6.046J Introduction to Algorithms, Fall 2005
Transcript – Lecture 23

OK, good morning. So today, we're going to continue our exploration of multithreaded algorithms. Last time we talked about some aspects of scheduling, and a little bit about linguistics to describe a multithreaded competition. And today, we're going to actually deal with some algorithms. So, we're going to start out with a really simple, actually, what's fun about this, actually, is that everything I'm going to teach you today I could have taught you in week two, OK, because basically it's just taking the divide and conquer hammer, and just smashing problem after problem with it.

OK, and so, actually next week's lectures on caching, also very similar. So, everybody should bone up on their master theorem and substitution methods for occurrences, and so forth because that's our going to be doing. And of course, all the stuff will be on the final. So let's start with matrix multiplication. And we'll do n by n. So, our problem is to do C equals A times B. And the way we'll do that is using divide and conquer, as we saw before, although we're not going to use Strassen's method.

OK, we'll just use the ordinary thing, and I'll leave Strassen's as an exercise. So, the idea is we're going to look at matrix multiplication in terms of an n by n matrix, in terms of n over 2 by n over 2 matrices. So, I partition C into four blocks, and likewise with A and B. OK, and we multiply those out, and that gives us the following. Make sure I get all my indices right. OK, so it gives us the sum of these two n by n matrices. OK, so for example, if I multiply the first row by the first column, I'm putting the first term, A_1-1 times B_1-1 in this matrix, in the second one, A_1-2 times B_2-1 gets placed here.

So, when I sum them, and so forth, for the other entries, and when I sum them, I'm going to get my result. So, we can write that out as a, let's see, I'm not sure this is going to all fit on one board, but we'll see we can do. OK, so we can write that out as a multithreaded program. So this, we're going to see that n is an exact power of two for simplicity. And since we're going to have two matrices that we have to add, we're going to basically put one of them in our output, C; that'll be the first one, and we're going to use a temporary matrix, T, which is also n by n.

OK, and the code looks something like this, OK, n equals one, and C of one gets A of 1-1 times B of 1-1. Otherwise, what we do then is we partition the matrices. OK, so we partition them into the block. So, how long does it take me to partition at matrix into blocks if I'm clever at my programming? Yeah? No time, or it actually does take a little bit of time. Yeah, order one, basically, OK, because all it is is just index calculations. You have to change what the index is. You have to pass in what you're passing these in addition to A, B, and C for example, pass and arrange which would have essentially a constant overhead. But it's basically order one time.

Basically order one time, OK, to partition the matrices because all we are doing is index calculations. And all we have to do is just as we go through, is just make sure we keep track of the indices, OK? Any questions about that? People follow? OK,

that's sort of standard programming. So then, what I do is I spawn multiplication of, woops, the sub-matrices, and spawn -- -- and continue, C_2-1, gets A_2-1, B_1-1, two, and let's see, 2-2, yeah, it's 2-1. OK, and continuing onto the next page.

Let me just make sure I somehow get the indentation right. This is my level of indentation, and I'm continuing right along. And now what I do is put the results in T, and then -- OK, so I've spawn off all these multiplications. So that means when I spawn, I get to, after I spawn something I can go onto the next statement, and execute that even as this is executing. OK, so that's our notion of multithreaded programming. I spawn off these eight things. What do I do next? What's the next step in this code? Sync. Yeah. OK, I've got to wait for them to be done before I can use their results.

OK, so I put a sync in, say wait for all those things I spawned off to be done, and then what? Yeah. That you have to add T and C. So let's do that with a subroutine call. OK, and then we are done. We do a return at the end. OK, so let's write the code for add, because add, we also would like to do in parallel if we can. And what we are doing here is doing C gets C plus T, OK? So, we're going to add T into C. So, we have some code here to do our base case, and partitioning because we're going to do it divide and conquer as before.

