PROFESSOR: The last time we talked about nondeterministic programming. And I think actually the mic is up pretty high. If we can tone that down just a little bit. We talked about nondeterministic programming. And as you recall, the rule with nondeterministic programming is you should never do it unless you have to.

Today we're going to talk about synchronizing with locks. And it goes doubly that you should never synchronize without locks unless you have to. There's some good reasons for synchronizing without locks as we'll see. But it, once again, becomes even more difficult to test correctness and to ensure that the program that you think you've written is, in fact, the program you meant to write.

So we're going to talk about a bunch of really important topics. The first is memory consistency. And then we'll talk a little bit about lock free protocols and one of the problems that arises called the AVA problem. And then we're going to talk about a technology that we're using in the Cilk++ system, which tries to make an end run around some of these problems and allows you to do synchronization without locks, with low overhead. But it only works in certain context.

So we're going to start with memory consistency. So here is a very simple parallel program. So initially a and b are both 0. And processor zero moves a 1 into a. And then it moves whatever is in location b into the EBX register.

Processor one does something complementary. It moves a 1 into b. And then it moves whatever is in a into the EAX register. Into the EAX register as opposed to the EBX register.

And the question is what are the final possible values of EAX and EBX after both processors have executed. Seems like a straightforward enough question. What

values can EAX and EBX have, depending upon-- there may be scheduling of when things happen and so forth. So it's not always going to give the same answer. But the question is what's the set of answers that you can get?

Well, it turns out you can't just answer this question for any particular machine without knowing the machine's memory model. So it depends upon how memory operations behave in the parallel computer system. And different machines have different memory models. And we'll give you different answers for this code. There'll be some answers that you get on some machines, different answers on different machines.

So probably the bedrock of memory models is a model called sequential consistency. And this is intuitively what you might think you want. So Lamport in 1979 said, "The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

So what does that mean? So what it says is that if I look at the processor's program and the sequence of operations that are issued by that processor's program, they're interleaved with the corresponding sequences defined by the other processors to produce a global linear order. So the first thing is that there's a global linear order that consists of all of these processors' instructions being interleaved.

In this linear order, whenever you perform a load from memory into register, it receives the value that was stored by the most recent store operation in that linear order to that location, i.e. it's memory. So you don't get something if you have in this linear order that two processors wrote. Well, one of them came last. The most recent one before you read, that's the one that you get.

Now, there may be many different interleavings and so forth. And you could get any of the values that correspond to any of those interleavings. But the point is that you must get a value that is represented by some interleaving.

The hardware can then do anything it wants, but for the execution to satisfy the

sequential consistency model, for it to be sequentially consistent, must appear as if the loads and storage obey some global linear order. So let's be concrete about that with the problem that I gave before. So initially, we have a and b are 0. And now, we have these instructions executed.

So what I have to do is say, I get any possible outcome based on interleaving these instructions in this order. So if I look at it, I've got two instructions here, two over here. So that there are six possible interleavings because 4 choose 2 is 6 for those people who've taken 6042. So there are six possible interleavings.

So for example, if I execute first move a 1 into a, and then I execute move load into register, the value of b, and then I move 1 into b, and then I load the value of a, I get a value of 1 for EAX and a value of 0 for EBX. For this particular interleaving of those instructions.

That's what happens if I execute these two before these two. If I execute these two instructions here before these two here, I get the order 3412. And essentially, the opposite thing happens. EAX gets 0 and EBX gets 1. And then, if I interleave them in some way, where 1 and 3 somehow come first before I do the 2 and 4, then I'll get a value of 11 for each of them. Those are the middle cases.

So what don't I get?

AUDIENCE: 00

PROFESSOR: You never gets 00 in a sequentially consistent execution. Sequential consistent implies that no execution-- whoops, that should be EAX. That EAX equals EBX equals 0. I don't ever get that outcome. If I did, then I would say my machine wasn't sequentially consistent.

So now let me take a detour a little bit to look at mutual exclusion again. And understand what happens to mutual exclusion algorithms in the context of memory consistency. So everybody understood what sequential consistency is. I simply look at my program as if I'm interleaving instructions.

So most implementations of mutual exclusion, as I showed previously, employ some kind of atomic read-modify-write. So the example I gave you last time was using the exchange operation to atomically exchange a value in a register with a value in memory. People remember that? To implement a lock? So in order to implement a lock, I atomically switch two values.

So we, in particular, use the exchange one. And there are a bunch of other commands that people can use. Test-and-set, compare-and-swap, load-linked-store-conditional, which essentially do some kind of read-modify-write on memory. These tend to be expensive instructions, as I mentioned. They usually tend to cost something like an L2 cache hit.

Now, the question is can mutual exclusion be implemented with only atomic loads and stores? Do you really need one of these heavyweight operations to implement mutual exclusion? What if I don't use our read-modify-write? Is that possible to do it?

And in fact, the answer is yes. So Dekker and Dijksra show that it can as long as the computer system is sequentially consistent. So as long as you have sequential consistency, you in fact, can implement a mutual exclusion with read-modify-write.

We're actually not going to use either the Dekker or Dijksra algorithms, although you can read about those in the literature. We're going to look at what is probably the simplest such algorithm that's been devised to date, which is devised by Peterson.

And I'm going to illustrate it with these two smileys. That's a she. And that's a he. And they want to operate on widget x. And she wants to frob it. And he wants to borf it. And we want to preserve the property that we are not frobbing and borfing at the same time.

So how do we do that? Well, here's the code. So we're going to set up some things before we start he and she operating. So we're going to have our widget x. That's our protected variable. And we're going to have a Boolean set initially to false that says whether she wants to frob it. So we don't want to make them frob it unless they

want to frob it. And we don't want him to borf it unless he wants to borf it.

And we're going to have an extra auxiliary variable, which is whose turn it is. So they're going to sort of do a take turn. But that only is going to come into account if the other one doesn't have a conflict. If they don't have a conflict, then one of them is going to be able to go.

So here's what she basically does. She initially sets that she wants to operate on the widget. And then, what she does is she sets the turn to be his. And then, while he wants it, and the turn is his, she's going to just spin.

Notice that you're not frobbing it in while loop. The body of the while loop is empty. So this is a spinning solution. So while he wants it, and it's his turn, you're just going to sit there, continually testing the variables he wants and turn equals his until one of them ends up being false.

So if he doesn't want it, or it's not his turn, then she gets to frob it. And when she's done, she sets she wants to false. And he does a similar thing. He sets it to true, says it's her turn. And then, while she wants it, and the turn is hers, just sits there waiting, continually re-executing this until finally, one of these turns out to be false. And then he borfs it. And he sets it to false.

