JULIAN SHUN: All right. So today we're going to talk about cache oblivious algorithms. Who remembers what a cache oblivious algorithm is? So what is a cache oblivious algorithm oblivious to? Cache size.

So a cache oblivious algorithm is an algorithm that automatically tunes for the cache size on your machine. So it achieves good cache efficiency. And the code doesn't need to have any knowledge of the cache parameters of your machine. In contrast, a cache-aware algorithm would actually know the parameters of the cache sizes on your machine. And the code would actually put the size of the cache inside.

So today we're going to talk a lot more about cache oblivious algorithms. Last time we talked about one cache oblivious algorithm that was for matrix multiplication. And today we're going to talk about some other ones. So first example I want to talk about is simulation of heat diffusion. So here's a famous equation known as the heat equation.

And this equation is in two dimensions. And we want to simulate this function u that has three parameters t, x, and y. t is the time step. X and y are the x, y coordinates of the 2D space. And we want to know the temperature for each x, y coordinate for any point in time t.

And the 2D heat equation can be modeled using a differential equation. So how many of you have seen differential equations before? OK. So good, most of you. So here I'm showing the equation in two dimensions. But you can get similarly equations for higher dimensions.

So here it says the partial derivative of u with respect to t is equal to alpha. Alpha is what's called the thermo diffusivity. It's a constant times the sum of the second partial derivative of u with respect to x, and the second partial derivative of u with respect to y.

So this is a pretty famous equation. And you can see that we have a single derivative on the left side and a double derivative on the right side. And how do we actually write code to simulate this 2D heat process? So oftentimes in scientific computing, people will come up with

these differential equations just to describe physical processes. And then they want to come up with efficient code to actually simulate the physical process.

OK. So here's an example of a 2D heat diffusion. So let's say we started out with a configuration on the left. And here the color corresponds to a temperature. So a brighter color means it's hotter. Yellow is the hottest and blue is the coldest.

In on the left we just have 6172, which is the course number. So if you didn't know that, you're probably in the wrong class. And then afterwards we're going to run it for a couple time steps. And then the heat is going to diffuse to the neighboring regions of the 2D space. So after you run it for a couple of steps, you might get the configuration on the right where the heat is more spread out now.

And oftentimes, you want to run this simulation for a number of time steps until the distribution of heat converges so it becomes stable and that doesn't change by much anymore. And then we stop the simulation. So this is the 1D heat equation. I showed you a 2D one earlier.

But we're actually going to generate code for the 1D heat equation since it's simpler. But all the ideas generalize to higher dimensions. And here's the range of colors corresponding to temperature, so the hottest colors on the left and the coldest colors on the right. And if you had a heat source that's on the left hand side of this bar, then this might possibly be a stable distribution. So if you keep running the simulation, you might get a stable distribution of heat that looks like this.

OK. So how do we actually write code to simulate this differential equation? So one commonly used method is known as finite difference approximation. So we're going to approximate the partial derivative of u with respect to each of its coordinates. So the partial derivative of u with respect to t is approximately equal to u of t plus delta t where delta t is some small value, and x, and then minus u of tx. And then that's all divided by delta t.

So how many of you have seen this approximation method before from your calculus class? OK, good. So as you bring the value of delta t down to 0, then the thing on the right hand side approach is the true partial derivative. So that's a partial derivative with respect to t. We also need to get the partial derivative with respect to x.

And here I'm saying the partial derivative with respect to x is approximately equal to ut of x plus delta x over 2 minus ut of x minus delta x over 2, all divided by delta x. So notice here that

instead of adding delta x in the first term and not adding anything in the second term, I'm actually adding delta x over 2 in the first term and subtracting text over 2 in the second term. And it turns out that I can do this with the approximation method.

And it still turns out to be valid as long as the two terms that I'm putting in, their difference is delta x. So here the difference is delta x. And I can basically decide how to split up this delta x term among the two things in the numerator.

And the reason why I chose delta x over 2 here is because the math is just going to work out nicely. And it's going to give us cleaner code. This is just the first partial derivative with respect to x. Actually need the second partial derivative since the right hand side of this equation has the second partial derivative.

So this is what the second partial derivative looks like. So I just take the partial derivative with respect to x of each of the terms in my numerator from the equation above. And then now I can actually plug in the value of this partial derivative by applying the equation above using the arguments t and x plus delta x over 2, and similarly for the second term.

So for the first term when I plug it into the equation for the partial derivative with respect to x, I'm just going to get ut of x plus delta x minus utx. And then for the second term, I'm going to get ut of x minus delta x. And then I subtract another factor of utx.

So that's why I'm subtracting 2 times utx in the numerator here. And then the partial derivative of each of the things in a numerator also have to divide by this delta x term. So on the denominator, I get delta x squared.

So now I have the second partial derivative with respect to x. And I also have the first partial derivative with respect to t. So I can just plug them into my equation above.

So on the left hand side I just have this term here. And I'm multiplying by this alpha constant. And then on this term just comes from here. So this is what the 1d heat equation reduces to using the finite difference approximation method. So any questions on this

So how do we actually write code to simulate this equation here? So we're going to use what's called a stencil computation. And here I'm going to set delta at x and delta t equal to 1, just for simplicity. But in general you can set them to whatever you want. You can make them smaller to have a more fine grained simulation.

So my set delta x and delta eat t equal to 1. Then the denominators of these two terms just become one and I don't need to worry about them. And then I'm going to represent my 2D space using a 2D matrix where the horizontal axis represents values of x, and the vertical axis represents values of t. And I want to fill in all these entries that have a black dot in it. The ones with the orange dot, those are my boundaries.

So those actually fixed throughout the computation. So I'm not going to do any computation on those. Those are just given to me as input. So they could be heat sources if we're doing the heat simulation.

And then now I can actually write code to simulate this equation. So if I want to compute u of t plus 1x, I can just go up here. And I see that that's equal to this thing over here. And then I bring the negative utx term to the right.

So I get ut of x plus alpha times ut x plus 1 minus 2 times utx plus ut x minus 1. As I said before, we just want to keep iterating this until the temperatures becomes stable. So I'm going to proceed in time, which in time is going up here.

And to compute one of these points-- so let's say this is ut plus 1x-- I need to know the value of utx, which is just a thing below me in the matrix. And I also need to know utx plus 1 and utx minus 1. And those are just the things below me and diagonal to either the left or the right side.

