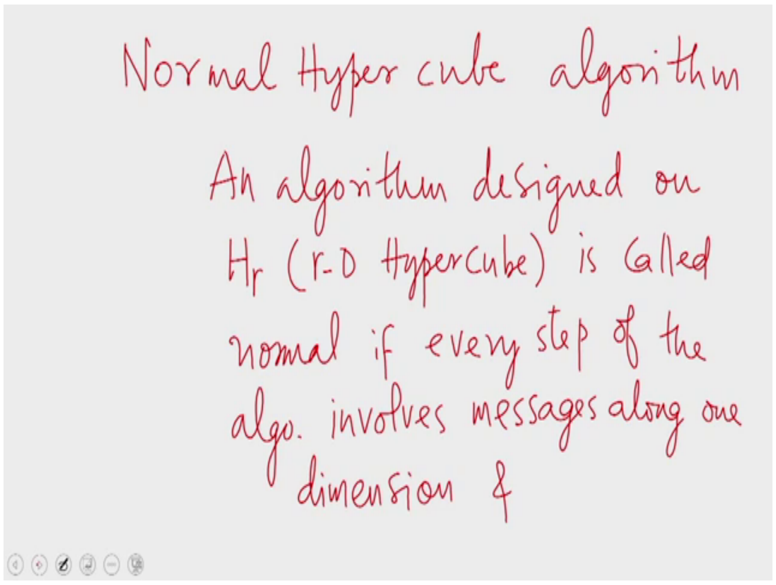
Parallel Algorithms

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

Lecture - 34
Interconnection Networks Algorithms

Welcome to the 34 lecture of the MOOC on Parallel Algorithms. In the last class, we saw a couple of architectures that are very closely related to hypercubes. For examples, shuffle exchange, graphs and De Bruijn graphs, we saw in the last class. Today we shall see some use of them in simulating algorithms that are designed for hyper cubes. In particular, we shall see normal hypercube algorithms first.

(Refer Slide Time: 00:55)

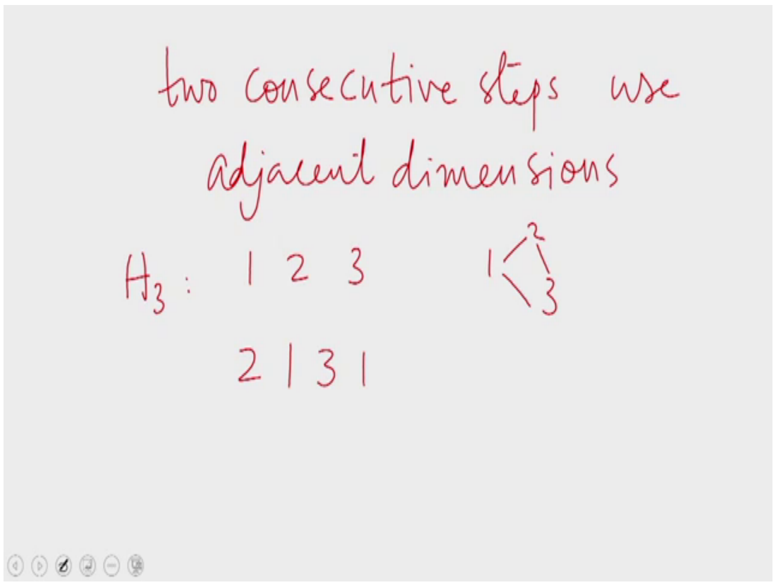


Normal Hyper cube algorithm

An algorithm designed on H_r (r-D Hypercube) is Called normal if every step of the algo. involves messages along one dimension &

An algorithm designed on H r, the r-dimensional hypercube is called normal if every step of the algorithm involves messages along one dimension.

(Refer Slide Time: 01:35)


two consecutive steps use adjacent dimensions

H_3 : 1 2 3

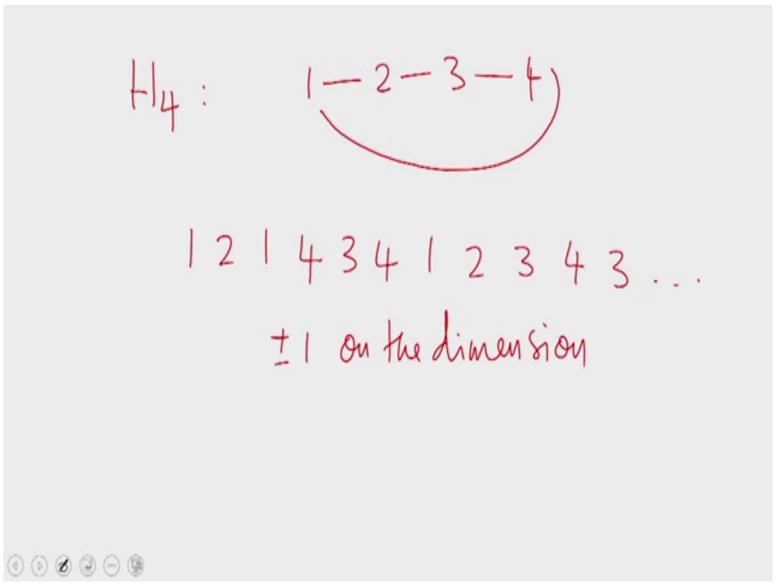
2 1 3 1

And two consecutive steps use adjacent dimensions. For example, if you consider H 3 and if the dimensions are numbered 1 2 3, then the requirement is that every step should use any one of these, these, these three dimensions and two consecutive steps should use adjacent dimensions.

For example, if the first step of the algorithm uses dimension 2, then all messages that are passed in the first step should be along the second dimension. Now, in the second step the dimension used should be consecutive with 2, should be adjacent to 2, therefore it should be either 1 or 3. So, let us say we choose 1 for the second step.

Then in the third step, the dimension used should be an adjacent 1, the adjacent dimension could be either 2 or 3, because adjacency also go cyclically. So, the adjacencies are then defined in this fashion for a three-dimensional hypercube. So, the next dimension could be 2 or 3; then for 3, it could be either 1 or 2 and so on.

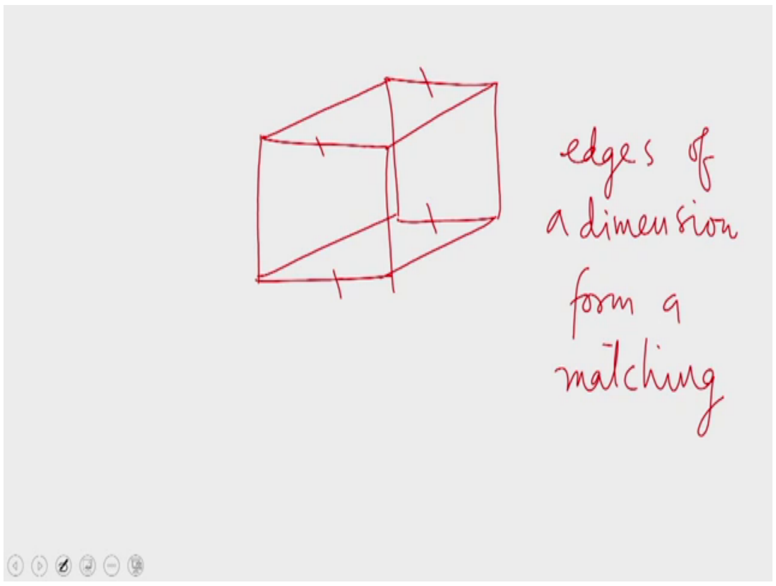
(Refer Slide Time: 02:44)



In particular for a hypercube 4, the adjacencies are like this, dimension 1 is adjacent to both 2 and 4; 2 is adjacent to 1 and 3, 3 is adjacent to 2 and 4; and 4 is the adjacent to 3 and 1. So, one possible dimension sequence for a normal hypercube algorithm that runs on H 4 would be 1, then 2, then back to 1, then 4, then 3, then 4 again, then 1, then forward to 2, then 3, then 4 3 and so on.

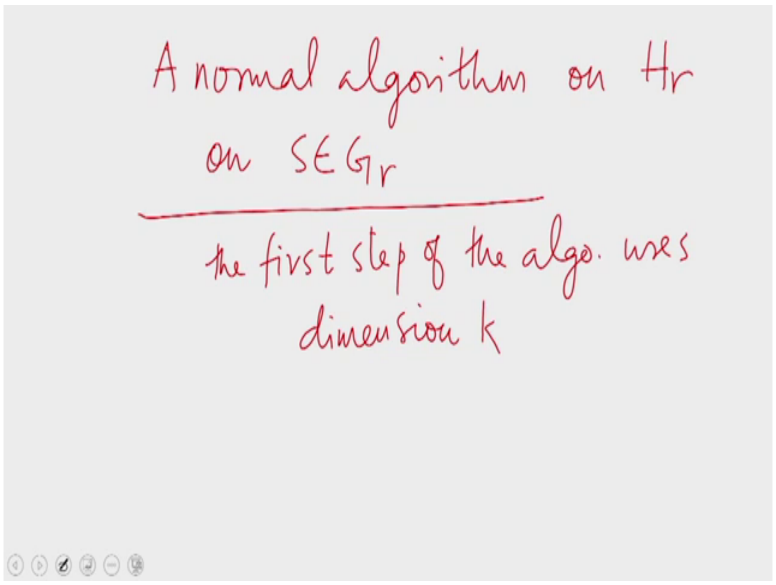
So, always we have plus or minus 1 on dimension, when you go from one step to the next. The dimension either increases or decreases, when the dimension decreases from 1, it goes to 4; when the dimension increases from 4, it goes to 1. So, an algorithm which uses dimensions of this sort is called a normal algorithm.

