MITOCW | MIT6_172_F10_lec17_300k-mp4

PROFESSOR: Let's get started. So today, I'm going to talk a little bit more about performance issues in parallelization. A little bit more out of the [INAUDIBLE] to what people are doing otherwise. So normally what we have done so far, is we looked at Cilk.

It provides a very robust and environment for parallelization. It hides many issues and eliminates many of the problems out there if you find other areas of parallelization that you deal with. And in last lectures, we looked at things like cache [UNINTELLIGIBLE] algorithims, algorithmic issues, like looking at work and spend. And in fact, in your projects, you're going to use all these nice concepts to get you a nice parallel learning CorD.

But if you look at a lot of these CorD [UNINTELLIGIBLE] the very people normally for parallelized CorD outside probably Cilk. And there are a lot of other issues that arise, things like synchronization issues and memory issues.

Today, I think we are going to focus mostly on memory issues. And we are going to use open OpenMP instead of [INAUDIBLE]. And most of these issues will be affected on Cilk sometimes, too. But Cilk tries to hide them from you. There's a layer of abstract. And it's hard to kind of get to those issues in there. So we are going to look at this thing called OpenMP.

So today, we are going to address things like granularity of parallelism. There are so many things that just went out on the page, I guess. True sharing, false sharing, load balancing issues, the [UNINTELLIGIBLE]. So from the license and keep talking about that we want to be out of all this not dealing with Voodoo parameter. Today, we actually are dealing mainly with Voodoo. So I guess this should be basically the Halloween lecture. So we are all about Voodoo today and see how we can deal with Voodoo issues.

So if you look at a Cilk program, here is a nice simple matrix multiply, seem to be [INAUDIBLE] example these days. What you can do is you can put a Cilk formula in these two loops and get a nice parallel performance. However, [UNINTELLIGIBLE] from where how the memory is arranged is up to the Cilk scheduler.

Cilk scheduler is doing some work stealing. Depending on how the work gets distributed, the process will get worked, it will happen. Hopefully, everything will go nicely. And so what that means is it ties the distribution and load balancing issues.

So it's nice if you have access to Cilk, but many other [UNINTELLIGIBLE] you might not be. And even within Cilk, some of these issues might show up. So what we are going to do is step one below the Cilk scheduler.

So there's this system called OpenMP. It's a more simplified model of parallelism. So what it tries to do is instead of giving this very [UNINTELLIGIBLE] system, it lets you basically direct access to the processors. So what that means is there's normally what we call a fork-join model. [UNINTELLIGIBLE] we have with Cilk, basically. We can do fork into different workers and join. And more or less, you can actually bind these workers to [UNINTELLIGIBLE] sometimes or make sure that the number-- I'll give you some techniques how to do that as I go on.

So for parallel loops, you can do data parallelism, different [UNINTELLIGIBLE] parallelism you can do something like fork-join. And you can see a bunch of static or dynamic scheduling policies.

So for example in OpenMP, you can see for this loop that add a pragma to [UNINTELLIGIBLE] in front of this loop and say this is OpenMP. Parallel loop in here. Parallel full loop in this one. And schedule it using static chunk. I will tell you what exactly that means. And that gives you direct access to how each of these parts will be run.

So let me get a little bit in detail. So assume you have [UNINTELLIGIBLE] courses in there, [UNINTELLIGIBLE] processors. So now in OpenMP, you are basically opening the entire world underneath. And you have to kind of see what's going on.

And if you say, schedule a static chunk of four, assume you have 16 iterations. Here are my 16 iterations. So each of these dots represent a value for i. So what it says is you take chunks of four and basically send it across it. So what happens is the first four iterations will go to [UNINTELLIGIBLE] or core zero. Next four will go to core one, core two and core three.

So you know exactly which iterations run where. It's a very static thing. You have full control of what's going on. Whereas in Cilk, it's up to the scheduler. So the nice thing here is you can have full control. But you get enough room to harm yourself if you do things wrong. So this is a double-edged sword in that sense.

So instead of doing static five you do static two. You're assigning chunks of size two. What it will do is it will assign chunks of size two to the four cores. And then you're not done yet. And then you start with again core zero and assign chunks of size two.

This is called block cyclic schedule. And if you do a chunk of size one, it's called a cyclic schedule. [UNINTELLIGIBLE] cycles just assigning iterations to cores. OK. So far so good? OK.

So I want to do something. So I have this program. This is again your run-of-the-mill matrix multiply. And I ran a sequential single machine, and I got this performance. Then, I said look, I want to parallelize the outer loop. So I parallelize this loop.

What should I get? [UNINTELLIGIBLE] fast or slow. I want to just check whether you are awake or sleep. How do [UNINTELLIGIBLE PHRASE] to run slower. It's not a trick question. This is just to make sure that actually participate. How do people think it's run faster?

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK. Good. What do others think? OK. They're probably checking their email or something. OK. So actually it ran faster. The source not run on the common cloud machines. This was a previous generation that I ran. So [UNINTELLIGIBLE] was seven times faster. So this was great. I parallized outer loop.

What happens if I parallize inner loop? So this test, this i loop, runs parallel. Here, I launch the [UNINTELLIGIBLE] parallel. How much people thinks this runs faster?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Compared to this one. How many people thinks this runs slower? OK. There's some consistent answers here. Why do you think it would run slower? So OK. It ran slower, so it can improve that. And that's a little bit slow. Why is it slow?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Exactly. So what it's doing here, it's basically spawning many, many times in here. Because every time you have parallelism, you chunkify into the processor. Here you are getting a lot more smaller chunks inside. So let's look at how this is basically run.

