CHARLES LEISERSON: Today, we're going to talk about analyzing task parallel algorithms-- multi-threaded algorithms. And this is going to rely on the fact that everybody has taken an algorithms class. And so I want to remind you of some of the stuff you learned in your algorithms class. And if you don't remember this, then it's probably good to bone up on it a little bit, because it's going to be essential.

And that is the topic of divide-and-conquer recurrences. Everybody remember divide and conquer recurrences? These are-- and there's a general method for solving them that will deal with most of the ones we want, called the Master Method. And it deals with recurrences in the form T of n equals a times T of n over b plus f of n. And this is generally interpreted as I have a problem of size n. I can solve it by solving a problems of size n over b, and it costs me f of n work to do that division and accumulate whatever the results are of that to make my final result.

For all these recurrences, the unstated base case is that this is a running time. So T of n is constant if n is small. So does that makes sense? Everybody familiar with this? Right? Well we're going to review it anyway, because I don't like to go ahead and just assume, and then leave 20% of you or more, or less, left behind in the woods. So let's just remind ourselves of what this means.

So it's easy to understand this in terms of a recursion tree. I start out, and the idea is a recursion tree is to take the running time, here, and to reexpress it using the recurrence. So if I reexpress this and I've written it a little bit differently, then I have an f of n. I can put an f of n at the root, and have a copies of T of n over b. And that's exactly the same amount of work as I had-- or running time as I had in the T of n. I've just simply expressed it with the right hand side.

And then I do it again at every level. So I expand all the leaves. I only expanded one here because I ran out of space. And you keep doing that until you get down to T of 1. And so then the trick of looking at these recurrences is to add across the rows. So the first row adds up to f of n. The second row adds up to a times f of n over b. The third one is a squared f of n over b squared, and so forth.

And the height here, now. Since I'm taking n and dividing it by b each time, how many times can I divide by b until I get to something that's constant size? That's just log base b of n. So, so far, review-- any questions here? For anybody? OK. So I get the height, and then I look at how many-- if I've got T of 1 work at every leaf, how many leaves are there? And for this analysis we're going to assume everything works out-- n is a perfect power of b and so forth.

So if I go down k levels, how many sub problems are there at k levels? a of the k. So how many levels am I going down? h, which is log base b of n. So I end up with a to log base b of n times what's at the leaf, which is T of 1. And T of 1 is constant. a log base b of n-- that's the same as n to the log base b of a. OK. That's just a little bit of exponential algebra.

And you can-- one way to see that is, take the log of both sides of both equations, and you realize that all that's used there is the commutative law. Because if you take the log base-- if you take log of a log bn, you get log bn times log-- if you take a base b, log ba. And then you get the same thing if you take the log base b of what I have as the result. Then you get the exponent log base ba times log base b of n. So same thing, just in different orders. So that's just a little bit of math, because this is-- basically, we're interested in, what's the growth in n? So we prefer not to have log n's in the denominator. We prefer to have n's-- sorry, in the exponent we prefer to have n's.

So that's basically the number of things. And so then the question is, how much work is there if I add up all of these guys all the way down there? How much work is in all those levels? And it turns out there's a trick, and the trick is to compare n to log base b of a with f of n. And there are three cases that are very commonly arising, and for the most part, that's what we're going to see, is just these three cases.

So case 1 is the case where n to the log base b of a is much bigger than f of n. And by much bigger, I mean it's bigger by a polynomial amount. In other words, there's an epsilon such that the ratio between the two is at least n to the epsilon. There's an epsilon greater than 0. In other words, f of n is O of n to the log base b of a minus epsilon in the numerator there, which is the same as n log base b of a divided by n to the epsilon. In that case, this is geometrically increasing, and so all the weight-- the constant fraction of the weight-- is in the leaves. So then the answer is T of n is n to the log base b of a. So if n to log base b of a is bigger than f of n, the answer is n to the log base b of a, as long as it's bigger by a polynomial amount.

Now, case 2 is the situation where n to the log base b of a is approximately equal to f of n. They're very similar in growth. And specifically, we're going to look at the case where f of n is n to the log base b of a poly-logarithmic factor-- log to the k of n for some constant k greater than or equal to 0. That greater than or equal to 0 is very important. You can't do this for negative k. Even though negative k is defined and meaningful, this is not the answer when k is negative.

But if k is greater than or equal to 0, then it turns out that what's happening is it's growing arithmetically from beginning to end. And so when you solve it, what happens is, you essentially add an extra log term. So the answer is, if f of n is n to the log base b of a log to the k n, the answer is n to the log base b of a log to the k plus 1 of n. So you kick in one extra log. And basically, it's like-- on average, there's basically-- it's almost all equal, and there are log layers. That's not quite the math, but it's good intuition that they're almost all equal and there are log layers, so you tack on an extra log.

And then finally, case 3 is the case when no to the log base b is much less than f of n, and specifically where it is smaller by, once again, a polynomial factor-- by an n to the epsilon factor for epsilon greater than 0. It's also the case here that f has to satisfy what's called a regularity condition. And this is a condition that's satisfied by all the functions we're going to look at-- polynomials and polynomials times logarithms, and things of that nature. It's not satisfied for weird functions like sines and cosines and things like that. It's also not-- more relevantly, it's not satisfied if you have things like exponentials. So this is-- but for all the things we're going to look at, that's the case. And in that case, things are geometrically decreasing, and so all the work is at the root. And the root is basically cos f of n, so the solution is theta f of n.

