PROFESSOR: I guess [OBSCURED] Let's get going. OK, should I introduce you?

BRADLEY KUSZMAUL: If you want. I can introduce myself.

PROFESSOR: We have Bradley Kuszmaul who's been doing articles on Cilk? He's a very interesting paralleling and also what you can say about the program It's a very interesting project that coming for a while, and there's a lot of interesting things he's developed, and multi core becoming very important.

BRADLEY KUSZMAUL: So how many of you people have ever heard of Cilk? Have used it? So those of you who have used it may find this talk old or whatever. So Cilk is a system that runs on a shared-memory multiprocessor. So this is not like the system you've been programming for this class. This kind of machine you have processors, which each have cache and some sort of a network and a bunch of memory and when the processors do memory operations they are all on the same address space and it's typically-- the memory system provides some sort of coherence like strong consistency or maybe released consistency. We're interested in the case where the distance from processors to other processors into a processors to memory may be nonuniform and so it's important to use the cache well in this kind of machine because you can't just ignore the cache. So sort of the technology that I'm going to talk about for this kind of system is called Cilk. Cilk is a C language and it does dynamic multithreading and it has a provably good runtime system. So I'll talk about what those all mean.

Cilk runs on shared-memory machines like Suns and SGIs and well, you probably can't find Alphaservers anymore. It runs on SMPs like that are in everybody's laptops now. There's been several interesting applications written in Cilk including virus shell assembly, graphics rendering, n-body simulation. We did a bunch of chess programs because they were sort of the raison d'etre for Cilk. One of the features about Cilk is that it automatically manages a lot of the low-level issues. You don't have to do load balancing, you don't have to write in protocols. You basically write programs that look a lot more like the ordinary Cilk programs

instead of saying first I'm going to do this and then I'm going to set this variable and then somebody else is going to read that variable and that's a protocol and those are very difficult to get right.

AUDIENCE: [OBSCURED]

BRADLEY KUSZMAUL: Yeah, I'll mention that a little bit later. We had award-winning chess player. So to explain what Cilk's about I'll talk about Fibonacci. Now Fibonacci, this is just to review in case you don't know C. You all know C right? So Fibonacci is the function that each number is the sum of the previous two Fibonacci numbers. And this is an implementation that basically does that computation directly. The Fibonacci of n if n is less than 2, it's just n. So Fibonacci of zero is zero, 1 is 1. 2, the Fibonacci's-- well, then you have to do the recursion, so you compute Fibonacci of n minus 1 and Fibonacci of n minus 2 and sum them together and that's Fibonacci of n. One observation about this function is it's a really slow implementation of Fibonacci. You all know how to do this faster? How fast can you do Fibonacci? You all know this, How fast is this one?

AUDIENCE: [OBSCURED].

BRADLEY KUSZMAUL: So for those of you who don't know-- certainly know how to compute Fibonacci in linear time just by keeping track of the most recent two. 1, 1, 2, 3, 5, you just do it. This is exponential time and there's an algorithm that does it in logarithmic time. So this implementation is doubly, exponentially bad. But it's good as a didactic example because it's easy to understand. So to turn this into Cilk we just add some key words and I'll talk about what the key words are in a minute, but the key thing to understand about this is if you delete the key words you have a C program and Cilk programs have the property that one of the legal semantics for the Cilk program is the C program that you get by deleting the key words. Now there's other possible semantics you could get because-- not for this function, this function always produces the same answer because there's no race conditions in it. But for programs that have races you may have other semantics that the system could provide. And so this kind of a language extension where you can sort of delete the extensions and get a correct implementation of the parallel program is called a faithful extension. A lot of languages like OpenMP have properties that if you had these directives and if you delete them, it will change the semantics of your program and so you have to be very careful.

Now if you're careful about programming OpenMP you can make it so that it's faithful, that has

this property. But that's not always the case that it is. Sure.

AUDIENCE: Is it built on the different..

BRADLEY KUSZMAUL: C 77. No, C 89.

AUDIENCE: OK, so there's no presumption or any alias involved? It's assumed that the [OBSCURED].

BRADLEY KUSZMAUL: So the issue of restricted pointers, for example?

AUDIENCE: Restricted pointers.

BRADLEY KUSZMAUL: So Cilk turns out to work with C 99 as well.

AUDIENCE: But is the presumption though for a pointer that it could alias?

BRADLEY KUSZMAUL: The Cilk compiler makes no assumptions about that. If you write a program and the back end-- Cilk works and I'll talk about this in a couple minutes. Cilk works by transforming this into a C program that has-- when you run it on one processor it's just the original C program in effect. And so if you have a dialect of C that has restricted pointers and a compiler that--

PROFESSOR: You're taking the assumptions that if you make a mistake--

BRADLEY KUSZMAUL: If you make a mistake the language doens't stop you from making the mistake.

AUDIENCE: Well, but in C 89 there's not a mistake. There's no assumption about aliasing, right? It could alias. So if I said --

BRADLEY KUSZMAUL: Because of the aliasing you write a program that has a race condition in it, which is erroneous--

AUDIENCE: It would be valid?