(Refer Slide Time: 03:41)



For example, when you consider a hypercube of three dimensions, edges of one particular dimension, this is the horizontal dimension here; these edges form a matching. Therefore, in a normal hypercube algorithm the therefore, in a normal hypercube algorithm the messages that are passed in any one particular step will be along a matching. Therefore, one message will involve only the source and the destination and one node will be involved only with a couple of messages one message it sends and one message it receives. So, now let us see how a normal hypercube algorithm can be simulated on a shuffle exchange graph.

(Refer Slide Time: 04:27)

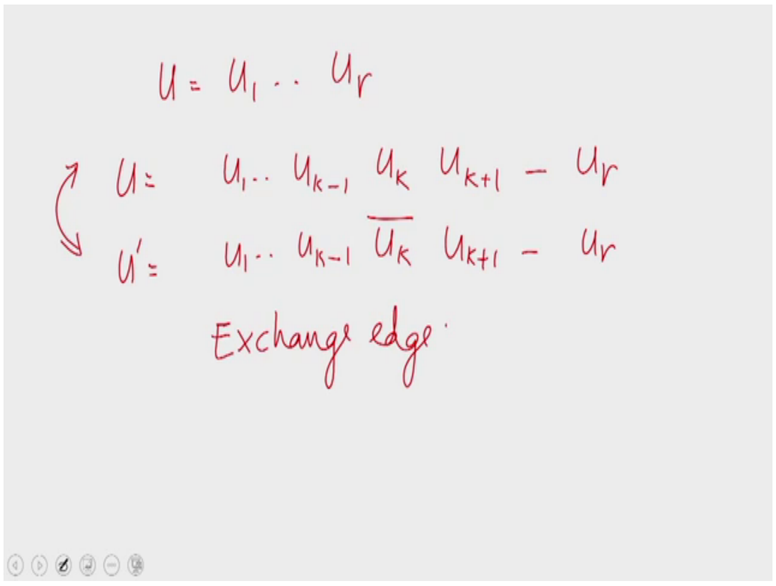


A normal algorithm on H_r on SEG_r

the first step of the algo. uses dimension k

A normal algorithm on H r on a shuffle exchange graph of dimension r. Let us say, the first step of the algorithm uses dimension k. The algorithm that is to be simulated has this property, let us say the first step of the algorithm uses dimension k.

(Refer Slide Time: 04:51)



$$u = u_1 \cdots u_r$$

$$u = u_1 \cdots u_{k-1}\; u_k\; u_{k+1} - u_r$$

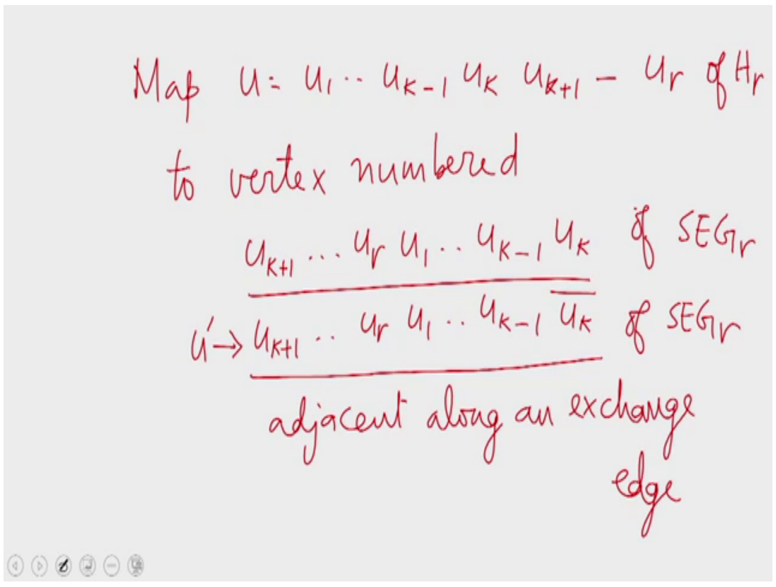
$$u' = u_1 \cdots u_{k-1}\; \overline{u_k}\; u_{k+1} - u_r$$

Exchange edge.

In that case, if you consider a vertex u equals u 1 through u r in particular if I make the dimension k explicit, then this is what u is u will be communicating with dimension k neighbor, the number of which will be identical to u at all positions other than the k th. So, u prime is the dimension k neighbor of u, these two should be communicating in the first step of the algorithm.

Now, how do we ensure that these two communicate in the first step on a shuffle exchange graph. On a shuffle exchange graph, we will use the exchange edges for the actual communication.

(Refer Slide Time: 05:36)



Map $u = u_1 \cdots u_{k-1} u_k u_{k+1} - u_r$ of H_r

to vertex numbered

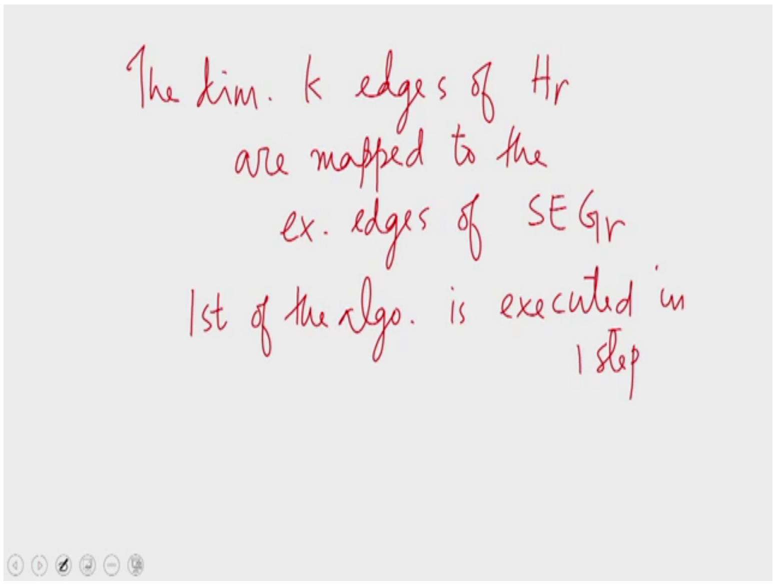
$\underline{u_{k+1} \cdots u_r u_1 \cdots u_{k-1} u_k}$ of SEG_r

$u' \rightarrow \underline{u_{k+1} \cdots u_r u_1 \cdots u_{k-1} \overline{u_k}}$ of SEG_r

adjacent along an exchange edge

What we do is this? Map this vertex to the vertex number of the shuffle exchange graph that is vertex u of H r is mapped to this vertex of the shuffle exchange graph. In that case, u prime will get mapped to this node of the shuffle exchange graph. You can see that these two vertices are adjacent and they are adjacent along the exchange edge.

(Refer Slide Time: 06:19)

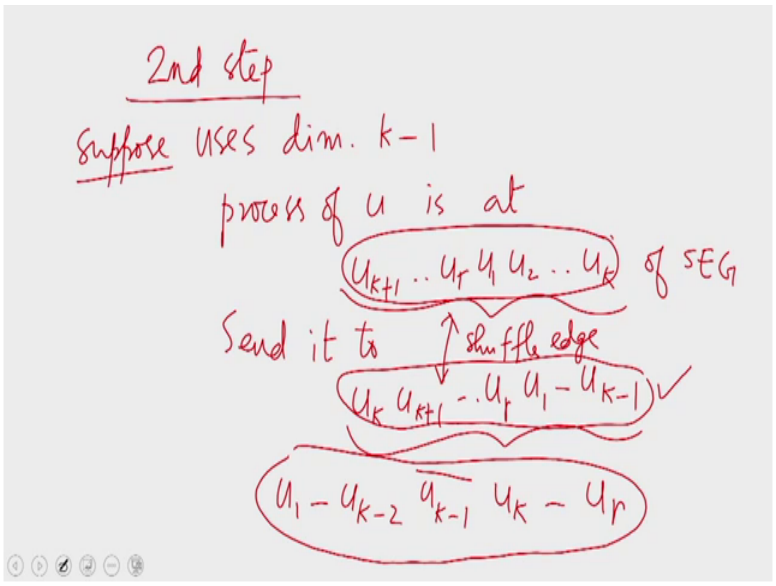


The dim. k edges of H_r are mapped to the ex. edges of SEG_r

1st of the algo. is executed in 1 step

What we find is that the dimension k edges of the hypercube are mapped to the exchange edges of the shuffle exchange graph. Therefore all of them can now communicate in order one time, in other words the first step of the algorithm is executed in one step.