We're going to hand out a cheat sheet. So if you could conscript some of the TAs to get that distributed as quickly as possible. OK. So let's do a little puzzle here. So here's the cheat sheet. That's basically what's on it. And we'll do a little in-class quiz, self-quiz. So we have T of n is 4T n over 2 plus n. And the solution is? This is a thing that, as a computer scientist, you just memorize this so that you can-- in any situation, you don't have to even look at the cheat sheet. You just know it. It's one of these basic things that all computer scientists should know. It's kind of like, what's 2 to the 15th? What is it?

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Yes. And interestingly, that's my office number. I'm in 32-G768. I'm the only one in this data center with a power of 2 office number. And that was totally unplanned. So if you need to remember my office number, 2 to the 15th.

OK, so what's the solution here?

AUDIENCE: Case 1.

CHARLES LEISERSON: It's case 1. And what's the solution?

AUDIENCE: n squared?

CHARLES LEISERSON: n squared. Very good. Yeah. So n to the log base b of a is n to the log base 2 of 4. Log base 2 of 4 is 2, so that's n squared. That's much bigger than n. So it's case 1, and the answer is theta n squared. Pretty easy. How about this one? Yeah.

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Yeah. It's n squared log n. Once again, the first part is the same. n to the log base b of a is n squared. n squared is n squared log to the 0 n. So it's case 2 with k equals 0, and so you just tack on an extra log factor. So it's n squared log n. And then, of course, we've got to do this one. Yeah.

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Yeah, n cubed, because once again, n to log base b of a is n squared. That's much less than n cubed. n cubed's bigger, so that dominates. So we have theta n squared. What about this one? Yeah.

AUDIENCE: Theta of n squared.

CHARLES LEISERSON: No. That's not the answer. Which case do you think it is?

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Case 2?

AUDIENCE: Yeah.

CHARLES LEISERSON: OK. No. Yeah.

AUDIENCE: None of the cases?

CHARLES LEISERSON: It's none of the cases. It's a trick question. Oh, I'm a nasty guy. I'm a nasty guy. This is one where the master method does not apply. This would be case 2, but k has to be greater than or equal to 0, and here k is minus 1. So case two doesn't apply. And case 1 doesn't apply, where we're comparing n squared to n squared over log n, because the ratio there is 1 over log n, and that-- sorry, the ratio there is log n, and log n is smaller than any n to the epsilon. And you need to have an n to the epsilon separation.

There's actually a more-- the actual answer is n squared log log n for that one, by the way, which you can prove by the substitution method. And it uses the same idea. You just do a little bit different math. There's a more general solution to this kind of recurrence called the Akra-Bazzi method. But for most of what we're going to see, it's sufficient to just-- applying the Akra-Bazzi method is more complicated than simply doing the table lookup of which is bigger and if sufficiently big, it's one or the other, or the common case where they're about the same within a log factor. So we're going to use the master method, but there are more general ways of solving these kinds of things.

OK. Let's talk about some multi-threaded algorithms. First thing I want to do is talk about loops, because loops are a great thing to analyze and understand because so many programs have loops. Probably 90% or more of the programs that are parallelized are parallelized by making parallel loops. The spawn and sync types of parallelism, the subroutine-type parallelism, is not done that frequently in code. Usually, it's loops.

So what we're going to look at is a code to do an in-place matrix transpose, as an example of this. So if you look at this code, I want to swap the lower side of the matrix with the upper side of the matrix, and here's some code to do it, where I parallelize the outer loop. So we're running the outer index from i equals 1 to n. I'm actually running the indexes from 0 to n minus 1. And then the inner loop goes from 0 up to i minus 1.

Now, I've seen people write transpose code-- this is one of these trick questions they give you in interviews, where they say, write the transpose of a matrix with nested loops. And what many people will do is, the inner loop, they'll run to n rather than running to i. And what happens if you run the inner loop to n? It's a very expensive identity function. And there's an easier, faster way to compute identity than with doubly nested loops where you swap everything and you swap them all back.

So it's important that the iteration space here, what's the shape of the iteration space? If you look at the i and j values and you map them out on a plane, what's the shape that you get? It's not a square, which it would be if they were both going from 1 to n, or 0 to n minus 1. What's the shape of this iteration space? Yeah, it's a triangle. It's basically-- we're going to run through all the things in this lower area. That's the idea. And we're going to swap it with the things in the upper one. But the iteration space runs through just the lower triangle-- or, correspondingly, it runs through the upper triangle, if you want to view it from that point of view. But it doesn't go through both triangles, because then you will get an identity.

So anyway, that's just a tip when you're interviewing. Double-check that they've got the loop indices to be what they ought to be. And here what we've done is, we've parallelized the outer loop, which means, how much work is on each iteration of this loop? How much time does it take to execute each iteration of loop? For a given value of i, what does it cost us to execute the loop? Yeah.

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Yes. Theta i, which means that-- if you think about this, if you've got a certain number of processors, you don't want to just chunk it up so that each processor gets an equal range of i to work on. You need something that's going to load balance. And this is where the Cilk technology is best, is when there are these unbalanced things, because it does the right thing, as we'll see.

So let's talk a little bit about how loops are actually implemented by the Open Cilk compiler and runtime system. So what happens is, we have this doubly-nested loop here, but the only one that we're interested in is the outer loop, basically. And what it does is, it creates this recursive program for the loop.

