JULIAN SHUN: Today, we're going to talk about multicore programming. And as I was just informed by Charles, it's 2018. I had 2017 on the slide.

So first, congratulations to all of you. You turned in the first project's data. Here's a plot showing the tiers that different groups reached for the beta. And this is in sorted order. And we set the beta cutoff to be tier 45. The final cutoff is tier 48. So the final cutoff we did set a little bit aggressively, but keep in mind that you don't necessarily have to get to the final cutoff in order to get an A on this project.

So we're going to talk about multicore processing today. That's going to be the topic of the next project after you finish the first project. So in a multicore processor, we have a whole bunch of cores that are all placed on the same chip, and they have access to shared memory. They usually also have some sort of private cache, and then a shared last level cache, so L3, in this case. And then they all have access the same memory controller, which goes out to main memory. And then they also have access to I/O.

But for a very long time, chips only had a single core on them. So why do we have multicore processors nowadays? Why did semiconductor vendors start producing chips that had multiple processor cores on them?

So the answer is because of two things. So first, there's Moore's Law, which says that we get more transistors every year. So the number of transistors that you can fit on a chip doubles approximately every two years.

And secondly, there's the end of scaling of clock frequency. So for a very long time, we could just keep increasing the frequency of the single core on the chip. But at around 2004 to 2005, that was no longer the case. We couldn't scale the clock frequency anymore.

So here's a plot showing both the number of transistors you could fit on the chip over time, as well as the clock frequency of the processors over time. And notice that the y-axis is in log

scale here. And the blue line is basically Moore's Law, which says that the number of transistors you can fit on a chip doubles approximately every two years.

And that's been growing pretty steadily. So this plot goes up to 2010, but in fact, it's been growing even up until the present. And it will continue to grow for a couple more years before Moore's Law ends.

However, if you look at the clock frequency line, you see that it was growing quite steadily until about the early 2000s, and then at that point, it flattened out. So at that point, we couldn't increase the clock frequencies anymore, and the clock speed was bounded at about four gigahertz.

So nowadays, if you go buy a processor, it's usually still bounded by around 4 gigahertz. It's usually a little bit less than 4 gigahertz, because it doesn't really make sense to push it all the way. But you might find some processors that are around 4 gigahertz nowadays.

So what happened at around 2004 to 2005? Does anyone know? So Moore's Law basically says that we can fit more transistors on a chip because the transistors become smaller. And when the transistors become smaller, you can reduce the voltage that's needed to operate the transistors.

And as a result, you can increase the clock frequency while maintaining the same power density. And that's what manufacturers did until about 2004 to 2005. They just kept increasing the clock frequency to take advantage of Moore's law. But it turns out that once transistors become small enough, and the voltage used to operate them becomes small enough, there's something called leakage current.

So there's current that leaks, and we're unable to keep reducing the voltage while still having reliable switching. And if you can't reduce the voltage anymore, then you can't increase the clock frequency if you want to keep the same power density.

So here's a plot from Intel back in 2004 when they first started producing multicore processors. And this is plotting the power density versus time. And again, the y-axis is in log scale here. So the green data points are actual data points, and the orange ones are projected. And they projected what the power density would be if we kept increasing the clock frequency at a trend of about 25% to 30% per year, which is what happened up until around 2004.

And because we couldn't reduce the voltage anymore, the power density will go up. And you can see that eventually, it reaches the power density of a nuclear reactor, which is pretty hot. And then it reaches the power density of a rocket nozzle, and eventually you get to the power density of the sun's surface.

So if you have a chip that has a power density equal to the sun's surface-- well, you don't actually really have a chip anymore. So basically if you get into this orange region, you basically have a fire, and you can't really do anything interesting, in terms of performance engineering, at that point.

So to solve this problem, semiconductor vendors didn't increased the clock frequency anymore, but we still had Moore's Law giving us more and more transistors every year. So what they decided to do with these extra transistors was to put them into multiple cores, and then put multiple cores on the same chip.

So we can see that, starting at around 2004, the number of cores per chip becomes more than one. And each generation of Moore's Law will potentially double the number of cores that you can fit on a chip, because it's doubling the number of transistors. And we've seen this trend up until about today. And again, it's going to continue for a couple more years before Moore's Law ends. So that's why we have chips with multiple cores today.

So today, we're going to look at multicore processing. So I first want to introduce the abstract multicore architecture. So this is a very simplified version, but I can fit it on this slide, and it's a good example for illustration. So here, we have a whole bunch of processors. They each have a cache, so that's indicated with the dollar sign. And usually they have a private cache as well as a shared cache, so a shared last level cache, like the L3 cache. And then they're all connected to the network.

And then, through the network, they can connect to the main memory. They can all access the same shared memory. And then usually there's a separate network for the I/O as well, even though I've drawn them as a single network here, so they can access the I/O interface. And potentially, the network will also connect to other multiprocessors on the same system. And this abstract multicore architecture is known as a chip multiprocessor, or CMP. So that's the architecture that we'll be looking at today.

So here's an outline of today's lecture. So first, I'm going to go over some hardware

challenges with shared memory multicore machines. So we're going to look at the cache coherence protocol. And then after looking at hardware, we're going to look at some software solutions to write parallel programs on these multicore machines to take advantage of the extra cores.

And we're going to look at several concurrency platforms listed here. We're going to look at Pthreads. This is basically a low-level API for accessing, or for running your code in parallel. And if you program on Microsoft products, the Win API threads is pretty similar. Then there's Intel Threading Building Blocks, which is a library solution to concurrency. And then there are two linguistic solutions that we'll be looking at-- OpenMP and Cilk Plus. And Cilk Plus is actually the concurrency platform that we'll be using for most of this class.