So normally, you can think about an OpenMP program as you have one sequential thread. You run the main program. And then assume you have, in cores, you might have n minus 1 other thread just waiting for work.

And then, when you finally come to the parallel loop, it says, OK. Set up. What do you want to run on other basic cores. And release it. Release these waiting people. And let them start working on the parallel work. And also, I will start doing on my own chunk. So suddenly, when you say parallel four, it releases all other cores to go run that part of the core.

And once it's done, it's will say, OK, I'm done. I have to wait until everybody is done. So even if the main guy is done, it has to wait until everybody is finished. And then, start executing the sequence [UNINTELLIGIBLE]. So this is the gist of how OpenMP program is run.

And if you realize that it all heads here because you have basically make sure all these cases are broken up. So there's some things that has to be issued. And there's a delay between these guys can start if everybody has equal work.

Despite not finishing on time because it may take some time for this to start. And then, it has to also tell this back OK, I am done. So there's a lot of synchronization going on. Locks and unlocks. Here it's called various synchronization here. And so if this work is small, this synchronization starts dominating.

So what happens is [UNINTELLIGIBLE] fine grain parallelism. Do a little work in the parallel region, and synchronization will basically start dominating your time. So how do you take this? And sometimes when you run something parallel, it might even run slow because the amount of stuff in the parallel region is so small, [UNINTELLIGIBLE] will start dominating. And that's not a good way.

And also, sometimes you assume. And you keep increasing the number of cores. Hopefully, you want to see a nice parallelism increase, but it doesn't, even though you have enough information. But that means you're running a lot of small chunks, even though you seem to have a lot of parallelism available.

And also, you can make sure the synchronization in the time in the parallel region. If the parallel regions are on a very short time, this happens. We saw this effect when we were doing Cilk. Remember?

When did we see this granularity affecting Cilk? And what did he do? When you write Cilk programs. You write [UNINTELLIGIBLE] programs. Where did the granularity start showing up on us? It may not be exactly this because the scheduling is complicated. OK. Yes?

AUDIENCE: The two by two matrix [INAUDIBLE]

PROFESSOR: Yeah. Something like two by-- for example, that's the reason we wanted to have a large base case. Because if you didn't put a large base case, it keeps dividing into smaller and smaller problems. And if the schedule is smart, it won't be doing exactly this. But it's always good to have these large granulated chunks at the bottom.

So how to get [UNINTELLIGIBLE] granulated parallelism. What we need to do is reduce the number of [UNINTELLIGIBLE] equations. So you want to always try to look for the outer most loop you can get at all the really large independent regions. So you go look, and not [UNINTELLIGIBLE] thing you want to parallelize. You go up, up, up, up until the point you can parallelize. And that's the best way to get good performance. OK?

So if you really compare these three programs here, again, what you see-- of course, this has no synchronization. This has n amount of synchronizations. Here in [UNINTELLIGIBLE] synchronization, that's obviously a lot more synchronization going on. And that is where this [UNINTELLIGIBLE] comes from.

OK. So now, I am switching a little bit in here. I want you guys to look at this program a little bit. So what am I doing here? I have two [UNINTELLIGIBLE]. And I am just basically adding matrix B to matrix A. OK? And then I have another loop test here, adding matrix C to matrix A. And what am I doing in here? I am basically going through matrix A in another direction in here.

AUDIENCE: [INAUDIBLE]

PROFESSOR: It's not really a transpose. I'm not transposing. What I'm doing is I'm actually doing a mirror because the C gets mirrored on ix. It's because [UNINTELLIGIBLE] ix [UNINTELLIGIBLE] the other direction. OK? So it's not really a transpose. So I do a mirror addition. And then I'm asking for the two outer most loops to be parallel.

So if you run this sequential-- OK, you get about 30 milliseconds, I guess, to run in here. So that is in [UNINTELLIGIBLE]. But if you're running parallel, what do you get? Should you get faster or slower? OK. Anyone want to take a guess [UNINTELLIGIBLE] Sometimes some of these questions, you might not have enough information to answer. But it's still good to just take a stand on one direction or another. How many people think it runs faster? How many people think it runs slower? OK. Some.

Oops. What happened? What happened in here? Can anybody point out why it might be running slower parallely than running sequentially?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah. There's a cache issue. Watch the possible cache issue in here.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah. If you think about, the first equations of, I guess, the first core-- I have some diagram. I'll show it to you in there. And here, only the last data elements we'll get for the first iterations because we are going in the other direction. So if you look at it a little more deeply into what's going on.

Number of instructions seem to be a little higher. This one I couldn't actually explain why this might be the case in here. If anybody has an idea, you can say that. But this was kind of [UNINTELLIGIBLE]. This might be [UNINTELLIGIBLE] the cycles. Huh. OK. I can explain this.

Because this is a sequential run, this is a sum total of a parallel run. So because of all the overhead that happens because this was running on, I think, an eight core machine. So you're running eight times of small companies. There's a lot of overhead that goes around, synchronization, and stuff like that. So a number of instructions just blows up. But for each core, you don't have this blow up.

AUDIENCE: [INAUDIBLE] Cilk? Because does Cilk have different processor affinity, things that open [UNINTELLIGIBLE]? Because it seems like if the program, the language--

PROFESSOR: [INAUDIBLE]. Let's see if we can process the affinity information or if not. It's just pure [UNINTELLIGIBLE].

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah. I mean if you like executed locally if you have good cache [UNINTELLIGIBLE] with them. But if there's no cache [UNINTELLIGIBLE] you might steal something where data might be somewhere else.

