

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

**Lecture – 21**  
**Sorting Lower bound; Connected Components**

Welcome to the 21st lecture of the MOOC on Parallel Algorithms. In the previous lecture we were proving a lower bound for sorting on the parallel comparison model; in particular we had the expression  $c(t, n)$  which represents the number of comparisons that you must necessarily perform if you try to sort  $n$  elements in  $t$  steps on the parallel comparison model. So, we were seeking to prove a bound for  $c(t, n)$ .

(Refer Slide Time: 00:59)

$$c(t, n) \geq \frac{t n^{1+1/t}}{e} - nt$$

Basis  $c(1, n) \checkmark$   
 $c(t, 1) \checkmark$

$c(T, N)$   $\forall n, t \quad n \leq N \ \& \ t < T$   
or  $n < N \ \& \ t \leq T$   
the claim held

In particular we wanted to show that  $c(t, n)$  is greater than or equal to  $t n^{1+1/t}$  by  $t$  divided by  $e$  minus  $n t$ . We proved the basis which involved showing that the claim holds for all ordered pairs of the form  $c(1, n)$  and all ordered pairs of the form  $c(t, 1)$ , this we had done in the last lecture. Now we were considering the expression  $C(T, N)$  where  $T$  and  $N$  are specific values of small  $t$  and small  $n$ .

What we assumed is that by the way of hypothesis the statement held for all values of small  $n$  and small  $t$  that are less than capital  $N$  and capital  $T$  respectively. For example, we assumed that for all  $n, t$  such that  $n$  is less than or equal to capital  $N$  and  $t$  is less than capital  $T$  or  $n$  is less than capital  $N$  and  $t$  is less than or equal to capital  $T$  the claim held.

(Refer Slide Time: 02:35)

$$\begin{aligned}
 c(T, N) &\geq c(T, N-x) + \\
 &\quad (N-x) + c(T-1, x) \\
 &\geq \frac{T(N-x)^{1+\frac{1}{T}}}{e} - (N-x)T + (N-x) \\
 &\quad + \frac{(T-1)x^{1+\frac{1}{T-1}}}{e} - x(T-1)
 \end{aligned}$$

And then we showed that  $C T$  of  $N$  is given by this recurrence relation. So, once we have this recurrence relation we can use the induction hypothesis and substitute the claimed quantity for each of the  $c$  terms on the right hand side therefore, we can say that this is greater than or equal to and then substituting for the last term we have  $T$  minus 1 into  $x$  power 1 plus 1 divided by  $T$  minus 1.

(Refer Slide Time: 04:11)

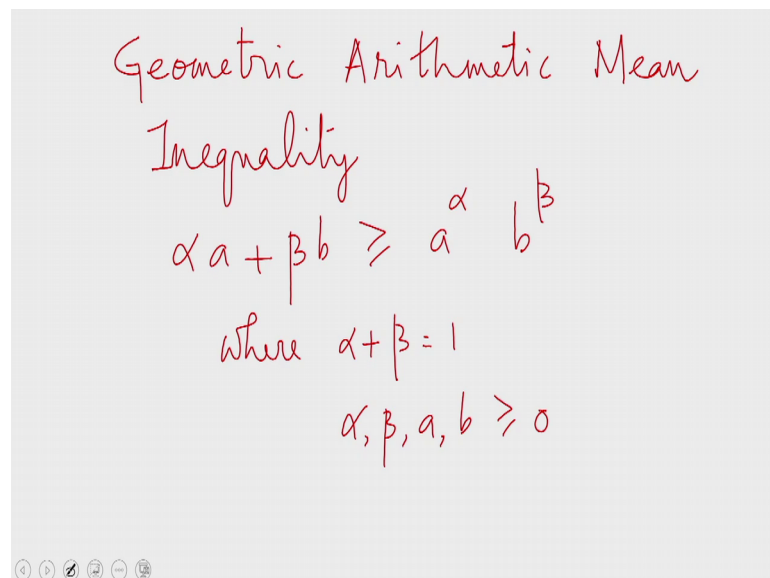
$$\begin{aligned}
 &= \frac{T}{e} N^{1+\frac{1}{T}} \left[ \left(1 - \frac{x}{N}\right)^{1+\frac{1}{T}} + \left(1 - \frac{1}{T}\right) \frac{x^{1+\frac{1}{T-1}}}{N^{1+\frac{1}{T}}} \right] \\
 &\quad + \cancel{T x} + N - \cancel{x} - \cancel{T x} + \cancel{x} - NT \\
 &= \frac{T N^{1+\frac{1}{T}}}{e} \left[ \left(1 - \frac{x}{N}\right)^{1+\frac{1}{T}} + \left(1 - \frac{1}{T}\right) \frac{x^{1+\frac{1}{T-1}}}{N^{1+\frac{1}{T}}} + \frac{e}{T N^{1/T}} \right] - NT \\
 &\quad \text{if } [-] \geq 1 \text{ then we are done}
 \end{aligned}$$

So, that is the expression we get; if you gather some of the terms we can simplify this as;

you would call that  $e$  here is the base of the natural logarithm, but here these cancel out and we are left with  $N$ . So, if you take  $N$  inside the square bracket we will have; and when  $N$  goes into the square bracket we have and of course, minus  $NT$ .

So, we know that  $CT$  of  $N$  is greater than or equal to this. Now this is almost in the desired form except for the contents in the square bracket here, that is we want to show that  $CT$  of  $N$  is greater than or equal to  $T$  times  $N$  power  $1 + \frac{1}{T}$  by  $T$  by  $e$  minus  $N T$ . Instead we have shown that it is  $T$  times  $N$  power  $1 + \frac{1}{T}$  minus  $e$  into this contents in the square bracket minus  $NT$ ; so, if the contents in the square bracket turn out to be greater than or equal to 1 then we are done. So let us focus on the contents in the square bracket and then show that that will be greater than or equal to 1.

(Refer Slide Time: 07:13)



Geometric Arithmetic Mean  
Inequality  
 $\alpha a + \beta b \geq a^\alpha b^\beta$   
where  $\alpha + \beta = 1$   
 $\alpha, \beta, a, b \geq 0$

For this we use geometric arithmetic mean inequality, which says that  $\alpha a + \beta b$  is greater than or equal to  $a$  power  $\alpha$  times  $b$  power  $\beta$  where  $\alpha + \beta = 1$  and  $\alpha, \beta, a, b$  are all non negative.

(Refer Slide Time: 07:59)

$$\alpha = 1 - \frac{1}{T} \quad \beta = \frac{1}{T}$$

$$a = \frac{x^{1 + \frac{1}{T}(T-1)}}{N^{1 + \frac{1}{T}}} \quad b = \frac{e}{N^{\frac{1}{T}}}$$

$$\text{2nd + 3rd} \geq \left( \frac{x^{T/(T-1)}}{N^{1 + \frac{1}{T}}} \right)^{1 - \frac{1}{T}} \left( \frac{e}{N^{\frac{1}{T}}} \right)^{\frac{1}{T}}$$

So, let us use this and take alpha equals 1 minus 1 by T and beta equals 1 by T. And a as x power 1 plus 1 divided by t minus 1 divided by N power 1 plus 1 by T and b as e by N power 1 by T. So, we are applying this to the second term and the third term within the square bracket.

So, if you apply the geometric arithmetic mean inequality to the second and the third term you find that, these 2 terms together this greater than or equal to x power 1 plus 1 by t minus 1 which is T by T minus 1 this power that is a power alpha and then b is n e by N power 1 by T that power beta which is 1 by T. 1 minus 1 by T is T minus 1 by T therefore, in the exponent of x the 2 will cancel out that is T by T minus 1 and T minus 1 by t will cancel out and we will be left with x.



(Refer Slide Time: 09:39)

$$= \frac{x}{N^{1-1/T^2}} \cdot \frac{e^{1/T}}{N^{1/T^2}} = \frac{x e^{1/T}}{N^{(1+1/T)}} N$$

the increasing sequence  $(1 + \frac{1}{t})^t$   
tends to  $e$  as  $t \rightarrow \infty$   
 $(1 + \frac{1}{T})^T < e \Rightarrow 1 + \frac{1}{T} < e^{1/T}$

Therefore this is nothing, but  $x$  divided by  $n$  power  $1$  minus  $1$  by  $T$  square that is a power  $\alpha$  and  $b$  power  $\alpha$  is this. So, simplifying we get the sum of the second term and the third term put together will be at least  $x e$  power  $1$  by  $T$  divided by  $N$ . But then we know that the increasing sequence  $1$  plus  $1$  by  $t$  power  $t$  tends to  $e$  as  $t$  tends to infinity.

