Parallel Computing

Prof. Subodh Kumar
Department of Computer Science & Engineering
Indian Institute of Technology - Delhi

Module No # 07
Lecture No # 33
Lower Bounds Locks Free Synchronization, Load Stealing

Alright, let us continue over the question we were looking at, how long does it take right the time and I will leave the work for you to figure out. Once you know the time you know resolve the order and things active at a stage you can figure out the work. How long does it take and our world was long and hopefully it takes long hand. But why?

Each stage is taking longer time, yes its merge is taking order one time right, and what is the total number of stages? And each stage is getting done not at one time right, all the active notes are doing their own thing in parallel. What is the period starting at equal to zero when every note has become active and inactive again, at which point will you be done. Thrice of login right, after three times the height of the root, root has merged everything.

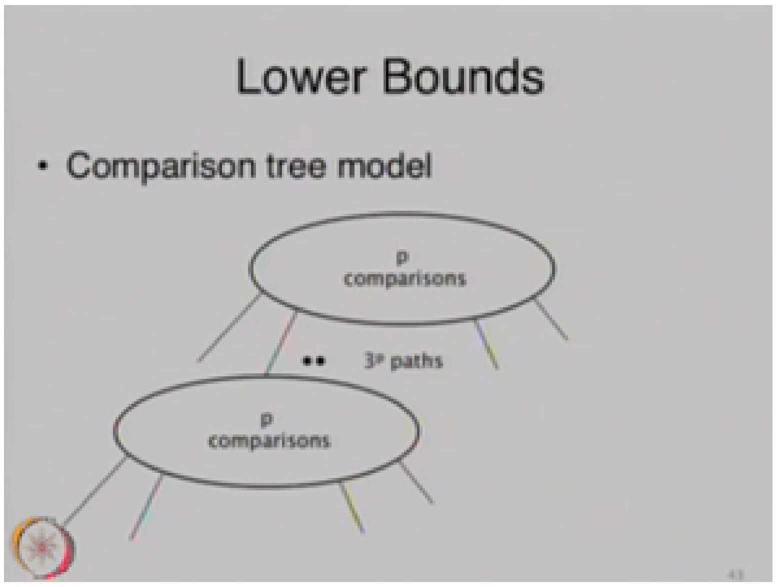
Right, so login time you are done. And login is going to be the work because we just one like I said. Only order and, because any time only order and things are being handled. I have not proved it, whether it is a login but you can go and reason that out it does not increase the work, it just makes things faster. Okay, alright now the next question that you may of course want to ask is can you do better. Why just logon, what is the lower bound.

If I said or sequential algorithm it is a logon, we know how to prove it, but it is. why give it an end process, that is the other question right, I want, if it is in peek then I should be able to type all serializes right I am going to able to type polynomial number of process and do it in poly log time. Of course, in this case we have already reached the log so that theory says that is good enough.

But how do you prove lower bounds in general, you basically launch the same kind of theory that you did for sequential algorithm to the parallel context, you say instead of one comparison at every note in a comparison tree. We are now allowed to make P comparison, where P is the number of processes, the level of parallelism in our system. And so instead of three options three possible parts to take from every note.

You are stick to the three constant, three meaning less than each comparison can say less than equal to or greater than and they are pieces comparisons you are making. So T to the different P options. Different possible answers and for each of those possible answers it would take one choice and we are assuming here that figuring out which of those possible answers is Constantine.

(Refer Slide Time: 04:25)



Lower Bounds

- Comparison tree model

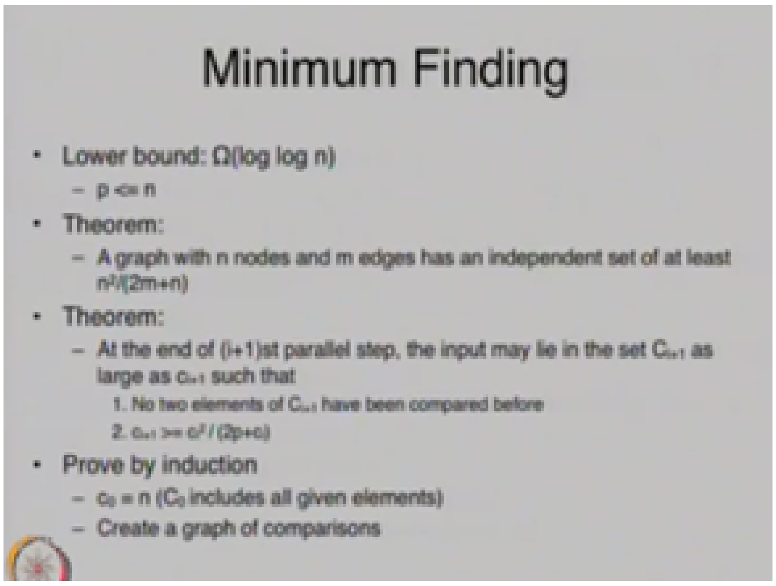
And then again the nod you get to, you have three more comparisons to make, take one of the choices and you figure out at the end of the day how many different leads you have and how many notes that be taken instead of binary tree, you have tree to the P earned or instead of a tertiary. Tree to the P earned, that is all, so P is going to then start to pay roll, and we had to say that if I had so many processes.

I will be able to do so many processes and it turns out that and it gets quite interesting and even convoluted when you start to talk about bigger than N processes and sorting. How does it, how long does it take and that line of research is mostly just academic but has been done at least from the past. Alright, so but let us at least consider when we have N processes and even that may be a little bit high.

So for Logon it is clear you can do no better than for sorting it is clear do know better than logon. What about finding the M, U and M, N or fuel. Yes, **yes** so that automatically establishes the lower logon, it could not do anything better than logon. Because if you could then you would take it to. In other cases, so for example we know umm an algorithm and minimal finding.

They say it is a really fast minimal finding that you can do in order one time. End square processes but if you brought it down to N processes you mean N work we say it is going to take log, logon time, could you do better, could you still say order in work, order one time and that be done. You can ask some questions of that sort.

