JULIAN SHUN: Good afternoon, everyone. So let's get started. So today, we're going to be talking about races and parallelism. And you'll be doing a lot of parallel programming for the next homework assignment and project.

One thing I want to point out is that it's important to meet with your MITPOSSE as soon as possible, if you haven't done so already, since that's going to be part of the evaluation for the Project 1 grade. And if you have trouble reaching your MITPOSSE members, please contact your TA and also make a post on Piazza as soon as possible.

So as a reminder, let's look at the basics of Cilk. So we have cilk_spawn and cilk_sync statements. In Cilk, this was the code that we saw in last lecture, which computes the nth Fibonacci number. So when we say cilk_spawn, it means that the named child function, the function right after the cilk_spawn keyword, can execute in parallel with the parent caller. So it says that fib of n minus 1 can execute in parallel with the fib function that called it.

And then cilk_sync says that control cannot pass this point until all of this spawned children have returned. So this is going to wait for fib of n minus 1 to finish before it goes on and returns the sum of x and y.

And recall that the Cilk keywords grant permission for parallel execution, but they don't actually force parallel execution. So this code here says that we can execute fib of n minus 1 in parallel with this parent caller, but it doesn't say that we necessarily have to execute them in parallel. And it's up to the runtime system to decide whether these different functions will be executed in parallel. We'll talk more about the runtime system today.

And also, we talked about this example, where we wanted to do an in-place matrix transpose. And this used the cilk_for keyword. And this says that we can execute the iterations of this cilk_for loop in parallel.

And again, this says that the runtime system is allowed to schedule these iterations in parallel,

but doesn't necessarily say that they have to execute in parallel. And under the hood, cilk_for statements are translated into nested cilk_spawn and cilk_sync calls. So the compiler is going to divide the iteration space in half, do a cilk_spawn on one of the two halves, call the other half, and then this is done recursively until we reach a certain size for the number of iterations in a loop, at which point it just creates a single task for that.

So any questions on the Cilk constructs? Yes?

AUDIENCE: Is Cilk smart enough to recognize issues with reading and writing for matrix transpose?

JULIAN SHUN: So it's actually not going to figure out whether the iterations are independent for you. The programmer actually has to reason about that. But Cilk does have a nice tool, which we'll talk about, that will tell you which places your code might possibly be reading and writing the same memory location, and that allows you to localize any possible race bugs in your code. So we'll actually talk about races. But if you just compile this code, Cilk isn't going to know whether the iterations are independent.

So determinacy races-- so race conditions are the bane of concurrency. So you don't want to have race conditions in your code. And there are these two famous race bugs that cause disaster. So there is this Therac-25 radiation therapy machine, and there was a race condition in the software. And this led to three people being killed and many more being seriously injured. The North American blackout of 2003 was also caused by a race bug in the software, and this left 50 million people without power.

So these are very bad. And they're notoriously difficult to discover by conventional testing. So race bugs aren't going to appear every time you execute your program. And in fact, the hardest ones to find, which cause these events, are actually very rare events. So most of the times when you run your program, you're not going to see the race bug. Only very rarely will you see it.

So this makes it very hard to find these race bugs. And furthermore, when you see a race bug, it doesn't necessarily always happen in the same place in your code. So that makes it even harder.

So what is a race? So a determinacy race is one of the most basic forms of races. And a determinacy race occurs when two logically parallel instructions access the same memory location, and at least one of these instructions performs a write to that location. So let's look at

a simple example.

So in this code here, I'm first setting x equal to 0. And then I have a cilk_for loop with two iterations, and each of the two iterations are incrementing this variable x. And then at the end, I'm going to assert that x is equal to 2.

So there's actually a race in this program here. So in order to understand where the race occurs, let's look at the execution graph here. So I'm going to label each of these statements with a letter. The first statement, a, is just setting x equal to 0.

And then after that, we're actually going to have two parallel paths, because we have two iterations of this cilk_for loop, which can execute in parallel. And each of these paths are going to increment x by 1. And then finally, we're going to assert that x is equal to 2 at the end.

And this sort of graph is known as a dependency graph. It tells you what instructions have to finish before you execute the next instruction. So here it says that B and C must wait for A to execute before they proceed, but B and C can actually happen in parallel, because there is no dependency among them. And then D has to happen after B and C finish.

So to understand why there's a race bug here, we actually need to take a closer look at this dependency graph. So let's take a closer look. So when you run this code, x plus plus is actually going to be translated into three steps. So first, we're going to load the value of x into some processor's register, r1. And then we're going to increment r1, and then we're going to set x equal to the result of r1.

And the same thing for r2. We're going to load x into register r2, increment r2, and then set x equal to r2.

So here, we have a race, because both of these stores, x1 equal to r1 and x2 equal to r2, are actually writing to the same memory location. So let's look at one possible execution of this computation graph. And we're going to keep track of the values of x, r1 and r2.

So the first instruction we're going to execute is x equal to 0. So we just set x equal to 0, and everything's good so far. And then next, we can actually pick one of two instructions to execute, because both of these two instructions have their predecessors satisfied already. Their predecessors have already executed.

So let's say I pick r1 equal to x to execute. And this is going to place the value 0 into register

r1. Now I'm going to increment r1, so this changes the value in r1 to 1. Then now, let's say I execute r2 equal to x.

So that's going to read x, which has a value of 0. It's going to place the value of 0 into r2. It's going to increment r2. That's going to change that value to 1. And then now, let's say I write r2 back to x. So I'm going to place a value of 1 into x.

Then now, when I execute this instruction, x1 equal to r1, it's also placing a value of 1 into x. And then finally, when I do the assertion, this value here is not equal to 2, and that's wrong. Because if you executed this sequentially, you would get a value of 2 here.