Which means the sequence converges to  $e$  from below. In particular if you take a particular value of  $t$  you have  $1$  plus  $1$  by capital  $T$  power capital  $T$  is less than  $e$  or in other words  $1$  plus  $1$  by  $t$  is less than  $e$  power  $1$  by  $T$ . This enables us to say that this is at most  $x$  times  $1$  plus  $1$  by  $T$  by  $N$ .

(Refer Slide Time: 11:29)

$$\left[ \dots \right] \geq \left(1 - \frac{x}{N}\right)^{1 + \frac{1}{T}} + \frac{x}{N} \left(1 + \frac{1}{T}\right)$$

Bernoulli's inequality

$$(1 - \alpha)^t \geq 1 - \alpha t \quad \forall t \geq 1$$
$$\forall \alpha \leq 1$$
$$\left[ \dots \right] \geq 1 - \left(1 + \frac{1}{T}\right) \frac{x}{N} + \left(1 + \frac{1}{T}\right) \frac{x}{N} = 1$$

In other words the content within the square bracket is greater than or equal to 1 minus x by N the power of 1 plus 1 by T which is the first term plus x by N into 1 plus 1 by T which is the second term. Now to the first term let us apply Bernoulli's inequality, which says that for any alpha less than or equal to 1 and for any t greater than or equal to 1, 1 minus alpha power t is greater than or equal to 1 minus alpha t that establishes. That the quantity within the square bracket is greater than or equal to 1 minus 1 plus 1 by t into x by n plus 1 plus 1 by t into x by N which is equal to 1.

(Refer Slide Time: 12:53)

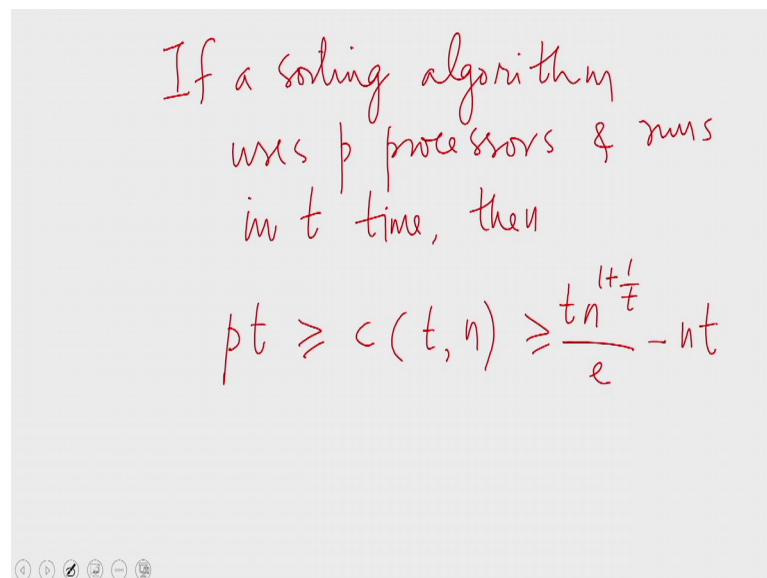
$$\left[ \dots \right] \geq 1$$
$$c(T, N) \geq \frac{T N^{1 + \frac{1}{T}}}{e} - NT$$

---

So, that establishes that the contents within the square bracket is greater than or equal to  $1 + c T$  of  $N$  is greater than or equal to  $T$  into  $N$  power  $1 + 1$  by  $T$  by  $e$  minus  $NT$ . So, that establishes the claim now we say that from this claim we can derive a lower bound for sorting  $n$  items using  $p$  processor why is that let us say we have an algorithm that sorts  $n$  items using  $p$  processors then the cost of this algorithm is  $p$  into  $t$  the processor time product.

But then this parallel algorithm would have performed at most  $p t$  comparisons, which means at least  $c t$  of  $n$  comparisons which means  $p t$  is greater than or equal to  $c t$  of  $n$ ; the algorithm must necessarily perform  $c t$  of  $n$  comparisons and it has run in a total cost of  $p t$  therefore,  $p t$  has to be greater than or equal to  $c t$  of  $n$ .

(Refer Slide Time: 13:57)



If a sorting algorithm uses  $p$  processors & runs in  $t$  time, then

$$p t \geq c(t, n) \geq \frac{t n^{1+\frac{1}{t}}}{e} - n t$$

In other words if a sorting algorithm uses  $p$  processors and runs in  $t$  time, then the cost of the algorithm  $p t$  is greater than or equal to  $c t$  of  $n$  which is the smallest number of comparisons the algorithm has to perform if it has to run in  $t$  steps and yet sort  $n$  elements. So,  $p t$  is greater than or equal to  $c t$  of  $n$ . But then we have already claimed that  $c t$  of  $n$  is greater than or equal to  $t n$  power  $1 + 1$  by  $t$  by  $e$  minus  $nt$ .

(Refer Slide Time: 14:59)

$$\begin{aligned}pt &\geq \frac{t n^{1+\frac{1}{t}}}{e} - nt \\p &\geq \frac{n^{1+\frac{1}{t}}}{e} - n \\p/n+1 &\geq \frac{n^{1/t}}{e} \\n^{1/t} &\leq e(p/n+1)\end{aligned}$$

So, that establishes that  $pt$  is greater than or equal to  $t n^{1+\frac{1}{t}}$  by  $e$  minus  $nt$  dividing both sides by  $t$  we have  $p$  greater than or equal to  $n^{1+\frac{1}{t}}$  by  $e$  minus  $n$  or in other words  $p$  by  $n+1$  is greater than or equal to that is dividing both sides by  $n$  and then moving the negative term to the other side we have or  $n^{1/t}$  is less than or equal to  $e$  times  $p$  by  $n+1$ .

(Refer Slide Time: 15:57)

$$\begin{aligned}\frac{1}{t} \log n &\leq \log e(p/n+1) \\ \frac{\log n}{\log(e(p/n+1))} &\leq t \\ t &= \Omega\left(\frac{\log n}{\log(p/n+1)}\right)\end{aligned}$$

If you take logarithm on both sides you have  $1$  by  $t$  times  $\log n$  less than or equal to or  $t$  is greater than or equal to  $\log n$  divided by  $\log$  of  $e$  times  $p$  by  $n$  plus  $1$ , that establishes that  $t$  is small omega of  $\log n$  by  $\log$  of  $p$  by  $n$  plus  $1$ .

(Refer Slide Time: 16:55)

$$t = \Omega\left(\log_{(p/n+1)} n\right)$$


---

A lower bound for sorting  
 $n$  items using  $p/n$  processors

$p = n$     $p/n = 1$     $p/n + 1 = 2$     $t = \Omega(\log_2 n)$

$p = n \log n$     $p/n = \log n$     $t = \Omega\left(\frac{\log n}{\log \log n}\right)$

This is the same as saying that  $t$  is omega of  $\log$  of  $n$  to the base  $p$  by  $n$  plus  $1$ . So, this is a lower bound for sorting  $n$  items using  $p$  by  $n$  processors. So, let us try substituting some particular values for  $p$ , in particular if  $p$  equal to  $n$ ;  $p$  by  $n$  equals  $1$  and  $p$  by  $n$  plus  $1$  equal to  $2$  which means the lower bound is  $\log n$  to the base  $2$ .

So, that establishes that when you have  $n$  processors, you will not be able to sort faster than  $\log n$  time. So, in that sense coles merge sort is an optimal algorithm it sorts  $n$  elements using  $n$  processors in order  $\log n$  time. You cannot expect to better coles merge sort in an asymptotic sense. If you put  $p$  equal to  $n$  by  $\log n$   $n \log n$ , then  $p$  by  $n$  is  $\log n$  therefore,  $t$  would be omega  $\log$  of  $n$  divided by double  $\log$  of  $n$ ; when you have  $n \log n$  processors you cannot expect to sort faster than  $\log n$  by double  $\log n$  on any of the  $p$  ram models.

