)

## Models of Computation

- Consensus needed on which multiprocessor we work on
- How do the processors communicate with each other?
- Shared memory model: PRAM
- Interconnection networks: arrays, meshes, hypercubes
- Synchronous or asynchronous
- This course deals with synchronous algorithms

Therefore, what we need is first, need first is the consensus on which machine we would design the algorithms.

So, in the context of parallel algorithms we are talking about multi processors. So, we need to evolve the consensus on which multiprocessor we shall work on. Now, in the real world there is a plethora of multi processors, it is not easy to choose one and it is harder to get a consensus, we are embarking on a theoretical study of parallel algorithms. So, the machine that we are talking about need only be a theoretical one.

So, we will assume that we have multiple processors and we will assume each processor is like a random access machine, which is used in the design of sequential algorithms and it is an introduction to random access machines, can be found in the textbook by Aho Hopcroft and Ullman titled design and analysis of algorithms. I will encourage you to look up the introduction to random access machine in that textbook.

So, we shall assume that in our models, each processor is a random access machine. We shall come to the details of random access machine shortly. So, we assume that every processor is a random access machine. Now, in the context of parallel algorithms, we want the computing agents to cooperate with each other in solving the problem, which means, they have to communicate with each other.



So, the model should provide a way for the processors to communicate with each other. So, that is one aspect that we have to fix, how do the processors communicate with each other. There are various ways for the processors to communicate with each other; there is a model in which there is a shared memory, a memory that is shared by all the processors. So, the processors can communicate through the shared memory, one processor will write into the shared memory and the other processor will read from the shared memory, then a message has gone from the first processor to the second processor.

So, shared memory can be used, as a way of communicating between the processors. Such models are called parallel random access machine models. So, we shall study parallel random access machine model algorithms in detail, most of the first half of the course will be occupied with that. In the second half of the course, we shall see some inter connection networks. In inter connections networks, there are connections between processors, there is no shared memory.

But, processors can communicate with each other, through connections between them. So, through the connections, the processors can send and receive messages from their neighbors. So, we assume that for every processor, there is a fixed number of interconnections to some of the other processors, it is not necessary for every processor to be connected to every other processor. So, each processor can have a subset of the other processors as its neighbors because, it has connections to them.

So, in any one single step, each processor of the machine can communicate with its neighbors, by sending messages along the connections. Then of course, we have to answer the question is our model a synchronous model or an asynchronous model. In a synchronous model, all the processors will be executing in lock step, all of them will be fed the same clock. But, in an asynchronous model, each processor will have its own clock. This course mostly deals with algorithms for synchronous machines.

(Refer Slide Time: 15:40)

## Parallel vs Sequential Algorithms

- processor allocation
- Synchronization
- resource sharing
- There is no consensus on what is the ideal model of computation to be used in designing parallel algorithms.

There are some differences, when we between parallel algorithms and sequential algorithms, the task of designing parallel algorithms is more challenging.

That is because, we have to deal with a process or a location, that is we have a number of computing agents, now we have to decide, which computing agent will handle which part of the job. That is what is called processor allocation. Then there is the issue of synchronization. That is what we said just now in this course we will be dealing mostly with synchronized algorithms, resource sharing. In fact, there is no consensus on, which is the ideal model of computation to be used for designing parallel algorithms, in literature that is why you find different models in use and parallel algorithms historically have been designed on a number of models.

But, then to get on to the same platform we can try to simulate these models on each other, we shall see some of that soon.

(Refer Slide Time: 16:37)

## Parallel Random Access Machine (PRAM)

- Has  $p$  processors
- All having access to a shared random access memory
- Synchronous: all processors are fed the same clock
- Each processor is similar to a random access machine, the standard model of sequential algorithm design, and is assigned a unique index from the range  $[1...p]$ .

So, we will begin with an introduction to Parallel Random Access Machine. In a parallel random access machine, we have  $p$  processors. All the processors have access to a shared random access memory and then we assume that all the processors are synchronous, in that all of them are fed the same clock. Each processor is similar to a random access machine, the standard model of sequential algorithm design, and each processor has the unique index, that is if we have  $p$  processors. Let us say assume that the indices are the numbers ranging from 1 to  $p$ , a unique number has been assigned to each processor. Let us assume that the processor knows its index.

(Refer Slide Time: 17:18)

Random Access Memory

*potentially infinite*

*$n^k$  where  $k$  is a constant  $k \log n$  bits*

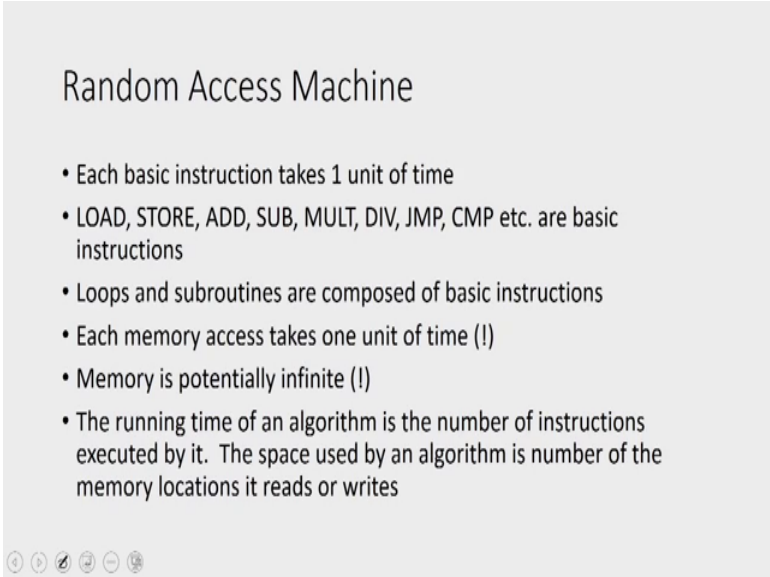
Address	Location
000000	
000001	
000002	
000003	
000004	
000005	
000006	
000007	
000008	
000009	
00000A	

*} word addressable*

So, the crucial aspect of this model is the random access memory. So, this is a word addressable memory, we assume that we have an array of words, each word we do not put any limit on the word size. But, nevertheless, we shall assume that in problems of size  $n$ , a word can contain order of  $\log n$  bits. In other words, an integer of the form  $n$  power  $k$ , where  $k$  is a constant, an integer of the form  $n$  power  $k$ , where  $k$  is the constant can be expressed using  $k \log n$  bits.

