MITOCW | MIT6_172_F10_lec12_300k-mp4

PROFESSOR: So John is going to present project three, beta.

JOHN: All right. So here's the performance grades. In general, the submission went a lot better than last time in that things were on time and nobody failed to build, or forgot to add files to their project, or so on. We did change the scoring mechanism a little bit. In the [? mdriver ?] that we gave you, if your validator failed you on any of your traces, your score is a zero.

In this one, we decided to be nicer. We replaced your validator with our correct validator. And for traces that you failed, you get a zero for the points that those traces contribute. But you did get an overall partial score, even if you failed a couple traces. So on that note, the reference implementation does get a 56 on this score. And there were people who had slower than reference implementations that landed below 56. So that might be something to think about for your final submission.

The high score was a 96. And there were actually quite a few groups in the 90s. So overall, people did really well on this. With that said, your validators didn't really-- I guess they were OK.

But there's some people whose validators failed projects that were correct, and other people whose validators failed to detect certain situations. So that's also something to work on for the final. We won't be releasing the stock validators. So it'll be up to you guys to find out what's wrong with your validators and fix them.

And along the same lines of correctness, once again, for the final submission, we'll be running-- actually even for the beta, I believe, we're going to Valgrind your projects and look for memory errors. So do that to your own projects and investigate any messages you get.

AUDIENCE: [INAUDIBLE]

JOHN: OK. So the highlighted column number 31 refers to the reference implementation of the validator. So that's the authority. If that's green, then your implementation is correct. And so hopefully, a correct validator would agree with column 31.

AUDIENCE: [INAUDIBLE]

JOHN: Yes.

AUDIENCE: [INAUDIBLE] question. How can it be that most of-- so an implementation is vertical, so tests are [UNINTELLIGIBLE]?

JOHN: No. The implementations are horizontal, and the tests are vertical.

AUDIENCE: So we want our column to look like column 31? Or we want--

JOHN: You want-- your validators correctness score will be determined by whether or not your column number corresponds with column 31. And then, your implementations correctness will purely be determined by whether 31 marks your row red or green. Does that make sense?

AUDIENCE: [INAUDIBLE PHRASE] columns that are all green, our validators are not [UNINTELLIGIBLE]? Is that what you're saying?

PROFESSOR: That's right.

JOHN: That's correct.

PROFESSOR: Whereas the rows that are green, that's what we like to see. We like green rows. And then, we like columns that match column 31.

AUDIENCE: [INAUDIBLE PHRASE]. The first row should be all red. And right now, [INAUDIBLE].

JOHN: Right.

PROFESSOR: That's correct.

JOHN: Whatever error this person had, very few validators seems to have caught them. Which is very surprising, because what we did for your validator.c is that we removed the line of code that it contained, and we added the comment that explained in English exactly what that line of code did. So it was kind of interesting to see that not everybody came up with the validator that's identical to reference one.

PROFESSOR: OK--

JOHN: Yeah. So please run Valgrind on your code before the final submission. And we'll be posting your personalized results to your repose sometime probably by the end of the day, either today or tomorrow.

PROFESSOR: Great. All right, you can take this [UNINTELLIGIBLE]. Or you can [UNINTELLIGIBLE]. Here you go. You guys can have it here, in case you need to chip in. OK. So today, we're going to talk about programming in parallel. Parallel programming and so forth. So this is I'm sure what you've all been waiting for.

Oops. Oh, we have no power here. There we go. There we go. Now I've got power. OK. Let's see here. How's that? Good. OK. So we talk about multicore programming. And let me start with a little bit of history.

So since the mid to late 1960s-- so how many years is that? 50 years. Wow. Semiconductor density has been increasing at the rate of about-- it's been doubling about every 18 to 24 months.

OK. So every year, every one to two years, every year and a half to two years, we get a doubling of density on the chips. And that's a trend that still is continuing. OK. So that's called Moore's law, the doubling of density of integrated circuits.

And so, this is basically a curve showing how transistor count is rising. OK. So all these green things are Intel CPUs and what the transistor count is on them. Yeah, question?

AUDIENCE: [INAUDIBLE PHRASE] the lines in [INAUDIBLE]?

PROFESSOR: So there have been some technology changes along the way. So in particular, the [UNINTELLIGIBLE] transition is back down here I think. I don't remember which one that is. Well, this is actually a different one. What we're looking at right now is the transistors, which have been very smooth. OK. So I'll explain this curve in a minute.

So there's two things plotted on here. One is the Intel CPU density, and the other is what the clock speed of those processes is. And so these are the clock speed numbers. And so, the integrated circuit technology has been-- the density has been doubling. And it's really an unbelievable sort of social and economic process, that this has basically been called a law.

Because what happens is if a-- there's so many people that contribute to making integrated circuits be dense. There's so many pieces of technology that go into that. And what happens is if you decide that you're going to try to jump and try to make something that goes faster than Moore's law, what happens is it's more expensive for you to do it. And none of the other participants in that economy can keep up. And you're just going to be more expensive. So people will op for the cheapest thing that gets the factor of two every 18 to 24 months.

Whereas if you're behind, then nobody uses your stuff. So everybody's got this sort of self-fulfilling prophecy that the rate at which the density is increasing has just been extremely stable for over 50 years. It's remarkable. Yeah, question?

AUDIENCE: [INAUDIBLE PHRASE] every six months. And somehow, [INAUDIBLE] you would have self-replicated?

PROFESSOR: No, I'm not saying that. What I'm saying is that there is some amount of everybody expecting that this is the point that everybody's going to be at. And so if you try to go more aggressively than that, you can get burned because you'll be more expensive.

If you don't go that fast, you're going to get burned because nobody's going to adopt your particular piece of the technology. And so, what happens is everybody sort of settles for this regular repeating. It's a remarkable social and economic phenomenon.

It's got very little to do at some level of technology. It's just that we know that we can improve things. But what's amazing is this growth has gone through many transitions. At one point, they said we aren't going to be able to build integrated circuits any more densely because all of the masks that were made-- it's basically, you make computers with a photographic process of exposing and using masks that you shine light through. It's the way they used to do it.

And what happened was the wave lengths of light were such that you were just simply not going to be able to get the resolutions. So what did they do? They switched to eBeams. OK. Electrons rather than photons to expose the silicon wafers and so forth.

And so, they've gone through a whole bunch of transitions and different technologies. And yet, throughout all of that, it's been just a very steady progress at about the rate of 18 to 24 months per doubling of density. And that is still going on, and is projected to go on maybe for 10 years more.