AUDIENCE: But you'll still mimic the cache behavior, considerably, except for when you steal. [INAUDIBLE]

PROFESSOR: Yeah. So OK. We don't have a mic in here. OK. There's a mic. There we go. But if you have two different of these regions, the way the parallelization happens can be different.

AUDIENCE: In Cilk, what happens is the code is mimicking, for the most part, exactly what the C or C++ code would be doing. And so you get exactly the same cache hits and misses. Except when you steal, it's like starting over with an empty cache. OK? But as long as you have sufficient parallelism, the steals don't occur very often. And so therefore, you end up getting the same kind of behavior that you would get out of the serial code.

PROFESSOR: Yeah. But Charles, in this one, because you had to steal everything here, the between here and here, the parallelism would be different.

AUDIENCE: There would be no affinity between those two.

PROFESSOR: No [? affinity ?] will be there. Yeah, exactly. So in the sequential one, everything that fits in the cache, so that would be affinity because we are not doing parallelism. And that's what I think happened here. Because you had no [? affinity-- ?]

AUDIENCE: No in a serial code, there's no [? affinity. ?]

PROFESSOR: No. Serial code-- if this fits in the cache, it's then running on one core. So if it fits in the one core's cache, you're happy.

AUDIENCE: So the issue is-- right-- is if you only access it once, by the time you fill up the cache-- It takes some time to fill up the cache to get them synchronized.

PROFESSOR: So it fits in the one core's cache, it's OK. Otherwise, it has no affinity in here. So the key difference in here is, of course, CPI is slow. We don't know exactly why. But in [UNINTELLIGIBLE]. So what you find is that there's a huge amount of cache in [UNINTELLIGIBLE] going on. So that should give you a feeling of what's going on. So let's look at what might happen.

So I'm showing this matches [UNINTELLIGIBLE] last year on-- what we had were Cagnode machines that were basically code to quad processor. So we had eight codes in here. And I put them-- you don't have to now look at this table. I put them in the slides so you can look at it later.

And so this is the last year's machine. And of course, this year's machine is different. We have two six core processors in here. So this is what we [UNINTELLIGIBLE] this year. [UNINTELLIGIBLE] OK.

And so right now, I'm showing numbers for this one. And later, I will show what happened in the [UNINTELLIGIBLE]. So if you look at a cache-- so what happened is each of the data items in the cache can be in multiple states. This is called MSI protocol here. What that means is the item might be modified. If it is modified, it can be only in one cache.

If anybody else wants to touch it, it has to get it out of the modified state. Or it can be sharing. Sharing means it's reading. So that means that item is read by multiple people. And that can have multiple covers. So sharing items can be in multiple places.

However if you're modifying, [UNINTELLIGIBLE] items in everybody else. So that means if I modify something, it can only be in mine. If other people had that data, I had to go in and validate this. So if you modify this, I had to go in and validate it. So that's a sharing state. That means I'm [UNINTELLIGIBLE] everybody [UNINTELLIGIBLE] read this. But if I ever want to change that, I have to go in and validate this one.

So what that means is when I start writing, I am validating it from everybody. So even if everybody kept a copy and they start modifying, I had to get my own copy. And everybody else will invalidate. And then, if somebody else wanted to read-- for example, if this guy wants to read-- basically, this has to make this a sharing and back to sharing. That means I have to get the value 13, propogate it, and this becomes sharing again.

OK. Did you get that? What's going on in here? In the cache? So reads, everybody can keep a copy if they want. Write-- only one guy can keep a copy.

So what happens then is true sharing. So you have these two different cores. So I want to read something. So I get it from outside probably on main memory, and I put it in my cache in here. And then, the next guy wants to write the same thing. Assume I'm writing that. And once I want to write that, I can keep this copy I had invalidated from here and get a copy here. And then, if this way, I want to write it again, I have to basically invalidate it from here and get a copy.

If I'm reading, both of us can keep a copy and just kind of keep bouncing back and forth, back and forth. And so if you bounce too many times, you get all of these invalidations. So the fact I looked at that I have invalidations basically tells me something like this is going on.

So what's happening in this program? When I parallelize this four loop, my four cores-- basically since I am doing here [UNINTELLIGIBLE]-- are going to get this nice distribution of data into the caches. Assume it fits in cache. OK. So all this data nicely fits into cache, and now I'm pretty happy when I run this one because I got this data into the cache. And I write it.

But the minute I go in here, basically this data has to [UNINTELLIGIBLE]. OK, because I am going this is n minus i in here so this data has a flip route. And by doing this, basically, I incur this huge amount of [UNINTELLIGIBLE], and it slows down. OK? So that's why it didn't work well.

So what can you do? When you have these read, write and write, write conflicts in here. And you have to actually move the data across in here. And what you can do is look for this true sharing. You can look at the [UNINTELLIGIBLE] and see if we have excessive [UNINTELLIGIBLE], we have a problem.

And how do we eliminate that? You want to make the sharing minimal. If you want to get some data into a cache, you want to try to keep it there as much as possible. And if you're using, you'd want to try to align everything across. So even across different regions, it'll use the same kind of things. And/or enforce some kind of [UNINTELLIGIBLE] technique to keep the data alive. So there are a lot of techniques in there, but true sharing can be an interesting problem.

So in here, simple change, yes. You're, basically, instead of changing A, you change C. So you write A the same way. But now what I have done is I am doing the mirror by changing the axis to C, is to [UNINTELLIGIBLE] is the same as this axis. So these two are the same thing.