So each value here just depends on three other values. And this is called a three point stencil. This is the pattern that this equation is representing. And in general, a stencil computation is going to update each point in an array using a fixed pattern. This is called a stencil.

So I'm going to do the same thing for all of the other points. And here, I'm going to compute all the values of x for a given time step. And then I move on to the next time step.

And then I keep doing this until the distribution of temperatures becomes stable. And then I'm done. OK. So these stencil computations are widely used in scientific computing.

They're used for weather simulations, stock market simulations, fluid dynamics, image processing probability, and so on. So they're used all over the place in science. So this is a very important concept to know.

So let's say I just ran the code as I showed you in the animation. So I completed one row at a

time before I moved on to the next row. How would this code perform with respect to caching? Yes?

AUDIENCE: I think if x is less than a third of the cache size, [INAUDIBLE]

JULIAN SHUN: Yeah. So if x is small, this would do pretty well. But what if x is much larger than the size of your cache?

AUDIENCE: [INAUDIBLE].

JULIAN SHUN: Yeah, you. Would do badly and why is that?

AUDIENCE: Because the whole [INAUDIBLE] second row, [INAUDIBLE]

JULIAN SHUN: Yeah. So it turns out that there could be some reuse here, because when I compute the second row, I'm actually using some values that are computed in the first row. But if the row is much larger than the cache size, by the time I get to the second row, the values that I loaded into cache from the first row would have been evicted. And therefore, I'm going to suffer a cache miss again for the second row, for the values I need to load in, even though they could have been used if we made our code have good locality.

Another question I have is if we only cared about the values of x at the most recent time step, do we actually have to keep around this whole 2D matrix? Or can we get by with less storage? Yeah. So how many rows would I have to keep around if I only cared about the most recent time step? Yeah?

AUDIENCE: Two.

JULIAN SHUN: Two. And why is that?

AUDIENCE: So one for the previous time step. One for the current time step. [INAUDIBLE]

JULIAN SHUN: Right. So I need to keep around two rows because I need the row from the previous time step in order to compute the values in the current time step. And after the current time step, I can just swap the roles of the two rows that I'm keeping around, and then reuse the previous row for the next one. Yes?

AUDIENCE: Would you only need one and then a constant amount of extra space, like if you had three extra things, you could probably do it with one.

JULIAN SHUN: So I need to know-- when I'm computing the second row, I need to keep around all of these values that I computed in the first row, because these values get fed to one of the computations in the second row. So I need to actually keep all of them around.

AUDIENCE: I think if you iterate to the right, then you have three that are this one and the next one. Just three. Then you can--

JULIAN SHUN: Oh, I see I see what you're saying. Yeah. So that's actually a good observation. So you only need to keep a constant amount more storage, because you'll just be overwriting the values as you go through the row. So if you keep around one row, some of the values would be for the current time step, and some of them would be from the previous time step. So that's a good observation, yes.

OK. So that code, as we saw, it wasn't very cache efficient. You could make a cache efficient using tiling. But we're going to go straight to the cache oblivious algorithm because it's much cleaner. So let's recall the ideal cache model. We talked about this in the previous lecture.

So here we have a two level hierarchy. We have a cache. And then we have the main memory. The cache has size of M bytes. And a cache line is B bytes.

So you can keep around M over B cache lines in your cache. And if something's in cache and you operate on it, then it doesn't incur any cache misses. But if you have to go to main memory to load the cache line in, then you incur one cache miss.

The ideal cache model assumes that the cache is fully associative. So any cache line can go anywhere in the cache. And it also assumes either an optimal omniscient replacement policy or the LRU policy.

So the optimal omniscient replacement policy knows the sequence of all future requests to memory. And when it needs to evict something, it's going to pick the thing that leads to the fewest cache misses overall to evict. The LRU policy just evict the thing that was least recently used.

But we saw from the previous lecture, that in terms of asymptotic costs, these two replacement policies will give you cache misses within a constant fact of each other. So you can use either one, depending on what's convenient.

And two performance measures that we care about when we're analyzing an algorithm and

the ideal cache model are the work and the number of cache misses. So the work is just the total number of operations that the algorithm incurs. And serially, this is just the ordinary running time. And the number of cache misses is the number of cache lines you have to transfer between the cache and your main memory.

So let's assume that we're running an algorithm or analyzing an algorithm in the ideal cache model, and it runs serially. What kinds of cache misses does the ideal cache model not capture? So remember, we talked about several types of cache misses. And there's one type of cache miss that this model doesn't capture when we're running serially.

So let's assume we're running this serially without any parallelism here. So the sharing misses has only come about when you have parallelism. Yes?

AUDIENCE: Conflictness?

JULIAN SHUN: Yes. So the answer is conflictnesses. And why is that? Why does this model not capture it?

AUDIENCE: There's not a specific sets that could get replaced then since it's fully associated.

JULIAN SHUN: Yes. So this is a fully associative cache. So any cache line can go anywhere in the cache. And you can only get conflict misses for set associated schemes where each cache line can only be mapped to a particular set.

And if you have too many cache lines that map to that particular set, then you're going to keep evicting each other even though the rest of the cache could have space. And that's what's called a conflict miss. The ideal cash model does capture capacity misses.

So therefore, it is still a very good model to use at a high level when you're designing efficient algorithms, because it encourages you to optimize for spatial and temporal locality. And once you have a good algorithm in the ideal cache model then you can start dealing with conflict misses using some of the strategies that we talked about last time such as padding or using temporary memory. So any questions on this?

OK. So this is the code that does the heat simulation that we saw earlier. So it's just two for loops, a nested for loop. In the outer loop, we're looping over the time dimension. In the inner loop, we're looping over the space dimension. So we're computing all the values of x before we move on to the next time step.

And then we're storing two rows here and we're using this trick called a even odd trick. And here's how it works. So to access the next row that we want to compute, that we just do a t plus 1 mod 2. And then to access the current row, it's just t mod 2. So this is implicitly going to swap the roles of the two rows that we're keeping around as we progress through time.

And then we're going to set u of t plus 1 mod 2x equal to kernel of u-- pointer to ut mod 2x. And this kernel function is defined up here. And recall, that when we're actually passing a pointer to this kernel function, we can actually treat a pointer as the beginning of an array. So we're using array notation up here inside the kernel function. So the array W is passed as input.

