CHARLES LEISERSON: So today, more parallel programming, as we will do for the next couple lectures as well. So today, we're going to look at how to analyze multi-threaded algorithms, and I'm going to start out with a review of what I hope most of you know from 6006 or 6046, which is how to solve divide and conquer recurrences. Now, we know that we can solve them with recursion trees, and that gets tedious after a while, so I want to go through the so-called Master Method to begin with, and then we'll get into the content of the course. But it will be very helpful, since we're going to do so many divide and conquer recurrences.

The difference between these divide and conquer recurrences and the ones for caching is that caching is all tricky by the base condition. Here, are all the recurrences are going to be nice and clean, just like you learn in your algorithms class. So we'll start with talking about it, and then we'll go through several examples of analysis of algorithms. And it'll also tell us something about what we need to do to make our code go fast.

So the main method we're going to use is called the Master Method. It's for solving recurrences of the form t of n equals a t of n over b plus f of n, where we have some technical conditions, a is greater than or equal to 1, b is greater than one, and f is asymptotically positive. As f gets large, it becomes positive. When we give a recurrence like this, normally if the base case is order one, it's convention not give it, to just assume, yeah, we understand that when n is small enough, the result is constant. As I say, that's the place where this differs from the way that we solve recurrences for caching, where you have to worry about what is the base case of the recurrence.

So the way to solve this is, in fact, the way we've seen before. It's a recursion tree.

So we start with t of n, and then we replace t of n by the right hand side, just by substitution. So what's always going to be in the tree as we develop it is the total amount of work. So we basically replace it by f of n plus a copies of t of n over b. And then each of those we replace by a copies so t of n over b squared. And so forth, continually replacing until we get down to t of 1. And at the point t of 1, we no longer can substitute here, but we know that t of 1 is order 1.

And now what we do is add across the rows. So we get f of n, we get a, f of n over b, a squared f of n over b squared, and we keep going to the height of this, which we're dividing by the argument by b each time. So to get down to 1 is just log base b of n. So the number of leaves is, since this is a regular [? a area ?] tree, is a to the height, which is a to the log base b of n. And for each of those, we're paying t of 1, which is order 1.

And so now, it turns out there's no closed form solution. If I add up all these terms, there's no closed form solution. But there are three common situations that occur in practice. So yes, this is just n to the log base b of a, just this term, not the sum, just this term. So three cases have to do with comparing the number of leaves, which is times order 1, with f of n. So the first case is the case where n the log base b of a is bigger than f of n. So whenever you're given a recurrence to compute n to the log base b of a-- I hope this is repeat for most people. If not, that's fine, but hopefully it'll get you caught up. n log base b of a, if it's much bigger than f of n, then these terms are geometrically increasing.

And since it's geometrically increasing, all that matters is the base case. In fact, actually, it has to be not just greater than, it's got to be greater than by a polynomial amount, by some n to the epsilon amount for some epsilon greater than 0. So it might be n to the 1/2, it might be n to the 1/3, it could be n to the 100th. But what it can't be is log n, because log n is less than any polynomial amount. So it's got to exceed it by at least n to the epsilon for some epsilon. In that case, it's geometrically increasing, the answer is just what's at the leaves. So that's case one, geometrically increasing.

Case two is when things are actually fairly equal on every level. And the general case we'll look at is when it's arithmetically increasing. So in particular, this occurs when f of n is n to the log base b of a times log to some power, n, for some constant k that's at least 0. So this is the situation. If k is equal to 0, it just says that f of n, the amount here is exactly the same as the number of leaves. And that case, it turns out that every level has almost exactly the same amount. And since there are log n levels, you tack on an extra log n for the solution. In fact, the solution is one more log.

Turns out that if it's growing arithmetically with layer, you basically take on one extra log. It's actually like doing the integral of a as an arithmetic series. If you're adding the terms of, say, i squared, the result is i cubed. If you have a summation that goes from i equals 1 to n of i squared, the result is proportional to n cubed. And similarly, if it's i to any k, the result is going to be n to the k plus 1, and that's basically what's going on here.

And then the third case is when it's geometrically decreasing, when the amount at the root dominates. So in this case, if it's much less than n, and specifically, f of n is at least n to the epsilon less than log base b of a, for some constant epsilon, it turns out in addition, you need f of n to satisfy a regularity condition, but this regularity condition is satisfied by all the normal functions that we're going to come up. It's not satisfied by things like n to the sine n, which oscillates like crazy. But it also isn't satisfied by exponentially growing functions. But it is satisfied by anything that's polynomial, or polynomial times a logarithm, or what have you. So generally, we don't really have to check this too carefully.

And then the answer there is just it's order f of n, because it's geometrically decreasing, f of n dominates. So is this review for everybody? Pretty much, yeah, yeah, yeah? You can do this in your head, because we're going to ask you to do this in your head during the lecture? Yeah, we're all good? OK, good.

One of the things that students, when they learn this in an algorithms class, don't recognize is that this also tells you where in your program, where in your recursive program, you should bother to try to eke out constant factors. So if you think about it, for example, in case three here, it's geometrically decreasing. Does it make sense to try to optimize the leaves? No, because very little time is spent there. It makes sense to optimize what's going on at the root, and to save anything you can at the root. And sometimes, the root in particular has special properties that aren't true of the internal nodes that you can take advantage of, that you may not be able to take advantage of regularly. But since it's going to be dominated by the root, trying to save in the root makes sense.

Correspondingly, if we're in case one, in case one, it's absolutely critical that you coarsen the recursion, because all the work is down at this level. And so if you want to get additional performance, you want to basically move this up high enough that you can cut off that constant amount and get factors two, three, sometimes more, out of your code. So understanding the structure of the recursion allows you to figure out where it is that you should optimize your code. Of course, with loops, it's much easier. Where do you spend your time with loops to make code go fast? The innermost loop, right, because that's the one that's executing the most. The outer loops are not that important. This is sort of the corresponding thing for recursion. Figure out where the recursion is spending the time, that's where you spend time eking out extra factors.

Here's the cheat sheet. So if it's n to the log base b of a minus epsilon, the answer's n to the log base b of a. If it's plus epsilon, it's f of n. And if it's n to the log base b of a times a logarithmic factor, where this logarithm has the exponent greater than or equal to 0, you add a 1. This is not all of the situations. There are situations which don't occur.