(Refer Slide Time: 06:43)



2nd step

Suppose uses dim. k − 1

process of u is at

$(u_{k+1} .. u_r u_1 u_2 .. u_k)$ of SEG

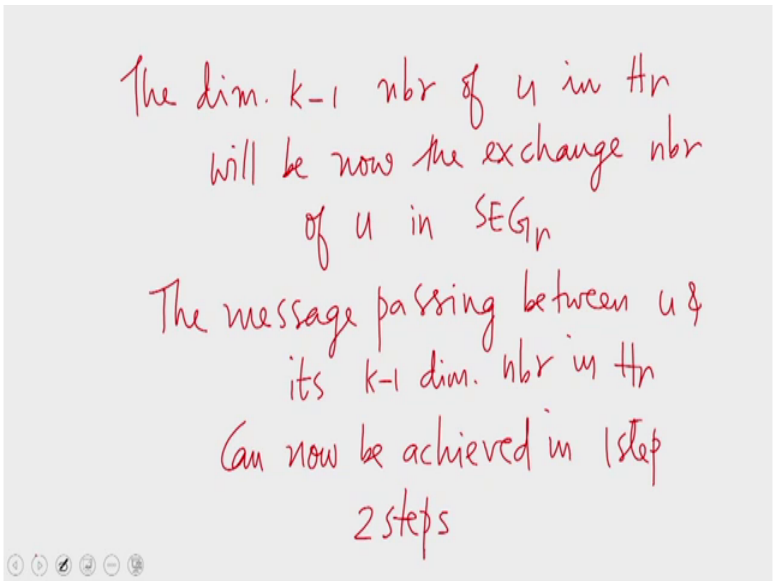
Send it to $\uparrow$ shuffle edge

$(u_k u_{k+1} .. u_r u_1 - u_{k-1})$ ✓

$(u_1 - u_{k-2} u_{k-1} u_k - u_r)$

Now, let us come to the second step. The second step of the algorithm will use either dimension k plus 1 or dimension k minus 1. Suppose it uses dimension k minus 1, suppose the second step uses dimension k minus 1, then what we do is this the process of vertex u is currently at this node of the shuffle exchange graph. Let us send it to this vertex instead, the process of vertex u is now sent it to is now sent to vertex u k u k plus 1 through u r and u 1 through u k minus 1.

(Refer Slide Time: 07:34)



The dim. k−1 nbr of u in H_r will be now the exchange nbr of u in SEG_r

The message passing between u & its k−1 dim. nbr in H_r can now be achieved in ~~1 step~~ 2 steps

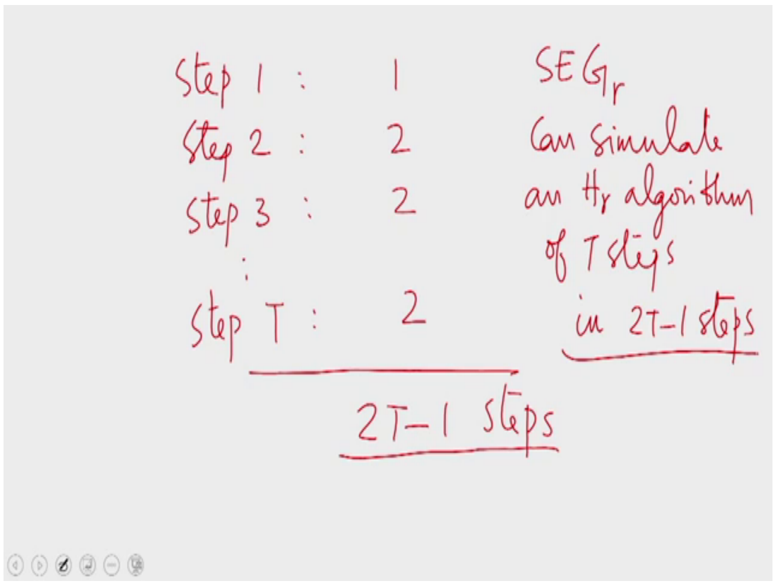
Then the k minus 1 dimensional neighbor of u will be now the exchange neighbor of u in the shuffle exchange graph or strictly speaking the dimension k minus 1 neighbor of u in H r will be now, the at the exchange neighbor of u in S E G r that is u and its k minus 1 dimensional neighbor in H r will now be exchanged neighbors in S E G r that is the process, which was which corresponds to vertex u that was initially at this vertex, has gone to this vertex now.

Similarly, the process that corresponds to the k minus 1 dimensional neighbor of u, which is this is the k minus 1 dimensional neighbor of u. The process of this node in H r would have now reached the exchange neighbor of this vertex in the shuffle exchange graph. Therefore the message passing between those two nodes u and its k minus 1 dimensional neighbor in H r can now be achieved in order one time in one step.

Then what is the total amount of time that we have taken for simulating the second step, one step is needed to send the process to the appropriate node, but we see that this vertex and this vertex are adjacent using a shuffle edge. So, what is needed is that the process of u which was at this vertex initially has to move to this vertex, which happens to be a shuffle neighbor of this vertex; so that moment can be achieved using a shuffle edge.

So, the time taken for this repositioning of the process of vertex u of H r is order one is one step, and then another one step is needed for simulating the actual step. So, we have taken a total of two steps for simulating the second step.

(Refer Slide Time: 09:48)



Step 1 :	1
Step 2 :	2
Step 3 :	2
⋮	
Step T :	2

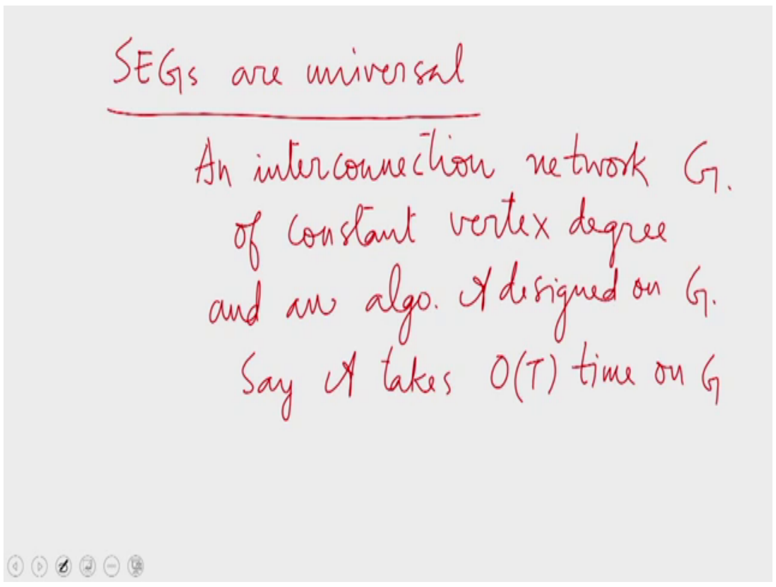
$\underline{2T-1 \text{ steps}}$

SEG_r can simulate an H_r algorithm of T steps $\underline{\text{in } 2T-1 \text{ steps}}$

And then continuing like this we find that step 1 is simulated in 1 step, step 2 requires the repositioning of the process therefore, we will take a total of 2 steps, step 3 again uses an adjacent dimension. So, the dimension would be either k or k minus 2; if the second step had used k minus 1, then it would take an additional 2 steps. And continuing like this, the last step of the algorithm which let us say is the T th step again would take 2 steps therefore, the entire simulation would take 2 T minus 1 steps.

What this concludes is that S E G r can simulate an H r algorithm of T steps that is algorithm takes T steps on H r in 2 T minus 1 steps. So, S E G r is quite versatile indeed, normal hypercube algorithms can be simulated on S E G r at double the time, nearly double the time. Now, let us prove that shuffle exchange graphs are universal as well.

(Refer Slide Time: 10:55)

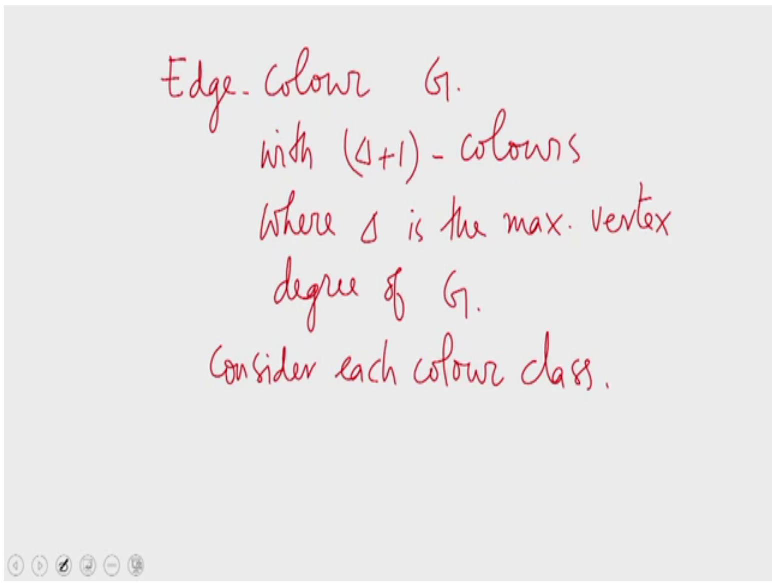


SEGs are universal

An interconnection network G. of constant vertex degree and an algo. $\mathcal{A}$ designed on G. Say $\mathcal{A}$ takes $O(T)$ time on G