And the minute I do that, voila! I get good speed up. Because if you look at that, my inundations has gone down. My L1 cache [UNINTELLIGIBLE] has now really, really gone down. I'm not doing anything. And of course, I am doing more instructions here than this one because-- I think, the difference between instruction here and here is because a lot of times synchronization operations are dynamic because in the [UNINTELLIGIBLE] miscounted as the instructions, you are busy waiting in there. So this number is not really a constant number. OK, question.

AUDIENCE: Not a question. So another thing one could do here is do loop fusion.

PROFESSOR: Yes. Yes. Here is a nice way of putting both of the loops into one and do loop fusion. And that works. In this case, you can do that.

AUDIENCE: So loop fusion is where you take two loops, and you convert it into one loop. So in this case, you could have just written one nest, which has two things going on inside. And then you would save all the loop overhead and the scheduling overhead. So rather than doing it twice, you actually have reduced the overhead to just the parallelism of the one loop. So if you look at that, you'll realize that you could somehow make it just be a single nest with two statements in there, rather than one.

PROFESSOR: So basically, instead of [UNINTELLIGIBLE] entire thing and move plus C into here, basically. And I could have just done it in one loop nest. That's what loop fusion would do here. So we can actually [UNINTELLIGIBLE] much nicer in here.

But just for example purposes, so now I really reduced this one and got that. So this is great. Cagnodes really showed this classic problem in the computer. And so I'm like, OK. Now we have new machines. Let's try it and see what happens.

What does this show? This is your nice cloud machines we've got now. I have no slow down. I was really disappointed because beforehand, I had this for sharing going on in here and had a really big slow down. But this one, in fact, the difference is very small. And when you look at any kind of performance counters, they are pretty comparable. There's nothing much going on here.

So what do you think is going on in this new architecture now? Why this might be?

AUDIENCE: [INAUDIBLE]

PROFESSOR: That's an interesting observation, but also we have-- yes, core seven basically is two by two in the die. But we also have two different processors. So that's there, too. So in some sense, when you get our two-way process [UNINTELLIGIBLE] So that's there. That might help. That's an interesting observation.

What else might be going on here? Why do you think they manage to get this one? What might be another answer? What can hide these kind of delays that can happen? Load delays, and cache misses, and stuff like that. What techniques and hardware can hide those? Just [UNINTELLIGIBLE] a speculation.

Prefetching. So most hardware has an internal mechanism. When you start fetching data, you say, aha! I see a pattern. I know you want to get this thing. Let me go forward and bring more data, thinking you are going to follow that pattern. OK. All or most of the [UNINTELLIGIBLE] for, I think, have [UNINTELLIGIBLE] even a Pentium had something for prefetching going on.

But most of the time, what happens is the prefetching engine can't keep up. If you are getting there, it's [UNINTELLIGIBLE] a little bit further. You are going to catch up, and you're going to start because you have more [UNINTELLIGIBLE] here. I think [UNINTELLIGIBLE] has a really, really good prefetcher. And then, we saw it in our architecture slides, too. That a lot of things that used to happen before is gone. So this is really good. What that means is a lot of weird stuff that's going on [UNINTELLIGIBLE] making them disappear. So these kind of problems don't show up. So that's the nice story.

The other part is, OK. Now if you start really tweaking your programs to one architecture, you wait a generation. And then now, we have done either the tweaking-- the best case, tweaking has no impact, and it's not affecting anything. In most of the time, worst case, tweaking actually slows down the program because you are trying to do something complicated. That's just not needed anymore.

So even though these kind of things showed up in [UNINTELLIGIBLE] architecture, it's not an issue. But if you go to many of the smaller architectures that have that don't have that much of the very popular prefetchers, this kind of issue you would see. So for example, if you go to a cell phone [UNINTELLIGIBLE], you would probably see these kind of issues happening.

Any questions here so far? So that's the good news. You guys don't have to worry about it too much. But at least it's good to know the technique because you'll see it in other architectures.

So now, I want to switch a little bit into looking at programs that don't have what we call data parallelism. That means you can start and say, [UNINTELLIGIBLE] parallels. Everybody get the different chunk and run. And we are going a little bit more deeply into looking at programs that are a little bit different.

So I wanted to come up with this little representation to represent the program. And so if you think about iteration space-- actually before you, I'll go down to dependence. I'll also do a little bit of load balance. So here's a loop that in my iterations-- the first one I transformed zero to eight. But J only runs from one to eight. So each I, I have less and less amount of J iterations, basically. OK? It's a triangular loop. OK?

OK. So this is the way to represent iteration space, so I will represent data and get back to this again. So if you look at a data space, you can assume data iteration space could be this funky, triangular, hyperplane type of thing. Whereas data is mostly [? rectangulineum ?], multi-dimensional rectangle.

So for example, if I have [UNINTELLIGIBLE] and it's a one-dimensional one, this is basically a two-dimensional data. And you can have three-dimensional cubes and stuff like that. You can represent data like that. So this is a way to nicely represent.

And when you start thinking about it, we can look at what's going on. OK?

So now you have this loop again. So here's the basic [UNINTELLIGIBLE] iterations. And here's the data. Assume this is both A and B. There will be another one for matrix [UNINTELLIGIBLE] B. One data into each iteration is going to touch. So these are the data that need to get touched, and here's the iterations you are going to run.

So we can say OpenMP parallel four. So what happens when you do parallel four? So I am going to get parallel. And so core one gets this one, another core, another core, another core get these iterations running.

So what happens if you do this one? Do you get really good performance? Why?

