

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 9

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](http://ocw.mit.edu).

PROFESSOR: So, yesterday in the recitation we talked a little bit about how to debug programs on Cell. Today I'm going to talk a little more about debugging parallel programs in general and give you some common tips that might be helpful in helping you track down problems that you run into.

As you might have gotten a feel for yesterday, debugging parallel programs is harder than normal sequential programs. So in normal sequential programs, you have your traditional set of bugs which parallel programs inherit. So that doesn't get much harder, but then you add on new things I can go wrong because of parallelization. So things like synchronization, things like deadlocks, things like data races. So you have to get those right.

Now you have to debug your program and figure out how to do that. One of the things you'll see here is a lot of tools might not be as good as you'd like them to in terms of providing you the functionality for debugging. Add to that that bugs in parallel programs often just go away if you change one statement in your code -- you reorder things and all of a sudden the bug is gone. It's kind of like those pointer problems in C, where you might add a word, add a new variable somewhere and problem's gone, or you add a print-off and the problem is gone. So here it gets harder because those things can get rid of deadlocks, so it makes it really hard to have an experiment that you can repeat and come down to where the problem is.

So what might you want in a debugger. So this is a list that I've come up with, and if you have some ideas we'll want to throw them out. I'm thinking in terms of debugging parallel program, what I want is a visual debugging system that really let's me see all the processors that I have in my network in my multi-processor system. That includes actual computing and the actual network that they're interconnecting all the processors that are going to be communicating with each other. So I'd like to be able to see what code is running on each processor. I'd like to see which edges are being used to send messages around. I might want to know which processors are blocked -- that might help me identify deadlock problems.

For these kinds of scenarios it might be tricky to define step, because there's no global clock, you can't force everybody to proceed through one step. What's one step on one processor might be different on another, especially if they're not all running the same code. So how do you actually do that without a global clock. So that can get a little bit tricky. It likely won't help with data races, because I'm looking at global communication problems, I'm looking at trying to identify what's deadlocked and what's not. So if there are data races, this kind of tool may or may not help with that.

In general, this is the tool that I would build for debugging. I looked around on the web to see what's out there for debugging parallel programs, and I found this called TotalView. This is actually something you have to buy, it's not free. I don't know if they have evaluation licenses or licences for academic purposes. It kind of gets close to some of the things I was talking about. You have processors that shows your communication between those processors, how much data is being sent through. This particular version uses NPI, which we talked about in previous lectures. So it's sort of helpful in being able to see the computation, looking at the communication, and tracking down bugs.

But it doesn't get much better from there. You know, how many people have used printouts for debugging? It's the most popular way of debugging, and even I still use it for debugging some of the Cell programs we've been writing. I know the TAs actually use this as well. Yesterday you got a hands-on experience with GDB, and GDB is a nice debugger, but it lacks a lot of things that you might want, especially for debugging parallel programs. You saw, for example, that you have multiple threads, you need to be able switch between the threads, getting the context right, being able to name the variables is tricky. So there's a lot of things that could be improved.

There are some research debuggers, like something we've built as part of the streaming projects, StreamIt debugger. I'll show you some screenshots of this so you can see what we can do. So in the StreamIt debugger, remember we have -- so this is actually built in Eclipse and you can download this off the web as well. You can look at your stream graph. Unfortunately, I couldn't get a split join in there, much to Bill's dismay. So you can't see, for example, the split join in all the communication. Each one of these is a filter, and if you recall the filter is the computational element in your stream graph and interconnected by channels.

So channels communicate data. So what you see here -- well, you might not be able to quite see it -- actually see the data that's being passed through from one filter to the other. You can actually go in there and change the value if you wanted to or highlight particular value and see how it flows down through the graph. If you had a split join, then you might be able to -- in fact, you can do this. You can look at each path of the split join independently and you can look at it in sequence. Because the split join has nice semantics, it's actually you can replicate the behavior because of the static nature is everything is deterministic.

So this is very helpful. We we did a user study two years ago almost with something like 30 MIT students who use the C bugger and gave us feedback on it. So we gave them like 10 problems, 10 code snippets, each of them had a bug in it, we asked them to find it. So a lot of them found to debugger to be helpful in being able to track the flow of data and being able to see what goes wrong. So if you had, for example, a division that resulted in NaN and not a number, floating point division you can immediately see on a screen, so you know exactly where to go look for it. Doing that would print-offs might not be as easy. So sometimes visual debugging can be very nice.