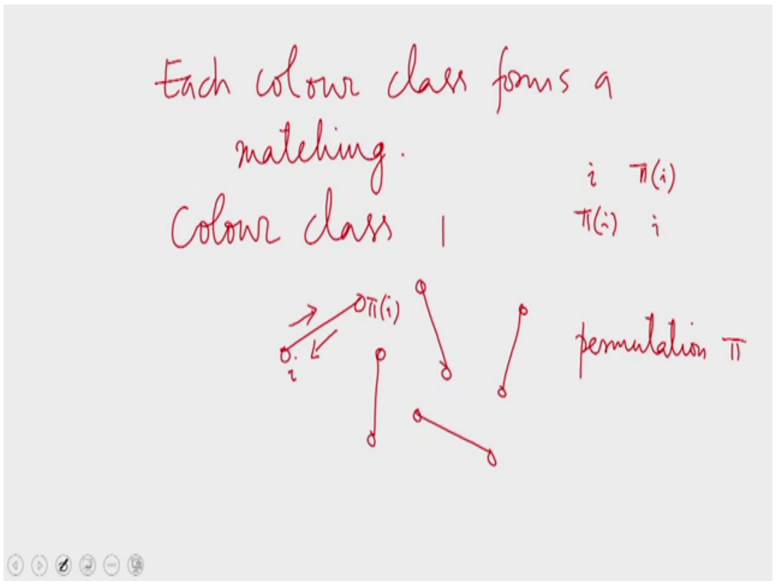
To show this, we consider an interconnection network of constant degree. So, let us call this network G and an algorithm A defined on G. Say A takes order of T steps on G. Let us say, we want to simulate this on a shuffle exchange graph. First we will see how to simulate this on a hypercube using a normal algorithm, the simulation would be a normal algorithm and then the normal simulation can be run on and S E G of the same dimension.

(Refer Slide Time: 11:43)


Edge. Colour G.
with $(\Delta+1)$ - colours
Where Δ is the max. vertex
degree of G.
Consider each colour class.

So, we have a fixed interconnection network, this is like the simulation we had seen earlier let us edge colour the graph G with the delta plus 1 colours, where delta is the maximum vertex degree of G. Now, that the graph is coloured consider each colour class separately.

(Refer Slide Time: 12:12)

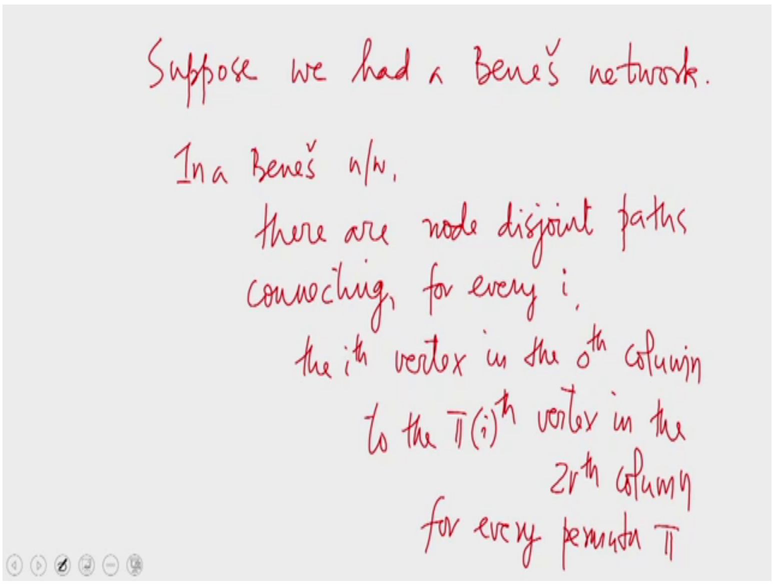

Each colour class forms a
matching.
Colour class 1

So, each colour class forms are a matching. So, let us consider colour class 1, every other colour class is identical. So, whatever we do for colour class 1 can be replicated for the other colour classes. So, when you look at colour the first colour class you find the matching of the sort in the network, and the messages have to be passed between these nodes.

So, if this is node i, let us say this node by a pi of i a message has to be sent in this direction, and a message also could may have to be sent in this direction. So, if you consider all these mappings i to pi of i and pi of i to i, what we get is a permutation of the vertices that is for every vertex we have a destination for the message which is originating at that vertex.

So, this essentially specifies the permutation pi that is the messages which are present at the sources have to be permuted and sent to the destinations. So, the message which is at i has to be sent to pi of i for every i, this is what is to be achieved for colour class 1. And once we manage this, we can do the same thing for every single colour class.

(Refer Slide Time: 13:30)



Suppose we had a Beneš network.

In a Beneš n/w,

there are node disjoint paths connecting, for every i,

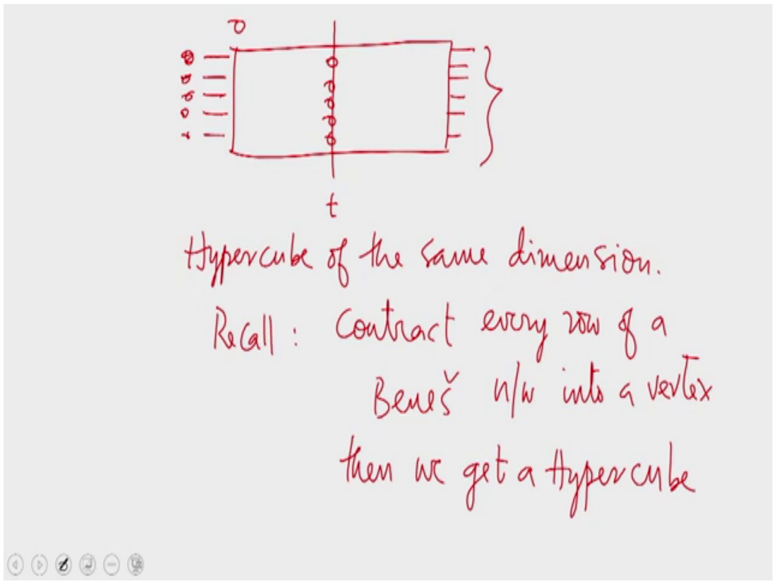
the i th vertex in the 0 th column

to the $\pi(i)$ th vertex in the 2r th column

for every permutn π

Now, suppose we had a Benes network. We know that in a Benes network, there are node disjoint edges, node disjoint paths connecting for every i. The i th vertex in the first column or the 0 th column to the pi i th vertex in the 2 r th column for every permutation pi of the inputs. So, we will use this property of a Benes network.

(Refer Slide Time: 14:24)



Hypercube of the same dimension.

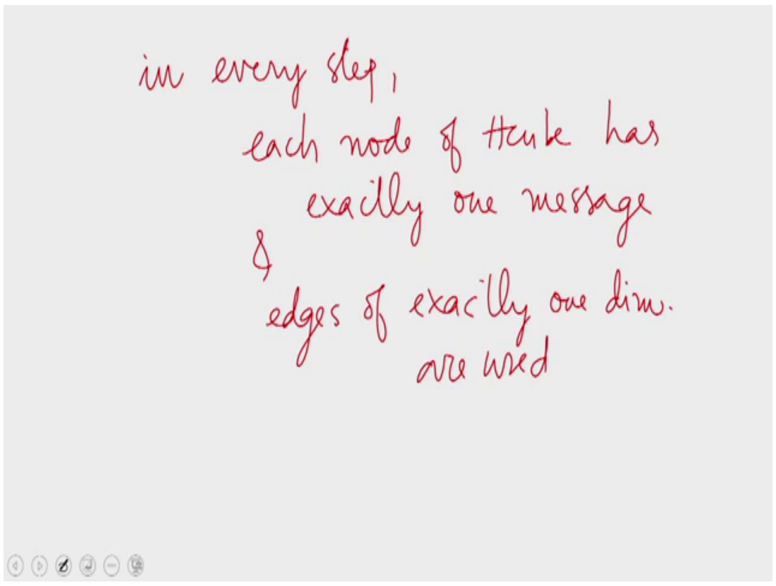
Recall : Contract every row of a Beneš n/w into a vertex then we get a hypercube

So, if we had a Benes network and if we were permuting the packets using a Benes network, then all we would have to do would be to line up the messages at the 0 th column, this is the 0 th column. So, we will line up the messages at the 0 th column, and then send the messages through the Benes network in lockstep in the sense that at any particular time all the messages will be in the same column that is after t steps all of them will be in the t th column.

So, after 2 r steps all of them would be coming out at the right end in the appropriately permuted form that is if we had a Benes network, but we do not have a Benes network, but let us assume that we have a hypercube of the same dimension, but how can a hypercube simulate a Benes network. Recall that if we contract every single row of a Benes network or a butterfly for that matter a Benes network is nothing but 2 butterfly pasted back to back. So, if you contract an entire row of a Benes network into a single vertex, then we get a hypercube.

And then if you run a Benes network algorithm, which pulses the data through the Benes network in a lockstep manner in such a way that all the messages are lined up in a column in every single step.