(Refer Slide Time: 06:56)



Minimum Finding

- Lower bound: $\Omega(\log \log n)$
  - $p <= n$
- Theorem:
  - A graph with n nodes and m edges has an independent set of at least $n^2/(2m+n)$
- Theorem:
  - At the end of (i+1)st parallel step, the input may lie in the set C_{i+1} as large as c_{i+1} such that
    1. No two elements of C_{i+1} have been compared before
    2. $c_{i+1} >= c_i^2 / (2p+c_i)$
- Prove by induction
  - $c_0 = n$ (C_0 includes all given elements)
  - Create a graph of comparisons

So let us look at minimum finding as just an example and its lower bound is log, log N, cannot do any better if few of the N processes and again going to go through just a brief sketch of the proof and just going to use the theorem from graph theory which says that if there are few edges in the graph then there are lot of independent sets. In particular if log n nodes with only N edges then they are guaranteed to be at least on independent set.

Meaning set which are not none of the members of the set are connected by edge in the original graph. Okay, we will be able to find an independent set of at least M square divided by 2 M + 1 elements nodes. Okay, meaning that there are enough independent things to do. Let us see how this is going to come in handy so this other theorem which says that let us look at a generic merges a min finding algorithm or was it min finding.

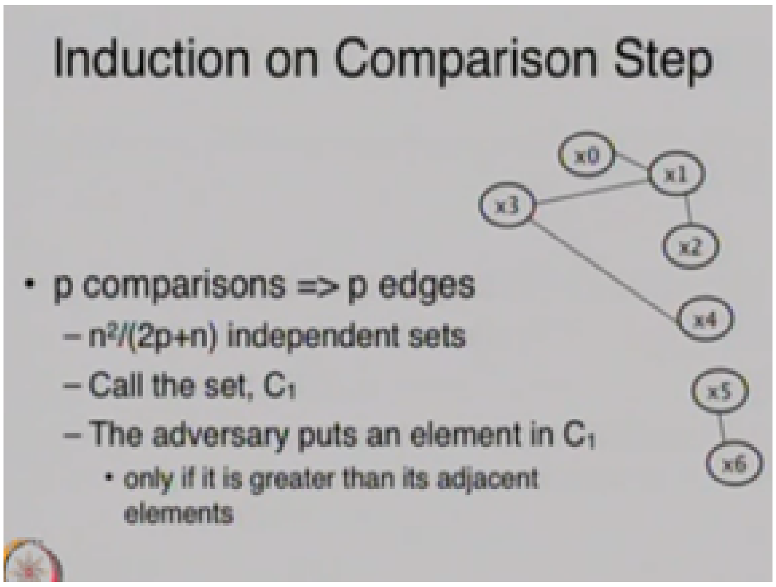
It is going to go through some number of changes and this change is going to go through some number of changes and this stage is going to make peek comparisons. You just have to make some decisions about it. Okay. Whatever the algorithm you choose maybe there is an advisory which is going to give you a bad case, similar argument that we do in sequential algorithms.

So whatever they say that algorithm maybe, we are not talking about any specific algorithm after the C + step, parallel step. You are going to say that it is down with some candidates among which the maxima is going to be. Okay, in the beginning that list of candidates is everybody and after certain number of things it has come down to a smaller set, and after it is done it is come to 1.

That is the minima it is founded. So we are going to say that the set that it still has not figured out which in this set is a real minima is capital C, again for stage I it says Saibai for I + 1. And small C indicates the size of the set. That is how the elements still remaining from which we have to find the minimum. Okay, so now the theorem says that is at least as many as the number of elements remaining in stage I squared dived by 2 P + C I which is the number of elements in the previous graph.

Okay, part looking thin. But it is derived by back engineering from the answer, so we are going to prove this by induction, that this is true for stage zero and it is going to be true for all stages. What is that stage zero? What is the set capital C zero, we have not found anything, we have not done any comparisons yet. And the size is N is in good size. And then we are going to create a graph which indicates what we have compared against.

(Refer Slide Time: 11:22)



Induction on Comparison Step

- p comparisons => p edges
  - $n^2/(2p+n)$ independent sets
  - Call the set, C_1
  - The adversary puts an element in C_1
    - only if it is greater than its adjacent elements

What we have compared against what? Right for example here is a graph where we say that X 1 is compared with X 0 so we make key comparisons does not say that P independent comparisons. Right, it does not say that 2 P unique elements on which these comparisons have to be made. Some P comparisons have been made, some arbitrary tacks which is driven by your algorithm.

You generate this graph, whenever you compare x zero with x one adversary puts a mark, puts an edge there. Okay, and so when you have made three comparisons, we have to sure that we put in pages into this graph. This graph because it is a regular graph has to go through, has to have the same property of how many independent sets it has. Right, and if you go back and look at the number of independent sets.

It is N square dived by 2 M + M, and you can see where this C I square dived by 2 P + C I is coming from, that independent. So this is an independent set, we are going to call it C 1. Why does C 1 have this two properties, at no two elements C I + 1 have been compared. It is an independent set, everything you have compared has an H.

So no two elements have been compared, and your answer is in the set, this is what we build upon in next slide. The answer is in this step, can be arranged right, in the worst case the answer will be in this step. That is the adversary is arranging where the answer is, so that means that whenever you make edges in the graph, like you were seeing in this diagram among the connected components.