And this one's actually a lot easier. We just spawn, add a C_1-1, T_1-1, n over 2, C_1-2, T_1-2, n over 2, C_2-1, T_2-1, n over 2, C_2-2, 2-2-2, n over 2, and then sync, and return the result. OK, so all we're doing here is just dividing it into four pieces, spawning them off. That's it. OK, wait until they're all done, then we return with the result. OK, so any questions about how this code works? So, remember that, here, we're going to have a scheduler underneath which is scheduling this onto our processors. And we're going to have to worry about how well that scheduler is doing. And, from last time, we learned that there were two important measures, OK, that can be used essentially to predict the performance on any number of processors. And what are those two measures?

Yeah, T_1 and T infinity so that we had some names. T_1 is the work, good, and T infinity is critical path length, good. So, you have to work in the critical path length. If we know the work in the critical path length, we can do things like say what the parallelism is of our program, and from that, understand how many processors it makes sense to run this program on. OK, so let's do that analysis. OK, so let's let M_P of n be the p processor execution time for our mult code, and A_P of n be the same thing for our matrix addition code.

So, the first thing we're going to analyze is work. And, what do we hope our answer to our work is? What we analyze work, what do we hope it's going to be? Well, we hope it's going to be small. I'll grant you that. What could we benchmark it against? Yeah, if we wrote just something that didn't used to have any parallelism. We'd like our parallel code when run on one processor to be just as fast as our serial code, the normal code that we would use to write to do this problem.

That's generally the way that we would like these things to operate, OK? So, what is that for matrix multiplication in the naïve way? Yeah, it's n^3. Of course, we use Strassen's algorithm, or one of the other, faster algorithms beat n^3. But, for this problem, we are just going to focus on n^3. I'm going to let you do the Strassen as an exercise. So, let's analyze the work. OK, since we have a subroutine for add that we are using in the multiply code, OK, we start by analyzing the add. So, we have

A_1 of n, OK, is, well, can somebody give me a recurrence here? What's the recurrence for understanding the running time of this code?

OK, this is basically week two. This is lecture one actually. This is like lecture two or, at worst, lecture three. Well, A of 1 of n. Plus order one, right. OK, that's right. So, we have four problems of size n over 2 that we are solving. OK, so to see this, you don't even have to know that we are doing this in parallel, because the work is basically what would happen if it executed on a serial machine. So, we have four problems of size n over 2 plus order one is the total work.

Any questions about how I got that recurrence? Is that pretty straightforward? If not, let me know. OK, and so, what's the solution to this recurrence? Yeah, order n^2. How do we know that? Yeah, master method, so n to the log base two of four, right, is n^2. Compare that with order one. This dramatically dominates. So this is the answer, the n to the log base two of four, n^2. OK, everybody remember that? OK, so I want people to bone up because this is going to be recurrences, and divide and conquer and stuff is going to be on the final, OK, even though we haven't seen it in many moons. OK, so that's good. That's the same as the serial. If I had to add 2N by n matrices, how long does it take me to do it? n^2 time.

OK, so the input is size n^2. So, you're not going to be the size of the input if you have to look at every piece of the input. OK, let's now do the work of the matrix multiplication. So once again, we want to get a recurrence here. So, what's our recurrence here? Yeah? Not quite. Eight, right, good. OK, eight, M1, n over 2, plus, yeah, there's theta n^2 for the addition, and then there's extra theta one that we can absorb into theta n^2.

Isn't asymptotics great? OK, it's just great. And so, what's the solution to that one? Theta n^3, why is that? Man, we are exercising old muscles. Aren't we? And they're just creaking. I can hear them. Why is that? Yeah, master method because we're looking at, what are we comparing? Yeah, n to the log base two of eight, or n^3 versus n^2, this one dominates order n^3. OK, so this is same as serial. This was the same as serial. This was the same as serial. That's good. OK, we know we have a program that on one processor, will execute the same as our serial code on which it's based.

Namely, we could have done this. If I had just got rid of all the spawns and syncs, that would have just been a perfectly good piece of pseudocode describing the runtime of the algorithm, describing the serial algorithm. And its run time ends up being exactly the same, not too surprising. OK? OK, so now we do the new stuff, critical path length. OK, so here we have A infinity of n. Ooh, OK, so we're going to add up the critical path of this code here.

