

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

Lecture – 04
Cost and Optimality

Welcome to the fourth lecture of the NPTEL MOOC on Parallel Algorithms. In the previous lectures we have seen several models of parallel computation in particular we have seen the parallel random access machine and several and several varieties of it. In particular we have seen the Exclusive Read Exclusive Write P RAM, the Concurrent Read Concurrent Write P RAM, the Concurrent Write and Exclusive Write P RAM. And we have also seen the several varieties of the CRCW P RAM namely priority, arbitrary, common, coalition and tolerant and we have also seen how some of these could be simulated on some of the others.

Today we shall see an interesting property of some of these P RAMs the property is called Self Simulation.

(Refer Slide Time: 01:20)

Self Simulation $3^2 \lceil \frac{5}{2} \rceil$

PRAM of N processors $O(\frac{N}{n})$
 PRAM of n processors steps

<u>EREW PRAM</u>	$P_1 \ P_2 \ P_3 \ P_4 \ P_5$ $M_1 \ M_2 \ M_3 \ M_4 \ M_5$ $M'_1 \ M'_2 \ M'_3 \ M'_4 \ M'_5$	$Q_1 \ Q_2$ 1. $Q_1 Q_2 \rightarrow P_1 P_2$ 2. $Q_1 Q_2 \rightarrow P_3 P_4$ 3. $Q_1 \rightarrow P_5$
------------------	--	---

A P RAM model is said to be self simulating if a larger P RAM can be simulated on a smaller P RAM of the same kind for a proportionate slowdown. So, what I mean is this let us say we have a P RAM of capital N processors and we have another P RAM of

small n processors and let us see we want to simulate the one step of the larger P RAM on the smaller P RAM.

If the simulation can be affected in order of capital N by small n steps, then we say the P RAM is self simulate able let us show that several of the models familiar to us or self simulatable. First let us consider the EREW P RAM on the EREW P RAM concurrent read or concurrent write are not allowed that is in any particular clock cycle the processors would be reading some memory locations and writing into some memory locations, no 2 processors should be reading from the same memory location and no 2 processors should be writing into the same memory location.

Let us say we have 5 processors, let us say these processes want to read from locations M_1, M_2, M_3, M_4 and M_5 respectively and let us say they want to write into locations M_1 prime, M_2 prime, M_3 prime etcetera. So, the processors in one clock cycle would perform all these reads and all these writes a clock cycle as we have seen in the previous classes is divided into read cycle and execute cycle and the write cycle.

So, in the read cycle all the reads will happen simultaneously, then the processors will execute and then in the write cycle all the writes will happen simultaneously since the reads and writes are all exclusive there will be no conflict. So, this is the step that we want to simulate. So, we have a 5 processor P RAM a step of which is shown here, this we want to simulate on a 2 processor P RAM let us say we have 2 processes Q_1 and Q_2 on a 2 processor P RAM. On a 2 processor P RAM we want to simulate this one step.

So, the simulation proceeds in this fashion, in the first step the processes Q_1 and Q_2 pretend to be processes P_1 and P_2 respectively. So, processes P_1 and P_2 read from locations M_1 and M_2 and write into locations M_1 prime and M_2 prime. So, Q_1 and Q_2 pretending to be P_1 and P_2 will read from locations M_1 and M_2 and write into locations M_1 prime and M_2 prime.

Since these reads and writes are exclusive from the reads and writes of the other processors there will be no consistency issues when these 2 steps are executed in isolation. So, after Q_1 and Q_2 have finished pretending to be P_1 and P_2 and have finished the finished executing this particular step. In the second step of the simulation the same pair of processors Q_1 and Q_2 will pretend to be processes P_3 and P_4 , pretending to be P_3 and P_4 they will read from locations M_3 and M_4 and write into

locations M_3 prime and M_4 prime. That is in the read cycle of the simulations step, they will read from locations M_3 and M_4 and in the write cycle of the simulation step they will write in locations M_3 prime and M_4 prime.

Then finally, in the third step of the simulation process a Q_1 alone will participate this will pretend to be process of P_5 . Process a Q_1 pretending to be process of P_5 will read from location M_5 and write into location M_5 prime with those all the steps of the 5 process a machine are simulated all the reads and writes have been simulated and the time taken is 3 steps and 3 happens to be the ceiling of 5 by 2.

So, this is the principle of the simulation, when we have capital N processors and when we have small n processors with which we have to simulate these processes what we do is this. The capital N processes will be divided into several groups each group is of size small n and then each group will be brought alive in turn starting from the left end in this case.

So, first P_1 and P_2 are brought alive, then P_3 and P_4 are brought alive and finally, P_5 was brought alive in each case we will be using only the actual processes Q_1 and Q_2 . So, the simulation runs in 3 steps. So, we can easily generalize this to a simulation of a capital N sized P RAM on a small n sized P RAM and the simulation will run in order of capital N by n steps. So, we say that EREW P RAM is a self simulatable model exactly the same principle will work for the CREW P RAM as well. Therefore both the concurrent read exclusive write and the exclusive read exclusive write P RAMs are self simulatable.