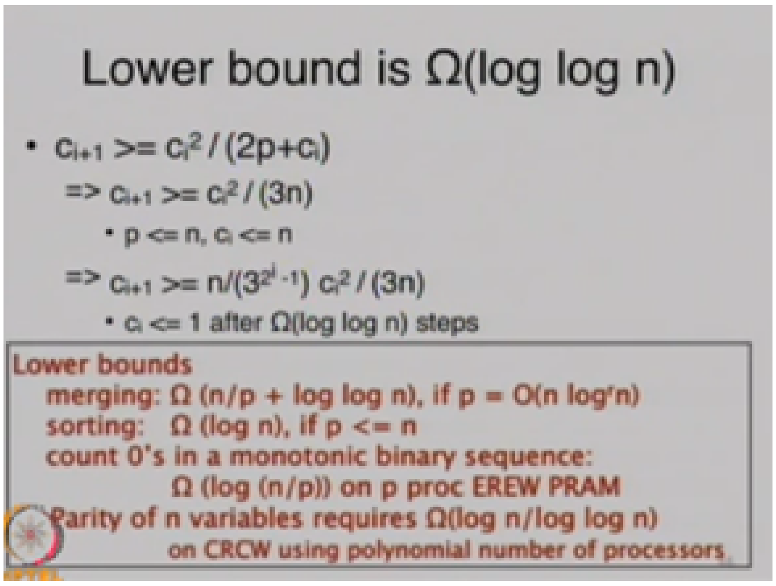
Adversary is going to pick one, not connected components among the one they connected managed, and adversary is going to pick one. For example between X 0, X 1, X 2 and X three adversary may pick X 1, which is the one it is going to pick, it is going to pick among them the X if X 0 and X 1 are connected and x one is smaller, X 0 is smaller than it is going to pick one.

For any pair of connected edges or any pair of connected nodes. It is always going to pick the one that is going to be in your final answer. It is likely to be in your final answer, because it is smaller, it is greater than now let us switched to ma finding. Okay, it is going to pick the smaller out of two, smaller of them for them finding so that whichever it picks is in your answer, is potentially your answer.

For example I know that X 1 is more than X 0, I know X 1 is not magnum, so I would not want as an adversary, it would not want X 1 to carry along. In the still remaining to be touched itself, it will carry along X 0. X 0 is a likely candidate for men. Adversary is going to give you the worst case right, adversary is creating the case, you are creating the edges. You are saying connect, compare these two.

And that is why it is going to be the worst case, yes, it is going to pick up the largest independent set and it is going to always make sure that your answer is in that large independent set somewhere. And so these two are satisfied, that no two elements of C I + 1 have been compared before, because it is sticking an independent set and it also has to satisfy C I + 1 is greater than or equal to C I + 1 is greater than or equal to C I dived by 2 P + C.

(Refer Slide Time: 15:40)



Lower bound is $\Omega(\log \log n)$

- $c_{i+1} >= c_i^2 / (2p+c_i)$
  - $=> c_{i+1} >= c_i^2 / (3n)$
    - $p <= n, c_i <= n$
  - $=> c_{i+1} >= n/(3^{2^i -1})\ c_i^2 / (3n)$
    - $c_i <= 1$ after $\Omega(\log \log n)$ steps

Lower bounds
merging: Ω (n/p + log log n), if p = O(n log'n)
sorting: Ω (log n), if p <= n
count 0's in a monotonic binary sequence:
Ω (log (n/p)) on p proc EREW PRAM
Parity of n variables requires Ω(log n/log log n)
on CRCW using polynomial number of processors

How does that work out, what is P, if we are P is less than equal to 1, will take the worst case so at the extreme of P can be big as N. So we will say C I + 1 is greater than or equal to C I square divided by 2 M + N. right, and C I is definitely less than equal to N, at any time the remaining elements C I is the number of elements remaining to it tested. It is less than equal to N so now you plug in N also.

To plug in N for C I also, so we are plugging N for C I also and for P, and still guarantee that C I plus one, is greater than equal to C I square divided by 2 N + N. any C I square divided by 3 N. okay, now if you go and solve this regression, recurrence relationship C P, C I square

C I + 1. C I square divided by 3 N, you are going to get the answer that C I + 1 is greater than and divided by 3 to the power to the N - 1.

And when does this set become one, when does C I + 1 become 1, when N divided by 3 2 I - 1 becomes 1. Which means three to the 2 I - 1 becomes N. so 3 to the 2 to the, I becomes N, so it is log of N. Only after log of N we guarantee you have reached a set that is only one way.

Okay, alright so there are lots of such if you make a comparison you have at least the algorithm has at least is going to get something right, every time you make an comparison you have rules something out. Yes, okay here are some more examples of lower bounds, similar arguments can be made for those, merging is out of N I have not written N over P every time but anything that takes out the N time in sequential is going to take N over P time in parallel if we are P processes.

So that is hidden somewhere in the analysis all the time. So for merging for example it big omega for M over P plus log of N as long as P is no more than N time sun logged to the constant power of N okay, sorting as long as P is less than N can be done in log in time. Here two interesting, important results that will be handling and proving other bounds, other lower bounds for EREW PRAM.

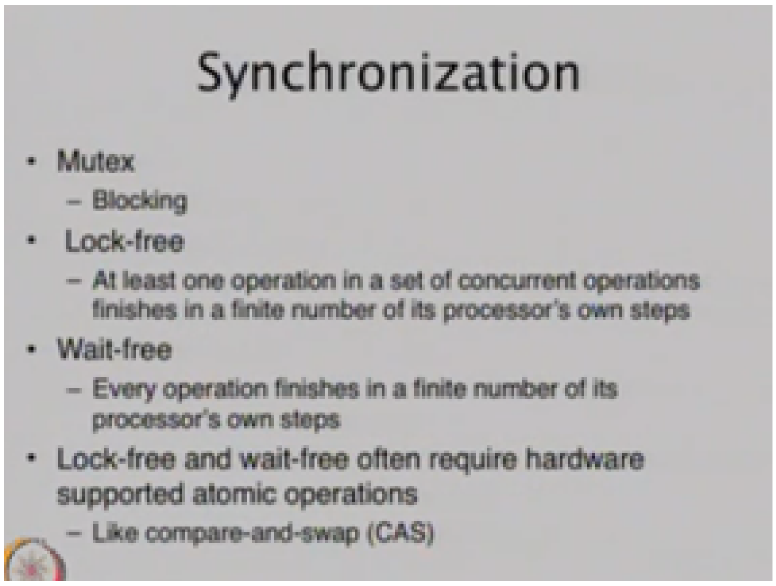
This is the simple problem is I will give you a monotonic binary sequence zeros followed by ones. But only N processes and exclusive read, exclusive right, I want to find where does this zero turn into one, where is that spot, if you can do with more than N processes, if you can do with CRCW you can do with order one line. With EREW to find how many zeros I have in this monotonic sequence takes log of N over P time.