(Refer Slide Time: 15:51)



in every step,

each node of Hcube has exactly one message

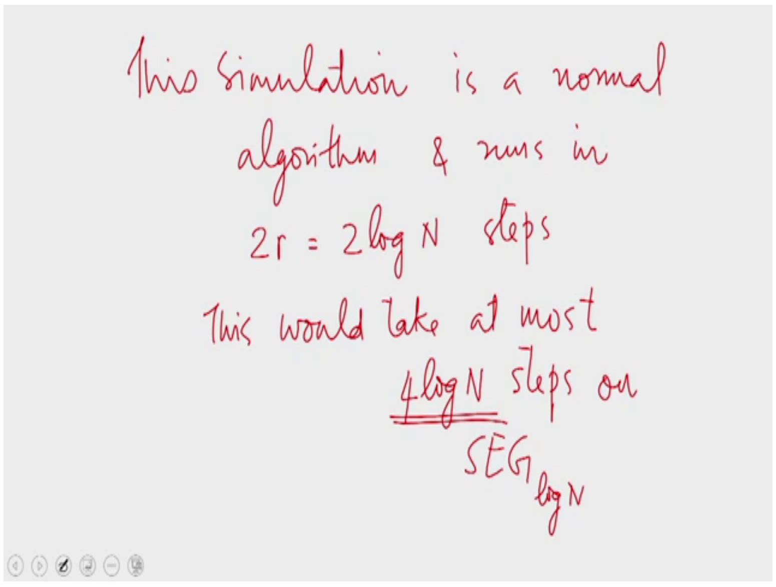
&

edges of exactly one dim. are used

Then you find that if you run it on a hypercube in every step, each node of the hypercube has exactly one message and in every step edges of exactly one dimension are used that is because in every step the messages are passing from one column of a Benes network to the adjacent one, when these are collapsed these edges will all correspond to edges along a particular dimension of the hypercube.

So, what we find is that, the permutation that we achieve using a Benes network can in fact, be achieved also using a hypercube, but then this hypercube algorithm this simulation in fact, turns out to be a normal algorithm.

(Refer Slide Time: 16:42)


This Simulation is a normal algorithm & runs in

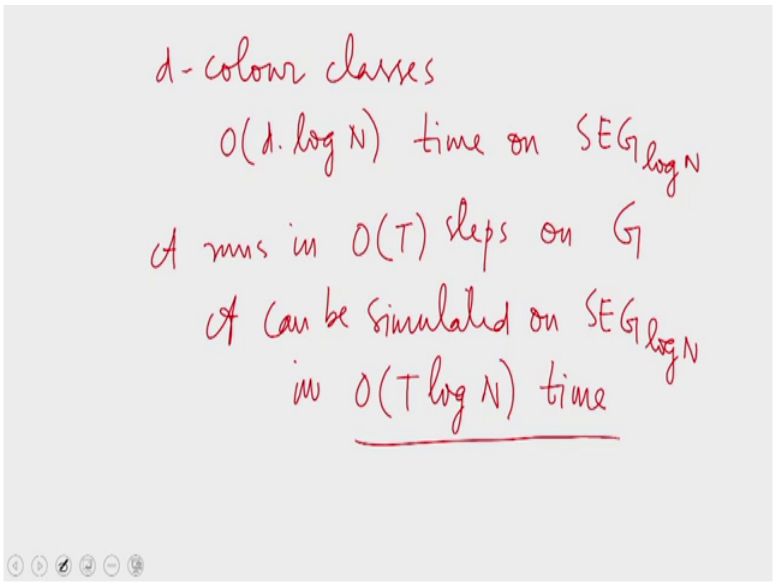
$2r = 2\log N$ steps

This would take at most $\underline{\underline{4\log N}}$ steps on $SEG_{\log N}$

This simulation is a normal algorithm and runs in 2 r steps, but 2 r is nothing but 2 log N. So, this algorithm runs in 2 log N steps at the most, but we have already seen that an algorithm which runs in t is a normal algorithm that runs in t steps on a hypercube can be simulated on a shuffle exchange graph of the same dimensions in 2 T minus 1 steps. Therefore this would take at most 4 log N steps to ignore the smaller order terms, on a shuffle exchange graph of the same dimensions which is log N.

So, what we find is that one single step can be simulated in order of log N time, in a shuffle exchange graph, but this is only one of the steps that is we considered only one colour class.

(Refer Slide Time: 17:51)



d - colour classes

$O(d.\log N)$ time on $SEG_{\log N}$

$\mathcal{A}$ runs in $O(T)$ steps on G

$\mathcal{A}$ can be simulated on $SEG_{\log N}$

in $\underline{O(T \log N) \text{ time}}$

But since the interconnection network had d - colour classes, simulating all these colour classes would take order of d log N time on shuffle exchange graph of dimensions log N. Therefore algorithm A that runs in order of T steps on our general constant degree interconnection network G can be simulated on an S E G of dimensions log N.

And S E G of dimensions log N and G have the same number of vertices. So, we are essentially considering an S E G of the same number of vertices, the simulation takes order of T log N time, which means a shuffle exchange graph is capable of simulating any general interconnection network of constant degree with an order log N factor overhead. This establishes the universality of the universe the universality of the model shuffle exchange graphs.

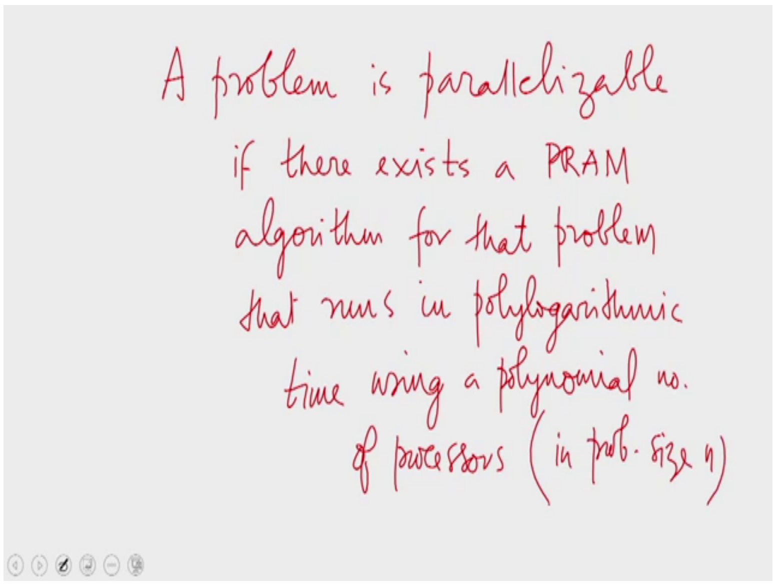
Whatever, you do on a shuffle exchange graph can also be done on a de Bruijn graph because of the as we saw in the last class at de Bruijn graph is a minor modification of a shuffle exchange graph, the two can simulate each other with a constant overhead. Therefore, these two models are universal indeed with that we come to the end of our discussion on interconnection network algorithms.

(Refer Slide Time: 19:20)

Limits to Parallelizat

Now, we are going back to PRAM. And let us explore some limits to parallelization.

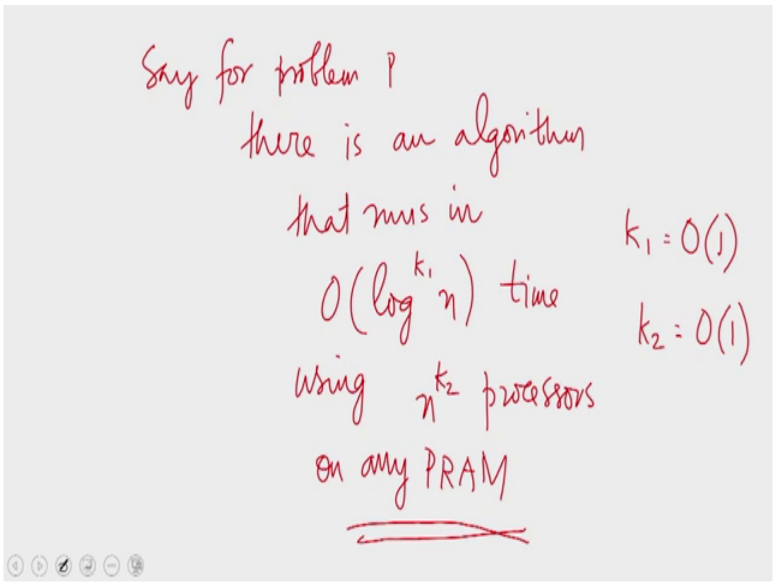
(Refer Slide Time: 19:28)



A problem is parallelizable if there exists a PRAM algorithm for that problem that runs in polylogarithmic time using a polynomial no. of processors (in prob. size n)

When do we say that a problem is parallelizable? So, we use this following definition a problem we say is parallelizable if there exists a PRAM algorithm for that problem, the variant of PRAM is of no relevance that runs in polylogarithmic time using a polynomial number of processors in problem size n, what it means is this.