So let's look at how caches work. So let's say that we have a value in memory at some location, and that value is-- let's say that value is x equals 3. If one processor says, we want to load x, what happens is that processor reads this value from a main memory, brings it into its own cache, and then it also reads the value, loads it into one of its registers. And it keeps this value in cache so that if it wants to access this value again in the near future, it doesn't have to go all the way out to main memory. It can just look at the value in its cache.

Now, what happens if another processor wants to load x? Well, it just does the same thing. It reads the value from main memory, brings it into its cache, and then also loads it into one of the registers. And then same thing with another processor.

It turns out that you don't actually always have to go out to main memory to get the value. If the value resides in one of the other processor's caches, you can also get the value through the other processor's cache. And sometimes that's cheaper than going all the way out to main memory.

So the second processor now loads x again. And it's in cache, so it doesn't have to go to main memory or anybody else's cache. So what happens now if we want to store x, if we want to set the value of x to something else?

So let's say this processor wants to set x equal to 5. So it's going to write x equals 5 and store that result in its own cache. So that's all well and good. Now what happens when the first processor wants to load x? Well, it seems that the value of x is in its own cache, so it's just going to read the value of x there, and it gets a value of 3.

So what's the problem there? Yes?

AUDIENCE: The path is stale.

JULIAN SHUN: Yeah. So the problem is that the value of x in the first processor's cache is stale, because another processor updated it. So now this value of x in the first processor's cache is invalid.

So that's the problem. And one of the main challenges of multicore hardware is to try to solve this problem of cache coherence-- making sure that the values in different processors' caches are consistent across updates.

So one basic protocol for solving this problem is known as the MSI protocol. And in this protocol, each cache line is labeled with a state. So there are three possible states-- M, S, and I. And this is done on the granularity of cache lines.

Because it turns out that storing this information is relatively expensive, so you don't want to store it for every memory location. So they do it on a per cache line basis. Does anyone know what the size of a cache line is, on the machines that we're using? Yeah?

AUDIENCE: 64 bytes.

JULIAN SHUN: Yeah, so it's 64 bytes. And that's typically what you see today on most Intel and AMD machines. There's some architectures that have different cache lines, like 128 bytes. But for our class, the machines that we're using will have 64 byte cache lines. It's important to remember that so that when you're doing back-of-the-envelope calculations, you can get accurate estimates.

So the three states in the MSI protocol are M, S, and I. So M stands for modified. And when a cache block is in the modified state, that means no other caches can contain this block in the M or the S states.

The S state means that the block is shared, so other caches can also have this block in shared state.

And then finally, I mean the cache block is invalid. So that's essentially the same as the cache block not being in the cache.

And to solve the problem of cache coherency, when one cache modifies a location, it has to inform all the other caches that their values are now stale, because this cache modified the

value. So it's going to invalidate all of the other copies of that cache line in other caches by changing their state from S to I.

So let's see how this works. So let's say that the second processor wants to store y equals 5. So previously, a value of y was 17, and it was in shared state. The cache line containing y equals 17 was in shared state.

So now, when I do y equals 5, I'm going to set the second processor's cache-- that cache line-- to modified state. And then I'm going to invalidate the cache line in all of the other caches that contain that cache line. So now the first cache and the fourth cache each have a state of I for y equals 17, because that value is stale.

Is there any questions? Yes?

AUDIENCE: If we already have to tell the other things to switch to invalid, why not just tell them the value of y?

JULIAN SHUN: Yeah, so there are actually some protocols that do that. So this is just the most basic protocol. So this protocol doesn't do it. But there are some that are used in practice that actually do do that. So it's a good point. But I just want to present the most basic protocol for now. Sorry.

And then, when you load a value, you can first check whether your cache line is in M or S state. And if it is an M or S state, then you can just read that value directly. But if it's in the I state, or if it's not there, then you have to fetch that block from either another processor's cache or fetch it from main memory.

So it turns out that there are many other protocols out there. There's something known as MESI, the messy protocol. There's also MOESI and many other different protocols. And some of them are proprietary. And they all do different things. And it turns out that all of these protocols are quite complicated, and it's very hard to get these protocols right. And in fact, one of the most earliest successes of formal verification was improving some of these cache [INAUDIBLE] protocols to be correct. Yes, question?

AUDIENCE: What happens if two processors try to modify one value at the same time

JULIAN SHUN: Yeah, so if two processors try to modify the value, one of them has to happen first. So the hardware is going to take care of that. So the first one that actually modifies it will invalidate all the other copies, and then the second one that modifies the value will again invalidate all of

the other copies.

And when you do that-- when a lot of processors try to modify the same value, you get something known as an invalidation storm. So you have a bunch of invalidation messages going throughout the hardware. And that can lead to a big performance bottleneck. Because each processor, when it modifies its value, it has to inform all the other processors.

And if all the processors are modifying the same value, you get this sort of quadratic behavior. The hardware is still going to guarantee that one of their processors is going to end up writing the value there. But you should be aware of this performance issue when you're writing parallel code. Yes?

AUDIENCE: So all of this protocol stuff happens in hardware?

JULIAN SHUN: Yes, so this is all implemented in hardware. So if you take a computer architecture class, you'll learn much more about these protocols and all of their variants. So for our purposes, we don't actually need to understand all the details of the hardware. We just need to understand what it's doing at a high level so we can understand when we have a performance bottleneck and why we have a performance bottleneck. So that's why I'm just introducing the most basic protocol here. Any other questions?