AUDIENCE: [INAUDIBLE]

PROFESSOR: It's not balanced. The load is not balanced in here. So basically if you run sequential and if you run block distribution, I get about 3x performance in here. So if I look at closely, here is the number of iterations given to each core. The first core gets almost nothing, and the last guy gets a lot of work. Here's where something like the Cilk runtime can come into play because with Cilk runtime, basically, this guy will finish the [UNINTELLIGIBLE] start stealing from somebody else. And so it would be done nicely. But whereas if you do a static schedule, you are in this big bind. You don't have too many things going on.

And basically, this is what we call load imbalance. So what you can do is figure out a complicated partitioning so you can statically partition this out. Or you can do something like the dynamic scheduler like the [UNINTELLIGIBLE] scheduler for a solution.

So how to detect load imbalance? Basically, what you might want to do is for each of the different sections you are running, you want to look at the time mistakes. And in the [UNINTELLIGIBLE] axis varying, huge varying, that means there's a load imbalance going on. So you might want to check and make sure each of the parallel regions time is taking. And that gives you this view.

How to eliminate load imbalance or the use of dynamic scheduler that will deal with that. Or you can do a different distribution statically. That will not partition in this large block. So let me show you a static part because we have already learned the dynamic part before.

So now instead of doing that, we do a cyclic distribution. We use a static one. That means if you have a lot more than and a little bit better distribution so what happens to the processor? Zero gets this one and this one. One gets this one and this one. So on and so forth. So that would be a little bit between balancing there. But if you have enough of cyclic, the imbalance would be much lower.

So should we run faster? So here's the iterations each guy gets in here. This looks very balanced because I had a lot more iterations than this eight one. This is not that balanced here because this guy gets a lot more than the first one. The first one gets six. And the second and last one gets a lot more.

Uh oh. What do you think is happening here now? I ran again slower. See I guess the people in class last year had things worse because they had this old processor that did all these crazy things on them. and you guys get the fast one that doesn't do that.

So why do you think cyclic distribution is running a lot slower? What might be going?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Spoiling [UNINTELLIGIBLE] it's not that much because if you don't run this and synchronize, what you do is you run the same amount of tread and say, now, instead of running continuously, you run jumping all iterations. You should run zero and nine or whatever jump over iterations. Why do you think?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, there's a cache issue. All this time and the question is not sure, it's probably a cache issue. What type of cache issue do you think is going on?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah. [UNINTELLIGIBLE]. But let me show you what happens. So you get off then. OK, so if you look at-- the data is here so let's look at what happens. So this is running a [UNINTELLIGIBLE] lower. It's showing a lot more instructions, but instruction doesn't tell you too much because a lot of them might be missing synchronization costs in here. So instruction is not that illuminating here.

The big illumination here is this one again. Invalidations. I have a huge amount of invalidations going on.

So here is a case of false sharing. So what happens is now things next to each other, you want to multiply different processors. We're not touching the same data. Everybody's looking at somebody else's data.

So what happens is assume I want to write this data item. I like that data item. But I get the entire cache line because when I ask for that, I get my synchronization by the cache line. I get this entire cache line coming in here into this one.

And the next guys [UNINTELLIGIBLE] at me. This core won't write this data because instead of blocks, I basically give each strips. There's a lot of overlap between strips. So this guy says not to write this one, I had to get the entire cache line going back here.

And so if you want to write that again, I had to get the entire cache line going back even though we are writing different data. Because we are sharing cache lines in here. This thinking was in back and forth, back and forth, back and forth. I have a lot of cache [UNINTELLIGIBLE]. Things are really shot. OK?

And so what happens here is if you look at the cache lines-- there's my animation. So cache lines basically mess this all up. You can see that really carefully. What happens is between these lines, there would be some overlap of cache lines. And this overlap in cache lines keeps bouncing back and forth, back and forth in here.

And so what happens is basically cache lines are bigger than the data size, or there's overlap in here, and the cache line is shared when the data is not shared.

And so how to detect false sharing in too many conflicts. You assume this is a nice parallelism, but suddenly, you don't have a speed up, and you have a lot of conflicts here, even though there isn't something to be sharing.

And how to eliminate false sharing. Make data used by each contiguous in memory. That's a good way of doing that. Or pad at the end. So these kind of at the corners, there's not going to be any overlapping.

So in here, one thing you can do is, you can measure each thing that each of the cores get. We can make [UNINTELLIGIBLE]. But before what happens was a core used to get this line and this line. There are different places in memory. But you can make these two contiguous in memory by basically now, instead of having a two-dimensional array, you made that a three-dimensional or disarrays.

AUDIENCE: Can you say that again?

PROFESSOR: So before you what just happened was each of them were going to get this line and this line, each core. All these lines that were in different parts of the memory. In here, each would get only two lines. But they're in a different place. So if you have more cyclic, you'll get a lot more lines or lower memory.

So what we can do is we can arrange the cache. So if you think about this, you can think the cache, now the data, is instead of two-dimensions is three-dimensional data. One dimension is this cyclic part in here. So we can do that.

And then, you can change any way that the cyclic part, the one that I got this line and this line, now become contiguous. So you think about data as a two-dimension. You think about it as a cube. And you kind of change the cube for the inner dimension to be the one that's contiguous. So you can do data [UNINTELLIGIBLE] transformation and get there.

So now what happens is the role of core zero just gets contiguous in memory. And core one gets contiguous in memory. So if you're trying to make it contiguous, that's great. So between padding and making things contiguous, you can get good performance.

And if you do data transformation, voila! My invalidations just went down drastically. I again have a nice load balancing here. Invalidations went down drastically. That means my [UNINTELLIGIBLE] increased a little bit and I get really nice speed up.