OK, quick quiz. So t of n equals 4t of n over 2 plus n. What's the solution? n squared, good. So this is n to the log base b of a, is n to the log base 2 of 4, which is n squared. That's much bigger than n. It's bigger by a factor of n. Here, the epsilon of 1 would do, so would an epsilon of 1/2 or an epsilon of 1/4, but in particular, an epsilon of 1 would do. That's case one. The n squared dominates, so the answer is n squared. The basic idea is whichever side dominates for case one and case three, that's the one that is the answer.

Here we go. What about this one? n squared log n, because the two sides are about the same size. It's n squared times log to the 0n, tack on the extra log. How about this one? n cubed. How about this one?

AUDIENCE: [INAUDIBLE]. Master Theorem [INAUDIBLE]?

CHARLES LEISERSON: Yeah, the Master Theorem does not apply to this one. It looks like it's case two with an an exponent of minus 1, but that's bogus because the exponent of the log must be greater than or equal to 0. So instead, this actually has a solution, so it does not apply, it actually has the solution n squared, log log n, but that's not covered by the Master Theorem. You can have an infinite hierarchy of narrowing in things.

So if you don't have a solution to something that looks like a Master Theorem type of recurrence, what's your best strategy for solving it?

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: What's that?

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: So a recursion tree can be good, but actually, the best is the substitution method, basically proving it by induction. And the recursion tree can be very helpful in giving you a good guess for what you think the answer is. But the most reliable way to prove any of these things is using substitution method. Good enough. So that was review for, I hope, most people? Yeah? OK, good.

OK, let's talk about parallel programming. We're going to start out with loops. So last time, we talked about how the cilk++ runtime system is based on, essentially, implementing spawns and syncs, using the work stealing algorithm, and we talked about scheduling and so forth. We didn't talk about how loops are implemented, except to mention that they were implemented with divide and conquer. So here I want to go into the subtleties of loops, because probably most programs that occur in the real world these days are programs where people just simply say, make this a parallel loop. That's it.

So let's take an example of the in-place matrix transpose, where we're basically trying to flip everything along the main diagonal. I've used this figure before, I think. So let's just do it not cache efficiently. So the cache efficient algorithm actually parallelizes beautifully also, but let's not look at the cache efficient version, the divide and conquer version. Let's look at a looping version to understand. And once again here, as I did before, I'm going to make the indices for my implementation run from 0, not 1. And basically, I have an outer loop that goes from i equals 1 up to n minus 1, and an inner loop that goes from j equals 0 up to i minus 1. And then I do a little swap in there. And in here, the outer loop I've parallelized, the inner loop is running serially.

So let's take a look at analyzing this particular piece of code to understand what's going on. So the way this actually gets implemented is as follows. So here's the code on the left. What actually happens in the cilk++ compiler is it converts it into recursion, divide and conquer recursion. So what it does is it has a range from low to high, is sort of the common case. We're going to call it on from 1 to n minus 1, because that's the indexes that I've given to the cilk_for loop. And what we do is, if I have a region to do divide and conquer on, a set of values for i, I basically divide that in half. And then I recursively execute the first half, and then the second half, and then cilk_sync. So the two sides are going off in parallel.

And then if I am at the base, then I go through the inner loop and do the swap of the values in the inner loop. So the outer loop is the one that's the parallel loop. That one we're doing divide and conquer on. So we basically recursively spawn the first half, execute the second, and then each of those recursively does the same thing until all the iterations have been done. Any questions about how that operates? So this is the way all the parallel loops are done, is basically this strategy.

Now here, I mention that this is in fact what happens is this test here is actually coarsened. So we don't want to go all the way to n equals 1, because then we'll have this recursion call overhead every time we do a call. So in fact, what happens is you go down to some grain size of some number of iterations, and at that number of iterations, it then just runs through it as an ordinary serial four loop, in order not to pay the function call overhead all the way down. We're going to look exactly at that issue.

So if I take a look at it from using the DAG model that I introduced last time, remember that the rectangles here kind of count as activation frames, stack frames on the call stack. And the circles here are strands, which are sequences of serial code. And so what's happening here is essentially, I'm running the code that divides it into two parts, and I spawn one part. Then this guy spawns the other and waits for the return, and then these guys come back. And then I keep doing that recursively, and then when I get to the bottom, I then run through the innermost loop, which starts out with just one element to do, two, three.

And so the number of elements-- for example, in this case where I've done eight, I go through eight elements at the bottom here if this were an eight by eight matrix that I was transposing. So there's more work in these guys than there is over here. So it's not something you just can map onto processors in some naive fashion. It does take some load balancing to parallelize this loop. Any questions about what's going on here? Yeah?

AUDIENCE: Why is it that it's one, two, three, four, up to eight?

CHARLES LEISERSON: Take a look. The inner loop goes from j to i. So this guy just does one iteration of the inner loop, this guy does two, this guy does three, all the way up to this guy doing eight iterations, if it were an eight by eight matrix. And in general, if it's n by n, it's going from one to n work up here, but only one work down here. Because I'm basically iterating through a triangular iteration space to swap the matrix, and this is basically swapping row by row. Questions? Is that good? Everybody see what's going on?

So now let's take a look and let's analyze this for work and span. So what is the work of this in terms of n, if I have an n by n matrix? What's the work? The work is the ordinary serial running time, right? It's n squared. Good. So basically, it's order n squared, because these guys are all adding up. This is an arithmetic sequence up to n, and so the total amount in here is order n squared.

What about this part up here? How much does that cost us for work? How much is in the control overhead of doing that outer loop? So asymptotically, how much is in here? The total is going to be n squared. That I guarantee you. But what's going on up here? How do I count that up?

AUDIENCE: I'm assuming that each [? strain ?] is going to be constant time?

CHARLES LEISERSON: Yeah, well in this case, it is constant time, for these up here, because what am I doing? All I'm doing is the recursion code where I divide the range and then spawn off two things. That takes me only a constant amount of manipulation to be able to do that. So this is all order n. Total of order n here. The reason is because in some sense, there are n leaves here. And if you have a full binary tree, meaning every child is either a leaf or has two children, then the number of the internal nodes of the tree is one less than the number of leaves. So that's a basic property of trees, that the number of the internal nodes here is going to be n minus 1, in this case. In particular, we have 7 here. Is that good? So this part doesn't contribute significantly to the work. Just this part contributes to the work. Is that good?