So I talked a little bit about the shared memory hardware. Let's now look at some concurrency platforms. So these are the four platforms that we'll be looking at today.

So first, what is a concurrency platform? Well, writing parallel programs is very difficult. It's very hard to get these programs to be correct. And if you want to optimize their performance, it becomes even harder. So it's very painful and error-prone. And a concurrency platform abstracts processor cores and handles synchronization and communication protocols. And it also performs load balancing for you. So it makes your lives much easier. And so today we're going to talk about some of these different concurrency platforms.

So to illustrate these concurrency platforms, I'm going to do the Fibonacci numbers example. So does anybody not know what Fibonacci is? So good. Everybody knows what Fibonacci is. So it's a sequence where each number is the sum of the previous two numbers. And the recurrence is shown in this brown box here.

The sequence is named after Leonardo di Pisa, who was also known as Fibonacci, which is a contraction of Bonacci, son of Bonaccio. So that's where the name Fibonacci came from. And

in Fibonacce's 1202 book, *Liber Abaci,* he introduced the sequence-- the Fibonacci sequence-- to Western mathematics, although it had been previously known to Indian mathematicians for several centuries. But this is what we call the sequence nowadays-- Fibonacci numbers.

So here's a Fibonacci program. Has anyone seen this algorithm before? A couple of people. Probably more, but people didn't raise their hands.

So it's a recursive program. So it basically implements the recurrence from the previous slide. So if n is less than 2, we just return n. Otherwise, we compute fib of n minus 1, store that value in x, fib of n minus 2, store that value in y, and then return the sum of x and y.

So I do want to make a disclaimer to the algorithms police that this is actually a very bad algorithm. So this algorithm takes exponential time, and there's actually much better ways to compute the end Fibonacci number. There's a linear time algorithm, which just computes the Fibonacci numbers from bottom up.

This algorithm here is actually redoing a lot of the work, because it's computing Fibonacci numbers multiple times. Whereas if you do a linear scan from the smallest numbers up, you only have to compute each one once. And there's actually an even better algorithm that takes logarithmic time, and it's based on squaring matrices. So has anyone seen that algorithm before? So a couple of people. So if you're interested in learning more about this algorithm, I encourage you to look at your favorite textbook, *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein.

So even though this--

[LAUGHTER]

Yes. So even though this is a pretty bad algorithm, it's still a good educational example, because I can fit it on one slide and illustrate all the concepts of parallelism that we want to cover today.

So here's the execution tree for fib of 4. So we see that fib of 4 is going to call fib of 3 and fib of 2. Fib of 3 is going to call fib of 2, fib of 1, and so on. And you can see that repeated computations here. So fib of 2 is being computed twice, and so on. And if you have a much larger tree-- say you ran this on fib of 40-- then you'll have many more overlapping computations.

It turns out that the two recursive calls can actually be parallelized, because they're completely independent calculations. So the key idea for parallelization is to simultaneously execute the two recursive sub-calls to fib. And in fact, you can do this recursively. So the two sub-calls to fib of 3 can also be executed in parallel, and the two sub-calls of fib of 2 can also be executed in parallel, and so on. So you have all of these calls that can be executed in parallel. So that's the key idea for extracting parallelism from this algorithm.

So let's now look at how we can use Pthreads to implement this simple Fibonacci algorithm. So Pthreads is a standard API for threading, and it's supported on all Unix-based machines. And if you're programming using Microsoft products, then the equivalent is Win API threads.

And Pthreads is actually standard in ANSI and IEEE, so there's this number here that specifies the standard. But nowadays, we just call it Pthreads. And it's basically a do-it-yourself concurrency platform. So it's like the assembly language of parallel programming. It's built as a library of functions with special non-C semantics. Because if you're just writing code in C, you can't really say which parts of the code should be executed in parallel. So Pthreads provides you a library of functions that allow you to specify concurrency in your program.

And each thread implements an abstraction of a processor, and these threads are then multiplexed onto the actual machine resources. So the number of threads that you create doesn't necessarily have to match the number of processors you have on your machine. So if you have more threads than the number of processors you have, then they'll just be multiplexing. So you can actually run a Pthreads program on a single core even though you have multiple threads in the program. They would just be time-sharing.

All the threads communicate through shared memory, so they all have access to the same view of the memory. And the library functions that Pthreads provides mask the protocols involved in interthread coordination, so you don't have to do it yourself. Because it turns out that this is quite difficult to do correctly by hand.

So now I want to look at the key Pthread functions. So the first Pthread is pthread_create. And this takes four arguments. So the first argument is this pthread_t type. This is basically going to store an identifier for the new thread that pthread_create will create so that we can use that thread in our computations.

pthread_attr_t-- this set some thread attributes, and for our purposes, we can just set it to null

and use the default attributes.

The third argument is this function that's going to be executed after we create the thread. So we're going to need to define this function that we want the thread to execute.

And then finally, we have this void *arg argument, which stores the arguments that are going to be passed to the function that we're going to be executing.

And then pthread_create also returns an error status, returns an integer specifying whether the thread creation was successful or not.

And then there's another function called pthread_join. pthread_join basically says that we want to block at this part of our code until this specified thread finishes. So it takes as argument pthread_t.

So this thread identifier, and these thread identifiers, were created when we called pthread_create. It also has a second argument, status, which is going to store the status of the terminating thread. And then pthread_join also returns an error status. So essentially what this does is it says to wait until this thread finishes before we continue on in our program.

So any questions so far?

