PROFESSOR: So last week, last few lectures, you heard about parallel architectures and started with lecture four on discussions of concurrency. How do you take applications or independent actors that want to operate on the same data and make them run safely together? And so just recapping the last two lectures, you saw really two primary classes of architectures. Although Saman talked about a few more.

There was the class of shared memory processors, you know, the multicores that Intel, AMD, and PowerPC, for example, have today, where you have one copy of the data. And that's really shared among all the different processors because they essentially share the same memory. And you need things like atomicity and synchronization to be able to make sure that the sharing is done properly so that you don't get into data race situations where multiple processors try to update the same data element and you end up with erroneous results.

You also heard about distributed memory processors. So an example of that might be the Cell, loosely said, where you have cores that primarily access their own local memory. And while you can have a single global memory address space, to get data from memory you essentially have to communicate with the different processors to explicitly fetch data in and out. So things like data distribution, where the data is, and what your communication pattern is like affect your performance.

So what I'm going to talk about in today's lecture is programming these two different kinds of architectures, shared memory processors and distributed memory processors, and present you with some concepts for commonly programming these machines. So in shared memory processors, you have, say, n processors, 1 to n. And they're connected to a single memory. And if one processor asks for the value stored at address X, everybody knows where it'll go look. Because there's only one address X. And so different processors can communicate through shared variables. And you need things like locking, as I mentioned, to avoid race conditions or erroneous computation.

So as an example of parallelization, you know, straightforward parallelization in a shared memory machine, would be you have the simple loop that's just running through an array. And you're adding elements of array A to elements of array B. And you're going to write them to some new array, C. Well, if I gave you this loop you can probably recognize that there's really no data dependencies here. I can split up this loop into three chunks -- let's say I have three processors -- where one processor does all the computations for iterations zero through three, so the first four iterations. Second processor does the next four iterations. And the third processor does the last four iterations.

And so that's shown with the little -- should have brought a laser pointer. So that's showing here. And what you might need to do is some mechanism to essentially tell the different processors, here's the code that you need to run and maybe where to start. And then you may need some way of sort of synchronizing these different processors that say, I'm done, I can move on to the next computation steps.

So this is an example of a data parallel computation. The loop has no real dependencies and, you know, each processor can operate on different data sets. And what you could do is you can have a process -- this is a single application that forks off or creates what are commonly called the threads. And each thread goes on and executes in this case the same computation. So a single process can create multiple concurrent threads. And really each thread is just a mechanism for encapsulating some trace of execution, some execution path.

So in this case you're essentially encapsulating this particular loop here. And maybe you parameterize your start index and your ending index or maybe your loop bounds. And in a shared memory processor, since you're communicating -- since there's only a single memory, really you don't need to do anything special about the data in this particular example, because everybody knows where to go look for it. Everybody can access it. Everything's independent. There's no real issues with races or deadlocks.

So I just wrote down some actual code for that loop that parallelize it using Pthreads, a commonly used threading mechanism. Just to give you a little bit of flavor for, you know, the complexity of -- the simple loop that we had expands to a lot more code in this case. So you have your array. It has 12 elements. A, B, and C. And you have the basic functions. So this is the actual code or computation that we want to carry out. And what I've done here is I've parameterized where you're essentially starting in the array. So you get this parameter. And then you calculate four iterations' worth. And this is essentially the computation that we're carrying out.

And now in my main program or in my main function, rather, what I do is I have this concept of threads that I'm going to create. In this case I'm going to create three of them. There are some parameters that I have to pass in, so some attributes which are now going to get into here. But then I pass in the function pointer. This is essentially a mechanism that says once I've created this thread, I go to this function and execute this particular code. And then some arguments that are functions. So here I'm just passing in an index at which each loop switch starts with. And after I've created each thread here, implicitly in the thread creation, the code can just immediately start running. And then once all the threads have started running, I can essentially just exit the program because I've completed.

So what I've shown you with that first example was the concept of, or example of, data parallelism. So you're performing the same computation, but instead of operating on one big chunk of data, I've partitioned the data into smaller chunks and I've replicated the computation so that I can get that kind of parallelism. But there's another form of parallelism called control parallelism, which essentially uses the same model of threading but doesn't necessarily have to run the same function or run the same computation each thread. So I've sort of illustrated that in the illustration there, where these are your data parallel computations and these are some other computations in your code.

So there is sort of a programming model that allows you to do this kind of parallelism and tries to sort of help the programmer by taking their sequential code and then adding annotations that say, this loop is data parallel or this set of code is has this kind of control parallelism in it. So you start with your parallel code. This is the same program, multiple data kind of parallelization. So you might have seen in the previous talk and the previous lecture, it was SIMD, single instruction or same instruction, multiple data, which allowed you to execute the same operation, you know, and add over multiple data elements.

So here it's a similar kind of terminology. There's same program, multiple data, and multiple program, multiple data. This talk is largely focused on the SPMD model, where you essentially have one central decision maker or you're trying to solve one central computation. And you're trying to parallelize that over your architecture to get the best performance. So you start off with your program and then you annotate the code with what's parallel and what's not parallel. And you might add in some synchronization directives so that if you do in fact have sharing, you might want to use the right locking mechanism to guarantee safety.

Now, in OpenMP, there are some limitations as to what it can do. So it in fact assumes that the programmer knows what he's doing. And the programmer is largely responsible for getting the synchronization right, or that if they're sharing that they get those dependencies protected correctly. So you can take your program, insert these annotations, and then you go on and test and debug.

So a simple OpenMP example, again using the simple loop -- now, I've thrown away some of the extra code -- you're adding these two extra pragmas in this case. The first one, your parallel pragma, I call the data parallel pragma, really says that you can execute as many of the following code block as there are processors or as many as you have thread contexts. So in this case I implicitly made the assumption that I have three processors, so I can automatically partition my code into three sets. And this transformation can sort of be done automatically by the compiler.