It's going to run out, I hope in my lifetime. And certainly within your lifetimes. So that has been going. Then, there's second phenomenon that has been going on since about mid-1980s. And that is that the clock speed has actually been growing on a similar curve, where basically, we've been getting 30% faster processors, clock speed, since the mid-1980s.

But something happened there, which was in around 2003, it flattened out. And the reason is, as a practical matter, clock speed for air cooled systems is bounded at somewhere around 5 gigahertz. If you want to liquid cool it or nitrogen cool it or something, you could make it go faster.

But basically, the problem is that things get too hot. And they cannot convey the heat out. So for a while, if you have greater density, the transistors get smaller. They switch faster. And you can make the clock speed go faster. But at some point, they hit the wall.

And so there the vendors were. People like Intel, AMD, Motorola. A variety of the semiconductor manufacturers. And what's happened is they can still make integrated circuits more and more dense. But they can't clock them any faster. OK.

So here's what's going on in the circuits. So here's essentially how much power was being dissipated by a variety of Intel processors along the way, and what they [INAUDIBLE] 2000. They started getting hot and hotter, until if they just continued this trend, they were going to be trying to have junction temperatures that are as hot as the surface of the sun.

Well, they clearly couldn't do that. OK. So you might say, well, let's put it off a few years. Yeah, but how many years are you going to put this off? And so, what happened was they got stuck. They simply could not make chips get clocked in faster.

So what did they decide to do? They got all the silicon area, but they can't make the processors faster with it. So their solution was to scale performance to put many processing cores on the microprocessor chip.

So this is an example of a Core i7. It's a four core. One, two, three, four cores processor. We actually have six core machines now. But I didn't update the figure.

And what's going to happen now is Moore's law is going to continue for a few more years. And so it looks like each new generation of Moore's law is going to potentially double the number of cores per chip. So you folks are using 12 core machines. Two six core chips. Well, that's going to basically keep increasing.

And so, we're going to get more and more cores per chip. OK. That's all well and good. But it turns out that there's a major issue. And that's software. Everybody has written their software. And there's billions and billions and billions of dollars invested in existing legacy software that's written for how many cores?

One. And moving it to multicore is a nightmare for these companies. OK. And it's potentially a nightmare for these vendors. Because if people say, gee, you can't make the processors go any faster, why should I buy a new processor? My old processor is as good as my new one.

OK. And so, anyway, so that's sometimes been called the multicore challenge. The multicore menace. The multicore revolution. Whatever. But that's what it's all about. It's all about the issue of the frequency scaling of the clocks, verses, Moore's law. Which talks about what the density is.

OK. So their solution is to do-- and so what we're going to talk about for a bunch of the rest of the term is going to be, how do you actually program multicore processors? We're going to look at some fairly new software technology for doing that.

So here's an abstract multicore architecture. It's not precise. This is only showing one level of cache. So we have processors connected to a cache. In fact, of course, you know that there are multiple levels of cache. Yeah, this is the international symbol for cache if you live in the US.

So the processors have their cache. Of course, you know that what actually happens is you have multiple levels of cache. And it's shared cache at some levels. OK. So it's more complex than this. But this is sort of an abstract way of understanding a bunch of the issues. And then, of course, they only get more complicated as we look at reality, as with all these hardware related things.

And so, this is a chip multiprocessor. Now there are other ways of using the silicon. So another way of using the silicon is building things like graphics processors and using silicon for a very special purpose thing. So that instead of saying, let's build multiple processors, you can say, let's dedicate some fraction of the silicon real estate. Instead of to general purpose computing, let's dedicate it to some specific purpose, like graphics, or some kind of stream processing, or what have you. Sensor processing. A variety of other things you can do.

But one main trend is doing chip multiprocessors. So we're going to talk a little bit about shared memory hardware. Just enough to get you folks off the ground to understand what's going on underneath the system. And then, we're going to talk about four concurrency platforms, which are not the only platforms one can program in. But they're ones that you should be familiar with.

The last one, Cilk++, is the one we're going to do our programming assignments in. And then, race conditions, we're going to talk about, because that's the biggest thing that comes up when you do parallel programming compared to ordinary serial programming. It's the most pernicious type of bugs. And you need to understand race conditions and need a way of handling it.

So here's basically-- so we'll start with shared memory hardware. So the main thing that shared memory hardware provides is a thing called cache coherence. OK. And the basic idea is that you want every processor to be able to fetch stuff out of local caches because that's fast.

But at the same time, you want them to have a common view of what is stored in a given location. So let's run through this example and see what the problem is. And then, I'll show you how they solve it in sketchy detail.

So here's a processor. Says he wants to load the value of x. And in main memory here, x has got the value of 3, up here in DRAM. OK. So x moves through to the processor, where it gets consumed. And it leaves behind the fact that x equals 3 in its local cache.

Well, now along comes the second processor. It says, I want x too. And perhaps the same thing happens. Very good. So far, no problem. So two caches may have the same value of x. They may both want to use x, and it's both in their local caches.

Now comes along the third processor. Says load x as well. Well, it turns out that it's actually-- what I showed you on the second case is not the common case. If these two processors, these two processing cores, are on the same chip, it's generally cheaper for this guy to fetch it out of one of these guys caches than it is to fetch it out of DRAM. DRAM is slow. Getting it locally is much cheaper.

So basically, in this case, he gets it from this processor. The first processor. All is well and good. They're all sharing merrily around. OK. And then this fella decides if he wants to load it, no problem. He can just load it. He loads it locally. No problem. OK.

This guy decides, oh, he's going to store some value to x. In this case, he's going to store the value 5. So he sets x equal to 5. OK. fine. OK, now what? Now this guy says, let me load x. He gets the value x equals 3.

Uh-oh. If your parallel program expected that this guy had gone first and it set x value x equal to 5, these guys are now incorrect. And so, the idea of cache coherence is not letting this happen, making it so that whenever a value is changed by a processor, the other processors see that change and yet, they're still able most of the time to execute effectively out of their own local caches.

OK. So that's the problem. So do people understand basically what the cache coherence problem is? Yes, question?

AUDIENCE: If the last processor was to store x and set x equals 5, as soon as that happens, wouldn't that write DRAM x equals 5?

PROFESSOR: Good. So there's actually two types of strategies that are used in caches. One is called write through. And one is called write back. What you're describing is write through. What right through caches do is if you write a value, it pushes it all the way out to DRAM. These days, nobody uses write through. You're always going to DRAM. You're always exercising the slow DRAM versus being able to just write it locally.