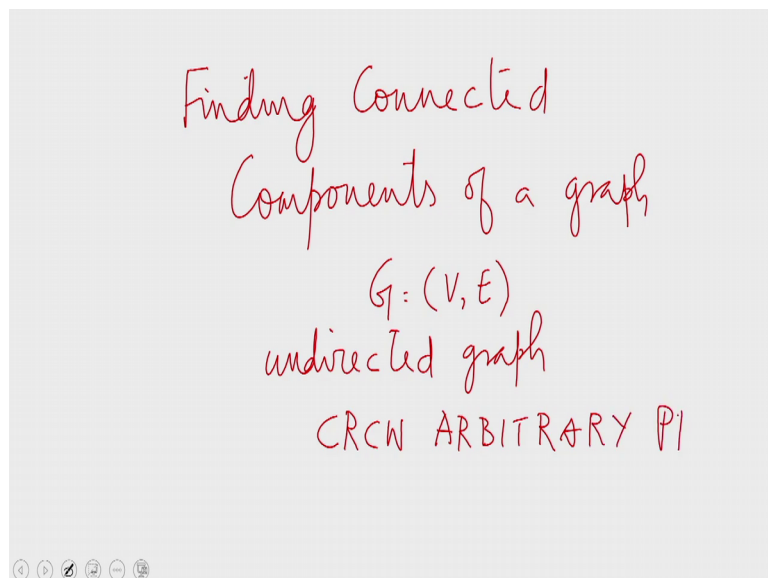
(Refer Slide Time: 19:05)



So, in particular what we have shown is that, coles merge sort is optimal.

Now, let us begin a new topic.

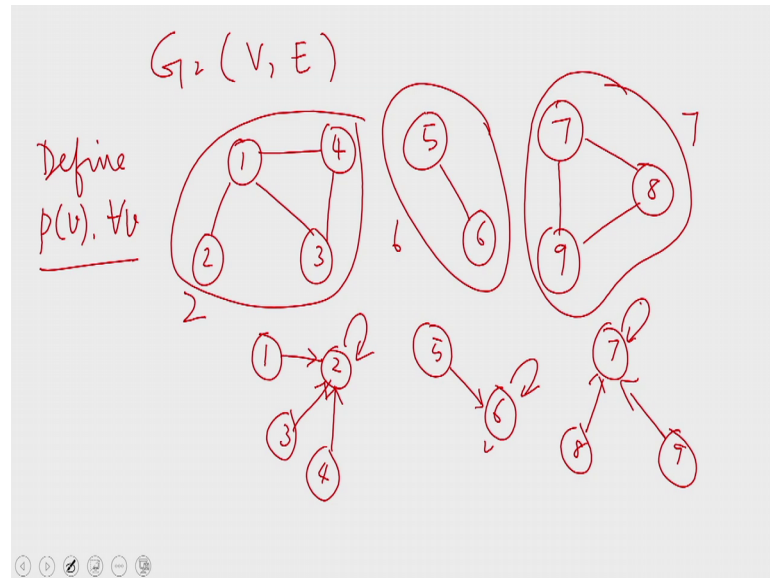
(Refer Slide Time: 19:25)



The new topic that we are going to see is the problem of finding connected components of a graph. So, we will assume an undirected and weighted undirected graph for now, and let us say the model that we are going to use is the CRCW ARBITRARY PRAM.

So, you must have all studied the connected number problem connected components problem in the sequential setting and you would have seen that this problem can be solved using d f s n b f s n 2 sequential setting.

(Refer Slide Time: 20:37)



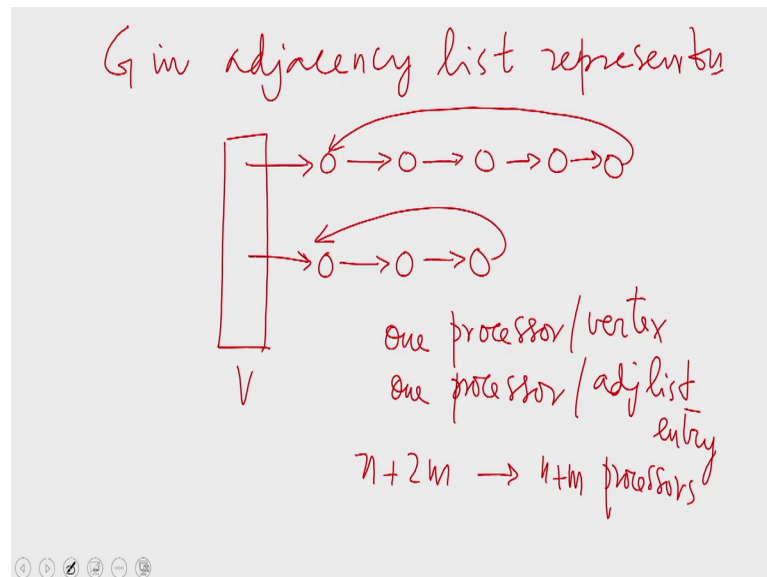
So, this is a reasonably easy problem to solve in the sequential setting in particular let me recall the problem for you. So, let us say we have given a graph  $G$  equal to  $V E$ . So, given this graph what is necessary is to define a parent pointer for each vertex, for every vertex we have to define a parent pointer.

So, that the vertex and its parent are in the same component and we will have one star for every component. In other words for the first component let us say we can have a star of this sort; what this signifies is that 1 2 3 and 4 are all in the same component. So, we have identified this component with vertex 2, that is in a sense we are identifying that vertices 1 2 3 and 4 all belong to the same component and we are choosing to name this component is 2. So, 1 3 and 4 all adopt to ask their parents and here let us say 5 points to 6 and therefore, 6 is chosen to represent this component and let us say the third component is chosen to be represented using the name of vortex 7.

So, we are going to establish 3 components in the graph with names 2 6 and 7 respectively the other vertices in the component will end up pointing to the representative vertex. So, finding of connected components in the graph is essentially defining parent pointers in the sense, every vertex ought to have a parent pointer every

vertex belonging to the same component should have the same parent and the parent of a component can point to itself therefore, every component essentially reduces to one single star graph. So, the goal is to find a star graph for every single component in the graph.

(Refer Slide Time: 23:21)

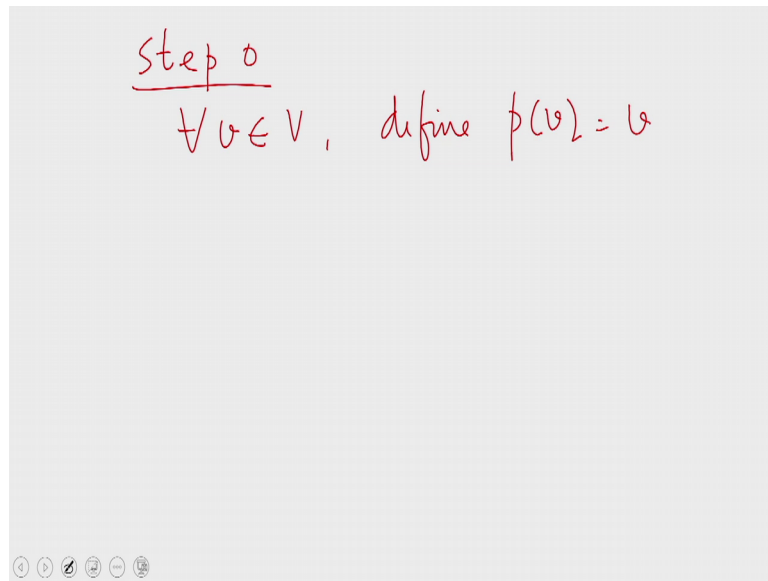


So, the algorithm goes like this let us say we are given G in adjacency list representation, which means we have an array of vertices and for each vertex we have an adjacency list, which you can assume as doubly linked. So, such a representation is the adjacency list representation of the graph. So, let us assume that we have enough processors to occupy all the data values. We assume that we have 1 processor per vertex and 1 processor per adjacency list entry.

So, this is as good as assuming you that you have  $n + 2m$  processors, but since the model is self simulating we can simplify this into assuming that we have  $n + m$  processors. So, when we say we have  $n + m$  processors that is as good as saying that we have one processor for every single data element of the graph.

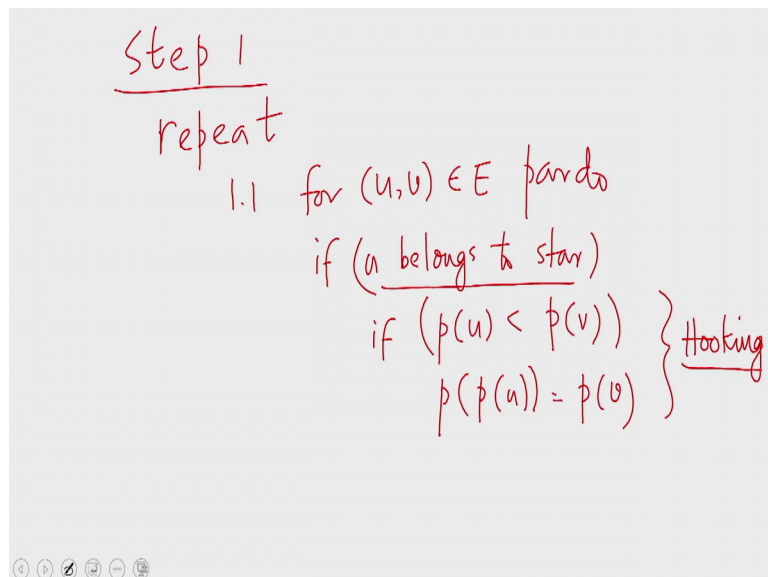


(Refer Slide Time: 24:53)



Now, the algorithm begins with initialization which we call step zero. In the zeroth step what we do is this for every  $v$  we define the parent pointer as pointing to itself the parent does one self for every vertex. So, this is the zeroth step and then in step 1 we have several iterations.