And what is this program doing? I'm highlighting, essentially, this part. This is basically the loop body here, which has been lifted into this recursive program. And what it's doing is, it is finding a midpoint and then recursively calling itself on the two sides until it gets down to, in this case, a one-element iteration. And then it executes the body of the loop, which in this case is itself a for loop, but not a parallel for loop. So it's doing divide and conquer. It's just basically tree splitting.

So basically, it's got this control on top of it. And if I take a look at what's going on in the control, it looks something like this. So this is using the DAG model that we saw before. And now what I have here highlighted is the lifted body of the loop-- sorry, of the control. And then down below in the purple, I have the lifted body. And what it's doing is basically saying, let me divide it into two parts, and then I spawn one recurrence and I call the other. And I just keep dividing like that till I get down to the base condition.

And then the work that I'm doing-- I've sort of illustrated here-- the work I'm doing in each iteration of the loop is growing from 1 to n. I'm showing it for 8, but in general, this is working from 1 to n for this particular one. Is that clear? So that's what's actually going on. So the Open Cilk runtime system does not have a loop primitive. It doesn't have loops. It only has, essentially, this ability to spawn and so forth. And so things, effectively, are translated into this divide and conquer, and that's the way that you need to think about loops when you're thinking in parallel. Make sense?

And so one of the questions is, that seems like a lot of code to write for a simple loop. What do we pay for that? How much did that cost us? So let's analyze this a little bit-- analyze parallel loops. So as you know, we analyze things in terms of work and span. So what is the work of this computation? Well, what's the work before you get there? What's the work of the original computation-- the doubly-nested loop? If you just think about it in terms of loops, if they were serial loops, how much work would be there? Doubly-nested loop. In a loop, n iterations. In your iteration, you're doing i work. Sum of i. i equals 1 to n. What do you get?

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Yes. Theta n squared. Doubly-nested group. And although you're not doing half the work, you are doing the other half of the work-- of the n squared work that you might think was there if you wrote the unfortunate identity function. So the question is, how much work is in this particular computation? Because now I've got this whole tree-spawning business going on in addition to the work that I'm doing in the leaves.

So the leaf work here, along the bottom here, that's all going to be order n squared work, because that's the same as in the serial loop case. How much does that other stuff up top [INAUDIBLE]? It looks huge. It's bigger than the other stuff, isn't it? How much is there? Basic computer science.

AUDIENCE: Theta of n?

CHARLES LEISERSON: Yeah. It's theta of n. Why is it theta if n in the upper part? Yep.

AUDIENCE: Because it's geometrically decreasing [INAUDIBLE]

CHARLES LEISERSON: Yeah. So going from the leaves to the root, every level is halving, so it's geometric. So it's the total number of leaves, because there's constant work in each of those phrases. So the total amount is theta n squared.

Another way of thinking about this is, you've got a complete binary tree that we've created with n leaves. How many internal nodes are there in a complete binary tree with n leaves? In this case, there's actually n over-- let's just say there's n leaves. Yeah. How many internal nodes are there? If I have n leaves, how many internal nodes to the tree-- that is, nodes that have children? There's exactly n minus 1. That's a property that's true of any full binary tree-- that is, any binary tree in which every non-leaf has two children. There's exactly n minus 1. So nice tree properties, nice computer science properties, right? We like computer science. That's why we're here-- not because we're going to make a lot of money.

OK. Let's look at the span of this. Hmm. What's the span of this calculation? Because that's how we understand parallelism, is by understanding work and span. I see some familiar hands. OK.

AUDIENCE: Theta n.

CHARLES LEISERSON: Theta n. Yeah. How did you get that?

AUDIENCE: The largest path would be the [INAUDIBLE] node is size theta n and [INAUDIBLE]

CHARLES LEISERSON: Yeah. So we're basically-- the longest path is basically going from here down, down, down to 8, and then back up. And so the eight is really n in the general case. That's really n in the general case. And so we basically are going down, And so the span of the loop control is log n. And that's the key takeaway here. The span of loop control is log n. When I do divide and conquer like that, if I had an infinite number of processors, I could get it all done in logarithmic time. But the 8 there is linear. That's order n. In this case, n is 8. So that's order n. So then it's log n plus order log n, which is therefore order n. So what's the parallelism here?

AUDIENCE: Theta n.

CHARLES LEISERSON: Theta n. It's the ratio of the two. The ratio of the two is theta n. Is that good?

AUDIENCE: Theta of n squared?

CHARLES LEISERSON: Well, parallelism of n squared, do you mean? Or-- is this good parallelism? Yeah, that's pretty good. That's pretty good, because typically, you're going to be working on systems that have maybe-- if you are working on a big, big system, you've got maybe 64 cores or 128 cores or something. That's pretty big. Whereas this is saying, if you're doing that, you better have a problem that's really big that you're running it on. And so typically, n is way bigger than the number of processors for a problem like this. Not always the case, but here it is.

Any questions about this? So we can use our work and span analysis to understand that, hey, the work overhead is a constant factor. And We're going to talk more about the overhead of work. But basically, from an asymptotic point of view, our work is n squared just like the original code, and we have a fair amount of parallelism. We have order n parallelism.

How about if we make the inner loop be parallel as well? So rather than just parallelize the outer loop, we're also going to parallelize the inner loop. So how much work do we have for this situation? Hint-- all work questions are trivial, or at least no harder than they were when you were doing ordinary serial algorithms. Maybe we can come up with a trick question on the exam where the work changes, but almost always, the work doesn't change. So what's the work? Yeah. n squared.