But you do have to do something about these guys that are going to have the shared values. So here's the mechanism that they use. So what most people do these days is write back caches. Which basically means you only write it back when you really need to evict or what have you. You don't always write it all the way through.

And so here's how these schemes work. So, right. So that's a bogus value for that kind to be getting. So let's take a look. So what they use is what's called-- the simplest is called an MSI protocol. There are somewhat more complicated ones called MESI protocols, and ones that are MOESI. "Mo-esi" and "messy".

Anyway, the MESI one is probably the one you'll hear most often. It's just a little bit more complicated than this one. But it saves you one extra access when we do a write. I'll explain it in just a minute. But let's first understand the simplest of these mechanisms.

So what you do is in each cache, you're going to label each cache line with a state. And basically, it's because of these states that you associate with a cache line that cache lines end up having to be long.

OK? Because if you think about, you'd like cache lines to be at some level very short, in that then you have more opportunity to have just the stuff in cache that you want, from a temporal locality point of view. It's one thing if you want to bring in extra lines, extra data, for spatial locality. But to insist that it all be there whether you access it or not, that's not clear how helpful that it is.

However, what instead is we have things like, on the Intel architecture, 64 bytes of cache line. And the reason is because they're keeping extra data with each cache line. And they want the data to be the larger fraction of what they're keeping compared to the control information about the data.

So in this case, they're keeping three values. Three bits. The M bit says this cache block has been modified. Somebody's written to it. And what they do is they, in this protocol, they guarantee in the protocol that if somebody has it in the M state, no other caches contain this block in either the M state or S state.

So what are those states? So the S state is when other caches may be sharing this block. And the I state is that this cache block is invalid. It's the same as if it's not there. It's empty entry. So it just marks this entry. There's no data there. The cache line that's there is not really there, is basically what it says.

So here, you see for example that this fella has x equals 13 in the modified state. And so, if you look across here, oh, nobody else has that in either the M or the S state. They only have it in the I state or not at all.

If you have it in the shared state, as these guys have, well, they all have it in the shared state and notice the values are all the same. And then, if it's in the invalid state, here this guy once again has it in the modified state, which means these guys don't have it in either the S or M state. So that's the invariant.

So what's the basic idea behind the cache? The MSI protocol? The idea is that before you can write on a location, you must first invalidate all the other copies. So whenever you try to write on something that's shared across a bunch of things or that somebody else has modified, what happens is over the network goes out a protocol to invalidate all the other copies.

So if they're just being shared, that's no problem. Because all you do is just have them drop it from the cache. If it's modified, then it may have to be written back or the value brought back to you, so that you're in a position of changing it. If somebody has it modified, then you don't have it. So therefore, you need to bring it in and make the change to it. Question?

AUDIENCE: [INAUDIBLE] three states?

PROFESSOR: Three states. Not three bits. Two bits. Right. OK. So the idea is you first invalidate the other copies. Therefore, when a processor core is changing the value of some variable, it has the only copy. And by making sure that it only has the only copy, you make sure that you never have copies out there that are anything except copies of what everybody else has. That they're all the same.

OK. Does everybody follow that? So there's hardware under there doing that. It's actually pretty clever hardware. In fact, the verification of cache protocols is a huge problem for which there's a lot of technology built to try to verify to make sure these cache protocols work the way they're supposed to work.

Because what happens in practice is there are all these intermediate states. What happens if this guy starts doing this while this guy is doing that, and these protocols start getting mixed, and so forth? And you've got to make sure that works out. And that's what's going on in the hardware.

The MESI protocol does a simple optimization. It says, look, before I store something, I probably want to read it. It's likely I'm going to read it. So I can read it in two ways. I can read it in a way that says that it is-- where it's just going to be shared. But if I expect that I'm going to write it, let me when I read it instead of getting a shared copy, let me get an exclusive copy. And that's where the E comes from. Let me get an exclusive copy. In other words, go through the invalidation protocols on the read, so that with the expectation that when you write, you don't have to then wait for the invalidation to occur at that point. So it's a way of reducing the latency of the protocol by getting it exclusively by the read that you do before you do the write.

So rather than doing a read, which would go out and get the value-- but everybody [? has them ?] shared-- then doing the write, and then doing a whole invalidation protocol, if I basically get it in exclusive mode on the read, then I go out, I get the value, and I invalidate everybody else. Now I've just saved myself half the work and half the latency. Or basically saved myself some latency. Not half the latency. OK?

So basically, what you should know is there is invalidation stuff going on behind when you start using shared memory, behind the scenes which can slow down your processor from executing. Because it can't do the things that it needs to do until it goes through the protocol. Any questions about that? That's basically the level we're going to cover the hardware at.

And so, you'll discover that in doing some your problems, that if you're not careful, you're going to create what are called invalidation storms, where you have a whole bunch of things that are red, and they're distributed across the processor. And then you go in, and you set one value. And suddenly, vrrrrrruuuum. Gee, how come that wasn't a fast store? The answer is it's going through and invalidating all those other copies. Good.

So let's turn to the real hard problem. So it turns out that building these things is not particularly well understood. But it's understood a lot better than programming these beasts. OK. And so, we're going to focus on some of the strategies for programming.

So it turns out that trying to program their processor cores directly is painful. And you're liable to make a lot of errors, as we'll see. Because we're going to talk about races soon.

And so the idea of a current currency platform is to do some level of abstraction of the processor cores to handle synchronization communication protocols, and often to do things like load balancing, so that the work that you're doing can be moved across from processor to processor.

And so, here are some examples of concurrency platforms. Pthreads and WinAPI threads, we're going to talk more in detail about. Pthreads is basically for Unix type systems, like Linux and such. WinAPI threads is for Windows.

There's threading building blocks, TBB, OpenMP, which is a standard, and Cilk++. Those are all examples of concurrency platforms that make it easier to program these parallel machines.

So I'm going to do, as an example, I'm going to use the Fibonacci numbers, which you have seen before I'm sure, because we've actually even used it in this class. This is Leonardo da Pisa, who was also known as Fibonacci. And he introduced-- he was the most brilliant mathematician of his day. He came basically out of the blue, doing all kinds of beautiful mathematics very early in the Renaissance. You'll recognize 1202 is very early Renaissance.