Now we come to the CRCW P RAM models first let us consider the strongest of them namely priority.

(Refer Slide Time: 07:57)

10 processors

4	②	5	7	9	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆	P ₇	P ₈	P ₉	P ₁₀
7	①	10			7	4	13	13	4	20	4	13	4	7
13	③	4	8		V ₁	V ₂	V ₃	V ₄	V ₅	V ₆	V ₇	V ₈	V ₉	V ₁₀
20	⑥													

PRIORITY : 1. Q₁ Q₂ Q₃ Q₄ → P₇ P₈ P₉ P₁₀
 write to 4 13 4 7 P₇ P₈ P₁₀ ✓

4 7 13 20 2. Q₁ Q₂ Q₃ Q₄ → P₃ - P₆
~~V₁~~ ~~V₂~~ ~~V₃~~ ~~V₄~~ ~~V₅~~ ~~V₆~~ ~~V₇~~ ~~V₈~~ ~~V₉~~ ~~V₁₀~~
 V₁ V₂ V₃ V₄ V₅ V₆ V₇ V₈ V₉ V₁₀

First of all let us set up an example let us say we have 10 processors, P 1 wants to write in location 7, P 2 wants to write in location 4, P 3 wants to write in location 13, P 4 wants to write in location 13, P 5 wants to write in location 4, P 6 wants to write in location 20, P 7 in 4, P 8 and 13, P 9 in 4, P 10 in 7, let us say and let us say they want to write values V 1, V 2, V 3 etcetera.

So, for example, process of P 5 wants to write in location 4 and the value that it wants to write is V 5. So, as you can see there are conflicts in this case, we have 4 locations namely 4 7 13 and 20 and the processes that want to write in location 4 are 2 5 7 and 9 processes 1 and 10 to write in location 7, processes 3 4 and 8 want to write in location 13, processor 6 wants to write in location 20.

So, let us say this is one step of a 10 processor P RAM and we want to simulate this let us say on a 4 processor P RAM, first let us consider the strongest of the CRCW P RAM models which is priority. So, let us say we want to simulate this step of a 10 processor priority CRCW P RAM, on a 4 processor CRC priority CRCW P RAM. So, the simulation proceeds as follows, in the first step processors Q 1, Q 2, Q 3 and Q 4 pretend to be P 7, P 8, P 9 and P 10.

So, as you can see they have divided the virtual processors P 1 through P 10 into groups of size 4 each and we are beginning from the right end that is we are bringing the rightmost group alive first you will see why this is. So, now, Q 1, Q 2, Q 3, Q 4

pretending to be P 7, P 8, P 9 and P 10 will write to locations 4, 13, 4 and 7 respectively. So, you can see that processors P 7 and P P 9 conflict over location 4 since the model is priority the least index processor will succeed therefore; P 7 will succeed in writing in location 4.

There is no conflict that the other locations therefore, P 8 and P 10 also succeed in writing. So, the processes that will succeed in this step of the simulation are P 7, P 8 and P 10. The content of the memory location now will be location 4, 7 and 13 have changed and they now contained V 7, V 10 and V 8 respectively location 20 has not changed let us say it is content remain the same old value which we denote as O 20.

So, this is the content of the memory after the first step of the simulation, now in the second step of the simulation as you would expect we would bring the next group alive. So, here Q 1, Q 2, Q 3, Q 4 pretend to be P 3 through P 6 respectively and they will write to locations 13, 13, 4 and 20 respectively. So, as you can see again there is a conflict at location 13 processes P 3 and P 4 will attempt to write in location 13, there is no conflict at locations 4 and 20.

So, P 3 and P 4 conflict at location 13 and P 3 will win because 3 is smaller than 4 therefore, at location 13 the value changes to V 3, at location 4 the value changes to V 5, at location 20 the value changes to V 6. So, locations 4, 7, 13, 20 now contain V 5, V 10, V 3 and V 6.

(Refer Slide Time: 13:45)

3. $Q_1 Q_2 \rightarrow P_1 P_2$ write to 7 & 4 ✓

4	7	13	20
V_2	V_1	V_3 V_4	V_6

$3 = \lceil 10/4 \rceil$ steps

Non n in $O(N/n)$ time

ARBITRARY ✓

COMMON $V_2 = V_5 = V_7 = V_9$

Now, we come to the third step of the simulation, in the third step there are only 2 processors left. So, these can be simulated using 2 of the actual processors. So, Q 1 and Q 2 will simulate processors P 1 and P 2 they write to locations 7 and 4 respectively and both of them will succeed because there is no conflict.

