

MIT OpenCourseWare
<http://ocw.mit.edu>

Multicore Programming Primer, IAP 2007

Please use the following citation format:

Rabbah, Rodric and Saman Amarasinghe, *Multicore Programming Primer, IAP 2007*. (Massachusetts Institute of Technology: MIT OpenCourseWare).
<http://ocw.mit.edu> (accessed MM DD, YYYY). License: Creative Commons Attribution-Noncommercial-Share Alike.

Note: Please use the actual date you accessed this material in your citation.

For more information about citing these materials or our Terms of Use, visit:
<http://ocw.mit.edu/terms>

MIT OpenCourseWare
<http://ocw.mit.edu>

Multicore Programming Primer, IAP 2007
Transcript – Lecture 11

The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

PROFESSOR: Let's get started. So what we are going to do today is go about discovering other alternating methods. We know you guys are amazing hackers and you can actually do all those things by hand. But to make multi-core generally acceptable, can we do things automatically? Can we really reduce a burden from the programmers?

So at the beginning I'm going to talk about general parallelizing compilers. What people have done. What's the state of the art. Kind of get your feel what is doable. Hopefully, that will be a little over an hour, and then we'll go talk about StreamEd compiler, what we have done recently, and how this automation part can do.

So, I'll talk a little bit about parallel execution. This is kind of what you know already. Then go into parallelizing compilers, and talk about how to determine if something is parallel by doing data dependence analysis, and how to increase the amount of parallelism available in code loop, what kind of transformation. Then we go look at how to generate code, because once you see that something is parallel, how you actually get to run parallel. And finish up with actually how to do communication code in a machine such as a server.

So in parallel execution, this is something -- it's a review. So there are many ways of parallelism, things like instruction level parallelism. It's basically effected by hardware or compiler scheduling. As of today this is in abundance. In all for scalars we do that, in [OBSCURES] we do that. Then password parallelism, it's what most of you guys are doing now. You probably find a program, you divide it into tasks, you get task level parallelism, mainly by hand. Some of you might be doing data level parallelism and also loop level parallelism. That can be the hand or compiler generated. Then, of course, pipeline parallelism is more mainly done in hardware and language extreme, do pipeline parallelism. Divide and conquer parallelism we went a little bit more than in hardware, mainly by hand for recursive functions.

Today we are going to focus on loop level parallelism, particularly how do loop level parallelism by the compiler.

So why loops? So loops is interesting because people observed in morse code, 90% of execution time is in 10% of the code. Almost 99% of the execution time is in 10% of the code. This called a loop, and it makes sense because running at 3 gigahertz, if only run one instruction one, then you run through the hard drive in only a few minutes because you need to have repeatability. A lot of time repeatability thing loops. Loops, if you can parallelize, you can get really good performance because loops most of the time, each loop iteration have the same amount of work and you get nice good load balance, it's somewhat easier to analyze, so that's why the

compiler start there. Whereas if you try to get task level parallelism, things have a lot more complexities that automatic compiler cannot do.

So there are two types of parallel loops. One is a for all loop. That means there are no loop carried dependences. That means you can get the sequential code executing, run everything in parallel, and at the end you have a barrier and when everybody finishes you continue on the sequential code. That is how you do a for all loop. Some languages, in fact, have explicitly parallel construct, say OK, here's a for all loop and go do that.

The other type of loop is called a foracross or doacross loop. That says OK, while the loop is parallel, there are some dependences. That means some value generated here is used somewhere here. So you can run it parallel, but you have some communication going too. So you had to move data. So it's not completely running parallel, there's some synchronization going on. But you can get large chunk running parallels.

So we kind of focus on dual loops today, and let's look at this example. We see it's a for far so it's a parallel loop or for all loop. When you know it's parallel, in here, of course, the user said that. What we can do is we can distribute the iteration by chunking up the iteration space into number of process chunks, and basically run that. If PMD mode, you can at the beginning the first processor can calculate the number of iterations you can run on each process in here, and then you synchronize, you put a barrier there, so everybody kind of sync up at that point. Or other process of waiting, and at that point, everybody starts, when you reach this point it's running, it's part of iterations, and then you're going to put a barrier synchronization in place. Kind of obvious, parallel code basically in here, running on shared memory machine at this point.

So this is what we can do. I mean this is what we saw before. Of course, instead of doing that, you can also do fork join types or once you want to run something parallel, you can fork a thread and each thread gets some amount of iterations you run, and after that you merge together. So you can do both. So that's my hand. How do you do something like that by the compiler? That sounds simple enough, trivial enough. But you don't automate the entire process. How to go about doing that. So, here are some normal loops, for loops. So the for all does this thing that was so simple, which is the for all construct that means somebody could look at that and said this loop is parallel. But you look at these FOR loops, how many of these loops are parallel? Is the first loop parallel? Why? Why not?

AUDIENCE: [OBSCURED.]

PROFESSOR: It's a loop because the iteration, one of that is using what you wrote in iteration zero. So iteration one has to wait until iteration zero is done, so and so. How about this one? Why?

AUDIENCE: [NOISE.]

PROFESSOR: Not really. So it's writing element 0 to 5, it's reading elements 6 to 11. So they don't overlap. So what you read and what you write never overlap, so you can keep doing it in any order, because the dependence means something you wrote, later you will read. This doesn't happen in here. How about this one?

AUDIENCE: There's no dependence in there.

PROFESSOR: Why?

AUDIENCE: [OBSCURED.]