And then, they're doing both of these things sort of in a loop, periodically coming back and executing it. And what you want to do is you don't want to make it so that it's forced. That it's one turn, then the other because maybe he never wants to borf it. And then, she would be stuck not being able to frob it, even though he doesn't want.

So if you think about this-- let's think about why this always is going to give you mutual exclusion. So basically, what's happening here is if he wants it-- by the way, these things are not easy to reason about. And usually, as much as I can talk and talk in class, what you really need to do is go home, and sit down with this kind of thing. And study it for 10 minutes. And then, you'll understand what the subtleties are as what's going on.

But basically, what we're doing is we're making it so that it's not going to be the case that both she's setting it and she wants it. And the turn is his. And then, if there's a race where he wants it also, then that's going to preclude both of them from going into it at the same time. And then whichever one sets the turn, one of those is going to occur first. And one is going to occur second. And whoever ends up coming second, the other one gets to go ahead. So it's very subtle how that is actually working to make sure that each one is gating the other to allow them to go.

But the way to reason about this is to reason about it is what are the possible interleavings? And the important interleavings here as you can see are what happens when setting these things. And once they're set, what happens in testing these things? And especially because when you go around the loop and so forth, you have to imagine that an arbitrarily long amount of time is gone.

So for example, between the time that you check that the turn is his, he may have already gone around this loop. And so you have to worry about-- even though, it may look like one instruction from this processors point of view for correctness purpose, you have to imagine that an arbitrary amount of computation could occur between any two instructions. So any question about this code? People see how it preserves mutual exclusion and how you use sequential consistency to reason about it by asking what are the possible interleavings? Questions? Yeah.

AUDIENCE: So, I don't know if I got it right. So basically, sets the [UNINTELLIGIBLE] to give him a chance before she goes to loop. So basically, she waits there until he has been able to go? That's why on the [UNINTELLIGIBLE].

PROFESSOR: So on this third line--

AUDIENCE: Both of them. Either before actually frobbing or borfing. And before that while you always give the turn to the other to give them a chance to go.

PROFESSOR: Yeah. So there are two things you want to show. One is that they can't both be stalled on the while loop there. And that can't happen because the turn can't be simultaneously his and hers. So you know that they're not both going to deadlock in

trying to do this by sitting there waiting for the other because of this.

And now, the question is well, how do you know that one can't get through while the other is also going through? And for that, you have to look and say, oh well, if you go through, then you know that it is either he doesn't want it, or it's not his turn. And in which case, if he doesn't want it, it's not his turn. If he does change it to that he wants it, then in fact, it's going to be your turn. Question?

AUDIENCE: This only works for exactly two threads, right?

PROFESSOR: This only works for exactly two threads. This does not work for three, but there are extensions of this sort of thing to end threads in an arbitrary large number. However, the data structures to implement this kind of mutual exclusion for end threads end up taking space proportional to n.

And so one of the advantages of the built in atomics-- the compare-and-swap, or the atomic exchange, or whatever-- is they work for an arbitrary number of threads with only a bounded amount of resource. You don't require extra data structures and so forth. So that's why they put those things in the architecture because in the architecture you can build things that will solve this problem much more simply than this sort of thing. However, there are going to be lessons here that you may want to use in your programming, depending on what you're doing.

So now, it turns out that no modern day processor implements sequential consistency. There have been machines that were built-- actually quite good machines-- that implemented sequential consistency. But today, nobody implements it.

They all implement some form of what's called relaxed consistency, where the hardware may reorder instructions. And so you have things executing not in program order. And the compilers may reorder instructions as well. So both the hardware and the software are going in there. So let's take a look at that.

So here's the program order for one of the things. We move 1 into a, and then move the value of b into EBX to do a load. Here's the program order. Most modern

hardware will switch these and execute it in this order. Why do you suppose?

Even if you write it this way, the instruction level parallelism within the processor will, in fact, execute it in the opposite order most of the time. Yeah?

AUDIENCE: Because loading takes longer.

PROFESSOR: Yeah. Because loading takes longer. Loading is going to take latency. I can't complete the load from the processor's point of view until I get an answer. So if I load, and I wait for it to go out to the memory system and back into the processor, and then I do a store-- well, as soon as I've done the store, I can move on.

Even if the store takes a while to get out to the memory system. But if I do it in the opposite order. I do the store first, and then I do the load, I've ended up wasting essentially one cycle, the cycle to do the store, when I could have been overlapping that with the time it took to do the load. So people follow that?

So if I execute the load first, I can go right on to execute the store. I can issue the load, go right on to execute the store without having to wait for the load to complete if I have a multi-issue CPU in the processor core. So you get higher instruction level parallelism.

Now when is it safe for the hardware compiler to perform this reordering? Can it always switch instructions like this to put loads before stores? When would this be a bad idea to put a load before a store? Yeah?

AUDIENCE: You're loading the variable you just stored.

PROFESSOR: Yeah, if you're loading the variable you just stored. Suppose you say store into x and then load from x. That's different from if I load from x, and then I store into x. So if you're going to the same location, then that's not a safe thing to do.

So basically, in this case, if a is not equal to b, then this is safe to do. But if a equals b, this is not safe to do. Because it's going to give you a different answer.

However, it turns out that there's another time when this is not safe to do. So this

would have been the end of the story if we were running on one processor. The other time that it's not safe to do it is-- if it's safe, the other assumption is that there's no concurrency. If there is concurrency, you can run into trouble as well. And the reason is because another processor may be changing the value that you're planning to read. And so if you read things out of order, you may violate sequential consistency.

Let me show you what's going on in the hardware so you have an appreciation of what the issue is here. So here's 30,000 feet of hardware reordering. So the processor is going to issue memory operations to the memory system. And results of memory operations are going to come back. But they really only have to come back when? If they're loads. If they're stores, they don't have to come back.

So the processor, in fact, can issue stores faster than the network can handle them. And the memory system can handle them. So the processors are generally very fast. The memory systems are relatively slow.

But the processor is not generally issuing a store on every cycle. It may do store, it may do some additions, it may do another store, et cetera. So rather than waiting for the memory system to do every store, they create a store buffer. And the memory system pulls things out of the store buffer as fast as it can. And the processor shoves stuff into the store buffer up to the point that the store buffer gets full, in which case it would have to stall. But for most many codes, it never has to stall because there is a sufficient frequency of other operations going on that you don't have to wait. So when a store occurs, it doesn't occur immediately on the store buffer.