So, we shall assume that, a word in random access memory is capable of holding such an integer. Then we assume that what we have is a word addressable memory. What I mean by this is that each word in the memory has a unique address. So, as I have shown here, the addresses starting from start from 0 and then go on. So, every word has a unique address. The word can be accessed by putting out the address, a processor can put out the address and the content of the location would be accessible to the processor. We assume that the memory is potentially infinite. That is we do not put any upper bound on the number of locations available. What it means is that, we will never refuse to execute an algorithm for want of memory.

(Refer Slide Time: 19:32)



### Random Access Machine

- Each basic instruction takes 1 unit of time
- LOAD, STORE, ADD, SUB, MULT, DIV, JMP, CMP etc. are basic instructions
- Loops and subroutines are composed of basic instructions
- Each memory access takes one unit of time (!)
- Memory is potentially infinite (!)
- The running time of an algorithm is the number of instructions executed by it. The space used by an algorithm is number of the memory locations it reads or writes

So, we assume that the memory can always be extended, if there is a need. A random access machine has these properties. Each basic instruction takes 1 unit of time, the machine has an instruction set including load, store, add, subtract, multiply, divide, jump which is a branch statement, compare etcetera. So, that is the basic instruction set. Loops

and subroutines are composed of these basic instructions that is more complicated instructions will have to be made up of these basic instructions. We assume that each memory access takes 1 unit of time and that the memory is potentially infinite.

The running time of an algorithm is the number of instructions executed by it. The space used by an algorithm is the number of memory locations, it reads or writes. Now, even in the case of random access machines, there can be a critic that, these two assumptions, that the memory access takes 1 unit of time and that the memory is potentially infinite, are not realistic and in fact, they are not.

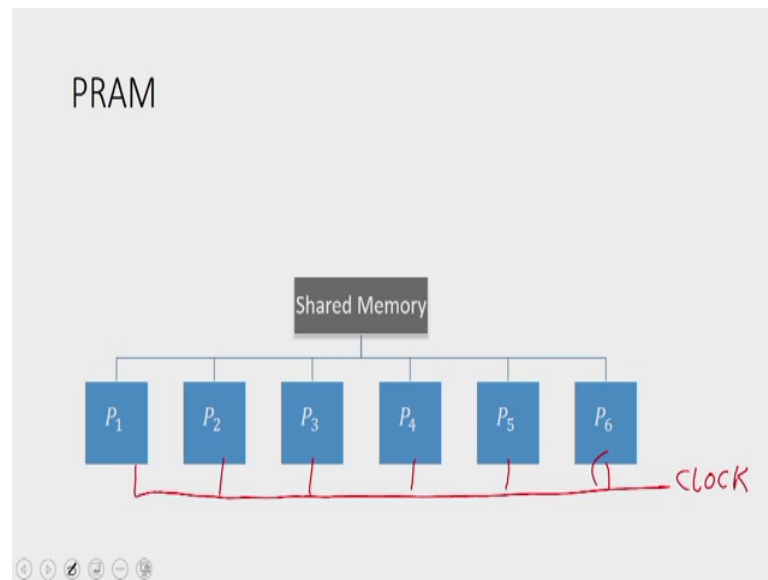
When you have a large number of memory locations, the memory access time certainly cannot be the same for every single location. Even, if you arrange the locations in the most compact spherical form, locations which are at the center of the sphere, are farther from the processor, than the locations which are on the outer surface. Therefore, accessing a memory location which is deeper inside will take longer.

But, what we assume is that the difference between accessing two different memory locations is minuscule. Therefore, we can for all practical purposes assume that every memory location is accessible in constant amount of time. Similarly, we also do not want to put any upper limit on the size of the memory. On any real machine there will be an upper bound on the size of the memory and the problems that are solvable will again have a limit.

But, we do not want to refuse to solve a problem for want of memory. Therefore, we assume that, the memory is potentially infinite because, if we run out of memory we can always extend the memory. Therefore, for the mathematical model that we are going to handle, we will assume that the memory is infinite. So, these are in a way unrealistic assumptions, which are necessary to keep the model, simple and accessible. Every model of computation in this manner, sweeps certain details under the carpet.

So, that is exactly what we have done with a parallel random access machine to we do not put any limit on the number of processors, the machine can have any number of processors that we want. But, then number of processors is assumed to be finite.

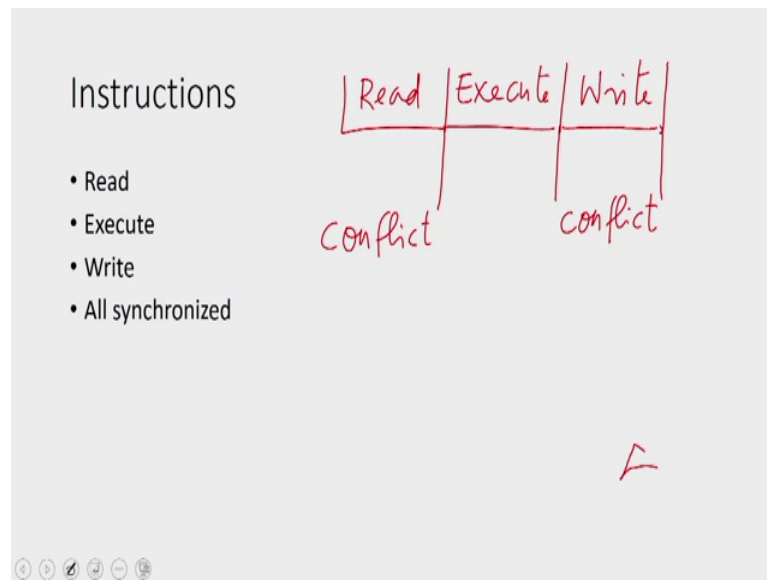
(Refer Slide Time: 22:15)



So, this is a diagram of a parallel random access machine, we have in this diagram 6 random access machines, all of them are connected to a shared memory, a shared memory that like the one we saw in the previous picture, with the potentially infinite locations. All the processors are connected to the shared memory using a common bus.