(Refer Slide Time: 25:27)

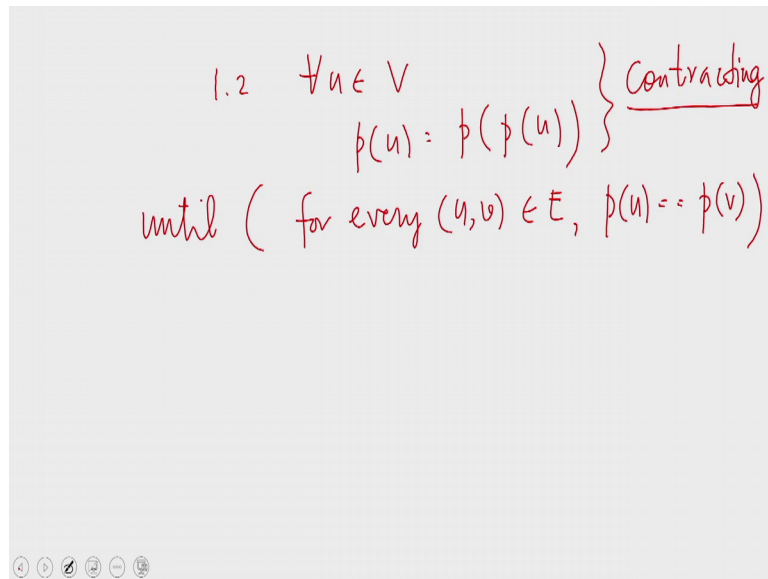


So, step 1 is essentially a loop, here we repeat the following steps; in step 1.1 for every edge  $u v$  every directed edge  $u v$  we assume that the graph consists of undirected edges, but then each undirected edge can be visualized as 2 directed edges 1 in each direction,

which means in the adjacency list we assume we have 1 entry corresponding to edge  $u \rightarrow v$  and another entry corresponding to edge  $v \rightarrow u$  which are the twins of each other.

So, for each adjacency list entry there is now I am visualizing the graph as a directed graph. So, for each directed edge  $u \rightarrow v$  belonging to  $E$  we do in parallel. If  $p$  of  $u$  less than  $p$  of  $v$   $p$  of  $u$  equal to  $p$  of  $v$ .

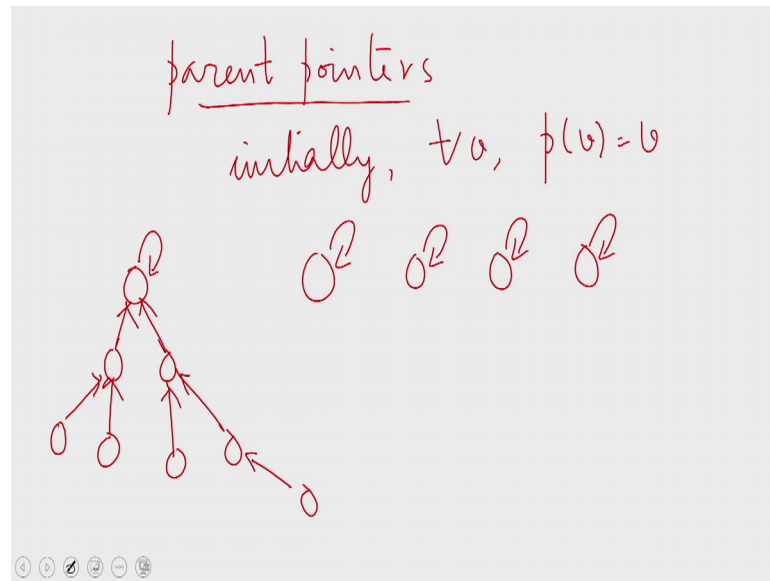
(Refer Slide Time: 27:05)



So, that is what we do in step 1.1 and then in step 1.2. For every  $u$  belonging to  $v$  we set  $p$  of  $u$  as  $p$  of  $p$  of  $v$  this is 1 step of pointer jumping exactly as we did in the case of list ranking. So, these are the 2 steps of the loop, we continue doing this until for every directed edge  $u \rightarrow v$   $p$  of  $u$  is the same as  $p$  of  $v$ .

So, that is what the algorithm is, but we have to explain some things now the parent pointers.

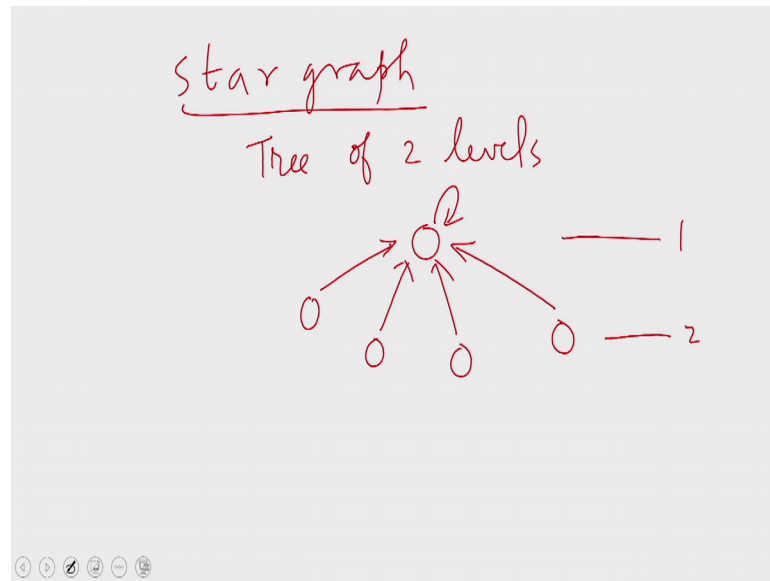
(Refer Slide Time: 27:55)



The invariant of the algorithm is that every vertex has a parent pointer. Initially for all  $v$  we set that  $p$  of  $v$  equal to  $v$ . So, initially we have a set of vertices of this form, but we shall show that as we go along, we will have the parent pointers defining a trick, a collection of trees a forest in particular. Each tree of the forest will be of this form every node in a tree will have an out degree of 1 and all nodes other than the root will be pointing to a node other than itself the root will be pointing to itself.

So, this establishes that every single node in the graph in the graph defined by the parent pointers, the out degree of every single vertex is 1 the every vertex has got a unique parent. So, this we shall establish that the parent pointers will always define a forest in this sense. Our idea is that finally, when the algorithm ends the there should be 1 forest corresponding to every single component.

(Refer Slide Time: 29:29)



What we define as a star graph is a tree of 2 levels. It has 1 root and the root has several children, all of them pointing to the root itself and then there are no grandchildren the tree has only 2 levels the root is at the first level and the children are at the second level. So, there are only 2 levels a star graph is a tree of exactly 2 levels.

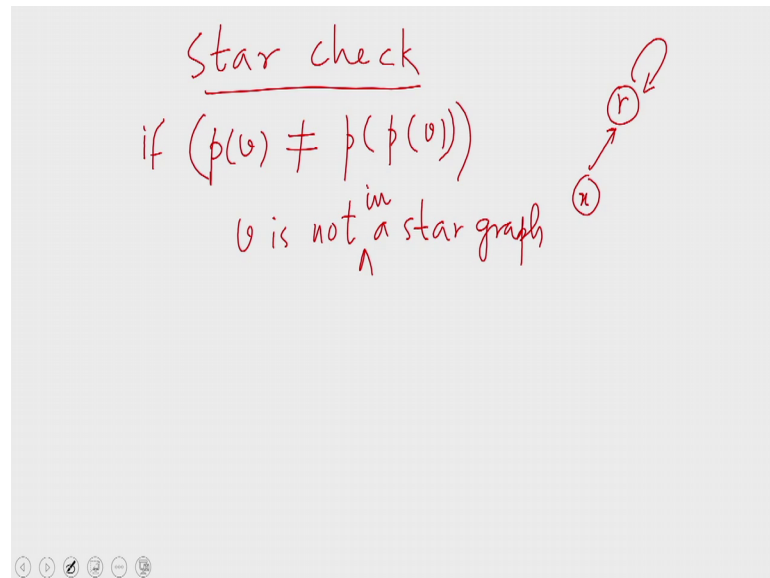
So, what we want is that when the algorithm terminates, every tree in the forest must be a star graph and there should be 1 forest 1 tree corresponding to every single connected component in the forest.