So here are the kind of crazy things you are to do if you are doing things like algorithms that are not cache obvious. And if you are doing directly parallizing yourself without letting a nice [UNINTELLIGIBLE] time to help you. Something like a [UNINTELLIGIBLE] assistant.

So I'm just going to summarize this because this is important. We looked at a bunch of cache issues. We looked at cold missiles, capacity missiles, and conflict missiles before. And today, here are some examples of true sharing missiles.

What happened was I am actually really using data, but I set up my parallelism in such a way that between different executions, my data has to move across. [UNINTELLIGIBLE] So I am truly sharing data, but the data has to go to somebody else's cache. So I've got a lot of [UNINTELLIGIBLE] violations here.

More into this one is more like false sharing, where you assume there's no sharing, nice parallelism, everything, except the program runs very slow. And that can be because of false sharing. So we just kind of touch on these two topics. OK?

So let me switch gears a little bit about dependences. We touched on the dependences a little bit. And these are two fine programs that are not completely parallel. So normally, what happens is a true dependence means that I'm writing and reading [UNINTELLIGIBLE] other way out. And if two guys are both fighting, then the order has to maintiain us out would be dependence.

And did our dependence even loop, because these are single items. So if you have an error here, this is becoming a lot more complicated. Because there's no simple thing in here. Because it's not just using the same iteration. You might be using data from different iterations. So what happens is there's a dynamic instance of iterations. So one iteration writes the data, and somebody else might be reading the data.

And that is basically the order we have to [UNINTELLIGIBLE]. Let me give you an example. This kind of demonstrates what's going on. OK? And when you edit, you say look, this is where you [UNINTELLIGIBLE] complicated.

So in order to give you and example, let me look at this program. I have a simple program here. Ai equals Ai plus one. My iterations-- I'm running five iterations in here. So this is my iteration space. I have a large array, so this is my data space.

And now, I keep running this program. So what happens is this is time going down in here. So the first situation basically, I first read and then write. Same in the second iteration, I read and write. Third iteration read and write. Fourth iteration, read and write.

Do you see how this is going on these four situations? Second iteration, third iteration, fourth iteration, [UNINTELLIGIBLE]. OK. So what happens is first iteration read this value is zero. And write the value as zero in the menu writing. Second iteration A1, A1, A2, A2, A3, A3.

So now, when this is writing, that's a dependence between these two. You see the true and entire output dependence between these two.

What type of dependence do we have? [UNINTELLIGIBLE] dependence. True dependence is what? What to what? What's the first thing that occurs? What's the next thing that occurs? Anybody want to answer?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Write to read. So you have the first thing has to be write to read. Watch this. This is a read to write. So what type of dependence is this? This is anti-dependent. So here is ante-dependence in here.

But the nice thing about that is this dependent didn't cross the iteration boundary.

So these black lines are my iteration boundaries. So these are for situations that [UNINTELLIGIBLE]. So there's no iteration crossing in here.

You can kind of [UNINTELLIGIBLE] it using each of these iterations and my dependencies within the very same iteration. So the same iteration I have dependency [UNINTELLIGIBLE]. OK? This is a simpler case.

So let's look at something a little bit more complicated. So I have Ai plus 1 equals Ai plus 1. So what happens is first I am reading Ai. And then, I am writing Ai plus 1 in the same iteration.

The next iteration, I am reading now Ai [UNINTELLIGIBLE] this is A0 and 1. This is A is 1. [UNINTELLIGIBLE] I am writing Ai plus 1. I am writing 2. So I have a dependence like this now. What type of dependence is this? This is a true dependence because I am writing. And this is actually reading what what it is writing.

So does this look parallel? No. Because what happens is if you look at each iteration depends on the previous iteration. So you have to actually have this dependence going back and forth, back and forth in here.

So let's look at a couple more other things. So here is Ai equals Ai plus 2. So I am basically reading Ai plus 2. So I am reading this one. I am writing this one. Reading this one. Writing this one. Here is my dependence that's in here. You see the two are anti in here. This is anti-dependence because I am going from a reading to a write in here.

Can this loop be parallel? Can this loop run parallel?

AUDIENCE: [INAUDIBLE]

PROFESSOR: So can every iteration run parallel? There could be basically. No because what happens is if you look at that, there's a dependence that goes like this. And of course, there are two chains. So if you are interested, you can run at least two-way parallelism. You can run one chain parallel to another chain when you do get that much parallelism.

How about this one? 2i and 2i plus 1. [UNINTELLIGIBLE] Is there independence in here? Nope because one is-- you are reading all the elements and even writing elements [UNINTELLIGIBLE] dependence. So you can have a missing parallel. OK?

So this is the kind of interesting thing that is going on. So next, I want to look at something a little bit more complicated. So let's look at this.

So here's a classic algorithm called successive over relaxation. So it kind of simulates a lot of times things like heat flow through a plane. So the idea there is-- let me illustrate what he does.

So assume you have a big metal sheet. And you put some kind of heat source in one place. And after sometime, it all reaches a steady state. The other side might be cold. And you want to know part of the sheet's temperature.

Because temperature can leak out. And there are more things like you have a heat source and others that [UNINTELLIGIBLE] to work a glass of water or some kind of a sink. So what is the heat distribution?

So one way to compare that this is basically the same [UNINTELLIGIBLE]. The heat value here is basically the average around all these other values right now. Because if something is too hot, the heat is going to propagate something that is too cold. The heat is going to propagate because it kind of has to be average around that.

Then, you take the average in here. So what it's doing is calculating the average in here. And then, you have to do it many, many times. So if you have a heat source, at that point, it would be very hard. And then, it will start propagating slowly and kind of propagate down.