With P processes, as long as P is less than N, even P is more than N that does not make a difference. Log of N over P is more than N will still be order one. Sorry, should this not take, it would if P is beginner and for CRCW which is the most powerful of the three E and C versions. Just to find the parent P variables. I have got N, numbers and I want to find whether they are odd of certain kind or even of certain kind.

It takes log of N dived by log of N okay, as long you are allowed only a polynomial number of processes. And they are again some piece sap by problems that you can get lower bounds that is not polynomial, circuit problems, we are not going to get into those, are there any questions on the basic sorting algorithm and the general discussions of lower bounds.

Okay, then I am going to, I do not know if the last one uses the discussion tree model but I guess you could.

(Refer Slide Time: 21:59)



Synchronization

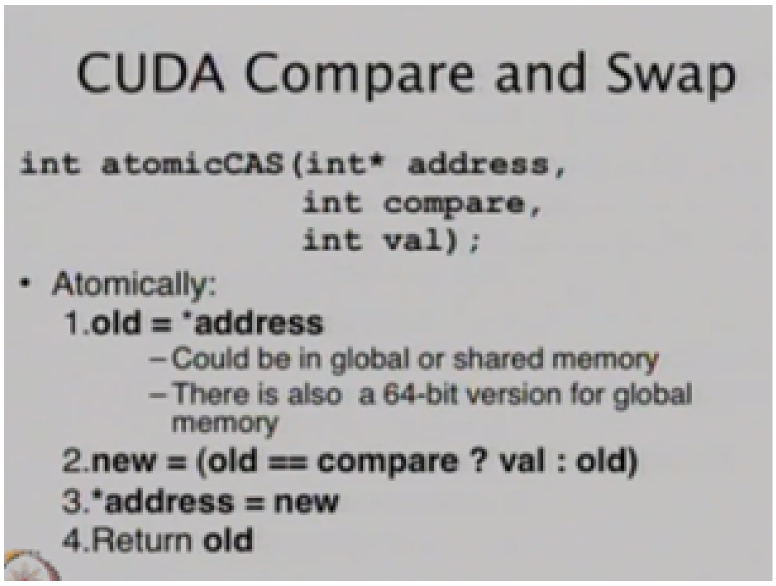
- Mutex
  - Blocking
- Lock-free
  - At least one operation in a set of concurrent operations finishes in a finite number of its processor's own steps
- Wait-free
  - Every operation finishes in a finite number of its processor's own steps
- Lock-free and wait-free often require hardware supported atomic operations
  - Like compare-and-swap (CAS)

Now what I want to switch to is wait free, lock free synchronization where you are not saying lock something and wake me up when that resource becomes available of that at least one example of how you can do that using a simple hardware primitive. Okay, so I do not want to go through this description any detail stage of the course, you should all understand what this slide is talking about.

That mutual exclusion is typically blocking, I go to a certain spot and somebody else is in that critical section then I stop until that critical section becomes available uh the notion of lock free is that, at least that if you make a set of lock free operations everybody is lock free at least one of them is going to complete in a fine item of time, it is fit to what the program is doing. Okay, you cannot guarantee that in regular locks, why?

You can have dead lock, I mean the program can cause a dead lock. But here at least one program is going to proceed. There is also a slightly stressed, stronger version which is called wait free. Which says every operation that a program is going to finish after a constant number of steps. Nobody else can block it for her. And so we are going to look at one example.

(Refer Slide Time: 24:01)



CUDA Compare and Swap

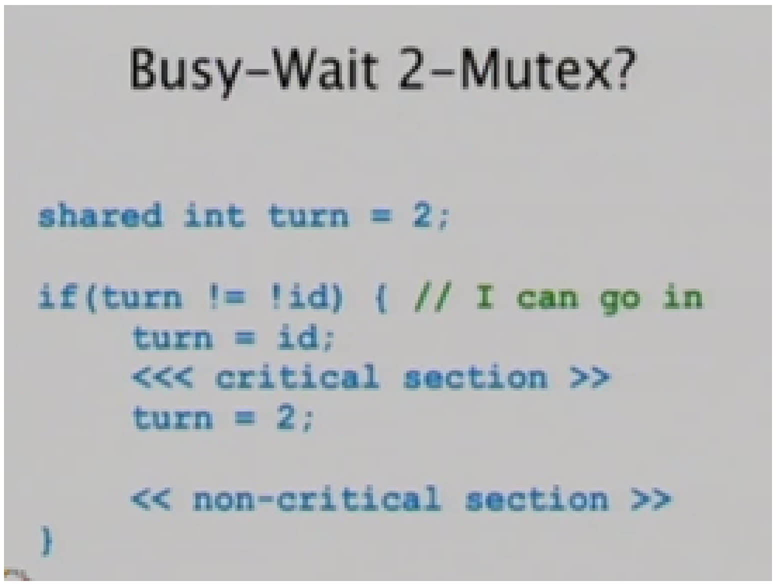
```
int atomicCAS(int* address,
              int compare,
              int val);
```

- Atomically:
  1. **old = *address**
     - Could be in global or shared memory
     - There is also a 64-bit version for global memory
  2. **new = (old == compare ? val : old)**
  3. ***address = new**
  4. Return **old**

One way in which this is done, which is comparisons of in particular you have looked at could has version of compare and swap, although not all compare and swap has the same flavor meaning exactly the same specifications. But they have the same semantics, meaning that I want to write something at some locations and if a certain condition is true that this writing should succeed that condition is false, then I should be told.

That it failed okay, this particular variant is could has variant where it return to the old Val. There may be a variant where you are just a bowline where just returns whether you succeeded or not, whether your new value was written or the value had. Okay, sometime I am just going to switch to keep the quotes small, to the Boolean variant, but this is bit more powerful.