Now along comes a load operation. And the load operation, if it's to a different address, you want to have that take priority because the processor can be waiting. It's next instructions may be waiting on the result. So you want that to go as fast as possible.

They have a passing lane here where the fast cars or the important cars, the ambulances, et cetera, in this case loads, can scoot by all the other things in traffic

and get to the memory system first. But as we said, we don't want to do that if the last thing that I stored was to the same address.

So in fact, there is content addressable memory here, which matches the address that is being loaded with everything in the store buffer. And if it does match, it gets satisfied immediately by the store buffer. And only does it make it out to the network if it's not in the store buffer.

But what you can see here is that this mechanism, which works great on one processor, violates sequential consistency because I may have operations going to two different memory locations, where the order, in fact, matters to me. So let's see how that works out.

So first of all, let me tell you what the memory can-- so a load can bypass a store to different address. First of all, any questions about this mechanism? So this accounts for a whole bunch of understanding of what happens in concurrency in systems. This one understanding of store buffers. It's absolutely crucial.

And I have talked, by the way, with lots of experts who don't understand this. That this is what's going on for why we don't have sequential consistency in our computers. It's because they made the decision to allow this optimization, even though it doesn't preserve sequential consistency.

There were machines in the past that did support sequential consistency. And what they did was they used speculation to allow the processor to assume that it was sequentially consistent. And if that turned out to be wrong, they were able to roll back the processor's state to the point before the access was done.

In fact, the processor is already doing that for branches, where it makes branch predictions and executes down a line. But it's wrong, it has to flush the pipeline and so forth. Why they don't do the same thing for hardware is an interesting-- for loads of stores-- is an interesting question. Because at some level there's no reason they couldn't do this.

Instead, it's sort of been a thing where the software people say, yeah we can handle

it. And the hardware people say, OK. You're willing to handle it. We won't worry about it then. When in fact, it just makes life complicated for everybody that you don't have sequential consistency.

AUDIENCE: [INAUDIBLE] you have to do speculation across both [INAUDIBLE].

PROFESSOR: Well here, you only have to do speculation over what actually is coming out of your memory system. And if it doesn't match, you could roll back. The issue, in part, is how many machine states are you ready to roll back to. Loads come more frequently than branches. That's one thing. So no doubt, there are good reasons for why they're doing it. Nevertheless, definitely loss of sequential consistency becomes a headache for a lot of people in doing a concurrent program. We had a question here? Yes, Sara?

AUDIENCE: So this does not preserve sequential consistency? But as long as there's only one processor, it should have the same effect, right?

PROFESSOR: But sequential consistency for one processor is easy because all you do is execute them--

AUDIENCE: Yeah, I'm just saying--

PROFESSOR: It should have the same effect, exactly. So on one processor, this works perfectly well. If there's no concurrency, this is going to give you the same behavior. And yet, you've now got this optimization that loads can bypass stores. And therefore, you can do a store and a load and be able to overlap their execution. So this definitely wins for serial execution.

Yep, good. Any other questions about this mechanism? You could reason about it on the quiz. That kind of thing, right? Yeah, OK?

So here's the x86 memory consistency model. For many years, Intel was unwilling to say what their memory consistency model was for fear that people would then rely on it. And then, they would be forced into it. But recently, they've started being more explicit about it. And this is the large part of it. I haven't put up all the things

because there are a whole bunch of instructions, such as locking instructions and so forth, for which for some of them, it's more complicated. But this is the basics.

So loads are not reordered with loads. So if you add a load to one location, a load to another location, they always execute in the same order. Stores are not reordered with stores. If you have store and then a subsequent store, those two stores always go in that order. Stores are not reordered with prior loads. So if you do a store after a load-- if you do a load and then a store, they're going to go in that order.

However, a load-- and this is what we just talked about-- may be reordered with a prior store to a different location but not with a prior store to the same location. So that's exactly what we just talked about on the previous slide. Then, loads and stores are not reordered with lock instructions.

So a certain set of instructions are called lock instructions. And they include all the atomic updates, the exchanges, comparisons-and-swaps, and a variety of other atomic operations that the hardware supports.

The stores to the same location always respect a global order. Everybody sees the store to a location in exactly the same order. And the lock instructions respect a global total order. So that everybody sees that this thread, or processor, got a lock before that one. You don't have two different processors disagreeing on what the order was that somebody acquired a lock or whatever.

And then, memory ordering preserves transitive visibility, which is sort of like saying it obeys causality. In other words, if after doing a, if you had some effect, and then you did b, it should look like to other people like a and then b happened. Like there's a causality going on. But that's not sequential consistency, mainly because of four here.

So what's the impact of reordering? So here, we have our example from the beginning for the memory bottle, where I'm storing a 1 into a and then loading whatever is in b. And similarly, over here the opposite. So what happens if I'm allowed to do reordering? What can happen to these two instructions?

Yeah. They can execute in the opposite order. Similarly, these two guys can execute in the opposite order. So they can actually execute in this order where we do the load and then the stores. So it executes as if this were the order. Did I do this right? Executes as if this were the order. So I could do 1, 2, 3, 4.

So if then, I do the ordering 2, 4, 1, 3.

AUDIENCE: [INAUDIBLE]

PROFESSOR: I got the screwed up, I think. Didn't I?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Because I should be swapping these guys, right?

AUDIENCE: Swapped the wrong [INAUDIBLE].

PROFESSOR: Ugh. OK. So if I did this one 2, 1, 4, 3. So ignore this thing. Suppose I do the order 2. So basically, I load b. Then, I load a. Then, I store a. And then, I store b. What's the result value that are in EAX and EBX? You get 00.

Remember 00 wasn't the legal value from sequential consistency. But in this case, the Intel architecture and many other architectures out there will give you the wrong value for the execution of these instructions. Any question about that? So it doesn't preserve sequential consistency. So that's kind of scary in some way because you got to reason about this.

Let's see what happens in Peterson's algorithm if you don't have sequential consistency. So here we go. We have the code where she wants is true, turn is his, et cetera. How is this going to fail? What could happen here? Where will the bug arise? What's going to happen? What's the reordering that might happen?

AUDIENCE: On the while you do loads, right? [INAUDIBLE] the he_wants and turn is.

PROFESSOR: Sorry?

AUDIENCE: On the while statement, you do a load, right? Because [INAUDIBLE].

PROFESSOR: Right. He_wants is a load.

AUDIENCE: And so that will get reordered.

PROFESSOR: Where could that be reordered to? That could be reordered all the way to the top. Similarly, this one can be reordered all the way to the top. So the loads could be ordered all the way to the top.