(Refer Slide Time: 30:23)

one processor / vertex  
detect if it belongs to a  
star graph  
ARBITRARY CRCW PRAM

Now, the question comes. If you have one processor per vertex, detect if it belongs to a star graph. This is a condition that we have to check within the algorithm, here we check whether  $u$  belongs to a star or not. So, we should be able to check whether a vertex  $u$  belongs to a star graph or not, but remember we are doing this on an ARBITRARY CRCW PRAM. This is done like this checking for the star property is done like this.

(Refer Slide Time: 31:19)



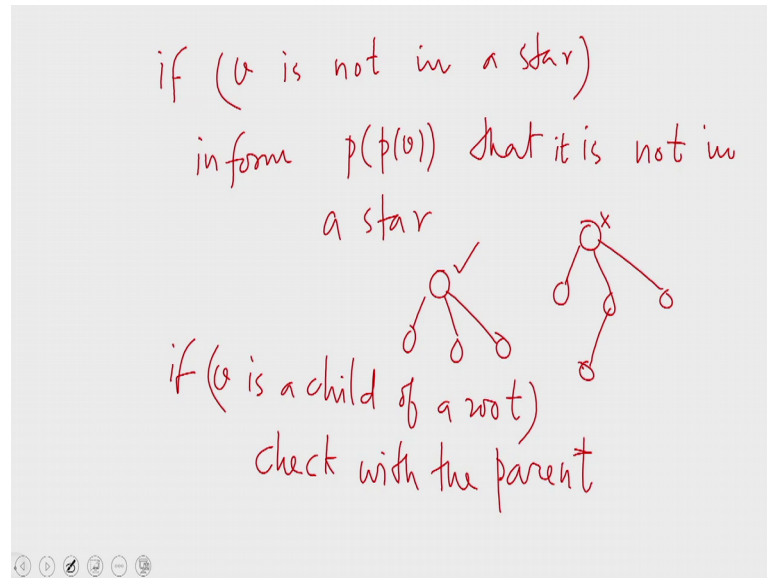
Assuming that we have one vertex one processor per vertex let every vertex in the star graph discover its parent and the grandparent.

So, for vertex  $v$  we take the parent and the grandparent. If the parent and the grandparent are not the same vertex, then  $v$  is neither the root nor a child of the root that is because the root points to itself, the parent of a root is itself now consider a child of the root. So, let  $r$  be the root and let  $x$  be this vertex now the parent of  $x$  is  $r$  and the grandparent of  $x$  is also  $r$ .

So, for every child of  $r$  the parent is  $r$  and the grandparent is also  $r$ , but here we are explicitly checking whether  $p$  of  $v$  is not the same as  $p$  of  $p$  of  $v$ . So, this condition is satisfied only if  $v$  is neither the root nor a child of the root, which means we does not belong to a star graph. So, for every vertex if this condition is satisfied,  $v$  is not in a star graph. So, it is easy to check for all vertices that are neither roots nor children of roots. So, this can be established in order of one time.

So, once this is done, every vertex other than the roots and children of roots know that they are not in star graphs. Now the roots and the children of the roots have to find out for themselves this can be done easily.

(Refer Slide Time: 33:25)



If  $v$  is not in a star then we has a grandparent which is not the same as  $v$ ; inform the grandparent of  $v$  that it is not in a star. Every  $v$  now informs its every  $v$  that has found out that it is not in a star informs its grandparent that it is not in a star too.

Now, if a tree has more than two levels, then the root certainly has grandchildren those grandchildren will now come and tell the root that it is not in a star. So, those roots which are of trees that are not stars, now know that they are not roots of stars. And by omission all the other roots now know that they are roots of stars. For example, if a node has no grandchild and in this step it was expecting a grandchild to come and inform it that it is not in a star.

But no grandchild comes, at that point that vertex knows that it does not have a grandchild which means it is a node of the star. So, all vertices all roots of stars have now identified themselves correctly, and all roots of non stars also have identified themselves correctly. And once the information is available with the parent the children can immediately see. So, in the next step every vertex  $v$  will go and check whether its parent belongs to a star or not. Now if a vertex is a child of a root it can check at the root now and discover that it belongs to the star or not.

If  $v$  is a child of a root check with the parent; so, we have a 3 step process to check whether a vertex belongs to star or not. Once these 3 steps are executed we would have informed every single vertex as to whether it belongs to a star or not. The first step applies to every vertex every vertex checks that its parent and the grandparent are different.

If these 2 are different certainly  $v$  is not in a star graph is not in a star graph. So, every vertex that is a grandchild now has found out that it is in a star graph, the other vertices are still non committal they do not know. So, if this condition does not satisfy then the vertex still does not know then it goes on to the second step and then if a vertex has found that it is not in a star graph which means if it is a grandchild.

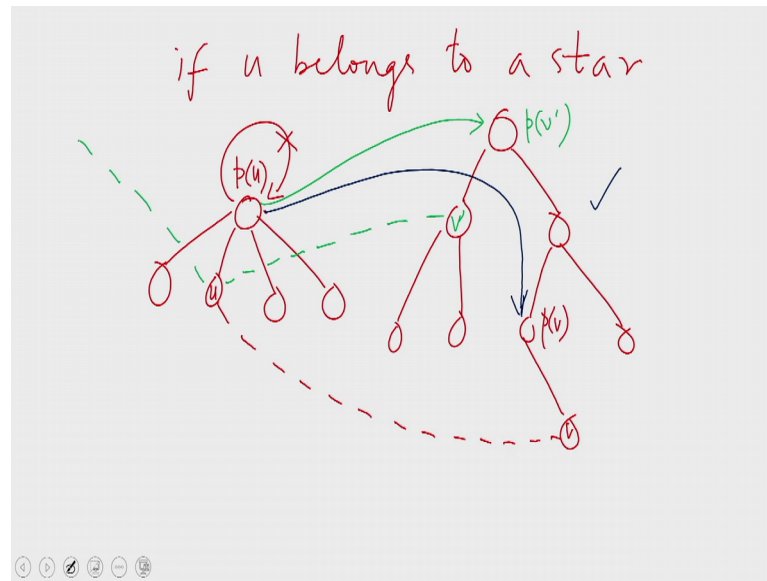
So, in the second step every grandchild is active. So, every grandchild will go and inform its grandparent that you are not in a star graph too. So, every grandparent does now be found out that it is not in a star graph. In particular every root which is a grandparent has found out that it is not in a star graph and by omission every root that is in a star graph also has found out that it is in a star graph.

Now, all that remains is to inform the vertices which are children of the roots. So, if a vertex is a child of a root that is easy to check, your parent and the grandparent are the same then you are a child of a root and your parent is different from yourself in that case you are a child of a root. Then all you have to do is to check with your parent the parent already knows, the parent has a root it already knows whether it is in a star or not. So, you only have to copy the information from the parent.

So, in 3 steps we have informed every single vertex of the graph as to whether it belongs to a star or not. Now we come to the description of the algorithm. In step 1 what we do is, this for every  $u$  which belongs to a star we perform what is called hooking. Step 1 can be called the hooking step and in step 2 we do a 1 round of pointer jumping this is called contracting.

So, the algorithm proceeds repeatedly hooking and contracting. So, what exactly do we do here?.

(Refer Slide Time: 38:28)



In step 1.1, we execute if  $u$  belongs to a star. So,  $u$  belongs to this star let us say  $u$  could be either the root or 1 of the children then. So, we are doing this for every single edge of the graph. So, consider some edge  $u-v$  of the graph where  $u$  belongs to a star. So,  $u$  may be one of these vertices let us take this as  $u$  and we are considering an edge in the graph, the dotted line represents a real edge in the graph whereas, a thick line represents a parent pointer. So, this is an edge in the graph. So, what we do is this,  $u$  belongs to a star and that is why this edge gets to execute step 1.1.