And the reason-- as I said, the reason why this occurs is because we have multiple writes to the same shared memory location, which could execute in parallel. And one of the nasty things about this example here is that the race bug doesn't necessarily always occur. So does anyone see why this race bug doesn't necessarily always show up? Yes?

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Right. So the answer is because if one of these two branches executes all three of its instructions before we start the other one, then the final result in x is going to be 2, which is correct. So if I executed these instructions in order of 1, 2, 3, 7, 4, 5, 6, and then, finally, 8, the value is going to be 2 in x. So the race bug here doesn't necessarily always occur. And this is one thing that makes these bugs hard to find.

So any questions?

So there are two different types of determinacy races. And they're shown in this table here. So let's suppose that instruction A and instruction B both access some location x, and suppose A is parallel to B. So both of the instructions can execute in parallel. So if A and B are just reading that location, then that's fine. You don't actually have a race here.

But if one of the two instructions is writing to that location, whereas the other one is reading to that location, then you have what's called a read race. And the program might have a non-deterministic result when you have a read race, because the final answer might depend on whether you read A first before B updated the value, or whether A read the updated value before B reads it. So the order of the execution of A and B can affect the final result that you see.

And finally, if both A and B write to the same shared location, then you have a write race. And again, this will cause non-deterministic behavior in your program, because the final answer could depend on whether A did the write first or B did the write first.

And we say that two sections of code are independent if there are no determinacy races between them. So the two pieces of code can't have a shared location, where one computation writes to it and another computation reads from it, or if both computations write to that location.

Any questions on the definition?

So races are really bad, and you should avoid having races in your program. So here are some tips on how to avoid races. So I can tell you not to write races in your program, and you know that races are bad, but sometimes, when you're writing code, you just have races in your program, and you can't help it. But here are some tips on how you can avoid races.

So first, the iterations of a cilk_for loop should be independent. So you should make sure that the different iterations of a cilk_for loop aren't writing to the same memory location.

Secondly, between a cilk_spawn statement and a corresponding cilk_sync, the code of the spawn child should be independent of the code of the parent. And this includes code that's executed by additional spawned or called children by the spawned child. So you should make sure that these pieces of code are independent-- there's no read or write races between them.

One thing to note is that the arguments to a spawn function are evaluated in the parent before the spawn actually occurs. So you can't get a race in the argument evaluation, because the parent is going to evaluate these arguments. And there's only one thread that's doing this, so it's fine. And another thing to note is that the machine word size matters. So you need to watch out for races when you're reading and writing to packed data structures.

So here's an example. I have a struct x with two chars, a and b. And updating x.a and x.b may possibly cause a race. And this is a nasty race, because it depends on the compiler optimization level. Fortunately, this is safe on the Intel machines that we're using in this class. You can't get a race in this example. But there are other architectures that might have a race when you're updating the two variables a and b in this case.

So with the Intel machines that we're using, if you're using standard data types like chars,

shorts, ints, and longs inside a struct, you won't get races. But if you're using non-standard types-- for example, you're using the C bit fields facilities, and the sizes of the fields are not one of the standard sizes, then you could possibly get a race. In particular, if you're updating individual bits inside a word in parallel, then you might see a race there. So you need to be careful.

Questions?

So fortunately, the Cilk platform has a very nice tool called the-- yes, question?

AUDIENCE: [INAUDIBLE] was going to ask, what causes that race?

JULIAN SHUN: Because the architecture might actually be updating this struct at the granularity of more than 1 byte. So if you're updating single bytes inside this larger word, then that might cause a race. But fortunately, this doesn't happen on Intel machines.

So the Cilksan race detector-- if you compile your code using this flag, minus f sanitize equal to cilk, then it's going to generate a Cilksan instrumentive program. And then if an ostensibly deterministic Cilk program run on a given input could possibly behave any differently than its serial elision, then Cilksan is going to guarantee to report and localize the offending race. So Cilksan is going to tell you which memory location there might be a race on and which of the instructions were involved in this race.

So Cilksan employs a regression test methodology where the programmer provides it different test inputs. And for each test input, if there could possibly be a race in the program, then it will report these races. And it identifies the file names, the lines, the variables involved in the races, including the stack traces. So it's very helpful when you're trying to debug your code and find out where there's a race in your program.

One thing to note is that you should ensure that all of your program files are instrumented. Because if you only instrument some of your files and not the other ones, then you'll possibly miss out on some of these race bugs.

And one of the nice things about the Cilksan race detector is that it's always going to report a race if there is possibly a race, unlike many other race detectors, which are best efforts. So they might report a race some of the times when the race actually occurs, but they don't necessarily report a race all the time. Because in some executions, the race doesn't occur. But the Cilksan race detector is going to always report the race, if there is potentially a race in

there.

Cilksan is your best friend. So use this when you're debugging your homeworks and projects.

Here's an example of the output that's generated by Cilksan. So you can see that it's saying that there's a race detected at this memory address here. And the line of code that caused this race is shown here, as well as the file name. So this is a matrix multiplication example. And then it also tells you how many races it detected.

So any questions on determinacy races?

So let's now talk about parallelism. So what is parallelism? Can we quantitatively define what parallelism is? So what does it mean when somebody tells you that their code is highly parallel?

So to have a formal definition of parallelism, we first need to look at the Cilk execution model. So this is a code that we saw before for Fibonacci. Let's now look at what a call to fib of 4 looks like. So here, I've color coded the different lines of code here so that I can refer to them when I'm drawing this computation graph.

So now, I'm going to draw this computation graph corresponding to how the computation unfolds during execution. So the first thing I'm going to do is I'm going to call fib of 4. And that's going to generate this magenta node here corresponding to the call to fib of 4, and that's going to represent this pink code here.