So, the contents of the memory will now be locations 4 and 7 were overheard and in this step therefore, they finally, contain V 2, V 1, V 3 and V 6. So, this is the end of the simulation. So, at the end of the simulation the locations would contain V 2, V 1, V 3 and V 6 exactly how they should be if the 10 processors step was executed on a 10 processor priority CRCW P RAM the result is identical.

If these were executed on a 10 processor P RAM the succeeding processes would be 2, 1, 3 and 6 they would succeed in getting values V 2, V 1, V 3 and V 6 into locations 4, 7, 13 and 20 respectively and that indeed is the net result of the simulation. Therefore, the simulation is a success the simulation has taken 3 steps which as the ceiling of 10 by 4. So, extending this technique of simulation we can now simulate an n processor P RAM on an n small n processor P RAM in order of capital N by small n time which proves that the priorities CRCW P RAM model is also self simulatable.

Now, we come to the arbitrary model in the arbitrary model when a set of processes attempt to write in the same memory location that is when they conflict on a particular memory location. Then the algorithm designer can be sure that any one of them will decide, but the designer cannot be sure which one of them will succeed. So, how should the simulation be different from priority? In fact, we find that the same simulation technique will work on arbitrary as well.

If we run the same simulation on the arbitrary model that is if we have a 4 processor the arbitrary CRCW P RAM and the same simulation has run on it then the final result will be almost identical. The only thing is that, in the first step whenever there is a conflict in the first step there is a conflict between processes P 7 and P 9 and on the priority model P 7 wins, but in the case of the arbitrary model instead of P 7 P 9 might win too. So, all we can say is that one of the processes P 7 and P 9 might win therefore, the content here could well have been V 9, but then we find that this value in any case will be overwritten by V 5 in the next step.

So, irrespective of whether it was V 7 or V 9 the value will be overwritten by V 5 in the second step of the simulation, but in the second step of the simulation location 13 has a conflict. So, P 3 and P 4 are both (Refer Time: 17:35) to write in to location 13 and one of them will succeed, arbitrary can guarantee only that one of them will succeed, but then all we know here is that the value which gets in location 13 is either V 3 or V 4.

So, the contents of the locations would be like this now at the end of the second step of the simulation we will have V 5 and 4, V 10 in 7, V 3 or V 4 in 13 and V 6 in 20 and finally, when we override the values in locations 4 and 7 we will have the contents of the locations like this, there is an ambiguity about the contents of the location 13 it could be either V 3 or V 4, but this is perfectly fine because this is all that the arbitrary model assumes that is one of the conflicting set of processors should win and that indeed happens here.

So, we find that the simulation works fine on the arbitrary model as well. So, arbitrary model is self simulatable as well. Now we come to the common model on the common model what we know is that when a set of processes conflict they will all be writing exactly the same value. For example, in this case processors 2, 5, 7 and 9 conflict over location 4 therefore, that is if the given 10 process of step were up step of a common CRCW P RAM, then we assure that V 2 equals V 5 equals V 7 equals V 9 otherwise this would be an illegal step.

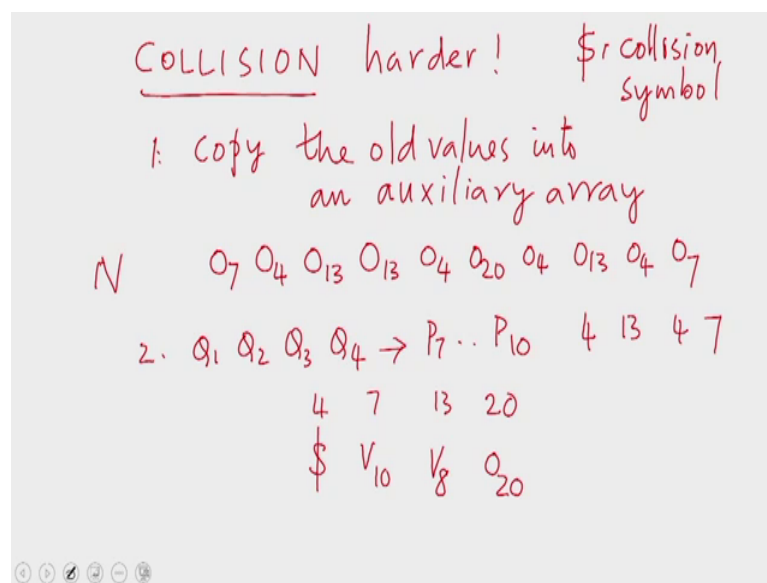
Since all these values are the same, it does not matter which processor succeeds, therefore, the same simulation will work for the common model as well whereas, in the first step of the simulation we attempt to simulate P 7, P 8, P 9 and P 10 and then P 3 through P 6 and then P 1 and P 2 exactly the same schedule of simulation will work for common as well.