So, the processors can put out addresses, of the locations that they want to access from the shared memory location and the contents of the memory location will be accessible to them for read or write purposes and we assume that all the processors are fed the same clock, which is not shown in this diagram. We assume that the machine is a synchronous machine. So, all of them are fed the same clock, which means the processors are executing in lockstep.

(Refer Slide Time: 23:13).



We assume that every processor has, instructions made up of 3 phases. An instruction has 3 phases, the first phase of the read phase, the next phase of the execute phase and then we have a write phase. We assume that there is a synchronization at the end of each of these phases.

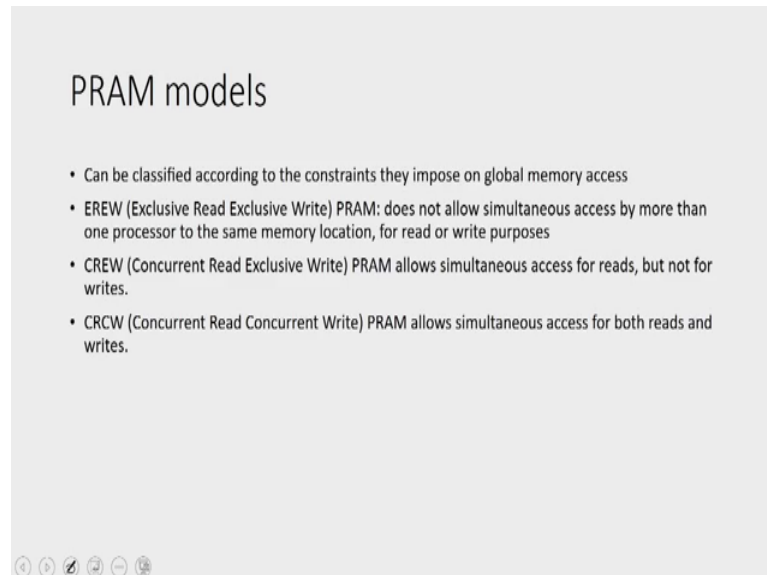
So, all the processors would read simultaneously and they would finish reading simultaneously and they would execute together and finish executing simultaneously and then write simultaneously. So, it is not possible for the read of one processor to overlap with the write of another processor which means, in the read cycle of an instruction. All the processors are attempting to read the memory locations. So, all the addresses that they are putting out would be for reading the memory locations.

Similarly, in the write phase, all the addresses that the processors put out are for writing into the memory locations. Therefore, it is possible for the processors to conflict on read during the read cycle, when multiple processors attempt to read from the same memory location, we have what is called a read conflict. Similarly, in the write phase it is possible for us to have a write conflict, when multiple processors attempt to write into the same memory location, we have what is called the write conflict.

So, we can have read conflicts and write conflicts in this model but, we assume we do not have read write conflicts that is because, the read cycle the read cycle and the write cycle of an instruction are separate, all the processors are synchronized, when one

processor reads every other processor reads. Therefore, it is not possible for us to have the read write conflicts in this model.

(Refer Slide Time: 25:18)



The parallel random access machines can be classified according to the constraints they impose on the global memory access. There are 3 kinds of models, the first one is the exclusive read exclusive write model, this is the most a constraint of the 3 models, it does not allow simultaneous access by more than one processor to the same memory location for either read or for write. CREW PRAM which is more powerful and therefore, more lenient, is a model which allows simultaneous access for reads, what it means is that read conflicts are allowed.

But, simultaneous writes are not allowed. In other words, multiple processors can read from the same memory location but, multiple processors cannot write into the same memory location and finally, we have the concurrent read concurrent write models, in which read conflicts as well as write conflicts are allowed. Multiple processors can access the same memory location in the read cycle. Similarly, multiple processors can access the same memory location in the write cycle as well.

So, let us look at these models in detail and EREW PRAM that is an exclusive read exclusive write PRAM does not allow simultaneous access of the memory locations for either read or write purposes.



(Refer Slide Time: 26:32)

EREW PRAM									
A step of an EREW PRAM									
	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>	P <sub>5</sub>	P <sub>6</sub>	P <sub>7</sub>	P <sub>8</sub>	P <sub>9</sub>
Read	4	8	14	35	20	92	11	7	6
Write	9	54	3	56	27	64	18	96	92

When you consider a step of an EREW PRAM, it will have a read phase and the write phase. In the read phase, the processors put out addresses that they want to access. Let us say we have a machine with, the 9 processors, let me call them P 1 through P 9. Let us say in one particular instruction, these processors want to read these memory locations.

So, as you can see, no two processors try to read from the same memory location. So, what is taking place is an exclusive read, there is no read conflict anywhere. Therefore, all these reads can be scheduled simultaneously and they will go through on the EREW PRAM. So, this is what is happening in the read cycle let us say, when we come to the write cycle, after executing. Let us say these same processors want to write in to locations. Again you can see that, no two processors are attempting to write in the same memory location. Therefore, this is a valid step, in this step no two processors attempt to read from the same memory location in the read cycle and no two processors attempt to write in the same memory location during the write cycle.

Therefore, the read cycle as well as the write cycle, are exclusively exclusive write. Therefore, this is a valid step of an EREW PRAM. Instead, if it were that, process of 3 wanted to access location 4 for read purpose in the read cycle, then there is a read conflict. Processor the P 1 is attempting to access location 4, P 3 is also attempting to access a location 4, then we would have a read conflict.

Therefore, this will not be a valid step of EREW PRAM. What we assume is that, if we schedule such a step on an EREW PRAM the program would crash. Therefore, it is the responsibility of the algorithm designer to make sure that, in every single step of his algorithm no two processors will have the read conflicts or write conflicts. Similarly, here if processor P 8 we have write in to the memory location 9, we would have a write conflict.

Processor P 1 is writing in location 9, processor of P 8 is also writing in location 9, and they have a conflict. So, there would be a write conflict. So, on EREW PRAM, the read conflicts and write conflicts are not allowed.