Hmm, how do I figure out the critical path on a piece of code like that? So, it's going to expand into one of those DAG's. What's the DAG going to look like? How do I reason? So, it's actually easier not to think about the DAG, but to simply think about what's going on in the code. Yeah? Yeah, so it's basically, since all four spawns are spawning off the same thing, and they're operating in parallel, I could just look at one.

Or in general, if I spawned off several things, I look at which everyone is going to have the maximum critical path for the things that I've spawned off. So, when we do work, we're usually doing plus when I have multiple subroutines. When we do critical

path, I'm doing max. It's going to be the max over the critical paths of the subroutines that I call. OK, and here they are all equal. So what's the recurrence that I get?

What's the recurrence I'm going to get out of this one? Yeah, A infinity, n over 2, plus constant, OK, because this is what the worst is of any of those four because they're all the same. They're all a problem looking at the critical path of something that's half the size, for a problem that's half the size. OK, people with me? OK, so what's the solution to this? Yeah, that's theta log n. That's just, once again, master theorem, case two, because the solution to this is n to the log base two of one, which is n to the zero.

So we have, on this side, we have one, and here, we're comparing it with one. They're the same, so therefore we tack on that extra log n. OK, so tack on one log n. OK, so case two of the master method. Pretty good. OK, so that's pretty good, because the critical path is pretty short, log n compared to the work, n^2. So, let's do, then, this one which is a little bit more interesting, but not much harder. How about this one? What's the recurrence going to be? Critical path of the multiplication?

So once again, it's going to be the maximum of everything we spawned off in parallel, which is, by symmetry, the same as one of them. So what do I get here? M infinity, n over 2, plus theta log n. Where'd the theta log n come from? Yeah, from the addition. That's the critical path of the addition. Now, why is it that the maximum of that with all the spawns? You said that when you spawn things off, you're going to do them, yeah, you sync first. And, sync says you wait for all those to be done. So, you're only taking the maximum, and across syncs you're adding.

So, you add across syncs, and across things that you spawned off in parallel. That's where you are doing the max. OK, but if you have a sync, it says that that's the end. You've got to wait for everything there to end. This isn't going on in parallel with it. This is going on after it. So, whatever the critical path is here, OK, if I have an infinite number of processors, I'd still have to wait up at this point, and that would therefore make it so that the remaining execution gear was whatever the critical, I would have to add whatever the critical path was to this one.

Is that clear to everybody? OK, so we get this recurrence. And, that has solution what? Yeah, theta log squared n. OK, once again, by master method, case two, where this ends up being a constant versus log n, those don't differ by a polynomial amount, or equal to a log factor. What we do in that circumstance is tack on an extra log factor. OK, so as I say, good idea to review the master method. OK, that's great. So now, let's take a look at the parallelism that we get.

We'll just do it right here for the multiplication. OK, so parallelism is what for the multiplication? What's the formula for parallelism? So, we have p bar is the notation we use for this problem. What's the parallelism going to be? What's the ratio I take? Yeah, it's M_1 of n divided by M infinity of n. OK, and that's equal to, that's n^3. That's n^2, or log n^2, sorry, log squared n. OK, so this is the parallelism. That says you could run up to this many processors and expect to be getting linear speed up. If I ran with more processors than the parallelism, I don't expect to be getting linear speed up anymore, OK, because what I expect to run in, is just time proportional to critical path length, and throwing more processors at the problem is not going to help me very much, OK?

So let's just look at this just to get a sense of what's going on here. Let's imagine that the constants are irrelevant, and we have, say, thousand by thousand matrices, OK, so in that case, our parallelism is 1,000^3 divided by log of 1,000^2. What's log of 1,000? Ten, approximately, right? Log base two of 1,000 is about ten, so that's 10^2. So, you have about 10^7 parallelism. How big is 10^7? Ten million processors. OK, so who knows of a machine with ten million processors?