But it turns out, for those of you of Indian descent, the Indian mathematicians had already discovered all this stuff. But it didn't make it into Western culture except for Leonardo da Pisa. So here's a program as you might write it in C. So Fib int n says, well, if n is less than 2, return n. So if it's 0 or 1, we return, Fib of 0 is 0. Fib of 1 is 1.

And otherwise, we compute Fib of n minus 1, compute Fib of n minus 2, and return the sum. Simple recursive program. Here's the main routine. We get the argument from the command line, compute the result, and then print out Fibonacci of whatever is whatever. Pretty simple piece of code.

So what we're going to do is take a look at what happens in each of these four concurrency platforms to see how it is that they make this easy to run this in parallel. Now just a disclaimer here. This is a really bad way-- I hope you all recognize-- of computing Fibonacci numbers.

So this is exponential time algorithm. And you all know the linear time algorithm, which is basically computed up from the bottom. And some of you probably know there's a logarithmic time algorithm based on squaring matrices. Two by two matrices.

So in any case, we're all about performance here. But obviously, this is a really poor choice to do performance on. But it is a good didactic example, because it's so the structure and the issues that you get into in doing this with a very simple program that I can fit on a slide.

OK. So when you execute Fibonacci, when you call Fib of 4, it calls Fib of 3 and Fib of 2. And Fib of 3 calls Fib of 2 and Fib of 1. And Fib of 1 just returns Fib of 2, calls [UNINTELLIGIBLE] 1, 0, et cetera.

And so basically, you get an execution trace that basically corresponds to walk of this tree. So if you were doing this in C, you'd basically call this, call this, call this. Get a value return. Call this. Add the two values together. Return here. Call this. Add the two values together. Call the return there. And so forth. You walk that using a stack, a call stack, in the execution.

The key idea for parallelization is, well, gee. Fib of n minus 1 and fib of n minus 2 are really, in this calculation, completely independently calculated. So let's just do them at the same time. And they can be executed at the same time without interference, because all they're doing is basing it on n. They're not using any shared memory or anything even for this particular program.

So let's take a look, to begin with, how Pthreads might do this. So Pthreads is a standard that ANSI and the IEEE have established for-- and I actually believe this is a little bit out of date. I believe there's now a 2010 version. I'm not sure. But I recall that they were working on a new version.

But anyway, this is a recent enough standard. It's a standard that has been revised over the years, the so-called POSIX standard. So you'll hear, Pthreads is basically POSIX threads. It's basically what you might characterize as a do it yourself concurrency platform. It's kind of like assembly language for parallelism.

It allows you to do the things you need to do, but you're sort of doing it all by hand, one step at a time. It's built as a library of functions with special non-C or C++ semantics. And we'll look at what some of those semantics are.

Each thread implements an abstraction of a processor, which are multiplexed onto the machine resources by the Pthread runtime implementation. Threads communicate through shared memory. And library functions mask the protocols involved in interthread coordination. So you can start up threads, et cetera, and their library function for doing that. So let's just see how that works.

So here are, basically, the two important Pthread functions. There are actually a whole bunch of them, because they also provide a bunch of other facilities. One is pthread_create, which creates Pthread. And one is pthread_join.

So pthread_create basically is return an identifier. So when you say create a Pthread, the Pthread system says, here's a handle by which you can name this thread in the future. OK. So it's a very common thing that the implementer says, here's the name that you get. It's called a handle.

So it returns a handle. It then has an object to set various thread attributes. And for most of what we're going to need, we're just going to need NULL for default. We don't need any special things like changing the priority or what have you.

Then what you pass is a void* pointer to a function, which is going to be the routine executed after creation. So you can name the function that you want to have it operate on. And then you have a single pointer to an argument that you're going to pass to the function.

So when you call something with Pthreads to create them, you can't say, and here's my list of arguments. If you have more than one argument, you have to pack it together into a struct and pass the pointer to the struct. And this function has to be smart enough to understand how to unpack it. We'll see an example in a minute.

And then, it returns an error status. So the most common thing people do is they don't bother to check the error status. OK. And yet sometimes, you try to create a Pthread, there's a reason it can't create one. And now you keep going thinking you have one, and then your program crashes and you wonder why.

So when you create things, you should check. I'm not sure in my code here whether I checked everywhere. But you should check. Do as I say, not as I do. OK. So the other key function is join. And basically, what you do is you say, you name the thread that you want to wait for. This is the name that would be returned by the create function.

And you also give a place where it can store the status of the thread when it terminated. It's allowed to say, I terminated normally. I terminated with a given error condition or whatever. But if you don't care what it is, you just put in NULL there. And then it returns to the error status of the join function. So those are the two functions that you program with. Question?

AUDIENCE: [INAUDIBLE PHRASE]?

PROFESSOR: It's different. It's different. So it's basically, if the error status, if it returns NULL, it just means everything went OK. The handle is you pass a name, and basically this is *thread. It stuffs the name into whatever you give it.

OK so you're not saying, here's the name. This is returned as an output parameter. So you're giving it an address of some place to put the name. OK. Let's see an example. So here's Fibonacci with Pthreads. So let's just go through that. So the first part is pretty good. This is your original code that does Fibonacci. And now what we do is we have a structure for the thread arguments.

And so we're going to have an input argument and an output argument in this example. Because Fib takes an input argument in and returns Fib of n. So we're going to call those input and output. And we'll call them thread_args. And now, here is my void* function, thread_func, which takes a pointer. And what it does is when it executes-- so what you're going to be able to do is, as we'll see in a minute--. Let me just go through this. This is going to be the function called when the thread is created.

So when the thread is created, you're just going to call this function. And what it's going to get is the argument that was passed, which is this *star thing. And what it does in this case is it's basically going to cast the pointer to a thread_arg struct and dereference the input, and stick that into I. Then going to compute Fib of I. And then it's going to take, once again, deference the pointer as if it's a thread_arg, and store into the output field the result of the Fib. And then it returns NULL.

So that's basically the function that's going to be called when the thread is created. So in your main routine now, what happens is we initialize a bunch of things. And now, if argc is less than 2, we'll return 1. That's fine.

Then we're going to get the reading that we fail. That's actually the reading of the input. So then, what we do here is we get n from the command line. And then if n is less than 30, we're just going to compute Fib of n. This is what I evaluated on my laptop was a good number.

So the idea is there's no point in creating the extra thread to do the work if it's going to be more expensive than me just doing the work myself. So I looked at the overhead of thread creation and discovered that if it was smaller than 30, it's going to be slower to create another thread to help me out.