PROFESSOR: So you're writing even, you're reading odd. So there's no overlapping or anything like that. Question? OK. So, the way to look at that -- I'm going to go a little bit of formalism. You can think about this as a iteration space. So iteration is if you look at each iteration separately, there could be thousands and millions of iterations and your compiler never [COUGHING] doing any work, and also some iteration space is defined by a range like 1 to n , so you don't even know exactly how many iterations are going to be there. So you can represent this as abstract space. Normally, most of this loops you look at you normalize to step one. So what that means is all the integer points in that space. So if you have a loop like this, y equals 0 to 6, J equals $1i$ to 7. That's the iteration space, there are two dimensions in there. The points that start iteration off because it's not a rectangular space, it can have this structure because j 's go in triangular in here.

So the way you can represent that is so you can represent iteration space by a vector i , and you can have each dimension or use two dimension. This was some i_1 , i_2 space in here. So you can represent it like that. It's the notion of lexicographic ordering. That means if you execute the loop, what's the order you're going to receive this thing. If you execute this loop, what you are going to do is you go from - - you go like this. This is lexicographical ordering of everything in the loops. That's the normal execution order. That's a sequential order. At some point you want to make sure that anything we do kind of has a look and feel of the sequential lexicographical order.

So, one thing you can say is if you have multiple dimensions, if there are two iterations, one iteration lexicographical and another iterations says if all outer dimensions are the same, you go to the first dimension where the numbers, they are in two different iterations. Then that dictates if it's lexicographical than the other. So if the outer dimensions are the same, that means the next one decides, the next one decides, next one decides going down. First one that's actually different decides who's before the other one.

So another concept is called affine loop nest. Affine loop nest says loop bounds are integer linear functions of constants, loop constant variable and outer loop indices. So that means if you want to get affine function within a loop, that has to be a linear function or integer function where all the things either has to be constant or loop constants -- that means that that variable doesn't change in the loop or outer loop indices. That makes it much easier to analyze. Also, array axes, each dimension, axis function has the same property.

So of course, there are many programs that doesn't satisfy this, for example, if we do FFD. That doesn't satisfy that because you have exponentials in there. But what that means is at 50, there's probably no way that the compiler's going to analyze that. But most kind of loops fit this kind of model and then you can put into nice mathematical framework and analyze that what I'm going to go through is kind of follow through some of the mathematical framework with you guys.

So, what you can do here is if you look at this one, instead of representing this iteration space by each iteration, which can be huge or which is not even known at compile time, what you can do is you can represent this by kind of a bounding space of iterations, basically. So what this is, we don't mark every box there, but we say OK, look, if you put these planes -- I put four planes in here, and everything inside these planes represent this iteration space. That's nice because instead of going 0 to 6, if you go 0 to 60,000, still I have the same equation, I don't suddenly have 6 million data points in here I need to represent. So, my representation doesn't grow with the size of my iteration space. It grows with the shape of this iteration space. If you have complicated one, it can be difficult.

So what you can do is you can iteration space, it's all iterations zero to six, j's I27. This is all linear functionns. That makes our analysis easier. So the flip side of that is the data space. So, if m dimension array has m dimensional discrete cartesian space. Basically, in the data space you don't have arrays that are odd shaped. It's almost a hypercube always. So something like that is a one dimensional space and something can be represented as a two dimensional space in here. So data space has this nice property, in that sense it's a t multi-dimensional hypercube. And what that gives you is kind of a bunch of mathematical techniques to kind of do and at least see some transformations we need to do in compiling.

As humans, I think we can look at a lot more complicated loops by hand, and get a better idea what's going on. But in a compiler you need to have a very simple way of describing what to analyze, what to formulate, and having this model helps you put it into a nice mathematical frame you can do.

So the next thing is dependence. We have done that so I will go through this fast. So the first is a true dependence. What that means is I wrote something, I write it here. So I really meant that I actually really use that value. There are two dependences mainly because we are finding dependence on some location, is an anti-dependence. That means I can't write it until this read is done because I can't destroy the value. Output dependence is there, so ordering of writing that you need to maintain.

So in a dynamic instance, data dependence exist between i and j if Either i and j is a write operation, and i and j refers to the same variable, and i executes before j. So it's the same thing, one execute before the other. So it's not that you don't have a dependence when they get there in time, then it become either true or anti. So it's always going to be positive over time.

So how about other accesses? So one element, you can figure out what happened. So how do you do dependence and other accesses? Now things get a little bit complicated, because arrays is not one element. So that's when you go to dependence analysis. So I will describe this using bunch of examples. So in order to look at arrays, there are two spaces I need to worry about. One is the iteration space, one is the data space. What we want to do is figure out what happens at every iteration for data and what other dependences kind of summarize this down. We don't want to look at, say OK, one iteration depend on second, two depend on third -- you don't want to list everything. We need to come up with a summary -- that's what basically dependence analysis will do.

So if you have this access, this is this loop. What happens is as we run down, so iterations we are running down here. So we have iteration zero, 1, 2, 3, 4, 5. First do the read, write, read, write. So this is kind of time going down there. What you do is

this one you are reading and you are writing. You're reading and writing, so you have a dependence like that. You see the two anti-dependence. Read -- anti-dependence, I have anti-dependence going on here. If you look at it, here's a dependence vector. What that means is there's a dependence at each of those things in there -- that's anti-dependence going on.

One way to look at summarizes of this, what is my iteration. My iteration goes like -- what's my dependence. I have anti-dependence with the same iteration, because my read and write has to be dependence in the same iteration. So this is a way to kind of describe that.