And then we need to return W of 0. That's just the element at the current pointer that we passed to kernel. And then we add alpha times w of negative 1. That's one element before the thing that we're pointing to, minus 2 times W of 0 plus W of 1. W of 1 is the next element that we're pointing to.

OK. So let's look at the caching behavior of this code. So we're going to analyze the cache complexity. And we're going to assume the LRU replacement policy here, because we can.

And as we said before, we're going to loop through one entire row at a time before we go onto the next row. So the number of cache misses I get, assuming that n is greater than M, so that the row size is greater than the cache size, the number of cache misses is theta of NT over B. So how do I get this cache complexity around here?

So how many cache misses do I have to incur for each row of this 2D space that I'm computing? Yes?

AUDIENCE: N over B.

JULIAN SHUN: Right. So I need N over B cache misses for each row. And this is because I can load in B bytes at a time. So I benefit from spatial locality there. And then I have N elements I need to compute. So it's theta of N over B per row.

And as we said before, when we get to the next row, the stuff that we need from the previous row have already been evicted from cache. So I basically have to incur theta of N over B cache misses for every row. And the number of rows I'm going to compute as t. So it's just theta of NT over B. Any questions on this analysis?

So how many of you think we can do better than this? OK. So one person. Two, three. OK. So turns out that we can do better than this. You can actually do better with tiling, but I'm not going to do the tiling version.

I want to do the cache oblivious version. And the cache oblivious version is going to work on trapezoidal regions in the 2D space. And recall that a trapezoid has a top base and a bottom base. And here the top base is at t1, the bottom base is at t0, and the height is just t1 minus t0. And the width of a trapezoid is just the width of it at the midpoint between t1 and t0, so at t1 plus t0 over 2.

So we're going to compute all of the points inside this trapezoid that satisfy these inequalities here. So t has to be greater than or equal to t0 and less than t1. And then x is greater than or equal to x0 plus dx0 times t minus t0. So dx0 is actually the inverse slope here. And then it also has to be less than x1 plus dx1 times t minus t0.

So dx1 is the inverse slope on the other side. And dx0 and dx1 have to be either negative 1, 0, or 1. So negative 1 just corresponds to inverse slope of negative 1, which is also a slope of negative 1. If it's 1, then it's just a slope or inverse of 1. And then if it's 0, then we just have a vertical line.

OK. So the nice property of this trapezoid is that we can actually compute everything inside the trapezoid without looking outside the trapezoid. So we can compute everything here independently of any other trapezoids we might be generating. And we're going to come up with a divide and conquer approach to execute this code. So the divide and conquer algorithm has a base case. So our base case is going to be when the height of the trapezoid is 1.

And when the height is 1, then we're just going to compute all of the values using a simple loop. And any order if the computation inside this loop is valid, since we have all the values in the base of the trapezoid and we can compute the values in the top of the trapezoid in whatever order. They don't depend on each other. So that's a base case. Any questions so far?

So here's one of the recursive cases. It turns out that we're going to have two different types of cuts. The first cut is called a space cut. So I'm going to do a space cut if the width of the trapezoid is greater than or equal to twice the height.

So this means that the trapezoid is too wide. And I'm going to cut it vertically. More specifically,

I'm going to cut it with a line, with slope negative 1 going through the center of the trapezoid. And then I'm going to traverse the trapezoid on the left side first. And then after I'm done with that, traverse the trapezoid on the right side.

So can I actually switch the order of this? Can I compute the stuff on the right side before I do this stuff on the left side? No. Why is that?

AUDIENCE: [INAUDIBLE].

JULIAN SHUN: Yeah. So there some points in the right trapezoid that depend on the values from the left trapezoid. And so for the left trapezoid, every point we want to compute, we already have all of its points, assuming that we get all the values of the base points. But for the right hand side, some of the values depend on values in the left trapezoid.

So we can't execute the right trapezoid until we're done with the left trapezoid. And this is the reason why I cut this trapezoid with a slope of negative 1 instead of using a vertical cut. Because if I did a vertical cut then inside both of the trapezoids, I would have points that depend on the other trapezoid.

So this is one of the two cuts. This is called a space cut. And it happens when the trapezoid is too wide. The other cut is the time cut I'm going to cut with respect to the time dimension. And this happens when the trapezoid is too tall, so when the width is less than twice the height of the trapezoid.

Then what I'm going to do is I'm just going to cut it with a horizontal line through the center. And then I'm going to traverse the bottom trapezoid first. And after I'm done with that, I can traverse a top trapezoid. And again, the top trapezoid depends on some points from the bottom trapezoid. So it's I can't switch the order of those. Any questions?

OK. So let's now look at the code that implements this recursive divide and conquer algorithm. So here's the C code. It takes as input t0 and t1. These are the coordinates of the top and the bottom up the trapezoid, or bottom and top of the trapezoid, then x. 0 is the left side of the trapezoid-- of the base of the trapezoid.

dx0 is the inverse slope, and the x1 is the right side of the bottom base of the trapezoid, and dx1 is the inverse slope on the right side. So we're first going to compute the height of our trapezoid. And we're going to let LT be the height. And that's just t1 minus t0. And if the height is 1, then we're just going to use a for loop over all the points in that height 1 trapezoid.

We're going to call this kernel function that we defined before. And otherwise, the height is greater than 1. And we're going to check whether we should do a space cut or time cut. So we do a space cut if the trapezoid is too wide. And this condition inside the if clause is checking if the trapezoid is too wide.

And if so, then we're going to make two recursive calls to trapezoid. And we're going to cut it in the middle using this slope of negative 1. So you see the negative ones here in the recursive calls. And otherwise, we'll do a time cut.

And for the time cut we just cut it in the middle. So we cut it at the value of t that's equal to LT divided by, 2 or t0 plus LT divided by 2. And again, we have two recursive calls to trapezoid.

OK. So even though I'm only generating slopes of negative 1 in this recursive call, this code is going to work even if I have slopes of 1 and 0, because I could start out with slopes of 1 and 0. For example, if I had a rectangular region, then the slopes are just going to be 0, and this code is still going to work. But most of the slopes that I'm dealing with are going to be a negative 1, because those are the new slopes and I'm generating. Any questions?