What about the span for this? What's the span?

AUDIENCE: Log n.

CHARLES LEISERSON: It's not log n, but your heads are in the right place.

AUDIENCE: The longest path is going [INAUDIBLE].

CHARLES LEISERSON: Which is the longest path going to be here, starting here and ending there, which way do we go?

AUDIENCE: Go all the way down.

CHARLES LEISERSON: Which way?

AUDIENCE: To the right.

CHARLES LEISERSON: Down to the right, over, down through this guy. How big is this guy? n. Go back up this way. So how much is in this part going down?

AUDIENCE: Log n.

CHARLES LEISERSON: Going down and up is log n, but this is n. Good. So it's basically order n plus order log n, order n down here plus order log n up and down, that's order n. So the parallelism is the ratio of those two things, which is order n. So that's got good parallelism. And so if you imagine doing this in a large number processors, very easy to get sort of your benchmark of maybe 10 times more parallelism than the number of processors that you're running on. Everybody follow this? Good.

So the span of the loop control is order log n. And in general, when you have a four loop with n iterations, the loop control itself is going to have log n is going to have to be added to the span every time you hit a loop, log of whatever the number of iterations is. And then we have the maximum span of the body. Well, the worst case for this thing in the body is when it's doing the whole thing, because whenever we're looking at spans, we're always looking what's the maximum of things that are operating in parallel. Everybody good? Questions? Great.

So now let's do something a little more parallel. Let's make both loops be parallel. So here we have a cilk_for loop here, and then another cilk_for loop on the interior. And let's see what we get here. So what's the work for this?

AUDIENCE: n squared.

CHARLES LEISERSON: Yeah, n squared. That's not going to change. That's not going to change. What's the span?

AUDIENCE: log n.

CHARLES LEISERSON: Yeah, log n. So it's log n because the span of the outer control loop is going to add log n. The max span of the inner control loop, well, it's going from log of 1 up to log of i, but the maximums of those is going to be proportional to log n because it's not regular. And the span of the body now is going to be order 1. And so we add the logs because those things are in series. We don't multiply them. What we're doing is we're looking at, what's the worst case? The worst case is I have to do the control for this, plus the control for this, plus the worst iteration here, which in this case is just order one. So the total is order log n. That can be confusing for people, why it is that we add here rather than multiply or do something else. So let me pause here for some questions if people have questions. Everybody with us? Anybody want clarification or make a point that would lead to clarification? Yes, question.

AUDIENCE: If you were going to draw a tree like the previous slide, what would it look like?

CHARLES LEISERSON: Let's see. I had wanted to do that and it got out of control. So what it would look like is if we go back to the previous slide, it basically would look like this, except where each one of these guys is replaced by a tree that looks like this with as many leaves as the number here indicates. So once again, this would be the one with the longest span because this would be log of the largest number. But basically, each one of these would be a tree that came from this. Is that clear? That's a great question. Anybody else have as illuminating questions as those? Everybody understand that explanation, what the tree would look like? OK, good. Get

So the parallelism here is n squared over log n. Now it's tempting, when you do parallel programming, to say therefore, this is better parallel code. And the reason is, well, it does asymptotically have more parallelism. But generally when you're programming, you're not trying to get the most parallelism. What you're trying to do is get sufficient parallelism.

So if n is sufficiently large, it's going to be way more-- if n is a million-- which is typical problem size for a loop, for example, for a big loop, or even if it's a few thousand or whatever-- it may be just fine to have parallelism on the order of 1,000, which is what the first one gives you. So 1,000 iterations is generally a small number of iterations. So 1,000 by 1,000 matrix is going to generate parallelism of 1,000. Here, we're going to get a parallelism of 1 million divided by about 20, log of 10 or 20, so like 100,000.

But if I have 1,000 by 1,000 matrix, the difference between having parallelism of 1,000 and parallelism of 100,000, when I'm running on 100 cores, let's say, it doesn't matter. Up to 100 cores, it doesn't matter. And in fact, running this on 100 cores, that's really a tiny problem compared to the amount of memory you're going to get. 1,000 by 1,000 matrix is tiny when it comes to the size of memory that you're going to have access to and so forth.

So for big problems and so forth you really want to look at this and say, of things that have ample parallelism, which ones really are going to give me the best bang for the buck for reasonable machine sizes? That's different from things like work or serial running time. Usually less running time is better, and it's always better. But here parallelism-- yes, it's good to minimize your span, but you don't have to minimize it extremely. You just have to get it small enough, whereas the work term, that you really want to minimize, because that's what you're going to have to do, even in a serial implementation. Question.

AUDIENCE: So are you suggesting that the other code was OK?

CHARLES LEISERSON: We're going to look a little bit closer at the issue of overheads. We're now going to take a look at what's the difference between these two codes? We'll come back to that in a minute. The way I want to do it is take a look at the issue of overheads with a simpler example, where we can see what's really going on. So here, what I've done is I've got a loop that is basically just doing vector addition. It's adding for i equals 0 to n minus 1, add b of i into a of i. Pretty simple code, and we want to make that be a parallel loop.

So I get a recursion tree that looks like this, where I have constant work at every step there. And of course, the work is order n, because I've got n leaves. And the number of internal nodes, the control, is all constant size strands there. So this is all just order n for work. And the span is basically log n, as we've seen, by going down one of these paths, for example. And so the parallelism for this is order n over log n. So a very simple problem.

But now let's take a look more closely at the overheads here. So the problem is that this work term contains substantial overhead. In other words, if I really was doing that, if I hadn't coarsened the recursion at all in the implementation of cilk_for, if the developers hadn't done that, then I've got a function call, I've got n function calls here for doing a single addition of values at the leaves. I've got n minus one of these guys, that's approximately n, and I've got n of these guys. And which are bigger, these guys or these guys? These guys are way bigger. They've got a function call in there. This guy right here just has what? One floating point addition.

And so if I really was doing my divide and conquer down to a single element, this would be way slower on one processor than if I just ran it with a for loop. Because if I do a for loop, it's just going to go through, and the overhead it has is incrementing i and testing for termination. That's it. And of course, that's a predictable branch, because it almost never terminates until it actually terminates, and so that's exactly the sort of thing that's going to have a really, really tight loop with very few instructions.