(Refer Slide Time: 29:47)

$P_1, P_3, P_6 \rightarrow 4$

CREW PRAM

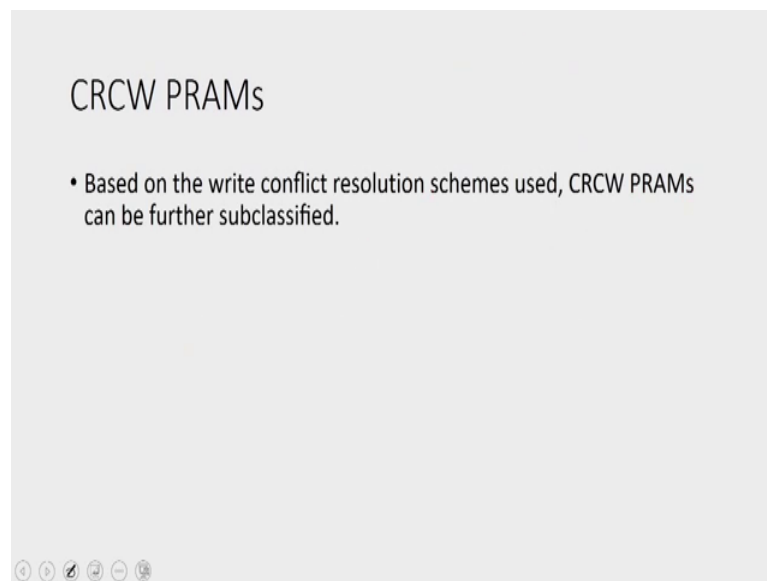
	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$	$P_6$	$P_7$	$P_8$	$P_9$
Read:	4	8	4	35	8	4	11	35	8
Write:	_____ exclusive _____								

Coming to a CREW PRAM, this is more lenient on this model. For example, for the same 9 processors, during a reach step we would have, accesses of the sort, this is the read phase. So, as you can see here there are read conflicts.

But, on this model we assume that the read conflicts are legal, that is, if we schedule multiple processors to access the same memory location for the read purpose, the program will not crash and the reads will in fact happen. So, here we see that, P 1, P 3 and P 6 are accessing location 4. So, on location 4 there is a read conflict of three processors.

But, this is legal all three processors will be able to read the contents of location 4. But, instead in the write cycle we need, exclusive writes that is every processor should be writing to a different memory location. If as an algorithm designer you would design a step that has a write conflict your program will crash. So, it is your responsibility to make sure that, while designing algorithms for CREW PRAMs to ensure that, there are no write conflicts.

(Refer Slide Time: 31:40)

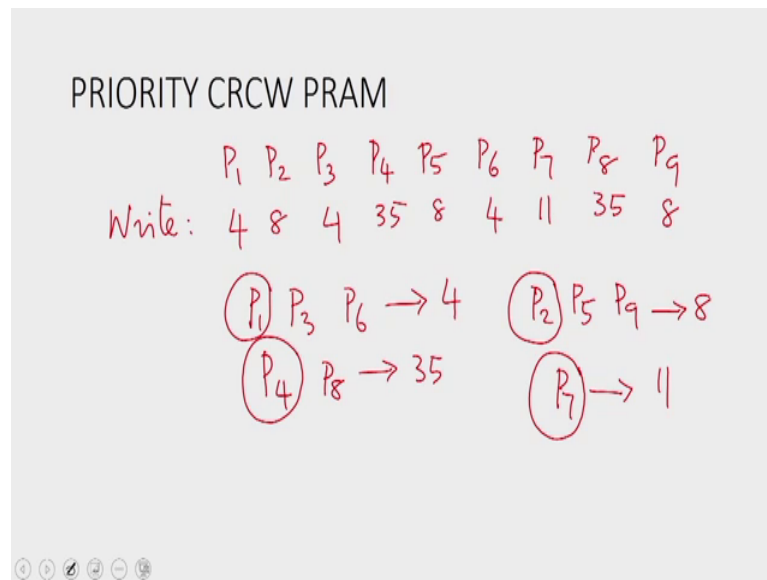


Now, coming to CRCW PRAMs, based on the write conflict resolution scheme that we use, CRCW PRAMs can be further sub classified. Now, here we know that different processors can attempt to write into the same memory locations simultaneously, these are what are called write conflicts. But, when a write conflict happens, multiple processors would be attempting to write different values. For example, if P 1 and P 7 are attempting to light write in location 4, then they are in a write conflict.

But, suppose P 1 is attempting to write 10 and P 7 is attempting to write 70, then there is a conflict and we do not know, which one to let through or whether to let through any one at all. The policy which will decide, who will win if at all, is called the conflict resolution scheme. So, depending on the conflict resolution scheme that we adopt, the models can be further sub classified.

So, the first of the models, that we consider is the priority CRCW PRAM.

(Refer Slide Time: 32:45).

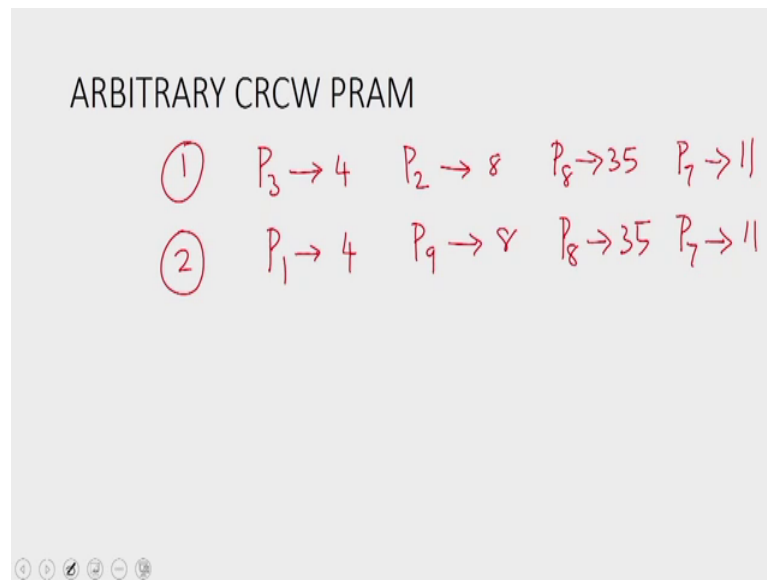


Let us say on a priority CRCW PRAM, with 9 processors, we have a write conflict of this form. In the write phase, the processors  $P_1$  through  $P_9$  are attempting to write into these locations. So, we find that,  $P_1$   $P_3$  and  $P_6$ , are accessing location 4,  $P_4$  and  $P_8$ , are accessing location 35,  $P_2$   $P_5$  and  $P_9$ , are accessing location 8,  $P_7$  alone is accessing location 11.