Parallelizing stuff doesn't change the work. What it hopefully does is reduce the span of the calculation. And by reducing the span, we get big parallelism. That's the idea. Now, sometimes it's the case when you parallelize stuff that you add work, and that's unfortunate, because it means that even if you end up taking your parallel program and running it on one processing core, you're not going to get any speedup. It's going to be a slowdown compared to the original algorithm. So we're actually interested generally in work-efficient parallel algorithms, which we'll talk more about later. So generally, we're after work efficiency. OK. What's the span of this?

AUDIENCE: Is it theta n still?

CHARLES LEISERSON: It is not still theta n. What was your thinking to say it was theta of n?

AUDIENCE: So the path would be similar to 8, and then--

CHARLES LEISERSON: But now notice that 8 is a for loop itself.

AUDIENCE: Yeah. I'm saying maybe you could extend the path another n so it would be 2n.

CHARLES LEISERSON: OK. Not quite, but-- so this man is commendable.

[APPLAUSE]

Absolutely. This is commendable, because this is-- this is why I try to have a bit of a Socratic method in here, where I'm asking questions as opposed to just sitting here lecturing and having it go over your heads. You have the opportunity to ask questions, and to have your particular misunderstandings or whatever corrected. That's how you learn. And so I'm really in favor of anybody who wants to come here and learn. That's my desire, and that's my job, is to teach people who want to learn. So I hope that this is a safe space for you folks to be willing to put yourself out there and not necessarily get stuff right.

I can't tell you how many times I've screwed up, and it's only by airing it and so forth and having somebody say, no, I don't think it's like that, Charles. This is like this. And I said, oh yeah, you're right. God, that was stupid. But the fact is that I no longer beat my head when I'm being stupid. Our natural state is stupidity. We have to work hard not to be stupid. Right? It's hard work not to be stupid. Yeah, question.

AUDIENCE: It's not really a question. My philosophy on talking in mid-lecture is that I don't want to waste other people's time.

CHARLES Yeah, but usually when-- my experience is-- and this is, let me tell you from-- I've been at MIT

LEISERSON: almost 38 years. My experience is that one person has a question, there's all these other people in the room who have the same question. And by you articulating it, you're actually helping them out. If I think you're going to slow, if things are going too slow, that we're wasting people's time, that's my job as the lecturer to make sure that doesn't happen. And I'll say, let's take this offline. We can talk after class. But I appreciate your point of view, because that's considerate. But actually, it's more consideration if you're willing to air what you think and have other people say, you know, I had that same question. Certainly there are going to be people in the class who, say, roll their eyes or whatever. But look, I don't teach to the top 10%. I try to teach to the top 90%. And believe me--

[LAUGHTER]

Believe me that I get farther with students and have more people enjoying the course and learning this stuff-- which is not necessarily easy stuff. After the fact, you're going to discover this is easy. But while you're learning it, it's not easy. This is what Steven Pinker calls the curse of knowledge. Once you know something, you have a really hard time putting yourself in the position of what it was like to not know it. And so it's very easy to learn something, and then when somebody doesn't understand, it's like, oh, whatever.

But the fact of the matter is that most of us-- it's that empathy. That's what makes for you to be a good communicator. And all of you I know are at some point going to have to do communication with other people who are not as technically sophisticated as you folks are. And so this is really good to sort of appreciate how important it is to recognize that this stuff isn't necessarily easy when you're learning it.

Later, you can learn it, and then it'll be easy. But that doesn't mean it's not so easy for somebody else. So those of you who think that some of these answers are like, come on, move along, move along, please be patient with the other people in the class. If they learn better, they're going to be better teammates on projects and so forth. And we'll all learn. Nobody's in competition with anybody here, for grades or anything. Nobody's in competition. We all set it up so we're going against benchmarks and so forth. You're not in competition. So we want to make this something where everybody helps everybody learn. I probably spent too much time on that, but in some sense, not nearly enough.

OK. So the span is not order n. We got that much. Who else would like to hazard to-- OK.

AUDIENCE: Is it log n?

CHARLES LEISERSON: It is log n. What's your reasoning?

AUDIENCE: It's the normal log n from the time before, but since we're expanding the n--

CHARLES LEISERSON: Yup.

AUDIENCE: --again into another tree, it's log n plus log n.

CHARLES LEISERSON: Log n plus log n. Good.

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: And then what about the leaves?

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: What's-- you got to add in the span of the leaves. That was just the span of the control.

AUDIENCE: The leaves are just 1.

CHARLES LEISERSON: The leaves are just 1. Boom. So the span of the outer loop is order log n. The inner loop is order log n. And the span of the body is order 1, because we're going down to the body, now it's just doing one iteration of serial execution. It's not doing i iterations. It's only doing one iteration. And so I add all that together, and I get log n. Does that makes sense?

So the parallelism is? This one, I should-- every hand in the room should be up, waving to call on me, call on me. Sure.

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Yeah. n squared over log n. That's the ratio of the two. Good. Any questions about that? OK. So the parallelism is n squared over log n, and this is more parallel than the previous one. But it turns out-- you've got to remember, even though it's more parallel, is it a better algorithm in practice? Not necessarily, because parallelism is like a thresholding thing. What you need is enough parallelism beyond the number of processors that you have-- the parallel slackness, remember? So you have to have the number-- the amount of parallelism, if it's much greater than the number of processors, you're good.

So for something like this, if with order n parallelism you're way bigger than the number of processors, you don't need to parallelize the inner loop. You don't need to parallelize the inner loop. You'll be fine. And in fact, we're talking a little bit about overheads, and I'm going to do that with an example from using vector addition.