(Refer Slide Time: 25:29)



Busy-Wait 2-Mutex?

```
shared int turn = 2;

if(turn != !id) { // I can go in
     turn = id;
     <<< critical section >>
     turn = 2;

     << non-critical section >>
}
```

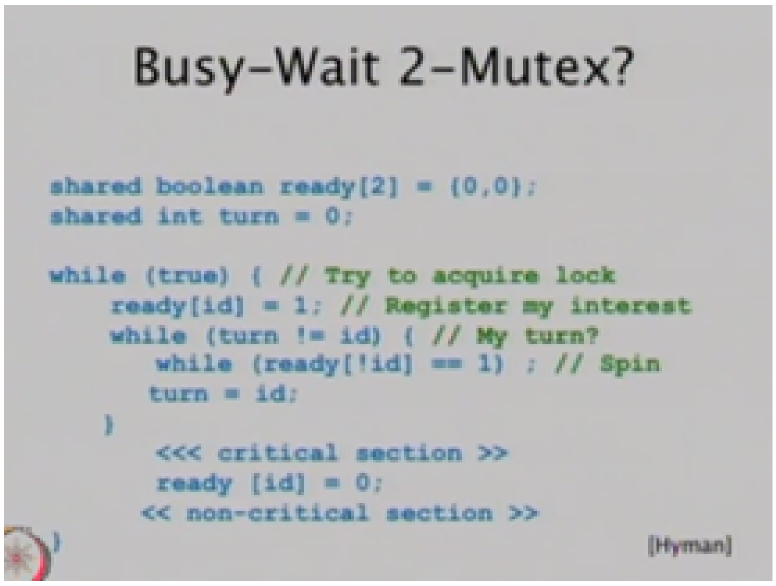
You will get also what address was written over there. Okay, here is on example I have a critical section and I write quote around that critical section which every thread executes, right, if says it is my turn is not equal to my id then I should get out to something else. And if otherwise this is for two only, in fact the slide says two new text and if my turn is not equal to not of my id, I mean not to other guys turn then I can go, proceed.

Then in my turn going to the critical section and after I am done I turn it back to the nil, nobody start. Will this fail not necessary well it should this is a trivial. This is going to fail so you have one thread that comes, checks if turns not equal to mine not id proceeds, before it gets a chance to turn the id, turn the turn to its own id another fellow also comes in.

Checks it and now both of them are ready to turn the ID, one of them are succeed, I mean both will succeed but one will over-ride the other. But both go to h critical section. Okay, and then after that the turn will turn into two and all that business. Is this which was proposed in the in paper in seventy something? As a possible solution to the problem one I am going to stop here, because we are out of time I am going to upload this slide.

Tell me what the problem is on Thursday we will discuss that and then we will figure out the solution of the product, solution in terms of lock free techniques, we can always it is very simple in the context of locking the door right, because lock the resource vertebrate two consumptions, unlock it. That works, there is only one lock to acquire no deadlock. But here we are trying to do without herniate lock so nobody actually needs to wait.

(Refer Slide Time: 28:28)



Busy-Wait 2-Mutex?

```
shared boolean ready[2] = {0,0};
shared int turn = 0;

while (true) { // Try to acquire lock
    ready[id] = 1; // Register my interest
    while (turn != id) { // My turn?
       while (ready[!id] == 1) ; // Spin
       turn = id;
    }
     <<< critical section >>
     ready [id] = 0;
    << non-critical section >>
}
```

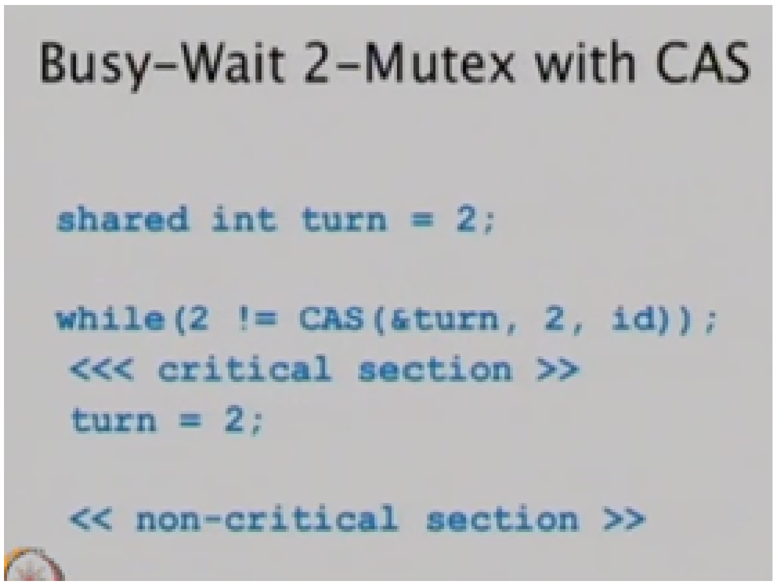
[Hyman]

For somebody else, okay so then time for the quiz, so there was a mutual fusion code that we had, before which turn? So where would one be? One goes in while true that is you are talking about? Ready of both is equal to zero so neither of them are ready. Whoever comes in says I am ready. Yes, which spin you are talking about tell me it is not y is equal to y ten lock, okay.

Yes, right it has not made its turn yet, right, so the other way if zero comes here and it is going to set its ready in the meantime on also is coming and it has gone past the tests and at this time zero has not set its ready yet, so one does not know whether zero is ready or things is ready. So it checks the other fellow is not ready so feels that is going to turn make the turn its own turn.

So it is going to set its turn to id, but it has not done yet. Waiting just before turn equal to id, in the meantime aero woke up went through all of this, because he just thought there was nobody else. And then when zero has gone through after that one says that it is okay and one is set my turn my id in turn variable proceed to the critical section, suppose if we have one in the critical section.