But in the parallel implementation, there's going to be this function call overhead everywhere. And so therefore, this cilk_for loop in principle would not be as efficient. It actually is, but we're going to explain why it is, what goes on in the runtime system, to understand that.

So here's the idea, and you can control this with a pragma. So a pragma is a statement to the compiler that gives it a hint. And here, the pragma says, you can name a grain size and give it a value of g. And what that says is rather than just doing one element when you get down to the bottom here, do g elements in a for loop when you get down to the bottom. And that way, you halt the recursion earlier. You have fewer of these internal nodes. And if you make the grain size sufficiently large, the cost of the recursion at the top you won't be able to see.

So let's analyze what happens when we do this. So we can understand this vis a vis this equation. So the idea here is, if I look at my work, imagine that t iter is the time for iteration of one iteration of the loop, basic time for one iteration of the loop. So the amount of work that I have to do is n times the time for the iterations of the loop. And then depending upon my grain size, I've got to do things having to do with the internal nodes, and there's basically going to be n over g of those, times the time for a spawn, which is I'm saying is the time to execute one of these things. So if these are batched into groups of g, then there are n over g such leaves. There's a minus 1 in here, but it doesn't matter. It's basically n over g times the time for the internal nodes. So everybody see where I'm getting this? So I'm trying to account for the constants in the implementation. People follow where I'm getting this? Ask questions. I see a couple of people who are sort of going, not sure I understand. Yes?

AUDIENCE: The constants [INAUDIBLE].

CHARLES LEISERSON: Yes. So basically, the constants are these t iter and t spawn. So t spawn is the time to execute all that mess. t iter is the time to execute one iteration within here. I'm doing, in this case, g of them. So I have n over g leaves, but each one is doing g, so it's n over g times g, which is a total of n iterations, which makes sense. I should be doing n iterations if I'm adding two vectors here. So that's accounting for all the work in these guys. Then in addition, I've got all of the work for all the spawning, which is n over g times t spawn.

And as I say, you can play with this yourself, play with grain size yourself, by just sticking in different grain size directives. Otherwise it turns out that the cilk runtime system will pick what it deems to be a good grain size. And it usually does a good job, except sometimes. And that's why there's a parameter here. So if there's a parameter there, you can get rid of that. Yes?

AUDIENCE: Is the pragma something that is enforced, or is it something that says, hey--

CHARLES LEISERSON: It's a hint.

AUDIENCE: It's a hint.

CHARLES LEISERSON: Yes, it's a hint. In other words, compiler could ignore it.

[? AUDIENCE: The compiler is ?] going to be like, oh, that's the total [INAUDIBLE] constant.

CHARLES LEISERSON: It's supposed to be something that gives a hint for performance reasons but does not affect the correctness of the program. So the program is going to be doing the same thing regardless. The question is, here's a hint to the compiler and the runtime system. And so then it's picked at-- yeah?

AUDIENCE: My question is, so there's these cases where you say that the runtime system fails to find an appropriate value for that [INAUDIBLE]. I mean, basically, chooses one that's not as good. If you put a pragma on it, will the runtime system choose the one that you give it, or still choose--

CHARLES LEISERSON: No, if you give it, the runtime system will-- in the current implementation, it always picks whatever you say is here. And that can be an expression. You can evaluate something there. It's not just a constant. It could be maximum of this and that times whatever, et cetera. Is that good? So this is a description of the work. Now let's get a description with the constants again of the span. So what is going to be the constants for the span? Well, I'm executing this part in here now serially.

So for the span part, we're basically going to go down on one of these paths and back up I'm not sure which one, but they're basically all fairly symmetric. But then when I get to the leaf, I'm executing the leaf serially. So I'm going to have whatever the cost is, g times the time per iteration, is going to be executed serially, plus now log of n over g-- n over g is the number of things I have here-- times the cost of the spawn, basically. Does that make sense?

So the idea is, what do we want to have here if I want a good parallel code? We would like the work to be as small as possible. How do I make the work small? How can I set g to make the work small?

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: Make g--

AUDIENCE: Square root of n.

CHARLES LEISERSON: Well, make g big or little? If you want this term to be small, you want g to be big. But we also want to have a lot of parallelism. So I want this term to be what? Small, which means I need to make g what? Well, we got an n over g here, but it's in a log. It's minus log. So really, to get this small, I want g to be small. So I have tension, trade off. I have trade off.

So let's analyze this a little bit. Essentially, if I look at this, I want g to be bigger-- from this one I want g to be small. But here, what I would like is to make it so that this term dominates this term. If the first term here dominates the second term, then the work is going to be the same as if I did an ordinary for loop to within a few percent. So therefore, I want t span over t iter, if I take the ratio of these things, I want g to be bigger than the time to spawn divided by the time to iterate. If I get it much bigger than that, then this term will be much bigger than that term and I don't have to worry about this term. So I want it to be much bigger, but I want to be as small as possible. There's no point in making it much bigger than that which causes this term to essentially be wiped out. People follow that?

So basically, the idea is we pick a grain size that's large but not too large, is what you generally want to do, so that you have enough parallelism, but you don't. The way that the runtime system does it is it has a somewhat complicated heuristic, but it actually looks to see how many processors you're running on. And it uses a heuristic that says, let's make sure there's at least parallelism 10 times more than the number of processors. But there's no point in having more iterations than like 500 or something, because at 500 iterations, you can't see the spawn overhead regardless. So basically, it uses a formula kind of that nature to pick this automatically. But you're free to pick this yourself.

But you can see the point is that although it's doing divide and conquer, you do this issue of coarsening and you do want to make sure that you have enough work to do in any of the leaves of the computation. And as I say, usually it'll guess right. But if you have trouble with that, you have a parameter you can play with.

Let's take a look at another implementation just to try to understand this issue. Suppose I'm going to do a vector add. So here I have a vector add of two arrays, where I'm basically saying ai gets the value of b added into it. That's kind of the code we had before. But now, what I want to do is I'm going to implement a vector add using cilk spawn. So rather than using a cilk_for loop, I'm going to parallelize this loop by hand using cilk spawn. What I'm going to do is I'm going to say for j equals 0 up to-- and I'm going to jump by whatever my grain size is here-- and spawn off the addition of things of size, essentially, g, unless I get close to the end of the array. But basically, I'm always spawning off the next g iterations to do that in parallel. And then I sync all these spawns.