And then there's a for pragma that says this loop is parallel and you can divide up the work in the mechanism that's work sharing. So multiple threads can collaborate to solve the same computation, but each one does a smaller amount of work. So this is in contrast to what I'm going to focus on a lot more in the rest of our talk, which is distributed memory processors and programming for distributed memories. And this will feel a lot more like programming for the Cell as you get more and more involved in that and your projects get more intense.

So in distributed memory processors, to recap the previous lectures, you have n processors. Each processor has its own memory. And they essentially share the interconnection network. Each processor has its own address, X. So when a processor, P1, asks for X it knows where to go look. It's going to look in its own local memory. So if all processors are asking for the same value as sort of address X, then each one goes and looks in a different place. So there are n places to look, really.

And what's stored in those addresses will vary because it's everybody's local memory. So if one processor, say P1, wants to look at the value stored in processor two's address, it actually has to explicitly request it. The processor two has to send it data. And processor one has to figure out, you know, what to do with that copy. So it has to store it somewhere.

So this message passing really exposes explicit communication to exchange data. And you'll see that there are different kinds of data communications. But really the concept of what you exchange has three different -- or four different, rather -- things you need to address. One is how is the data described and what does it describe? How are the processes identified? So how do I identify that processor one is sending me this data? And if I'm receiving data how do I know who I'm receiving it from?

Are all messages the same? Well, you know, if I send a message to somebody, do I have any guarantee that it's received or not? And what does it mean for a send operation or a receive operation to be completed? You know, is there some sort of acknowledgment process?

So an example of a message passing program -- and if you've started to look at the lab you'll see that this is essentially where the lab came from. It's the same idea. I've created -- here I have some two-dimensional space. And I have points in this two-dimensional space. I have points B, which are these blue circles, and I have points A which I've represented as these yellow or golden squares.

And what I want to do is for every point in A I want to calculate the distance to all of the points B. So there's sort of a pair wise interaction between the two arrays. So a simple loop that essentially does this -- and there are n squared interactions, you have, you know, a loop that loops over all the A elements, a loop that loops over all the B elements. And you essentially calculate in this case Euclidean distance which I'm not showing. And you store it into some new array.

So if I give you two processors to do this work, processor one and processor two, and I give you some mechanism to share between the two -- so here's my CPU. Each processor has local memory. What would be some approach for actually parallelizing this? Anybody look at the lab yet? OK, so what would you do with two processors?

AUDIENCE: One has half memory [INAUDIBLE]

PROFESSOR: Right. So what was said was that you split one of the arrays in two and you can actually get that kind of concurrency. So, you know, let's say processor one already has the data. And it has some place that it's already allocated where it's going to write C, the results of the computation, then I can break up the work just like it was suggested. So what P1 has to do is send data to P2. It says here's the data. Here's the computation. Go ahead and help me out.

So I send the first array elements, and then I send half of the other elements that I want the calculations done for. And then P1 and P2 can now sort of start computing in parallel.

But notice that P2 has its own array that it's going to store results in. And so as these compute they actually fill in different logical places or logical parts of the overall matrix. So what has to be done is at the end for P1 to have all the results, P2 has to send it sort of the rest of the matrix to complete it. And so now P1 has all the results. The computation is done and you can move on. Does that make sense? OK. So you'll get to actually do this as part of your labs.

So in this example messaging program, you have started out with a sequential code. And we had two processors. So processor one actually sends the code. So it is essentially a template for the code you'll end up writing. And it does have to work in the outer loop. So this n array over which it is iterating the A array, is it's only doing half as many.

And processor two has to actually receive the data. And it specifies where to receive the data into. So I've omitted some things, for example, extra information sort of hidden in these parameters. So here you're sending all of A, all of B. Whereas, you know, you could have specified extra parameters that says, you know, I'm sending you A. Here's n elements to read from A. Here's B. Here's n by two elements to read from B. And so on.

But the computation is essentially the same except for the index at which you start, in this case changed for processor two. And now, when the computation is done, this guy essentially waits until the data is received. Processor two eventually sends it that data and now you can move on.

AUDIENCE: I have a question.

PROFESSOR: Yeah?

AUDIENCE: So would processor two have to wait for the data from processor one?

PROFESSOR: Yeah, so there's a -- I'll get into that later. So what does it mean to receive? To do this computation, I actually need this instruction to complete. So what does it need for that instruction to complete? I do have to get the data because otherwise I don't know what to compute on. So there is some implicit synchronization that you have to do. And in some cases it's explicit. So I'll get into that a little bit later. Does that sort of hint at the answer? Are you still confused?

AUDIENCE: So processor one doesn't do the computation but it still sends the data --

PROFESSOR: So in terms of tracing, processor one sends the data and then can immediately start executing its code, right? Processor two, in this particular example, has to wait until it receives the data.

So once this receive completes, then you can actually go and start executing the rest of the code. So imagine that it essentially says, wait until I have data. Wait until I have something to do. Does that help?

AUDIENCE: Can the main processor [UNINTELLIGIBLE PHRASE]

PROFESSOR: Can the main processor --

AUDIENCE: I mean, in Cell, everybody is not peers. There is a master there. And what master can do instead of doing computation, master can be basically the quarterback, sending data, receiving data. And SPEs can be basically waiting for data, get the computation, send it back. So in some sense in Cell you probably don't want to do the computation on the master. Because that means the master slows down. The master will do only the data management. So that might be one symmetrical [UNINTELLIGIBLE]

PROFESSOR: And you'll see that in the example. Because the PPE in that case has to send the data to two different SPEs. Yup?

AUDIENCE: In some sense [UNINTELLIGIBLE PHRASE] at points seems to be [UNINTELLIGIBLE] sense that if -- so have a huge array and you want to [UNINTELLIGIBLE PHRASE] the data to receive the whole array, then you have to [UNINTELLIGIBLE]

PROFESSOR: Yeah, we'll get into that later. Yeah, I mean, that's a good point. You know, communication is not cheap. And if you sort of don't take that into consideration, you end up paying a lot for overhead for parallelizing things.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Well, you can do things in software as well. We'll get into that.