And now, what's going to happen is you're going to set that she_wants is true but get a value of he_wants that might be old. And so they won't see each other's values. And so then, both threads can now enter the critical section simultaneously. Yeah, Reid?

AUDIENCE: If you swap the order of the loads, does the [INAUDIBLE]?

PROFESSOR: If you swap the order of the loads--

AUDIENCE: If you swap-- put the turn equals his on the left, [INAUDIBLE] on the right. Because according to--

PROFESSOR: Put the turn equals his over here?

AUDIENCE: Because the he_wants can't cross the load.

PROFESSOR: Yeah, but that's not what you want to do.

AUDIENCE: Then you can't [INAUDIBLE].

PROFESSOR: The whole idea here is that when you're saying you want to do something, you give the other one a turn so that whoever ends up winning the race allows just one of them to go through. Yeah?

AUDIENCE: I think the point is that if you put turn equals his and he_wants--

PROFESSOR: You're saying this stuff here.

AUDIENCE: Swap those two [UNINTELLIGIBLE] turn equals his will not be reordered before the

store that--

PROFESSOR: You might be right. Let me think about that.

AUDIENCE: You both reorder the same [? word. ?]

AUDIENCE: But you just stored turn, right?

PROFESSOR: Yeah. So if do turn equals his-- I see what you're saying. Do this turn equals his. I was looking at this turn equals his.

AUDIENCE: You mean turn equals equals his.

AUDIENCE: So the Boolean expression [INAUDIBLE].

PROFESSOR: Yeah. OK, I hadn't thought about that. Let me just think about that a second. So if we do the turn equals his--

AUDIENCE: [INAUDIBLE] and you won't reorder those two [INAUDIBLE]?

PROFESSOR: Then the-- Yeah. You got to be-- I have to think about that. I don't know about you folks, but I find this stuff really hard to think about. And so do most people, I think. This is one of these things where I don't think I can do without sitting down for 10 minutes and thinking about it deeply. But it's an interesting thought that if you did it the other direction that maybe there would be a requirement there.

I'm skeptical that that is true because to my knowledge to do the mutual exclusion, you pretty much have to do what I'm going to talk about next. But it would be interesting if is true. Because you also have to worry about this guy getting reordered with respect to this one.

AUDIENCE: The loads can't be reordered with respect to each other.

PROFESSOR: So he_wants and turn equals his. Yeah. So the loads won't be reordered. Yeah. So that looks OK. And then, you're saying and then therefore, it can't go forward because this one won't get reordered with that one. You might be right. That'd be cute. So I have to update the slides for next year if that's true.

So one way out of this quandary is to use what's called a memory fence or memory barrier. And it's a hardware action that enforces an ordering constraint between the instructions before and after the fence. So a memory fence says don't allow the processor to reorder these things. So why would you not want to do a memory fence? Then we'll talk about why you do it. Yeah?

AUDIENCE: To force a hardware slowdown?

PROFESSOR: Yeah. You're forcing the hardware slowdown. You're also forcing compiler because the compiler has to respect that, too. You're not letting the compiler do optimizations across the fence. So generally, fences slow things down. In addition, it turns out that they have some significant overhead.

So you can issue a memory fence explicitly as an instruction. So the mfence instruction sets a memory fence. There's also, it turns out, on x86 an lfence and an sfence, which allow loads to go over but not stores and stores but not loads. And this one is basically both. From the point of view of what we're using it for, we're only going to worry about the fences.

They're done by the explicit one. But it also turns out all the locking instructions automatically put a fence in. One of the humorous things in recent memory is major manufacturers for whom the lock instruction was actually faster than doing a memory fence, which is kind of weird because a lock instruction does a memory fence.

So how do you think that sort of thing comes about? So when you looked at performance it would be like-- for this particular machine I'm thinking about-- it was 30 cycles to do a lock instruction. And it was on the order of 50 cycles to do a memory fence. And so if you want to do a memory fence, what should you do?

AUDIENCE: Do a lock.

PROFESSOR: Do a lock instruction instead to get the effect. But why do you suppose that came up in the hardware? Why is it that one instruction would be-- It's a social reason why

this sort of thing happens. So I don't know for sure. But I know enough about engineering to understand how these things come about.

So here's what goes on. They do studies of traces of programs. And how often do you think lock instructions occur? And how often do you think fence instructions occur? Turns out lock instructions occur all the time, whereas fences, they don't occur so often because usually it's somebody who really knows what they're doing who's using a memory fence.

So then, they say to the engineering team, we're going to make our code go faster. And lock instructions are going really fast. So they put a top engineer on making lock instructions go fast. They put a second-rate engineer on making memory fence operations go fast because they're not used as often, without sort of recognizing that, gee, what you do for one is the same problem. You can do the same thing for the other.

So it ends up you'll see things in architecture that are really quite humorous like that, where things are sort of like, wait a minute, how come this is slower when well, it probably has to do with the engineering team that built the system. And actually now I'm aware of two architectures where they did the same kind of thing by different manufacturers. Where they got these memory fences. It should be at least as fast because the one is doing-- anyway. Interesting story there.

Now, you can actually access a memory fence using a built in function called sync synchronize. And in fact, there whole set of atomics-- I've put the information here for where you can go and look at the atomic operations that include memory fences and so forth to using in the compiler. It turns out when I was trying to get this going last night, I couldn't get it to work.

And it turns out that's because our compiler had a bug where this instruction was compiling to nothing. There's a compiler bug. And so I messed around for far too much time and then finally sent out a help message to the T.A.s. And then, John figured out that there was a bug. And he's patched all the compilers so that you guys all have it. But anyway, it was like, how come this isn't working?

AUDIENCE: What compiler are we using?

PROFESSOR: This was GCC. I was trying 4 1, and I tried 4 3. And so the one that we're using in class for the most part, is 4 3. So anyway, John put the patch in. So now, when you use these instructions, they're all there.

And then, the last thing is that the typical cost of a memory fence operation is comparable to that of an L2 cache access. So memory fences tend to be on our machine-- and I haven't actually measured in our machine. I meant to do that, and I didn't get around to it. It's probably on the order of 10, or 15 cycles, or something, which is not bad. If it's less than 20, it's pretty good.

So here's Peterson's algorithm with memory fences. You just simply sticky in the memory fence there to prevent the reordering. And it's interesting if there's a way that we can play the game with the instruction stream to do the same thing because that would make this code go, generally, a lot faster in terms of overhead. And so using memory fences, you can restore consistency.