So here's what the implementation of Fibonacci looks like using Pthreads. So on the left, we see the original program that we had, the fib function there. That's just the sequential code. And then we have all this other stuff to enable it to run in parallel. So first, we have this struct on the left, thread_args. This struct here is used to store the arguments that are passed to the function that the thread is going to execute.

And then we have this thread_func. What that does is it reads the input argument from this thread_args struct, and then it sets that to i, and then it calls fib of i. And that gives you the output, and then we store the result into the output of the struct. And then that also just returns null.

And then over on the right hand side, we have the main function that will actually call the fib function on the left. So we initialize a whole bunch of variables that we need to execute these threads. And then we first check if n is less than 30. If n is less than 30, it turns out that it's actually not worth creating threads to execute this program in parallel, because of the overhead of thread creation. So if n is less than 30, we'll just execute the program

sequentially.

And this idea is known as coarsening. So you saw a similar example a couple of lectures ago when we did coarsening for sorting. But this is in the context of a parallel programming. So here, because there are some overheads to running a function in parallel, if the input size is small enough, sometimes you want to just execute it sequentially.

And then we're going to-- so let me just walk through this code, since I have an animation. So the next thing it's going to do is it's going to marshal the input argument to the thread so it's going to store the input argument n minus 1 in this args struct.

And then we're going to call pthread_create with a thread variable. For thread_args, we're just going to use null. And then we're going to pass the thread_func that we defined on the left. And then we're going to pass the args structure. And inside this args structure, the input is set to n minus 1, which we did on the previous line.

And then pthread_create is going to give a return value. So if the Pthread creation was successful, then the status is going to be null, and we can continue. And when we continue, we're going to execute, now, fib of n minus 2 and store the result of that into our result variable. And this is done at the same time that fib of n minus 1 is executing. Because we created this Pthread, and we told it to call this thread_func function that we defined on the left. So both fib of n minus 1 and fib of n minus 2 are executing in parallel now.

And then we have this pthread_join, which says we're going to wait until the thread that we've created finishes before we move on, because we need to know the result of both of the sub-calls before we can finish this function. And once that's done-- well, we first check the status to see if it was successful. And if so, then we add the outputs of the argument's struct to the result. So args.output will store the output of fib of n minus 1.

So that's the Pthreads code. Any questions on how this works? Yeah?

AUDIENCE: I have a question about the thread function. So it looks like you passed a void pointer, but then you cast it to something else every time you use that--

JULIAN SHUN: Yeah, so this is because the pthread_create function takes as input a void star pointer. Because it's actually a generic function, so it doesn't know what the data type is. It has to work for all data types, and that's why we need to cast it to avoid star. When we pass it to pthread_create and then inside the thread_func, we actually do know what type of pointer that

is, so then we cast it.

So does this code seem very parallel? So how many parallel calls am I doing here? Yeah?

AUDIENCE: Just one.

JULIAN SHUN: Yeah, so I'm only creating one thread. So I'm executing two things in parallel. So if I ran this code on four processors, what's the maximum speed-up I could get?

AUDIENCE: [INAUDIBLE].

JULIAN SHUN: So the maximum speed-up I can get is just two, because I'm only running two things in parallel. So this doesn't recursively create threads. It only creates one thread at the top level.

And if you wanted to make it so that this code actually recursively created threads, it would actually become much more complicated. And that's one of the disadvantages of implementing this code in Pthreads. So we'll look at other solutions that will make this task much easier.

So some of the issues with Pthreads are shown on this slide here. So there's a high overhead to creating a thread. So creating a thread typically takes over 10 to the 4th cycles. And this leads to very coarse-grained concurrency, because your tasks have to do a lot of work in order to amortize the costs of creating that thread.

There are something called thread pulls, which can help. And the idea here is to create a whole bunch of threads at the same time to amortize the costs of thread creation. And then when you need a thread, you just take one from the thread pull. So the thread pull contains threads that are just waiting to do work.

There's also a scalability issue with this code that I showed on the previous slide. The Fibonacci code gets, at most, 1.5x speed-up for two cores. Why is it 1.5 here? Does anyone know? Yeah?

AUDIENCE: You have the asymmetry in the size of the two calls.

JULIAN SHUN: Yeah, so it turns out that the two calls that I'm executing in parallel-- they're not doing the same amount of work. So one is computing fib of n minus 1, one is computing fib of n minus 2. And does anyone know what the ratio between these two values is? Yeah, so it's the golden ratio. It's about 1.6. It turns out that if you can get a speed-up of 1.6, then that's great. But

there are some overheads, so this code will get about a 1.5 speed up.

And if you want to run this to take advantage of more cores, then you need to rewrite this code, and it becomes more complicated.

Third, there's the issue of modularity. So if you look at this code here, you see that the Fibonacci logic is not nicely encapsulated within one function. We have that logic in the fib function on the left, but then we also have some of the fib logic on the right in our main function. And this makes this code not modular.

And if we want to build programs on top of this, it makes it very hard to maintain, if we want to just change the logic of the Fibonacci function a little bit, because now we have to change it in multiple places instead of just having everything in one place. So it's not a good idea to write code that's not modular, so please don't do that in your projects.

And then finally, the code becomes complicated because you have to actually move these arguments around. That's known as argument marshaling. And then you have to engage in error-prone protocols in order to do load balancing. So if you recall here, we have to actually place the argument n minus 1 into args.input and we have to extract the value out of args.output. So that makes the code very messy.

So why do I say shades of 1958 here? Does anyone know what happened in 1958? Who was around in 1958? Just Charles? So there was a first something in 1958. What was it?

So turns out in 1958, we had the first compiler. And this was the Fortran compiler. And before we had Fortran compiler, programmers were writing things in assembly. And when you write things in assembly, you have to do argument marshaling, because you have to place things into the appropriate registers before calling a function, and also move things around when you return from a function.