And the cold side in this way or after running many times, it basically stabilizes. And at that point, you have the kind of heat distribution that we [UNINTELLIGIBLE] have. This is the kind of calculation you do.

So this is the calculation. So what you're doing is calculating this. You are creating this, this, this, and this and updating that. And then, you do it for t time stamps. So you just go around doing each of these things first and doing it for t time stamps. OK?

So we would like to run this parallel. So here's my basically data space. There's my data items. So here's my array, two-dimensional array. So this is how I'm trying to update. I'm reading all this file.

So here's my iteration space. So what I have looked at this. I don't want to-- it's hard to give you a 3D diagram. I don't have a 3D projector. So what I'm showing here is three-dimension here. So this is the previous iteration, first iteration.

So if I still go tij. So you go through t, and then you go through i in here, and then, when you're done, you go to the [UNINTELLIGIBLE] iteration and you go this way. So here's how you would iterate. So you run this one, this one, this one, this one, this one, this one, this one, this one, this one. And then increase t by 1, and go like this.

And right now, we are here. We are trying to update this one. That's what we are trying to do. And that means we are already finished up to this point. All these points are finished up.

Now, what we have to do is figure out when I'm reading, who actually wrote this value. OK? First of all, let's figure out which iterations might be able to write this value. So if you look at this value, this relationship in between here. This one, basically, is ij. And this is ij, ij, ij. These three iterations can write this one. So and these iterations can write this one. Let me go to this one.

This is a pretty darn complicated [UNINTELLIGIBLE]. So what that means is in this one, this one already wrote something. This is what I'm reading in here. This one already wrote something. This is what I'm reading here. This iteration wrote something. I read it here. OK. Everybody following so far?

How about this, guys? Who wrote the value I am reading in these iterations? In this one, I haven't reached there yet. So who has written that? So I assume this is t equals 1. [UNINTELLIGIBLE] somebody has to write those things. So what that means is this also wrote all of those values because I have done those iterations.

But the interesting thing is some of these values got overwritten. This value got overwritten , this value got overwritten. So these two values disappear. This value got overwritten by this guy. This value got overwritten by this guy. OK. But we haven't overwritten this value, this value, and this value yet.

This one, basically, I've just updated. But I [UNINTELLIGIBLE] this one. Do you see this? Is everybody following me?

AUDIENCE: Once again, sir. I got lost. So what are [INAUDIBLE]

PROFESSOR: So what happens is I am trying to update in this iteration because this array get rid of multiple times. But in each iteration, you are only doing one update. So I am trying to read and write in here.

So I need to read all of these five elements in this iteration. So I want to figure out who wrote that. OK? This one can be written by this guy and this iteration. Could this iteration write its value in here? OK? [UNINTELLIGIBLE] This iteration write because we see it's writing ij. I mean my diagram is not that great because I have three in here and five in here. So just bear with me on that. So assume I am writing ij in here.

So my iterations go from 1 to n, but my data goes from 0 to n plus 1, basically. 1 to n minus 1 iterations. 0 to n plus 1 data. So data is bigger than iteration space because of [UNINTELLIGIBLE]. So what happens is when I'm in this iteration, we'll say this is 1 2 iteration. I will write this value. This iteration will also write this value. OK? You see that?

All of these iterations are the same. This iteration we will also write this value. But right now, who is the last guy who wrote it? The last guy that wrote it is this guy. Because this iteration wrote it, and after that, it got ordered in this one.

But this one hadn't occurred yet, so it hadn't been ordered by this guy. So the last guy who wrote it was this one. So that's why I had to eliminate this. But this data value-- I haven't executed this iteration yet. So nobody had written this one in this time stamp. So it has to be from the previous time stamp.

So I read two values from the current time stamp, three values from the previous time stamp. These three values have to come from the previous time stamp. These two values that come from the current time stamp. You see that? OK. Good.

So what that means is because dependence means-- OK. This line, this dark, red line. See. I am reading a value in a current iteration that was written by this iteration. So that means I have no dependence between these two iterations.

OK. This line, this dark, red line. I am reading a value written by this iteration. So I have a dependency in here. This line means I have a dependence between this iteration to the current one. This line means I have dependence between this iteration and the current one. This line means I have dependence between this iteration and the current one. You see that?

OK. So now, I want to see how we can parallelize this group. So what can I do? So I look at all this dependence. At this point, I don't have to think about all this where who wrote what. I can say this is dependence. In order to do this equation, all these iterations have to be done because I am losing the values produced by them. So these have to be finished before I can patch that.

So the parallelism means I tried to do things in parallel. So can we parallelize this loop? Can we run each time stamp separately? No because I am using these three values from the previous time stamp. So I can't run this time stamp, B equals 1, until B equals 0 is done. Or B plus 2 [UNINTELLIGIBLE] B plus 1 is done. OK? So I can't parallelize this loop.

OK. Can I parallelize this loop? Why? Will dependence stop me from parallelizing this one? So I'm looking at i. This is my i dimension. How many lines, at least, tell me. How many dependencies are going to stop me from doing that? OK. Good. I have [UNINTELLIGIBLE]. Somebody says three. Somebody says one.

OK. Let's get a vote. How many people think it's three? OK. There's one vote for three. How many people think it's three? How many people think it's one? Wait a minute. One vote for three and two votes for one? OK. Where's the rest? For two? For 0? Can't be 0 if the 0 is parallel. OK. So we'll start parallelizing.

OK. So what happens in here? Right now, this is actually one. This one. Because these things don't participate because this has already happened. When you go to ij iterations, these are already done. So you're going from t. So you're looking at the current iterations because you're ending in two loops.