BRADLEY KUSZMAUL: No, it'd still be valid. It would just have a race in it and you would have a non-determinate result.

PROFESSOR: It may not do what you want.

BRADLEY KUSZMAUL: It may not do what you want, but one of the legal executions of that parallel program is the original C program.

AUDIENCE: So there's no extra.

BRADLEY KUSZMAUL: At the sort of level of doing analysis, Cilk doesn't do analysis. Cilk is a compiler that compiles this language and the semantics are what they are, which is you the spawn is its-- and I'll talk about the semantics. The spawn means you can run the function in parallel and if that doesn't give you the same answer every time it's not the compilers fault.

AUDIENCE: [OBSCURED]

BRADLEY KUSZMAUL: Pardon?

AUDIENCE: There has to be some guarantee [OBSCURED].

[OBSCURED]

PROFESSOR: How in a race condition you get some [OBSCURED].

BRADLEY KUSZMAUL: One of the legal things the Cilk system could do is just run this, run that program. Now if you're running it on multiple processors that's not what happens because the other thing is there's some performance guarantees we get. So there's actually parallelism. But on one processor in fact, that's exactly what the execution does. So Cilk does dynamic multithreading and this is different from p threads for example where you have this very heavyweight thread that costs tens of thousands of instructions to create. Cilk threads are really small, so in this program there's a Cilk thread that runs basically from when the fib starts to here and then-- I feel like there's a missing slide in here. I didn't tell you about spawn.

OK, well let me tell you about spawn because what the spawn means is that this function can run in parallel. That's very simple. What the sync means is that all the functions that were spawned off in this function all have to finish before this function can proceed. So in a normal execution of C, when you call a function the parent stops. In Cilk the parent can keep running, so while that's running the parent-- this can spawn off this and then the sync happens and now the parent has to stop. And this key word basically just says that this function can be spawned.

AUDIENCE: Is the sync in that scope or the children scope?

BRADLEY KUSZMAUL: The sync is scoped within the function. So you could have a 4 loop that spawned off a whole bunch of stuff.

AUDIENCE: You could call the function instead of moving some spawns, but then [OBSCURED] in the sync.

BRADLEY KUSZMAUL: There's an explicit sync at the end of every function. So Cilk functions are strict.

PROFESSOR: [NOISE]

BRADLEY KUSZMAUL: You know, there's children down inside here, but this function can't return-- well, if I had omitted the sync and down in some leaf the compiler puts one in before the function returns. There's some languages that are like this where somehow the intermediate function can go away and then you can sync directly with your grandparent.

AUDIENCE: Otherwise it would stop.

BRADLEY KUSZMAUL: So this gives you this dag, so you have this part of the program that runs up to the first spawn and then part of the program that runs between the spawns and the part of the program that runs after-- well, after the last spawn to the sync and then from there to the return. So I've got this drawing that shows this function sort of running. So first the purple code runs at it gets to the spawn, it spawns of this guy, but now the second piece of code can start running. He does a spawn, so these two are running in parallel. Meanwhile. This guy started that pff. This is a base case, so he's going to not do anything. Just feels like there's something missing in this slide. Oh well.

Essentially now this guy couldn't run going back to here. This part of the code couldn't run until after sync so this thing's sitting here waiting. So when these guys finally return then this can run. This guy's getting stuck here. He runs and he runs. These two return and the value comes up here. And now basically the function is done. One observation here is that there's no mention of the number of processors in this code. You haven't specified how to schedule or how many processors. All you've specified is this directed acyclic graph that unfolds dynamically and it's up to us to schedule those onto the processors. So this code is processor oblivious. It's oblivious to the number of processors.

PROFESSOR: But because we're using the language we're probably have to create, write as many spawns depending on--

BRADLEY KUSZMAUL: No, what you do is you write as many spawns as you can. You expose all the parallelism in your code. So you want this dag to have millions of threads in it concurrently. And then it's up to us to schedule that efficiently. So it's a different mindset then, I have 4 processors, let me create 4 things to do. I have 4 processors, let me create a million things to do. And then the Cilk scheduler guarantees to give you-- you have 4 processors, I'll give you 4 fold speed up.

PROFESSOR: I guess what you'd like to avoid is the mindset of the programmer has to change or find the changing tuning the parameters for the performance.

BRADLEY KUSZMAUL: There's some tuning that you do in order to make the leaf code. There's some overhead for doing function calls. So it's small overhead. It turns out the cost of the spawn is like three function calls. If you were actually trying to make this code run faster you make the base case bigger and do something, trying to speed things up a little bit with the leaves of this call. So there's this call tree and inside the call tree is this dag. So it supports C's rule for pointers. For whatever dialect you have. If you have a pointer to a stack and then you have a pointer to the stack and then you call, you're allowed to use that pointer in C. So in Cilk you are as well. If you have a parallel thing going on where normally in C you would call A, then B returns, then C and D. So C and D can refer to anything on A, but C can't legally refer to something on B and the same rule applies to Cilk. So we have a data structure that implements this cactus stack is what it's called, after the sugauro cactus-- the view of the imagery there and it lets you support that rule.