(Refer Slide Time: 31:39)



Busy-Wait 2-Mutex with CAS

```
shared int turn = 2;

while(2 != CAS(&turn, 2, id));
 <<< critical section >>
 turn = 2;

 << non-critical section >>
```

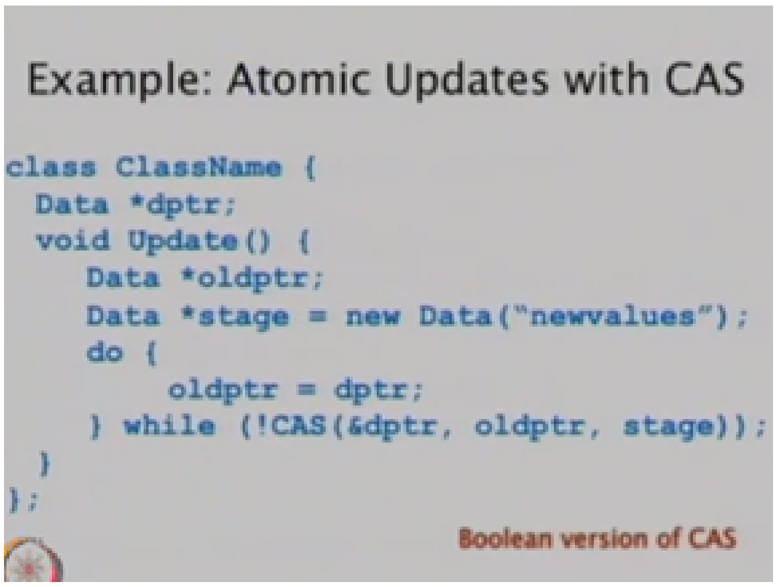
And there were other attempts such essentially all fail there is some kind of locking that is necessary and this is one kind of locking which is none weighting lock, it is not blocking lock. Where, remember we talking about compare and sect and cast and here we are using the int variant of cast, int is the turn and the initially turn is equal to two and I want to set it to my turn as long as it is equal to two.

If the turn is two then I want to set my turn, then if my turn is not true then I do not want to set now. Have to do this accordingly, so both of them may come to the class of the same type but only one of them will be able to see the tool and write their id atomically. Right, so the other one will fail and we will retry, right, it is busy writing in this case you can imagine that you did not get the lock task fail and then you go do something else and then you come back fighting and upset of tires.

And once you are done and then you set your turn to back to whatever it was, so that somebody else now can come in. is this clear, alright. Often you need many things to happen atomically around to set that value, this value and the third value automatically, there is a constructer in this last name which is going to set some different fields. And you can basically lack lock on the constructors, or lock on the entire set itself.

Let us say we are going to construct it in the new are, where so I create a stage pointer. I construct it in a stage area. And atomically simply switch the pointer to the new block of data that was created.

(Refer Slide Time: 34:09)



Example: Atomic Updates with CAS

```
class ClassName {
  Data *dptr;
  void Update() {
     Data *oldptr;
     Data *stage = new Data("newvalues");
     do {
          oldptr = dptr;
     } while (!CAS(&dptr, oldptr, stage));
  }
};
```

Boolean version of CAS

Right so now as long as the old pointer that in this variable called old pointer is which I had seen in the beginning remains in this data point. The pointer, I will update it, switch it to my value. Otherwise, I will just keep waiting or keep looking. So far busy, because in this case there is nothing else, this is a simple example where only thing you have to do is constructor, only thing you want to do is critical sections.

But that is not necessarily true, we will see some examples, where we see it in a more wider context, and technically where it would not be busy waiting, the structure is busy waiting but you will go do something else then come back and check it again, so it will going to be a loop. Why is it better and is it better than regular lock? Yeah, so, cash issues are subtle it is going to show here also.

We have to make sure its variable are volatile. Well lock also you determine, you mean here? It is at a very small granulite, it is a one memory location right, there is generally a performance game but more importantly you can do the text and go do something else. If you block your blocked you are waiting, which is perfectly fine you are doing a worst assignment with multiple threads on one pause, one core.

Right, if all the threads are sharing the same core perfectly fine for few of them to be waiting others will be working. But if there is one thread running on one core, or few enough threads running on one core then you do not want to block you do not want to say locked and wait until I get my lock. If I have something useful to do I would rather do something.

So this is much more handy in the parallel programming contest then the general concurrent thread programming context. Okay, so now let us look at what reallocation of this might be, I wanted to do dynamic load balancing, remember we talked about load balancing business in the beginning everybody has their key of work to do and somebody gets done early they want to take the work from somebody else.

Okay, so we will imagine that there is a cube that is being maintained for all independently, different cubes somebody in the beginning did an allocation in. Default allocation maybe one EF of the data has been put on each other cubes. Right, then they start working some of the processes because their job was less loaded or bitterly processes because they are generally more loaded gotten light.

So whoever gets done early they will go look for some other prospects to steal their jobs, steal some of their jobs. Okay, remember this so what are the critical reasons here, where is the data being shared? What are the rash conditions? So everybody has a cube right, if I am going to so if I get done right, what is my general loop, sorry I will show it to you later.

I will just go through my list of task, so my task equal to zero, for task equal to one size I am just going to go task plus and then I will take whatever input there simply to a procedure that will support the task. Once I am done and I realized that others have not done, right and become have mechanism that everybody are done for the time being we are ignoring that.

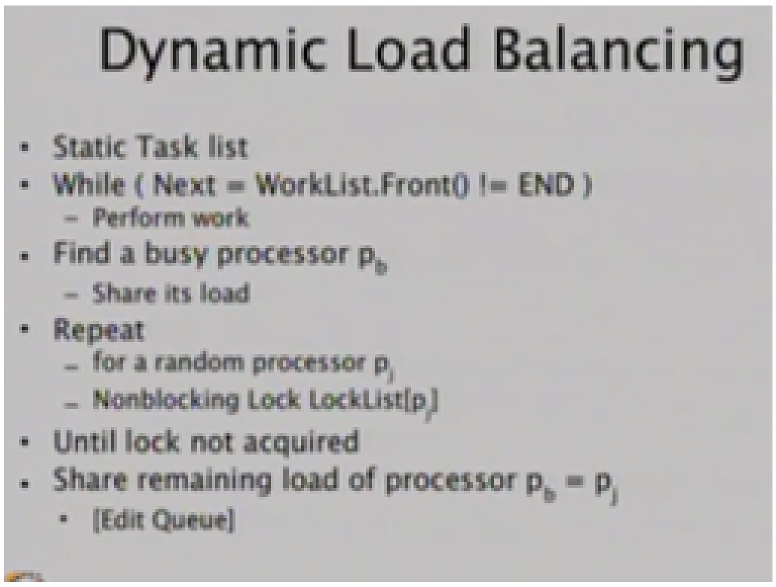
So I am done I need to go look for somebody else and take their task. And one of our goals here would be you are working on your task if you are going to be busy because you are behind you do not want to be locking every element in the cube. Then before saying give me the task assuming that lock has over that not accepted if this tasks are small number task with large number of them.