OK, so some crude performance analysis. So I have to calculate this distance. And given two processors, I can effectively get a 2x speedup. By dividing up the work I can get done in half the time. Well, if you gave me four processors, I can maybe get done four times as fast. And in my communication model here, I have one copy of one array that's essentially sending to every processor. And there's some subset of A. So I'm partitioning my other array into smaller subsets. And I'm sending those to each of the different processors.

So we'll get into terminology for how to actually name these communications later. But really the thing to take away here is that this granularity -- how I'm partitioning A -- affects my performance and communication almost directly. And, you know, the comment that was just made is that, you know, what do you do about communication? It's not free. So all of those will be addressed.

So to understand performance, we sort of summarize three main concepts that you essentially need to understand. One is coverage, or in other words, how much parallelism do I actually have in my application? And this can actually affect, you know, how much work is it worth spending on this particular application? Granularity -- you know, how do you partition your data among your different processors so that you can keep communication down, so you can keep synchronization down, and so on. Locality -- so while not shown in the particular example, if two processors are communicating, if they are close in space or far in space, or if the communication between two processors is far cheaper than two other processors, can I exploit that in some way? And so we'll talk about that as well.

So an example of sort of parallelism in an application, there are two essentially projects that are doing ray tracing, so I thought I'd have this slide here. You know, how much parallelism do you have in a ray tracing program. In ray tracing what you do is you essentially have some camera source, some observer. And you're trying to figure out, you know, how to color or how to shade different pixels in your screen. So what you do is you shoot rays from a particular source through your plane. And then you see how the rays bounce off of other objects. And that allows you to render scenes in various ways.

So you have different kinds of parallelisms. You have your primary ray that's shot in. And if you're shooting into something like water or some very reflective surface, or some surface that can actually reflect, transmit, you can essentially end up with a lot more rays that are created at run time. So there's dynamic parallelism in this particular example. And you can shoot a lot of rays from here. So there's different kinds of parallelism you can exploit.

Not all prior programs have this kind of, sort of, a lot of parallelism, or embarrassingly parallel computation. You know, you saw some basic code sequences in earlier lectures. So there's a sequential part. And the reason this is sequential is because there are data flow dependencies between each of the different computations. So here I calculate a, but I need the result of a to do this instruction. I calculate d here and I need that result to calculate e. But then this loop really here is just assigning or it's initializing some big array. And I can really do that in parallel.

So I have sequential parts and parallel parts. So how does that affect my overall speedups? And so there's this law which is really a demonstration of diminishing returns, Amdahl's Law. It says that if, you know, you have a really fast car, it's only as good to you as fast as you can drive it. So if there's a lot of congestion on your road or, you know, there are posted speed limits or some other mechanism, you really can't exploit all the speed of your car. Or in other words, you're only as fast as the fastest mechanisms of the computation that you can have.

So to look at this in more detail, your potential speedup is really proportional to the fraction of the code that can be parallelized. So if I have some computation -- let's say it has three parts: a sequential part that takes 25 seconds, a parallel part that takes 50 seconds, and a sequential part that runs in 25 seconds. So the total execution time is 100 seconds. And if I have one processor, that's really all I can do. And if she gave me more than one processor -- so let's say I have five processors. Well, I can't do anything about the sequential work. So that's still going to take 25 seconds. And I can't do anything about this sequential work either. That still takes 25 seconds. But this parallel part I can essentially break up among the different processors. So five in this case. And that gets me, you know, five-way parallelism. So the 50 seconds now is reduced to 10 seconds. Is that clear so far?

So the overall running time in that case is 60 seconds. So what would be my speedup? Well, you calculate speedup, old running time divided by the new running time. So 100 seconds divided by 60 seconds. Or my parallel version is 1.67 times faster. So this is great.

If I increase the number of processors, then I should be able to get more and more parallelism. But it also means that there's sort of an upper bound on how much speedup you can get. So if you look at the fraction of work in your application that's parallel, that's p. And your number of processors, well, your speedup is -- let's say the old running time is just one unit of work. If the time it takes for the sequential work -- so that's 1 minus p, since p is the fraction of the parallel work. And it's the time to do the parallel work. And since I can parallelize that fraction over n processors, I can sort of reduce that to really small amounts in the limit. Does that make sense so far?

So the speedup can tend to 1 over 1 minus p in the limit. If I increase the number of processors or that gets really large, that's essentially my upper bound on how fast programs can work. You know, how much can I exploit out of my program? So this is great. What this law says -- the implication here is if your program has a lot of inherent parallelism, you can do really well. But if your program doesn't have any parallelism, well, there's really nothing you can do. So parallel architectures won't really help you. And there's some interesting trade-offs, for example, that you might consider if you're designing a chip or if you're looking at an application or domain of applications, figuring out what is the best architecture to run them on.

So in terms of performance scalability, as I increase the number of processors, I have speedup. You can define, sort of, an efficiency to be linear at 100%. But typically you end up in sort of the sublinear domain. That's because communication is not often free.

But you can get super linear speedups ups on real architectures because of secondary and tertiary effects that come from register allocation or caching effects. So they can hide a lot of latency or you can take advantage of a lot of pipelining mechanisms in the architecture to get super linear speedups. So you can end up in two different domains.

So a small, you know, overview of the extent of parallelism in your program and how that affects your overall execution. And the other concept is granularity. So given that I have this much parallelism, how do I exploit it? There are different ways of exploiting it, and that comes down to, well, how do I subdivide my problem? What is the granularity of the sub-problems I'm going to calculate on? And, really, granularity from my perspective, is just a qualitative measure of what is the ratio of your computation to your communication? So if you're doing a lot of computation, very little communication, you could be doing really well or vice versa. Then you could be computation limited, and so you need a lot of bandwidth for example in your architecture.

AUDIENCE: Like before, you really didn't have to give every single processor an entire copy of B.