What's the most number of processors anybody knows about? Yeah, not quite, the IBM Blue Jean has a humungous number of processors, exceeding 10,000. Yeah. Those were one bit processors. OK, so this is actually a pretty big number, and so, our parallelism is much bigger than a typical, actual number of processors. So, we would expect to be able to run this and get very good performance, OK, because we're never going to be limited in this algorithm by performance.

However, there are some tricks we can do. One of the things in this code is that we actually have some overhead that's not apparent because I haven't run this code with you, although I could, which is that we have this temporary matrix, T, and if you look at the execution stack, we're always allocating T and getting rid of it, etc. And, one of the things when you actually look at the performance of real code, which is now that you have your algorithmic background, you're ready to go and do that with some insight. Of course, you're interested in getting more than just asymptotic behavior. You're interested in getting real performance behavior on real things. So, you do care about constants in that nature. OK, and one of the things is having a large, temporary variable. That turns out to be a lot of overhead.

And, in fact, it's often the case when you're looking at real code that if you can optimize for space, you also optimized for time. If you can run your code with smaller space, you can actually run it with smaller time, tends to be a constant factor advantage. But, those constants can add up, and can make a difference in whether somebody else's code is faster or your code is faster, once you have your basic algorithm. So, the idea is to, in this case, we're going to get rid of it by trading parallelism because we've got oodles of parallelism here for space efficiency.

OK, and the idea is we're going to get rid of T. OK, so let's throw this up. So, who can suggest how I might get rid of T here, get rid of this temporary matrix? Yeah? Yeah? So, if you just did adding it into C? So, the issue that you get there if they're both adding into C is you get interference between the two subcomputations. Now, there are ways of doing that that work out, but you now have to worry about things we're not going to talk about such as mutual exclusion to make sure that as you're updating it, somebody else isn't updating it, and you don't have race conditions.

But you can actually do it in this context with no race conditions. Yeah, exactly. Exactly, OK, exactly. So, the idea is spawn off four of them. OK, they all update their copy of C, and then spawn off the other four that add their values in. So, that is a piece of code we'll call mult add. And, it's actually going to do C gets C plus A times B. OK, so it's actually going to add it in. So, initially you'd have to zero out C, but we can do that with code very similar to the addition code with order n^2 work, and order log n critical path.

So that's not going to be a big part of what we have to deal with. OK, so here's the code. We basically, once again, do the base and partition which I'm not going to write out the code for. We spawn a mult add of C1-1, A1-1, B1-1, n over 2, and we

do a few more of those down to the fourth one. And then we put in a sync. And then we do the other four -- -- and then sync when we're done with that.

OK, does everybody understand that code? See that it basically does the same calculation. We actually don't need to call add anymore, because we are doing that as part of the multiply because we're adding it in. But we do have to initialize. OK, we do have to initialize the matrix in this case. OK, so there is another phase. So, people understand the semantics of this code. So let's analyze it. OK, so what's the work of multiply, add of n? It's basically the same thing, right?

It's order n^3 because the serial code is almost the same as the serial code up there, OK, not quite, OK, but you get essentially the same recurrence except you don't even have the add. You just get the same recurrence but with order one here, oops, order one up here. So, it's still got the order n^3 solution. OK, so that, I think, is not too hard. OK, so the critical path length, so there, let's write out, so multiply, add, of n, OK, what's my recurrence for this code?

Yeah, 2M infinity, M over 2 [ost that, so order one. Plus order one, yeah. OK, so the point is that we're going to have, for the critical path, we're going to spawn these four off, and so I take the maximum of whatever those is, which since they're symmetric is any one of them, OK, and then I have to wait. And then I do it again. So, that sync, once again, translates into, in the analysis, it translates into a plus of the critical path, which are the things I spawn off in parallel, I do the max. OK, so people see that? So, I get this recurrence, 2MA of n over 2 plus order one, and what's the solution to that?