There's some advanced features in Cilk that have to do with speculative execution and I'm going to skip over those today because it turns out that sort of 99% of the time you don't need this stuff. We have some debugger support, so if you've written code that relied on some semantics that maybe you didn't like when you went to the parallel world, you'd like to find out. This is a tool that basically takes a Cilk program and an input data set and it runs and it tells you is there any schedule that I could have chosen-- so it's that directed acyclic graph. So there's a whole bunch of possible schedules I could have chosen. Is there any schedule that changes the order of two concurrent memory operations where one of them is right? So we call this the non-determinator because it finds all the determinacy races in your program. And Cilk guarantees-- the Cilk race detector is guaranteed to find those. There's a lot of race detectors where if the race doesn't actually occur you have two things that are logically in

parallel, but if they don't actually run on different processors a lot of race detectors out there in the world won't report the race. So you get false negatives and there's a bunch of false positives that show up. This basically only gives you the real ones.

AUDIENCE: That might be indicatiors there might be still a data to arrays.

BRADLEY KUSZMAUL: So this doesn't analyze the program. It analyzes the execution. So it's not trying to solve some MP complete problem or Turing complete problem. And so this reduces the problem of finding data races to the situation that's just like when you're trying to do code release and quality control for serial programs. You write tests. If you don't test your program you don't know what it does and that's the same property here. If you do find some race someday later then you can write a test for it and know that you're testing to make sure that race didn't creep back into your code. That's what you want out of a software release strategy.

AUDIENCE: [NOISE]

BRADLEY KUSZMAUL: If you start putting in sync than maybe the race goes away because of that. But if just put in instrumentation to try to figure out what's going, it's still there. And the race detector sort of says, this variable in this function, this variable in this function, you look at it and say, how could that happen? And finally you figured out and you fix it and then you put it-- if you're trying to do software release you build a regression test that will verify that has that input.

AUDIENCE: What if you have a situation where the spawn graph falls into a terminal. So it's not a radius, but monitoring spawn is there but it spawns a graph a little bit deeper.

BRADLEY KUSZMAUL: Yes. For example, our race detector understands locks. So part of the rule is it doesn't report a race if the two memory accesses-- if there was a lock that they both held in common. Now you now you can write buggy programs because you can essentially do the memory at lock, you know, read the memory, unlock, lock, write the memory. Now the interleave happens and there's a race. So the assumption of this race detector is that if you put locks in there that you've sort of thought about. This is finding races that you forgot about rather than races that you ostensibly thought about. There are some races that are actually correct. For example, in the chess programs there's this big table that remembers all the chess positions that have been seen. And if you don't get the right answer out of the table it doesn't matter because you search it again anyway. Not getting the right answer means you don't get any answer. You look something up and it's not there so you search again. If you just waited a little longer maybe somebody else would have put the value in, you could have saved a little work. And so

in that case, well it turns out to be there's no parallel way to do that. So I'm willing to tolerate that race because that gives me performance and so you have what we call fake locks, which are basically things that look like lock calls, but they don't do anything except tell the race detector, pretend there was a lock held in common. Yeah?

AUDIENCE: [UNINTELLIGIBLE PHRASE]

BRADLEY KUSZMAUL: If it says there's no race it means that for every possible scheduling that--

AUDIENCE: [UNINTELLIGIBLE PHRASE].

BRADLEY KUSZMAUL: Well, you have that dag. And imagine running it on one processor. There's a lot of possible orders in which to run the dag. And the rule is well, was there a load in a store or a store in a store that switched orders in some possible schedule and that's the definition.

AUDIENCE: So, in practice, sorry, one of the [INAUDIBLE] techniques is loss. Assuming, dependent on the processor, that you have atomic rights, we want to deal with that data [UNINTELLIGIBLE] in the background --

BRADLEY KUSZMAUL: Those protocols are really hard to get right, but yes, it's an important trick.

AUDIENCE: Certainly [INAUDIBLE].

BRADLEY KUSZMAUL: So to convince the race detector not to complain you put fake locks around it. You've programmed a sophisticated algorithm it's up to you to get the details right. The other property about this race detector is that it's fast. It runs almost in liear time. A lot of the race detectors that you find out there run in quadratic time. So if you want to run a million instructions it has to compare every instruction to every other instruction. Turns out we don't have to do that. We run in time, which is n times alpha of n where alpha's the inverse Ackermann function. Anybody remember that from the union-find algorithm. It's got that graded So it's like the almost linear time. We actually now have a linear timed one that has performance advantages. So let me do a little theory in practice.

In Cilk we have some fundamental complexity measures that we worry about. So we're interested in knowing and being able to predict the runtime of a Cilk program on P processors. So we want to know T sub p, which is the execution time on P processors. That's the goal.