So here's a really simple piece of code. It's a vector-- add two vectors together, two arrays. And all it does is, it adds b into a. You can see every position as b into a. And I'm going to parallelize this by putting a Cilk for in front, rather than an ordinary for. And what that does is, it gives us this divide and conquer tree once again, with n leaves. And the work here is order n, because that's-- we've got n iterations of constant time. And the span is just the control-- log n. And so the parallelism is n over log n. So this is basically easier than what we just did.

So now-- if I look at this, though, the work here includes some substantial overhead, because there are all these function calls. It may be order n, and that's good enough if you're certain kinds of theoreticians. This kind of theoretician, that's not good enough. I want to understand where these overheads are going.

So the first thing that I might do to get rid of that overhead-- so in this case, what I'm saying is that as I do the divide and conquer, if I go all the way down to n equals 1, what am I doing in a leaf? How much work is in one of these leaves here? It's an add. It's two memory fetches and a memory store and an add. The memory operations are going to be the most expensive thing there. That's all that's going on. And yet, right before then, I've done a subroutine call-- a parallel subroutine call, mind you-- and that's going to have substantial overhead. And so the question is, do you do a subroutine call to add two numbers together? That's pretty expensive.

So let's take a look at how we can optimize away some of this overhead. And so this gets more into the realm of engineering. So the Open Cilk system has a pragma. Pragma is a compiler directive-- suggestion to the compiler-- where it can suggest, in this case, that there be a grain size up here of G, for whatever you set G to. And the grain size is essentially-- we're going to use-- and it shows up here in the code-- as instead of ending up-- it used to be high greater than low plus 1, so that you ended with a single element. Now it's going to be plus G, so that at the leaves, I'm going to have up to G elements per chunk that I do when I'm doing my divide and conquer. So therefore, I can take my subroutine overhead and amortize it across G iterations rather than amortizing across one iteration. So that's coarsening.

Now, if the grain size pragma is not specified, the Cilk runtime system makes its best guess to minimize the overhead. So what it actually does at runtime is, it figures out for the loop how many cores it's running on, and makes a good guess as to the actual-- how much to run serially at the leaves and how much to do in parallel. Does that make sense? So it's basically trying to overcome that.

So let's analyze this a little bit. Let's let i be the time for one iteration of the loop body. So this is i for iteration. This is of this particular loop body-- so basically, the cost of those three memory operations plus an add. And G is the grain size. And now let's take a look-- add another variable here, which is the time to perform a spawn and return. I'm going to call a spawn and return. It's basically the overhead for spawning.

So if I look at the work in this context, I can view it as I've got T1 work, which is n here times the number of iterations-- because I've got one, two, three, up to n iterations. And then I have-- and those are just the normal iterations. And then, since I have n over G minus 1-- there's n over G leaves here of size G. So I have n over G minus 1 internal nodes, which are my subroutine overhead. That's S. So the total work is n times i plus n over G minus 1 times S.

Now, in the original code, effectively, the work is what? If I had the code without the Cilk for loop, how much work is there before I put in all this parallel control stuff? What would the work be? Yeah.

AUDIENCE: n i?

CHARLES LEISERSON: n times i. We're just doing n iterations. Yeah, there's a little bit of loop control, but that loop control is really cheap. And on a modern out-of-order processor, the cost of incrementing a variable and testing against its bound is dwarfed by the stuff going on inside the loop. So it's ni. So this part here-- oops, what did I do? Oops, I went back. I see. So this part here-- this part here, there we go-- is all overhead. This is what it costs-- this part here is what cost me originally.

So let's take a look at the span of this. So the span is going to be, well, if I add up what's at the leaves, that's just G times i. And now I've got the overhead here for any of these paths, which is basically proportional-- I'm ignoring constants here to make it easier-- log of n over G times

S, because it's going log levels. And I've got n over G chunks, because each-- I've got G things at the iterations of each leaf, so therefore the number of leaves n over G. And I've got n minus 1 of those-- sorry, got log n of those-- actually, 2 log n. 2 log n over G of those times S. Actually, maybe I don't. Maybe I just have log n, because I'm going to count it going down and going up. So actually, constant of 1 is fine.

Who's confused? OK. Let's ask some questions. You have a question? I know you're confused. Believe me, I spend-- one of my great successes in life was discovering that, oh, confusion is how I usually am. And then it's getting confused that is-- that's the thing, because I see so many people going through life thinking they're not confused, but you know what, they're confused. And that's a worse state of affairs to be in than knowing that you're confused. Let's ask some questions. People who are confused, let's ask some questions, because I want to make sure that everybody gets this. And for those who you think know it already, sometimes it helps them to know it a little bit even better when we go through a discussion like this. So somebody ask me a question. Yes.

AUDIENCE: Could you explain the second half of that [INAUDIBLE]

CHARLES LEISERSON: Yeah. OK. The second half of the work part. OK. So the second half of the work part, n over G minus 1. So the first thing is, if I've got G iterations at the leaves of a binary tree, how many leaves do I have if I've got a total of n iterations?

AUDIENCE: Is it n over G?

CHARLES LEISERSON: n over G. That's the first thing. The second thing is a fact about binary trees-- of any full binary tree, but in particular complete binary trees. How many internal nodes are there in a complete binary tree? If n is the number of leaves, it's n minus 1. Here, the number of leaves is n over G, so it's n over G minus 1. That clear up something for some people? OK, good. So that's where that-- and now each of those, I've got to do those three colorful operations, which is what I'm calling S. So you got the work down? OK. Who has a question about span? Span's my favorite. Work is good right. Work is more important, actually, in most contexts. But span is so cool. Yeah.