So a different one. This one. I did $A_i + 1 = A_i$. So what you realize is iteration zero, you wrote iteration zero, you wrote a zero, you read these and you wrote A_1 , and iteration 1, you read A_1 and wrote A_2 , basically. Now what you have is your dependence is like that, going like that. So if you look at what's happening in here, if you summarize in here, what you have is a dependence going like that in iteration space. So in iteration that means iteration 1 is actually these two dependence, that uses something that wrote iteration zero, iteration 2 you have something iteration 1, and you have iteration going like that. Sometimes this can be summarized as the dependence vector of 1. Because the previous one was zero because there's no loop carry dependency. In the outer loop there's a dependence on 1. So if you have this one, $I + 2$, of course, it gets carried 1 across in here and then you have a 1 skipped representation in here. If you have $2I + 2$ by plus 1, what you realize is there's no overlap. So there's no basically dependency. You kind of get how that analytic goes.

So, to find data dependence in a loop, so there's a little bit of legalese. So let me try to do that. So for every pair of array accesses, what you want to find is is there a dynamic instance that happened? An iteration that wrote a value, and another dynamic instance happened that later that actually used that value. So the first access, so there's a dynamic instance that's wrote, or that access, and another iteration instance that also accessed the same location later. And one of them has to be right, otherwise there are two in anti. That's the notion about the second one came after the first one. You can also look at the same arrays. It doesn't have to be the same as different access, the same array access if you are writing. If you look at same array access writing you can have output dependences also. So it's basically between a read and a write, and a write and a write. Two different writes, it can be the same write too.

Key thing is we are looking at location. We're not looking at value path and say who's actually in the same location. Loop carry dependence means the dependence cross a loop boundary. That means the person who read and person who wrote are in different loop iteration. If it's in the same iteration, then it's all local, because in my iteration I deal with that, I moved data around. But what I'm writing is used by somebody else in different iteration, I have loop carry dependence going on. Basic thing is there's a loop carry dependence, that loop is not parallelized in that. What that means is I am writing in one iteration of the loop and somebody is reading in different iteration of the loop. That means I actually had to move the data across, they can happen in parallel. That's a very simple way of looking at that.

So, what we have done is -- OK, the basic idea is how to actually go and automate this process. The simple notion is called a data dependence analysis, and I will give you a formulation of that. So what you can formally do is using a set of equations.

So what you want to say is instead of two distinct iterations, one is the write iteration, one is the read iteration. One iteration writes the value, one iteration reads the value. So write iteration basically, writes a item loop plus 1, the read iteration reads A_i . So we know both read and write have to be within loop bound iteration, because we know that because we can't be outside loop bounds.

Then we also want to make sure that the loop carried dependence, that means read and write can't be in the same iteration. If it's in the same iteration, I don't have loop carry dependence. I am looking for loop carry dependence at this point. Then what makes both of the read and write write the same location. That means access 1 has to be the same. So the right access point is i_w plus 1, and read access function is $[? \text{ IEI. } ?]$ So the key thing is now we have set up equation. Are there any values for i_e and j , integer values, I'm sorry, i_w and i_r that these equations are true. If that is the case, we can say ah-ha, that is the case, there's an iteration that the write and read are writing into two different iterations -- one write iteration, one read iteration, writing to the same value. Therefore that's a different [OBSCURED]. Is this true? Is there a set of values that makes this true?

Yeah, I mean you can do i_r equals 1, i_w equals 1, and i_r equals 2. So there's a value in there so these equations will come up with a solution, and at that point you have a dependency.

AUDIENCE: [NOISE]

PROFESSOR: So that's very easy to make this formulation. So if the indices is calculated with some thing or loop value, I can't write the formulation. So the data that I can do this analysis is this indices has to be the constant or indefinite. This is A of b of i . So if my array is A of b of i , I don't know how the numbers work if you have A of b of i . I have no idea about A_i is without knowing values of B of i . And B of i , I can't summarize it. Each B of i might be different and I can't come up with this nice single formulation that can check out every B of i . And I'm in big trouble. This is doable, but this is not easy to do like this. Question?

AUDIENCE: [NOISE]

PROFESSOR: Yeah, that's right. So that the interesting thing that you're not looking at. Because when we summarized it, because what you are going to do is we are trying to summarize for everything, every iteration, and we are not trying to divide it into saying OK, can I find the parallel groups. Yes. You can do some more complicated analysis and do something like that. Yes.

So other interesting thing is OK, the next thing you want to see whether can find output dependence. OK, are there two different iterations that they're fighting the same thing. What that means is the iterations are I_1 , I_2 , and I_1 not equals I_2 , and I_1 plus 1 equals I_2 plus one. There's no solution to this one because the I_1 has to be equal to I_2 according to this, and I_1 cannot be equal to I_2 during this one. That says OK, look, I don't have output dependence because it can be satisfied. OK, so here I know I have a loop carried -- I haven't said the two anti depends on which directions this is. Two anti-dependents, but I don't have a loop carried out to [OBSCURED].

So how do we generalize this? So what you can do is as integer vector I , so in order to generalize this, you can use integer programming. How many of you know integer programming or linear programming? OK. We are not going to go into detail, but I'll

tell you what actually happen. So integer programming says there's a vector of variable I , and if you have a formulation like that, is array, AI is less than or equal to B , A and B are all constant integers, and you can use the integer programming, you can see that there's a solution for IE or not. This is if you do things like operations research, there's a lot of work around it. People actually want to know what value is Y . We don't care that much what values, we just want to know the solution or not. If there's a solution, we know that there's a dependent. If there's no solution we know there's no dependent.