What we've got to work with is some directed acyclic graph that is for a particular input set and if the program determines it and everything else it's a well defined graph and we can come up with some basic measures of this graph. So T sub 1 is the work of the graph, which is the total time it would take to run that graph on one processor. Or if you assume that these things are all cost unit times, just the number of nodes. So for this graph what's the work?

I heard teen, but something-- 18? And the critical path is the longest path. And if these nodes weren't unit time you'd have to weight the things according to actually how much time they run. So the critical path here is what? 9. So I think those are right. The lower bounds then that you know is that you don't expect the runtime on P processes to be faster than linear speedup. In this model that doesn't happen. It turns out cache does things. It's adding more than just processors. You're adding more cache too. So all sorts of things or maybe it means that there's a better algorithm you should have used. So there's some funny things that happen if you have bad algorithms and so forth. But in this model you can't have more than linear speedup. You also can't get things done faster than in linear time. This model assumes basically that these costs of running these things are fixed and the cache has the property that changing the order of execution means that the actual costs of the nodes in the graph change costs. So those are lower bounds and the things that we want to know are speedups, so that's T sub 1 over T sub p. And the parallelism of the graph is T sub 1 over T sub infinity. So the work over the critical path and we've been calling this span sometimes lately. Some people call that depth. Span is easier to say than critical path, depth has too many other meanings so I kind of like span.

So what's the parallelism for this program? 18/9. We said that T sub 1 was what? 18. The infinity is 9. So on average and if you had an infinite number of processors and you scheduled this as greedy as you good, it would take you 9 steps to run and you would you be doing 18 things worth of work. So on average there's two things to do. You know, 1 plus 1 plus 1 plus 3 plus 4 plus 4 plus 1 plus 1 plus 1 divided by 9 turns out to be 2. So the average parallelism or just the parallelism of the program is T sub 1 over T sub infinity. And this property is something that's not dependent on the scheduler, it's a property of the program. Doesn't depend on how many processors you have.

AUDIENCE: [OBSCURED] You're saying, you're calling that the span now? Is that the one for us [OBSCURED]

BRADLEY That's too long to say. I might as well say critical path length. Critical path length, longest trace

KUSZMAUL: span is a mathematical sounding name.

AUDIENCE: We just like to steal terminology.

BRADLEY KUSZMAUL: Well, yeah. So there's a theorem due to-- Graham and Brent said that there's some schedule that can actually achieve the sum of those two lower bounds. This linear speedup is one lower bound of the runtime and the critical path is the other. So there's some schedule that basically achieves the sum of those and how does that theorem work? Well, at each time step either-- suppose we had 3 processors. Either there's at least 3 things ready to run and so what you do is you do a greedy schedule. You grab any 3 of them. If there's fewer than p things to run, like here we have a situation where these have all run. The green ones are ready to go. Those are the only 2 that are ready to go. So what do you do then in a greedy schedule? You run them all. And the argument goes, well, how many times steps could you execute 3 things? At most you could do it the work divided by the number of processors times because then after that you've used up all the work. Well how many times could you execute less than p things? Well, every time you execute less than p things you're reducing the length of the remaining critical path. You can't do that more than the span times. And so a greedy scheduler will achieve some runtime which is within the sum of these 2. It's actually the sum of these 2 minus 1. It turns out that there has to be at least one node that's on both work and critical path. And so that means that you're guaranteed to be within a factor of 2 of optimal with a greedy schedule.

And it turns out that if you have a lot of parallelism compared to the number processors, so if you have a graph that has a million fold parallelism and a thousand fold processors Well, if this is really small compared to the work, if you have a graph with a million fold parallelism that means the critical path is small. If you only had 1000 processors that means this term's big. And that means that this term is very close to this term, so essentially the corollary to this is that you get linear speedup, perfect linear speed asymptotically if you have fewer processors then you have parallelism in your program. So the game here at this level of understanding, I haven't told you how the scheduler actually works-- is to write a program that's got a lot of parallelism that you can get linear speedup. Well, the work-stealing scheduler we actually use. The problem is the greedy schedulers can be hard to compute-- especially if you imagine having a million processors in a program with a billion fold parallelism. Finding on every clock cycle, finding something for each of the million guys to do is conceptually difficult, so instead we have a work-stealing scheduler. I'll talk about that in a second. It achieves bounds which are not quite as good as those. This bound is the same. It's the sum of two terms. One is the

linear speedup term, but instead of it being T sub infinity it's big O of T sub infinity because you actually have to do communication sometimes if the critical path length is long.

Basically, you can sort of imagine. If you have a lot of things to do, a lot of tasks and people to do it, it's easy to do that in parallel if there's no interdependencies among the tasks. But as soon as there's dependencies you end up having to coordinate a lot and that communication costs-- there's lots of lore about adding programmers to a task and it slowing you down. Because basically communication gets you. What we found empirically-- there's a theorem for this-- empirically the runtime is actually still very close to the sum of those terms. Or maybe it's those terms plus 2 times T sub infinity or something like that. And again, we basically get near-perfect speedup as long as the number of processors is a lot less than the parallelism. Should be sort of a less than less than.