Then lock is going to start to be very difficult. Because for every little element you have to start the lock. So we do not want the worker which is in this case is the victim from where the stealing is going to happen to ever slow down. Right, which means if it has to do a cash maybe it is okay it should never do a lock. And then the victim is going to the thief is going to go through the list and figure out which has enough work done to, for me to steal and I am going to steal it.

How do I make sure that so there is a race between me and the thief and the victim? But other things also, two things I want to steal from the same victim. Suppose in this case we made this choice that just for a simplifications that thieves can wait for each other. Let us lock the thief out meaning that if the two thieves trying to access the same victims data then they must decide upon themselves on who is going to access.

And if in the process they have to slow down a little bit it is okay. Because they are done their work. But what you do not want to do is slow down the busy processor who is still working on its tasks. Okay, so let us just design the session. So, let us get through this.

(Refer Slide Time: 40:50)



Dynamic Load Balancing

- Static Task list
- While (Next = WorkList.Front() != END)
  - Perform work
- Find a busy processor p_b
  - Share its load
- Repeat
  - for a random processor p_j
  - Nonblocking Lock LockList[p_j]
- Until lock not acquired
- Share remaining load of processor $p_b = p_j$
  - [Edit Queue]

We are going to work through the less and until the Q or whatever until the Q is less is finished, you are going to continue to perform the work. once we are done we are going to find the busy processor okay, which means I am going to choose one random other processes.

One J not equal to my eye and go see whether his front and end of car park, front is the front of the key, where he is still, he is going to take the limits out. And then there is a end of the key. And I want also to make sure that nobody else no other thieves locks, no other thief is trying to steal from other victim. So if I do anon blocking lock, right, so I choose processor number ten for the victim.

For some reasons somebody also choose processor number ten for victim. If I have a lock for ten one of us will acquire. The will go look for some other victim. Right, so there is no busy waiting about. If you fail then you pick out another random number and you try that for yourself. Okay, so it is still non-blocking. But we are now guaranteed that I f I was able to acquire the let us say the thief lock it's the victims never coordinate with this lock, only thieves try to lock.

So if the lock for PB, but is a busy processor that was acquired by EI. Then PI is the only processor which is going to steal from P. okay, now so we are restricted to how PI and PBB interact. How should PI take elements and how should PB having to suffer a slowdown. And without race conditions. Right, maybe it should not maybe in some situations it is okay to do some task twice but it is usually never okay to miss a task. So we are going to be very careful about those things.

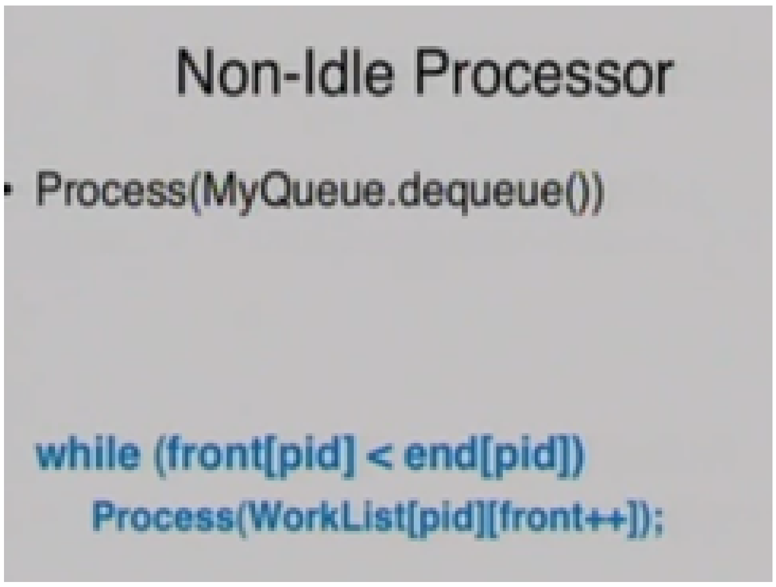
So we will repeat until this lock is acquired, some random busy processes. Whether we are going to share the load of that busy processes. PJ here becomes the PB. And we are going to edit that processes cube and that J becomes the PB. And we are going to edit that processor cube and that's where the majority of this work is going to be now. And after we do that a lock this thief locks.

And that if somebody wants to steal some more now they can okay. Alright, so how well non-blocking lock work we have been seeing so far, we will simply initialize certain variable which is for every processor and we will is that processors variable zero. If it is zero nobody is stealing from it right. If no otherwise am going to steal from it. I will put my value there. And I do not want to do this at all.

So I say I expect the value zero I want to put y thread id there if during putting my thread that zero was not seen then I tell I return and I go to something else. No the difference here is, right, for blocking lock you will say lock that thing right, which means if somebody is there then you are going to get struck until that guy unlocks. And that guy will unlock when he is finished stealing from that key after that you might go ahead and steal or you may choose somebody else.

Now as soon as you know that somebody else started stealing from there. You go to stealing somewhere else that is the difference.