AUDIENCE: What did you mean when you said [INAUDIBLE]

CHARLES LEISERSON: So what I was saying-- well, I think what I was saying-- I think I was mis-saying something, probably, there. But the point is that the span is basically starting at the top here, and taking any path down to a leaf and then going back up. And so if I look at that, that's going to be then log of the number of leaves. Well, the number of leaves, as we agreed, was n over G. And then each of those is, at most, S to do the subroutine calling and so forth that's bookkeeping that's in that node. That make sense? Still I didn't answer the question? Or--

AUDIENCE: Why is that the span? Why shouldn't it be [INAUDIBLE]

CHARLES LEISERSON: It could be any of the paths. But take a look at all the paths, go down, and back up. There's no path that's going down and around and up and so forth. This is a DAG. So if you just look at the directions of the arrows. You got to follow the directions of the arrows. You can't go down and up. You're either going down, or you've started back up. So it's always going to be, essentially, down through a set of subroutines and back up through a set of subroutines. Does that make sense?

And if you think about the code, the recursive code, what's happening when you do divide and conquer? If you were operating with a stack, how many things would get stacked up and then unstacked? So the path down and back up would also be logarithmic at most. Does that makes sense? So I don't have a-- if I had one subtree here, for example, dependent on-- oops, that's not the mode I want to be in-- so one subtree here dependent on another subtree, then indeed, the span would grow. But the whole point is not to have these two things-- to make these two things independent, so I can run them at the same time. So there's no dependency there. We good? OK.

So here I have the work and the span. I have two things I want out of this. Number one, I want the work to be small. I want work to be close to the work of the n times i, the work of the ordinary serial algorithm. And I want the span to be small, so it's as parallel as possible. Those things are working in opposite directions, because if you look, the dominant term for G in the first equation is dividing n. So if I want the work to be small, I want G to be what? Big. The dominant term for G in the span is the G multiplied by the i. There is another term there, but that's a lower-order term.

So if I want the span to be small, I want G to be small. They're going in opposite directions. So what we're interested in is picking a-- finding a medium place. We want G to be-- and in particular, if you look at this, what I want is G to be at least S over i. Why? If I make G be much bigger than S over i-- so if G is bigger than S over i-- then this term multiplied by S ends up being much less than this term. You see that? That's algebra.

So do you see that if I make G be-- if G is much less than S over i-- so get rid of the minus 1. That doesn't matter. So that's really n times S over G, so therefore S over G, that's basically much smaller than i. So I end up with something where the result is much smaller than n i. Does that make sense?

OK. How we doing on time? OK. I'm going to get through everything that I expect to get through, despite my rant. OK. Does that make sense? We want G to be much greater than S over i. Then the overhead is going to be small, because I'm going to do a whole bunch of iterations that are going to make it so that that function call was just like, eh, who cares? That's the idea.

So that's the goal. So let's take a look at-- let's see, what was the-- let me just see here. Did I-- somehow I feel like I have something out of order here, because now I have the other implementation. Huh. OK. I think-- maybe that is where I left it. OK. I think we come back to this. Let me see. I'm going to lecture on.

So here's another implementation of the for loop to add two vectors. And what this is going to use as a subroutine, I have this operator called v add, which itself does just a serial vector add. And now what I'm going to do is run through the loop here and spawn off additions-- and the min there is just for a boundary condition. I'm going to spin off things in groups of G. So I spin off, do a vector add of size G, go on vector add of size G, vector add of size G, jumping G each time.

So let's take a look at the analysis of that. So now what I've got is, I've got G iterations, each of which costs me i. And this is the DAG structure I've got, because the for loop here that has the Cilk spawn in it is going along, and notice that the Cilk spawn is in a loop. And so it's basically going-- it's spawning off G iterations. So it's spawning off the vector add, which is going to do G iterations-- because I'm giving basically G, because the boundary case let's not worry about. And then spawn off G, spawn off G, spawn off G, and so forth.

So what's the work of this? Let's see. Well, let's make things easy to begin with. Let's assume G is 1 and analyze it. And this is a common thing, by the way, is you as assume that grain size is 1 and analyze it, and then as a practical matter, coarsen it to make it more efficient. So if G is 1, what's the work of this? Yeah.

AUDIENCE: [INAUDIBLE]

Yeah. It was order n, because those other two things are constant. So exactly right. It's order n. In fact, this is a technique, by the way, that's called strip mining, if you take away the parallel thing, where you take a loop of length n. And you really have nested loops here-- one that has n over G iterations and one that has G iterations-- and you're going through exactly the same stuff. And that's the same as going through n iterations. But you're replacing a singly-nested loop by a doubly-nested loop. And the only difference here is that in the inner loop, I'm actually spawning off work. So here, the work is order n, because I basically-- if I'm spinning off just-- if G is 1, then I'm spinning off one piece of work, and I'm going to n minus 1, spinning off one. So I've got order n work up here, and order n work down below. What's the span for this. After all, I got in spans there now-- sorry, n spawns, not n spans. n spawns. What's the span going to be? Yeah.