Now, you might notice that for the case of priority it is crucial that we start from the rightmost group that is the group with the largest index. In this case P 7, P 8, P 9 and P 10 that is because on priority the processor with the least index is supposed to succeed. Therefore, we want the processor that is writing into a location to overwrite all the values written by processors of larger index for example, P 1 and P 10 here are attempting to write in location 7, in the first step of the simulation 10 gets a go. So, 10 will get to write value V 10 in location 7 and in step 3 when P 1 comes alive P 1 will override this value

with V 1. So, the wide final value that goes into the memory location is V 1 and not V 10 had we simulated the other way around that is had we started with the leftmost group then V 10 would have been overwritten on V 1 and the final value would have been V 10 that would not do.

Therefore for the priority model it is crucial that we start with the rightmost group, but for the arbitrary and common simulations, that is not crucial we might as well start with the leftmost end. So, now, we have shown that all these 3 models are self simulating.

(Refer Slide Time: 21:17)



Now, we come to the collision model on this model the proof is a little harder, this is because on the collision model when a set of processors conflict over a memory location, a collision symbol is supposed to appear in that location. So, as in our previous examples let us assume that dollar is the collision symbol which is a special integer.

So, here again we use the same example, here what we do is this, first we copy the old values into an auxiliary array what I mean is this processor one wants to write in location 7 suppose location 7 contains the an old value O 7. This old value is copied into one location processor O 2 wants to write in location 4, the old value of location 4 is copied into the adjacent location processor 3 wants to write in location 13 it is old value is copied in the third location, then in the fourth location we have O 13 again because processor of 4 wants to write in location 13 and so on.

So, in the first step of the simulation what we do is to copy these old values into an auxiliary array in the shared memory now this auxiliary array has a size of n because there are n virtual processors there is one old value for each of the virtual processors this auxiliary array has a size of n . So, what we now need is this in this step every processor should read a value and write it into a separate memory location. So, the writes are all exclusive the reads may not be exclusive because processor P 3 and P 4 are reading from the same location 13.

So, this is like a step of the CREW P RAM we have already seen that CREW P RAM is self simulating therefore, N steps of the CREW P RAM can be simulated on a small n sized P RAM in order of capital N by small n steps. So, this is an array of size 10 this can be simulated on a 4 processor machine in 3 steps. So, the first step of the simulation algorithm in fact, takes 3 clock cycles it is like the simulation of a CREW P RAM step. In the second step we let processors Q 1, Q 2, Q 3, Q 4 pretend to be P 7, P 8, P 9 and P 10 and write into locations 4, 13, 4 and 7.

So, the memory contents would now be like this location 4 has a conflict therefore, the collision symbol will appear in location 4, location 7 has no conflict therefore, V 10 will appear there, location 13 has no conflict therefore, V 8 will appear there, 20 has not been returned to therefore, 20 will contain the previous value with 20, this is how the location values would now be.

(Refer Slide Time: 25:20)

Self-Simulation

④ $Q_1, Q_2 \rightarrow P_1, P_2$ 7 4 4

4	7	13	20
\$	\$	\$	V ₆

- A PRAM model is said to be self-simulating, if for all $N \geq n \geq 1$, a PRAM of that model of size n can simulate a single step of another PRAM of the same model of size N in $O(N/n)$ time.
- All CRCW PRAMs we have seen except TOLERANT are self-simulating.
- TOLERANT is not known to be self-simulating.

⑤ $Q_1 \dots Q_4 \rightarrow P_3 \dots P_6$ 13 13 4 20

4	7	13	20
\$	V ₁₀	\$	V ₆

\$ found, back off

Then in the second step of the simulation we would have Q 1 through Q 4 pretending to be P 3 and P 6, P 3 through P 6 and writing into locations 13, 13, 4 and 20.

Now, here 13 has a conflict, but when location 4 already contains a collision symbol processor P 5 when it attempts to write in location P 4 finds that it already contains the collision symbol therefore, it will back off it finds that it already contains the collision symbol therefore, it backs off. Therefore, at the end of the step the memory will contain collisions symbol in locations 4 and 13, this is what happens in the third step of the simulation.

Then in the fourth step of the simulation processes Q 1 and Q 2 try to locations P 1 pretend to be P 1 and P 2 and write in locations 7 and 4, processor P 2 attempting to write in location 4 finds that it already contains a collision symbol therefore, as processor P 5 did in step 3 it would back off and the collision symbol will be left behind in location 4, what about processor 2, processor 1.

Processor 1 that attempts to write in location 7 finds that the value there is P 10 and it realizes that this value V 10 is different from the old value, the old value had been copied into the first memory location of the auxiliary array, the first location corresponds to the index of the first processor. So, the first processor will look into the first location of the auxiliary array and there it will get the old value of location 7, it finds that the old value has changed.