Now, memory fences are like data races. If you don't have them, how do you know that you don't have them. It's very difficult to regression test for them, which is one reason I think there was a bug in the GCC compiler. How do you know that some piece of code is failing because most of the time it will work correctly. It's just occasionally, they'll be some reordering, and timing, and race condition that causes it not to work out. In this case, you both have to have the race and the reordering happening at the same time for Peterson's algorithm, for example.

So compilers can be very difficult for things like this. Really, the way to do it, which is what I was doing, was do an objdump and search for is fence in there. And in this case, it wasn't in there.

AUDIENCE: And also compiler's self-analyzers, by itself. And that's this instruction that basically can take code.

PROFESSOR: Right. It's not doing anything. Right. So it says, oop, get out of it. Yep. Good.

So any questions about consistency. So what turns out to be most of the time when you're designing things where you want to synchronize through memory directly, rather than using locks or what have you. The methodology that I found works pretty well. Work it out for sequential consistency, and then figure out where you have to put the fences in.

And that's a pretty good methodology for working out where-- here's sequential consistency. Now, what reorderings do I need to ensure in order to make sure that it works properly. And that can be error prone. So once again, big skull and cross bones on whether you actually try this in practice. It really better make a difference.

Now, the fact that you can synchronize directly through memory has led to a lot of protocols that are called lock-free protocols, which have some advantages, even though, in particular, because they don't use locks. And so I want to illustrate some of those because you'll see these in certain places.

So recall the summing problem from last time. So here we have an array. And what we're going to do is run through all the elements in the array, computing something on every element, and adding into result. And we wanted to parallelize that. So we parallelize that with a Cilk 4.

And what was the problem when we parallelize this? We get a race. So there's the race. We get a race on result because we've got two parallel instructions both trying to update results at the same time.

So we can solve that with a lock. And I showed you last time that we could solve this for lock. By declaring a mutex, and then locking before we update the results, and then unlock. And of course, as we argued yesterday, that could cause severe contention.

Now, contention can be an issue. But if it turns out that the compute here, which I've moved outside the lock notice. I've put it into temp and then added temp in so I can lock for the shortest possible time. If this compute is sufficiently large, there may be contention. But it may not be a significant contention in your execution because the

update here could be very, very short compared with the time it takes to compute.

So for example, if computing on array i cost you more than say order n time, then the fact that you have contention there isn't going to matter, generally, because the total amount of time that you're going to be locking is just small compared to the total execution time. Still in a multiprogram setting, there may be other problems that you can get into, even when you have this and even if you think that contention is going to be minimal.

So can anybody think of what the issues might be? Why could this be problematic even if contention is not a big issue? And the hint here is it's in a multiprogram setting. So what happens in a multiprogram setting. Yeah.

AUDIENCE: [INAUDIBLE] PROFESSOR: Because the resolve is--

AUDIENCE: [INAUDIBLE PHRASE]

PROFESSOR: It actually doesn't have to do with resolve here. It has to do with locking explicitly. It's a problem with locking in a multiprogrammed environment. What happens in a multiprogrammed environment? What do I mean by multiprogrammed environment?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Have multiple jobs running, right? And what happens to the processor when there are multiple jobs running?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Contact switches. So now, what can go wrong here? What can be really bad here? Yeah.

AUDIENCE: You aquire the lock and then the contacts switch out.

PROFESSOR: Yeah. You acquire the lock. And then, the operating system contact switches you out. And so what happens? You hold the lock while some other job is running. And

what are those guys doing. They go and spin and wait on the lock.

Now, this is a good time where you'd rather not have a spinning lock. You'd rather have a yielding lock. But even so, suddenly you're talking about something that's operating at the level of 100 times a second, 10 milliseconds, versus something that is operating on a nanosecond level. So you're talking six orders of magnitude of performance difference if you end up getting switched out while you hold a lock. That's the issue. What happens.

And then, if that happens, all the other loop iterations must wait.

AUDIENCE: [INAUDIBLE] in the large program here [UNINTELLIGIBLE PHRASE]. I don't have a mic. If one [UNINTELLIGIBLE] crashes or one [UNINTELLIGIBLE PHRASE] the lock [UNINTELLIGIBLE PHRASE].

AUDIENCE: Can you specify whether those are yielding locks or spinning locks?

PROFESSOR: Usually, the mutex type will tell you. I'm just using a simple name of mutex. I probably should have been using the ones that-- we were using one called Cilk mutex. And I probably should've used that here rather than just simple mutex.

AUDIENCE: Are they yielding?

PROFESSOR: There's a good question. I used to know the answer to this. I believe that those spin for a while, are competitive. They spin for a while and then yield. But I'm not sure. They may just spin. They don't just automatically yield. They're either competitive, or they'll spin and yield. I believe they spin and yield. And I believe there's actually a switch where you can tell it-- if you're doing timing measurements-- make it so that it purely spins so that you can get better benchmark results.

AUDIENCE: So my question is does the colonel have power to switch out a spinning lock or not?

PROFESSOR: Yeah. Well, the colonel, the scheduler, can come in at any moment and say, whip you're out. You're out. That's it. And wherever it is, it interrupts it at that moment in time.

So one solution to this problem is to use a lock-free method. And one of the common ways of doing that is with what's called compare-and-swap instruction. So this is what's called a locking instruction, meaning it's one of these ones that goes out to L2, in terms of timing and so forth. And what it does is it does the following thing.

It has an address of a location. And it's got the old value that was stored in the location and a new value. And it says if the value that is there is the old value, well, then stick the new value in there. And then return essentially true. Otherwise return false. So it's basically saying what you tend to do is you first look to see what's the value. You then update the value. And then you say, if it hasn't changed, stick it back in and return true. If it has changed, return false. So it's only swaps the value if it is true.

There's actually two versions. One which says bool and one which says val. And if you do the bool version, it returns a flag. If you do the val version, it actually returns the value that was in there. So it's more like a compare-and-swap. The main thing about this is this code essentially executes atomically with the single instruction, which is called-- The instruction is cmpxchg. Is it up there? Oh, there it is.

Yeah. So the cmpxchg instruction on x86. So when you compile this, you should find on your assembly output that instruction somewhere unless the compiler figures out a better way to optimize that. But generally, you should find that.

Also, one of the things about this is it works on values that are sort of integer type values. But it doesn't work on floating point numbers, in particular. So you can't compare-and-swap a value, which is a floating point value. You can only do it with energy type values.