And the nice thing about the first compiler is that it actually did all of this argument marshaling for you. So now you can just pass arguments to a function, and the compiler will generate code that will do the argument marshaling for us.

So having you do this in Pthreads is similar to having to write code in assembly, because you have to actually manually marshal these arguments. So hopefully, there are better ways to do this. And indeed, we'll look at some other solutions that will make it easier on the programmer.

Any questions before I continue?

So we looked at Pthreads. Next, let's look at Threading Building Blocks. So Threading Building Blocks is a library solution. It was developed by Intel. And it's implemented as a C++ library that runs on top of native threads. So the underlying implementation uses threads, but the programmer doesn't deal with threads. Instead, the programmer specifies tasks, and these tasks are automatically load-balanced across the threads using a work-stealing algorithm inspired by research at MIT-- Charles Leiserson's research.

And the focus of Intel TBB is on performance. And as we'll see, the code written using TBB is simpler than what you would have to write if you used Pthreads. So let's look at how we can implement Fibonacci using TBB.

So in TBB, we have to create these tasks. So in the Fibonacci code, we create this fib task class. And inside the task, we have to define this execute function. So the execute function is the function that performs a computation when we start the task. And this is where we define the Fibonacci logic. This task also takes as input these arguments parameter, n and sum. So n is the input here and sum is the output.

And in TBB, we can easily create a recursive program that extracts more parallelism. And here, what we're doing is we're recursively creating two child tasks, a and b. That's the syntax for creating the tasks. And here, we can just pass the arguments to FibTask instead of marshaling the arguments ourselves.

And then what we have here is a set_ref_count. And this basically is the number of tasks that we have to wait for plus one, so plus one for ourselves. And in this case, we created two children tasks, and we have ourselves, so that's 2 plus 1.

And then after that, we start task b using the spawn(b) call. And then we do spawn_and_wait_for_all with a as the argument. In this place, he says, we're going to start task a, and then also wait for both a and b to finish before we proceed. So this spawn_and_wait_for_all call is going to look at the ref count that we set above and wait for that many tasks to finish before it continues.

And after both and a and b have completed, then we can just sum up the results and store that into the sum variable. And here, these tasks are created recursively. So unlike the Pthreads implementation that was only creating one thread at the top level, here, we're

actually recursively creating more and more tasks. So we can actually get more parallelism from this code and scale to more processors.

We also need this main function just to start up the program. So what we do here is we create a root task, which just computes fib of n, and then we call spawn_root_and_wait(a). So a is the task for the root. And then it will just run the root task.

So that's what Fibonacci looks like in TBB. So this is much simpler than the Pthreads implementation. And it also gets better performance, because we can extract more parallelism from the computation. Any questions?

So TBB also has many other features in addition to tasks. So TBB provides many C++ templates to express common patterns, and you can use these templates on different data types. So they have a parallel_for, which is used to express loop parallelism. So you can loop over a bunch of iterations in parallel.

They also have a parallel_reduce for data aggregation. For example, if you want to sum together a whole bunch of values, you can use a parallel_reduce to do that in parallel. They also have pipeline and filter. That's used for software pipelining.

TBB provides many concurrent container classes, which allow multiple threads to safely access and update the items in a container concurrently. So for example, they have hash tables, trees, priority cues, and so on. And you can just use these out of the box, and they'll work in parallel. You can do concurrent updates and reads to these data structures.

TBB also has a variety of mutual exclusion library functions, such as locks and atomic operations. So there are a lot of features of TBB, which is why it's one of the more popular concurrency platforms. And because of all of these features, you don't have to implement many of these things by yourself, and still get pretty good performance.

So TBB was a library solution to the concurrency problem. Now we're going to look at two linguistic solutions-- OpenMP and Cilk. So let's start with OpenMP.

So OpenMP is a specification by an industry consortium. And there are several compilers available that support OpenMP, both open source and proprietary. So nowadays, GCC, ICC, and Clang all support OpenMP, as well as Visual Studio. And OpenMP is-- it provides linguistic extensions to C and C++, as well as Fortran, in the form of compiler pragmas.

So you use these compiler pragmas in your code to specify which pieces of code should run in parallel. And OpenMP also runs on top of native threads, but the programmer isn't exposed to these threads. OpenMP supports loop parallelism, so you can do parallel for loops. They have task parallelism as well as pipeline parallelism.

So let's look at how we can implement Fibonacci in OpenMP. So this is the entire code. So I want you to compare this to the Pthreads implementation that we saw 10 minutes ago. So this code is much cleaner than the Pthreads implementation, and it also performs better. So let's see how this code works.

So we have these compiler pragmas, or compiler directives. And the compiler pragma for creating a parallel task is omp task. So we're going to create an OpenMP task for fib of n minus 1 as well as fib of n minus 2. There's also this shared pragma, which specifies that the two variables in the arguments are shared across different threads. So you also have to specify whether variables are private or shared.

And then the pragma omp wait just says we're going to wait for the preceding task to complete before we continue. So here, it's going to wait for fib of n minus 1 and fib of n minus 2 to finish before we return the result, which is what we want. And then after that, we just return x plus y. So that's the entire code.

And OpenMP also provides many other pragma directives, in addition to task. So we can use a parallel for to do loop parallelism. There's reduction. There's also directives for scheduling and data sharing. So you can specify how you want a particular loop to be scheduled. OpenMP has many different scheduling policies. They have static parallelism, dynamic parallelism, and so on. And then these scheduling directives also have different grain sizes. The data sharing directives are specifying whether variables are private or shared.