OK, that's order n, OK, because n to the log base two of two is n, and that's bigger than one so we get order n. OK, so the parallelism, we have p bar is equal to MA one of n over MA infinity of n is equal to, in this case, n^3 over n, or order n^2. OK, so for 1,000 by 1,000 matrices, for example, by the way, 1,000 by 1,000 is considered a small matrix, these days, because that's only one million entries.

You can put that on your laptop no sweat. OK, so, but for 1,000 by 1,000 matrices, our parallelism is about 10^6. OK, so once again, ample parallelism for anything we would run it on today. And as it turns out, it's faster in practice -- -- because we have less space. OK, so here's a game where, so, often the game you'll see in theory papers if you look at research papers, people are often striving to get the most parallelism, and that's a good game to play, OK, but it's not necessarily the only game.

Particularly, if you have a lot of parallelism, one of the things that's very easy to do is to retreat on the parallelism and gain other aspects that you may want in your code. OK, and so this is a good example of that. In fact, and this is an exercise, you can actually achieve work n^3, order n^3 work, and a critical path of log n, so even better than either of these two algorithms in terms of parallelism.

OK, so that gives you n^3 over log n parallelism. So, that's an exercise. And then, the other exercise that I mention that that's good to do is parallel Strassen, OK, doing the same thing with Strassen, and analyze, what's the working critical path and parallelism of the Strassen code? OK, any questions about matrix multiplication? Yeah? Yeah, so that would take, that would add a log n to the critical path, which is nothing compared to the n. Excuse me? Well, you got to make sure C is zero to begin with. OK, so you have to set all the entries to zero, and so that will take you

n^2 work, which is nothing compared to the n^3 work you're doing here, and it will cost you log n additional to the critical path, which is nothing compared to the order n that you're spending.

Any other questions about matrix multiplication? OK, as they say, this all goes back to week two, or something, in the class. Did you have a comment? Yes, you can. OK, yes you can. It's actually kind of interesting to look at that. Actually, we'll talk later. We'll write a research paper after the class is over, OK, because there's actually some interesting open questions there. OK, let's move on to something that you thought you'd gotten rid of weeks ago, and that would be the topic of sorting.

Back to sorting. OK, so we want to parallel sort now, OK? Hugely important problem. So, let's take a look at, so if I think about algorithms for sorting that sound easy to parallelize, which ones sound kind of easy to parallelize? Quick sort, yeah, that's a good one. Yeah, quick sort is a pretty good one to parallelize and analyze. But remember, quick sort has a little bit more complicated analysis than some other sorts. What's another one that looks like it should be pretty easy to parallelize? Merge sort. When did we teach merge sort? Day one. OK, so do merge sort because it's just a little bit easier to analyze. OK, we could do the same thing for quick sort.

Here's merge sort, OK, and it's going to sort A of p to r. So, if p is less than r, then we get the middle element, and then we'll spawn off since we have to, as you recall, when you merge sort you first recursively sort the two sub-arrays. There's no reason not to do those parallel. Let's just do them in parallel. Let's spawn off, merge sort of A, p, q, and spawn off, then, merge sort of A, q plus one r.

And then, we wait for them to be done. Don't forget your syncs. Sync or swim. OK, and then what to do what we are done with this? OK, we merge. OK, so we merge of A, p, q, r, which is merge A of p up to q with A of q plus one up to r. And, once we've merged, we're done. OK, so this is the same code as we saw before in day one except we've got a couple of spawns in the sync. So let's analyze this. So, the work is called T_1 of n, what's the recurrence for this?

This really is going back to day one, right? We actually did this on day one. OK, so what's the recurrence? 2T1 of n over 2 plus order n merges order n time operation, OK? And so, that gives us a solution of n log n, OK, even if you didn't know the solution, you should know the answer, OK, which is the same as the serial code, not surprisingly. That's what we want. OK, critical path length, AT infinity of n is equal to, OK, T infinity of n over 2 plus order n again.