Unfortunately, visual debugging for the Cell isn't that great. So this is the Cell plug-in in Eclipse. I've mentioned just to some of you if you want to run it you can run it from a Playstation 3, but if more than one of you is running it then it becomes unusable because of memory issues. You can install this on other Linux machines and remotely debug on the Playstation 3 hardware. So the two remote machines can

talk through GDB ports. I can talk to you about how to set that up if you want to, but it doesn't really add anything over E-Max, for example. It just might look fancier than an E-Max window or a GDB on the command line prompt.

So this is the code from yesterday. These are the exercises we asked you to do. You can look at the different threads. If you have debug Java programs in Eclipse this should look very familiar. You can look at the different variables. You still have the naming problem. So yesterday, remember you had to qualify which control box you were looking at? Still the same kind of issue -- have to do some trickery to find it here. It doesn't have the nice visual aspect of showing you which code is running on which SPE, and you might not be able to find mailbox synchronization problems. Maybe those things will come in the future, and the fact, they likely will. But a lot of that is sort of still lacking.

So what do you do in the meantime, In the next two weeks as you're writing your programs? So I've looked around for some tips or some talks and lectures and what people have done in terms of improving the process for debugging parallel codes. Probably the best thing I've found is this talk that was given at University of Maryland on defect patterns. So the rest of these slides are largely drawn from that talk. I'm going to identify just a few of them to give you some examples so you can understand what to look for and what are some common symptoms, what are some common prevention techniques.

So defect patterns, just like the programming patterns we talked about, are meant to help you conjure up to write contextual information. You had what are things you should look for if you're communicating with somebody else. What kind of terminology do you use so that you don't have to explain things down to every last detail. At the end of this course, one thing I'd like to do is maybe get some feedback from each of you as to what are some of the problems that you ran into in writing your programs, and how you actually went about debugging them, and maybe we can come up with Cell defect patterns, and maybe Cell defect recipes for resolving those defect patterns.

So, probably the worst one of all, and the easiest one to fix is that you have new language features or new language extensions that are not well understood. This is especially true when you take a class of students and they don't really know the language, they don't know all the tools, and you ask them to do a project in four weeks and you expect things to work. So there's a lot for everybody to pick up and understand.

So you might have inconsistent types that you use in terms of calling a function. There might be alignment issues, which some of you have run into. You might use the wrong functions. You know the functionality you want but you just don't know how to name it and so you might use the wrong function. Some of these are easy to fix because you might get a compile time error. If you have mismatch in function parameters then you can fix that very easily. Some defects -- you know, very natural parallel programs might not come up until run time, so you might end up with crashes or just erroneous behavior. I really think this is probably the easiest one to fix, and the prevention technique that I would recommend is if there's something you're unfamiliar about or you're not sure about how to use something, ask. But also, you don't need to know all the functions that are available in something like the Cell language extensions for C.

Yes, there are a lot of functions -- you know, the manuals, hundreds of pages, and you can't possibly go through it all and nobody becomes an expert in everything. But understand just a few basic concepts and features. So, David identified a bunch that he found useful for writing the programs, and some of the ones that are up on the web page under the recipes for this course list a few more. And so this might help you in just understanding how these functions work, understanding basic mechanisms that they give you, and that's good enough because it'll help you get by. Certainly for doing the project under short time constraints, you don't need to know all the advanced features that Cell might have. Or you can probably just pick them up on the fly as you need them.

So what are some more interesting problems that come up. I think one that is probably not too unfamiliar is this space decomposition problems. So, if you remember, space decomposition is really data distribution. You have a serial program that you want to parallelize. And what that means if you have to actually send data around to different processors so that each one knows how to compute locally. Here you might get things like segmentation faults, alignment problems, you might have index out of range errors. What this comes of is forgetting to change things or overlooking some simple things that don't carryover from the sequential case that are parallel case. So what you might want to do is validate your distributions and your memory partitions correctly. So what's an example?

So suppose you had an array or a list of cells, each cell has a number. What you want to do is at each step of the computation for any given cell, you want add the value to the left of it and the value to the right of it. So here we have cell zero has the value 2. We'll assume that the  $n$  disconnectors are first, so this is like a circular list, a circular buffer. So adding the left and right neighbor would get me, in this case, 3 plus 1, 4. And so on and so forth. You want to repeat this computation for  $n$  steps. So this might be very common in computations where you're doing unit [? book ?] communication.

So what's a straightforward sequential implementation? Well, you can use two buffers -- one for the current time step, and you do all the calculations in that. Then you use another buffer for next time step. Then you swap the two. So the code might look something like this. Sequential c code, my two buffers, here's my loop. I write into one buffer and then I switch the two buffers around.

Any questions so far?