OpenMP also supplies a variety of synchronization constructs, such as barriers, atomic updates, mutual exclusion, or mutex locks. So OpenMP also has many features, and it's also one of the more popular solutions to writing parallel programs. As you saw in the previous example, the code is much simpler than if you were to write something using Pthreads or even TBB. This is a much simpler solution.

Any questions? Yeah?

AUDIENCE: So with every compiler directive, does it spawn a new [INAUDIBLE] on a different processor?

JULIAN SHUN: So this code here is actually independent of the number of processors. So there is actually a scheduling algorithm that will determine how the tasks get mapped to different processors. So if you spawn a new task, it doesn't necessarily put it on a different processor. And you can have more tasks than the number of processors available. So there's a scheduling algorithm that will take care of how these tasks get mapped to different processors, and that's hidden from the programmer. Although you can use these scheduling pragmas to give hints to the compiler how it should schedule it. Yeah?

AUDIENCE: What is the operating system [INAUDIBLE] scheduling [INAUDIBLE]?

JULIAN SHUN: Underneath, this is implemented using Pthreads, which has to make operating system calls to, basically, directly talk to the processor cores and do multiplexing and so forth. So the operating system is involved at a very low level.

So the last concurrency platform that we'll be looking at today is Cilk. We're going to look at Cilk Plus, actually. And the Cilk part of Cilk Plus is a small set of linguistic extensions to C and C++ that support fork-join parallelism. So for example, the Fibonacci example uses fork-join parallelism, so you can use Cilk to implement that. And the Plus part of Cilk Plus supports vector parallelism, which you had experience working with in your homeworks.

So Cilk Plus was initially developed by Cilk Arts, which was an MIT spin-off. And Cilk Arts was acquired by Intel in July 2009. And the Cilk Plus implementation was based on the award-winning Cilk multi-threaded language that was developed two decades ago here at MIT by Charles Leiserson's research group.

And it features a provably efficient work-stealing scheduler. So this scheduler is provably efficient. You can actually prove theoretical bounds on it. And this allows you to implement theoretically efficient algorithms, which we'll talk more about in another lecture-- algorithm design. But it provides a provably efficient work-stealing scheduler. And Charles Leiserson has a very famous paper that has a proof of that this scheduler is optimal. So if you're interested in reading about this, you can talk to us offline.

Cilk Plus also provides a hyperobject library for parallelizing code with global variables. And you'll have a chance to play around with hyperobjects in homework 4.

The Cilk Plus ecosystem also includes useful programming tools, such as the Cilk Screen Race Detector. So this allows you to detect determinacy races in your program to help you

isolate bugs and performance bottlenecks. It also has a scalability analyzer called Cilk View. And Cilk View will basically analyze the amount of work that your program is doing, as well as the maximum amount of parallelism that your code could possibly extract from the hardware.

So that's Intel Cilk Plus. But it turns out that we're not actually going to be using Intel Cilk Plus in this class. We're going to be using a better compiler. And this compiler is based on Tapir/LLVM. And it supports the Cilk subset of Cilk Plus. And Tapir/LLVM was actually recently developed at MIT by T. B. Schardl, who gave a lecture last week, William Moses, who's a grad student working with Charles, as well as Charles Leiserson. So talking a lot about Charles's work today.

And Tapir/LLVM generally produces better code, relative to its base compiler, than all other implementations of Cilk out there. So it's the best Cilk compiler that's available today. And they actually wrote a very nice paper on this last year, Charles Leiserson and his group. And that paper received the Best Paper Award at the annual Symposium on Principles and Practices of Parallel Programming, or PPoPP. So you should look at that paper as well.

So right now, Tapir/LLVM uses the Intel Cilk Plus runtime system, but I believe Charles's group has plans to implement a better runtime system. And Tapir/LLVM also supports more general features than existing Cilk compilers. So in addition to spawning functions, you can also spawn code blocks that are not separate functions, and this makes writing programs more flexible. You don't have to actually create a separate function if you want to execute a code block in parallel.

Any questions?

So this is the Cilk code for Fibonacci. So it's also pretty simple. It looks very similar to the sequential program, except we have these cilk_spawn and cilk_synv statements in the code. So what do these statements do?

So cilk_spawn says that the named child function, which is the function that is right after this cilk_spawn statement, may execute in parallel with the parent caller. The parent caller is the function that is calling cilk_spawn. So this says that fib of n minus 1 can execute in parallel with the function that called it. And then this function is then going to call fib of n minus 2. And fib of n minus 2 and fib of n minus 1 now can be executing in parallel.

And then cilk_sync says that control cannot pass this point until all of the spawn children have

returned. So this is going to wait for fib of n minus 1 to return before we go to the return statement where we add up x and y.

So one important thing to note is that the Cilk keywords grant permission for parallel execution, but they don't actually force or command parallel execution. So even though I said cilk_spawn here, the runtime system doesn't necessarily have to run fib of n minus 1 in parallel with fib of n minus 2. I'm just saying that I could run these two things in parallel, and it's up to the runtime system to decide whether or not to run these things in parallel, based on its scheduling policy.

So let's look at another example of Cilk. So let's look at loop parallelism. So here we want to do a matrix transpose, and we want to do this in-place. So the idea here is we want to basically swap the elements below the diagonal to its mirror image above the diagonal.

And here's some code to do this. So we have a cilk_for. So this is basically a parallel for loop. It goes from i equals 1 to n minus 1. And then the inner for loop goes from j equals 0 up to i minus 1. And then we just swap a of i j with a of j i, using these three statements inside the body of the for loop. So to execute a for loop in parallel, you just have to add cilk underscore to the for keyword. And that's as simple as it gets.