And that's equal to order n, OK? So, the parallelism is then p bar equals T_1 of n over T infinity of n is equal to theta of log n. Is that a lot of parallelism? No, we have a technical name for that. We call it puny. OK, that's puny parallelism. Log n? Now, so this is actually probably a decent algorithm for some of the small scale processors, especially the multicore processors that are coming on the market, and some of the smaller SMP, symmetric multiprocessors, that are available.

You know, they have four or eight processors or something. It might be OK. There's not a lot of parallelism. For a million elements, log n is about 20. OK, and so and then there's constant overheads, etc. This is not very much parallelism at all. Question? Yeah, so how can we do better? I mean, it's like, man, at merge, right, it takes order n. if I want to do better, what should I do? Yeah?

Sort in-place, but for example if you do quick sort and partition, you still have a linear time partition. So you're going to be very much in the same situation. But what can I do here? Parallel merge. OK, let's make merge go in parallel. That's where all the critical path is. Let's figure out a way of building a merge program that has a very short critical path. You have to parallelize the merge. This is great. It's so nice to see at the end of a course like this that people have the intuition that, oh, you can look at it and sort of see, where should you put in your work?

OK, the one thing about algorithms is it doesn't stop you from having to engineer a program when you code it. There's a lot more to coding a program well than just having the algorithm as we talked about, also, in day one. There's things like making it modular, and making it maintainable, and a whole bunch of things like that. But one of the things that algorithms does is it tells you, where should you focus your work?

OK, there's no point in, for example, sort of saying, OK, let me spawn off four of these things of size n over 4 in hopes of getting, I mean, it's like, that's not where you put the work. You put the work in merge because that's the one that's the bottleneck, OK? And, that's the nice thing about algorithms is it very quickly lets you hone in on where you should put your effort, OK, when you're doing algorithmic design in engineering practice. So you must parallelize the merge.

The merge we're taking, so here's the basic idea we're going to use. So, in general, when we merge, when we do our recursive merge, we're going to have two arrays. Let's call them A and B. I called them A there. I probably shouldn't have used A. I probably should have called them something else, but that's what my notes have, so we're going to stick to it. But we get a little bit more space there and see what's going on. We have two arrays. I'll call them A and B, OK? And, what we're going to do, these are going to be already sorted. And our job is going to be to merge them together. So, what I'll do is I'll take the middle element of A.

So this, let's say, goes from one to l, and this goes from one to m. OK, I'll take the middle element, the element at l over 2, say, and what I'll do is use binary search to figure out, where does it go in the array B? Where does this element go? It goes to some point here where we have j here and j plus one here. So, we know, since this is sorted, that all these things are less than or equal to A of l over 2, and all these things are greater than or equal to A of l over 2. And similarly, since that element falls here, all these are less than or equal to A of l over 2.

And all these are going to be less greater than or equal to two. OK, and so now what I can do is once I figured out where this goes, I can recursively merge this array with this one, and this one with this one, and then if I can just concatenate them altogether, I've got my merged array. OK, so let's write that code. Everybody get the gist of what's going on there, how we're going to parallelize the merge? Of course, you can see, it's going to get a little messy because j could be anywhere. Secures my code, parallel merge of, and we're going to put it in C of one to n, so I'm going to have n elements.

So, this is doing merge A and B into C, and n is equal to l plus n. OK, so we're going to take two arrays and merge it into the third array, OK? So, without loss of generality, I'm going to say, let's see, without loss of generality, I'm going to say l is bigger than m as I show here because if it's not, what do I do? Just do it the other

way around, right? So, I figure out which one was bigger. So that only cost me order one to test that, or whatever.

And then, I basically do a base case, you know, if the two arrays are empty or whatever, what you do in practice, of course, is if they're small enough, you just do a serial merge, OK, if they're small enough, and I don't really expect to get much parallelism. There isn't much work there. You might as well just do serial merge, and be a little bit more efficient, OK? So, do the base case. So then, what I do is I find the j such that B of j is less than or equal to A of l over 2, less than or equal to B of j plus one, using binary search. What did recover binary search? Oh yeah, that was week one, right?