So everybody understand the code? I see nods. I want to see everybody nod, actually, when I do this. Otherwise what happens is I see three people nod, and I assume that people are nodding. Because if you don't do it, you can shake your head, and I promise none of your friends will see that you're shaking your head. And since the TAs are doing the grading and they're facing this way, they won't see either. And so it's perfectly safe to let me know, and that way I can make sure you understand. So everybody understand what this does? OK, so I see a few more. No. OK, question? Do you have a question, or should I just explain again?

So this is basically doing a vector add of b into a, of n iterations here. And we're going to call it here, when I do a vector add, of basically g iterations. So what it's doing is it's going to take my array of size n, bust it into chunks of size g, and spawn off the first one, spawn off the second one, spawn off the third one, each one to do g iterations. That make sense? We'll see it. So here's sort of the instruction stream for the code here.

So basically, it says here is one, we spawn off something of size g, then we go on, we spawn off something else of size g, et cetera. We keep going up there until we hit the cilk sync. That make sense? Each of these is doing a vector add of size g using this serial routine. So let's analyze this to understand the efficiency of this type of looping structure.

So let's assume for this analysis that g equals 1, to make it easy, so we don't have to worry about it. So we're simply spawning off one thing here, one thing here, one iteration here, all the way to the end. So what is the work for this, if I spawn off things of size one, asymptotic work? It's order n, because I've got n leaves, and I've got n guys that I'm spawning off. So it's order n. What's the span?

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: Yeah, it's also order n, because the critical path goes something like brrrup, brrrup, brrrup. That's order n length. It's not this, because that's only order one length, all those. The longest path is order n. So that says the parallelism is order one. Conclusion, at least with grain size one, this is a really bad way to implement a parallel loop. However, I guarantee, it may not be the people in this room, but some fraction of students in this class will write this rather than doing a cilk for. Bad idea. Bad idea. Generally, bad idea. Question?

AUDIENCE: Do you think you could find a constant factor, not just [INAUDIBLE]?

CHARLES LEISERSON: Well here, actually, with grain size one, this is really bad, because I've got this overhead of doing a spawn, and then I'm only doing one iteration. So the ideal thing would be that I really am only paying for the leaves, and the internal nodes, I don't have to pay anything for. Yeah, Eric?

AUDIENCE: Shouldn't there be a sort of keyword in the b add too?

CHARLES LEISERSON: In the where?

AUDIENCE: In the b add?

CHARLES No, that's serial. That's a serial code.

LEISERSON:

AUDIENCE: No, but if you were going to call it with cilk spawn, don't you have to declare it cilk? Is that not the case?

CHARLES LEISERSON: No.

AUDIENCE: Never mind.

CHARLES LEISERSON: Yes, question.

AUDIENCE: If g is [INAUDIBLE], isn't that good enough?

CHARLES LEISERSON: Yeah, so let's take a look. That's actually the next slide. This is basically, this we call puny parallelism. We don't like puny parallelism. It doesn't have to be spectacular. It has to be good enough. And this is not good enough for most applications. So here's another implementation. Here's another way of doing it.

Now let's analyze it where we have control over g. So we'll analyze it in terms of g, and then see whether there's a setting for which this make sense. So if I analyze it in terms of g, now I have to do a little bit more careful analysis here. How much work do I have here in terms of n and g?

AUDIENCE: It's the same.

CHARLES LEISERSON: Yeah, the work is still asymptotically order n. Because I always have n work in the leaves, even if I do more iterations here. What increasing g does is it shrinks this, right? It shrinks this. The span for this is what? So I heard somebody say it. n over g plus g. And it corresponds to this path. So this is the n over g part up here, and this is the plus g.

So what we want to do to minimize this, is we can minimize this. This has the smallest value when these two terms are equal, which you can either know as a basic fact of the summation of these kinds of things, or you could take derivatives and so forth. Or you can just eyeball it and say, gee, if g is bigger than square root of n, then this is going to be the dominant, and if g is smaller than square root of n, then this is going to be dominant. And so when they're equal, that sounds like about when it should be the smallest, which is true. So we pick it to be about square root of n, to minimize the span. Since g, I don't have anything to minimize here. So pick it around square root of n, then the span is around square root of n. And so then the parallelism is order square root of n. So that's pretty good. So that's not bad. So for something that's a big array, array of size 1 million, parallelism might be 1,000. That might be just hunky dory. Question. What's that?

AUDIENCE: I don't see where--

CHARLES LEISERSON: We've picked g to be equal to square root of n.

AUDIENCE: [INAUDIBLE] plus n over g, plus g. I don't see where [INAUDIBLE].

CHARLES LEISERSON: You don't see where this g came from? This g comes from, because I'm doing g iterations here. So remember that these are now of size g. I'm doing g iterations in each leaf here, if I set g to be large. So I'm doing n over g pieces here, plus g iterations in my [INAUDIBLE]. Is that clear? So the n over g is this part. This size here, this is not one. This has g iterations in it. So the total span is g plus n over g. Any other questions about this?

So basically, I get order square root of n. And so this is not necessarily a bad way of doing it, but the cilk for is a far more reliable way of making sure that you get the parallelism than spawning things off one by one. One of the things, by the way, in this, I've seen people write code where their first instinct is to write something like this, where this that they're spawning off is only constant time. And they say, gee, I spawned off n things. That's really parallel. When in fact, their parallelism is order one. So it's really seductive to think that you can get parallelism by this, [? right. ?] It's much better to do divide and conquer, and cilk for does that for you automatically. If you're going to do it by hand, sometimes you do want to do it by hand, then you probably want to think more about divide and conquer to generate parallelism, because you'll have a small span, than doing many things one at a time.

So here's some tips for performance. So you want to minimize the span, so the parallelism is the work over the span. So you want to minimize the span to maximize parallelism. And in general, you should try to generate something like 10 times more parallelism than processors, if you want to get near perfect linear speed-up. In other words, a parallel slackness of 10 or better is usually adequate. If you can get more, you're now talking that you can get more performance, but now you're getting performance increases in the range of 5% or so, 5% to 10%, something like that.