The compiler has the mode where you basically can insert instrumentations. So you can run your program, it'll tell you the critical path length. You can compute these numbers. Clear how to compute work, you just sum up the runtime of all the threads. To compute the critical path length, well you have to do some max's and stuff as you go through the graph. And the average cost of a spawn these days is about 3 on like a dual core pentium. Three times the cost of a function call. And most of that cost actually has to do with the memory barrier that we do at the spawn because that machine doesn't have strong consistencies. So you have to put this memory barrier in and that just empties all the pipelines. It does better on like an SGI machine, which has strong-- well, traditional. A MIPS machine that has strong consistency actually does better for the cost of that overhead.

Let me talk a little bit about chess. And we had a bunch of chess programs. I wrote one in 1994, which placed third at the International Computer Chess Championship and that was running on a big connection machine CM5. I was one of the architects of that machine, so it was double fun. We wrote another program that placed second in '95 and that was running on an 1800 node Paragon and that was a big computer back then. We built another program called Cilk chess, which placed first in '96 running on a relatively smaller machine. And then on a larger SGI origin we ran some more and then at the World Computer Chess Championship in 1999 we beat Deep Blue and lost to a PC. And people don't realize this, but at the time that Deep Blue beat Kasparov it was not the World Computer Chess Champion, a PC was. So what? It's running a program. You know, there's this head and a tape. I don't know what it did.

So this was a program called Fritz, which is a commercially available program. And those guys

were very good, the PC guys playing were very good at getting on sort of the algorithm side. We got advantage by brute force. And we also had some real chess expertise on our team, but those guys were spending full time on things like pruning away sub-searches that they were convinced weren't going to pan out. Computer chess programs spend most of their time looking at situations that any person would look at and say, ah, blacks won. Why are you even looking at this? And it keeps searching. It's like, well maybe there's a way to get the queen. So computers are pretty dumb at that. So basically these guys put a lot more chess intelligence in and we also lost due to what-- in this particular game, we were tied for first place and we decided to do a runoff game to find out who would win and we lost due to a classic horizon effect. So it turns out that we were searching to depth 12 in the tree and Fritz was searching to depth 11. Even with all these heuristics and stuff they had in it, they were still not searching as deeply as we were. But there was a move that was a good move that looked OK at depth 11 and looked bad at depth 11 and at depth 13 it looked really good again. So they saw the move and made it for the wrong reason, we saw the move and didn't make it for the right reason, but it was wrong and the right move-- if we'd been able to search a little deeper, we would have seen that it was really the wrong thing to do. This happens all the time in chess. There's a little randomness in there. This horizon effect shows up and again, it boils down to the programs are not intelligent. A human would look at it and say, eventually that knight's going to fall. But if the computer can't see it with a search, you know?

We plotted the speedup of star Socrates, which was the first one on this funny graph. So this looks sort of like a typical linear speedup graph. Sort of when you're down here with few numbers processors you get good linear speedup and eventually you stop getting linear speedup. That's sort of in broad strokes what this graph looks like. But the axes are kind of funny. The axes aren't the number of processors and the speedup-- it's the number processors divided by the parallelism of the program. And here is the speedup divided by the parallelism of the program. And the reason we did that is the each of these data points is a different program with different work in span. If I'm trying to run a particular problem on a bunch of different processors I can just draw that curve and see what happens as get more processors. I'm not getting any advantage because I've got too many processors. I've exceeded the parallelism of the program. But if I'm running, trying to compare two different programs, how do I do that?

Well, you can do that by normalizing by the parallelism. So down in this domain the number of processors is small compared to the average parallelism and we get good linear speedups.

And up in this the domain the number of processors is large and it starts asymptoting to the point where the speedup approaches the parallelism and that's sort of what happened. You get some noise out here so one of the things down here, it's nice and tight. And that's because we're in that domain where the communication costs are infrequently paid because there's lots of work to do. You don't have to communicate very much. Up here there's a lot of communication that happens and so the noise is showing up more in the data. This curve here is the T sub 1 over P plus T sub infinity curve. The T sub P equals T sub infinity curve and that's the linear speedup curve on this graph. So I think there's an important lesson in this graph besides the data itself, which is if you're careful about choosing the axes, you can take a whole bunch of data that you couldn't see how to plot it together and you can plot it together and get something meaningful. So in my Ph.D. thesis I had hundreds of little plots for each chess position and I didn't figure out how-- it's like they all look the same, right? But I didn't sort of figure out that if I was careful I could actually make them be the same. That happened after I published my thesis. Oh, we could just overlay them. Well, what's the normalization that makes that work?

So there's a speedup paradox that happened. Pardon?