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Sorry? Sorry I, couldn't hear--

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Theta S? No, it's bigger than that. Yeah, you'd think, gee, I just have to do one thing to go down and up. But the span is the longest path in the whole DAG. It's the longest path in the whole DAG. Where's the longest path in the whole DAG start? Upper left, right? And where does it end? Upper right. How long is that path? What's the longest one? It's going to go all the way down the backbone of the top there, and then flip down and back up. So how many things are in the-- if G is 1, how many things are my spawning off there? n things, so the span is? Order n? So order n. It's long. So what's the parallelism here?

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Yeah. It's order 1. And what do we call that?

AUDIENCE: Bad.

CHARLES LEISERSON: Bad. Right. But there's a more technical name. They call that puny.

[LAUGHTER]

It's like, we went through all this work, spawned off all that stuff, added all this overhead, and it didn't go any faster. I can't tell you how many times I've seen people do this when they start parallel programming. Oh, but I spawned off all this stuff! Yeah, but you didn't reduce the span. Let's now-- that was the analyze it in terms of n-- sorry, in terms of G equals 1. Now let's increase the grain size and analyze it in terms of G. So once again, what's the work now? Work is always a gimme. Yeah.

AUDIENCE: Same as before, n.

CHARLES LEISERSON: n. Same as before. n. The work doesn't change when you parallelize things differently and stuff like that. I'm doing order n iterations. Oh, but what's the span? This is a tricky one. Yeah.

AUDIENCE: n over G.

CHARLES LEISERSON: Close. That's half right. That's half right. Good. That's half right. Yeah.

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: n over G plus G. Don't forget that other term. So the path that we care about goes along the top here, and then goes down there. And this has span G. So we've got n over G here, because I'm doing chunks of G, plus G. So it's G plus n over G. And now, how can I choose G to minimize the span? There's nothing to choose to minimize the work, except there's some work overhead that we're trying to do. But how can I choose G to minimize the span? What's the best value for G here? Yeah.

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: You got it. Square root of n. So one of these is increasing. If G is increasing, n over G is decreasing, where do they cross? When they're equal. That's when G equals n over G, or G is square root of n. So this actually has decent-- n big enough, square root of n, that's not bad. So it is OK to spawn things off in chunks. Just don't make the chunks real little. What's the parallelism? Once again, this is always a gimme. It's the ratio. So square root of n.

Quiz on parallel loops. I'm going to let you folks do this offline. Here's the answers. If you quickly write it down, you don't have to think about it.

[RAPID BREATHING]

OK.

[LAUGHTER]

OK. So take a look at the notes afterwards, and you can try to figure out why those things are so. So there's some performance tips that make sense when you're programming with loops. One is, minimize the span to maximize the parallelism, because the span's in the denominator. And generally, you want to generate 10 times more parallelism than processors if you want near-perfect linear speed-up. So if you have a lot more parallelism than the number of processors-- we talked about that last time-- you get good speed-up. If you have plenty of parallelism, try to trade some of it off to reduce the work overhead.

So the idea was, for any of these things, you can fiddle with the numbers, the grain size in particular, to reduce-- it reduces the parallelism, but it also reduces the overhead. And as long as you've got sufficient parallelism, your code is going to run just fine parallel. It's only when you're in the place where, ooh, don't have enough parallelism, and I don't want to pay the overhead. Those are the sticky ones. But most of the time, you're going to be in the case where you've got way more parallelism than you need, and the question is, how can you reduce some of it in order to reduced the work overhead?

Generally, you should use divide and conquer recursion or parallel loops, rather than spawning one small thing after another. So it's better to do the Cilk for, which already is doing divide and conquer parallelism, than doing the spawn off one thing at a time type of strategy, unless you can chunk them so that you have relatively few things that you're spawning off. This would be fine. The thing I say not-- this would be fine if foo of i was really expensive. Fine, then we'll have lots of parallelism, because there's a lot of work there. But generally, it's better to do the divide and conquer.

Generally, you should make sure that the work that you're doing per number of spawns is sufficiently large. So the spawns say, well, how much are you busting your work into in terms of chunks? Because the spawn has an overhead, and so the question is, well, how big is that? And so you can coarsen by using function calls and in-lining near the leaves.

Generally better to parallelize outer loops as opposed to inner loops, if you're forced to make a choice, because the inner loops, they're the overhead you're incurring every single time. The outer loop, you can amortize it against the work that's going on inside that doesn't have the overhead.

And watch out for scheduling overhead. So here's an example of two codes that have parallelism 2, and one of them is an efficient code, and the other one is lousy code. The top one is efficient, because I have two iterations that I run in parallel, and each of them does a lot of work. There's only one scheduling operation that happens. The bottom one, I have n iterations, and each iteration does work, too, so I basically have n iterations with overhead. And so if you just look at these, look at the overhead, you can see what the difference is.

OK. I want to talk a little bit about actually, I have a whole bunch of things here that I'm not going to get to, but I didn't expect to get to them. But I do want to get to some of matrix multiplication. People familiar with this problem? OK. We're going to assume for simplicity that n is a power of 2. So here's the typical way of parallelizing matrix multiplication. I take the two outer loops and I parallelize them. I can't easily parallelize the inner loop, because if I do, I get a race condition, because I'll have two iterations that are both trying to update C of i, j. So I can't just parallelize k, so I'm just going to parallelize i and j.

The work for this is what? Triply-nested loop. n cubed. Everybody knows-- matrix multiplication, unless you do something clever like Strassen, or one of the more recent-- Virgie Williams algorithm-- you know that the running time for the standard algorithm is n cubed. The span for this is what? Yeah. That inner loop is linear size, and then you've got two log n's-- log n plus log n plus n-- so it's order n. So the parallelism is around n squared. If I ignore constants, and I said I was working on matrices of, say, 1,000 by 1,000 or so, the parallelism is something like n squared, which is about-- 1,000 squared is a million. Wow. That's a lot of parallelism. How many processors are you running on? Is it bigger than 10 times the number of processors? By a little bit.