It's sort of like you folks when you're doing pair programming, which you're supposed to be doing, versus handing it off. Sometimes, there are some things that are too small to ask somebody else to do. You might as well just do it, by time you explain what it is, and so forth.

Same thing here. What's the point in starting up a thread to do something else, because the startup cost is rather substantial. So if it's less than 30, well, we'll just be done. Otherwise, what we do is we marshall the argument to the thread. We basically set args.input to n minus 1. Because args is going to be what I'm going to pass in. So I say the input number is n minus 1.

And now what I do is I create the thread by saying, give me the name of the thread that I'm creating. This was the field that I said you could put to be NULL, which basically lets you set some policy parameters and so forth. I say, execute the thread_func. This guy here. And here's the argument list that I want to provide it, which is this args thing.

Once you do the thread_create, and this is where you depart from normal C or C++ semantics. And in fact, we're going to be doing more moving in the direction of C++. We'll have some tutorials on that.

What happens is we check the status. OK, I actually did check the status to see whether or not it created it properly. But basically now, what's happening is after I execute this, it goes off and all the magic in Pthreads starts another thread doing that computation. And control returns to the statement after the pthread_create.

So when the pthread_create returns, that doesn't mean it's done computing the thing you told it to do. Then, what would be the point? It returns after it's set up to operate in parallel the other thread. People follow that?

So now at this point, there are two threads operating. There's the thread we've called thread. And there's whatever the name of the thread is that we started on. So then we, in our own processor here, we compute Fib of N minus 2. And now, what we do is we go on to join this thread with the thread that we had created.

So let's see here. And the thing that the join does is if the other thread isn't done, it sits there and waits until it is done. And it does that synchronization automatically for you. And this is the kind of thing a concurrency platform provides. It provides the coordination under the covers for you to be able to synchronize with it without you having to synchronize on your own.

And then, once it does return, it adds the results together by taking the result which came from the Fib of n minus 2 and adds to it the value that this thread has returned in the args.output. And then it prints the result.

So any question about that? Wouldn't this be fun to write a really big system in? People do. People do. Yeah, question?

AUDIENCE: [INAUDIBLE PHRASE]

PROFESSOR: That's a tuning parameter. That's a voodoo parameter.

AUDIENCE: Right. But in this particular case, it makes no difference at all. It would've made a difference if it was an actual person [INAUDIBLE]?

PROFESSOR: No, it does make a difference. For how fast it computes this? Absolutely does.

AUDIENCE: That's not recursive?

PROFESSOR: No, that's right. This is not recursive. I'm just doing two things and then quitting.

AUDIENCE: [INAUDIBLE] if it's less than 30, then it's going to be [INAUDIBLE], right?

PROFESSOR: If it's less than 30, it's fast enough that I might as well just return.

AUDIENCE: Then why [INAUDIBLE PHRASE] to do it. It would return [INAUDIBLE] too.

PROFESSOR: No. But it would be slower. It would be wasteful of resources. Maybe somebody--

AUDIENCE: Well, because you're using such a bad algorithm, I guess?

PROFESSOR: Yeah.

AUDIENCE: Oh, I see. Oh, OK.

PROFESSOR: OK. So in any case, that's Pthread's programming. There are a bunch of issues. One is that the overhead of creating a thread is more than 10,000 cycles. So it leaves you to only be able to do very coarse grain concurrency.

There are some tricks around that. One is to use what's called thread pools. What I do is I start up, and I create a bunch of threads. And I have their names. I put them in a link list. And whenever I need to create one, rather than actually creating one, I take one out of the list, much as I would do memory allocation. Which you folks are familiar with. OK. Ha, ha, ha, ha, ha. [MANIACAL LAUGHTER]

So basically, you can have a free list of threads. And when you need a thread, you grab the thread. The second thing is scalability. So this code gets about a 1.5 speed up for two cores. If I want to use three cores or four cores, what do I have to do? Rewrite the whole program. This program only works for two cores. It will also work for one core. but basically, it doesn't really exploit three or four cores.

It's really bad for modulatary. The Fibonacci logic is no longer neatly encapsulated in the Fib function. So where do we see if we go back to this code? Here's the Fib function. Oh, but now, I've kind of got-- well, this is sort of just marshaling and calling.

But over here, oh my goodness, I've got some arguments here. If n is less than 30, I give a result. Otherwise, I'm adding together-- but wait a minute. I already specified Fib up here. So I'm specifying my serial implementation, and I'm specifying a parallel way of doing it. And so that's not modular. If I decided I wanted to change the Fib, I've got to change things in two places. If Fib were something I did.

Code simplicity. The programmers for this are actually marshalling arguments. This is what I call shades of 1958. What happened in 1958 that's relevant to computer science? What was the big innovation in 1958?

Programming language. Fortran. So, Fortran. Before Fortran, people wrote in assembly language. If you wanted to put three arguments to a function, you did a push, push, push, or passed them in parameters.

Actually, their machines were so much more primitive than that it was even more complicated than you could imagine, given how complicated it is today what the compilers are doing. But you had marshal the arguments yourself. What Fortran did was say, no, you can actually write f of a, b, c. Close paren. And that it will cause a, b, and c all to be marshalled automatically for you.

Well, Pthreads doesn't have that automatic marshalling. You got to marshall by hand if you're going to use pthreads. And of course, as you can imagine, that was error prone. Because there is no type safety. Are you calling things with the right types and so forth? And so forth.

And also, one of the things here is that we've created two jobs that aren't the same size. So there's no way that they have of load balancing. So this is why pthreads is sort of the assembly language level, so that you can do anything you want in pthreads. But you have to program at this kind of very protocol-laden level.

Next thing I want to talk about is threading building blocks. This is a technology developed by Intel. It's implemented as a C++ library that runs on top of the native Pthreads, typically, or WinAPI threads.

So it's basically a layer on top of the Pthread layer. In this case, the program specifies tasks rather than threads. And tasks are automatically load balanced across the threads using a strategy called work-stealing, which we'll talk about a little bit more later. And the focus for this is on performance. They want to write programs that actually perform well.

So here's Fibonacci in TBB. So as you'll see, it's better. But maybe not ideal for what you might like to express. So what we do is we declare the computer, the computation, it's going to organized as a bunch of explicit tasks. So you say that it's going to be a task. And FibTask is going to have an input parameter, n, and an output parameters, sum.