Why did the old value change that is because in some of the earlier step of the steps of the simulation some processor wrote in location 7. What it means is that, if the given step would have been executed on a 10 processor P RAM then there would have been a conflict at location 7 and the collision symbol would have appeared in location 7.

Therefore, the first processor now realizes that there is in fact, a conflict at location 7 and therefore, it should write the collision symbol there therefore, what the first processor does is to write the collision symbol in 7 not the value V 1. Therefore, at the end of the third step we find that the concerned memory locations contain the collision symbol in locations 4, 7 and 13 and location 20 will contain V 6, this is exactly the behavior of the collision model.

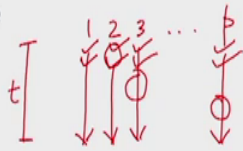
So, we have managed to self simulate collision as well the self simulation has taken order of N by small n steps for the copying of the old values into the auxiliary array, after that we have again order of N by small n steps of simulations therefore, the total cost of the simulation is order of N by n again. Therefore, to summarize we can say a P RAM model is self simulating if for every N greater than or equal to small n a P RAM of that model of size small n can simulate a single step of another P RAM of the same model of size capital N in order of capital N by small n time.

We have seen that all CRCW P RAMs we know except for tolerant are self simulating tolerant is not known to be self simulating because tolerant is not known to have this nice property in the rest of this course we will be seeing very little of tolerant, consider an algorithm that runs in T steps using P processors.

(Refer Slide Time: 29:50)

Cost

- The cost or work of an algorithm that runs in t time using p processors is the time-processor product pt
- Simulation on a one processor machine



So, here we have p processors and the p processors are all active for t steps each processor executes t instructions therefore, the total number of instructions executed by all the processors together is p into t this is what we call the cost of the algorithm. It could be that during some part of the algorithm some processors are idle, but nevertheless those processors are active and are dedicated for the execution of this algorithm.

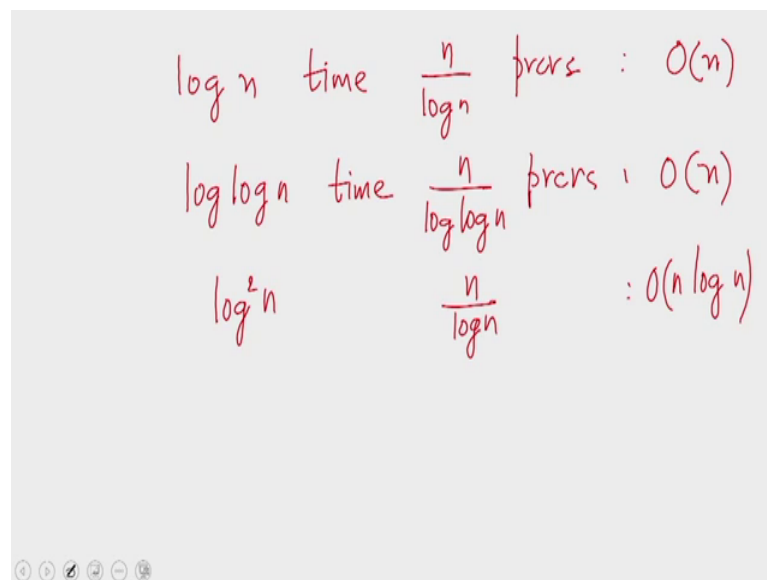
Therefore we have to count all the instructions the no operation instructions executed by these processors also towards the cost of the algorithm. So, if you keep p processors alive

for a time duration of t for the execution of the algorithm. The total cost of the execution is p times t this is what we call the cost of an algorithm, if this algorithm is simulated on a one processor P RAM which is identical to a random access machine or conventional computer, if we simulate this on a random access machine then that one machine can simulate the first step of every processor in the first phase of the simulation.

So, first all the first steps of the processes are executed one by one. So, the one actual processor will pretend to be each of the virtual processors one after another and then execute all the first steps. Once all the first steps are executed the processor can in turn execute the second steps.

So, proceeding like this the first step of the algorithm can be simulated in p steps the second step can be simulated in an additional p steps and so on. Therefore, to simulate the p steps of the algorithm it would take order of pt time on a sequential machine therefore, the cost of a parallel algorithm corresponds to the time taken by the sequential simulation of that algorithm, cost is a measure that we will be considering very often.

(Refer Slide Time: 32:18)



$$\begin{array}{lll} \log n \text{ time} & \frac{n}{\log n} \text{ prors} & : O(n) \\ \log \log n \text{ time} & \frac{n}{\log \log n} \text{ prors} & : O(n) \\ \log^2 n & \frac{n}{\log n} & : O(n \log n) \end{array}$$

Let us say we have an algorithm with a (Refer Time: 32: 19) $\log n$ steps using n by $\log n$ processors, then the cost of the algorithm is order of n the product of the time and the number of processes. If an algorithm runs in \log of $\log n$ time using n divided by \log of $\log n$ processors the cost of this algorithm is order n^2 .