That was first recitation or something. Yeah, it's amazing. OK, and now, what we do is we spawn off p merge of A of one, l over 2, B of one to j, and stick it into C of one, two, l over 2 plus j. OK, and similarly now, we can spawn off a merge of A of l over 2 plus one up to l. B of j plus one up to M, and a C of l over two plus j plus one up to n. And then, I sync. So, code is pretty straightforward, doing exactly what I said we were going to do over here, analysis, a little messier, a little messier. So, let's just try to understand us before we do the analysis. Why is it that I want to pick the middle of the big array rather than the small array?

What sort of my rationale there? That's actually a key part, going to be a key part of the analysis. Yeah? OK. Yeah, imagine that B, for example, had only one element in it, OK, or just a few elements, then finding it in A might mean finding it right near the beginning of A. And now, I'd be left with subproblems that were very big, whereas here, as you're pointing out, if I start here, if my total number of elements is n, what's the smallest that one of these recursions could be?

n over 4 is the smallest it could be, OK, because I would have at least a quarter of the total number of elements to the left here or to the right here. If I do it the other way around, my recursion, I might get a recursion that was nearly as big as n, and that's sort of, once again, sort of like the difference when we were analyzing quick sort with whether we got a good partitioning element or not.

The partitioning element is somewhere in the middle, we're really good, but it's always at one end, it's no better than insertion sort. You want to cut off at least a constant fraction in your divided and conquered in order to get the logarithmic behavior. OK, so we'll see that in the analysis. But the key thing here is that what we are going to do the recursion, we're going to have at least n over 4 elements in whatever the smaller thing is. OK, but let's start. It turns out the work is the hard part of this. Let's start with critical path length. OK, look at that, critical path length.

OK, so parallel merge, so infinity of n is going to be, at most, so if the smaller piece has at least a quarter, what's the larger piece going to be of these two things here? So, we have two problems responding off. Now, we really have to do max because they're not symmetric. Which one's going to be worse? One could have, at most, three quarters, OK, of n. Woops, 3n, of 3n over 4 plus, OK, so the worst of those two is going to be three quarters of the elements plus, what?

Plus log n. What's the log n? The binary search. OK, and that gives me a solution of, this ends up being n to the, what? n to the zero, right. OK, it's n to the log base four thirds of one. OK, it was the log of anything of one is zero. So, it's n to the zero. So

that's just one compared with log n, tack on this log squared n. So, we have a critical path of log squared n. That's good news.

Now, let's hope that we didn't blow up the work by a substantial amount. OK, so the work is PM_1 of n is equal to, OK, so we don't know what the split is. So let's call it alpha. OK, so alpha n in one side, and then the work on the other side will be PM of one of one minus alpha n plus, and then still order of log n to the binary search where, as we've said, alpha is going to fall between one quarter and three quarters.

OK, how do we solve a recurrence like this? What's the technical name for this kind of recurrence? Hairy. It's a hairy recurrence. How do we solve hairy recurrences? Substitution. OK, good. Substitution. OK, so we're going to say PM one of k is less than or equal to, OK, I want to make a good guess here, OK, because I've fooled around with it. I want it to be linear, so it's going to have a linear term, a times k minus, and then I'm going to do b log k. So, this is this trick of subtracting a low order term. Remember that in substitution in order to make it stronger? If I just did ak it's not going to work because here I would get n, and then when I did this substitution I'm going to get a alpha n, and then a one minus alpha n, and those two together are already going to add up to everything here.

So, there's no way I'm going to get it down when I add this term in. So, I need to subtract something from both of these so as to absorb this term, OK? So, I'm skipping over those steps, OK, because we did those steps in lecture two or something. OK, so that's the thing I'm going to guess where a and b are greater than zero. OK, so let's do the substitution. OK, so we have PM one of n is less than or equal to, OK, we substitute this inductive hypothesis in for these two guys.

So, we get a alpha n minus b log of alpha n plus a of one minus alpha n minus b log of one minus alpha, maybe another parentheses there, one minus alpha n, and even leave myself enough space here plus a of one minus alpha n minus b log of one minus alpha, maybe another parenthesis there, one minus alpha n. I didn't even leave myself enough space here. Plus, let me just move this over so I don't end up using too much space.