So this code is actually going to run in parallel and get pretty good speed-up for this particular problem. And internally, Cilk for loops are transformed into nested cilk_spawn and cilk_sync calls. So the compiler is going to get rid of the cilk_for and change it into cilk_spawn and cilk_sync.

So it's going to recursively divide the iteration space into half, and then it's going to spawn off one half and then execute the other half in parallel with that, and then recursively do that until the iteration range becomes small enough, at which point it doesn't make sense to execute it in parallel anymore, so we just execute that range sequentially.

So that's loop parallelism in Cilk. Any questions? Yes?

AUDIENCE: How does it know [INAUDIBLE] something weird, can it still do that?

JULIAN SHUN: Yeah, so the compiler can actually figure out what the iteration space is. So you don't necessarily have to be incrementing by 1. You can do something else. You just have to guarantee that all of the iterations are independent. So if you have a determinacy race across the different iterations of your cilk_for loop, then your result might not necessarily be correct.

So you have to make sure that the iterations are, indeed, independent.

Yes?

AUDIENCE: Can you nest cilk_fors?

JULIAN SHUN: Yes, so you can nest cilk_fors. But it turns out that, for this example, usually, you already have enough parallelism in the outer loop for large enough values of n, so it doesn't make sense to put a cilk_for loop inside, because using a cilk_for loop adds some additional overheads. But you can actually do nested cilk_for loops. And in some cases, it does make sense, especially if there's not enough parallelism in the outermost for loop. So good question.

Yes?

AUDIENCE: What does the assembly code look like for the parallel code?

JULIAN SHUN: So it has a bunch of calls to the Cilk runtime system. I don't know all the details, because I haven't looked at this recently. But I think you can actually generate the assembly code using a flag in the Clang compiler. So that's a good exercise.

AUDIENCE: Yeah, you probably want to look at the LLVM IR, rather than the assembly, to begin with, to understand what's going on. It has three instructions that are not in the standard LLVM, which were added to support parallelism. Those things, when it's lowered into assembly, each of those instructions becomes a bunch of assembly language instructions. So you don't want to mess with looking at it in the assembler until you see what it looks like in the LLVM first.

JULIAN SHUN: So good question. Any other questions about this code here? OK, so let's look at another example. So let's say we had this for loop where, on each iteration i, we're just incrementing a variable sum by i. So this is essentially going to compute the summation of everything from i equals 0 up to n minus 1, and then print out the result.

So one straightforward way to try to parallelize this code is to just change the for to cilk_for. So does this code work? Who thinks that this code doesn't work? Or doesn't compute the correct result? So about half of you. And who thinks this code does work? So a couple people. And I guess the rest of the people don't care.

So it turns out that it's not actually necessarily going to give you the right answer. Because the cilk_for loop says you can execute these iterations in parallel, but they're all updating the same

shared variable sum here. So you have what's called a determinacy race, where multiple processors can be writing to the same memory location. We'll talk much more about determinacy races in the next lecture. But for this example, it's not necessarily going to work if you run it on more than one processor.

And Cilk actually has a nice way to deal with this. So in Cilk, we have something known as a reducer. This is one example of a hyperobject, which I mentioned earlier. And with a reducer, what you have to do is, instead of declaring the sum variable just has an unsigned long data type, what you do is you use this macro called CILK_C_REDUCER_OPADD, which specifies we want to create a reducer with the addition function.

Then we have the variable name sum, the data type unsigned long, and then the initial value 0. And then we have a macro to register this reducer, so a CILK_C_REGISTER_REDUCER. And then now, inside this cilk_for loop, we can increment the sum, or REDUCER_VIEW, of sum, which is another macro, by i.

And you can actually execute this in parallel, and it will give you the same answer that you would get if you ran this sequentially. So the reducer will take care of this determinacy race for you. And at the end, when you print out this result, you'll see that the sum is equal to the sum that you expect. And then after you finish using the reducer, you use this other macro called CILK_C_UNREGISTER_REDUCER(sum) that tells the system that you're done using this reducer.

So this is one way to deal with this problem when you want to do a reduction. And it turns out that there are many other interesting reduction operators that you might want to use. And in general, you can create reduces for monoids. And monoids are algebraic structures that have an associative binary operation as well as an identity element. So the addition operator is a monoid, because it's associative, it's binary, and the identity element is 0.

Cilk also has several other predefined reducers, including multiplication, min, max, and, or, xor, et cetera. So these are all monoids. And you can also define your own reducer. So in fact, in the next homework, you'll have the opportunity to play around with reducers and write a reducer for lists.

So that's reducers. Another nice thing about Cilk is that there's always a valid serial interpretation of the program. So the serial elision of a Cilk program is always a legal interpretation. And for the Cilk source code on the left, the serial elision is basically the code

you get if you get rid of the cilk_spawn and cilk_sync statements. And this looks just like the sequential code.

And remember that the Cilk keywords grant permission for parallel execution, but they don't necessarily command parallel execution. So if you ran this Cilk code using a single core, it wouldn't actually create these parallel tasks, and you would get the same answer as the sequential program. And this is-- in the serial edition-- is also a correct interpretation.

So unlike other solutions, such as TBB and Pthreads, it's actually difficult, in those environments, to get a program that does what the sequential program does. Because they're actually doing a lot of additional work to set up these parallel calls and create these argument structures and other scheduling constructs. Whereas in Cilk, it's very easy just to get this serial elision. You just define cilk_spawn and cilk_sync to be null. You also define cilk_for to be for. And then this gives you a valid sequential program.