So let's take a look at how we can use the compare-and-swap for the summing problem. So what we do is we have the same sort of code. And now, what I'm going to do is compute my temporary value. And then, what I'll do is I'll read the value of result into old. I'll then update my new value for what I think I want the result to be the result plus the thing that I computed.

And now, what I do is I attempt to compare-and-swap as long as the old value is what I read it to be. Swap in the new value. If the old value turns out to be different from what is currently in the result location, then it returns false. And I redo this again. Then, I have to redo the whole loop again.

So this is a do-while loop. Do-while is like a while loop, except you do the body first. And then you test the condition. So if this fails, I go back. I then get a new value for the result and so forth. So let me show you how that works. Let's see. So first, I'll show you how this works. Actually, I'll show it on a more interesting example how this works.

So what happens if I get swapped out in the middle of a loop iteration? All I do is when I do the compare-and-swap it fails. So no other instructions can wait. They can all march ahead and do the thing they need to do. And then, the one that got swapped out, eh. It gets some old value. It discovers that and has to re-execute the loop.

So is that fine? So what this means is that the amount work that's going on, however, could in fact, be greater, depending upon how much contention there is. If there's a lot of contention, you could end up having these guys fighting and not re executing a lot of code. But that's really not much worse than them spinning is what it comes down to. Any questions?

Let's do a more interesting example. Here's a lock-free stack. So what we're going to do is we're going to have a node, which has a next pointer and some data. All we really care about is the next pointer. And we have a stack, which has basically a head pointer.

So we have a linked list here. We want to basically be able to insert things at the front and take things out of the front. So here's a lock-free push. So remember, this could be concurrent. So these guys want to operate on it at a time. We saw last time how in doing very simple updates on link structures, you could get yourself into a mess if you didn't properly synchronize when we did the insertion in the hash table.

So here's my push [? up ?] code. Well let's walk through it. It says, basically, here's my node that I want to insert. It says, first of all, make node.next point to the head. So we basically have it pointing to 77. So then what we say is OK. Let's compare-and-swap to make the head point to the node but only if the value of the head has not changed. It's still the value of the node.next. And if so, it does the swap. Question?

AUDIENCE: You say compare-and-swap. But you compare it to what?

PROFESSOR: In this case it's comparing to the-- so this is basically the location that you're doing the compare-and-swap on, the old value that you expect to see in that location, and the new value. So here, what it says-- when we're at this point here-- we're saying before you do the compare-and-swap, we're saying, I only want you to set that pointer to go to here if this value is still pointing to there.

So only move this here if this value is still 77. In other words, if somebody else came in-- well, I'll do an example in a second that shows what happens when we have concurrency, and one of them might fail. But if it is true, then it basically sets it. And now I'm home free.

So let's take a look at what happens when we have contention. So I have two guys. So 33 says, OK I'll come in. Let me set my next pointer to the head. But then comes 81. And it says, OK. Let me try to set my pointer to also be 77 because I look at what the head is, and that's where it's supposed to go.

So now, what happens is we do the compare-and-swap operation. And they both are going to try to do it. And one of them is going to, essentially, do it first because the hardware preserves that the compare-and-swaps, their locking operations, they will happen in some definite order.

So in this case, 81 got in there and did its compare-and-swap first. When it looked, 77 was still a value that it said. So it allowed that pointer to be changed. But now what happens when 33 tries. 33 tries to do the compare-and-swap. And the compare-and-swap fails because it's saying, I want to swap 33 in as long as the

value of head is the pointer to 70, the node was 77. The value is no longer the pointer to the node of 77. It's now the pointer to the value of the node with 81. So the compare-and-swap fails. People follow that?

And so what does 33 have to do? It's got to start again. So it goes back around the loop, and now it sets it to 81, which is now the head. And now, it can compare-and-swap in the value. And they both get in there perfectly well. Question?

AUDIENCE: What if there's [INAUDIBLE]? What if two nodes have--

PROFESSOR: Well, notice here, it's not looking at the value of the data. Nowhere does data appear here. It's actually looking at the address of this chunk of memory. There is a similar problem, which I will raise in just a moment. There is still a problem. Yeah, question?

AUDIENCE: So I'm confused about the interface. So you give it the address of where you want to compare the value of. And you're giving it what you're pointing at and--

PROFESSOR: And here's the value that I expect to be stored in this location. The value I expect to be in there is node dot next. So if I go back a couple things. Here.

Here, the guy says, the value I expect to be there is in this case. the address of this chunk of memory here. He expects the address of the node containing 77 is going to be in this location. It's not. What's in this location is the address of this chunk of memory.

But you're saying, if it's equal to this, then you can go ahead and do the swap. Otherwise you're going to fail. And the swap consists of now sticking this value into-- conditionally sticking it in there. So you either do it or you don't do it.

So let's now do a pop. So pop you can also do with things. So here, I'm going to want to extract an element. And what I'm going to do is create a current value that I want to make point to the element that gets eliminated.

So what I do is I say, well, the element that I want is that guy there. And now, what I want to do is make the head jump around and point to 94. So what I do is I say,

well, as long as the-- and I want to do that unless I get down to the fact that I have an empty list. So basically, I say, if the head still has the value of current-- so they're pointing to the same place-- then, I want to move in current arrow next. And then I'm done.

Otherwise, I want to set current to head, reset it, and go back to the beginning and try to pop again. And I'm going to keep doing that until I get my pop to succeed or until current points to nil. If it ended up at the end, then I don't want to keep popping if the list ended up being empty. So basically, it sets that one to jump over.

And now, once it's done that, I can go, and I can clean up, I can get rid of this pointer, et cetera. But nobody else who's coming in to use this link list, can see 15 now because I'm the only one with a pointer to it. So people understand that?

So where's the bug? Turns out this has a but after all that work. Each of these individually does what it's supposed to do. But here's the bug. And it's a famous problem because you see it all the time when people are synchronizing through memory with lock-free algorithms. It's called the ABA problem.

So here's the problem. And it's similar to what some people were concerned about earlier. So here's the ABA problem. Thread 1 begins to pop 15. So imagine that what it does is it sets its current there, and then it reads the value here, and starts to set the head here using the compare-and-swap.

But it doesn't complete the compare-and-swap yet. The compare-and-swap hasn't executed. It's simply gotten this value, and it's about to swap it here.

So then, thread 2 comes along. And it says, oh, I want to pop something as well. So it comes in. And it turns out it's faster, and manages to pop 15 off, and set up its pointers.

Now, what would normally happen here is if this completed, what would happen? The compare-and-swap instruction would discover that this pointer is no longer the pointer to the head. And so it would fail. We'd be all hunky dory. No problem.