So, there are write conflicts and we have to resolve the write conflicts. On the priorities here CRCW PRAM, we resolve the write conflicts using the priority of the processors. We assume that a processor has a higher priority, the lower its index, which means,  $P_1$  has a greater priority than  $P_2$  and  $P_2$  has a greater priority than  $P_3$  and so on. So, here we have  $P_1$   $P_3$  and  $P_6$ , accessing location 4 simultaneously. Since  $P_1$  is the highest priority of processor of these 3, we assume that  $P_1$  wins.

So, out of the first set,  $P_1$  wins.  $P_4$  and  $P_8$  are conflicting on location 35;  $P_4$  has a greater priority than  $P_8$ . So, we assume that  $P_4$  wins, what that means, is that  $P_4$  will be successful in writing, the value that it wants in location 35. But, the write attempt by  $P_8$  will fail. Similarly in the third set,  $P_2$   $P_5$  and  $P_9$  are conflicting on 8, location 8,  $P_2$  will win and in the final group, there is only one member,  $P_7$  is attempting to write in location 11 alone and  $P_7$  will win, it does not have any competition. So, on the priority CRCW PRAM, the winner in any write conflict is the processor with the greatest priority.

(Refer Slide Time: 35:14).



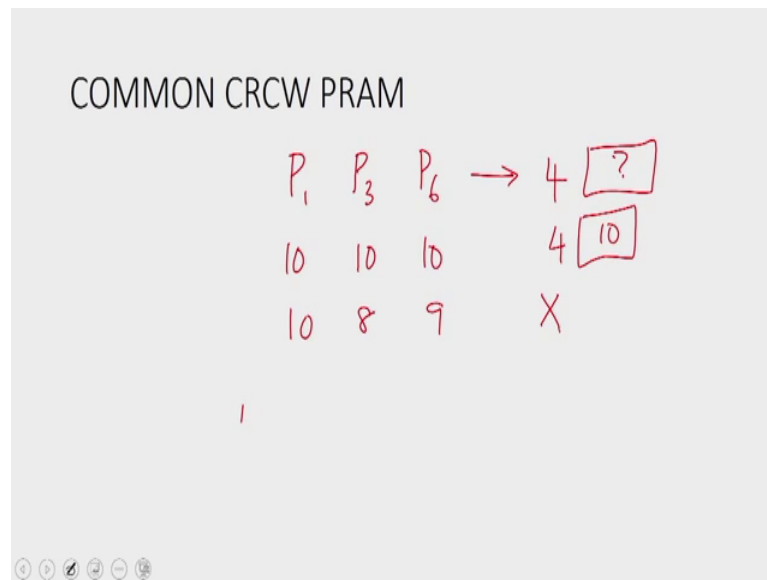
The second CRCW PRAM that we consider is the arbitrary CRCW PRAM.

So, here what we assume is that, out of any set of conflicting processors, an arbitrary one of them will win. So, if we consider the same example that we considered for the priority model, the same write step and we run this on the arbitrary model, it could be that, the processors win in this order, processor P 3 manages to write in location 4, P 2 manages to write in location 8, P 8 manages to write in location 35, P 7 manages to write in location 11. As you can see, P 1 P 3 and P 6 are conflicting on location 4 and now we say that, P 3 wins. The model allows for an arbitrary one of the processors to win.

So, when the step is executed P 3 wins out of the first group in the second group P 2 has 1, in the third group P 8 is 1 and in the fourth group there is only 1 member and that member anyway wins. So, this let us say is what happens, in the first execution of the step, when we execute the same step on the arbitrary CRCW PRAM again, it could be that out of the first group P 1 wins, in the second group P 9 wins, in the third group P 8 wins and in the fourth group of course, there is only 1 member, that member has to win.

So, what I am trying to say is that, if the same step is executed on the arbitrary CRCW PRAM again and again, different processors would win, at different memory locations. All that we know is that, an arbitrary one of the conflicting processors will win but, as an algorithm designer we are not allowed to assume anything about the identity of the winner, all we know is that one of the set will win.

(Refer Slide Time: 37:13).



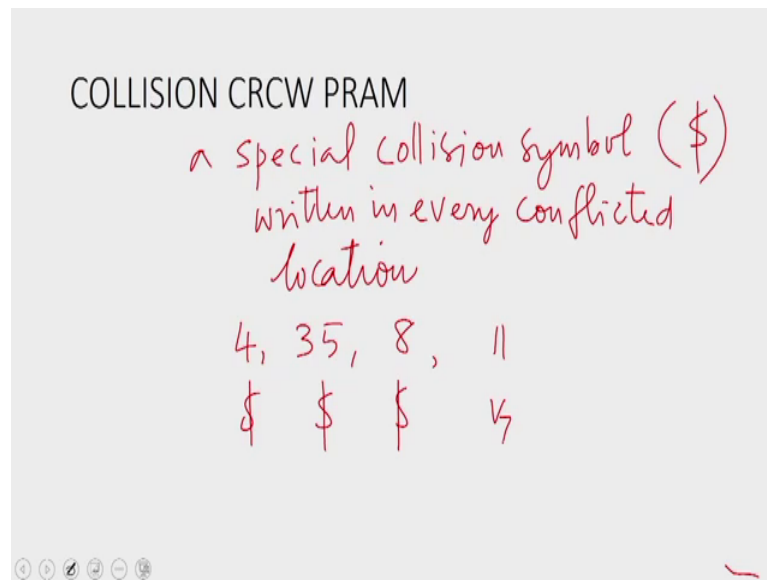
The third model we shall consider, is the common CRCW PRAM. In a common CRCW PRAM, write conflicts are allowed.