So this code is very concise. It turns out that even, odd tricks still works here. You can still keep around just two rows, because you're guaranteed that you're not going to overwrite any value until all the things that depend on it are computed. But when you're using just two rows, the values in a particular row might not all correspond to the same time step, because we're not finishing an entire row before we move on to the next one. We're actually partially computing rows.

OK. So let's analyze the cash complexity of this algorithm. Again, we're going to use the recursion tree approach that we talked about last time. And our code is going to split itself into two cell problems at every level until it gets to a leaf. And each leaf represents theta of hw points where h is theta of w. So h is the height. w is the width.

And we have the property that h is equal to theta w because of the nature of the algorithm. When the trapezoid becomes too wide, we're going to make it less wide by doing a space cut. And if it becomes too tall, we're going to do a horizontal cut. So we're guaranteed that the height and the width are going to be within a constant factor of each other when we get to the base case. And each leaf in the base case is going to incur theta of w over B misses because we have to load in the base of the trapezoid from main memory.

And once that's in cache, we can compute all of the other points in the trapezoid without incurring any more cache misses. So the cache misses per leaf is just theta w over B. And we're going to set w equal to theta of M in the analysis, because that's the point when the trapezoid fits into cache.

The algorithm doesn't actually have any knowledge of this M parameter. So it's still going to keep divide and conquering until it gets the base case of size 1. But just for the analysis, we're going to use a base case when w is theta of M.

So the number of leaves we have is theta of NT divided by w because each leaf is a size theta hw. The number of internal notes we have is equal to a number of leaves minus because we have a tree here. But the internal notes don't contribute substantially to the cache complexity, because each of them is just doing a constant number of cache misses to set up the two recursive calls. And it's not doing anything more expensive than that.

So we just need to compute the number of cache misses at the leaves. We have theta of NT over hw leaves, each one of which takes theta of w over B cache misses. And we're going to multiply that out.

So the w term cancels out. We just have NT over hB and we set h and w to be theta of M. So we're just left with NT over MB as our cache complexity. Yes?

AUDIENCE: Can you explain why hB [INAUDIBLE]?

JULIAN SHUN: Sure. So each leaf only incurs theta w over B cache misses because we need to compute the values of the base of the trapezoid. And that's going to incur theta of w over B cache misses because it's w wide, and therefore, everything else is going to fit into cache. So when we compute them, we already have all of our previous values that we want in cache. So that's why it's not going to incur any more cache misses. So does that makes sense?

AUDIENCE: Yeah, I forgot that it was [INAUDIBLE].

JULIAN SHUN: Yeah.

OK. So this was just analysis for one dimension. But it actually generalizes to more than one dimension. So in general, if we have d dimensions, then the cache complexity is going to be theta of NT divided by M to the 1 over d times B.

So if d is 1, then that just reduces to NT over MB. If d is 2, then it's going to be NT over B times square root of M and so on. And it turns out that this bound is also optimal.

So any questions? So compared to the looping code, this code actually has much better cache complexity. It's saving by a factor of M. The looping code had a cache complexity that was theta of NT over B. It didn't have an M in the denominator. OK.

So we're actually going to do a simulation now. We're going to compare how the looping code in a trapezoid code runs. And in this simulation, the green points correspond to a cache hit, the purple points correspond to a cache miss. And we're going assume a fully associative cache using the LRU replacement policy where the cache line size is 4 points and cache size is 32 points. And we're going to set the cache hit latency to be one cycle, and the cache miss latency to be 10 cycles.

So an order of magnitude slower for cache misses. And we're doing this for a rectangular region where N is 95. And we're doing it for 87 time steps. So when we pull out the simulation now.

OK. So on the left hand side, we're going to have the looping code. On the right hand side, we're going to have the trapezoidal code. So let's start this. So you can see that the looping code is going over one row at a time whereas the trapezoidal code is not doing that. It's partially computing one row and then moving on to the next row.

I can also show the cuts that are generated by the trapezoidal algorithm. I have to remember how to do this. So C--

So there there's the cuts that are generated by the trapezoidal algorithm. And I can speed this up. So you can see that the trapezoidal algorithm is incurring much fewer cache misses than the looping code.

So I just make this go faster. And the trapezoid code is going to finish, while the looping code is still slowly making its way up the top. OK. So it's finally done. So any questions on this? Yeah?

AUDIENCE: Why doesn't the trapezoid [INAUDIBLE]?

JULIAN SHUN: In which of the regions? So it's loading-- you get a cache miss for the purple dots here. And then the cache line size is 4.

So you get a cache miss for the first point, and then you hit on the next three points. Then you get another cache miss for the fifth point. And then you hit on the 6, 7, and 8 points.

AUDIENCE: I was indicating the one above it.

JULIAN SHUN: So for the one above it-- so we're assuming that the two arrays fitting cache already. So we don't actually need to load them from memory. The thing above it just depends on the values that we have already computed.

And that fits in cache. Those are ready in cache. This base of the trapezoid is already in cache. And the row right above it, we just need to look at those values. Does that makes sense?

AUDIENCE: OK. Because of the even odd?

JULIAN SHUN: Yeah. Yeah. Any other questions on this? Does anyone want to see this again? Yeah?

So I could let this run for the rest of the lecture, but I have more interesting material that I want to talk about. So let's just stop after this finishes. And as you see again, the looping code is slowly making its way to the top. It's much slower than the trapezoid code.

OK. So that was only for one dimensions. Now let's look at what happens in two dimensions. And here, I'm going to show another demo. And this demo, I'm actually going to run the code for real. The previous demo was just a simulation.

So this is going to happen in real time. And I'm going to simulate the heat in a 2D space. And recall that the colors correspond to the temperature. So a brighter color means it's hotter. So let me pull out the other demo.

OK. So here, my mouse cursor is the source of heat. So you see that it's making the points around my mouse cursor hot. And then it's slowly diffusing its way to the other points. Now I can actually move this so that my heat source changes, and then the heat I generated earlier slowly goes away. OK.

So in the lower left hand corner, I'm showing the number of iterations per second of the code. And we can see that the looping code is taking-- it's doing about 1,560 iterations per second. Let's see what happens when we switch to the trapezoid code. So the trapezoid code is doing about 1,830 iterations per second. So it's a little bit faster, but not by too much.

Does anyone have an idea why we're seeing this behavior? So we said that the trapezoid code incurs many fewer cache misses than the looping code, so we would expect it to be significantly faster. But here it's only a little bit faster. Yeah?