So we need to do is we need to get this set of equations and put it on that form. That's simple. For example, what you want is AI less than B -- that means you have constant $A_1 I_1$, plus $A_2 I_2$, which is less than or equal to B . So you won't have this kind of a system. Not equals doesn't really belong there. So the way you deal with not equals if you do it in two different problems. You can say IW less than IER is one problem, and W is greater than IER is other problem, and if either problem has a solution, you have a dependence. So that means one is true and one is anti. You can see the true dependence or anti-dependence, you can look at that.

This one is a little bit easier. This is less than, not actually less than -- less than equal. How do you deal with equal? So the way you deal with equal is you write in both directions. So if A is less than B , A less than or equal to B , B is less than or equal to A means actually is equal to B . So you can actually try two different inequalities and get equal to down there. So you have to kind of massage things a little bit in here. So here are our original iteration bounds, and here's our one problem because we are saying write happens before read, so these are two dependents that we are looking at. This is saying that write location is the same as the read location and this is equal, so I have two different equations in here. So kind of massage this a little bit to put it in i form, and we can come up with A 's and B 's. These are just manual steps, A 's and B 's, and now we are going to throw it into some super duper integer linear program package and it will say yes or no and your set.

And of course, you had to do another problem for the other side. You can generalize it for much more complete loop nest. So if you have this complicated loop nest in here, you had to solve you've got n deepness, you have to solve two end problems with all these different constraints. I'm not going to go over this. I have the slides in here. So that's the single dimension.

So how about multi-dimension dependences? So I have two dimensional iteration space here, and I have I, J equals AI, J minus 1. That's my iteration space. What does my dependence look like? We have arrows too. Which direction are the arrows going?

AUDIENCE: [OBSCURED]

PROFESSOR: We have something like this. Yup. We have something like this because that's J minus 1, the I 's are the same. Of course, if you have the other way around, go other direction, one is anti and one is it two dependence, so you can figure that one out. And do something complicated. First one. So IJ, I minus 1, J plus 1. Which has to be diagonal. Which diagonal does it go? This way or this way? Who says this way? Who says this way? So, this is actually going in this direction. This is where you have to actually think which iteration is actually write and read in here. So things get complicated.

This one is even more interesting. This one. There's only one dimensional array or two dimensional loop nest. So what that means is who's writing and who's reading? If you look at it basically -- actually this actually is a little bit wrong, because the dependence analysis says -- actually, all these things, all this read has to go into all the write, because they are writing any J, just writing the same thing. So this is a little bit wrong. This is actually more data flow analysis. This is a different -- their dependence means I don't care who the guy wrote, because he's the last guy who wrote, but everybody's reading, everybody else is writing the same location.

AUDIENCE: [OBSCURED].

PROFESSOR: Keep rewriting the same thing again and again and again. You start depending on -- It's not dependant on J's it's dependant on I. But location says you used to have iterations right in the same location, different J. So not matter what J, it's writing in the same location. You know what I'm saying? Because J thinks J.

AUDIENCE: [NOISE].

PROFESSOR: This is iteration space. I am looking at iteration. I am looking at I and J.s

AUDIENCE: [OBSCURED].

PROFESSOR: B is a one dimensional array. So B is a one dimensional array. So what that means is -- The reason I'm saying it's the iteration space and array space is a match. I'll correct this and put it in there because this is a data flow diagram. It's row independant. This one writing to what?

AUDIENCE: [OBSCURED].

PROFESSOR: Iteration space is I and J. So, this is writing to what? I zero is -- This is writing to what? B1. All those things are writng to B1. This is really -- So this is writing to B1, this is reading B zero. So this iteration is reading B1 again. So this was B1, this is iteration B1. So each of these is writing to B1, each of these are reading from B1, so each has to be dependent from each other.

AUDIENCE: So I guess one thing that's confusing here is why isn't it just -- why don't we just have arrows going down the column? Why do we have all these--?

PROFESSOR: Arrows going down the column means each is trying to do different location. So what happens is that this one, arrays going down this way. Is this one -- what's wrote here is only that location, only this side I accidentally located. These are all writing to the same location and reading from the same location.

AUDIENCE: Why isn't B iterated?

PROFESSOR: This is iteration space. I have two different loops here.

AUDIENCE: But I don't understand why B [NOISE.]

PROFESSOR: This is my program. I can write this program. This is a little bit of a stupid program because I am kind of trying to do the same thing again and again.

But hey, my program doesn't say array dimensions has to match your loop dimension. It doesn't say that so you can have programs like that. You can have other way too. So the key thing is to make -- don't confuse iteration space versus array space. They are two different spaces, two different number of dimensions. That's all the point that I'm going to make here.

So by doing dependence analysis, you can figure out -- now you can formulate this nicely -- figure out where the loops are parallel. So that's really neat. The next thing I'm going to go is trying to figure out how you can increase the parallelism opportunities. Because there might be cases where the original code you wrote, there might be some loops that are not parallelizable, arrays, and can you go and increase that. So I'm going to talk about few different possibilities of doing that.

Scalar privatization, I will just go in each of these separating. So here is interesting program. To get parallel to the temporary and use the temporary in here. You might not know you had written that but the compiler normally generates something like that because you always had temporaries in here, so this might be what compiler generate. Is this loop parallel?

AUDIENCE: Yup.

PROFESSOR: Why?

AUDIENCE: [OBSCURED].