But, then a write conflict is legal only if, all the conflicting processors are attempting to write the same value. For example, we have for the same write sequence,  $P_1$ ,  $P_3$  and  $P_6$ , conflicting on location 4. For example, if all 3 wanted to write 10, then the write conflict would be considered legal on the common CRCW PRAM and the write would go through.

The content of the location 4, will now be changed to 10, instead of the previous value but, instead if they attempted to write different values, then the write would fail, the program would fail, on the common CRCW PRAM we assume that, a write conflict is valid, precisely when all the conflicting processors are attempting to write the same value. But, if they write different values, then we have an invalid instruction, which is akin to a division by zero in conventional algorithms.

So, we assume that on a common CRCW PRAM, the algorithm designer would have ensured that, all the conflicting processors would be writing the same value for every single write conflict.

(Refer Slide Time: 38:45)



And the fourth model that we are consider, that we consider is the collision CRCW PRAM. On a collision CRCW PRAM we want, a special collision symbol to be written in every location, that encounters a conflict, it is conventional to use a dollar symbol, for the collision symbol.

So, let us assume that, the special collision symbol is written in, every conflicting location. Therefore, in our example, locations 4, 35 and 8, will end up getting the collision symbol. Whereas, location 11, has only one processor writings into it therefore, what gets into location 11 will be the value that is written by the seventh processor, let me denoted by  $V_7$ , let 7 be the value that the seventh processor wants to write, location 11 will contain this value.

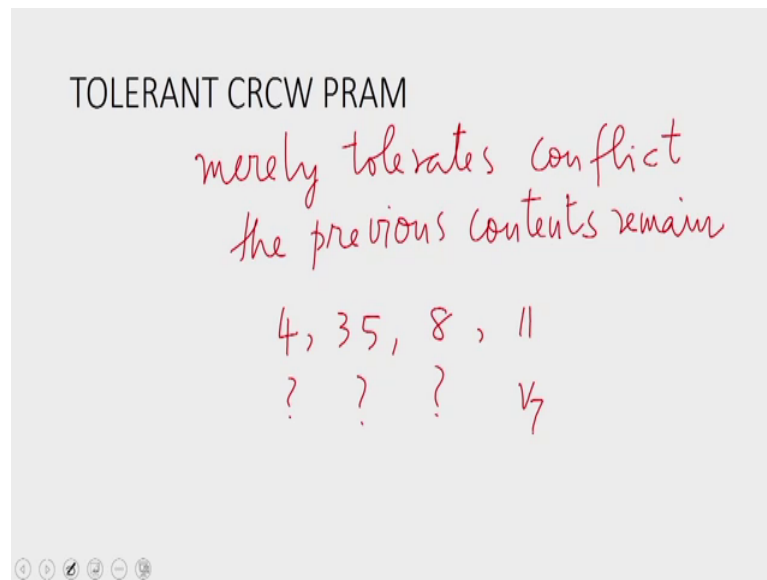
But, all the 3 other locations, 4, 35 and 8 will contain the collision symbol. So, this will be the result of the same step on a collision CRCW PRAM.

(Refer Slide Time: 40:12).

TOLERANT CRCW PRAM

*merely tolerates conflict  
the previous contents remain*

*4, 35, 8, 11*  
*? ? ? V<sub>7</sub>*

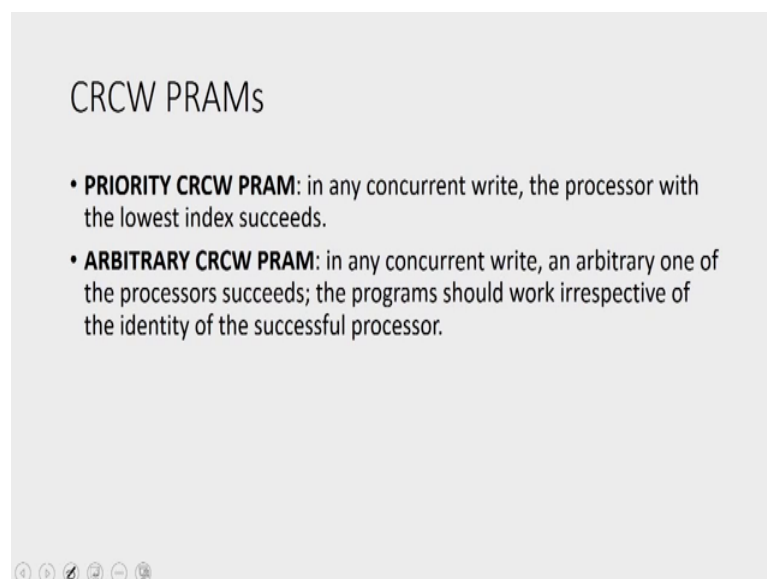


And our final, model of final CRCW PRAM is the tolerance CRCW PRAM, this is a model which merely tolerates, write conflicts. In the sense that, when a write conflict happens, the previous contents remain, in every conflicted location the previous content would remain. What that means is that? In locations, 4, 35 and 8, the previous contents would remain as it is, whereas, in location 11, V 7 would be written, V 7 is the value that process 7 wants to write in location 11.

(Refer Slide Time: 41:12)

CRCW PRAMs

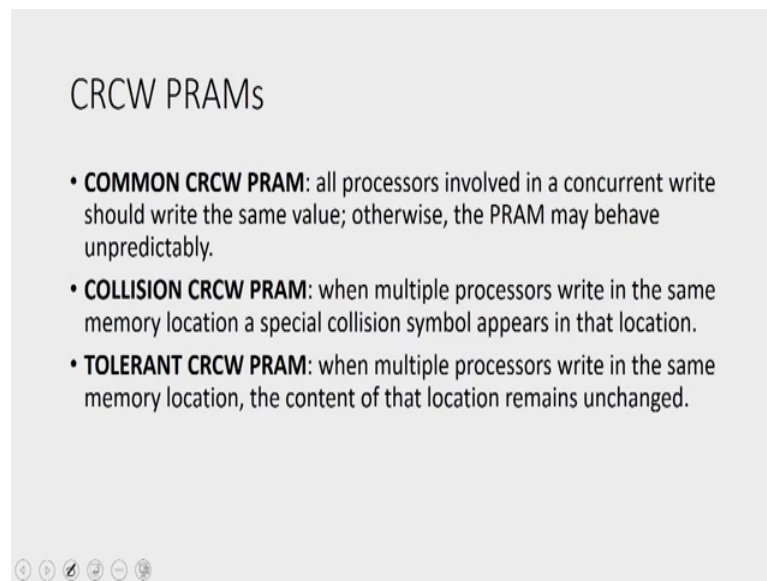
- **PRIORITY CRCW PRAM:** in any concurrent write, the processor with the lowest index succeeds.
- **ARBITRARY CRCW PRAM:** in any concurrent write, an arbitrary one of the processors succeeds; the programs should work irrespective of the identity of the successful processor.