And what we're going to do is when the task is started, it automatically executes the execute method of this tasking object here. And the execute method now starts to do something that looks very much like Fibonacci. It says if n is less than 2, sum is equal to n. That's we had before. And otherwise.

And now what we're going to do is recursively create two child tasks, which we basically do with this function, allocate_task, giving it the fib task a name, where this is basically a method for allocating out of a particular type of the pool, which is an allocate child pool.

And then similarly for b, we recursively do for n minus 2. And then what it does is it sets the number of tasks to wait for. In this case, it's basically two children plus 1 for bookkeeping. So this ends up always being one more than the things that you created as subtasks.

And then what we do is we say, OK, let's spawn. So this will only set up the task. It doesn't actually say, do it. So the spawn command says actually do this computation here that I set up. So it actually does b.

Start task b. And then itself, it executes a and waits for all of the other tasks, namely both a and b, to finish. And once it's finished, it adds the results together to produce the final output. So this, notice, has the big advantage over the previous implementation that this is actually recursive. So in doing Fib, you're not just getting two tasks. You're recursively getting each of those two more, and two more, and two more, down to the leaves of the computation.

And then what TBB does is it load balances those across the number of available processors by creating these tasks. And then, it automatically does all the load balancing of the tasks and so forth. Questions about that? Any questions?

I don't expect you to be able to program a TBB, unless I gave you a book and said, program a TBB. But I'm not going to do that. This is mainly to give you a flavor of what's in there. What the alternatives are.

So TBB provides many C++ templates that simplify common patterns. So rather than having to write that kind of thing for everything, for example, if you have loop parallelism. If you have n things that you want to have that operate parallel, you can do a parallel four and not actually see the tasks. It covers them over and creates the tasks automatically, so that you can just say, for I gets 1 to n, do this to all I, and do them at the same time essentially. And it then balances those and so forth.

It also has to things like parallel reduce. Sometimes what you want to do across an array is not just do something for every element of the array. You may want to add up all the elements into a single value. And so it basically has what's called a reduction function. It does parallel reduce to aggregate. And it's got various other things, like pipelining and filtering for doing what's called software pipelining, where you have one subsystem that basically is going to process the data and pass it to the next. So you're going to process it and pass it to the next. And it allows you to set up a software pipeline of things.

It also collides with some container classes, such as hash tables, concurrent hash tables, that allow you to have multiple tasks beating on a hash table. Inserting and deleting from the hash table at the same time and a variety of mutual exclusion library functions, including locks and atomic updates. So it has a bunch of other facilities that make it much easier to use than just using the raw task interface.

OpenMP. So OpenMP is a specification produced by an industry consortium of which the principal players-- the original principal player was Silicon Graphics, which essentially has become less important in the industry, let's say. Put it that way. And for the most part, recently, it's been players from Intel and Sun, which is now no longer Sun, except that it is Sun part of Oracle, and of IBM, and variety of other industry players.

There's several compilers available. Both open source and proprietary, including gcc, has OpenMP built-in. And also, Visual Studio has OpenMP built-in. These are a set of linguistic extensions to C and C++ or Fortran in the form of compiler practice pragmas. So who knows what a pragma is? OK. Good. Can you tell us what a pragma is?

AUDIENCE: [INAUDIBLE PHRASE]

PROFESSOR: Yeah, it's kind of like a compiler hint. It's a way of saying to the compiler, here's something I want to tell you about the code that I'm writing. And it basically is a hint. So technically, it's not supposed to have any semantic impact, but rather suggest how something might be implemented by the compiler.

However, in OpenMP's case, they actually have a compiler-- it does change the semantics in certain cases. It runs on top of native threads and it supports, especially, loop parallelism. And then, in the latest version, it supports a kind of task parallelism like we saw with TBB.

So, in fact, their task parallelism is fairly to specify. So here's the Fib code. So now, this is not looking too bad. We basically inserted a few lines here. And otherwise, we actually have the original Fibonacci code. So the sharp pragma says, here's a compiler directive. And it says, the OMP says it is an OpenMP compiler directive.

The task says, oh, the following things should be interpreted as an independent task. And now, the sharing of memory in OpenMP is managed explicitly, because they're trying to allow for programming both of distributed memory clusters, as well as shared memory machines.

And so, you have to explicitly name the shared variables that you're using. And here, we're basically saying, wait for the two things that we spawned off here to complete. So pretty simple code.

It provides many pragma directives to express common patterns, such as a parallel for parallelization. It also has reduction. It also has directives for scheduling and data sharing. And it has a whole bunch of synchronization constructs and so forth. So it's another interesting one to do.

The main downside, I would say, of OpenMP is that the performance is not really very composable. So if you have a program you've written with OpenMP over here, another one here, and you want to put them together, they fight with each other. You have to have your concept of what are going to be the programs.

The task parallelism helps a bit with that. But the basic OpenMP is very much of the model, I know how many cores I'm running on. I can set that. And then I can have it automatically parse up the work for those many.

But once you've done that, some other job, some other part of the system that wants to do the same thing, then you get oversubscription and perhaps some [UNINTELLIGIBLE]. Nevertheless, a very interesting system. And very accessible, because it's in most of the standard compilers these days.

What we're going to look at is Cilk++. So this is actually a small set of linguistics extensions to C++ to support fork-join parallelism. And it was developed by Cilk Arts, which is an MIT spin-off, which was acquired by Intel last year. So this is now an Intel technology.

And the reason I know about it is because I was the founder of Cilk Arts. It was based on 15 years of research at MIT out of my research group. And we won a bunch of awards, actually, for this work.

In fact, the work-stealing scheduler that's in it is provably efficient. In other words, it's not just a heuristic scheduler. It's actually got a mathematical proof that it's an effective scheduler. And in fact, was the inspiration for things like the work-stealing in TBB and the new task mechanisms and so forth in OpenMP, as well as a bunch of other people who've done work-stealing.

It in addition provides a hyperobject library for parallelizing code with global variables, which we'll talk about later. And it includes two tools that you'll come to know and love. One is the Cilkscreen race detector, and the other is the Cilkview scalability analyzer.

Now, what we're going to be using in this class is going to be the Cilk++ technology that was developed at Cilk Arts and then massaged a little bit when it got to Intel. There is a brand new Intel technology with Cilk built into their compiler. And it is due to come out in like, two weeks.

So our timing for this was it would've been nice to have you folks on the new Intel Cilk+ technology. But we're going to go with this one for now. It's not going to make too big a difference to you folks. But you should just be aware that coming down the pike, there's actually some much more cleanly integrated technology that you can use that's in the Intel compiler.