PROFESSOR: Is the loop carry dependence true or anti -- What's the true dependence which to which? We didn't loop true dependence. What is the loop carry dependence? Anti-dependence. Because I cannot -- you see I equal 1, basically wrote here in this reading. I can't write I equals 2x until I equals 1 is done and done reading that. I have one location and everybody's trying to read or write that, even though I don't really use data. This is the sad thing about this. That I'm really using this guy's data, but I'm just waiting for the same space to occupy. So, there's a loop carry dependence in here, and it's anti-dependent. So what you can do is if you find any anti or output loop carry dependence, you can get rid of them. I'm not really using that value, I'm just keeping a location in here. So how can we get rid of that?

AUDIENCE: [OBSCURED].

PROFESSOR: Yeah. That's one thing. There's two ways of doing it. One is I assign something local. So each processor will have its own copy, so I don't do that. So it's something like this, so that's [OBSCURED]. Or I can look at the array. In the array you can have either number of process or iterations for each iteration. But uses a different location. This is more efficient than this one because we are touching lot more locations in here. I haven't done one thing here. I'm not complete. What have I forgotten to do in both of these?

AUDIENCE: [OBSCURED].

PROFESSOR: Yeah, because it was beforehand somebody might use final assignment of the loop nest, so what you had to do is you had to kind of finalize x. Because I had a temporary variable, so with n, the last value has to go into x. You can't keep just not calculating value in something. So in here, also, you just say last value is x. But after you do that, basically now each of this loop is faster. Everybody go that?

OK, here's another example. x equals x plus AI . Do I have loop carry dependent? What did the loop-carried dependence? True or anti? True dependence. So this guy is actually creating previous value and adding something in the event. So of course in true dependence I cannot seem to parallelize. But there are some interesting things we can do. That was an associative operation. I didn't care which order this initial happened, so I'm just keeping a lean bunch of values in here. And the results were never used in the other loop. So we just keep adding things and at the end of the loop you get the sum total in here. I never used any kind of partial values anywhere. So that gives the idea. So what you can do is we can translate this into each of the guys doing a temporary addition into its own variable.

So each processor, just do a partial sum. At the end, once they're done, you basically do the full sum. Of course, you can do a tree or whatever much more complicated thing than that -- you can also parallelize this part at the tree addition. But you can do that. I mean Roderick talked about this in hand parallelization. But we are doing something very simple in here. So these compilers can figure out associative operations and do that. So this is where all the people who are in parallelizing, and all the people who are writing this scientific code kind of start having arguments. Because they say oh my God, you're doing operations and it's going to have numerical stability issues. Yes all true. In compilers you have these flags that say OK, just forget about all these very issues, and most probably it will be right, and in most code it will work. You might find that problem, too -- you change operation order to get some parallelism and suddenly you are running unstability. There are some algorithms that you can't do that, but most algorithms you can.

So here's another interesting thing. So, I have a program like that, 2 to the power I , and of course, most of the time exponentiation is very expensive. If you have a smart compiler -- I don't have to exponentiate. This thing called strength reduction. They say wait a minute -- I will keep variable t . This 2 to the power i means basically every time I multiply it by 2 and I can't keep repeating that. Do you see why these two are equal there? This is good. A lot of good compilers do that. But now what did I suddenly do?

AUDIENCE: [OBSCURED.]

PROFESSOR: Yeah, I reduced the amount of computation, obviously, but I just introduce a loop-carried true dependence here. Because now I have t dependent on the previous t to calculate the next value, and while order-wise or sequential-wise this is a win, now suddenly I can't parallelize. Of course, a lot of times what you had to do is you have a very smart programmer. They say aha, I know this operation is expensive so I am going to do this myself and create you a much simpler program in sequentially. Then you try to parallelizes this and you can't. So what you might try to do is kind of do this direction transformation many times to make the program run a little bit slower sequentially just so you can actually go and parallelize it. So this get's a little bit counterintuitive. You just look at a program and say yeah there is a loop carried dependence, I can do it a little bit more expensive without the loop carried dependence, and then suddenly my loop is parallelized. So there might be cases where you might have to do it by hand, and a lot of compilers automatic parallelizing compilers, try to do this also. Kind of look at these kind of things and try to move in that direction. Whereas, most of the sequential compiler is trying to find this and move this direction.

So, OK I said that. So, another thing called array privatization. So scalars, I show you where when you have anti and output dependence on a variable, you need to privatize. And in arrays, you have a lot more complexity. I'm not going to go into that, you can actually do private copies also in there. You can do bunch of transformation.

Another thing people do is called interprocedural parallelization. So the thing is you have a nice loop and you start analyzing loop and in the middle of a loop you have a function call. Suddenly what are you going to do with it? You have no idea what the function does, and most of the simple analysis says OK, I can't parallelize anything that has a function call. That's not a good parallelizing compiler because a lot of loops have function calls and you might call it something simple as sine function or some simple exponentiation function and then suddenly it's not parallelizable. This is a big problem. There are two things you can do. One is interprocedural analysis and another inlining. So the interprocedural analysis says I'm going to analyze the entire program and I have function, I'm going to go and try to analyze the function itself also. What happens is -- so assume if the functions are used many, many times, so fine function might be used hundreds of time.