So now, what are some things that can go wrong when you try to parallelize this? So how would you actually parallelize this code? Well, we saw in some of your labs, for example, that you can take a big array, split it up in smaller chunks and assign each chunk to one particular processor. So we can use that technique here. So each processor, we have  $n$  of them, rather size of them, and it's going to get some number of elements. So each time step, we have to compute locally all the communications, but then there's some special cases that we need to treat at the boundaries, right. So if I have this chunk and I need to do my new neighbor communication, I don't have this particular cells. I have to go out there and request it. Similarly, somebody has to send me this particular data item. So there's some data exchange that has to happen.

So in the decomposition, you write your parallel code. Here, each buffer is a different size. What you do is you have some local, which says this is how much of the data

I'm getting, and has a total number of elements, besides the number of processors. Local essentially tells me the size of my chunk. I'm iterating through from zero and local and I'm doing essentially the same computation. So what's a bug in here? Anybody see it? Sort of giving you some hints of things highlighted in red or there's something wrong with the things highlighted in red. There's another hint. So this is essentially the computations going on at every processor, this is my buffer, and at every step I have to do the calculations, taking care of the boundary edges. Anybody want to take a stab? Mark?

AUDIENCE: Is it that the nextbuffer zero needs to look at data from 1?

PROFESSOR: Next buffer zero, right. So what might be a fix to that? So the next buffer is zero. So if this is zero then buffer of  $x$  minus 1 plus this points to what?

AUDIENCE: So you need to start at 1 and iterate.

PROFESSOR: Right, exactly. It's a local plus 1, if you were going to do these. So that's one bug. The other thing is the assumption that your data elements might be divisible by the number of processors that you have. So you pick the decomposition that might not be symmetric across all processors. So it's more subtle, I think, to keep in mind. So that's one particular kind of problem that might come up on your decomposing data and replicating among different processors. So you have to be careful about what are your boundary cases going to be and how are you going to deal with those.

The more difficult one is synchronization. So synchronization is when you're sending data from one processor to the other and you might end up with deadlock, because one is trying to send, the other's trying to send, and neither can make progress until the other's received. So your program hangs, you get non-deterministic behavior or output, every time you run your program you get a different result -- that can drive you crazy. So some of the defects can be very subtle. This is probably where you'll spend most of your time trying to figure it out. So one of the ways to prevent this is to actually look at how you're orchestrating your communication and doing it very carefully.

So look at, for example, what's going on here. So this is the same problem, and what I'm doing is now this is the parallel version and I'm sending the boundary cases, the boundary cells to the different processors. This is an SPMD program. So an SPMD program has every processor running essentially the same code. So this code is replicated over  $n$  processors and everybody's trying to do this same thing. So what's the problem with this code? We're doing send of next buffer zero. Here, rank essentially just says each processor has a rank. So this is a way of identifying things. So I'm trying to send it to my previous guy, I'm trying to send it to the next guy, and here I'm sending the value at the far extreme of the buffer to the next processor and then to the previous processor. Anybody see what's wrong here?

AUDIENCE: So are these blocking things?

PROFESSOR: Yeah, imagine they're blocking things.

AUDIENCE: Why will that deadlock?

PROFESSOR: Right. So this will deadlock, right. So this will deadlock because this processor is trying to send here, this processor is trying to send here, but neither is receiving yet, so neither makes progress. So how would you fix it at this point? You might not want to use a blocking send all the time. So if your architecture allows you to have different flavors of communication, so synchronous versus an asynchronous, a blocking versus non-blocking, you'll want to avoid using constructs that can lead you to deadlock if you don't need to.

The other mechanism -- this was pointed out briefly in the talk on parallel programming -- you want to order your sends and receives properly. So alternate them. So you have a send in one processor, a receive in the other. You can use that to prevent deadlock and get the communication patterns right. There could be more interesting cases that come up if you're communicating over a network where you might end up with cyclic patterns leading to loops, and that also can create some problems for you.

The last two I'll talk about aren't really bugs in that they might not cause your program to break or compute incorrectly. Things might work properly, but you might not get the actual performance that you're expecting. So these are performance bucks or performance defects. So side effects of parallelization is often case that you're focusing on your parallel code and you might ignore things that are going on in your sequential code, and that might lead you to, essentially you've spent all this time trying to parallelize your code, but your end result is not getting the performance that you expect because things look sequential. So what's wrong here?

So as an example, imagine that we're doing instead of reading data from a -- so, in the previous case I didn't show you how we were reading data into the different buffers, but suppose we were getting it from some files, so input buffer. So now we have an SPMD program again, everybody's trying to read from this buffer. What could go wrong here? Anybody have an idea? So every processor is opening the file and then it's going to figure out how much to skip and it'll start reading from that location. So everybody's reading from a file, so that's OK, nobody's modifying it. But what can go wrong here?