And this illustration is similar to the computation graphs that you saw in the previous lecture, but this is happening in parallel. And I'm only labeling the argument here, but you could actually also write the local variables there. But I didn't do it, because I want to fit everything on this slide.

So what happens when you call fib of 4? It's going to get to this cilk_spawn statement, and then it's going to call fib of 3. And when I get to a cilk_spawn statement, what I do is I'm going to create another node that corresponds to the child that I spawned. So this is this magenta node here in this blue box. And then I also have a continue edge going to a green node that represents the computation after the cilk_spawn statement. So this green node here corresponds to the green line of code in the code snippet.

Now I can unfold this computation graph one more step. So we see that fib 3 is going to call fib

of 2, so I created another node here. And the green node here, which corresponds to this green line of code-- it's also going to make a function call. It's going to call fib of 2. And that's also going to create a new node.

So in general, when I do a spawn, I'm going to have two outgoing edges out of a magenta node. And when I do a call, I'm going to have one outgoing edge out of a green node. So this green node, the outgoing edge corresponds to a function call. And for this magenta node, its first outgoing edge corresponds to spawn, and then its second outgoing edge goes to the continuation strand.

So I can unfold this one more time. And here, I see that I'm creating some more spawns and calls to fib. And if I do this one more time, I've actually reached the base case. Because once n is equal to 1 or 0, I'm not going to make any more recursive calls.

And by the way, the color of these boxes that I used here correspond to whether I called that function or whether I spawned it. So a box with white background corresponds to a function that I called, whereas a box with blue background corresponds to a function that I spawned.

So now I've gotten to the base case, I need to now execute this blue statement, which sums up x and y and returns the result to the parent caller. So here I have a blue node. So this is going to take the results of the two recursive calls, sum them together.

And I have another blue node here. And then it's going to pass its value to the parent that called it. So I'm going to pass this up to its parent, and then I'm going to pass this one up as well. And finally, I have a blue node at the top level, which is going to compute my final result, and that's going to be the output of the program.

So one thing to note is that this computation dag unfolds dynamically during the execution. So the runtime system isn't going to create this graph at the beginning. It's actually going to create this on the fly as you run the program. So this graph here unfolds dynamically. And also, this graph here is processor-oblivious. So nowhere in this computation dag did I mention the number of processors I had for the computation.

And similarly, in the code here, I never mentioned the number of processors that I'm using. So the runtime system is going to figure out how to map these tasks to the number of processors that you give to the computation dynamically at runtime. So for example, I can run this on any number of processors. If I run it on one processor, it's just going to execute these tasks in

parallel.

In fact, it's going to execute them in a depth-first order, which corresponds to the what the sequential algorithm would do. So I'm going to start with fib of 4, go to fib of 3, fib of 2, fib of 1, and go pop back up and then do fib of 0 and go back up and so on. So if I use one processor, it's going to create and execute this computation dag in the depth-first manner. And if I have more than one processor, it's not necessarily going to follow a depth-first order, because I could have multiple computations going on.

Any questions on this example? I'm actually going to formally define some terms on the next slide so that we can formalize the notion of a computation dag. So dag stands for directed acyclic graph, and this is a directed acyclic graph. So we call it a computation dag.

So a parallel instruction stream is a dag G with vertices V and edges E. And each vertex in this dag corresponds to a strand. And a strand is a sequence of instructions not containing a spawn, a sync, or a return from a spawn. So the instructions inside a strand are executed sequentially. There's no parallelism within a strand.

We call the first strand the initial strand, so this is the magenta node up here. The last strand-- we call it the final strand. And then everything else, we just call it a strand.

And then there are four types of edges. So there are spawn edges, call edges, return edges, or continue edges. And a spawn edge corresponds to an edge to a function that you spawned. So these spawn edges are going to go to a magenta node.

A call edge corresponds to an edge that goes to a function that you called. So in this example, these are coming out of the green nodes and going to a magenta node.

A return edge corresponds to an edge going back up to the parent caller. So here, it's going into one of these blue nodes.

And then finally, a continue edge is just the other edge when you spawn a function. So this is the edge that goes to the green node. It's representing the computation after you spawn something.

And notice that in this computation dag, we never explicitly represented cilk_for, because as I said before, cilk_fors are converted to nested cilk_spawns and cilk_sync statements. So we don't actually need to explicitly represent cilk_fors in the computation DAG.

Any questions on this definition? So we're going to be using this computation dag throughout this lecture to analyze how much parallelism there is in a program.

So assuming that each of these strands executes in unit time-- this assumption isn't always true in practice. In practice, strands will take different amounts of time. But let's assume, for simplicity, that each strand here takes unit time. Does anyone want to guess what the parallelism of this computation is? So how parallel do you think this is? What's the maximum speedup you might get on this computation?

AUDIENCE: 5.

JULIAN SHUN: 5. Somebody said 5. Any other guesses? Who thinks this is going to be less than five? A couple people. Who thinks it's going to be more than five? A couple of people. Who thinks there's any parallelism at all in this computation?

Yeah, seems like a lot of people think there is some parallelism here. So we're actually going to analyze how much parallelism is in this computation. So I'm not going to tell you the answer now, but I'll tell you in a couple of slides. First need to go over some terminology.

So whenever you start talking about parallelism, somebody is almost always going to bring up Amdahl's Law. And Amdahl's Law says that if 50% of your application is parallel and the other 50% is serial, then you can't get more than a factor of 2 speedup, no matter how many processors you run the computation on. Does anyone know why this is the case? Yes?

AUDIENCE: Because you need it to execute for at least 50% of the time in order to get through the serial portion.