If an algorithm runs in order of $\log^2 n$ time using $n/\log n$ processors the pt value in this case is order of $n \log n$ there is an algorithm that runs in $\log^2 n$ time using $n/\log n$ processors can be simulated on a single processor machine in order of $n \log n$ time. So, the cost of the sequential simulation is order of $n \log n$ that is also the cost of the algorithm.

(Refer Slide Time: 33:46)

Optimality

- If $\text{Seq}(n)$ is the worst-case running time of the fastest known sequential algorithm for a problem of size n , an optimal parallel algorithm for the same problem runs in $O(\text{Seq}(n)/p)$ time using p processors

$\text{cost} = \text{seq}(n)$

The slide features a light gray background. The title 'Optimality' is at the top left. Below it is a bullet point with text that includes underlined terms 'fastest known sequential algorithm' and 'optimal parallel algorithm'. A handwritten red formula 'cost = seq(n)' is written to the right of the bullet point. At the bottom left, there are small navigation icons.

So, the notion of cost leads us to the notion of an optimal algorithm for a problem p let us consider the fastest known sequential algorithm. Suppose this algorithm runs in seq of n time in the worst case then an optimal parallel algorithm for the same problem runs in order of seq of n by p time using p processors.

In other words if the cost of a parallel algorithm is equal to seq of n then we say that the parallel algorithm is optimal for example, consider the problem of merging.

(Refer Slide Time: 34:29)

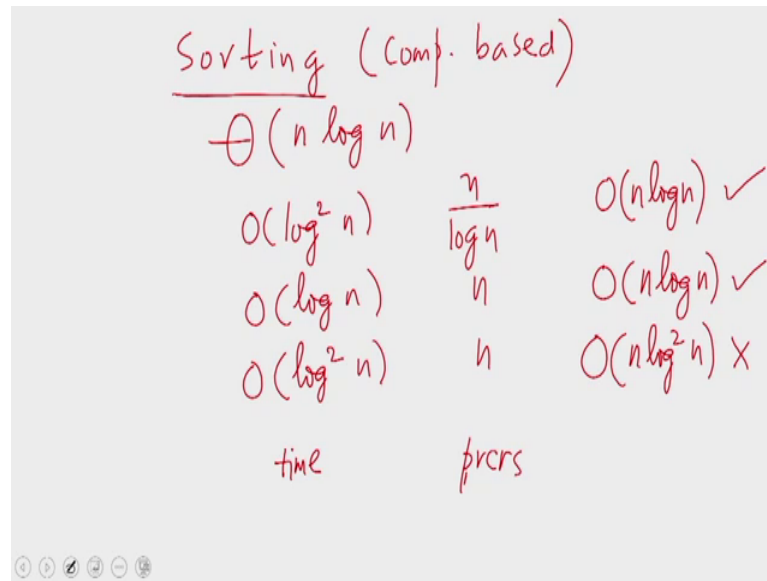
<u>Merging</u> $\Theta(n)$		<u>cost</u>
$O(\log n)$	$\frac{n}{\log n}$ prs	$O(n)$ ✓
$O(\log^2 n)$	$\frac{n}{\log^2 n}$ prs	$O(n)$ ✓
$O(\log n)$	n	$O(n \log n)$ X
time		

We know that 2 arrays of size n can be merged in theta of n time there exists a merging algorithm that runs in order of n time and the lower bound for merging is omega of n on an input of size n . So, the problem of merging 2 arrays of size n each has a complexity of theta of n .

Let us say we have an algorithm that runs in \log of n time using n by $\log n$ processors then the cost of this algorithm is order of n this is an optimal algorithm in this column, we write the time. So, we if we have an order $\log n$ time algorithm that runs in that runs using n by $\log n$ processors, then the cost of the algorithm is order of n that is an optimal algorithm.

If an algorithm runs in order of \log squared n time an algorithm for merging runs in order of \log squared n time using n by \log square n processors this has a cost of order n^2 and therefore, this is optimal 2. If we have a merging algorithm that runs in order of $\log n$ time using n processors then it has a cost of order of $n \log n$ this is not optimal because the sequential time complexity of the problem of merging is order of n .

(Refer Slide Time: 36:31)



The image shows a handwritten table on a light gray background. The title 'Sorting (comp. based)' is underlined in red. Below it, the sequential complexity $\Theta(n \log n)$ is written. The table has two columns: 'time' and 'procs'. It lists three scenarios with their respective time and processor complexities, and a final column with a checkmark or cross indicating optimality.