AUDIENCE: [OBSCURED]

BRADLEY KUSZMAUL: Yeah, OK. There was a speedup paradox that happened while we were developing star Socrates. We were developing this for 512 processor connection machine that was at University of Illinois, but we only had a smaller machine on which to do our development. We had a 128 processor machine at MIT and most days I could only get 32 processors because the machine was in heavy demand. So we had this program and it ran on 32 processors in 65 seconds. And one of the developers said, here's a variation on the algorithm, it changes the dag. It's a heuristic. It makes the program run more efficient. Look, it runs in only 40 seconds on 32 processors. And so is that a good idea? It sure seemed like a good idea, but we were worried that we knew that the transformation increased the critical path length of the program, so we weren't sure it was a good idea.

So we did some calculation. We measured the work and the speedup. And so the work here-- these numbers have been cooked a little bit to make the math easy, but the numbers-- this really did happen, but not with these exact numbers. So we had a work which was 2048 seconds and only 1 second of critical path. And over this new program had only 1/2 as much work to do, but the critical path length was longer. It was 8 seconds long. If you predict on 32

processors what the runtime's going to be that formula says well, 65 seconds. If you predict it on 32 processors this-- well, it's 40 seconds and that looks good, but we were going to be running the tournament on 512 processors where this term would start being less important than this term. So this really did happen and we actually went back and validated that these numbers were right after we did the calculation and it allowed us to do the engineering to make the right decision and not be misled by something that looked good in the test environment. We were able to predict what was going to happen on the big machine without actually having access to the big machine and that was very important.

Let me do some algorithms. You guys probably have done some matrix multipliers over the past 3 weeks, right? That's probably the only thing you've been able to do would be my guess. So matrix multiplication is this operation. I won't talk about it, but you know what it is. In Cilk instead of doing the standard triply nested loops you do divide and conquer. We don't parallelize loops we parallelize function calls, so you want to express a loops as recursion. So to multipliy two big matrices you do a whole bunch of little matrix multiplications of the sub-blocks and then you express those little matrix multiplications themselves and go off and recursively do smaller matrix multiplications. So this requires 8 multiplications of matrices these of 1/2 the number of rows and 1/2 the number columns an one edition at the end where you add these two matrices together. That's the algorithm that we do, it's the same total work as the standard one, but it's just expressed recursively. So a matrix multiply is you do these 8 multiplies. I had to create a temporary variable, so the first four multiplies the A's and B's into C. The second four multiply the A's and B's into T and then I have to add T into C. So I do all those spawns, do all the multiplies. I do a sync because I better not start using the results on the multiplies and adding them until the multiplies are done.

AUDIENCE: Which four do you add?

BRADLEY KUSZMAUL: What? There's parallelism in add. Matrix addition.

AUDIENCE: Yeah, but it doesn't add spawn extent

BRADLEY KUSZMAUL: Well, we spawn off add. I don't understand--

[INTERPOSING VOICES]

BRADLEY KUSZMAUL: So you have to spawn Cilk functions even if you're only executing one of them at a time. Cilk functions are spawned, C functions are called. It's a decision that's built into the language. It's not really a fundamental decision. It's just that's the way we did it. AUDIENCE: Why'd you choose to have the key word then? That's just documentation from the caller side?

BRADLEY KUSZMAUL: Yeah, we found we were less likely to make a mistake if we sort of built it into the type system in this way. But I'm not convinced that this is the best way to do this type system.

AUDIENCE: Can the C functions spawn a Cilk function.

BRADLEY KUSZMAUL: No. You can only call spawn, spawn, spawn, spawn then you can call C functions at the leaves. It turns out you can actually spawn Cilk functions if you're a little clever about-- there's a mechanism for a Cilk system running in the background and if you're running C you can say OK, do this Cilk function in parallel. So we have that, but that's not didactic.

AUDIENCE: Sorry, I have a question about the sync spawning. Is the sync actually doing a whole wave or -- like, in the case of-- maybe not in the case of the add here, but in plenty of other practical functions you get inside the spawn function looking at the tendencies of the parameters, right? Based on how those were built from previous spawned funcitons. You can actually just start processing so long as it's guaranteed that the results are available before you actually read them.

BRADLEY KUSZMAUL: So there's this other style of expressing parallelism which you see in some of the data flow languages where you say well, I've computed this first multiply, why can't I get started on the corresponding part of the addition. And it turns out that in those models there's no performance guarantees. The real issue is you run out of memory. It's a long topic, let's not go into it, but there's a serious technical issue with those programming models. We have very tight memory bounds as well, so we simultaneously get these good scheduling bounds and good memory bounds and if you are doing that you could have sort of a really large number of temporaries required and run out of memory. The data flow machine used to have this number-- there was a student, Ken Traub, who was working on Monsoon when Greg Papadapolous was here and he came up with this term which we called Traub's constant, which was how long the machine could be guaranteed to run before it crashed from being out of memory. And that was-- well, he took the rate at which it Kahn's divided by the amount of memory and that was it. And many data flow programs had that property that Monsoon could run for 40 seconds and then after that you never knew. It might start crashing at any moment,

so everybody wrote short data flow programs.