AUDIENCE: [INAUDIBLE].

JULIAN SHUN: Right. So that's a good point. So in 2D you're only saving a factor of square root of M instead of M. But square root of M is still pretty big compared to the speed up we're getting here. So any other ideas? Yeah.

So there is a constant factor in the trapezoidal code. But even after accounting for the constant factor, you should still see a better speed up than this. So even accounting for the constant factors, what other problem might be going on here? Yeah?

AUDIENCE: Is it that we don't actually have an ideal cache?

JULIAN SHUN: Yeah. So that's another good observation. But the caches that we're using, they still should get pretty good cache. I mean, they should still have cache complexly that's pretty close to the ideal cache model. I mean, you might be off by small constant factor, but not by too much. Yeah?

AUDIENCE: Maybe because [INAUDIBLE]

JULIAN SHUN: Sorry. Could you repeat that?

AUDIENCE: There are [INAUDIBLE] this time like [INAUDIBLE]

JULIAN SHUN: Yeah. So OK. So if I move the cursor, it's probably going to be a little bit slower, go slower by a little bit. But that doesn't really affect the performance. I can just leave my cursor there and this is what the iterations per second is. Yes?

AUDIENCE: Maybe there's like, a lot of similar programs doing this [INAUDIBLE].

JULIAN SHUN: Yeah. So there is some other factor dominating. Does anyone have an idea what that factor might be?

AUDIENCE: Rendering?

JULIAN SHUN: No. It's not the rendering. I ran the code without showing the graphics, and performance was similar. Yes?

AUDIENCE: Maybe similar to what she said there could be other things using cache [INAUDIBLE].

JULIAN SHUN: Yes. Yeah. So there could be other things using the cache. But that would be true for both of the programs. And I don't actually have anything that's intensive running, except for PowerPoint. I don't think that uses much of my cache. All right.

So let's look at why this is the case. So it turns out that the hardware is actually helping the looping code. So the question is how come the cache oblivious trapezoidal code can have so many fewer cache misses, but the advantage gained over the looping version is so marginal? Turns out that for the looping code, the hardware is actually helping it by doing hardware prefetching.

And hardware prefetching for the looping code is actually pretty good, because the access pattern is very regular. It's just going one row at a time. So the hardware can predict the memory access pattern of the looping code, and therefore, he can bring in the cache lines that the looping code would need before it actually gets to that part of the computation. So prefetching is helping the looping code. And it's not helping the trapezoid code that much because the access pattern is less regular there.

And prefetching does use memory bandwidth. But when you're using just a single core, you have more than enough bandwidth to take advantage of the hardware prefetching capabilities of the machine. But later on, we'll see that when we're running in parallel the memory bandwidth does become more of an issue when you have multiple processors all using the memory. Yeah? Question?

AUDIENCE: Is there a way of touching a cache [INAUDIBLE] or touching a piece of memory before you need it so that you don't need [INAUDIBLE]

JULIAN SHUN: You can do software prefetching. There are instructions to do software prefetching. Hardware prefetching is usually more efficient, but it's like less flexible than the software prefetching. But here we're not actually doing that. We're just taking advantage of hardware prefetching.

AUDIENCE: [INAUDIBLE]?

JULIAN SHUN: Yeah. So we didn't actually try that. It could benefit a little bit if we used a little bit of software prefetching. Although, I think it would benefit the looping code probably as well if we did that. Yes?

AUDIENCE: Is hardware prefetching [INAUDIBLE]?

JULIAN SHUN: Sorry? Sorry?

AUDIENCE: Is hardware prefetching always enabled?

JULIAN SHUN: Yeah. So hardware prefetching is enabled. It's always done by the hardware on the machines that we're using today. This was a pretty surprising result. But we'll actually see the fact of the memory bandwidth later on when we look at the parallel code. Any other questions before I continue?

OK. So let's now look at the interplay between caching and parallelism. So this was a theorem that we proved in the previous lecture. So let's recall what it says. It says let Q sub p be the number of cache misses in a deterministic Cilk computation when run on p processors, each with a private cache, and let s sub p be the number of successful steals during the computation.

In an ideal cache model with a cache size of M and a block size of B, the number of cache misses on p processors equal to the number of cache misses on one processor plus order number of successful steals times M over B. And last time we also said that the number of successful steals is upper bounded by the span of the computation times the number of processors. If you minimize the span of your computation, then you can also minimize the number of successful steals. And then for low span algorithms, the first term is usually going to dominate the Q1 term.

So I'm not going to go over the proof. We did that last time. The moral of the story is that minimizing cache misses in the serial elision essentially minimizes them into parallel execution, assuming that you have a low span algorithm.

So let's see whether our trapezoidal algorithm works in parallel. So does the space cut work in parallel? Recall that the space cut, I'm cutting it with a slope of negative 1 through the center, because it's too wide. So can I execute the two trapezoids in parallel here? No.

The reason is that the right trapezoid depends on the result of the left trapezoid. So I can't

execute them at the same time. But there is a way that I can execute trapezoids in parallel. So instead of just doing the cut through the center, I'm actually going to do a V-cut. So now I have three trapezoids.

The two trapezoid in black-- I can actually execute those in parallel, because they're independent. And everything in those two trapezoids just depends on the base of that trapezoid. And after I'm done with the two trapezoids labeled 1, then I can compute the trapezoid label 2. And this is known as a parallel space cut. It produces two black trapezoids as well as a gray trapezoid and two black trapezoids executed in parallel.

And afterwards, the gray trapezoid executes. And this is done recursively as well. Any questions? Yeah? No. OK.

We also have the time cut. Oh, sorry. So if the trapezoid is inverted, then we're going to do this upside down V-cut. And in this case, we're going to execute the middle trapezoid before we execute the two trapezoids on the side.

For the time cut, it turns out that we're just going to use the same cut as before. And we're just going to execute the two trapezoids serially. So we do get a little bit of parallelism here from the parallel space cut. Let's look at how the parallel codes perform now.

So, OK. So this was a serial looping code. Here's the parallel looping code. So we had 1,450 before.

About 3,700 now. So little over twice the speed up. And this is on a four core machine. It's just on my laptop.

AUDIENCE: [INAUDIBLE]?

JULIAN SHUN: Sorry?

AUDIENCE: [INAUDIBLE]?