So, b log of one minus alpha n plus theta of log n. How's that? Are we OK on that? OK, so that's just substitution. Let's do a little algebra. That's equal to a of times alpha na times one minus alpha n. That's just an, OK, minus, well, the b isn't quite so simple. OK, so I have a b term. Now I've got a whole bunch of stuff there. I've got log of alpha n. I have, then, this log of one minus alpha n, OK, I'll start with the n, and then plus theta log n.

Did I do that right? Does that look OK? OK, so look at that. OK, so now let's just multiply some of this stuff out. So, I have an minus b times, well, log of alpha n is just log alpha plus log n. And then I have plus log of one minus alpha plus log n, OK, plus theta log n. That's just more algebra, OK, using our rules for logs. Now let me express this as my solution minus my desired solution minus a residual, an minus b log n, OK, minus, OK, and so that was one of these b log n's, right, is here. And the other one's going to end up in here. I have B times log n plus log of alpha times one minus alpha minus, oops, I've got too many.

Do I have the right number of closes. Close that, close that, that's good, minus theta log n. Two there. Boy, my writing is degrading. OK, did I do that right? Do I have the parentheses right? That matches, that matches, that matches, good. And then B

goes to that, OK, good. OK, and I claim that is less than or equal to AN minus B log n if we choose B large enough. OK, this mess dominates this because this is basically a log n here, and this is essentially a constant. OK, so if I increase B, OK, times log n, I can overcome that log n,

whatever the constant is, hidden by the asymptotic notation, OK, such that B times log n plus log of alpha times one minus alpha dominates the theta log n. OK, and I can also choose my base condition to be big enough to handle the initial conditions, whatever they might be. OK, so we'll choose A big enough -- -- to satisfy the base of the induction. OK, so thus PM_1 of n is equal to theta n, OK?

So I actually showed O, and it turns out, the lower bound that it is omega n is more straightforward because the recurrence is easier because I can do the same substitution. I just don't have to subtract off low order terms. OK, so it's actually theta, not just O. OK, so that gives us a log, what did we say the critical path was? The critical path is log squared n for the parallel merge. So, let's do the analysis of merge sort using this. So, the work is, as we know already, T_1 of n is theta of n log n because our work that we just analyzed was order n, same as for the serial algorithm, OK?

The critical path length, now, is T infinity of n is equal to, OK, so in normal merge sort, we have a problem of half the size, T of n over 2 plus, now, my critical path for merging is not order n as it was before. Instead, it's just over there. Log squared n, there we go. OK, and so that gives us theta of log cubed n. So, our parallelism is then theta of n over log cubed n. And, in fact, the best that's been done is, sorry, log squared n, you're right. Log squared n because it's n log n over log cubed n.

It's n over log squared n, OK? And the best, so now I wonder if I have a typo here. I have that the best is, p bar is theta of n over log n. Is that right? I think so. Yeah, that's the best to date. That's the best to date. By Occoli, I believe, is who did this. OK, so you can actually get a fairly good, but it turns out sorting is a really tough problem to parallelize to get really good constants where you want to make it so it's running exactly the same. Matrix multiplication, you can make it run in parallel and get straight, hard rail, linear speed up with a number of processors.

There is plenty of parallelism, and running on more processors, every processor carries a full weight. With sorting, typically you lose, I don't know, 20% in my experience, OK, in terms of other stuff going on because you have to work really hard to get the constants of this merge algorithm down to the constants of that normal merge, right? I mean that's a pretty good algorithm, right, the one that just goes, BUZZING SOUND, and just takes two lists and merges them like that. So, it's an interesting issue. And a lot of people work very hard on sorting, because it's a hugely important problem, and how it is that you can actually get the constants down while still guaranteeing that it will scale up with a number of processors. OK, that's our little sojourn into parallel land, and next week we're going to talk about caching, which is another very important area of algorithms, and of programming in general.