So one of the things you actually do when you're implementing, when you're trying to engineer this to go fast, is you course in the base case, which I didn't describe up there. You don't just do a 1 by 1 matrix multiplied down there at the leaves of this recursion. Because then you're not using the processor pipeline efficiently. You call the Intel Math Kernel Library or something on an 8 by 8 matrix so that it really gets the pipeline a chance to chug away.

So analysis. This matrix addition operation-- well, what's the work for matrix addition? Well the work to do a matrix operation on n rows is well, you have to do 4 additions of size n over 2. Plus there's order 1 work here for the sync. And that recurrence has solution order n squared. Well, that's not surprising. You have to add up 2 matrices which are n by n. That's going to be n squared so that's a good result. The critical path for this is well, you have to do all of these in parallel. So whatever the critical path of the longest one is, they're all the same so it's just the critical path of the size n over 2 plus quarter 1, so the critical path is order log n.

For matrix multiplication, sort of the reason I do this is I can. This is a model which I can do this analysis, so I have to do it. But really, being able to do this analysis is important when you're trying to make things run faster. Matrix multiplication, well, the work is I have to do 8 little matrix multiplies plus I have to do the matrix add. The work has solution order n cubed and everybody knows that there's order n cubed multiply adds in a matrix multiplier, so that's not very surprising. The critical path is-- well, I have to do a add so that takes log n, plus I have to do a multiply on a matrix that's 1/2 the size. So the critical path length of the whole thing has solution order log squared n. So the total parallelism of matrix multiplication is the work over the span, which is n cubed over log squared n. So if you have a 1000 by 1000 matrix that means your parallelism is close to 10 million. There's a lot of parallelism and in fact, we see perfect linear speedup on matrix multiply because there's so much parallelism in it.

It turns out that this stack temporary that I created so that I could do these multiplies all in parallel is actually costing me work because I'm on a machine that has cache and I want to use the cache effectively. So I really don't want to create a whole big temporary matrix and blow my cache out if I can avoid it. So I proposed the following matrix multiply, which is I first do 4 of the matrix multiplies into C1 then I do a sync and then I do the other 4 into C1 and another sync. And I forgot to do the add-- oh, no those are multiply adds so they're multiplying and adding in. And this saves space because it doesn't need a temporary, but it increases the critical path. So is that a good idea about or a bad idea?

Well, we can answer part of that question with analysis. Saving space we know is going to save something. What does it do to the work in critical path? Well, the work is still the same, it's n cubed because we didn't change the number of flops that we're doing. But the critical path has grown. Instead of doing 1 times a matrix multiply, we have to do one and then sync and then do another one. So it's 2 matrix multiplies of 1/2 the size plus the order 1 and that recurrence has solution order n instead of order log squared n. So that sounds bad, we've made the critical path longer.

AUDIENCE: [OBSCURED]

BRADLEY KUSZMAUL: What? Yeah. So parallelism is now order n squared instead of n cubed over log squared n and for a 1000 by 1000 matrix that means you still have a million fold parallelism. So for relatively modest sized matrices you still have plenty of work to do this optimization. So this is a good transformation to do it. One of the advantages of Cilk is that you can do this kind of You could say, let me do an optimization. I can do an optimization in my C code and I get to take advantage of it in the Cilk code. I could do this kind of optimization of trading work for parallelism. If I have a lot of work that sometimes is a good idea. Ordinary matrix multiplication just is really bad. Basically you can imagine spawning off the n squared inner dot products here and computing them all in parallel. It has work n cubed parallelism log n. I mean, critical path log n so the parallelism's even better. It's n cubed over log n instead of n squared. That looks better theoretically, but it's really bad in practice because it has such poor cache behavior. So we don't do that.

I'll just briefly talk about how it works. So Cilk does work-stealing. We had did double ended queue-like decque. So at the bottom of the queue is the stack where you push and pop things and the top is something where you can pop things off if you want to. And so what's running is all these processors are running each on their own stack. They're all running the ordinary serial code. That's sort of the basic situation. They're pretty much running the serial code most of the time. So some processor runs. It pushes. Well, it doesn't spawn, so what does it do? It pushes something onto its stack because it's just a function call. And it does another couple more spawns so things pop off. Somebody returns so he pops his stack. So far everything's going on, they're not communicating, they're completely independent computations. This guy spawns and now he's out of work. Now he has to do something. What he does is he goes and picks another processor at random and he steals the thing from the other end of the stack. So he's unlikely to conflict because this guy's pushing and popping down here, but there's a lock

in there, thers's a little algorithm. A non-blocking algorithm actually, it's not lock. And so he goes and he steals something and come on, slide over there. Whoa. Yes, that's animation, right? That's the extent of my animation. And then he starts working away.

And the theorem is that a work-stealing scheduler like this gives expected running time with high probability actually of T sub 1 over P plus T sub infinity on P processors. And the pseudoproof is a little bit like the proof for Brent's Theorem, which is either you're working or stealing. If you're working well, that goes against T sub 1 over P. You can't do that very much or you run out of work. If you're stealing well, each steal has a chance that it steals the thing that's on the critical path. You may actually steal the wrong thing, but you actually have a 1 in P chance that you're the one who steals the thing that it's on the critical path and then in which case the expected number-- so you had this chance of 1 over P of reducing the critical path length by 1, so after this many steals the critical path is all gone. So you can only do P times T infinity steals. This high probability it comes out. And that gives you these bounds.