(Refer Slide Time: 20:07)



Say for problem P
there is an algorithm
that runs in
$O(\log^{k_1} n)$ time
using n^{k_2} processors
on any PRAM

$k_1 = O(1)$

$k_2 = O(1)$

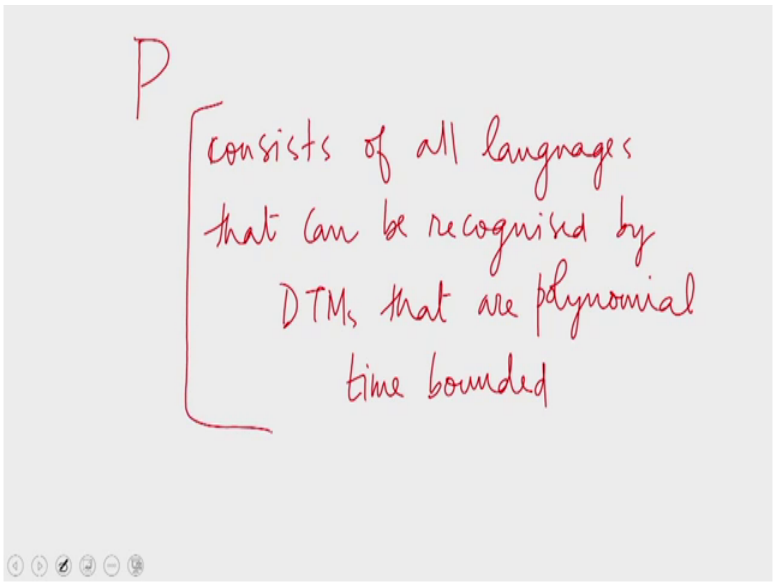
Say for problem P, there is an algorithm that runs in order of log n to the power k 1 time. So, here k 1 is a constant it is independent of n, when we say that it is a constant what we mean is that it is independent of n, using n power k 2 processors where k 2 is order of 1 again, it is a constant independent of n on any PRAM.

Now, this definition is independent of the variant of the PRAM that is because all models of PRAMs can simulate each other in poly logarithmic time using a polynomial number of processors. There is even if you have an algorithm that runs on a priority C R C W PRAM, it can be simulated an E R E W PRAM using a polynomial number of processors in poly logarithmic time. All you have to do is to sort the processors on the address of the location that they want to write to.

And then the conflict resolution can be achieved by finding the least process that wants to write in each memory location. The details I will leave to you as an exercise, but then you will find that the algorithm runs in order of log squared n time extra factor that is if the original algorithm ran in log of n power k 1, then the simulation would run in order of log n power k plus 2, which again is poly logarithmic using a polynomial number of processors.

Therefore, this definition is independent of the variant of the PRAM. If this holds on any one variant of the PRAM, it will hold on another variant of the PRAM. Therefore, if a problem is parallelizable on one variant of the PRAM, it is parallelizable on any other variant in this sense. So, this is let us say our definition of parallelizability. Now, let us see what are the consequences of this definition.

(Refer Slide Time: 22:15)

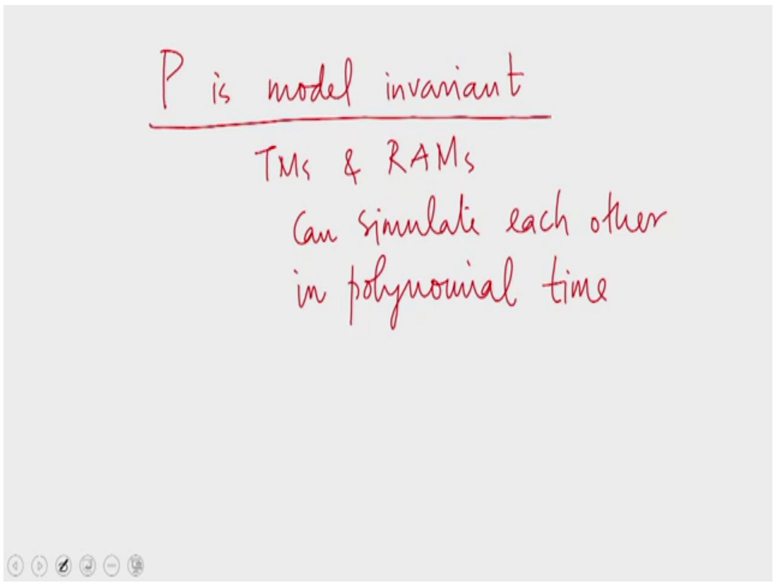


P

consists of all languages that can be recognised by DTMs that are polynomial time bounded

There is this complexity class P that you are all familiar with. P is the class of all languages that can be recognized by Deterministic Turing Machines that are polynomial time bounded. In other words these are the problems that can be solved in polynomial amount of time. So, we are considering only the language recognition problems. So, P is the class of all such languages.

(Refer Slide Time: 22:51)



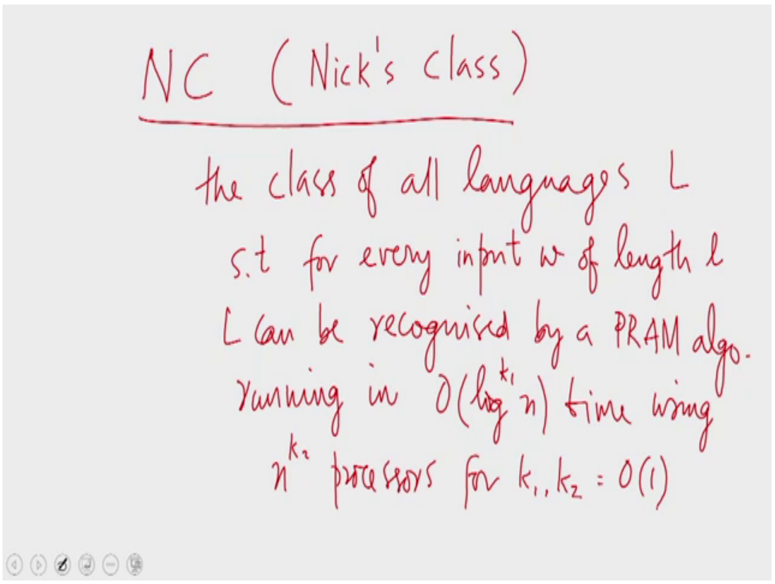
P is model invariant

TMs & RAMs can simulate each other in polynomial time

Now, we know that P is model invariant. Exactly as we spoke about the variance of PRAMs, we can also consider the variance of models of computations. For example, Turing Machines and Random Access Machines and various other models of computation that have been proposed, can simulate each other in polynomial time.

Therefore, what you managed to do on one model can be done on the other model in polynomial time as well, that is why we see that P is model invariant; exactly the way that our notion of parallelizability is independent of the PRAM invariant.

(Refer Slide Time: 23:49)



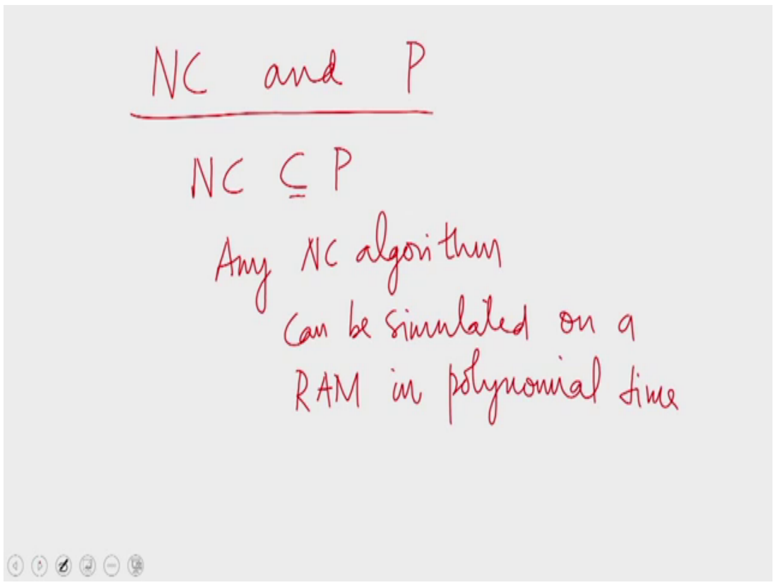
NC (Nick's class)

the class of all languages L s.t for every input w of length l L can be recognised by a PRAM algo. running in $O(\log^{k_1} n)$ time using n^{k_2} processors for $k_1, k_2 = O(1)$

Now, let us define the complexity class NC, which is short for Nick's Class after Nick Dipendra. We define NC as the class of all languages, L such that for every input w of length l. L can be recognized by a PRAM algorithm, as I said before the variant of the PRAM algorithm this irrelevant.

This PRAM algorithm should run in order of log n power k 1 time using n power k 2 processors for k 1 and k 2 that are order 1, that is k 1 and k 2 are independent of n all languages of this sort belong to the complexity class NC.

(Refer Slide Time: 25:10)



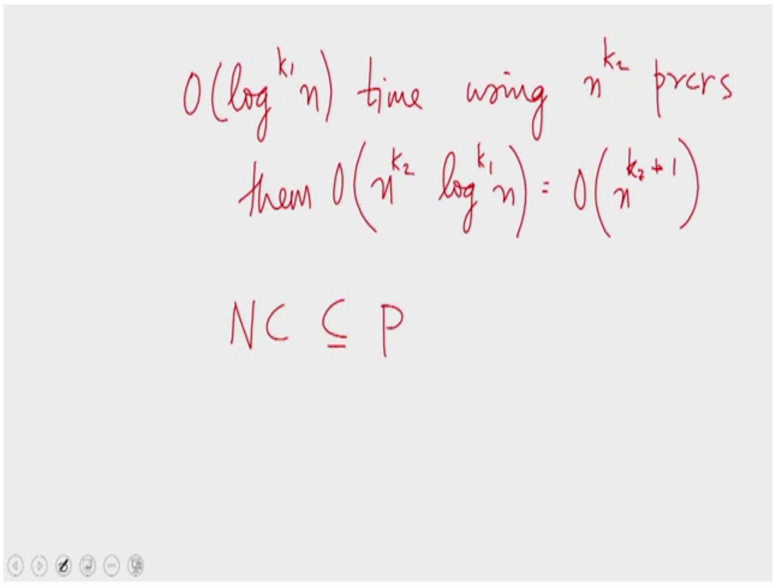
NC and P

$NC \subseteq P$

Any NC algorithm can be simulated on a RAM in polynomial time

Now, what can we say about NC and P now, it is obvious that NC is a subset of P that is because any NC algorithm. In other words an algorithm that runs in poly logarithmic time using a polynomial number of processors is an NC algorithm; it witnesses the membership of some language in NC. So, any NC algorithm can be simulated on a single processor machine, which is a random access machine in polynomial time.