PROFESSOR: Right. Yeah. Good point. And as you saw in the previous slides, you have -- computation stages are separated by communication stages. And your communication in a lot of cases essentially serves as synchronization. I need everybody to get to the same point before I can move on logically in my computation.

So there are two kinds of sort of classes of granularity. There's fine grain and, as you'll see, coarse grain. So in fine-grain parallelism, you have low computation to communication ratio. And that has good properties in that you have a small amount of work done between communication stages. And it has bad properties in that it gives you less performance opportunity. It should be more, right? More opportunity for --

AUDIENCE: No. Less.

PROFESSOR: Sorry. Yeah, yeah, sorry. I didn't get enough sleep. So less opportunities for performance enhancement, but you have high communication ratio because essentially you're communicating very often. So these are the computations here and these yellow bars are the synchronization points. So I have to distribute data or communicate. I do computations but, you know, computation doesn't last very long. And I do more communication or more synchronization, and I repeat the process. So naturally you can adjust this granularity to sort of reduce the communication overhead.

AUDIENCE: [UNINTELLIGIBLE PHRASE] two things in that overhead part. One is the volume. So one, communication. Also there's a large part of synchronization cost. Basically you get a communication goal and you have to go start the messages and wait until everybody is done. So that overhead also can go. Even if you don't send that much data, just the fact that you are communicating, that means you have to do a lot of this additional bookkeeping stuff, that especially in the distributed [? memory ?] [? machine is ?] pretty expensive.

PROFESSOR: Yeah. Thanks. So in coarse-grain parallelism, you sort of make the work chunks more and more so that you do the communication synchronization less and less. And so that's shown here. You do longer pieces of work and have fewer synchronization stages.

So in that regime, you can have more opportunities for performance improvements, but the tricky thing that you get into is what's called load balancing. So if each of these different computations takes differing amounts of time to complete, then what you might end up doing is a lot of people might end up idle as they wait until everybody's essentially reached their finish line. Yep?

AUDIENCE: If you don't have to acknowledge that something's done can't you just say, [? OK, I'm done with your salt ?]. Hand it to the initial processor and keep doing whatever?

PROFESSOR: So, you can do that in cases where that essentially there is a mechanism -- or the application allows for it. But as I'll show -- well, you won't see until the next lecture -- there are dependencies, for example, that might preclude you from doing that. If everybody needs to reach the same point because you're updating a large data structure before you can go on, then you might not be able to do that. So think of doing molecular dynamics simulations. You need everybody to calculate a new position before you can go on and calculate new kinds of coarse interactions.

AUDIENCE: [UNINTELLIGIBLE] nothing else to calculate yet.

PROFESSOR: Right.

AUDIENCE: But also there is pipelining. So what do you talk about [UNINTELLIGIBLE] because you might want to get the next data while you're computing now so that when I'm done I can start sending. [UNINTELLIGIBLE PHRASE] you can all have some of that.

PROFESSOR: Yep. Yeah, because communication is such an intensive part, there are different ways of dealing with it. And that will be right after load balancing. So the load balancing problem is just an illustration. And things that appear in sort of this lightish pink will serve as sort of visual cues. This is the same color coding scheme that David's using in the recitations. So this is PPU code. Things that appear in yellow will be SPU code. And these are just meant to essentially show you how you might do things like this on Cell, just to help you along in picking up more of the syntax and functionality you need for your programs.

So in the load balancing problem, you essentially have, let's say, three different threads of computation. And so that's shown here: red, blue, and orange. And you've reached some communication stage. So the PPU program in this case is saying, send a message to each of my SPEs, to each of my different processors, that you're ready to start.

And so now once every processor gets that message, they can start computing. And let's assume they have data and so on ready to go. And what's going to happen is each processor is going to run through the computation at different rates. Now this could be because one processor is faster than another. Or it could be because one processor is more loaded than another. Or it could be just because each processor is assigned sort of differing amounts of work. So one has a short loop. One has a longer loop.

And so as the animation shows, sort of, execution proceeds and everybody's waiting until the orange guy has completed. But nobody could have made progress until everybody's reached synchronization point because, you know, there's a strict dependence that's being enforced here that says, I'm going to wait until everybody's told me they're done before I go on to the next step of computation. And so you know, in Cell you do that using mailboxes in this case. That clear so far?

So how do you get around this load balancing problem? Well, there are two different ways. There's static load balancing. I know my application really, really well. And I understand sort of different computations. So what I can do is I can divide up the work and have a static mapping of the work to my processors. And static mapping just means, you know, in this particular example, that I'm going to assign the work to different processors and that's what the processors will do. Work can't shift around between processors.

And so in this case I have a work queue. Each of those bars is some computation. You know, I can assign some chunk to P1, processor one, some chunk to processor two. And then computation can go on. Those allocations don't change. So this works well if I understand the application, well and I know the computation, and my cores are relatively homogeneous and, you know, there's not a lot of contention for them. So if all the cores are the same, each core has an equal amount of work -- the total amount of work -- this works really well because nobody is sitting too idle.

It doesn't work so well for heterogeneous architectures or multicores. Because one might be faster than the other. It increases the complexity of the allocation I need to do. If there's a lot of contention for some resources, then that can affect the static load balancing. So work distribution might end up being uneven. So the alternative is dynamic load balancing. And you certainly could do sort of a hybrid load balancing, static plus dynamic mechanism. Although I don't have that in the slides.

So in the dynamic load balancing scheme, two different mechanisms I'm going to illustrate. So in the first scheme, you start with something like the static mechanism. So I have some work going to processor one. And I have some work going to processor two. But then as processor two executes and completes faster than processor one, it takes on some of the additional work from processor one. So the work that was here is now shifted. And so you can keep helping out, you know, your other processors to compute things faster.