Now, there's another strategy that one can use, which is divide and conquer, and this is the strategy that's used in Strassen. We're not going to do the Strassen's algorithm. We're just going to use the eight multiply version of this. For people who know Strassen, more power to you. It's a great algorithm. Really surprising, really amazing. And it's actually worthwhile doing in practice, by the way, for sufficiently large matrices. So the idea here is, I can multiply two n by n matrices by doing eight multiplications of n over 2 by n over 2 matrices, and then add two n by n matrices.

So when we start talking matrices-- this is a little bit of a diversion from the algorithms, but it's so important, because representation of matrices is one of the things that gets people into trouble when they're doing any kind of two-dimensional coding stuff. And so I want to talk a little bit about index, and we're going to talk about this more later when we do cache behavior and such.

So how do you represent sub-matrices? The standard way of representing those either in row-major or column-major order, depending upon the language you use. Fortran uses column-major ordering, so there are a lot of subroutines that are column-major. But for the most part, C, which we're using, is row-major.

And so the question is, if I take a sub-matrix of a large matrix, how do I calculate where the i, j element of that matrix is? Here I have the i, j element here. I've got a matrix M, which is embedded. And by row major, remember, that means I just take row after row, and I just put them in linear order through the memory. So every two-dimensional matrix, you can index as a one-dimensional matrix, because all you have to do is-- which is exactly what the code is doing-- you need to know the beginning of the matrix. But if you have a sub-matrix, it's a little more complicated.

So here's the idea. Suppose that you have a sub-matrix m-- so starting in location m of this outer matrix. Here we have the outer matrix, which has length n sub M. This is the big matrix-- actually I should have called that m. I should not have called this n instead of m. I should have called it m sub something else, because this is my m that I'm interested in, which is this location here.

And what I'm interested in doing is finding out-- I named these variables stupidly-- is finding out, where is the i, j-th element of this sub-matrix M? If I tell you the beginning, what do I add to get to i, j? And the answer is that I've got to add the number of rows that comes down here. Well, that's i times the width of the full matrix that you're taking it out of, not the width of your local sub-matrix. And then you have to add in the-- and then you add in j from that point. There we go. OK.

So I have to add in the length of the long matrix plus j for each row i. Does that make sense? Because it's embedded in there. You have to skip over full rows of the outer matrix. So you can't generally just pass a sub-matrix and expect to do indexing on that when it's embedded in a large matrix. If you make a copy, sure, then you can index it according to whatever the new copy is. But if you want to operate in place on matrices, which is often the case, then you have to understand that every row, you have to jump a row of the outer matrix, not a row of whatever your sub-matrix is, when you're doing the divide and conquer.

So when we look at doing divide and conquer-- I have a matrix here which I want to now divide into four sub-matrices of size M over 2. And the question is, where's the starting corners of each of those matrices? So M 0, 0, that starts at the same place as M. That upper left one. Where does M 0, 1 start? Where's M 0, 1 start?

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Yeah. M plus n over 2. Where does M 1, 0 start? This is the tricky one. Here's the answer. M plus the long matrix times n over 2, because I'm going down m over 2 rows, and I've got to go down the number of rows of the outer matrix. And then M 1, 1 is the same as the 2 there. So here's the-- in general, for row and column being 0 and 1, in some sense, this is a general formula that matches up with that, where I plug in 0 1 for each one.

And now here's my code. And I just want to point out a couple of things, and then we'll quit and I'll let you take a look at the rest of this on your own. Here's my divide and conquer matrix multiply. I use restrict. Everybody familiar with restrict? It says, don't tell the compiler these things you can assume are not aliased, so that when you change one, you're not changing another. That lets the compiler produce better code. And then the row sizes are going to be n sub c, n sub a, and n sub b. And then the matrices that we're taking them out of, those are the sizes of the sub-matrix. The outer matrix is going to have size n by n, for which-- when I have my recursion, I want to talk about sub-matrices that are embedded in this larger outside matrix.

Here is a great piece of bit tricks. This says, n is a power of 2. So go back and remind yourself of what the bit tricks are, but that's a clever bit trick to say that n is a power of 2. Very quick. And so take a look at that. And then we're going to coarsen leaves with a base case. The base case just goes through and solves the problem for small n, just with a typical triply-nested loop. And what we're going to do is allocate a temporary n by n array, and then we're going to define the temporary array to having underlying row size n.

And then here is this fabulous macro that makes all the index calculations easy. It uses the sharp sharp operator, which pastes together tokens, so that I can paste n sub c. When I pass r and c, it passes-- whatever value I pass for that, it pastes it together. So it allows me to do the indexing of the-- and have the right thing, so that for each of these address calculations, I'm able to do them by just saying x of, and just give the formulas these. Otherwise, you'd be driven nuts by the formula. So take a look at that macro, because that may help you in some of your other things. And then I sync, and then add it up. And the addition is just going to be a doubly-nested parallel addition, and then I free it.

So what I would like you to do is go home and take a look at the analysis of this. And it turns out this has way more panels than you need, and if you reduce the amount of parallelism, you get much better performance. And there's several other algorithms I put in there as well. so I'll try to get this posted tonight. Thanks very much.