<u>Sorting (comp. based)</u>			
$\Theta(n \log n)$			
$O(\log^2 n)$	$\frac{n}{\log n}$	$O(n \log n)$	✓
$O(\log n)$	n	$O(n \log n)$	✓
$O(\log^2 n)$	n	$O(n \log^2 n)$	✗
time	procs		

If we consider the problem of sorting let us say comparison based sorting we know that the sequential time complexity of the problem of sorting is $\Theta(n \log n)$ if the comparison is the only operation that is allowed on the keys. This is because there is a lower bound of $\Omega(n \log n)$ using comparison trees and we have algorithms that run in order of $n \log n$ time for example, heap sort and merge sort both have a worst case time complexity of order of $n \log n$.

Therefore the problem has a complexity of $\Theta(n \log n)$, if we manage to find a sorting algorithm that runs in order of $\log^2 n$ time using $n / \log n$ processors then this algorithm has a cost of order of $n \log n$ which is identical to the sequential complexity of the problem therefore, this algorithm is optimal. If we have a sorting algorithm that runs in order of $\log n$ time using n processors this again has a complexity of the cost of order of $n \log n$ the processor time product of this algorithm is order of $n \log n$ which is identical to the sequential complexity of the problem therefore, this is optimal as well.

If a third algorithm runs in order of $\log^2 n$ time using n processors then the cost of the algorithm is order of $n \log^2 n$ which is the processor time product, but this is not the time complexity of the problem in the sequential setting therefore, this is not optimal. So, now, you have an idea of what an optimal algorithm is many times optimality is some more important concern than speed up.

(Refer Slide Time: 38:47)

Degree of Parallelism

- The degree of parallelism of a parallel step is the number of instructions in it
- It is the same as the number of processors required to execute it in one clock cycle

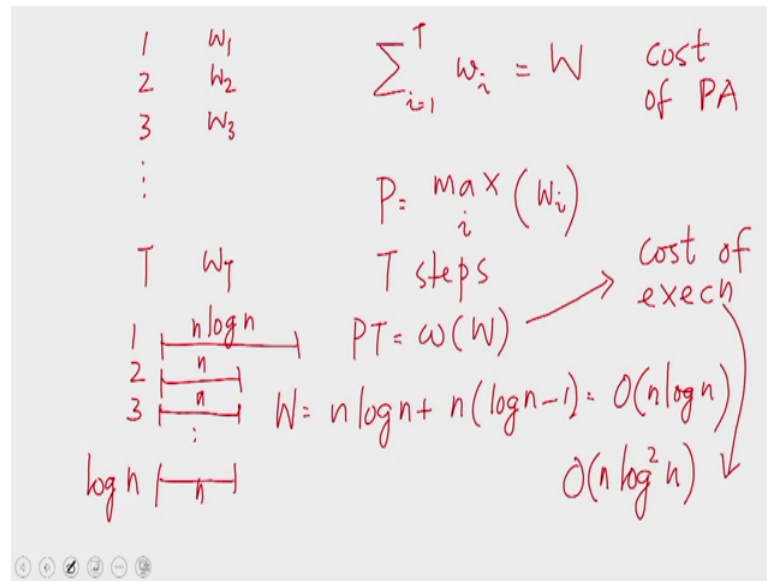
1	4 ins.
2	3 ins.
3	9 ins.
4	1
5	

Now, we come to the notion of the degree of parallelism when we specify an algorithm the algorithm will be usually divided into several steps, each step will have some number of instructions specified for example, let us say we have an algorithm with 5 steps.

Let us say we have a parallel algorithm with 5 steps, in the first step let us say we have 4 instructions, in the second step we have let us say 3 instructions, in the third step we have let us say 9 instructions and so on. So, what this means is that, the first step can be executed in one clock cycle if we have 4 processors we say the degree of parallelism of the first step is 4, similarly the degree of parallelism of the second step is 3, and the degree of parallelism of a third step is 9, in general the degree of parallelism of a parallel step is the number of instructions in it.

It is the same as the number of processors required to execute it in one clock cycle. So, when we specify an algorithm it is not necessary that the degree of parallelism of every step be the same.

(Refer Slide Time: 40:16)



So, let us say we have an algorithm that is specified which has T steps let us say the degrees of parallelism of the steps are w_1 through w_T then the total number of instructions in the algorithm we define as capital T capital W this is the cost of the algorithm, the cost of the parallel algorithm is capital W that is the total number of instructions specified in the parallel algorithm.

If we have enough processors we will be able to execute this parallel algorithm in T steps, but what is that enough number of processors, suppose we have these many processors the number of processors is equal to the largest degree of parallelism. If the number of processors available to us happens to be this then every single step of the specified algorithm can be executed in one single clock cycle therefore, the total time taken by the algorithm is capital T .

So, the algorithm runs in T steps, but it is possible that in this case the cost of the simulation the cost of the execution is omega of capital W that is the cost of the execution need not be the same as the cost of the specified algorithm in particular. Let us take an example where the first step of the algorithm has $n \log n$ instructions, let us say the second step has a size of n the degree of parallelism of the second step is n .