In the other scheme, you have a work queue where you essentially are distributing work on the fly. So as things complete, you're just sending them more work to do. So in this animation here, I start off. I send work to two different processors. P2 is really fast so it's just zipping through things. And then P1 eventually finishes and new work is allocated to the two different schemes. So dynamic load balancing is intended to sort of give equal amounts of work in a different scheme for processors. So it really increased utilization and spent less and less time being idle.

OK. So load balancing was one part of sort of how granularity can have a performance tradeoff. The other is synchronization. So there were already some good questions as to, well, you know, how does this play into overall execution? When can I wait? When can't I wait? So I'm going to illustrate it with just a simple data dependence graph. Although you can imagine that in each one of these circles there's some really heavy load computation. And you'll see that in the next lecture, in fact.

So if I have some simple computation here -- I have some operands. I'm doing an addition. Here I do another addition. I need both of these results before I can do this multiplication. Here I have, you know, some loop that's adding through some array elements. I need all those results before I do final substraction and produce my final result.

So what are some synchronization points here? Well, it really depends on how I allocate the different instructions to processors. So if I have an allocation that just says, well, let's put all these chains on one processor, put these two chains on two different processors, well, where are my synchronization points? Well, it depends on where this guy is and where this guy is. Because for this instruction to execute, it needs to receive data from P1 and P2. So if P1 and P2 are different from what's in that box, somebody has to wait. And so there's a synchronization that has to happen.

So essentially at all join points there's potential for synchronization. But I can adjust the granularity so that I can remove more and more synchronization points. So if I had assigned all this entire sub-graph to the same processor, I really get rid of the synchronization because it is essentially local to that particular processor. And there's no extra messaging that would have to happen across processors that says, I'm ready, or I'm ready to send you data, or you can move on to the next step. And so in this case the last synchronization point would be at this join point. Let's say if it's allocated on P1 or on some other processor. So how would I get rid of this synchronization point?

AUDIENCE: Do the whole thing.

PROFESSOR: Right. You put the entire thing on a single processor. But you get no parallelism in this case. So the coarse-grain, fine-grain grain parallelism granularity issue comes to play.

So the last sort of thing I'm going to talk about in terms of how granularity impacts performance -- and this was already touched on -- is that communication is really not cheap and can be quite overwhelming on a lot of architectures. And what's interesting about multicores is that they're essentially putting a lot more resources closer together on a chip. So it essentially is changing the factors for communication. So rather than having, you know, your parallel cluster now which is connected, say, by ethernet or some other high-speed link, now you essentially have large clusters or will have large clusters on a chip. So communication factors really change.

But the cost model is relatively captured by these different parameters. So what is the cost of my communication? Well, it's equal to, well, how many messages am I sending and what is the frequency with which I'm sending them? There's some overhead for message. So I have to actually package data together. I have to stick in a control header and then send it out. So that takes me some work on the receiver side. I have to take the message. I maybe have to decode the header, figure out where to store the data that's coming in on the message. So there's some overhead associated with that as well.

There's a network delay for sending a message, so putting a message on the network so that it can be transmitted, or picking things up off the network. So there's a latency also associated with how long does it take for a message to get from point A to point B. What is the bandwidth that I have across a link? So if I have a lot of bandwidth then that can really lower my communication cost. But if I have little bandwidth then that can really create contention.

How much data am I sending? And, you know, number of messages. So this numerator here is really an average of the data that you're sending per communication. There's a cost induced per contention. And then finally there's -- so all of these are added factors. The higher they are, except for bandwidth, because it's in the denominator here, the worse your communication cost becomes.

So you can try to reduce the communication cost by communicating less. So you adjust your granularity. And that can impact your synchronization or what kind of data you're shipping around. You can do some architectural tweaks or maybe some software tweaks to really get the network latency down and the overhead per message down. So on something like raw architecture, which we saw in Saman's lecture, there's a really fast mechanism to communicate your nearest neighbor in three cycles. So one processor can send a single operand to another reasonably fast. You know, you can improve the bandwidth again in architectural mechanism. You can do some tricks as to how you package your data in each message.

And lastly, what I'm going to talk about in a couple of slides is, well, I can also improve it using some mechanisms that try to increase the overlap between messages. And what does this really mean? What am I overlapping it with? And it's really the communication and computation stages are going to somehow get aligned.

So before I actually show you that, I just want to point out that there are two kinds of messages. There's data messages, and these are, for example, the arrays that I'm sending around to different processors for the distance calculations between points in space. But there are also control messages. So control messages essentially say, I'm done, or I'm ready to go, or is there any work for me to do? So on Cell, control messages, you know, you can think of using Mailboxes for those and the DMAs for doing the data communication. So data messages are relatively much larger -- you're sending a lot of data -- versus control messages that are really much shorter, just essentially just sending you very brief information.

So in order to get that overlap, what you can do is essentially use this concept of pipelining. So you've seen pipelining in superscalar. Someone talked about that. And what you are essentially trying to do is break up the communication and computation into different stages and then figure out a way to overlap them so that you can essentially hide the latency for the sends and the receives.

So let's say you have some work that you're doing, and it really requires you to send the data -- somebody has to send you the data or you essentially have to wait until you get it. And then after you've waited and the data is there, you can actually go on and do your work. So these are color coded. So this is essentially one iteration of the work.

And so you could overlap them by breaking up the work into send, wait, work stages, where each iteration trying to send or request the data for the next iteration, I wait on the data from a previous iteration and then I do my work. So depending on how I partition, I can really get really good overlap. And so what you want to get to is the concept of the steady state, where in your main loop body, all you're doing is essentially pre-fetching or requesting data that's going to be used in future iterations for future work. And then you're waiting on -- yeah. I think my color coding is a little bogus. That's good.

So here's an example of how you might do this kind of buffer pipelining in Cell. So I have some main loop that's going to do some work, that's encapsulating this process data. And what I'm going to use is two buffers. So the scheme is also called double buffering. I'm going to use this ID to represent which buffer I'm going to use. So it's either buffer zero or buffer one. And this instruction here essentially flips the bit. So it's either zero or one.