(Refer Slide Time: 25:49)



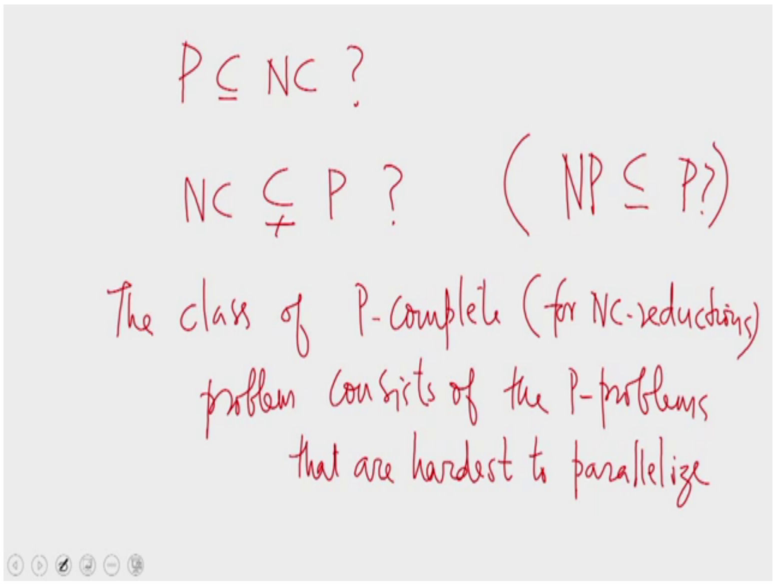
$O(\log^{k_1} n)$ time using n^{k_2} prcrs

then $O(n^{k_2} \log^{k_1} n) = O(n^{k_2+1})$

$NC \subseteq P$

For example, if an algorithm runs in order of log n power k 1 time using n power k 2 processors, then the cost of this algorithm is n power k 2 times log of n power k 1, which happens to be order of n power k plus 2, k 2 plus k 2 plus 1, which is a polynomial. Therefore any algorithm in NC is any problem in NC is also in P or in other words NC is a subset of P.

(Refer Slide Time: 26:37)



$P \subseteq NC$?

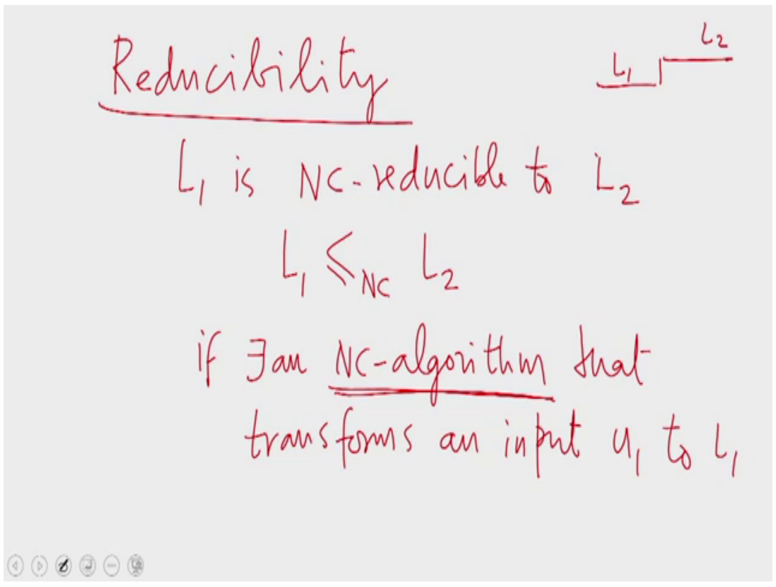
$NC \subsetneq P$? $(NP \subseteq P?)$

The class of P-complete (for NC-reductions) problem consists of the P-problems that are hardest to parallelize

But then does the converse hold; is P a subset of NC or in other words this NC a proper subset of P, this is a question to which we do not have an answer here. This is analogous to the P equal to NP question, this question is still open, similarly the question whether P is a subset of NC is still open.

We shall define a class of problems, which we call the P - complete problems for NC reductions. The class of P complete problems consist of the P problems that are hardest to paralyze, that is we will be able to paralyze them only if NC equal to P. So, if P is not equal to NC, then there are some P - problems which are not parallelizable. And then the P complete problems would indeed be not parallelizable.

(Refer Slide Time: 27:52)



Reducibility

L_1 — L_2

L_1 is NC-reducible to L_2

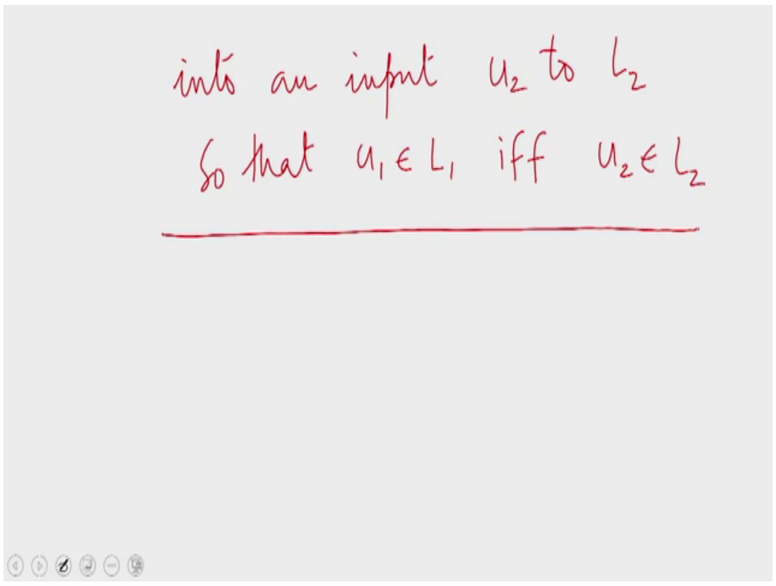
$L_1 \leqslant_{NC} L_2$

if $\exists$ an NC-algorithm that transforms an input u_1 to L_1

So, what do I mean by NC reductions, let me define the reducibility. We say that L 1 is NC reducible to L 2, which we denote as in this manner. So, you can imagine that L 1 has been reduced to L 2. So, you must be clear about the order of this inequality L 1 is reducible to L 2 means L 2 is the harder problem.

So, L 1 is NC reducible to L 2, if there exist an NC algorithm. As I said before, an NC algorithm is one which runs in poly logarithmic time using a polynomial number of processors. There exist an NC algorithm that transforms an input u 1 to L 1 that is we are given a string u 1, and we want to check whether u 1 belongs to L 1.

(Refer Slide Time: 28:58)

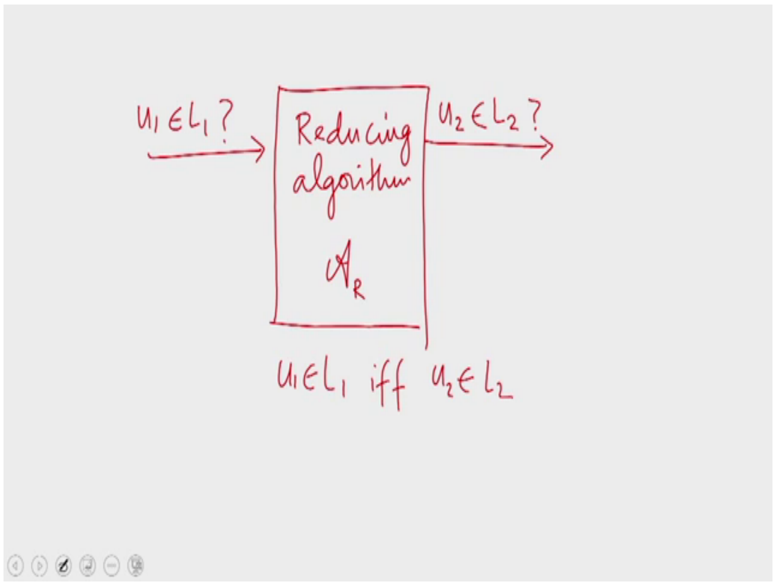


into an input u_2 to L_2

so that $u_1 \in L_1$ iff $u_2 \in L_2$

Into an input u 2 to L 2, so that u 1 belongs to L 1, if and only if u 2 belongs to L 2; if this is the case then we say that L 1 is NC reducible to L 2. So, they must exist an NC algorithm which is the reducer. So, there is a reducing algorithm.