So here's how we do nested parallelism in Cilk++. So basically, this is Fibonacci. And now, what I have here is, if you notice, I've got two keywords, cilk_spawn and cilk_sync. And this is how you write parallel Fibonacci in Cilk. This is it.

I've inserted two key words, and my program is parallel. The cilk_spawn keyword says that the named child function can execute in parallel with the parent caller. So when you say x equals cilk_spawn or Fib of n minus 1, it does the same thing that you normally think. It calls the child. But after it calls the child, rather than waiting for it to return, it goes on to the next statement.

So then, the statement y equals Fib of n minus 2 is going on at the same time as the calculation of Fib of n minus 1. And then, the cilk_sync says, don't go past this point until all the children you've spawned off have returned.

And since this is a recursive program, it generates gobs of parallelism, if it's a big thing. So one of the key things about Cilk++, is unlike Pthreads-- Pthreads, when you say, pthread_create, it actually goes and creates a piece of work.

In Cilk++, these keywords only grant permission. They say you may execute these things in parallel. It doesn't insist that they be executed in parallel. The program may decide, no, in fact, I'm going to just call this, and then return, and then execute this.

So it only grants permission, and the Cilk++ runtime system figures out how to load balance it and schedule it. Cilk++ also supports loop parallelism. So here's an example of an in-place matrix transpose. So I want to take this matrix and flip it on its major axis.

And we can do it with for loops. As you know, for loops are not the best way to do matrix transpose. Right? It's better to do divide and conquer. But here's how you could do it. And here, I made the indices run from 0, not 1, because that's the way you do it in programming.

But if I did it up here, then these things get to be n minus 1, n minus 1, and then it gets too crowded on the slide. And I said, OK, I'll just put a comment there rather than try to sort it out. So here's what I'm saying, is this outer loop is parallel. It's going from 1 to n minus 1. And saying, do all those things in parallel. And each one is going through a different number of iterations of j.

So you can see you actually need some load balancing here, because some of these are going through just one step, and some are going through n minus 1 steps. It's basically the amount of work in every iteration of the outer loop here is different. I'm sorry?

AUDIENCE: [INAUDIBLE PHRASE].

PROFESSOR: No. i equals 1 is where you want to start. Because you don't have to move the diagonal. You only have to go across the top here. And for each of those, copy it into the appropriate column. Flip it into the appropriate column. Flip the two things.

Actually, transpose is one of these functions. I remember writing my first transpose functions. And when I was done, I somehow had the identity. Because I basically made the loops go from 1 to n and 1 to n and swapped them. So I swapped them. So I said, oh, that was a lot of work to compute the identity.

No, you've got to make sure you only go through a triangular iteration space in order to make sure you swap-- and then swap. This is an in-place swap. So that's cilk_for. That's basically it. There are some more facilities we'll talk about. But that's basically it for parallel programming in Cilk++. The other part is, how do you do it so you get fast code? Which we'll talk about.

Now, Cilk has serial semantics. And what that means is unlike some of the other ones, it's kind of what OpenMP was aspiring to do. The idea is that if I, for example here, delete these two keywords, I get a C++ code. And that code is always a legal way to execute this parallel code.

So the parallel code may have more behaviors of its nondeterministic code. But always, it's legal to treat it as if it's just straight C++. And the reason for that is that, really, we're only granting permission for parallel execution. So even though I put in these keywords, I still can execute it serially if I wish. They don't command parallel execution.

To obtain this serialization, you can do it by hand by just defining a cilk_for to be for, and the cilk_spawn and cilk_sync to be empty. Or there's a switch to the Cilk++ composite that does that for you automatically. And it's probably the preferred way of doing it.

But the idea is conceptually, you can sprinkle in these keywords, and if you don't want it anymore, fine. If you want to compile it with the straight c compilers, it's better to use the Cilk++ compiler to do it. But if you wanted to ship it off to somebody else, you could just do these sharp defines, and they could compile it with their compilers, and it would be the same as a serial C++ code.

So the Cilk++ concurrency platform allows the program to express potential parallelism in application. So it says, where is the parallelism? It doesn't say how to schedule it. It says, where is it? And then, it gets mapped onto, at runtime, dynamically mapped onto the processor cores.

And the way that it does the mapping is mathematically provably a good way of doing it. And if you take one of my graduate courses, I can teach you how that works. We'll do a little bit of study of simple scheduling. But the actual schedule it uses is more involved. But we'll cover it a little bit.

Here's the components of the Cilk++ platform on a single slide. So let me just say what they are. The first one is the keywords. So you get to put things in there. And if you elide or create the serialization, then you get the C++ code or C code, for which then you can run your regression test and demonstrate you have some good single-threaded program.

Alternatively, you can send it through the Cilk++ compiler, which is based on a conventional compiler. In our case, it will be GCC. You can link that with the hyperobject library, which we'll talk about when we start talking about synchronization. It produces a binary. If you run that binary on the runtime system, you can also run it to the regression test.

And in particular, if you run it on the runtime system, running on one core, it should behave identically to having run it through this path with just the serial code. And of course, you get exceptional performance. These, I think, were originally marketing slides.

However, there's also the fact that you may get what are called races in your code, which are bugs that will come up that won't occur in your serial code, but will occur in your parallel code. Cilk has a race detector to detect those, for which you can run parallel regression tests to produce your reliable multi-threaded code.

And then, the final piece of it is there's this thing called Cilkview, which allows you to analyze the scalability of your software. So you can run, in fact, on a single core or on a small number of cores. And then, you can predict how it's going to behave on a large number of cores.

So let's just, to conclude here, talk about races. Because they're the nasty, nasty, nasty thing we get into parallel programming. And then next time, we'll get deeper into the Cilk technology itself.

So the most basic kind of race there is what's called a determinacy race. Because if you have one of these things, your program becomes nondeterministic. It doesn't do the same thing every time.

A determinacy race occurs when two logically parallel instructions access the same memory location, and at least one of the instructions performs a write, performs a store, to that location. So here's an example.

I have a cilk_for here, both branches of which are incrementing x. This is basically going. The index is going. i equals 0 and i equals 1. And then, it's asserting that x equals 2. If I run this serially, the assertion passes. But when I run it in parallel, it may not produce a 2. It can produce a 1.

And let's see why that is. So the way to understand this code is to think about its execution in terms of a dependency [? dag ?]. So here I have my initialization of x. Then once that's done, the cilk_for loop allows me to do two things at a time, b and c, which are both incrementing x.