JULIAN SHUN: Oh yeah, sure. Yeah, it's slowing down a little bit, but not by too much. OK. Let's look at the trapezoidal code now. So as we saw before, the trapezoid code does about 1,840 iterations per second. And we can paralyze this.

So now it's doing about 5,350 iterations per second. So it's getting about a factor of three speed up. I can move it around a little bit more if you want to see it. So serial trapezoid and

parallel trapezoid. Is everyone happy? OK.

Because I had to do this in real time, the input size wasn't actually that big. So I ran the experiment offline without the rendering on a much larger input. So this input is a 3,000 by 3,000 grid. And I did this for 1,000 time steps using four processor cores. And my cache size is 8 megabytes.

So last level cache size. So the input size here is much larger than my last level cache size. And here are the times that I got. So the serial looping code took about 129 seconds. And when we did it in parallel, it was about 66 seconds. So it got about a factor of two speed up, which is consistent with what we saw.

For the trapezoidal code, it actually got a better speed up when we increased the input size. So we got about a factor of four speed up. And this is because for larger input size, the cache efficiency plays a much larger role, because the cache is so small compared to our input size. So here we see that the parallel looping code achieves less than half of the potential speed up, even though the parallel looping code has much more potential parallelism than the trapezoidal code. So trapezoidal code only had a little bit of parallelism only for the space cuts, whereas the trapezoidal code is actually getting pretty good speed up. So this is near linear speed up, since I'm using four cores and it's getting 3.96 x speed up.

So what could be going on here? Another thing to look at is to compare the serial trapezoid code with the serial looping code, as well as the parallel trapezoid code with the parallel looping code. So if you look at the serial trapezoid code, you see that it's about twice as fast as the serial looping code.

But the parallel trapezoid or code is about four times faster than the parallel looping code. And the reason here is that the harbor prefetching can't help the parallel looping code that much. Because when you're running in parallel, all of the cores are using memory. And there's a memory bandwidth bottleneck here. And prefetching actually needs to use memory bandwidth.

So in the serial case, we had plenty of memory bandwidth we could use for a prefetching, but in the parallel case, we don't actually have much parallel-- but much memory bandwidth we can use here. So that's why in a parallel case, the trapezoid code gets a better speed up over the parallel looping code, compared to the serial case. And the trapezoid code also gets better speed up because it does things more locally, so it needs to use less-- fewer memory operations.

And there's a scalability bottleneck at the memory interconnect. But because the trapezoidal code is cache oblivious, it does a lot of work in cache, whereas the looping code does more accesses to the main memory. Any questions on this?

So how do we know when we have a parallel speed up bottleneck, how do we know what's causing it? So there are several main things that we should look at. So we should see if our algorithm has insufficient parallelism, whether the scheduling overhead is dominating, whether we have a lack of memory bandwidth, or whether there is contention going on. And contention can refer to either locking or true and false sharing, which we talked about in the last lecture.

So the first two are usually quite easy to diagnose. You can compute the work in the span of your algorithm, and from that you can get the parallelism. You can also use Cilkscale to help you diagnose the first two problems, because Cilkscale can tell you how much parallelism you're getting in your code. And it can also tell you the burden of parallelism which includes the scheduler overhead.

What about for memory bandwidth? How can we diagnose that? So does anyone have any ideas? So I can tell you one way to do it. I can open up my hardware and take out all of my memory chips except for one of them, and run my serial code.

And if it slows down, then that means it was probably memory bandwidth bound when I did it in parallel. But that's a pretty heavy handed way to diagnose this problem. Is there anything we can do was just software? Yes?

AUDIENCE: Can we simulate like Valgrind would do it and count how memory accesses [INAUDIBLE]

JULIAN SHUN: Yeah, so you could use a tool like Valgrind to count the number of memory accesses. Yes?

AUDIENCE: It's like toolset or something where you can make sure that only one processor is being use for this.

JULIAN SHUN: So you can make sure only one processor is being used, but you can't-- but it might be using like, more memory than just the memory from one chip. There's actually a simpler way to do this. The idea is that we'll just run p identical copies of the serial code.

And then they will all executing in parallel. And if the execution slows down, then that means they were probably contending for memory bandwidth. Does that make sense?

One caveat of this is you can only do this if you have enough physical memory, because when you're running p identical copies, you have to use more DRAM than if you just ran one copy. So you have to have enough physical memory. But oftentimes, you can usually isolate some part of the code that you think has a performance bottleneck, and just execute that part of the program with p copies in parallel.

And hopefully that will take less memory. There are also hardware counters you can check if you have root access to your machine that can measure how much memory bandwidth your program is using. But this is a pretty simple way to do this.

So there are ways to diagnose lack of memory bandwidth. Turns out that contention is much harder to diagnose. There are tools that exist that detect lock contention in an execution, but usually they only detect a contention when the contention actually happens, but the contention doesn't have to happen every time you run your code. So these tools don't detect a potential for lock contention. And potential for true and false sharing is even harder to detect, especially false sharing, because if you're using a bunch of variables in your code, you don't know which of those map to the same cache line.

So this is much harder to detect. Usually when you're trying to debug the speed up bottleneck in your code, you would first look at the first three things here. And then once you eliminated those first few things, then you can start looking at whether contention is causing the problem. Any questions?

OK. So I talked about stencil computation. I want to now talk about another problem, sorting. And we want to do this cache efficiently. OK. So let's first analyze the cache complexity of a standard merge sort. So we first need to analyze the complexity of merging, because this is used as a subroutine in merge sort.

And as you recall in merging, we're given two sorted input arrays. And we want to generate an output array that's also sorted containing all the elements from the two input arrays. And the algorithm is going to maintain a pointer to the head of each of our input arrays.

And then it's going to compare the two elements and take the smaller one and put it into the output, and then increment the pointer for that array. And then we keep doing this, taking the smaller of the two elements until the two input arrays become empty, at which point we're done with the algorithm. We have one sorted output array.

OK. So to merge n elements, the time to do this is just theta of n. Here n is the sum of the sizes of my two input arrays. And this is because I'm only doing a constant amount of work for each of my input elements.

What about the number of cache misses? How many cache misses will incur when I'm merging n elements? Yes?

AUDIENCE: [INAUDIBLE].