(Refer Slide Time: 45:19)



Non-Idle Processor

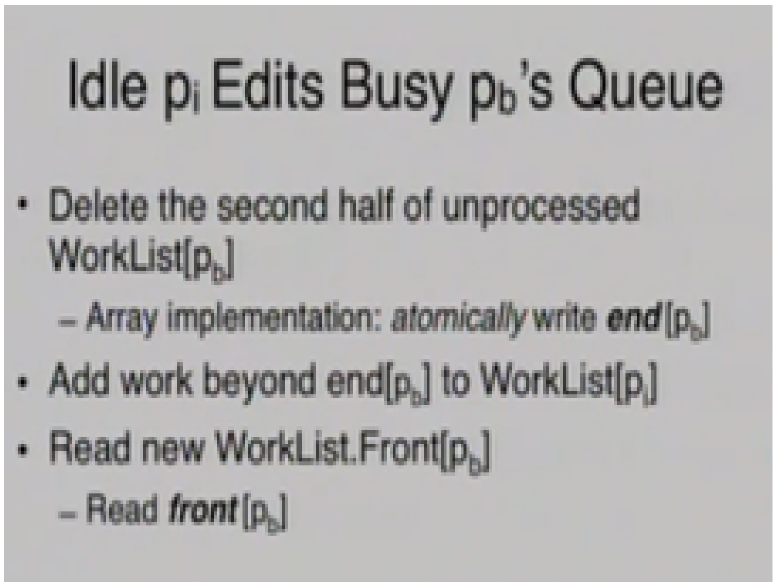
- Process(MyQueue.dequeue())

```
while (front[pid] < end[pid])
    Process(WorkList[pid][front++]);
```

Okay, so what does this non ideal processor do? Generally just that look we had earlier. We will there are various ways this might be done and here they are saying that it is one in the form of an array okay. So there is a front and end for every processor and I will, check my id, check my front if my front is less than my end then I, this is not going to be exactly this way but this is at least in sequential version it will look something like this.

Okay, now what I do not want again is every time lock front because there is herein. Front and end are going to be, front will be red by the thief to figure out where I am. End will also be red by the thief to figure out how much remains. And end will be overwritten by the thief so that it reduces my end to something earlier. Right so there is significant amount of red is going on.

(Refer Slide Time: 46:35)



Idle p_i Edits Busy p_b's Queue

- Delete the second half of unprocessed WorkList[p_b]
  - Array implementation: *atomically* write ***end***[p_b]
- Add work beyond end[p_b] to WorkList[p_i]
- Read new WorkList.Front[p_b]
  - Read ***front***[p_b]

So you would not want necessarily a lock on this cube. And that is a, that is what we are trying to so we have now us a thief pi, which is going to steal from PB okay. And the general theme is you are going to take the second half of the remaining work approximately. It does not have to be precise approximately the second half of the work so, we are going to read the other persons end.

Read the other persons front and make the end, front plus end minus front divided by two. Right, and if I am off by a little bit by that time that fellow has done a few more things, it is okay. I do not need it to be precisely half of the remaining work because I do not know when the remaining work is to be serialized. Okay, at some point it is hard. And then I am going to take those stars to my cube.

I am going to start working and until I have taken it to my cube it is possible that I have just stolen from somebody and somebody is not trying to steal from me. Because I suddenly get some unfinished business. And somebody may get done and somebody may get my random number and say give me some and take half of my work. So I have to be careful about that also.

Okay, so this is the overall algorithm we are going to read the front I am going to have much more formal code like description shortly. Read the other persons front, figure out the end. Write the end to the other person. Read the front of the person. So let us say this a simple case, I read the front read the end take the half right, suppose the new is N prime. I write N prime into that fellows end.

And then make a private copy and forget about somebody stealing from me for time being and proceed. What is going to happen, what kinds of things might happen? Okay, that is one problem others. We have already taken care of that, yea so because they are stealing you are ignoring. Of course that is an issue. Because it is a stealing it is easy to solve. Nobody is, if I do not want anybody to steal from me right now.

For some reason I am busy doing things all I have to do is not update my end, right. Do not even want to put a lock at my end. My cue is this much, my front and end are here the global end that everybody else is remains here. I start filling the rest of the cube, as far as the other circumstance they think that the my cue is full. Only when I am finished putting things in a cube I automatically update it to my bottom. And after I have updated it people can steal, so there is no lock needed over there.

So what other problems can we go on, one is that I have stolen something and that fellow is finished? It may not even be finished it may be proceeded alright. It may have finished. It may will it ever get into a cordon type situation. If I am atomically writing an end right, and that guys front has got beyond the end I is going to stop. Next time it says that give me the front and the front is greater than end. It is going to stop

Because it says why front less than equal to where. Right, so it is going to stop as soon as it sees a new end. So after I write my end it is not going to do any more work beyond the end. It may have done it before though. Before I get to write my end it may have finished altogether it may have gone past my end. We are not considering what the tasks are, it could be. Possibly, right. For example I am drawing triangles somebody draw this triangle twice.

Maybe does not matter right. So that is not a rest condition. In other cases you want to make sure that. Yea, there would be a race condition then you have to make sure that race condition is also handled. So right now we are only considering the Q business. So how we are stealing the work no assuming that the works are independent. So there is no race among the work between the works.

Well, if it is a queue of task and there is dependency then you would not have a queue of task anywhere. Right, no but there is queue over here I am proceeding at my speed, there is a queue over there that fellow is proceeding at his own speed and there is no dependence. So there are already, that will still be maintained. Can still be maintained. Right, because I will always do things in the order of the cube.

And if something gets stolen that has to be done, well no that cannot be stolen then, right, if this part is sequential then no point in stealing. If it has to wait for other things that is not done then what will the other processes do. It has to wait for the thing that has not done. So this is not taking the sequential dependency into the count. Okay, this is more about I have a cube and we are sharing the cube.

But sharing is limited in certain ways we will move on to also have a general linguist let us say, all queue implements of linguist. This again static queue, somebody has given us the amount of work and we had to together finish it as soon as possible. And work does not get added. So this is defiantly listed in the sense, there is no demonstrations of how cast is working, cast would be used.

In a real application than a general load balancing sigma. Although this is a powerful load balancing sigma. And surprisingly large number of applications are happy just this much. Because they actually do have independent tasks. Which can be done in any order actually. Right, that would involve not necessarily locks but cast you atomically increment something and that is done to peek then you are done. So let me stop for just a minute and continue the those two issues that we were talking about.