Second thing is if you have plenty of parallelism, try to trade some of it off to reduce work overhead. So this is a general case. This is what actually goes on underneath in the cilk++ runtime system, is they are trying to do this themselves. But you in your own code can play exactly the same game. Whenever you have a problem and it says, whoa, look at all this parallelism, think about ways that you could reduce the parallelism and get something back in the efficiency of the work term, because the performance in the end is going to be something like t1 over p plus t infinity. If t infinity is small, it's like t1 over p, and so anything you save in the t1 term is saving you overall. It's going to be a savings for you overall.

Use divide and conquer recursion on parallel loops rather than sprawling one small thing after another. In other words, do this not this, generally. And here's some more tips. Another thing that can happen that we looked at here was make sure that the amount of work you're doing is reasonably large compared to the number of spawns. You could also say this is true when you do recursion for function calls. Make sure if you're just in serial programming, you always want to make sure that the amount of work you're doing is small compared to the number of function calls are doing if you can, and that'll make things go faster. So same thing here, you want to have a lot of work compared to the total number of spawns that you're doing in your program. So spawns, by the way, in this system, are about three or four times the cost of a function call. They're sort of the same order of magnitude as a function call, a little bit heavier than a function call. So you can spawn pretty readily, as long as the total number of spawns you're doing isn't dominating your work.

Generally parallelize outer loops as opposed to inner loops if you're forced to make a choice. So it's always better to have an outer loop that runs in parallel rather than an inner loop that runs in parallel, because when you do an inner loop that runs in parallel, you've got a lot of overhead to overcome. But in an outer loop, you've got all of the inner loop to amortize against the cost of the spawns that are being used to parallelize the outer loop. So you'll do many fewer spawns in the implementation.

Watch out for scheduling overheads. So if you do something like this-- so here we're paralyzing an inner loop rather than an outer loop. Now this turns out, it doesn't matter which order we're going in or whatever. It's generally not desirable to do this because I'm paying scheduling overhead n times through this loop, whereas here, I pay for scheduling overhead just twice. So is generally better, if I have n pieces of work to do, rather than, in this case, parallelizing-- let me slow down here.

So let's look at what this code does. This says, go for two iterations. Do something for which it is going to take n iterations for j. So two iterations for i, n iterations for j. If you look at the parallelism of this, what is the parallelism of this assuming that f is constant time? What's the parallelism of this code? Two. The parallelism of two, because I've got two things on the outer loop here, and then each is n. So my span is essentially n. My work is like 2n, something like that. So it's got a parallelism of that, too.

What's the parallelism of this code? What's the parallelism? It's not n, because I'm basically going through this serially, the outer loop serially. What's the theoretical parallelism of this? So for each iteration here, the parallelism is two. No, not n. It can't be n, because I'm basically only parallelizing two things, and I'm doing them serially. The outer loop is going serially through the code and it's spawning off two things, two things, two things, two things, two things. And waiting for them to be done, two things, wait for it to be done, two things, wait for it to be done. So the parallelism is two.

These have the same parallelism. However if you run this, this one will give you a speedup of two on two cores, very close to it. Because there's the scheduling overhead here, you've only paid once for the scheduling overhead, and then you're doing a whole bunch of stuff. So remember, to schedule it, it's got to be migrated, it's got to be moved to another processor, et cetera. This one, it's not even worth it probably to steal each of these individual things. You're spawning off things that are so small, this may even have parallelism that's less than 1 in practice.

And if you look at the cilkview tool, this will show you a high burden parallelism. Because the cilkview tool, the burden parallelism tells you what the overhead is from scheduling, as well as what the actual parallelism is. And it recognizes that oh, gee whiz. This thing really has very small-- there's almost no work in here. So you're trying to parallelize something where the work is so small, it's not even worth migrating it to take advantage of it. So those are some tips.

Now let's go through and analyze a bunch of algorithms reasonably quickly. We'll start with matrix multiplication. People seen this problem before? Here's the matrix multiplication problem. And let's assume for simplicity that n is a power of 2. So basically, let's start out with just our looping version. In fact, this isn't even a very good looping version, because I've got the order of the loops wrong, I think. But it is just illustrative. Basically let's parallelize the outer two loops. I can't parallelize the inner loop. Why not? What happens if I tried to parallelize the inner loop with a cilk_for in this implementation? Why can't I just put a cilk_for there? Yes, somebody said it.

AUDIENCE: It does that in cij.

CHARLES LEISERSON: Yeah, we get a race condition. We have more than two things in parallel trying to update the same cij, and we'll have a race condition. So always run cilkview to tell your performance. But always, always, run cilk screen to tell whether or not you've got races in your code. So yeah, you'll have a race condition if you try to naively parallelize the loop here.

So the work of this is what? It's order n cubed, just three nested loops each going to n. What's the span of this? What's the span of this? It's order n, because it's log n for this loop, log n for this loop, plus the maximum of this, well, that's n. Log n plus log n plus n is order n. So order n span, which says parallelism is order n squared. So for 1,000 by 1,000 matrices, the parallelism is on the order of a million. Wow. That's great.

However, it's on the order of a million, but as we know, this doesn't use cache very effectively. So one of the nice things about doing divide and conquer is, as you know, that's a really good way to take advantage of caching. And this works in parallel, too. In particular because whenever you have sufficient parallelism, these processors are executing the code just as if they were executing serial code. So you get all the same cache locality you would get in the serial code in the parallel code, except for the times that you're actually migrating work. And if you have sufficient parallelism, that isn't too often.

So let's take a look at recursive divide and conquer multiplication. So we're familiar with this, too. So this is eight multiplications of n over 2 by 2 matrices, and one addition of n by n matrix. So here's a code using a little bit of C++ism. So I've made the type a variable t. So we're going to do matrix multiplication of an array, a, the result is going to go in c, and we're going to basically have a and b, and we're going to add the result into c. We have n, which is the side of the submatrix that we're working on, and we're also going to have an n size, which is the length of the row in the original matrix. So remember when we do matrix things, if I take a submatrix, it's not contiguous in memory. So I have to know the row size of the matrix that I'm in in order to be able to calculate what the elements are.