So every time you have a call of a sine function, if you keep analyzing, reanalyzing what's happening inside of the sine function, you kind of have exponential blow up. So if you code size n , you might have an exponential time of a number of lines that need to be analyzed because every call need to go there, call some other functions, you can see the blow up. And so analysis might be expensive. Other option is you analyze each function once. Yeah, OK. I analyze this function once and every time I use that function I just use that analysis information. What that means is you have a kind of summary of what that function does for every call. This is not that easy and this runs into a thing called unrealizable part problem, because you go into function in one part -- assume you call foo from here and return here. You call it here and return and here. So when you analyze, normally you can go from here to here, here to here, but if you treat foo as only one thing you might be able to even think that you can go here to here and here to here. So this looks like one thing in here. You see that control here goes here, comes here do a function call goes here, because we are not treating this as separate instance.

So why did are we analyzing it once? This cleared all this additional mess and then can have problems in here. So these are the kind of researchy things people are working on. There's no perfect answer, these are complicated problems, so you had to do some interesting balance in here. Because other thing is every analyst has to deal with that, so you had to kind of an anti-compiler, which is not simple.

Inlining is much more easy. It's a poor man solution, so every time you have function call, you just bring the function and just copy it in there. And every time you have function call you bring the function and you can run it through the same compiler, but of course, you can have huge code blow up. It's not only analysis expense, you might have a function that before had only 100 lines, now we have millions of lines in there and then try and do cache problems, all those other issues. So can be very expensive too. So what people do is things like selective inlining and a lot of kind of interesting combinations of these.

Finally, loop transformations. So i have this loop, so I have A_{ij} equals A_{ij} minus 1, A_{i-1} So look at my -- my arrowheads look too big there, but look at my dependences. Is any of this parallel?

AUDIENCE: [OBSCURED.]

PROFESSOR: Yeah. So, arrays neither -- you can't parallelize I because there's a loop carry dependence in I dimension. You can't parallelize J because there's loop carry dependence in J dimension. She has idea because you can actually pipeline. So pipelining, we haven't figured out how to parallelize pipeline. So the way you can do this simply is a thing called loop skewing. You can kind of -- because iteration space has changed from a data space. You can come up with a new iteration space that kind of skew the loop in there. So what it does is normally iteration space, what this J outside, so you go execute like this. The skill that -- loop basically say I am executing this way, so I'm executing the pipeline, basically pipeline here. So I'm kind of going like this way, executing that way. If I could run that loop in that fashion, what I can do is I can run this -- after this iteration, when you go run the next iteration, there's no dependence across here. If I run here, I don't have dependence, so I can run each of these and I have a parallel set of iterations to run. So in here, what happens is this inner loop it can be parallel, basically like your pipeline, but it's written in a way that I still have my two loops in here, but I have done this weird transformation.

Another interesting is granularity of parallelism. Assume I have a loop like that, i and j. Which loop is that in here? i or j? j is parallel. OK, I do something like that. I say I run i, every iteration I do a barrier, I run j parallel and I end up doing a barrier again. What might be a problem in something like this?

I mean inner parallelism can be expensive, because every time I had to do this probably expensive barrier, run a few iterations, a few in this one, probably only like a few cycles. And write this very expensive barrier again, and everybody communicates -- all of those things. Most of the time when you do inner loop parallelism it actually slows down the program. You will probably find it too sometimes, if you define the parallelism inner array to be too small, it actually has a negative impact, because all the communication you need to do, synchronization you need to do all of them out of the program.

So inner loop is expensive. What are your choices? Don't parallelize. Pretty good choice for a lot of cases. You look at this and this is actually going to win you basically by doing that. Or can you transform it to outer loop parallelism. Take inner loop parallelism and you change it to get outer loop parallelism. This program is actually nice, there are some complex analysis you need to do to make sure that's legal. So you can basically take this one and transform in other direction. What that means is kind of do a loop interchange. So now instead of i, you have a j outer dimension, i inner dimension, inner loop. When you do that what you have is your barrier, and then you can run this is parallel and this like this. Suddenly, instead of having n barriers for that loop, you have only one barrier. Suddenly you have a much larger chunk you're running, and this can be run.

OK, so this is great. So I talked to all about all this nice transformation, stuff like that. So at some point when you know something is parallel you might want to go and generate parallel form. So the problem is, depending on how you partition, the loop bound has to be changed, and I'm going to talk to you about how to get loop bound. So let's look at this program. So I have something in here and there's an inner loop that actually reads, outer loop writes. Inner loop reads. And it's a triangular thing. It's a big mess. Now I assume I want to run the i loop parallel. So

what that means is I want to run the first process -- there is no for this one, this one on one iteration, two iteration, three, four, whatever, each one's in here. How do I actually go about generating code that actually does that? Each processor runs its right number of iteration. This is a non-trivial thing because triangularly you get something different and you can assume all this complexity. One thing I did is my iteration space between i and j , this is my iteration space. So I assume, assume I am running a processor. Each I iteration run by your processor, you can say you have then another dimension P , and say i equals P . So I can look at now instead of a two dimensional space in a three dimensional space. So in this analysis, if you can think multi-dimensionally it's actually very helpful because we can kind of keep adding dimensions in here.

So what are the loop bounds in here? What we can do is use another technique called Fourier-Motzkin Elimination to calculate loop bounds by using projections of the iteration space. I will go through later a bit to give you a flavor for what it is. It's also, if you are in to linear programming, this is kind of extension techniques on that. So the way we look at that is -- A little bit too far. I didn't realize MAC can be this slow.

[ASIDE CONVERSATION] See this is why we need parallelism if you think this running fast. So what you can do is you can think about this as this three dimensional space. i , j and p . And because i is equal to p , if you get i and p , get a line in that dimension and then j goes there. So this is the kind of iteration space in here, and that represents inequalities here.