So, to summarize, these are the CRCW PRAM models that we have seen. A priority CRCW PRAM is where, in any concurrent write the processor with the lowest index succeeds. An arbitrary CRCW PRAM is where, in any concurrent write an arbitrary one of the processors succeeds; the program should work irrespective of the identity of the successful processor.

(Refer Slide Time: 41:34)



The slide is titled "CRCW PRAMs" and lists three models of Concurrent Read, Concurrent Write Parallel Random Access Memory (CRCW PRAM). Each model is preceded by a bullet point. The slide has a light gray background and a small navigation bar at the bottom left.

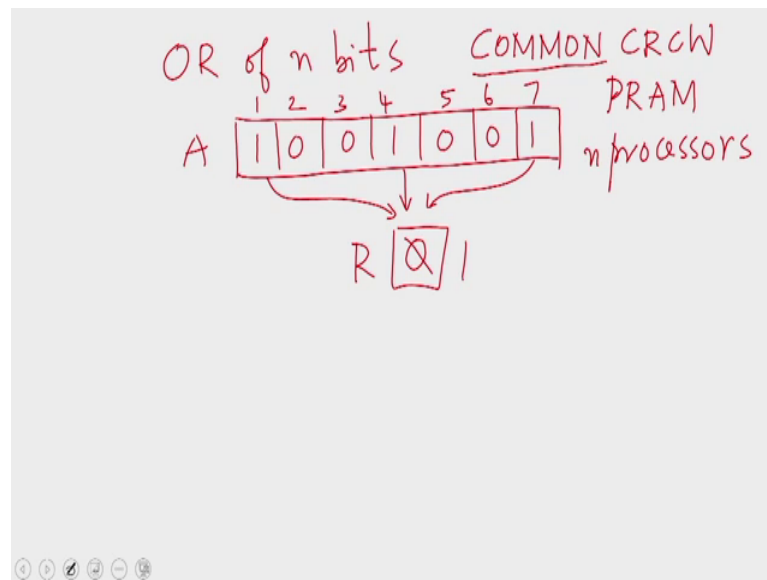
### CRCW PRAMs

- **COMMON CRCW PRAM:** all processors involved in a concurrent write should write the same value; otherwise, the PRAM may behave unpredictably.
- **COLLISION CRCW PRAM:** when multiple processors write in the same memory location a special collision symbol appears in that location.
- **TOLERANT CRCW PRAM:** when multiple processors write in the same memory location, the content of that location remains unchanged.

On a common CRCW PRAM, all the processors involved in a concurrent write, should be writing the same value. Otherwise the PRAM may behave unpredictably or we will assume that the algorithm is invalid.

In a collision CRCW PRAM, when multiple processors write in the same memory location, a special collision symbol appears in that location. Finally, on a tolerant CRCW PRAM, when multiple processors write in the same memory location, the content of that location remains unchanged which means, this model merely tolerates, a concurrent write.

(Refer Slide Time: 42:13)



Now, let us see an example, of a parallel algorithm. Let us say we want to find the OR of,  $n$  bits, given in an array  $a$ . So, let us say we have an array  $a$ , the locations of the array  $a$ , contained bits ones and zeros and suppose, we want to find the OR of these bits. Let us assume that OR model is, common CRCW PRAM, how do we find the OR of these bits. If we have  $n$  processors that is we assume that we have one processor for every single location of the array.

So, you can imagine that, you have one processor sitting on every single location of the array. Then let us take a location  $R$ , in which the final result should go. First let us assume that, all the processors will simultaneously write a value of 0 into  $R$ , this is an initialization. The location  $R$  initially contained some garbage, that garbage is rewritten with 0 by all the processors, simultaneously. All the processors executing simultaneously will change the value to 0, this is a write conflict, all the processors are conflicting on the same memory location  $R$ . Let us say the variable  $R$  corresponds to some memory location in the random access memory.

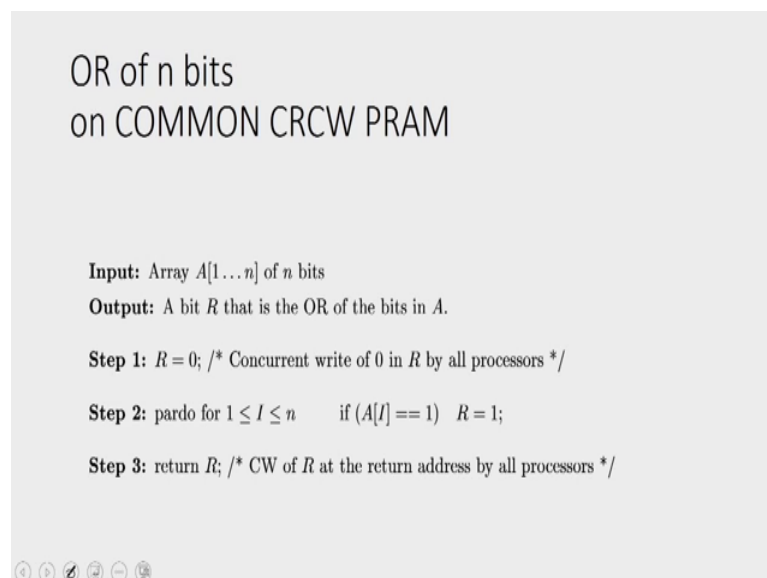
But, then since all of them are writing the value 0, this is a valid write conflict and the value returned by them will succeed, since the model is the common CRCW PRAM. So, the content of the location  $R$  is now 0. Now, let us say, every processor examines its own bit. So, processors 1 4 and 7, find that, they are sitting on a 1 bit each and they all will attempt to change the content of  $R$  to one. So, processors 1 4 and 7 will attempt to

change the content of  $R$  to 1. Again we have a write conflict, we have multiple processors accessing the same memory location and trying to change its content to the same value and the write will go through because, the model is the common CRCW PRAM.