AUDIENCE: [INAUDIBLE PHRASE].

PROFESSOR: Right. So essentially, sequentialize your execution because reading from the file system becomes the bottleneck. So you'll want to schedule input and output carefully. You might find that not everybody needs to do the input and output. Only one processor has to do the input and then it can distribute it to all the different processors. So, in the Master/Slave model, which a lot of you are using for the Cell programming, the Master can just read the data from the input files and distribute it to everybody else. So this avoids some of the problems with input and output. You can have similar kinds of problems if you're reading from other devices. It doesn't have to be the file system. So here's another one, a little more subtle. So you're generating data--.

Hey, Allen, what's up?

AUDIENCE: I somehow missed the distinction between when you're waiting for the master to read all the data and distribute it, and waiting for the other [? processes ?] to get through so I can read my private data, isn't it going to be about the same time on this?

PROFESSOR: No. So here, just essentially, the Master reads the file as part of the initialization. Then you distribute it. So distribution can happen at run time. So, the initialization you don't care about because hopefully that's a small part of the code. So this code is guarded by rank equals Master, so only it does this code. Then here you might have the command that says wait until I've received it and then execute, or on the cells, then these might be the SPE create threads that happen after you've read the data. So hopefully, initialization time is not something you have concern about too much.

So if you're generating data on the fly or dynamically, so here we might use the Srand function to sort of start with a random seeing and then fill in the buffer with some random data. So what could go wrong here? So in Srand, when you're using a random function -- sorry, this is the same function. When you're using a random, a pseudo random number generator, you have to give it a seed, then if everybody starts off with the same seed, then you might end up with the same random number sequence. If that's something you're using to parallelize your computation, you might, in effect, end up with the same kind of sequence on each processor and you lose all kinds of parallelization. So there's some hidden serialization issues in some of the functions that you might use that you should be aware of.

The last one I'll talk about is performance scalability defect. So here you parallelize your code, things look good, but you're still not getting -- you've taken care of all your IO issue, you're still not getting the performance you want. So, why is that? You might have -- remember your Amdahl's law, and what you want is an efficiency that's linear. Every time you add one processor you want a straight line curve between the number of processors and speed up. This should be a linear relationship. So you might see sublinear speed ups, and you want to figure out why that is.

Some of the common causes here, and this will be the end up focus of the next talk is, unbalanced amount of computation. Remember, dynamic load balancing versus static load balancing. Your work estimation might be wrong and so you might end up with some processors idling, other processors doing too much work. So the way to prevent this is to actually look at the work that's being done and figure out whether it's actually roughly the same amount of work everywhere. Here you might need profiling tools to help, and so I'm going to talk about this in a lot more detail in the next lecture.

So in summary, there are lots of different bugs that you might come up with. There's a few that I've identified here, some common things you should look out for. So the erroneous use of language features understand only a few basic concepts of the entire language extension set that you have. Space decomposition, side effects from parallelization. Don't ignore sequential code. Last one is trying to understand your performance scalability. But there are other kinds of bugs, like data races, for example. So what can you do with those? So remember, data races you have different concurrent threads and they're trying to update the same memory location. So depending on who gets to write first and when you actually do your read, you might get a different result.

So with data race detection, these things are actually getting better. There are tools out there that will essentially generate traces as your program is running. So for each thread you instrumented and you look at every load stored at executes. Then



what you do is you look at the load in stores between the difference threads and see if there's any intersections, any orderings that might give you erroneous behavior. So this is getting better, it's getting more automated. Intel Threadchecker is one example. There are others. I really think the trend in debugging will be towards trace-based systems. You'll have things like checkpointing. So as your program is running you can take a snapshot of where it is in the execution, and then you can use that snapshot later on to inspect it and see what went wrong.

I think you might even have features like replay. In fact, some people are working on this in research and in industry. So you might be able to say uh-oh, something went wrong. Here's my list of checkpoints, can you replay the execution from this particular stage in the computation. So it helps you focus down in the entire lifetime of execution on a particular chunk where things have gone wrong. This is sort of a personal dream. I think one day we'll have the equivalent of a TiVo for your programs, and you can use it for debugging. So my program is running, something goes wrong, I can rewind it, I can inspect things, do my traditional debugging, change things maybe even, and then start replaying things and letting the program execute. In fact, we're working on things like this here at MIT and with collaborators elsewhere.

So, this was a short lecture. We'll take a break. You can do the quizzes. Note on the quizzes, there are two different kinds of questions. They're very similar, just one word is different, and so you'll want to just keep that in mind when you're discussing it with others. Then about 5, 10 minutes and then we'll continue with the rest of the talk, lecture 2. Thanks.