So what I want is a loop where outer dimension is p , then the next dimension is i and j . We can think about it like that. So what that means is I need to get my iteration ordering -- when it happens, you just go like that. All right, about doing that. So this is the kind of loop I want to generate -- let me go and show you how we generate that. So here's my space in here, so first one I want to do is my inner most dimension is j . And what I can do is I can look at this thing and say what are the bounds of j . So, for each of the bounds of j can be described by -- with p and i . I'll actually show you how to do that in little while. Then I will get j goes from 1 to i minus 1. Then after that I can basically project it into to eliminate j dimension.

So what I'm doing is I'm going to have a three dimension and I project into two dimensions without j anymore, because now all I have left is i p and I get a line in that dimension. Then what I have to do is now I had to find i . What are my bounds of i ? And bounds of i is actually i is equal to p . You can figure that one out because there's a line in there. Then you eliminate i and now you get this one. Then what are bounds of p ? p goes from basically 2 to n . You just basically get that. So you can do this projection in here -- let me go in there, and now what you end up doing is you can get this, and of course, outer loop p is not a true -- like a loop. You can say you get p , my pid . p is with this range. i equals p . Do this one. So this one, -- generated that piece of code. So I will go a little bit detail and show how this happens, pretty much can happen. So I have my little bit of different space. I'm doing a different projection. I'm doing i , j , p . I want to predict first i of a , j of a , and p of a instead of j , i , p before I do anything.

So here's my iteration space, what do I do? The first thing I do is I find the bounds of i , So I have this thing. I just basically expanded this, and eliminated the j this one doesn't contribute to the bounds of i , but everybody else. So there are a bunch of things that i has to be less than that and i have to be greater than these two. Then

what I have is bound of i is, it has to be maximum of this because it has to be greater than all three. So it has to be max of this, this, and this. It has to be less than these two, it has to be mean of this one. Question?

AUDIENCE: Well why did you have to go through all this. At least in this case, the outer loop was very simple, you could have just directly mapped that.

PROFESSOR: I agree with you, it's very simple thing, but the problem is that's because you are smart and you can think a little bit ahead in there, and if I'm programming a computer, I can't say find these special cases. So I want to come up with a mathematical way that is a bullet proof way that will work from the simplest one to very complicated, like for example, finding the loop bounds for that loop transpose that I showed you before -- no, the skew that what we called before.

AUDIENCE: So it's not so much just defining an index to iterate on, it's to find the best index to map, to parallelize.

PROFESSOR: Any could be issue, because you have -- for example, if the inner dimension depends on i , and i goes outside, then I can't make it depend on i . So if I have something like for i equals something, for j equals i to something. Now if I switch these two I have $4j$. I can't say it's i to something. I have to get rid of i and I have to figure out in the for i , this has to be something with j , with some function with j in here. So what is this function, how do you get that? You need this kind of transformations do that. Next time I'll talk to you about can you do it a little bit even better.

So I get this bound in here. Then actually you found this is going from p to p . So I can actually set p because, mean and max in here. Then after you do that, what you have to do is eliminate i . The way you eliminate i is you take this has to be always less than n and less than p . So you take this n constraints here and you get a n times m constraints tier in here. So the first three has to be less than n , again, we repeat it again, has to be less than p . Then, of course, the missing constraint that 1 is less than j . You put all those constraints together. Now, nice think is in that one, it's still legal, it still represents that space, but you don't have i there anymore. You can completely get rid of i .

So, by doing that -- and then of course, there's a lot of redundancy in here, and then you can do some analysis and eliminate redundancy and you end up in this set of constraints. That's where when you say what's the best, you can be best -- it has to be correct or that means you can't have additional iterations or less iterations. But best depends on how complicated is the loop bound calculation. You can come up with a correct solution, and the best is depending on which order you do that. When you have two redundant thing, which one you eliminate, so you can have a lot of heuristics saying OK, look if this one looks harder to calculate, eliminate that one with the other one. So you get this set of constraints.

Then you have to do is now find the bounds of j . So you have this set again. To find a bound of j only two constraints are there, and you know j goes to 1 to p minus 1 , and you find the bound of j . Getting rid of j means there's only two. One get rid of p minus 1 . There are two left for p . You put it there, and then you can eliminate the redundancy in here, and now you can find the bounds of p which goes from 2 to n . And suddenly you have the loop nest. So now I actually do parallelization and a loop

transpose in here. I could combine those two, use this simple mathematical way and find loop bounds in here.

So, I'm going to give you something even a little bit interesting beyond that, which is communication code generation. So if you are dealing with a cache coherent shared memory machine, you are done. You generate code for parallel loop nest, you can go home because everything else will be done automatically. But as we all know in something like Cell, if you have a no cache coherent shared memory or distributed memory, you have to do this one first. Then you write identify communication and then you generate communication code. This have additional burden in here.

So until now in data dependence analysis, what we looked at was location-centric dependences. Which location is written by processor one is used by processor two. That's kind of a location-centric kind of view. How about if multiple writes the same location? We show that in example, if multiple people write the same location, which one should I use? That's not clear. What you are using in the last last guy who wrote that location before I read that thing, and that's not in these data flow analysis. No data dependence analysis doesn't get it. What you want is something of a value-centric. Who was the last write before my iteration, who wrote that location? If I know the last write, he's the one I should be getting the value from. If the last write happened in the same processor, I am set because I wrote the local copy and I don't need to deal with anything. If the last write happened in a different processor, you need to get that value from the guys who wrote it and say, OK, you wrote that value, give it to me. If nobody wrote it and I'm reading it, that means the value came from the original array because nobody had written it in my iteration. Then I'm reading something that has come from the previous iteration. So I have to get it from the original array. But I have these three different conditions.