JULIAN SHUN: Yeah. So I'm going to incur theta of n over B cache misses because my two input arrays are stored contiguously in memory so I can read them at B bytes at a time with just one cache miss. And then my output array is also stored contiguously so I can write things out B bytes at a time with just one cache miss. I might waste to cache line at the beginning and end of each of my three arrays, but that only affects the bound by a constant factor. So it's theta of n over B.

So now let's look at merge sort. So recall that merge sort has two recursive calls to itself on inputs of half the size. And then it doesn't merge at the end to merge the two sorted outputs of its recursive calls. So if you look at how the recursion precedes, its first going to divide the input array into two halves. It's going to divide it into two halves again again, until you get to the base case of just one element, at which point you return.

And then now we start merging pairs of these elements together. So now I have these arrays of size 2 in sorted order. And I merged pairs of those arrays. And I get subarrays of size 4. And then finally, I do this one more time to get my sorted output.

OK. So let's review the work of merge sort. So what's the recurrence for merge sort if we're computing the work? Yes?

AUDIENCE: [INAUDIBLE].

JULIAN SHUN: Yeah. So that's correct. So I have two subproblems of size N over 2. And then I need to do theta n work to do the merge. And this is case two of master theorem.

So I'm computing log base b of a, which is log base 2 of 2. And that's just 1. And that's the same as the exponent in the term that I'm adding in. So since they're the same, I add in an additional log factor. And my overall work is just theta of n log n.

OK. So now I'm going to solve this recurrence again using the recursion tree method. I'm still going to get theta of n log n. But I'm doing this because it's going to be useful when we analyze the cache complexity. So at the top level I have a problem of size n. And I'm going to branch into two problems of size n over 2.

And when I'm done with them, I have to do a merge, which takes theta of n work. And I'm just putting n here. I'm ignoring the constants. And I'm going to branch again. And each one of these is going to do n over 2 work to merge.

And I'm going to get all the way down to my base case after log base 2 of n levels. The top level is doing n work. Second level is also doing n work. And it's going to be the same for all levels down to the leaves.

Leaves is also doing a linear amount of work. So the overall work is just the work per level times the number of levels. So it's just theta of n log n.

OK. So now let's analyze this with caching. OK. So we said earlier that the cache complexity of the merging subroutine is theta of n over B. And here's the recurrence for the cache complexity of merge sort. So my base case here is when n is less than or equal to cM, for some sufficiently small constant c. And this is because at this point, my problem size fits into cache.

And everything else I do for that problem is still going to be in cache. And to load it into cache, the base case, I need to incur theta and over B cache misses. And otherwise, I'm going to have to recursive calls of size n over 2. And then I need to do theta of n over B cache misses to do the merge of my two results.

So here, my base case is larger than what I did for the work. The algorithm actually is still recursing down to a constant size base case. But just for analysis, I'm stopping the recursion when n is less than or equal to CM.

So let's analyze this. So again, I'm going to have the problems of size n at the beginning. And then I'm going to split into two problems of size n over 2. And then I'm going to have to pay n over B cache misses to merge the results together. Similarly for the next level, now I'm paying n over 2B cache misses for each of my two problems here to do the merge.

And I keep going down until I get to a subproblem of size theta of cM. At that point, it's going to fit in cache. And I don't need to recurse anymore in my analysis.

So number of levels of this recursion tree is just log base 2 of n over cM. So I'm basically chopping off the bottom up this recursion tree. The number of levels I had below this is log base 2 of cM. So I'm taking a log base 2 of n and subtracting log base 2 of cM. And that's equivalent to log base 2 of n divided by cM.

The number of leaves I have is n over cM since each leaf is state of cM large. And the number of cache misses I need to incur-- to process a leaf is just theta of m over B, because I just need to load the input for that subproblem into cache. And then everything else fits in cache.

So for the top level, I'm incurring n over B cache misses. The next level, I'm also incurring n over B cache misses. Same with the third level.

And then for the leaves, I'm incurring m over B times n over cM cache misses. The n over cM is the number of leaves I have and theta of m over B is the number of cache misses per leaf. And that also equals theta of n over B.

So overall, I multiply theta of n over B by the number of levels I have. So the number of levels I have is log base 2 of n over cM. And I just got rid of the constant here, since doesn't affect the asymptotic bound. So the number of cache misses I have is theta of n over B times log base 2 of-- or any base for the log of n over M. So any questions on this analysis?

So I am saving a factor of B here in the first term. So that does reduce. That just gives me a better cache complexity than just a work bound. But for the M term, it's actually inside the denominator of the log. And that doesn't actually help me that much.

So let's look at how much we actually save. So one n is much greater than M. Then log base 2 of n over M is approximately equal to log base 2 of n. So the M term doesn't contribute much to the asymptotic costs, and therefore, compared to the work, we're only saving a factor of B. When n is approximately equal to M, then log base 2 of n over m is constant, and we're saving a factor of B log n.

So we save more when the memory size-- or when the problem size is small. But for large enough problem sizes, we're only saving a factor of B. So does anyone think that we can do better than this?

So I've asked this question several times before, and the answer is always the same. Yes? It's a good answer. So we're going to do this using multiway merging. So instead of just merging

two sorted subarrays, we're going to merge together R sorted subarrays.

So we're going to have R subarrays, each of size n over R. And these are sorted. And I'm going to merge them together using what's called a tournament tree. So how the tournament tree works is I'm going to compare the heads of each pair of these subarrays and store it in the node of the tournament tree.

And then after I do that, I compare these two nodes. And I get the minimum of those two. Eventually, after I compare all of the elements, I'm just left with the minimum element at the root of the tournament tree. And then I can place that into my output.

So the first time I want to fill this tournament tree, it's going to theta of R work because there are R nodes in my tournament tree. So when I compare these two elements, the smaller one is 6. For these two, the smaller one is 2. And then I compare 2 and 6, take the smaller one. And then on the other side, I have a 7 here.

So I compare 2 and 7. 2 is smaller, so it appears at the root. And then I know that that's going to be my minimum element among the heads of all of the R subarrays that I'm passing it.

So the first time to generate this tournament tree takes theta of R work because I have to fill in R nodes. But once I generated this tournament tree, for all subsequent rounds, I only need to fill in the path from the element that one to the output or right, or to the root of the tournament tree, because those are the only values that would have changed. So now I'm going to fill in this path here. And this only takes theta of log R work to do it, because the height of this tournament tree is log base 2 of R.