JULIAN SHUN: Right. So you have to spend at least 50% of the time in the serial portion. So in the best case, if I gave you an infinite number of processors, and you can reduce the parallel portion of your code to 0 running time, you still have the 50% of the serial time that you have to execute. And therefore, the best speedup you can get is a factor of 2.

And in general, if a fraction alpha of an application must be run serially, then the speedup can be at most 1 over alpha. So if 1/3 of your program has to be executed sequentially, then the speedup can be, at most, 3. Because even if you reduce the parallel portion of your code to tab a running time of 0, you still have the sequential part of your code that you have to wait for.

So let's try to quantify the parallelism in this computation here. So how many of these nodes have to be executed sequentially? Yes?

AUDIENCE: 9 of them.

JULIAN SHUN: So it turns out to be less than 9. Yes?

AUDIENCE: 7.

JULIAN SHUN: 7. It turns out to be less than 7. Yes?

AUDIENCE: 6.

JULIAN SHUN: So it turns out to be less than 6.

AUDIENCE: 4.

JULIAN SHUN: Turns out to be less than 4. You're getting close.

AUDIENCE: 2.

JULIAN SHUN: 2. So turns out to be more than 2.

AUDIENCE: 2.5.

JULIAN SHUN: What's left?

AUDIENCE: 3.

JULIAN SHUN: 3. OK. So 3 of these nodes have to be executed sequentially. Because when you're executing these nodes, there's nothing else that can happen in parallel. For all of the remaining nodes, when you're executing them, you can potentially be executing some of the other nodes in parallel. But for these three nodes that I've colored in yellow, you have to execute those sequentially, because there's nothing else that's going on in parallel.

So according to Amdahl's Law, this says that the serial fraction of the program is 3 over 18. So there's 18 nodes in this graph here. So therefore, the serial factor is 1 over 6, and the speedup is upper bound by 1 over that, which is 6. So Amdahl's Law tells us that the maximum speedup we can get is 6. Any questions on how I got this number here?

So it turns out that Amdahl's Law actually gives us a pretty loose upper bound on the

parallelism, and it's not that useful in many practical cases. So we're actually going to look at a better definition of parallelism that will give us a better upper bound on the maximum speedup we can get.

So we're going to define T sub P to be the execution time of the program on P processors. And T sub 1 is just the work. So T sub 1 is if you executed this program on one processor, how much stuff do you have to do? And we define that to be the work. Recall in lecture 2, we looked at many ways to optimize the work. This is the work term.

So in this example, the number of nodes here is 18, so the work is just going to be 18. We also define T of infinity to be the span. The span is also called the critical path length, or the computational depth, of the graph. And this is equal to the longest directed path you can find in this graph.

So in this example, the longest path is 9. So one of the students answered 9 earlier, and this is actually the span of this graph. So there are 9 nodes along this path here, and that's the longest one you can find. And we call this T of infinity because that's actually the execution time of this program if you had an infinite number of processors.

So there are two laws that are going to relate these quantities. So the work law says that T sub P is greater than or equal to T sub 1 divided by P. So this says that the execution time on P processors has to be greater than or equal to the work of the program divided by the number of processors you have. Does anyone see why the work law is true?

So the answer is that if you have P processors, on each time stub, you can do, at most, P work. So if you multiply both sides by P, you get P times T sub P is greater than or equal to T1. If P times T sub P was less than T1, then that means you're not done with the computation, because you haven't done all the work yet. So the work law says that T sub P has to be greater than or equal to T1 over P.

Any questions on the work law?

So let's look at another law. This is called the span law. It says that T sub P has to be greater than or equal to T sub infinity. So the execution time on P processors has to be at least execution time on an infinite number of processors. Anyone know why the span law has to be true?

So another way to see this is that if you had an infinite number of processors, you can actually

simulate a P processor system. You just use P of the processors and leave all the remaining processors idle. And that can't slow down your program. So therefore, you have that T sub P has to be greater than or equal to T sub infinity. If you add more processors to it, the running time can't go up.

Any questions?

So let's see how we can compose the work and the span quantities of different computations. So let's say I have two computations, A and B. And let's say that A has to execute before B. So everything in A has to be done before I start the computation in B. Let's say I know what the work of A and the work of B individually are. What would be the work of A union B? Yes?

AUDIENCE: I guess it would be T1 A plus T1 B.

JULIAN SHUN: Yeah. So why is that?

AUDIENCE: Well, you have to execute sequentially. So then you just take the time and [INAUDIBLE] execute A, then it'll execute B after that.

JULIAN SHUN: Yeah. So the work is just going to be the sum of the work of A and the work of B. Because you have to do all of the work of A and then do all of the work of B, so you just add them together.

What about the span? So let's say I know the span of A and I know the span of B. What's the span of A union B? So again, it's just a sum of the span of A and the span of B. This is because I have to execute everything in A before I start B. So I just sum together the spans.

So this is series composition. What if I do parallel composition? So let's say here, I'm executing the two computations in parallel. What's the work of A union B?

So it's not it's not going to be the maximum. Yes?

AUDIENCE: It should still be T1 of A plus T1 of B.

JULIAN SHUN: Yeah, so it's still going to be the sum of T1 of A and T1 of B. Because you still have the same amount of work that you have to do. It's just that you're doing it in parallel. But the work is just the time if you had one processor. So if you had one processor, you wouldn't be executing these in parallel.

What about the span? So if I know the span of A and the span of B, what's the span of the

parallel composition of the two? Yes?

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Yeah, so the span of A union B is going to be the max of the span of A and the span of B, because I'm going to be bottlenecked by the slower of the two computations. So I just take the one that has longer span, and that gives me the overall span. Any questions?