So you know to represent that. I'm not going to go into detail on into detail on this representation called Last Write Trees. So what it says is in this kind of a loop nest in here, you have some read access and write accesses in here, and if you look at it location-centrally you get this entire complex graph, because this is the graph that should have been in that example we gave. So these arrays going in here. I'm switching notation. before i was going the other way around. j was in here. But if you go look at value-centric, this is what happens. So you say all these guys basically got the value from outside. Nobody wrote it. This got from -- this is the write, this is the last write, this is the last write -- I actually have my last write information. So where to look at that is there are some part of iteration got value from somewhere, other part go somewhere else. You can't kind of do a big summary, as you point out that kind of dependence depend on where the iterations are. So you can represent it using a tree when it shows up. So you can say if j greater than 1, here's the relationship between reads and writes. Otherwise relationship means it came from outside. So I can say for each different places. So you can think about this tree can be a lot more complicated tree. So each part of the iteration space, I got data from somewhere else.

So, you get this function here. I think I'll go to the next slide. So what you can do is now I have processor who read, processor who write, and iterations that I can reading and writing. One thing I can do is I can represent i using a huge multi-dimensional space. So what happens in here is the receive iterations, those are the iterations that actually data has to be received in communication. Assume that the part I'm actually communicating is also within the loop bound, so I can write that. And the last write relation is that i send has to be i receive. We know that. What you

have is the parallel with the processors -- this is i iterations are parallel, so processor, receive processor, is running iteration i , process i . Send iterations are the same because you want to parallelize that loop basically. In each iteration get assigned to each process. Of course, you want to make sure the process communication is non-local. If it's local I don't have loop communication. I can represent this as this gigantic system of equalities. It has one, two, three, four, five, and there's a j receiver also in here, because you've got to remember I think the program I wrote, the original program basically, write happen in outer loop and the read happen inner loop. So there's only j receive, the i send in here. I'll show that later.

So I have five dimensions. So I can't really draw five dimensions, but can I wait until it comes back? So what I have here is I have this set of complete system of inequalities for receive and in communication. Of course, since I can't draw five dimensions, and these dimensions are the same, I just wrote it in the same. So you can actually assume that there's another two dimensions for this one, and that's a line in that dimension. Actually, this is wrong. Sorry. This should be x_i here written. My program is wrong, sorry. Now what do I do? One more time it has to go. It makes me slow down my lectures which is probably a good thing. There we go.

So what you can do is you can just scan these by predicting different ways to calculate the send loop nest and receive loop nest. So if you scan in that direction, what you end up is something saying for this processor you need to send, for this iteration, this processor. For what you need to send will be received by these processors and this iteration and this, and this you can send x_i to this iteration at this processor. Because you had that relationship, you can get the loop nest that actually will do the send. The send there you can actually get a loop nest do receive and it shows up.

So what that means is, so all these guys have to send all these iterations have to do the receive. So, if you predicted a different ordering, what you end up is you can say now for this processor has to receive. All these processors had to receive something send by these guys. So now you can get that entire loop nest for receiving and entire loop nest for sending, and you have computation loop nest also. The problem is you can't run them sequentially because you're run in some into the order. So what you have is something that next slide will show. So you have this iteration, there's some computation happen from all the one, and I will get a loop nest do some send, I need loop nest do some receive, in a one dimensional, these kind of, you get three separate things. But of course, what you had to do is you had to generate code. So the way to do that is --. So what you have to do is kind of break this apart into pieces where things happen, so this one you do computation, this one you do computation and receive, and computation send receive and whatever. Should be probably send here and receive but --

For that one, if you combine this you get a complicated mess like this. But this all can be done very in an automated fashion by using this Fourier-Motzkin Elimination and this linear representation. Of course, you can do a lot of interesting things on top of that. You can eliminate redundant communication, if you're keeping sending the same thing again that have a send unit, eliminate that, you can aggregate communication. You want to send a word at a time, you can send bunch of things into one packet. You can do multitask. So same thing, send to multiple people. Doesn't have that much in Cell, but assume some machines have multitask support, you can do that, and also you can do some local memory management because if

you have distributed memory, you don't have to allocate everybody's memory and only use a part. You can say OK, look everybody only had to allocate that part.

OK. In summary, I think automatic parallelism of loops and arrays -- we talked about data dependence analysis, and we talked about iteration and data spaces, a how to do that, and how the formulate array integer programming problem. We can look at lot of optimization that can increase parallelism and then do that. Also, we can deal with things like communication code generation and generating loop nest by doing this Fourier-Motzkin Elimination.

So what I want to show out of this talk is that, in fact, this parallelization -- automatic parallelization of normal loop can be done by mapping into some nice mathematical framework, and basically manipulating in that map. So there are many other things that really complicates the life take out of parallelizing programs. So like C, there are pointers, you have to deal with that. So this problem is not this simple, but what compiler writers try to do most of the time is trying to find this kind of thing. Find interesting mathematical models and do a mapping in there and then operating that model and hopefully you can get the analysis needed and even the transformation needed using that kind of a nice model.

So I just kind of gave you a good feel for general parallelizing compilers. We will take a ten-minute break and talk about streaming. We'll see if I can make this computer run faster in the meantime.