So when you're debugging code, and you might first want to check if the sequential elision of your Cilk program is correct, and you can easily do that by using these macros. Or actually, there's actually a compiler flag that will do that for you and give you the equivalent C program. So this is a nice way to debug, because you don't have to start with the parallel program. You can first check if this serial program is correct before you go on to debug the parallel program.

Questions? Yes?

AUDIENCE: So does cilk_for-- does each iteration of the cilk_for its own task that the scheduler decides if it wants to execute in parallel, or if it executes in parallel, do all of the iterations execute in parallel?

JULIAN SHUN: So it turns out that by default, it groups a bunch of iterations together into a single task, because it doesn't make sense to break it down into such small chunks, due to the overheads of parallelism. But there's actually a setting you can do to change the grain size of the for loop. So you could actually make it so that each iteration is its own task. And then, as you the scheduler will decide how to map these different task onto different processors, or even if it wants to execute any of these tasks in parallel. So good question.

So the idea in Cilk is to allow the programmer to express logical parallelism in an application. So the programmer just has to identify which pieces of the code could be executed in parallel, but doesn't necessarily have to determine which pieces of code should be executed in parallel.

And then Cilk has a runtime scheduler that will automatically map the executing program onto the available processor cores' runtime. And it does this dynamically using a work-stealing scheduling algorithm. And the work-stealing scheduler is used to balance the tasks evenly across the different processors. And we'll talk more about the work-stealing scheduler in a future lecture.

But I want to emphasize that unlike the other concurrency platforms that we looked at today, Cilk's work-stealing scheduling algorithm is theoretically efficient, whereas the OpenMP and TBB schedulers are not theoretically efficient. So this is a nice property, because it will guarantee you that the algorithms you write on top of Cilk will also be theoretically efficient.

So here's a high-level illustration of the Cilk ecosystem. It's a very simplified view, but I did this to fit it on a single slide. So what you do is you take the Cilk source code, you pass it to your favorite Cilk compiler-- the Tapir/LLVM compiler-- and this gives you a binary that you can run on multiple processors. And then you pass a program input to the binary, you run it on however many processors you have, and then this allows you to benchmark the parallel performance of your program.

You can also do serial testing. And to do this, you just obtain a serial elision of the Cilk program, and you pass it to an ordinary C or C++ compiler. It generates a binary that can only run on a single processor, and you run your suite of serial regression tests on this single threaded binary. And this will allow you to benchmark the performance of your serial code and also debug any issues that might have arised when you were running this program sequentially.

Another way to do this is you can actually just compile the original Cilk code but run it on a single processor. So there's a command line argument that tells the runtime system how many processors you want to use. And if you set that parameter to 1, then it will only use a single processor. And this allows you to benchmark the single threaded performance of your code as well. And the parallel program executing on a single core should behave exactly the same way as the execution of this serial elision. So that's one of the advantages of using Cilk.

And because you can easily do serial testing using the Cilk platform, this allows you to separate out the serial correctness from the parallel correctness. As I said earlier, you can first debug the serial correctness, as well as any performance issues, before moving on to the parallel version. And another point I want to make is that because Cilk actually uses the serial

program inside its task, it's actually good to optimize the serial program even when you're writing a parallel program, because optimizing the serial program for performance will also translate to better parallel performance.

Another nice feature of Cilk is that it has this tool called Cilksan, which stands for Cilk Sanitizer. And Cilksan will detect any determinacy races that you have in your code, which will significantly help you with debugging the correctness as well as the performance of your code.

So if you compile the Cilk code using the Cilksan flag, it will generate an instrumented binary that, when you run, it will find and localize all the determinacy races in your program. So it will tell you where the determinacy races occur, so that you can go inspect that part of your code and fix it if necessary. So this is a very useful tool for benchmarking your parallel programs.

Cilk also has another nice tool called Cilkscale. Cilkscale is a performance analyzer. It will analyze how much parallelism is available in your program as well as the total amount of work that it's doing. So again, you pass a flag to the compiler that will turn on Cilkscale, and it will generate a binary that is instrumented. And then when you run this code, it will give you a scalability report.

So you'll find these tools very useful when you're doing the next project. And we'll talk a little bit more about these two tools in the next lecture. And as I said, Cilkscale will analyze how well your program will scale to larger machines. So it will basically tell you the maximum number of processors that your code could possibly take advantage of.

Any questions? Yes?

AUDIENCE: What do you mean when you say runtime?

JULIAN SHUN: So I mean the scheduler-- the Cilk runtime scheduler that's scheduling the different tasks when you're running the program.

AUDIENCE: So that's included in the binary.

JULIAN SHUN: So it's linked from the binary. It's not stored in the same place. It's linked.

Other questions?

So let me summarize what we looked at today. So first, we saw that most processors today have multiple cores. And probably all of your laptops have more than one core on it. Who has

a laptop that only has one core?

AUDIENCE: [INAUDIBLE].

JULIAN SHUN: When did you buy it? Probably a long time ago.

So nowadays, obtaining high performance on your machines requires you to write parallel programs. But parallel programming can be very hard, especially if you have the program directly on the processor cores and interact with the operating system yourself. So Cilk is very nice, because it abstracts the processor cores from the programmer, it handles synchronization and communication protocols, and it also performs provably good load-balancing.

And in the next project, you'll have a chance to play around with Cilk. You'll be implementing your own parallel screensaver, so that's a very fun project to do. And possibly, in one of the future lectures, we'll post some of the nicest screensaver that students developed for everyone to see. OK, so that's all.