So here's another definition. So T1 divided by TP is the speedup on P processors. If I have T1 divided by TP less than P, then this means that I have sub-linear speedup. I'm not making use of all the processors. Because I'm using P processors, but I'm not getting a speedup of P.

If T1 over TP is equal to P, then I'm getting perfect linear speedup. I'm making use of all of my processors. I'm putting P times as many resources into my computation, and it becomes P times faster. So this is the good case.

And finally, if T1 over TP is greater than P, we have something called superlinear speedup. In our simple performance model, this can't actually happen, because of the work law. The work law says that TP has to be at least T1 divided by P. So if you rearrange the terms, you'll see that we get a contradiction in our model.

In practice, you might sometimes see that you have a superlinear speedup, because when you're using more processors, you might have access to more cache, and that could improve the performance of your program. But in general, you might see a little bit of superlinear speedup, but not that much. And in our simplified model, we're just going to assume that you can't have a superlinear speedup. And getting perfect linear speedup is already very good.

So because the span law says that TP has to be at least T infinity, the maximum possible speedup is just going to be T1 divided by T infinity, and that's the parallelism of your computation. This is a maximum possible speedup you can get.

Another way to view this is that it's equal to the average amount of work that you have to do per step along the span. So for every step along the span, you're doing this much work. And after all the steps, then you've done all of the work.

So what's the parallelism of this computation dag here?

AUDIENCE: 2.

JULIAN SHUN: 2. Why is it 2?

AUDIENCE: T1 is 18 and T infinity is 9.

JULIAN SHUN: Yeah. So T1 is 18. There are 18 nodes in this graph. T infinity is 9. And the last time I checked, 18 divided by 9 is 2. So the parallelism here is 2.

So now we can go back to our Fibonacci example, and we can also analyze the work and the span of this and compute the maximum parallelism. So again, for simplicity, let's assume that each of these strands takes unit time to execute. Again, in practice, that's not necessarily true. But for simplicity, let's just assume that. So what's the work of this computation?

AUDIENCE: 17.

JULIAN SHUN: 17. Right. So the work is just the number of nodes you have in this graph. And you can just count that up, and you get 17.

What about the span? Somebody said 8. Yeah, so the span is 8. And here's the longest path. So this is the path that has 8 nodes in it, and that's the longest one you can find here.

So therefore, the parallelism is just 17 divided by 8, which is 2.125. And so for all of you who guessed that the parallelism was 2, you were very close.

This tells us that using many more than two processors can only yield us marginal performance gains. Because the maximum speedup we can get is 2.125. So we throw eight processors at this computation, we're not going to get a speedup beyond 2.125.

So to figure out how much parallelism is in your computation, you need to analyze the work of your computation and the span of your computation and then take the ratio between the two quantities. But for large computations, it's actually pretty tedious to analyze this by hand. You don't want to draw these things out by hand for a very large computation.

And fortunately, Cilk has a tool called the Cilkscale Scalability Analyzer. So this is integrated into the Tapir/LLVM compiler that you'll be using for this course. And Cilkscale uses compiler instrumentation to analyze a serial execution of a program, and it's going to generate the work and the span quantities and then use those quantities to derive upper bounds on the parallel speedup of your program. So you'll have a chance to play around with Cilkscale in homework 4.

So let's try to analyze the parallelism of quicksort. And here, we're using a parallel quicksort algorithm. The function quicksort here takes two inputs. These are two pointers. Left points to the beginning of the array that we want to sort. Right points to one element after the end of the array. And what we do is we first check if left is equal to right. If so, then we just return, because there are no elements to sort.

Otherwise, we're going to call this partition function. The partition function is going to pick a random pivot-- so this is a randomized quicksort algorithm-- and then it's going to move everything that's less than the pivot to the left part of the array and everything that's greater than or equal to the pivot to the right part of the array. It's also going to return us a pointer to the pivot.

And then now we can execute two recursive calls. So we do quicksort on the left side and quicksort on the right side. And this can happen in parallel. So we use the cilk_spawn here to spawn off one of these calls to quicksort in parallel. And therefore, the two recursive calls are parallel. And then finally, we sync up before we return from the function.

So let's say we wanted to sort 1 million numbers with this quicksort algorithm. And let's also assume that the partition function here is written sequentially, so you have to go through all of the elements, one by one. Can anyone guess what the parallelism is in this computation?

AUDIENCE: 1 million.

JULIAN SHUN: So the guess was 1 million. Any other guesses?

AUDIENCE: 50,000.

JULIAN SHUN: 50,000. Any other guesses? Yes?

AUDIENCE: 2.

JULIAN SHUN: 2. It's a good guess.

AUDIENCE: Log 2 of a million.

JULIAN SHUN: Log base 2 of a million. Any other guesses? So log base 2 of a million, 2, 50,000, and 1 million. Anyone think it's more than 1 million? No. So no takers on more than 1 million.

So if you run this program using Cilkscale, it will generate a plot that looks like this. And there

are several lines on this plot. So let's talk about what each of these lines mean.

So this purple line here is the speedup that you observe in your computation when you're running it. And you can get that by taking the single processor running time and dividing it by the running time on P processors. So this is the observed speedup. That's the purple line.

The blue line here is the line that you get from the span law. So this is T1 over T infinity. And here, this gives us a bound of about 6 for the parallelism.

The green line is the bound from the work law. So this is just a linear line with a slope of 1. It says that on P processors, you can't get more than a factor of P speedup.

So therefore, the maximum speedup you can get has to be below the green line and below the blue line. So you're in this lower right quadrant of the plot.

There's also this orange line, which is the speedup you would get if you used a greedy scheduler. We'll talk more about the greedy scheduler later on in this lecture.