So I fetch data into buffer zero and then I enter my loop. So this is essentially the first send, which is trying to get me one iteration ahead. So I enter this mail loop and I do some calculation to figure out where to write the next data. And then I do another request for the next data item that I'm going to -- sorry, there's an m missing here -- I'm going to fetch data into a different buffer, right. This is ID where I've already flipped the bit once.

So this get is going to write data into buffer zero. And this get is going to write data into buffer one. I flip the bit again. So now I'm going to issue a wait instruction that says is the data from buffer zero ready? And if it is then I can go on and actually do my work. Does that make sense? People are confused? Should I go over it again?

AUDIENCE: [INAUDIBLE]

PROFESSOR: So this is an [? x or. ?] So I could have just said buffer equals zero or buffer equals one. Oh, sorry. This is one. Yeah. Yeah. So this is a one here. Last-minute editing. It's right there. Did that confuse you?

AUDIENCE: No. But, like, I don't see [INAUDIBLE]

PROFESSOR: Oh. OK. So I'll go over it again. So this get here is going to write into ID zero. So that's buffer zero. And then I'm going to change the ID. So imagine there's a one here. So now the next time I use ID, which is here, I'm trying to get the data. And I'm going to write it to buffer one.

The DMA on the Cell processor essentially says I can send this request off and I can check later to see when that data is available. But that data is going to go into a different buffer, essentially B1. Whereas I'm going to work on buffer zero. Because I changed the ID back here. Now you get it? So I fetch data into buffer zero initially before I start to loop. And then I start working. I probably should have had an animation in here.

So then you go into your main loop. You try to start fetching into buffet one and then you try to compute out of buffer zero. But before you can start computing out of buffer zero, you just have to make sure that your data is there. And so that's what the synchronization is doing here. Hope that was clear. OK, so this kind of computation and communication overlap really helps in hiding the latency. And it can be real useful in terms of improving performance.

And there are different kinds of communication patterns. So there's point to point. And you can use these both for data communication or control communication. And it just means that, you know, one processor can explicitly send a message to another processor. There's also broadcast that says, hey, I have some data that everybody's interested in, so I can just broadcast it to everybody on the network. Or a reduce, which is the opposite. It says everybody on the network has data that I need to compute, so everybody send me their data.

There's an all to all, which says all processors should just do a global exchange of data that they have. And then there's a scatter and a gather. So a scatter and a gather are really different types of broadcast. So it's one to several or one to many. And gather, which is many to one. So this is useful when you're doing a computation that really is trying to pull data in together but only from a subset of all processors. So it depends on how you've partitioned your problems.

So there's a well-known sort of message passing library specification called MPI that tries to specify all of these different communications in order to sort of facilitate parallel programming. Its full featured actually has more types of communications and more kinds of functionality than I showed on the previous slides. But it's not a language or a compiler specification. It's really just a library that you can implement in various ways on different architectures.

Again, it's same program, multiple data, or supports the SPMD model. And it works reasonably well for parallel architectures for clusters, heterogeneous multicores, homogeneous multicores. Because really all it's doing is just abstracting out -- it's giving you a mechanism to abstract out all the communication that you would need in your computation. So you can have additional things like precise buffer management. You can have some collective operations. I'll show an example of for doing things things in a scalable manner when a lot of things need to communicate with each other.

So just a brief history of where MPI came from. And, you know, very early when, you know, parallel computers started becoming more and more widespread and there were these networks and people had problems porting their applications or writing applications for these [? came, ?] just because it was difficult, as you might be finding in terms of programming things with the Cell processor. You know, there needed to be ways to sort of address the spectrum of communication. And it often helps to have a standard because if everybody implements the same standard specification, that allows your code to be ported around from one architecture to the other.

And so MPI came around. The forum was organized in 1992. And that had a lot of people participating in it from vendors, you know, people like IBM, a company like IBM, Intel, and people who had expertise in writing libraries, users who were interested in using these kinds of specifications to do their computations, so scientific people who were in the scientific domain. And it was finished in about 18 months. I don't know if that's a reasonably long time or a short time. But considering, you know, I think the MPEG-4 standard took a bit longer to do, as a comparison point. I don't have the actual data.

So point-to-point communication -- and again, a reminder, this is how you would do it on Cell. These are SPE sends and receives. You have one processor that's sending it to another processor. Or you have some network in between. And processor A can essentially send the data explicitly to processor two. And the message in this case would include how the data is packaged, some other information such as the length of the data, destination, possibly some tag so you can identify the actual communication. And, you know, there's an actual mapping for the actual functions on Cell. And there's a get for the send and a put for the receive.

So there's a question of, well, how do I know if my data actually got sent? How do I know if it was received? And there's, you know, you can think of a synchronous send and a synchronous receive, or asynchronous communication. So in the synchronous communication, you actually wait for notification. So this is kind of like your fax machine. You put something into your fax. It goes out and you eventually get a beep that says your transmission was OK. Or if it wasn't OK then, you know, you get a message that says, you know, something went wrong. And you can redo your communication. An asynchronous send is kind of like your --

AUDIENCE: Most [UNINTELLIGIBLE] you could get a reply too.

PROFESSOR: Yeah, you can get a reply. Thanks. An asynchronous send, it's like you write a letter, you go put it in the mailbox, and you don't know whether it actually made it into the actual postman's bag and it was delivered to your destination or if it was actually delivered. So you only know that the message was sent. You know, you put it in the mailbox. But you don't know anything else about what happened to the message along the way.

There's also the concept of a blocking versus a non-blocking message. So this is orthogonal really to synchronous versus asynchronous. So in blocking messages, a sender waits until there's some signal that says the message has been transmitted. So this is, for example if I'm writing data into a buffer, and the buffer essentially gets transmitted to somebody else, we wait until the buffer is empty. And what that means is that somebody has read it on the other end or somebody has drained that buffer from somewhere else.

The receiver, if he's waiting on data, well, he just waits. He essentially blocks until somebody has put data into the buffer. And you can get into potential deadlock situations. So you saw deadlock with locks in the concurrency talk. I'm going to show you a different kind of deadlock example.