But what could actually happen here? Thread 2 keeps going on. It says, oh, let me pop 94. So it does the same thing. So thread 1 is still stalled here, not having completed its compare-and-swap. It swaps 94.

Then, thread 2 goes on and says, oh, let's put 15 back on. So it puts 15 back on because after all, it had 15. So now, what happens here? Thread 1 now looks, and it now completes, and does its compare-and-swap, it resumes, splicing out 15, which it thinks it has. But it doesn't realize that other stuff has gone on. And now, we've got a mess.

So this is the ABA problem because what happened was we were checking to see whether the value was still the same value, the same chunk of memory. It got popped off. But it got popped back on. But now, it could be in any configuration. We don't know what it is. And now, the code is thinking, that oh, nothing happened. But in fact, something happened.

So it's ABA because basically, you've got 15 there. It goes away. Then, 15 comes back. Question?

AUDIENCE: Can you compare two things and then swap because that would solve this, right?

PROFESSOR: That's called a double compare-and-swap. And we'll talk about it in a second. So the classic way to solve this problem is to use versioning. So what you do is you pack a version number with each pointer in the same atomically updatable word.

So that when 15 comes back, you've got the pointer. But you also have a version on that pointer so that the value has to be the same as the version you had and not the value. What you do is you increment the version number every time the pointer is changed. So you just do an increment. But you do the compare-and-swap on the version number and the pointer at the same time.

Now, it turns out that some architectures actually have what's called a double compare-and-swap, which will do compare-and-swap on two distinct locations. And that simplifies things even more because it means you don't have to pack and make sure that things fit in one word. You can keep versioning elsewhere. And there are a

whole bunch of other places where you can, in fact, optimize and get even tighter code than you could if you have to pack.

So that's generally the way you solve this. And, of course, you can see this gets-- as I say, this week has been skull and cross bones lecture. It's appropriate it comes right after Halloween because really, you do not want to play these games unless you have to.

But you should know about them because you will find times where you need this, or you need to understand somebody's code that they've written in a lock-free way. Because remember lock-free has the nice property that hey, the operating system swaps something out, it just keeps running nice and jolly if it's correct.

So the other issue is that version numbers may need to be very large. So if you have a version number, how many bits to you assign to that version number. Well, 64 bits, that's no problem. You never run out of 64 bits. 2 to the 64th is a very, very, very big number. And you'll never run out of 2 to the 64th.

We did that calculation at the beginning of the term. How big did we say it was? It's pretty big, right? It's like this big. Or is it this big? My two-year-old is this big.

So anyway, it's pretty big. So is it bigger than-- no, it's not bigger than the number of particles in the universe, right? That's 10 to the 80th, which is much bigger than 2 to the 64th. But it's still a big number. I think it's like more than there are atoms in the earth or something. It's still pretty big. You never get through it if you calculate it. I think we calculated it and it was hundreds of years or whatever.

Anyway, it's a long time. Many, many, many years at the very fastest, updating with biggest supercomputers, and the most processors, et cetera. Never run out of 64 bits. 32 bits. Four billion. Maybe you run out. Maybe you don't.

So that's one of the issues. You have to say, well, how often do I have to do that. Really, you only have to worry about this. You can wraparound. But you've got to make sure that then you never have a situation where something could be swapped out for long enough that it would come back and bite you because you're coming

around and then eating your tail. And you've got to make sure you wouldn't have things overlap and get a [? thing. ?]

So that might be a risk you're willing to take. You can do an analysis and say, what are the odds my system crashes from this reason or from a different reason? That can be reasonable engineering trade-off.

So there's an alternative to compare-and-swap. One is the double compare-and-swap. Another one is some machines have what's called a load-linked, store conditional instruction.

What those are actually is a pair of instructions. One is load-linked. When you load-linked, it basically says, let's set a bit, essentially, in that word. And if that word ever changes when you do store conditional, it will fail.

So even if some other processor changes it to the exact same value, it's keeping track of whether anybody else wrote it using the memory consistency mechanism. The MSI type protocol that we talked about. It's using that kind of mechanism to make sure that if it changes. And so this is actually much more reliable as a mechanism. x86 does not have load-linked, store conditional. I'm not sure why. I don't know if there's a patent on it or what's going on. But they don't have it.

Final topic is reducers. So once again, recall the summing problem. In Cilk++, they have a mechanism called reducer hyperobjects, which lets you do an end run around some of these synchronization problems. And the basic idea behind it is we actually could code this fairly easily as we talked about last time by just doing divide and conquer on the array.

We add up the first half of the elements, add up the second half of the elements, when they return, add them together. But the problem is that coding that is a pain to do. So the hyper object mechanism sort of does that automatically for you.

What you can do is declare result to be an integer, which is going to have the operation add performed on it. And what happens then is you can just go ahead

and add the values up like this. And basically, what it does is essentially adds things locally and will combine them on an as needed basis. So you don't actually have to do any synchronization at all. In the end, you have to get the result by doing a get value.

So let me show you a little bit more what's going on in this situation. So the first thing here is we're saying result is a summing reducer over int. The updates are resolved automatically without races or contention because they're basically doing it by keeping local values and copying them. And then, at the end, you can get the underlying value.

So the way this works is that when you declare the variable, you're declaring it as a reducer over some associative operation, such as addition. So it only works cleanly if your operation is associative. And there are a lot of associative operations. Addition, multiplication, logical, and, list concatenation. I can concatenate two lists.

So what does associative mean? I think I have a slide on this in a minute. It means a times b times c. I can parenthesize it any way I want and get the same answer. Associative, right? It's not associative like associative memory or whatever.

So now, the individual strands in the computation can update x as if it were an ordinary non-local variable. But in fact, it's maintained as a set of different copies called views. The Cilk++ runtime system coordinates the views and combines them when appropriate. And when only one view remains, now you can get the actual value.

So for example, you may have a summing reducer where the actual value at this point in time is 89. But locally, each processor may only see a different value whose sum is 89. But locally, I could do something like increment this. And this guy can independently increment his view and has the effect that it increments whatever the total sum is. And then, the runtime system manages to combine everything at the end to make it be the value when there's no more parallelism associated with that reducer.

So here's the conceptual behavior. Imagine I have this code. I set x equal to 0. I then add 3. I then increment. I had 4, increments at 5. Fa da da da da. At the end, I get some value, which I don't think I put down.

Another way I could do this is the following. Let me do exactly the same here but with a local view that I'll call x1. For this set of operations, let me start a new view that I start out with the identity for addition, which is 0 and add those guys up. And then, at the end, let me add x1 and x2. It should give me the same answer if addition is associative. In particular, these now can operate in parallel with no races.