So, in step 1.1 we check whether the parent of  $u$ , this is the parent of  $u$  and this is the parent of  $v$  we check if the parent of  $v$  has a number that is smaller than the number of parent of  $u$ . If that is the case, we make the we make  $p$  of  $u$  right now  $p$  of  $u$  is pointing to itself right now  $p$  of  $u$  is pointing to itself this we remove and then make  $p$  of  $v$  the parent of  $p$  of  $u$ . So, this star ends up being a part of this tree why do we do this? At present in the algorithm we have 2 components defined by the parent pointers, that is  $u$  belongs to 1 component and  $v$  belongs to another component. Now  $u$  is in a star which means this tree cannot be contracted any further using pointer jumping.

So, this component to which  $u$  belongs to this component in the graph defined by the parent pointers,  $u$  has come to a point of stagnation. This component cannot shrink any further. But then we find that there is an edge  $u-v$  in the graph which connects this star graph this 2 level tree to another 2 level tree. Therefore, we attempt to hook these 2 trees



together that is because in the final reckoning, all these vertices should be in one single component and therefore, in one single star graph therefore, what we are attempting to do here is legitimate. But then there could be several such attempts we are doing this for every single edge  $u v$  in step 1 1 we do this for every single edge  $u v$ .

So, when we consider these 2 trees, the tree to which  $u$  belongs and the tree to which  $v$  belongs, there could be multiple edges going between these 2 trees; since the model that we are using is the c r c w arbitrary p ram.

When multiple processors attempt to hook these 2 trees together exactly one of them will succeed. That is multiple processors might be attempting to reset the value of  $p$  of  $p$  of  $u$ , but exactly one of them will succeed. For example, it could be that there is another vertex  $v$  prime, which does the same thing;  $v$  prime attempts to hook  $p$  of  $u$  to  $p$  of  $v$  prime that is because we have an edge  $u v$  prime 2 in the graph.

So, there is one processor sitting on  $u v$  prime and another processor sitting on  $u v$  both the processors recognize that  $p$  of  $u$  is less than  $p$  of  $v$  and  $p$  of  $v$  prime therefore, both the hooking are valid. Therefore, the first processor will attempt to reset  $p$  of  $u$  as  $p$  of  $v$  whereas, the second processor will attempt to reset  $p$  of  $u$  as  $p$  of  $v$  prime exactly one of these attempts will succeed on the arbitrary model.

It does not really matter which one succeeds, either way we will be hooking this star to another tree. Now there could be another edge going out of  $u$  into some other component into some other tree which will attempt to hook the star to that particular tree. So, once again it does not really matter which of these attempts succeed whichever succeeds will ensure that the star graph is hooked to one of the existing trees. So, this is a way of coalescing multiple trees into one single tree.

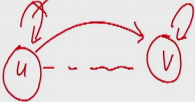
And then after doing this, we will reduce the height of the trees by one step of pointer jumping. That is every node will now adopt the grandparent of that node as its parent. When you apply pointer jumping to a star graph nothing changes, the star graph continues to remain the same star graph. Therefore, you can perform a pointer jumping for every single vertex without concern about the star graphs ruining themselves.

(Refer Slide Time: 43:47)

Correctness

No cycle is formed  
in the hooking step.

$\forall v \quad p(v) \leq v$        $p(v) = v$  initially



The diagram shows two vertices, u and v, represented as circles. Vertex u has a self-loop arrow above it and a dashed arrow pointing to vertex v. Vertex v has a self-loop arrow above it. This represents the hooking step where u is hooked to v.

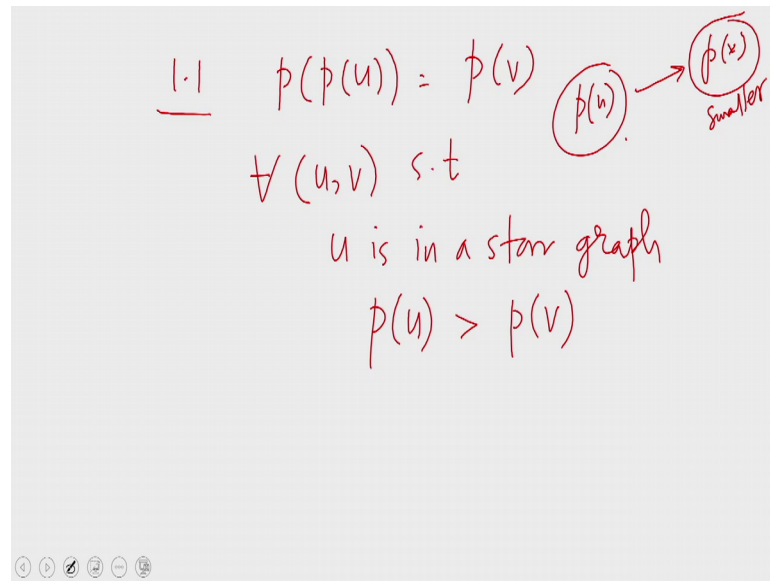
So, that is what the algorithm is. So, the correctness of the algorithm is easy to see. The correctness depends on the fact that no cycle is formed in the hooking step. This is because for all  $v$   $p$  of  $v$  is always less than  $v$  or less than or equal to  $v$ . Initially  $p$  of  $v$  is the same as  $v$  and whenever hooking happens in the first step we will be hooking a vertex to a smaller vertex.

For example let us say you have an edge from  $u$  to  $v$  real edge in the graph, the processor sitting on this edge looks at  $p$  of  $u$  and  $p$  of  $v$  it finds that  $p$  of  $u$  is larger than  $p$  of  $v$  and therefore, it ends up hooking  $u$  to  $v$ . So, now, this parent pointer that is defined is from a larger numbered vertex to a smaller numbered vertex and this is true for every single pointer that is defined in the first step.

Therefore the parent pointer graph that we get after the first step is 1 where every vertex is pointing to a smaller numbered vertex, then no cycle can be formed. And the pointer jumping will not violate this condition every node is adopting its grandparent as its parent. The parent is smaller than  $v$  and the grandparent is smaller than the parent therefore, the grandparent is smaller than the vertex itself therefore, pointer jumping will not violate this condition.

So, this condition is satisfied at the end of the first iteration and then it will be satisfied in the further iterations.

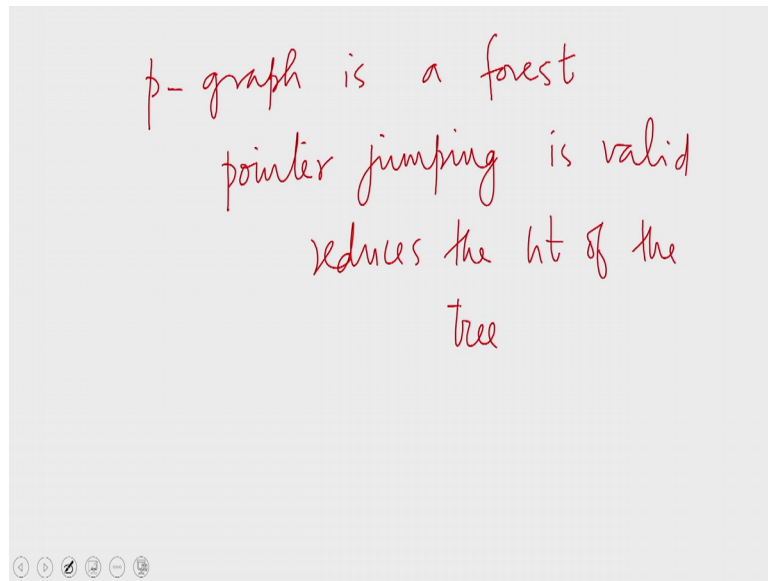
(Refer Slide Time: 46:01)



For example, in step 1.1 what we do is this, we set  $p$  of  $p$  of  $u$  as  $p$  of  $v$  for all  $u, v$  such that  $u$  is in a star graph and  $p$  of  $u$  is greater than  $p$  of  $v$ .

Now, the new pointer being defined here is this, the pointer is from  $p$  of  $u$  to  $p$  of  $v$ . So, we explicitly check that  $p$  of  $u$  is greater than  $p$  of  $v$  therefore, the new pointer that is being defined is from a larger node to a smaller node. Therefore, this condition is maintained by every single pointer that is defined in step 1.1 whenever you change the parent, you ensure that the new parent is smaller than the vertex itself. So, this invariant is always maintained therefore, the  $p$  graph the graph defined by the  $p$  pointers is always a forest therefore, pointer jumping defined on this is a valid operation.