So I'm going to fill this in. Now 14 goes here. 6 is a smaller of the two. And then 6 is a smaller of the two again. So my next element is 6.

And I keep doing this until all of the elements for my R subarrays get put into the output. The total work for merging is going to be theta of R for the first round, plus n log R for all the remaining rounds. And that's just equal to theta of n log R, because we're assuming that n is bigger than R here. Any questions on how the multiway merge works? No?

OK. So let's analyze the work of this multiway merge when used inside merge sort. So the recurrence is going to be as follows. So if n is 1, then we just do a constant amount of work. Otherwise, we have R subproblems of size n over R. And then we're paying theta of n log R to

do the multiway merge.

So here's the recursion tree. At the top level, we have a problem of size n. And then we're going to split into R subproblems of size n/R. And then we have to pay n log R work to merge the results of the recursive call together. And then we keep doing this.

Turns out that the work at each level sums up to n log base 2 of R. And the number of levels we have here is log base R of n, because each time we're branching by a factor of R. For the leaves, we have n leaves. And we just pay linear work for that, because we don't have to pay for the merge. We're not doing anymore recursive calls.

So the overall work is going to be theta of n log R times log base R of n, plus n for the leaves, but that's just a lower order term. And if you work out the math, some of these terms are going to cancel out, and you just get theta of log n for the work. So the work is the same as the binary merge sort.

Let's now analyze the cash complexity. So let's assume that we have R less than cM over B for a sufficiently small constant C less than or equal to 1. We're going to consider the R way merging of contiguous arrays of total size n.

And if R is less than cM over B, then we can fit the entire tournament tree into cache. And we can also fit one block from each of the R subarrays into cache. And in that case, the total number of cache misses to do the multiway merge is just theta of n over B, because we just have to go over the n elements in our input arrays.

So the recurrence for the R way merge sort is as follows. So if n is less than cM, then it fits in cache. So the number of cache misses we pay is just theta of n over B.

And otherwise, we have R subproblems of size n over R. And that we add theta of n over B to do the merge of the results of the subproblems. Any questions on the recurrence here? Yes?

AUDIENCE: So how do we pick up the value of R? Does it make it cache oblivious?

JULIAN SHUN: Good question. So we didn't pick the value of R. So this is not a cache oblivious algorithm. And we'll see what to choose for R in a couple of slides.

So let's analyze the cache complexity of this algorithm again, using the recursion tree analysis. So at the top level, we're going to have R subproblems of size n over R. And we have to pay n

over B cache misses to merge them. And it turns out that at each level, the number of cache misses we have to pay is n over B, if you work out the math. And the number of levels of this recursion tree is going to be log base R of n over cM, because we're going to stop recurring when our problem size is cM.

And on every level of recursion, we're branching by a factor of R. So our leaf size is cM, therefore the number of leaves we have is n over cM. And for each leaf, it's going to take theta of m over B cache misses to load it into cache. And afterwards, we can do everything in cache.

And multiplying the number of leaves by the cost per leaf, we get theta of n over B cache misses. And therefore, the number of cache misses is n over B times the number of levels. Number of levels is log base R of n over M.

So compared to the binary merge sort algorithm, here we actually have a factor of R in the base of the log. Before, the base of the log was just 2. So now the question is what we're going to set R to be. So again, we have a voodoo parameter. This is not a cache oblivious algorithm.

And we said that R has to be less than or equal to cM over B in order for the analysis to work out. So let's just make it as large as possible. Let's just set R equal to theta of M over B. And now we can see what this complexity works out to be. So we have the total cache assumption.

We also have the fact that log base M of n over M is equal to theta of log n over log M. So the cache complexity is theta of n over B times log base M over B of n over M. But if we have the tall cache assumption that log base M over B is the same as log base of M asymptotically.

So that's how we get to the second line. And then you can rearrange some terms and cancel some terms out. And we'll end up with theta of n log n divided by B log M.

So we're saving a factor of theta of b log M compared to the work of the algorithm, whereas for the binary version of merge sort, we were only saving a factor of B for large enough inputs. So here we get another factor of log M in our savings. So as I said, the binary one cache misses is n log n over B, whereas the multiway one is n log n over B log M. And as longest as n is much greater than M, then we're actually saving much more than the binary version. So we're saving a factor of log M in cache misses.

And let's just ignore the constants here and look at what log M can be in practice. So here are some typical cache sizes. The L1 cache size is 32 kilobytes, so that's 2 to the 15th.

And log base 2 of that is 15. So we get a 15x savings. And then for the larger cache sizes, we get even larger saving. So any questions on this?

So the problem with this algorithm is that it's not cache oblivious. We have to tune the value of R for a particular machine. And even when we're running on the same machine, there could be other jobs running that contend for the cache. Turns out that there are several cache oblivious sorting algorithms. The first one that was developed was by paper by Charles Leiserson, and it's called funnel sort.

The idea here is to recursively sort n to the 1/3 groups of n to the 2/3 elements, and then merge the sorted groups with an n to the 1/3 funnel. And this funnel is called a k funnel, more generally, and it merges together k cubed elements in a k sorted list. And the costs for doing this merge is shown here. And if you plug this into the recurrence, you'll get that, the asymptotic number of cache misses is the same as that of the multiway merge sort algorithm while being cache oblivious. And this bound is actually optimal.

So I'm not going to have time to talk about the details of the funnel sort algorithm. But I do have time to just show you a pretty picture of what the funnel looks like. So this is what a k funnel looks like. It's recursively constructed. We have a bunch of square root of k funnels inside. They're all connected to some buffers.

So they feed elements the buffer, and then the buffers feed element to this output square root of k funnel, which becomes the output for the k funnel. So this whole blue thing is the k funnel. And the small green triangles are square root of k funnels. And the number of cache misses incurred by doing this merge is shown here. And it uses the tall cache assumption for analysis.

So a pretty cool algorithm. And we've posted a paper online that describes this algorithm. So if you're interested in the details, I encourage you to read that paper. There are also many other cache oblivious algorithms out there. So there's been hundreds of papers on cache oblivious algorithms.

Here are some of them. Some of these are also described in the paper. In fact, I think all of these are described in the paper we posted.

There are also some cool cache oblivious data structures that have been developed, such as for B-trees, ordered-file maintenance, and priority queues. So it's a lot of literature on cache

oblivious algorithms. And there's also a lot of material online if you're interested in learning more.