So the t is done. So these all are already done when you go try to parallelize sides. So I don't have to worry about these three. In here because actually I'm losing t of something here, I am in trouble. So when you go look at this one, I have this one. So every dimension has a dependence in here. So I can't run it in parallel.

So does this mean that there's no parallelism? Who think there's no parallelism? Who thinks there is? Oh, somebody thinks there's no parallelism. Who thinks there's parallelism? OK. More people think there's parallelism. Let's see what we can do. Question?

AUDIENCE: Do you really think [INAUDIBLE] I'm trying to figure out how to word this. Do you really want to have dependence on the same concept? [INAUDIBLE]?

PROFESSOR: Yeah. I mean you can do-- this is the way this SOR is sitting so there's a dependence between time stamp. There's another SOR. What they do is kind of a red, black. So when you calculate the next time stamp, you calculate it right and complete the new array. So there's no dependence. So that's a different algorithm.

This algorithm, basically, uses sum value from [UNINTELLIGIBLE] because the value-- the algorithm you're talking-- you already created the other copies. You had two copies. You're bouncing back and forth. Nice. No real problem in here. But then you had to have twice the amount of storage.

Here, you are updating in. And since this is kind of running enough iterations until it converges, it doesn't seem to matter that the [UNINTELLIGIBLE PHRASE].

OK. So we cannot find a loop, what we call doall loop. The doall loop means there's no loop carried dependences. It's fully parallel. This is the best case.

So what happens is when you get there, everybody can run parallel. And when you're done, you can stop and then do that. So this is the doall loop. Of course, there's no doall loop. We can look at every dimension. We had some kind of dependence.

So there's another choice, what we call doacross loop. What that means is we have some loop carried dependence. There's something I have to use for the previous iteration. But it's only one thing.

I have a lot of other things I can run around that only I just have to wait one thing. One is done. I can just keep running. And if I calculate and send this one early, then I can do my other calculations later.

This is not that great. If you look at the difference here. This definitely has very little overhead in here. This can run slow. And of course, this thing gets produced very late. It's [? almost ?] sequential. So I hope you can just-- it the other guy wants something, I can immediately send it very early. And then I can run there. So you can get some kind of doacross patterns in here.

So if you want to do this one-- this is a little bit crazy in here. But they'll do it in here. And so what first we are to do is you are to say, OK. Look. I'm running this loop, the i loop in parallel. But I have to exchange some data. Before I want to run this one, I have to basically get the previous i value produced. And when it's done, I can say the next guy can use it.

So this is a very complicated one. I don't want you to understand it too well. So the reason I put it is to show that OK, if you want to spend a week trying to really call this up and understand and make sure that it works OK. So you can do things like that. OK? Aha. So this is the true voodooness. OK.

AUDIENCE: So in Cilk, if you do this with divide and conquer, you can make it be what I called in the Tableau construction. Each layer here is basically constructing a Tableau. And so if you do it with divide and conquer, you can do it with a very simple recursive code. But you can also do it with a loop that goes diagonally.

AUDIENCE: [INTERPOSING VOICES]

PROFESSOR: Yes. I'm going to get that. That's next.

AUDIENCE: Sorry.

PROFESSOR: That's OK. So the reason that I'm showing that is because this class is not just about how to make the cores exactly run faster. Think about algorithmic issues and stuff like that. So sometimes, when you look at a problem, it looks crazy. And there might be some changes you can do that you can get to run things in parallel.

So I'm actually doing not diagonal. I'm actually doing something very simple. So what I have done here is I have all these dependences in here. OK? So the problem here is I can't find a single [UNINTELLIGIBLE] that basically has no crossing.

But if you look at this [UNINTELLIGIBLE] diagonal here. What you see is, in fact, there's nothing that crosses the diagonal. OK? So this one basically doesn't depend on this one or this one. It only depends on the previous one. So I can run everything in the diagonal parallel in here in this one.

So of course, I can't write anything [UNINTELLIGIBLE] in here, but there's a cute trick you can do. What you can do is you can take iteration space and skew it. So what I have done is now instead off the same thing, instead of this being a square, now I skewed it a little bit. OK?

So what that means is when I'm running first i, I basically don't run any here. Then, I run this one and this iteration here. So what I have done is I have kind of moved my iteration space around. Do you see how this might be?

So now, the interesting thing is when I skew, if I look at this line, I can parallelize in this one because all the dependences come from the previous iteration. Am I right? [UNINTELLIGIBLE] Yeah. I skewed it. Yes, everything in here, these ones are parallel. OK?

And any dependence comes from the previous iteration. There's no current iteration in here. Everything in this one is parallel. So I can parallelize this. So this one doesn't depend on this one or this one. So this is all parallel. This is a little bit more complicated. So if you're interested to go deep, just go stare at the slides. I have the slides out there to understand how that happens.

So if you think about what I'm running here in parallel is the one basically this diagonal in here. So what happens is if you run this, this, and this parallel, there's no dependence. I don't need this one or this one to run this one. So I can run this, this, this, this, all this diagonal in parallel.

But the trouble with just the diagonal is I don't have a place in here to say [UNINTELLIGIBLE] for a diagonal. So I basically skewed it and then made a diagonal into one loop. So then, now what happens is basically j loop I can run parallel. This one. So I can do it four [UNINTELLIGIBLE] four. OK?

So here's something you found a problem that has no nice parallelism. But you realize there's kind of a what you call a wavefront going on here. Wave going on here. So not the given dimension, but there's another dimension that you can parallel. So you kind of skewed your space to get that nice [UNINTELLIGIBLE] line. And you run parallel. So that's all I have for today.