(Refer Slide Time: 47:13)



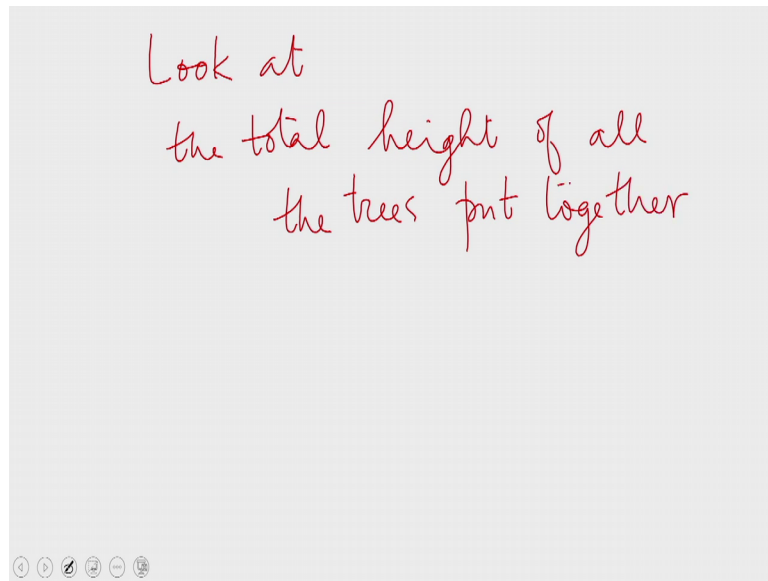
It reduces the height of the tree and when the tree becomes a star graph finally, it will hook once again.

So, the correctness of the algorithm follows from this, that is if you keep doing this eventually all the fragmented trees that we had from 1 single component at the end of the first iteration will end up hooking to each other and eventually all of them will contract into 1 single star graph. So, if you imagine how the 1 single component operates you will get to see that the algorithm operates correctly.

Initially every vertex is pointing to itself; then at the end of the first step the vertices are hooking up somehow, but every hookup is guided by 1 single edge therefore, every vertex is hooking to some of its neighbors. But then if you look at one particular component, you find that it might have several trees within this component, then these trees coalesce as steps go by whenever a tree shrinks to a star graph it will hook again.

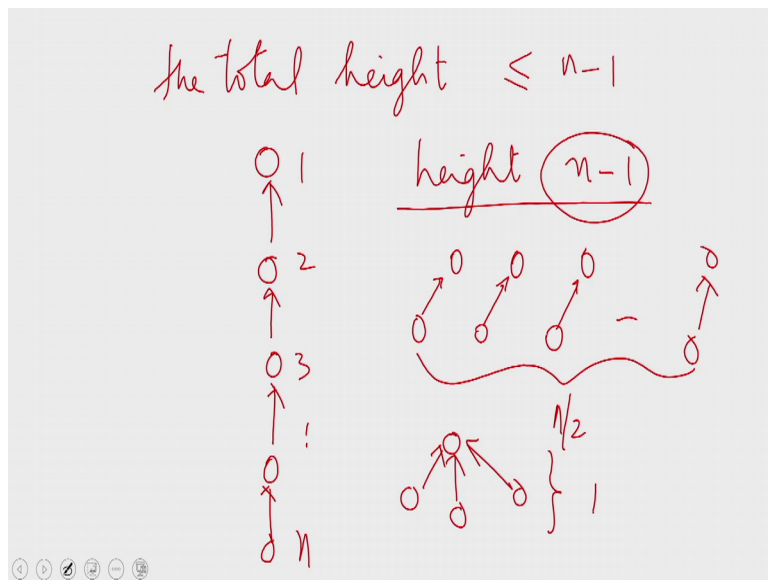
But the hooking again will be directed by one real edge of the graph. Therefore, it will be hooking to another tree belonging to the same component. So, trees are now coalescing in this fashion and then once the trees coalesce we allow them to shortcut. So, we will continue this operation until every vertex has belong to a star graph now we have to estimate the running time.

(Refer Slide Time: 49:13)



The running time can be estimated by looking at the total height of all the trees put together.

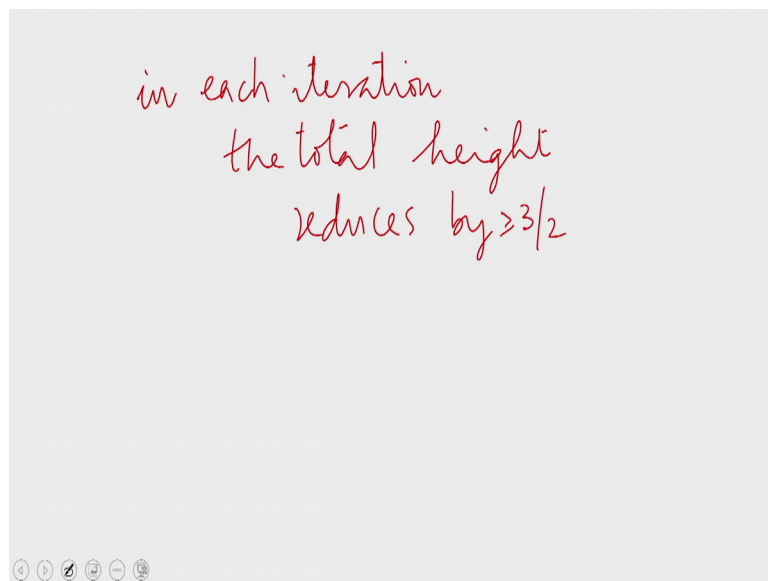
(Refer Slide Time: 49:47)



Initially when the vertices hook to form a collection of trees, the total height is less than or equal to  $n-1$ . This is because the worst that can happen is trees hooking up in this fashion.

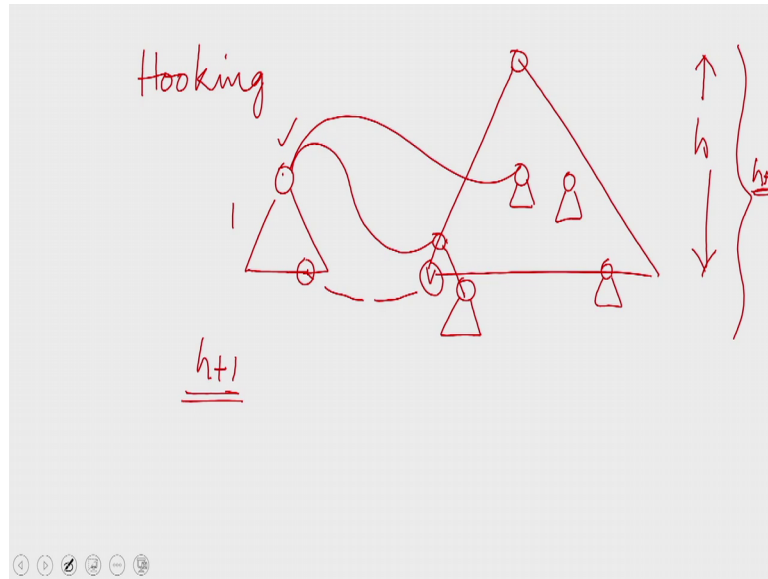
If the tree if the nodes hook up in this fashion, we have a tree of a single tree of height  $n$  minus 1 therefore, the total height of all the trees put together is  $n$  minus 1 itself. The degenerate cases where the nodes pair off in this fashion, in this case the total height of all the trees put together is  $n$  by 2. So, either way we can see that the total height of all the trees put together is even more degenerate is a single star graph, it could be that all vertices belong to the same component and one single star graph is formed in 1 go in the beginning.

(Refer Slide Time: 51:07)



Therefore, we can say that the total weight of all the trees put together that are formed in the beginning is  $n$  minus 1 and then we claim that in each iteration the total height reduces by 3 by 2 by at least 3 by 2 by a fraction which is at least 3 by 2.