And then, I assert that x equals 2 when they're both done. Because that's the semantics of the cilk_for. So let's see where the race occurs. So remember that it occurs when I have two logically parallel instructions that access the same memory location. Here, it's going to be the location x. And at least one of them performs a write execution.

So if we actually looked closer, I want to expand this into this larger thing. Because as you know, X++ is not done on a memory location. It's not done as a single instruction. It's done as a load, x into a register. Increment the register, and then store the value back in.

And meanwhile, there's another register on another processor, presumably, that's doing the same thing. So this is the one I want to look at. This is just a zooming in, if you will, on this dependency graph to look a little bit finer grain at what's actually happening one step at a time.

So the determinacy race, recall, occurs-- this is by something, I'm going to say again, you should memorize. So you should know what this is. You should be able to say what a determinacy race is.

It's when you have two instructions that are both accessing the same location, and one of them performs write. And here, I have that. This guy is in parallel. He's being stored to here. This is also a race. He's been reading it, and this guy is writing it.

So let's see what can happen and what can go wrong here. So here's my value, x, in memory. And here's my two registers on, presumably, two different processors. So one thing is that you can typically-- and this is not quite the case with real hardware-- but an abstraction of the hardware is that you can treat the parallel execution from a logical point of view as if you're interleaving instructions from the different processors.

OK. We're going to talk in three or four lectures about where that isn't the right abstraction. But it is close to the right abstraction. So here, basically, we execute statement one, which causes x to become 0.

Now let's execute statement two. That causes r1 to become 0. Then, I can increment that. It becomes a 1. All well and good. But now if the next logical thing that happens is that r2 is set to the value x, then it becomes 0. Then we increment it.

And now, he stores back 1 into x. And now, this guy stores 1 back into x. And notice that now, we [UNINTELLIGIBLE] go to the assertion. And we assert that it's 2, and it's not the 2. It's a 1. Because we lost one of the updates.

Now the reason race bugs are really pernicious is, notice that if I had executed this whole branch, and then this whole branch, I get the right answer. Or if I executed this whole branch, and then this whole branch, I get the right answer. The only time I don't get the right answer is when those two things happen to interleave just so.

And that's what happens with race conditions generally, is that you can run your code a million times and not see the bug, and then run it once, and it crashes out in the field. Or what's happened is there have been race bugs responsible for failure of space shuttle to launch. You have the North American blackout of 2001? 2003?

It wasn't that long ago. It was like, 10 years ago. We had big black out caused by a race condition in the code run by the power companies. There been medical instruments that have fried people, killed them and maimed them, because of race conditions. These are really serious bugs. Question?

AUDIENCE: [INAUDIBLE] when you said, the only time that that code is actually execute serially?

PROFESSOR: It could execute in parallel if it happened that these guys executed before these guys. If you think of a larger context, a whole bunch of these things, and I have two routines where they're both incrementing x in the middle of great big parallel programs, it could be that they're executing perfectly well in parallel.

But if those two small sections of code happen to execute like this or like this, then you're going to end up with it executing correctly. But if they execute sort of at the same time, it would not necessarily behave correctly.

So there are two types of races that people talk about, a read race and a write race. So suppose you have two instructions that access a location, x. And suppose that a is parallel to b. Both a and b are both reads, you get no race. That's good. Because there's no way.

But if one is a read and one is a write, then one of them is going to see a different value, depending upon whether it occurred before and after the write. Or if they both are writing, one can lose a value. So these are read races. And this is a write race.

So we say that the two sections of code are independent if they have no determinacy races between them. So for example, this piece of code is incrementing y, and this is incrementing x. And y is not equal to x. Those are independent pieces of code.

So to avoid races, you want to make sure that the iterations of your cilk_for are independent. So what's going on in one iteration is different from what's going on in another. That you're not writing something in one that you're using in the next, for example.

Between a cilk_spawn and the corresponding cilk_sync, the code of the spawn child should be independent of the code of the parent. OK? Including any code executed by additional spawned or called children. So it's basically saying, when you spawn something off, don't then go and do something that's going to modify the same locations. You really want to modify different locations.

It's fine if they both read the same locations. But it's not fine for one of them to read and one of them to write. One thing here to understand is that when you spawn a function, the arguments are actually executed serially before the actual spawn occurs.

So you evaluate the arguments, and you set it all up, then you spawn the function. So the actual spawn occurs after the evaluation of arguments. So they're evaluated in the parent.

Machine word size matters. So this is generally the case for races. By the way, races are not just Cilk stuff. These races occur in all of these concurrency platforms. I'm illustrating Cilk because that's what we're going to be using in our labs and so forth.

So it turns out machine word size matters. And you can have races in packed data structures. So for example, on some machines, if you declare a char a and char b in a struct, then updating x and x, b in parallel may cause a race, because they're both actually operating on a word basis.

Now on the Intel architectures, that doesn't happen. Because Intel supports atomic updates of single bytes. So you don't have to worry about it. But if you were accessing bits within a word, you could end up with the same thing. You access bit five and bit three, you think you're acting independently, but in fact, you're reading the whole word or the whole byte in order to access it.

The technology that you're going to be using fortunately comes with a race detector, which you will find invaluable for debugging your stuff. And so this is kind of like a Valgrind for races. What's good about this race detector is it provides a rock hard guarantee.

If you have a deterministic program that on a given input could possibly behave any differently from your serial program, from the corresponding serial program, if you got rid of the parallel keywords, this tool, Cilkscreen, guarantees to report and localize the offending race. It'll tell you, you got a race between this location and that location. And it's up to you to find it and fix it, but it can tell you that.

It employs regression test methodology, where the programmer provides test inputs. So if you don't provide test inputs to elicit the race, you still can have a bug. But if you have a test input that in any way could behave differently than the serial execution, bingo. It'll tell you.

It identifies a bunch of things involving the race, including a stack trace. It runs off the binary executable using what's called dynamic instrumentation. So that's kind of like Valgrind, except it actually does this as it's running. It uses a technology called PIN, which you can read about. P-I-N, which is a nice platform for doing code rewriting and analysis on the fly.

It runs about 20 times slower than real time. So you basically use it for debugging. So the first part of project four is basically coming up to speed with this technology. And so, there's some good things. And that's going to be available tomorrow. Is that what we said? Yeah, that will be available tomorrow.

So this is actually-- this is tons of fun. Most people in most places don't get to play with parallel technology like this.