An example of a blocking send on Cell -- allows you to use mailboxes. Or you can sort of use mailboxes for that. Mailboxes again are just for communicating short messages, really, not necessarily for communicating data messages. So an SPE does some work, and then it writes out a message, in this case to notify the PPU that, let's say, it's done. And then it goes on and does more work. And then it wants to notify the PPU of something else. So in this case this particular send will block because, let's say, the PPU hasn't drained its mailbox. It hasn't read the mailbox. So you essentially stop and wait until the PPU has, you know, caught up.

AUDIENCE: So all mailbox sends are blocking?

PROFESSOR: Yes. David says yes. A non-blocking send is something that essentially allows you to send a message out and just continue on. You don't care exactly about what's happened to the message or what's going on with the receiver. So you write the data into the buffer and you just continue executing. And this really helps you in terms of avoiding idle times and deadlocks, but it might not always be the thing that you want.

So an example of sort of a non-blocking send and wait on Cell is using the DMAs to ship data out. You know, you can put something, put in a request to send data out on the DMA. And you could wait for it if you want in terms of reading the status bits to make sure it's completed.

OK, so what is a source of deadlock in the blocking case? And it really comes about if you don't really have enough buffering in your communication network. And often you can resolve that by having additional storage. So let's say I have processor one and processor two and they're trying to send messages to each other. So processor one sends a message at the same time processor two sends a message. And these are going to go, let's say, into the same buffer. Well, neither can make progress because somebody has to essentially drain that buffer before these receives can execute.

So what happens with that code is it really depends on how much buffering you have between the two. If you have a lot of buffering, then you may never see the deadlock. But if you have a really tiny buffer, then you do a send. The other person can't do the send because the buffer hasn't been drained. And so you end up with a deadlock. And so a potential solution is, well, you actually increase your buffer length. But that doesn't always work because you can still get into trouble.

So what you might need to do is essentially be more diligent about how you order your sends and receives. So if you have processor one doing a send, make sure it's matched up with a receive on the other end. And similarly, if you're doing a receive here, make sure there's sort of a matching send on the other end. And that helps you in sort of making sure that things are operating reasonably in lock step at, you know, partially ordered times.

That was really examples of point-to-point communication. A broadcast mechanism is slightly different. It says, I have data that I want to send to everybody. It could be really efficient for sending short control messages, maybe even efficient for sending data messages. So as an example, if you remember our calculation of distances between all points, the parallelization strategy said, well, I'm going to send one copy of the array A to everybody. In the two processor case that was easy. But if I have n processors, then rather than sending point-to-point communication from A to everybody else, what I could do is just, say, broadcast A to everybody and they can grab it off the network.

So in MPI there's this function, MPI broadcast, that does that. I'm using sort of generic abstract sends, receives and broadcasts in my examples. So you can broadcast A to everybody. And then if I have n processors, then what I might do is distribute the m's in a round robin manner to each of the different processes. So you pointed this out. I don't have to send B to everybody. I can just send, you know, in this case, one particular element. Is that clear?

AUDIENCE: There's no broadcast on Cell?

PROFESSOR: There is no broadcast on Cell. There is no mechanism for reduction either. And you can't quite do scatters and gathers. I don't think. OK, so an example of a reduction, you know, I said it's the opposite of a broadcast. Everybody has data that needs to essentially get to the same point. So as an example, if everybody in this room had a value, including myself, and I wanted to know what is the collective value of everybody in the room, you all have to send me your data.

Now, this is important because if -- you know, in this case we're doing an addition. It's an associative operation. So what we can do is we can be smart about how the data is sent. So, you know, guys that are close together can essentially add up their numbers and forward me. So instead of getting n messages I can get log n messages. And so if every pair of you added your numbers and forwarded me that, that cuts down communication by half. And so you can, you know -- starting from the back of room, by the time you get to me, I only get two messages instead of n messages.

So a reduction combines data from all processors. In MPI, you know, there's this function MPI reduce for doing that. And the collective operations are things that are associative. And subtract -- sorry. And or and -- you can read them on the slide. There is a semantic caveat here that no processor can finish the reduction before all processors have at least sent it one data or have contributed, rather, a particular value.

So in many numerical algorithms, you can actually use the broadcast and send to broadcast and reduce in place of sends and receives because it really improves the simplicity of your computation. You don't have to do n sends to communicate there. You can just broadcast. It gives you a mechanism for essentially having a shared memory abstraction on distributed memory architecture. There are things like all to all communication which would also help you in that sense. Although I don't talk about all to all communication here.

So I'm going to show you an example of sort of a more detailed MPI. But I also want to contrast this to the OpenMP programming on shared memory processors because one might look simpler than the other. So suppose that you have a numerical integration method that essentially you're going to use to calculate pi. So as you get finer and finer, you can get more accurate -- as you shrink these intervals you can get better values for pi.

And the code for doing that is some C code. You have some variables. And then you have a step that essentially tells you how many times you're going to do this computation. And for each time step you calculate this particular function here. And you add it all up and in the end you can sort of print out what is the value of pi that you calculated. So clearly as, you know, as you shrink your intervals, you can get more and more accurate measures of pi. So that translates to increasing the number of steps in that particular C code.

So you can use that numerical integration to calculate pi with OpenMP. And what that translates to is -- sorry, there should have been an animation here to ask you what I should add in. You have this particular loop. And this is computation that you want to parallelize. And there is really four questions that you essentially have to go through. Are there variables that are shared? Because you have to get the process right. If there are variables that are shared, you have to explicitly synchronize them and use locks to protect them.

What values are private? So in OpenMP, things that are private are data on the stack, things that are defined lexically within the scope of the computation that you encapsulate by an OpenMP pragma, and what variables you might want to use for a reduction. So in this case I'm doing a summation, and this is the computation that I can parallelize. Then I essentially want to do a reduction for the plus operator since I'm doing an addition on this variable.