So the way it's going to work is I'm going to assign this temporary d, by using the new-- which is basically memory allocation C++-- array of size n by n. And what we're going to do is then do four of the recursive multiplications, these guys here, into the elements of c, and then four of them also into d using the temporary. And then we're going to sync, after we get all that parallel work done, and then we're going to add d into c, and then we'll delete d, because we allocated it up here. Everybody understand the code? So we're doing this, it's just we're going to do it in parallel. Good? Questions? OK.

So this is the row length of the matrices so that I can do the base cases, and in particular, partition the matrices effectively. I haven't shown that code. And of course, the base case, normally, we would want to coarsen for efficiency. I would want to go down to something like maybe a eight by eight or 16 by 16 matrix, and at that point switch to something that's going to use the processor pipeline better. The base cases, once again, I want to emphasize this because a couple people on the quiz misunderstood this. The reason you coarsen has nothing to do with caches. The reason you coarsen is to overcome the overhead of the function calls, and the coarsening is generally chosen independent of what the size of the caches are. It's not a parameter that has to be tuned to cache size. It's a parameter that has to be tuned to function call, versus ALU instructions, and what that balance is. Question?

AUDIENCE: I mean, I understand that's true, but I thought-- I mean, maybe I [? heard the call ?] wrong, but I thought we wanted, in general, in terms of caching, that you would choose it somehow so that all of the data that you have would somehow fit--

CHARLES LEISERSON: That's what the divide and conquer does automatically. The divide and conquer keeps halving it until it fits in whatever size cache you have. And in fact, we have three caches on the machines we're using.

AUDIENCE: Yeah, but I'm saying if your coarsened constant is too big, that's not going to happen.

CHARLES LEISERSON: If the coarsened constant is too big, that's not going to happen. But generally, the caches are much bigger than what you need to do to amortize the cost. But you're right, that is an assumption. The caches are generally much bigger than the size that you need in order to overcome function call overhead. Function call overhead is not that high. OK? Good. I'm glad I raised that issue again. And so we're going to determine the submatrices by index calculation. And then we have to implement this parallel add, and that I'm going to do just with a doubly nested for loop to add the things. There's no cache behavior I can really take advantage of here except for spatial locality. There's no temporal locality because I'm just adding two matrices once, so there's no real temporal locality that I'll get out of it. And here, I've actually done the index calculations by hand.

So let's analyze this. So to analyze the multiplication program, I have to start by analyzing the addition program. So this should be, I think, fairly straightforward. What's the work for adding two n by n matrices here? n squared, good, just doubly nested loop. What's the span?

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: Yeah, here it's log n, very good. Because I've got log n plus log n plus order one. I'm not going to analyze the parallelism, because I really don't care about the parallelism of the addition. I really care about the parallelism of the matrix multiplication. But we'll plug those values in now. What is the work of the matrix multiplication? Well for this, what we want to do is get a recurrence that we can then solve. So what's the recurrence that we want to get for m of 1 of n? Yeah, it's going to be 8m sub 1 of n over 2, that corresponds to these things, plus some constant stuff, plus the work of the addition. Does that make sense?

We analyze the work of the addition. What's the work of the addition? Order n squared. So that's going to dominate that constant there, so we get 8. And what's the solution to this? Back to Master Theorem. Now we're going to start pulling out the Master Theorem multiple times per slide for the rest of the lecture. n cubed, because we have log base 2 of 8. That's n cubed compared with n squared. So we get a solution which is n cubed-- Case 3 of the Master Theorem. So that's good. The work we're doing is the same asymptotic work we're doing for the triply nested loop.

Now let's take a look at the span. So what's the span for this? So once again, we want a recurrence. What's the recurrence look like? So the span of this is going to be the span of-- it's going to be the sum of some things. But the key observation is that it's going to be-- we want the maximum of these guys. So we're going to basically have the allocation as constant time, we have the maximum of these, which is m of infinity of n over 2, and then we have the span of the add. So we get this recurrence. m infinity sub n over 2, because we have only to worry about the worst of these guys. The worst of them is-- they're all symmetric, so it's basically the same. We have a of n, and then there's a constant amount of other overhead here. Any questions about where I pulled that out of, why that's the recurrence?

So this is the addition, the span of the addition of this guy that we analyzed already. What is the span of the addition? What did we decide that was? log n. So basically, that dominates the order one. So we get this term, and what's the solution of this recurrence?

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: What case is this?

AUDIENCE: [INAUDIBLE] log n squared.

CHARLES LEISERSON: Yes, it's log squared n. So basically, it's case two. So if I do n to the log base b of a, that's n to the log base 2 of 1, that's just 1. And so this is basically a logarithmic factor times the 1, so we add an extra log. We get log squared n. That's just Master Theorem plugging in.

So here, the span is order log squared n. And so we have the work of n cubed, the span of log squared n, so the parallelism is the ratio, which is n cubed over log squared n. Not too bad for a 1,000 by 1,000 matrices, the parallelism is about 10 million. Plenty of parallelism. So let's use the fact that we have plenty of parallelism to say, let's get rid of some of that parallelism and put it back into making our code more efficient.

So in particular, this code uses an extra temporary d, which it allocates here and it deletes here. And generally, there's a good rule that says, if you use more storage you're going to use more time, because you're going to have to look at that storage, it's going to take up space in your cache, and it's generally going to make you slower. So things that use less storage are generally faster. Not always the case, sometimes there's a trade off. But often it's the case, use more storage, it runs slower. So let's get rid of this guy. How do we get rid of this guy? Yeah?

AUDIENCE: [INAUDIBLE PHRASE].

CHARLES LEISERSON: You're going to do this serially, you're saying?

AUDIENCE: Yeah, you do those serially in add.

CHARLES LEISERSON: If you do this serially in add, it turns out if you do that, you're going to be in trouble because you're going to not have very much parallelism, unfortunately. Actually, analyzing exactly what the parallelism is there is actually pretty good. It's a good puzzle. Maybe we'll do that on the quiz, the take home problem set we're calling it now, right? We're going to have a take home problem set, maybe that's a good one.

Yeah, so the idea is, you can sync. And in particular, why not compute these, then sync, and then compute these, adding their results into the places where we added these in? So it's making the program more serial, because I'm putting in a sync. That shouldn't have an impact on the work, but it will have an impact on the span. So we're going to trade it off, and the way we'll do that is by putting essentially a sync in the middle. And since they're adding it in, I don't even have we call the addition routine, because it's just going to add it in in place. So I spawn off these four guys, putting their results into c, then I spawn off these four guys, and they add their results into c. Is that clear what the code is? So let's analyze this.