(Refer Slide Time: 51:37)



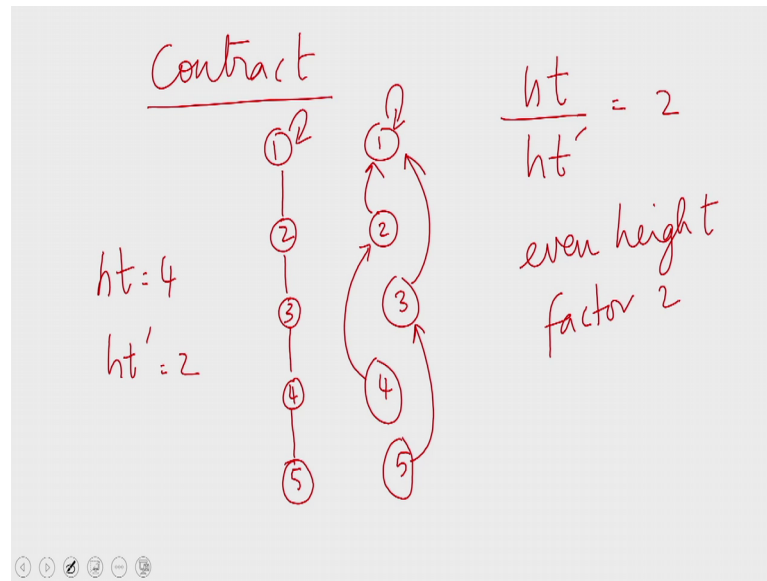
Why is this? First consider the hooking step; in the hooking step we have a star graph of height 1 hooking to a tree of let us say height  $h$ . The total height of these 2 trees put together is  $h$  plus 1 in the beginning before the hooking. Now let us say we are considering some  $u$  here  $u$  could be either the root or a leaf here, and we consider a  $v$  which happens to be a leaf of this tree.

Therefore the hooking happens this way, the parent of  $u$  is hooking to the parent of  $v$ . So, this tree the tree of height 1 ends up hanging as a child of the parent of  $v$ . So, this is how the tree is going to be hung therefore, the total height of the new tree is going to be  $h$  plus 1. Now this tree has vanished it has become a part of the bigger tree which has a height of  $h$  plus 1 therefore, hooking preserves the total height in this case.

But in case the tree is hooking to some other node inside further inside this bigger tree, then the height of the bigger tree will not increase because of this hooking. And it could be that several stars are hooking to this 1 tree therefore, we can say that hooking will not increase the total height of all the trees put together it might decrease the total weight of all the trees put together.

But it can never increase the total weight of all the trees put together; therefore; the total height in the worst case remains the same when we hook.

(Refer Slide Time: 53:25)

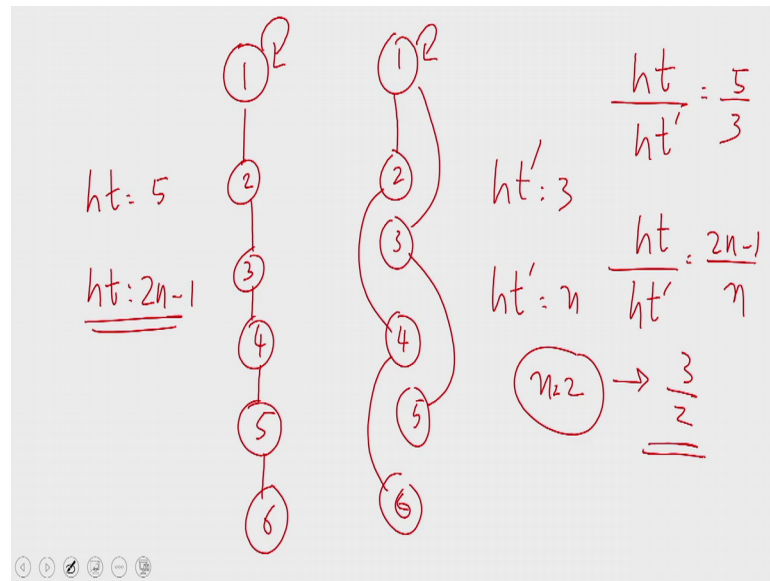


When we come to the contract step, this is a pointer jumping case. So, let us consider the individual trees of course, since we are considering pointer jumping we might as well consider linked lists. So, in this case when you have a linked list of 5 nodes for example, there are 5 nodes therefore, the height is 4 when we perform pointer jumping 1 is pointing to itself which is the root.

So, 1 will continue to point to itself 2 will continue to point to 1, 4 will point to 2, now 4 has a depth of 2 3 will point to 1 and 5 will point to 3. So, you find that the height has now reduced to 2. So, if you take the ratio of the old height to the new height, you find that this 2 which means in the contract step whenever we are considering a tree of even height, the height reduces by a factor of 2. For all trees of even height the height reduces by a factor of 2.



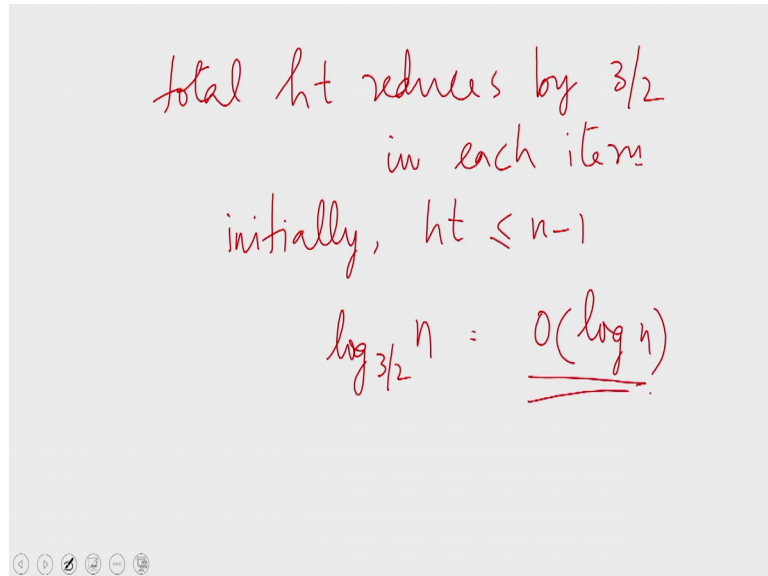
(Refer Slide Time: 54:51)



If you consider a tree of odd height, where 1 points to itself you find that the point is flow like this 6 becomes a child of 4, 4 becomes a child of 2, 5 becomes a child of 3, which becomes a child of one.1.

So, here we had a height of 5 whereas, here we have a height of 3 the new height is 3. So, the old height divided by the new height in this case is 5 by 3. In general if you take a tree of height  $2n - 1$ , the new height is going to be  $n$  or in other words the ratio is going to be  $2n - 1$  divided by  $n$ . This is maximized when  $n$  equal to 2. When  $n$  equal to 2 the ratio becomes  $3$  by  $2$  for this is the minimum the ratio the least ratio is  $3$  by  $2$  for any other value of  $n$  we have a better ratio.

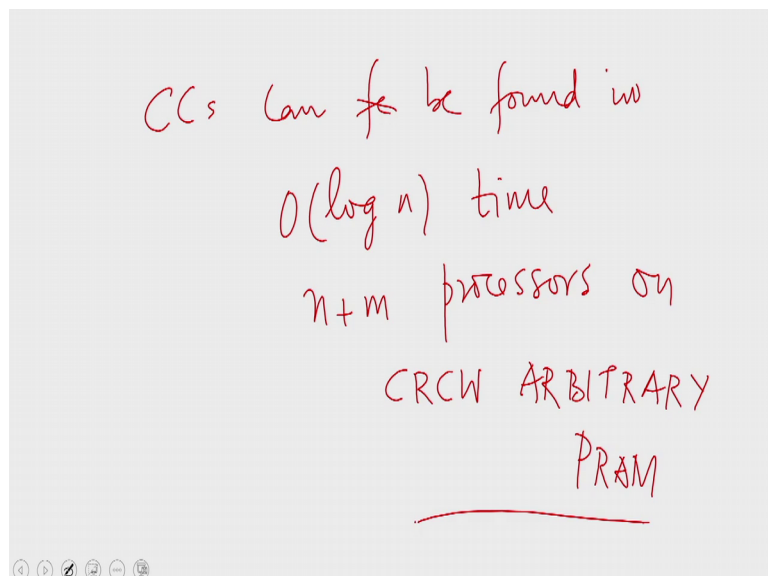
(Refer Slide Time: 56:31)



So, for the worst that can happen is that the total height reduces by a factor of 3 by 2 in every iteration. The total height that we are beginning with of less than or equal to  $n$  minus 1 and finally, we will have a height of 1 for every single star graph, that is we will have 1 star for each component therefore, the total height for each component is going to be at most 1.

So, this establishes that the total time required to reduce the height were constant or to the least possible is  $\log n$  to the base 3 by 2, which is order of  $\log n$ .

(Refer Slide Time: 57:31)



So, what we have established is that, connected components can be found in order of  $\log n$  time using  $n$  plus  $m$  processors, where  $m$  is the number of edges and  $v$   $n$  is the number of vertices on a CRCW ARBITRARY PRAM. In the next lecture we will try to solve the same problem on a CRCW PRAM. So, that is it from this lecture hope to see you in the next lecture.

Thank you.