This loop here is parallel. It's data parallel. I can split it up. The for loop is also -- I can do this work sharing on it. So I use the parallel for pragma. And the variable x here is private. It's defined here but I can essentially give a directive that says, this is private. You can essentially rename it on each processor. Its value won't have any effect on the overall computation because each computation will have its own local copy. That clear so far?

So computing pi with integration using MPI takes up two slides. You know, I could fit it on one slide but you couldn't see it in the back. So there's some initialization. In fact, I think there's only six basic MPI commands that you need for computing. Three of them are here and you'll see the others are MPI send and MPI receive. And there's one more that you'll see on the next slide.

So there's some loop that says while I'm not done keep computing. And what you do is you broadcast n to all the different processors. N is really your time step. How many small intervals of execution are you going to do? And you can go through, do your computation. So now this -- the MPI essentially encapsulates the computation over n processors. And then you get to an MPI reduce command at some point that says, OK, what values did everybody compute? Do the reduction on that. Write that value into my MPI.

Now what happens here is there's processor ID zero which I'm going to consider the master. So he's the one who's going to actually print out the value. So the reduction essentially synchronizes until everybody's communicated a value to processor zero. And then it can print out the pi. And then you can finalize, which actually makes sure the computation can exit. And you can go on and terminate.

So the last concept in terms of understanding performance for parallelism is this notion of locality. And there's locality in your communication and locality in your computation. So what do I mean by that? So in terms of communication, you know, if I have two operations and let's say -- this is a picture or schematic of what the MIT raw chip looks like. Each one of these is a core. There's some network, some basic computation elements. And if I have, you know, an addition that feeds into a shift, well, I can put the addition here and the shift there, but that means I have a really long path that I need to go to in terms of communicating that data value around. So the computation naturally should just be closer together because that decreases the latency that I need to communicate. So rather than doing net mapping, what I might want to do is just go to somebody who is close to me and available.

AUDIENCE: Also there are volume issues. So assume more than that. A lot of other people also want to communicate. So if [UNINTELLIGIBLE] randomly distributed, you can assume there's a lot more communication going into the channel. Whereas if you put locality in there then you can scale communication much better than scaling the network.

PROFESSOR: There's also a notion of locality in terms of memory accesses. And these are potentially also very important or more important, rather, because of the latencies for accessing memory. So if I have, you know, this loop that's doing some addition or some computation on an array and I distribute it, say, over four processors -- this is, again, let's assume a data parallel loop. So what I can do is have a work sharing mechanism that says, this thread here will operate on the first four indices. This thread here will operate on the next four indices and the next four and the next four. And then you essentially get to join barrier and then you can continue on.

And if we consider how the access patterns are going to be generated for this particular loop, well, in the sequential case I'm essentially generating them in sequence. So that allows me to exploit, for example, on traditional [? CAT ?] architecture, a notion of spatial locality. If I look at how things are organized in memory, in the sequential case I can perhaps fetch an entire block at a time. So I can fetch all the elements of A0 to A3 in one shot. I can fetch all the elements of A4 to A7 in one shot. And that allows me to essentially improve performance because I overlap communication. I'm predicting that once I see a reference, I'm going to use data that's adjacent to it in space.

There's also a notion of temporal locality that says that if I use some particular data element, I'm going to reuse it later on. I'm not showing that here. But in the parallel case what could happen is if each one of these threads is requesting a different data element -- and let's say execution essentially proceeds -- you know, all the threads are requesting their data at the same time. Then all these requests are going to end up going to the same memory bank. The first thread is requesting ace of zero. The next thread is requesting ace of four, the next thread ace of eight, next thread ace of 12. And all of these happen to be in the same memory bank.

So what that means is, you know, there's a lot of contention for that one memory bank. And in effect I've serialized the computation. Right? Everybody see that? And, you know, this can be a problem in that you can essentially fully serialize the computation in that, you know, there's contention on the first bank, contention on the second bank, and then contention on the third bank, and then contention on the fourth bank.

And so I've done absolutely nothing other than pay overhead for parallelization. I've made extra work for myself [? concreting ?] the threads. Maybe I've done some extra work in terms of synchronization. So I'm fully serial.

So what you want to do is actually reorganize the way data is laid out in memory so that you can effectively get the benefit of parallelization. So if you have the data organized as is there, you can shuffle things around. And then you end up with fully parallel or a layout that's more amenable to full parallelism because now each thread is going to a different bank. And that essentially gives you a four-way parallelism. And so you get the performance benefits.

So there are different kinds of sort of considerations you need to take into account for shared memory architectures in terms of how the design affects the memory latency. So in a uniform memory access architecture, every processor is either, you can think of it as being equidistant from memory. Or another way, it has the same access latency for getting data from memory. Most shared memory architectures are non-uniform, also known as NUMA architecture. So you have physically partitioned memories. And the processors can have the same address space, but the placement of data affects the performance because going to one bank versus another can be faster or slower. So what kind of architecture is Cell? Yeah. No guesses?

AUDIENCE: It's not a shared memory.

PROFESSOR: Right. It's not a shared memory architecture. So a summary of parallel performance factors. So there's three things I tried to cover. Coverage or the extent of parallelism in the application. So you saw Amdahl's Law and it actually gave you a sort of a model that said when is parallelizing your application going to be worthwhile? And it really boils down to how much parallelism you actually have in your particular algorithm. If your algorithm is sequential, then there's really nothing you can do for programming for performance using parallel architectures.

I talked about granularity of the data partitioning and the granularity of the work distribution. You know, if you had really fine-grain things versus really coarse-grain things, how does that translate to different communication costs? And then last thing I shared was locality. So if you have near neighbors talking, that may be different than two things that are further apart in space communicating. And there are some issues in terms of the memory latency and how you actually can take advantage of that. So this really is an overview of sort of the parallel programming concepts and the performance implications. So the next lecture will be, you know, how do I actually parallelize my program? And we'll talk about that.