OK, I'm not going to give you all this stuff. Message passing sucks, you know. You guys know. There's probably nothing else in here. So basically the pitch here is that you get some high level linguistics support for these very fine-grained parallelism. It's an algorithmic programming model so that means that you can do engineering for performance. There's fairly easy conversion of existing code, especially when you combine it with the race detector. You've got this factorization of the debugging problem and to debugging your serial code is you run it with all the Cilk stuff turned off. You allied the program and make sure your program works. Then you run it with the rate detector to make sure you get the same answer in parallel and then you're done.

Applications in Cilk don't just scale to large number of processors, they scale down to small numbers, which is important if you only have two processors or one. You don't suddenly want to pay a factor of 10 to get off the ground, which happens sometimes on clusters running MPI. You have to pay a big overhead before you've made any progress. And one of the advantages for example is that the number of processors might change dynamically. In this model that's OK because it's not part of the program. So you may have the operating system reduce the number of actual worker threads that you have doing that work-stealing and that can work. One of the bad things about Cilk is that it doesn't support sort of data parallel or program model kind of parallelism. You really have to think of things as this divide and conquer kind of the world. And if you have trouble expressing that-- situations where you're doing Jacobi

update and you very carefully put things on, had each processor work on its local memory and then they only have to communicate at the boundaries. That's difficult to do right in Cilk because essentially every time you go around the loop of I have all these things to do. All the work-stealing happens randomly and it happens on a different processor. So it's not very good at that sort of thing, although it turns out Jacobi update's not a very good example for that because there are more sophisticated algorithms that use cache effectively that you can express in Cilk and I would have no idea how to no say those in some of these sort of data parallel languages. Using the cache efficiently is really important on modern processors.

PROFESSOR: Thank you. Questions?

BRADLEY KUSZMAUL: You can download Cilk, there's a bunch of contributors. Those are the Cilk worms and you can download Cilk off our webpage. Just Google for Cilk and you'll find it. It's a great language, you'll love it. You'll love it much more than what you've been doing.

AUDIENCE: How does the Cilk play with processor [OBSCURED]?

BRADLEY KUSZMAUL: Well, you have to have a language, a compiler that can generate those. If you have an assembly command or you have some other complier that can generate those. So I just won the HPC challenge, which is this challenge where everybody tries to run parallel programs and argue that they get productivity. For that there were some codes like matrix multiply and LUD composition with pivoting. Basically at the leads of the computation I call the Intel Math Kernel Library. Which in turn uses the SSE instructions. You could do anything you can do in C in the C parts of the code because Cilk compiler just passes those through. So if you have some really efficient pipeline code for doing something, up to some point it made sense to use that.

AUDIENCE: [OBSCURED]

BRADLEY KUSZMAUL: So I ran it on NASIS Columbia. So the benchmark consists of-- well, there's 7 applications they have. 6 of which are actually well-defined. One of them is this thing that just measures network performance or something, so it doesn't have any real semantics. There's 6 benchmarks. One of them is LUD composition, one of them is DJEM matrix multiplication and this FFT and 3 others. So I implemented all 6, nobody else implemented all 6. It turns out that you had to implement 3 in order to enter. Almost everybody implemented 3 or 4, but I did all 6 which is part of why I won. So I could argue that in a weeks work I just implemented--

AUDIENCE: What is [OBSCURED]?

BRADLEY KUSZMAUL: So the prize has two components. Performance and productivity or elegance or something and it's completely whatever the judges want that to be. So it was up to me as a presenter to make the case that I was elegant. Because I had my performance numbers, which were pretty good and it turned out that the IBM entry for x10 did me more good than I did, I think. Because they got up there and they compared the performance of x10 to their Cilk implementation and their x10 thing was almost as good as Cilk. So after that I think the judges said they had to give me the prize. So basically, it went down to supercomputing and each of us got 5 minutes to present and there were 5 finalists. We did our presentation and then they gave out the -- So they divided the prize three ways: the people who got the absolute best performance, which were some people running UPC and the people who had the most elegance based on minimal number of lines of codes and that was Cleve at -- what's his name? The Mathworks guy, MATLAB guy. Who said, look, matrix, LUD composition. LU of P. It's very elegant, but I don't think that it really sort of explains what you have to do to solve the problems. So he won the prize for most elegant and I got the prize for best combination, which they then changed-- in the final citation for the prize they said, most productivity. That was the prize. So I actually won the contest because that was what the contest was supposed to be was most productivity. But I only won 1/3 of the prize money because they divided it three ways.

PROFESSOR: Any other question? Thank you.

BRADLEY KUSZMAUL: Thank you.

PROFESSOR: We'll take a 5 minute break and since you had guest lecturer I do have [OBSCURED]