(Refer Slide Time: 29:28)

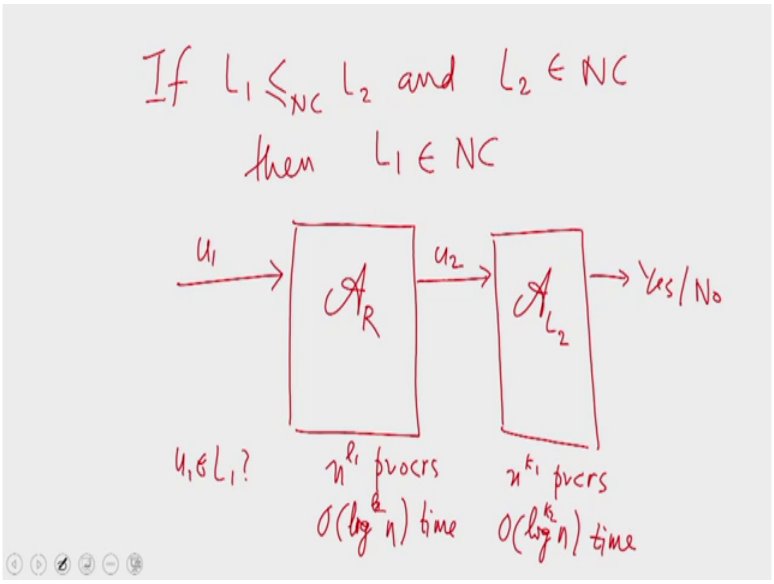


So, this is the reducing algorithm let me denote that by A R, this is the reducing algorithm which takes an input u 1, this is an input for L 1 that is we want to decide whether u 1 belongs to L 1 or not? And then what it produces is u 2, which is an input to

L 2. And this translation of u 1 into u 2 will take poly logarithmic time using a polynomial number of processors.

And what we know is that u 1 belongs to L 1, if and only if u 2 belongs to L 2. So, the translation is so that u 1 belongs to L 1, if and only if u 2 belongs to L 2.

(Refer Slide Time: 30:15)



If $L_1 \leq_{NC} L_2$ and $L_2 \in NC$ then $L_1 \in NC$

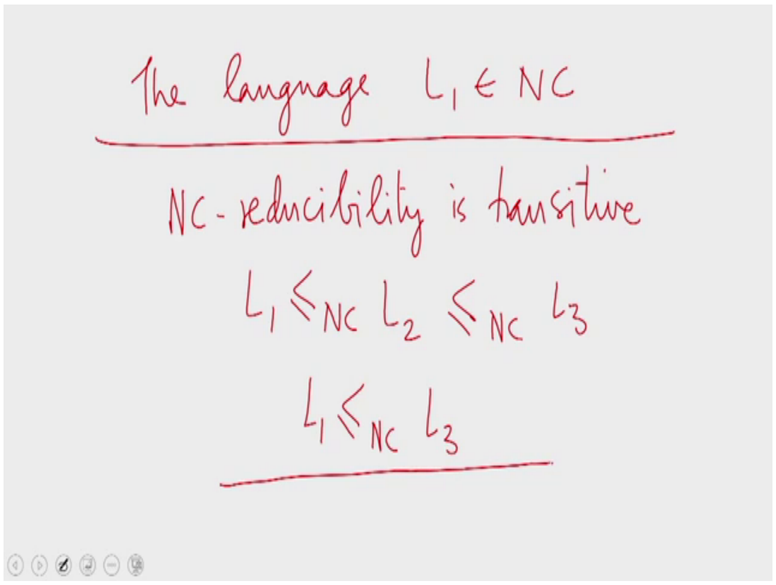
What this entails is that if L 1 is NC reducible to L 2 and L 2 belongs to NC, then L 1 also belongs to NC why is this, let us say we want to construct a recognizer for L 2, for L 1. What we have is a recognizer for L 2? Let us say, L 2 belongs to NC. So, L 2 has an algorithm A which runs in poly logarithmic time using a polynomial number of processors.

And then we have this reducer A R. And let us say, we are given an input u 1, A R produces an input output u 2; what we know is that u 1 belongs to L 1, if and only if u 2 belongs to L 2. The question that is posed to us is this does u 1 belongs to L 1? Does u 1 belong to L 1? Then the translator ensures that u 2 is in L 2, if and only if u 2 u 1 belongs to L 1, but we have a decider for u 2, we have a decider for language L 2. So, if u 2 is fed to that it will produce either yes or no, depending on whether u 2 belongs to L 2 or not.

So, the membership of u 2 in L 2 can be tested in poly logarithmic time using a polynomial number of processors. So, let us say this takes n power k 1 processors and order of log n power k 2 time and let us say, A R takes n power l 1 processors and order of log n power l 2 time.

Then when you juxtapose these two algorithms, and use the larger of the number of processors. Then we find that using a polynomial number of processors in poly logarithmic time, you will be able to execute this.

(Refer Slide Time: 32:21)



The language $L_1 \in NC$

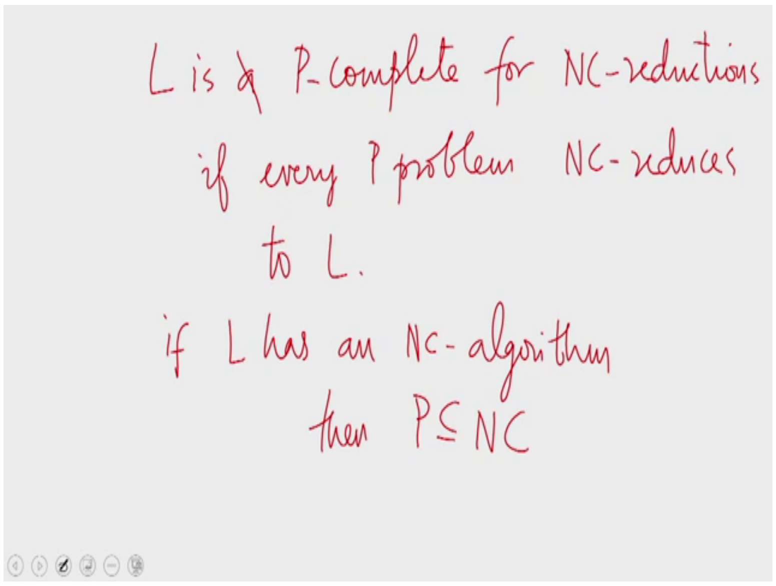
NC-reducibility is transitive

$$L_1 \leq_{NC} L_2 \leq_{NC} L_3$$

$$L_1 \leq_{NC} L_3$$

In other words, the language L 1 belongs to NC; it is also obvious that NC reducibility is transitive. If L 1 NC reduces to L 2 and L 2 NC reduces to L 3 by juxtaposing the two reducers we find that L 1 NC reduces to L 3. So, NC reducibility is transitive as well.

(Refer Slide Time: 32:55)



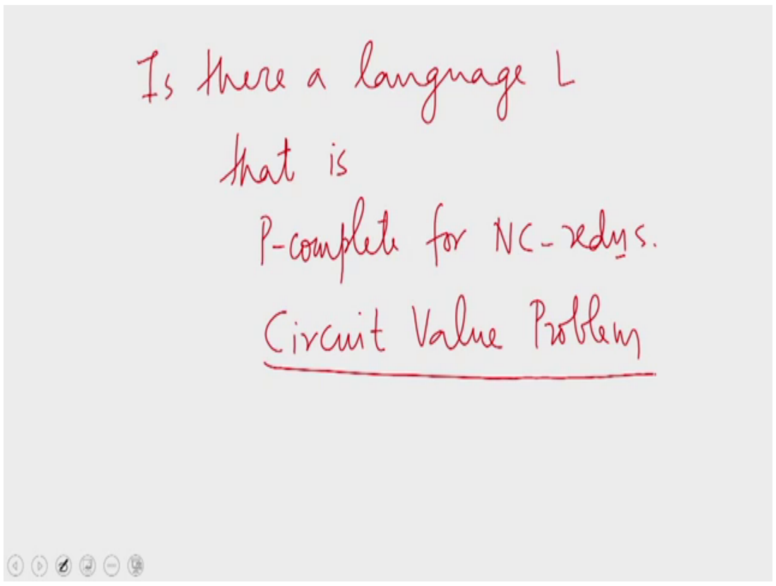
L is ~~a~~ P-complete for NC-reductions if every P problem NC-reduces to L.

if L has an NC-algorithm then $P \subseteq NC$

Now, we say that L is P - complete for NC reductions; if every P problem, NC reduces to L. If L has an NC algorithm that is if L belongs to NC, then P is a subset of NC that is because every P problem NC reduces to L; and if L had an NC algorithm, then every problem in P would have an NC algorithm.

So, to reduce every P problem to NC to prove that a problem in P is parallelizable, all we have to do is to crack one of the P complete problems in NC. Once we do that every P problem has a parallel solution, but then these definitions are all fine, but just because we define a term does not mean that it applies to any actual object that is we do not know; whether there exist an algorithm there exist a language which us P complete for NC reductions yet.

(Refer Slide Time: 34:10)



Is there a language L

that is

P-complete for NC-redns.

Circuit Value Problem

So, there is the next question. Is there a language L that is P - complete for NC reductions? We shall show that the circuit value problem is P - complete for NC reductions and once we show that one language is P - complete for NC reductions, we can show that several other languages are P - complete for NC reductions as well by reduction to this language.

So, this language we shall define in the next class and show that is P - complete for NC reductions. And then we shall show several other languages to be P - complete for NC reductions by establishing NC reduction from the circuit value problem to those, so that is it from this lecture today. Hope to see you in the next.

Thank you.