So, is that the case for the third step and so is the case for every subsequent step let us say we have a total of $\log n$ steps. So, here is a parallel algorithm that has $\log n$ steps the

degree of parallelism of the first step is $n \log n$ for every other step the degree of parallelism is n .

Therefore, the cost of this parallel algorithm is $n \log n$ plus n times $\log n$ minus 1 which is order of $n \log n$ this is what the cost of the parallel algorithm is, but if we use $n \log n$ processors then we will be able to execute every single step of this algorithm in one single clock cycle and the execution will take exactly $\log n$ steps, but in this case we would be using $n \log n$ processors and these $n \log n$ processes will have to be kept alive for $\log n$ steps.

Therefore the cost of the execution would be order of $n \log^2 n$ which is way more than the cost of the algorithm. So, if we use $n \log n$ processors then a large number of processors will have to remain idle for a majority of the steps and these idle steps will cause a lot of wastage. Now the question is, is there an execution of this algorithm that has a complexity identical to that of the cost of the parallel algorithm.

In fact, there is if the algorithm is specified on a self simulating model.

(Refer Slide Time: 44:35)

Model is self simulating
 p processors are there

$$\left\lceil \frac{w_1}{p} \right\rceil + \left\lceil \frac{w_2}{p} \right\rceil + \dots + \left\lceil \frac{w_T}{p} \right\rceil \leq$$

$$\left(\frac{w_1}{p} + 1 \right) + \left(\frac{w_2}{p} + 1 \right) + \dots + \left(\frac{w_T}{p} + 1 \right)$$

$$= \frac{W}{p} + T \quad p = \left\lceil \frac{W}{T} \right\rceil \quad \text{cost} = O(W)$$

\searrow
 $O(T)$

So, let us assume that the model is self simulating this is crucial and let us say we have small p processors.

Let us say with small p processors we try to simulate the various steps of the specified algorithm, then the first step can be specified the simulated in W_1 by P ceiling steps, the

second step can be simulated in the ceiling of $W/2$ by steps $W/2$ by P steps and so on. So, the simulation will take a total of this much time this we know is less than or equal to $W/2$ by P plus 1 $W/2$ by P plus 1 which is nothing, but capital W by P plus capital T if the model is self simulating with small p processors this algorithm can be executed in this much time.

What if we take small p as the ceiling of capital W by T , in this case we find that the execution will take order of T time W by P will become capital T and that plus another capital T will be order of capital T anyway. Therefore, this execution will take order of capital T time and the cost of the algorithm therefore, is P into the execution time which has order of W .

So, coming back to our previous example when we have an algorithm in which the first step has a degree of parallelism of $n \log n$ and every other step has a degree of parallelism of n and there are $\log n$ steps. Then in this case capital W is order of $n \log n$, T is $\log n$ therefore, the number of processes you should use is order of $n \log n$ by $\log n$ which is order of n .

So, if we have n processors let us see how the execution would proceed, the first step of the algorithm which is which has the degree of parallelism of $n \log n$ can be simulated with n processors in order of $\log n$ time, every subsequent step can be executed in exactly one step with n processors therefore, the total time taken would be $\log n$ plus $\log n$ minus 1, which is $2 \log n$ minus 1 which is order of $\log n$ in with n processors therefore, the cost of the execution is order of $n \log n$ which is also the cost of the algorithm.

So, what we find is that with P equal to ceiling of W by T processors if we execute the specified parallel algorithm the complexity of the cost of the execution is identical to the specified cost of the parallel algorithm.

(Refer Slide Time: 48:27)

Brent's Scheduling Principle

- Consider a parallel algorithm presented in T steps.
- Say the degree of parallelism of the i -th step is w_i
- So, the total number of instructions in the algorithm is $W = \sum_{i=1}^n w_i$
- If we use $P = \max_i(w_i)$ processors, the algorithm runs in exactly T steps
- But the cost of this execution may be $\omega(W)$
- For example, say $w_1 = n \log n$, while $w_i = n$, for $i > 1$, and $T = \log n$
- The above execution takes $T = \log n$ time with $w_1 = n \log n$ processors
- The cost is $O(n \log^2 n)$, whereas $W = O(n \log n)$

This principle is called Brent's scheduling principle; Brent's scheduling principle allows us to specify an algorithm with varying degrees of parallelism for its steps that is we do not have to explicitly worry about the processor allocation of the individual steps. We can freely design the algorithm by choosing an appropriate degree of parallelism for every single step.

The fact that the degrees of parallelism vary from step to step will not affect the cost of the execution using Brent scheduling principle we can still ensure that the cost of the execution is identical to the cost of the specified cost of the parallel algorithm. The only condition is that the model that we use must be a self-simulating model that is it from this lecture, hope to see you in the next lecture.