So this is the plot that you would get. And we see here that the maximum speedup is about 5. So for those of you who guessed 2 and log base 2 of a million, you were the closest.

You can also generate a plot that just tells you the execution time versus the number of processors. And you can get this quite easily just by doing a simple transformation from the previous plot.

So Cilkscale is going to give you these useful plots that you can use to figure out how much parallelism is in your program. And let's see why the parallelism here is so low.

So I said that we were going to execute this partition function sequentially, and it turns out that that's actually the bottleneck to the parallelism. So the expected work of quicksort is order n log n. So some of you might have seen this in your previous algorithms courses. If you haven't seen this yet, then you can take a look at your favorite textbook, *Introduction to Algorithms.* It turns out that the parallel version of quicksort also has an expected work bound of order n log n, if you pick a random pivot. So the analysis is similar.

The expected span bound turns out to be at least n. And this is because on the first level of recursion, we have to call this partition function, which is going to go through the elements one by one. So that already has a linear span. And it turns out that the overall span is also order n,

because the span actually works out to be a geometrically decreasing sequence and sums to order n.

And therefore, the maximum parallelism you can get is order log n. So you just take the work divided by the span. So for the student who guessed that the parallelism is log base 2 of n, that's very good. Turns out that it's not exactly log base 2 of n, because there are constants in these work and span bounds, so it's on the order of log of n. That's the parallelism.

And it turns out that order log n parallelism is not very high. In general, you want the parallelism to be much higher, something polynomial in n.

And in order to get more parallelism in this algorithm, what you have to do is you have to parallelize this partition function, because right now I I'm just executing this sequentially. But you can actually indeed write a parallel partition function that takes linear your work in order log n span. And then this would give you an overall span bound of log squared n. And then if you take n log n divided by log squared n, that gives you an overall parallelism of n over log n, which is much higher than order log n here.

And similarly, if you were to implement a merge sort, you would also need to make sure that the merging routine is implemented in parallel, if you want to see significant speedup. So not only do you have to execute the two recursive calls in parallel, you also need to make sure that the merging portion of the code is done in parallel. Any questions on this example?

AUDIENCE: In the graph that you had, sometimes when you got to higher processor numbers, it got jagged, and so sometimes adding a processor was making it slower. What are some reasons [INAUDIBLE]?

JULIAN SHUN: Yeah so I believe that's just due to noise, because there's some noise going on in the machine. So if you ran it enough times and took the average or the median, it should be always going up, or it shouldn't be decreasing, at least. Yes?

AUDIENCE: So [INAUDIBLE] is also [INAUDIBLE]?

JULIAN SHUN: So at one level of recursion, the partition function takes order log n span. You can show that there are log n levels of recursion in this quicksort algorithm. I didn't go over the details of this analysis, but you can show that. And then therefore, the overall span is going to be order log squared. And I can show you on the board after class, if you're interested, or I can give you a reference. Other questions?

So it turns out that in addition to quicksort, there are also many other interesting practical parallel algorithms out there. So here, I've listed a few of them. And by practical, I mean that the Cilk program running on one processor is competitive with the best sequential program for that problem.

And so you can see that I've listed the work and the span of merge sort here. And if you implement the merge and parallel, the span of the overall computation would be log cubed n. And log n divided by log cubed n is n over log squared n. That's the parallelism, which is pretty high. And in general, all of these computations have pretty high parallelism.

Another thing to note is that these algorithms are practical, because their work bound is asymptotically equal to the work of the corresponding sequential algorithm. That's known as a work-efficient parallel algorithm. It's actually one of the goals of parallel algorithm design, to come up with work-efficient parallel algorithms. Because this means that even if you have a small number of processors, you can still be competitive with a sequential algorithm running on one processor. And in the next lecture, we actually see some examples of these other algorithms, and possibly even ones not listed on this slide, and we'll go over the work and span analysis and figure out the parallelism.

So now I want to move on to talk about some scheduling theory. So I talked about these computation dags earlier, analyzed the work and the span of them, but I never talked about how these different strands are actually mapped to processors at running time. So let's talk a little bit about scheduling theory. And it turns out that scheduling theory is actually very general. It's not just limited to parallel programming. It's used all over the place in computer science, operations research, and math.

So as a reminder, Cilk allows the program to express potential parallelism in an application. And a Cilk scheduler is going to map these strands onto the processors that you have available dynamically at runtime. Cilk actually uses a distributed scheduler.

But since the theory of distributed schedulers is a little bit complicated, we'll actually explore the ideas of scheduling first using a centralized scheduler. And a centralized scheduler knows everything about what's going on in the computation, and it can use that to make a good decision. So let's first look at what a centralized scheduler does, and then I'll talk a little bit about the Cilk distributed scheduler. And we'll learn more about that in a future lecture as well.

So we're going to look at a greedy scheduler. And an idea of a greedy scheduler is to just do as much as possible in every step of the computation. So has anyone seen greedy algorithms before? Right. So many of you have seen greedy algorithms before. So the idea is similar here. We're just going to do as much as possible at the current time step. We're not going to think too much about the future.

So we're going to define a ready strand to be a strand where all of its predecessors in the computation dag have already executed. So in this example here, let's say I already executed all of these blue strands. Then the ones shaded in yellow are going to be my ready strands, because they have all of their predecessors executed already.

And there are two types of steps in a greedy scheduler. The first kind of step is called a complete step. And in a complete step, we have at least P strands ready. So if we had P equal to 3, then we have a complete step now, because we have 5 strands ready, which is greater than 3.

So what are we going to do in a complete step? What would a greedy scheduler do? Yes?