So the work for this is order n cubed. It's the same as anything else, we can come up with a recurrence, slightly different from before because I only have an order one there, but it doesn't really matter. The answer is order n cubed. The span, now this gets a little trickier. What's the recurrence of the span?

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: What is that?

AUDIENCE: Twice the span of m of n over 2.

CHARLES LEISERSON: Twice the span of m of n over 2, that's right. So basically, we have the maximum of these guys, the maximum of these guys, and then this is making those things be in series. So things that are in parallel I take the max, if it's in series, I have to add them. So I end up with 2m infinity of n over 2 plus order one. Does that make sense? OK, good.

So let's solve that recurrence. What's the answer to that one? That's order in. Which case is it? I never know what the cases are. I know two, but one and three, it's like-- they're the same thing, it's just which side it's in, so I never remember what the number is. But anyway, case one, yes. Case one. It's the one where this thing is bigger, so that's order n. Good.

So then the work is n cubed, the span is order n, the parallelism is order n squared. So for 1,000 by 1,000 matrices, I get parallelism on the order of a million, instead of before, I had parallelism on the order of 10 million. So this turns out way better code than the previous one because it avoids the temporary and therefore runs, you get a constant factor improvement for that, and it's still, on 12 cores, it's going to run pretty fast. And in practice, this is a much better way to do it.

The actual best code that I know for doing this essentially does divide and conquer in only one dimension at a time. So basically, it looks to see what's the long dimension, and whatever the long dimension is, it slices it in half and then recurses, and just does that as a binary thing. And it basically is the same work, et cetera. It's a little bit more tricky to analyze.

Let me quick do merge sort. So you know merge sort. There's merging two sorted arrays, we saw this before. If I spend all this time doing animations, I might as well get my mileage out of it. There we go. So you merge, that's basically what this code does. Order n time to merge. So here's merge sort. So what I'll do in merge sort is the same thing I normally do, except that I'll make recursive routines go in parallel. So when I do that, it basically divide and conquers down, and then it sort of does this to merge things together. So we saw this before, except now, I've got the fact that I can sort two things in parallel rather than sorting them serially.

So let's take a look at the work. What's the work of merge sort? We know that. n log n, right? 2t of n over 2 plus order n, so that's order n log n. The span is what? What's the recurrence of the span? So we're going to take the maximum of these two guys. So we only have one term that involves t infinity, and then the merge costs us order n, so we get this recurrence. So that says that the solution is order n. So therefore, the work is n log n, the span is order n, and so the parallelism is order log n. Puny. Puny parallelism. Log n is like, you can run it, and it'll work fine on a few cores, but it's not to be something that generally will scale and give you a lot of parallelism.

So it's pretty clear from this that the bottleneck-- where's all the span going to? It's going to that merge. So when you understand that that's the structure of it, now you say if you want to get parallelism, you've got to go after the merge. So here's how we parallelize the merge. So we're going to look at merging of two arrays that are of possibly different length. So one we'll call A, and one we'll call B, with na and nb elements. And let me assume without loss of generality that na is greater than or equal to nb, because otherwise I can just switch the roles of A and B.

So the way that I'm going to do it is I'm going to find the middle element of A. These are sorted arrays that I'm going to merge. I find the middle element of A, so these guys are or less than or equal to a of ma, and these are greater than or equal to. And now I binary search and find out where that middle element would fall in the array B. So that costs me log n time to binary search. Remember binary search?

Then what I'm going to do is recursively merge these guys, because these are sorted and less than or equal to ma, recursively merge those and put this guy in the middle. So when I do that, the key question when we analyze-- it turns out the work is going to basically be the same, but the key thing is going to be what happens to the span? And the idea here is that the total number of elements in the larger of these two things is going to be at most what? Another way of looking at it is in the smaller partition, if n is the total number of elements, the smaller partition has how many elements at least relative to n? No matter where this binary search finds itself. So the worst case is sort of going to come when this guy is like at one end or the other. And then the point is that because A is the larger array, at least a quarter of the elements will still be in the smaller partition. Of all the elements here, at least a quarter will be in the smaller partition, which will occur when B is equal to in size to A. So the number, in the larger of the recursive merges, is at most 3/4 n. Sound good?

That's the main, key idea behind this. So here's the parallel merge. Basically you do binary search, you spawn, then, the two merges. Here's one merge, and here's the other merge, and then you sync. So that's the code for the doing the parallel merge. And now you want to incorporate that parallel merge into the parallel merge sort. Of course, you coarsen the base cases for efficiency.

So let's analyze the span of this. So the span is basically then the span of something of 3/4, at most 3/4, the size plus the log n for the binary search. So the span of parallel merge is therefore order log squared n, because the important thing is, I'm whacking off a constant fraction here every time. So I get log squared n as the span, and the work I get this hairy recurrence, that it's t of alpha n plus t1 minus alpha n plus log n, where alpha falls in this range. This does not satisfy the Master Theorem. You can actually do this pretty easily with a recursion tree, but the way to verify is-- we call this technically a hairy recurrence. That's the technical term for it. So it turns out, this has order n, just like ordinary merge, order n time. here's You can use the substitution method, and I won't drag you through it, but you can look at it in the notes. And this should be very familiar to you as having all aced 6006, right? Otherwise you wouldn't be here, right?

So the parallelism of the parallel merge is something like n over log squared n. So that's much better than having n order n bound. And now, we can plug it into merge sort. So the work is going to be the same as before, because I just have the work of the merge, which is still order n. So the work is order n log n, once again pulling out the Master Theorem. And then the span is n over 2 plus log n, because basically, I have the span of a problem of half the size plus the span that I need to merge things. That's order log squared n.

This I want to pause on for moment. People get this recurrence? Because this is the span of the merge. And so what I end up with is I get another log, log cubed n. And so the total parallelism is n over log squared n. And this is actually quite a practical thing to implement, to get the n over log squared n parallelism versus just a log n parallelism. We're not going to do tableau construction. You can read that up, that's on the notes that are online, but you should read through that part of it. It's got some nice animations which you don't get to see. This is like when you do longest common subsequence and stuff like that, how you would solve that type of problem in parallel. OK, great.