So, if at all there are processors here; that are sitting on once, they will all attempt to change the content of  $R$  to 1 and all of them will succeed. So, there are 3 cases, there is no such processor, there is exactly 1 such processor and there are multiple such processors. If there is no such processor, nobody comes forward to change the location  $R$ . Therefore, after this step, when all the processors examine this location, they find that a zero is still in that location therefore, they realize that the result of  $R$  is 0.

But, if there is exactly 1 processor, that processor will come forward and change the value from 0 to 1, which is the result of the  $R$ . Finally, if there are multiple processors, as is shown here, there are 3 processors here sitting on once, all 3 will come forward and attempt to change the location up to 1 and since all of them are attempting to write the same value, the value will change to 1. Therefore, after this when all the processors come and examine the location, all of them will find a 1. Therefore, 1 is the answer, that all of them will realize.

(Refer Slide Time: 46:23)



OR of  $n$  bits  
on COMMON 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:**  $R = 0$ ; /\* Concurrent write of 0 in  $R$  by all processors \*/

**Step 2:** pardo for  $1 \leq I \leq n$  if  $(A[I] == 1)$   $R = 1$ ;

**Step 3:** return  $R$ ; /\* CW of  $R$  at the return address by all processors \*/

So, formally this algorithm can be specified in this fashion, to find the OR of  $n$  bits on a common CRCW PRAM, when we are given an array  $A$  1 to  $n$  of  $n$  bits, we proceed as

follows, we take a variable  $R$  and initialize it to 0. This is an initialization done concurrently by all the processors. So, every processor attempts to write 0 in variable  $R$ . So, here I have specified  $R$  equal to 0 without specifying which processor has to do it, if I have not specified which processor has to do something, what I assume is that all the processors have to do that simultaneously.

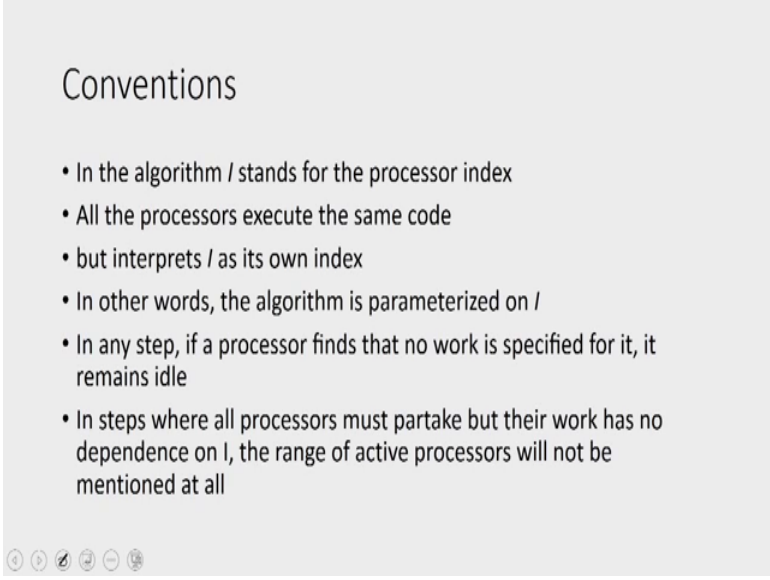
So,  $R$  is assigned 0 by all the processors executing simultaneously, then in the second step, we have a parallel execution, every processor with its index ranging from 1 to  $n$ , will do this, the processor will check its own bit in array  $a$ , if the bit turns out to be 1, it will change the location  $R$  to 1. So, this is specified using the following syntax, we have what is called a pardo construct, we say pardo for one less than or equal to  $I$  less than or equal to  $n$ . The meaning of that is, that every processor, whose indexes  $I$ , where  $I$  ranges from 1 to  $n$ , should execute the following in parallel.

So, what the processor has to do is this, that is the processor with index  $I$  has to do this, it checks whether  $A[I]$  is 1, if  $A[I]$  happens to be 1, then it has to change the content of  $R$  to 1. So, we assume that, every single step has an appropriate synchronization. So, the condition check happens simultaneously, which means every single processor will check its own bit simultaneously, after that all of them will attempt to update  $R$  simultaneously.

So, if there is at all one processor that is attempting to write, the location  $R$  will now contain 1 and finally, in the third step, we say return  $R$ , this is once again a concurrent write of  $R$ , that is the content of  $R$  is returned to the return address, by all the processors simultaneously, which is again a concurrent write.

So, since I have not specified which processor has to do this write, we assume all the processors have to do this simultaneously.

(Refer Slide Time: 48:36)



### Conventions

- In the algorithm  $I$  stands for the processor index
- All the processors execute the same code
- but interprets  $I$  as its own index
- In other words, the algorithm is parameterized on  $I$
- In any step, if a processor finds that no work is specified for it, it remains idle
- In steps where all processors must partake but their work has no dependence on  $I$ , the range of active processors will not be mentioned at all

In this algorithm specification, we have adopted some conventions. In the algorithm  $I$  stands for the processor index, we assume that every processor executes the same code, that is this code is available to every single processor. Only that this code has to be parameterized by the processors index. Every processor interprets  $I$  as its own index in other words, the algorithm is parameterized on  $I$  in any step, if a processor finds that no work is specified for it remains idle and in steps where all processors must partake.

But, their work has no dependence on  $I$ , the range of the active processors will not be mentioned at all. So, that is what I was mentioning here, in the first step as well as the third step, we do not specify which processors have to execute these steps ; that means, all the processors have to execute these steps. So, these are some of the conventions we shall use in algorithm specifications. So, that is it from the first lecture, hope to see you in the next lecture.

Thank you.