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Yeah, so a greedy scheduler would just do as much as it can. So it would just run any 3 of these, or any P in general. So let's say I picked these 3 to run. So it turns out that these are actually the worst 3 to run, because they don't enable any new strands to be ready. But I can pick those 3.

And then the incomplete step is one where I have fewer than P strands ready. So here, I have 2 strands ready, and I have 3 processors. So what would I do in an incomplete step?

AUDIENCE: Just run through the strands that are ready.

JULIAN SHUN: Yeah, so just run all of them. So here, I'm going to execute these two strands. And then we're going to use complete steps and incomplete steps to analyze the performance of the greedy scheduler.

There's a famous theorem which was first shown by Ron Graham in 1968 that says that any greedy scheduler achieves the following time bound-- T sub P is less than or equal to T1 over P plus T infinity. And you might recognize the terms on the right hand side-- T1 is the work, and T infinity is the span that we saw earlier.

And here's a simple proof for why this time bound holds. So we can upper bound the number of complete steps in the computation by T1 over P. And this is because each complete step is going to perform P work. So after T1 over P completes steps, we'll have done all the work in our computation. So that means that the number of complete steps can be at most T1 over P.

So any questions on this?

So now, let's look at the number of incomplete steps we can have. So the number of incomplete steps we can have is upper bounded by the span, or T infinity. And the reason why is that if you look at the unexecuted dag right before you execute an incomplete step, and you measure the span of that unexecuted dag, you'll see that once you execute an incomplete step, it's going to reduce the span of that dag by 1.

So here, this is the span of our unexecuted dag that contains just these seven nodes. The span of this is 5. And when we execute an incomplete step, we're going to process all the roots of this unexecuted dag, delete them from the dag, and therefore, we're going to reduce the length of the longest path by 1. So when we execute an incomplete step, it decreases the span from 5 to 4.

And then the time bound up here, T sub P, is just upper bounded by the sum of these two types of steps. Because after you execute T1 over P complete steps and T infinity incomplete steps, you must have finished the entire computation.

So any questions?

A corollary of this theorem is that any greedy scheduler achieves within a factor of 2 of the optimal running time. So this is the optimal running time of a scheduler that knows everything and can predict the future and so on. So let's let TP star be the execution time produced by an optimal scheduler.

We know that TP star has to be at least the max of T1 over P and T infinity. This is due to the work and span laws. So it has to be at least a max of these two terms. Otherwise, we wouldn't have finished the computation.

So now we can take the inequality we had before for the greedy scheduler bound-- so TP is less than or equal to T1 over P plus T infinity. And this is upper bounded by 2 times the max of these two terms. So A plus B is upper bounded by 2 times the max of A and B.

And then now, the max of T1 over P and T infinity is just upper bounded by TP star. So we can substitute that in, and we get that TP is upper bounded by 2 times TP star, which is the running time of the optimal scheduler. So the greedy scheduler achieves within a factor of 2 of the optimal scheduler.

Here's another corollary. This is a more interesting corollary. It says that any greedy scheduler achieves near-perfect linear speedup whenever T1 divided by T infinity is greater than or equal to P.

To see why this is true-- if we have that T1 over T infinity is much greater than P-- so the double arrows here mean that the left hand side is much greater than the right hand side-- then this means that the span is much less than T1 over P. And the greedy scheduling theorem gives us that TP is less than or equal to T1 over P plus T infinity, but T infinity is much less than T1 over P, so the first term dominates, and we have that TP is approximately equal to T1 over P. And therefore, the speedup you get is T1 over P, which is P. And this is linear speedup.

The quantity T1 divided by P times T infinity is known as the parallel slackness. So this is basically measuring how much more parallelism you have in a computation than the number of processors you have. And if parallel slackness is very high, then this corollary is going to hold, and you're going to see near-linear speedup.

As a rule of thumb, you usually want the parallel slackness of your program to be at least 10. Because if you have a parallel slackness of just 1, you can't actually amortize the overheads of the scheduling mechanism. So therefore, you want the parallel slackness to be at least 10 when you're programming in Cilk.

So that was the greedy scheduler. Let's talk a little bit about the Cilk scheduler. So Cilk uses a work-stealing scheduler, and it achieves an expected running time of TP equal to T1 over P plus order T infinity. So instead of just summing the two terms, we actually have a big O in front of the T infinity, and this is used to account for the overheads of scheduling. The greedy scheduler I presented earlier-- I didn't account for any of the overheads of scheduling. I just assumed that it could figure out which of the tasks to execute.

So this Cilk work-stealing scheduler has this expected time provably, so you can prove this using random variables and tail bounds of distribution. So Charles Leiserson has a paper that

talks about how to prove this. And empirically, we usually see that TP is more like T1 over P plus T infinity. So we usually don't see any big constant in front of the T infinity term in practice. And therefore, we can get near-perfect linear speedup, as long as the number of processors is much less than T1 over T infinity, the maximum parallelism.

And as I said earlier, the instrumentation in Cilkscale will allow you to measure the work and span terms so that you can figure out how much parallelism is in your program.

Any questions?

So let's talk a little bit about how the Cilk runtime system works. So in the Cilk runtime system, each worker or processor maintains a work deque. Deque stands for double-ended queue, so it's just short for double-ended queue. It maintains a work deque of ready strands, and it manipulates the bottom of the deck, just like you would in a stack of a sequential program. So here, I have four processors, and each one of them have their own deques, and they have these things on the stack, these function calls, saves the return address to local variables, and so on.

So a processor can call a function, and when it calls a function, it just places that function's frame at the bottom of its stack. You can also spawn things, so then it places a spawn frame at the bottom of its stack. And then these things can happen in parallel, so multiple processes can be spawning and calling things in parallel.