So if you don't actually look at the intermediate values-- if all I'm doing is updating them, but I'm not actually looking to see what the absolute value of the thing is, I should get the same answer at the end. The answer to the result is then determinant. It's not deterministic because it's going to get done in a different way with different memory state. But it's determinant, meaning the output answer is going to give you the same no matter how it executes, even if the resulting computation is nondeterministic.

So this is a way of encapsulating, if you will, nondeterminism. And it worked because addition is associative. It didn't matter which order I did it. And once again, I could have broken it here instead of there, and I still get the same answer. It doesn't matter.

So the idea is as these things are work stealing around. they're accumulating things locally but combining them in a way that maintains the invariant that the final value is going to be the sum. So there's a lot of other related work where people do reduction types of things, but they're all tied to specific control or data structures.

And the neat thing about the Cilk++ version is that it is not tied to anything. You can name it anywhere. You can write recursive programs. You can update locally your reducer wherever you want, and it figures out exactly how to combine them in order to get your final answer.

So the algebraic framework for this is that we have a monoid, which is a set, an

operator, and an identity, where the operator is an associative binary operator. And the identity is, in fact, the identity. So here are some examples.

Integers with plus and 0, the real numbers with times and 1, true and false, Booleans with and, where true is the identity, strings over some alphabet with concatenation, where the empty string is the identity. You can do MAX with minus infinity as the operation, and so forth. And you can come up with your own. It's easy to come up with examples of monoids.

So what we do in Cilk++ is we represent a monoid over a set t by a C++ class that inherits from this base class that's predefined for you, which is parameterized using templates with the types. So the set that we're going to use is, in fact, going to be a type.

And the member function reduced-- this monoid has to have a member function reduced that implements the binary operator times. And it also has an identity member function. So we set up the algebraic framework.

So here's, for example, how I could define a sum monoid. I inherit from the base with int, for example, here. And I define my reduced function. And it actually turns out to be important, you always do the right one into the left. Otherwise, you won't have it be associative. And then, you have an identity, which gives you in this case a new element, which is 0.

And so you can now define the reducer as so. You just say Cilk reducer, the sum monoid you've defined and x. And now, the local view of x can be accessed as x open close parenthesis.

Now, in the example I showed you, you didn't need to do the open close parenthesis. And the way you get rid of those open close parenthesis is you define a wrapper class. So it's generally inconvenient to replace every access with x over brown. That's one issue.

The other thing is accesses aren't safe. Nothing prevents a programmer from writing x times equals 2, even though the reducer was defined over plus. And that

will screw up the logic of this code if somewhere he's multiplying when, in fact, it's only supposed to be combined with addition.

So the way you solve that is with a wrapper class. You can do a wrapper class that will protect all of the operations inside and export things that you can just refer to the variable. And it will actually call that. For most of what you're doing, you probably don't need to write a wrapper class. You'll do fine just operating with the extra parentheses.

In addition, there's a whole bunch of commonly use reducers. Lists, appends, max, min, adds, an output stream, and some strings, and also you can roll your own using things. One issue with addition is that, in fact, this doesn't preserve-- for floating point addition-- does not preserve the same answer.

And the reason is because floating point numbers are not associative. If I had a to b and add that to c, I can get something different because of round off error from adding a to the result of b and c. So generally, floating point operations don't give you-- they'll give you something that is close enough for most things, but it's not actually associative. So you will get different answers.

A quick example. I'm sorry to run over a little bit here. I hope people have a couple minutes.

Here's a real world example. A company had a mechanical assembly represented a tree of assemblies down to individual parts. A pickup truck has all these parts and all of these extra subparts all the way down to some geometric description of what the part is. And what they want to do is the so-called collision detection problem, which has nothing to do with colliding autos.

What they're doing is saying, find collisions between the assembly and a target object. And that object might be something like a half space because they're computing a cutaway. Tell me all the things that fall within this. Or maybe, here's an engine compartment, and does the engine fit in with it?

So here's a code that does that. Basically, it does a recursive walk, where it looks to

see whether it's an internal node or a leaf. If it's a leaf, it says, oh, let me check to see whether the target collides with a particular element of the tree. And if so, add that object to the end of a list. So this is the standard a C++ library for putting something on the end of the list.

If it's an internal node, then go through all of the children recursively. And walk the children recursively. So basically, you're going to look through the whole tree. Does it intersect this particular object, x?

So how do we parallelize this? We can parallelize the recursion. We turn the 4 loop here into a Cilk 4. So it goes through all the children at the same time. They all can do their comparisons completely the same.

Oops, but we have a bug. What's the bug?

AUDIENCE: Is it push back?

PROFESSOR: Yeah. The push back here. We have a race here because they're all trying to push on to this output list at the same time.

So we could resolve it with a lock or whatever. But it turns out it's much better to resolve it with a-- so we could do this, right? But now, you've got lock contention. And also, the list ends up getting produced in a jumbled order.

So it turns out if you use a reducer, you declare this to be a reducer with list append. And what happens then is turns out list concatenation is associative. If I concatenate a to b, and then concatenate c, that's the same as concatenating a to the concatenation of b and c. And I can concatenate lists in constant time by keeping a pointer to the head and tail of each list.

So if you do that, and that turns out to be one of the built in functions, then, in fact, this code operates perfectly well with no contention and so forth. And in fact, produces the output in the same order as the original C++. It runs fast. And there's a little description of how it works.

The actual protocol is kind of tricky. And we'll put the paper-- let's make sure we get this paper up on the web. I think it was there from last year. So we should be able to find it. If you're interested in how the details work. Here's the important thing to know from a programmer point of view.

So typically, the cost-- it turns out the reduce operations you're only calling when there's actually a steal. It's actually a return from a steal. But since stealing occurs relatively infrequently the load balance, the number of times you actually do one of these reduce operations is small.

The most of the cost is actually accessing the reducer to do the updates. And it's never worse than a hash table lookup the way it's implemented. If the reducer is accessed several times within a region of code, the compiler can optimize the lookups using common subexpression elimination.

And in the common case, then, what happens is it basically has an access cost equal to one additional level of indirection, which is typically an L1 cache hit. So the overhead of actually updating one of these things is really just like an extra L1 cache hit for most of these things, for most of the time. If you have the case that you're accessing a reducer several times within the same block of code.

Otherwise, at the very worst, you have to actually do a hash table lookup. And that tends to be a little bit more like a function call overhead just in terms of order of magnitude. Sorry for running over.