And you can also return from a spawn or a call. So here, I'm going to return from a call. Then I return from a spawn. And at this point, I don't actually have anything left to do for the second processor. So what do I do now, when I'm left with nothing to do? Yes?

AUDIENCE: Take a [INAUDIBLE].

JULIAN SHUN: Yeah, so the idea here is to steal some work from another processor. So when a worker runs out of work to do, it's going to steal from the top of a random victim's deque. So it's going to pick one of these processors at random. It's going to roll some dice to determine who to steal from.

And let's say that it picked the third processor. Now it's going to take all of the stuff at the top of the deque up until the next spawn and place it into its own deque. And then now it has stuff to do again. So now it can continue executing this code. It can spawn stuff, call stuff, and so on.

So the idea is that whenever a worker runs out of work to do, it's going to start stealing some work from other processors. But if it always has enough work to do, then it's happy, and it doesn't need to steal things from other processors. And this is why MIT gives us so much work to do, so we don't have to steal work from other people.

So a famous theorem says that with sufficient parallelism, workers steal very infrequently, and this gives us near-linear speedup. So with sufficient parallelism, the first term in our running bound is going to dominate the T1 over P term, and that gives us near-linear speedup.

Let me actually show you a pseudoproof of this theorem. And I'm allowed to do a pseudoproof. It's not actually a real proof, but a pseudoproof. So I'm allowed to do this, because I'm not the author of an algorithms textbook. So here's a pseudo proof.

AUDIENCE: Yet.

JULIAN SHUN: Yet. So a processor is either working or stealing at every time step. And the total time that all processors spend working is just T1, because that's the total work that you have to do. And then when it's not doing work, it's stealing. And each steal has a 1 over P chance of reducing the span by 1, because one of the processors is contributing to the longest path in the compilation dag. And there's a 1 over P chance that I'm going to pick that processor and steal some work from that processor and reduce the span of my remaining computation by 1.

And therefore, the expected cost of all steals is going to be order P times T infinity, because I have to steal P things in expectation before I get to the processor that has the critical path. And therefore, my overall costs for stealing is order P times T infinity, because I'm going to do this T infinity times. And since there are P processors, I'm going to divide the expected time by P, so T1 plus O of P times T infinity divided by P, and that's going to give me the bound-- T1 over P plus order T infinity.

So this pseudoproof here ignores issues with independence, but it still gives you an intuition of why we get this expected running time. If you want to actually see the full proof, it's actually quite interesting. It uses random variables and tail bounds of distributions. And this is the paper that has this. This is by Blumofe and Charles Leiserson.

So another thing I want to talk about is that Cilk supports C's rules for pointers. So a pointer to a stack space can be passed from a parent to a child, but not from a child to a parent. And this

is the same as the stack rule for sequential C programs.

So let's say I have this computation on the left here. So A is going to spawn off B, and then it's going to continue executing C. In then C is going to spawn off D and execute E. So we see on the right hand side the views of the stacks for each of the tasks here.

So A sees its own stack. B sees its own stack, but it also sees A's stack, because A is its parent. C will see its own stack, but again, it sees A's stack, because A is its parent. And then finally, D and E, they see the stack of C, and they also see the stack of A. So in general, a task can see the stack of all of its ancestors in this computation graph.

And we call this a cactus stack, because it sort of looks like a cactus, if you draw this upside down. And Cilk's cactus stack supports multiple views of the stacks in parallel, and this is what makes the parallel calls to functions work in C.

We can also bound the stack space used by a Cilk program. So let's let S sub 1 be the stack space required by the serial execution of a Cilk program. Then the stack space required by a P-processor execution is going to be bounded by P times S1. So SP is the stack space required by a P-processor execution. That's less than or equal to P times S1.

Here's a high-level proof of why this is true. So it turns out that the work-stealing algorithm in Cilk maintains what's called the busy leaves property. And this says that each of the existing leaves that are still active in the computation dag have some work they're executing on it.

So in this example here, the vertices shaded in blue and purple-- these are the ones that are in my remaining computation dag. And all of the gray nodes have already been finished. And here-- for each of the leaves here, I have one processor on that leaf executing the task associated with it. So Cilk guarantees this busy leaves property.

And now, for each of these processors, the amount of stack space it needs is it needs the stack space for its own task and then everything above it in this computation dag. And we can actually bound that by the stack space needed by a single processor execution of the Cilk program, S1, because S1 is just the maximum stack space we need, which is basically the longest path in this graph.

And we do this for every processor. So therefore, the upper bound on the stack space required by P-processor execution is just P times S1. And in general, this is a quite loose upper bound, because you're not necessarily going all the way all the way down in this

competition dag every time. Usually you'll be much higher in this computation dag.

So any questions? Yes?

AUDIENCE: In practice, how much work is stolen?

JULIAN SHUN: In practice, if you have enough parallelism, then you're not actually going to steal that much in your algorithm. So if you guarantee that there's a lot of parallelism, then each processor is going to have a lot of its own work to do, and it doesn't need to steal very frequently. But if your parallelism is very low compared to the number of processors-- if it's equal to the number of processors, then you're going to spend a significant amount of time stealing, and the overheads of the work-stealing algorithm are going to show up in your running time.

AUDIENCE: So I meant in one steal-- like do you take half of the deque, or do you take one element of the deque?

JULIAN SHUN: So the standard Cilk work-stealing scheduler takes everything at the top of the deque up until the next spawn. So basically that's a strand. So it takes that. There are variants that take more than that, but the Cilk work-stealing scheduler that we'll be using in this class just takes the top strand.

Any other questions?

So that's actually all I have for today. If you have any additional questions, you can come talk to us after class. And remember to meet with your MITPOSSE mentors soon.