PROFESSOR: OK. So today we're going to continue on with some of the design patterns that we started talking about last week. So to recap, there are really four common steps to taking a program and then parallelizing it. Often you're starting off with a program that's designed or written in a sequential manner. And what you want to do is find tasks in the program -- and these are sort of independent work pieces that you are going to be able to decompose from your sequential code. You're going to group tasks together into threads or processes. And then you'll essentially map each one of these threads or processes down to the actual hardware. And that will get you, eventually when these programs run, the concurrency and the performance speedups that you want.

So as a reminder of what I talked about last week in terms of finding the task or finding the concurrency, you start off with an application. You come up with a block level diagram. And from that you sort of try to understand where the time is spent in the computations and what are some typical patterns for how the computations are carried out. So we talked about task decomposition or sort of independent tasks or tasks that might be different that the application is carrying out.

So in the MPEG encoder, we looked at decoding the motion vectors for temporal compression versus spatial compression. It does sort of substantially different work. We talked about data decomposition. So if you're doing a process -- so if you have some work that's really consuming a large chunk of data, and you realize that it's applying the same kind of work to each of those data pieces, then you can partition your data into smaller subsets and apply the same function over and over again. So in the motion compensation phase, that's one example where you can replicate the function and split up the data stream in different ways and have these tasks proceed in parallel. So that's data decomposition.

And then we talked a little bit about sort of making a case for a pipeline decomposition. So you have a data assembly line or producer-consumer chains, and you essentially want to recognize those in your computation and make it so that you can exploit them eventually when you're doing your mapping down to actual hardware.

But what does it mean for two tasks to actually be concurrent? And how do you know that you can safely actually run two tasks in parallel? So there's something I crudely went over last time. So as to make it more concrete, highlighting Bernstein's condition, which says that given two tasks, if the input set to one task is different from or does not intersect with the output set of another, and vice versa, and neither task sort of updates the same data structures in memory, then there's really no dependency issues between them. You can run them safely in

parallel.

So task T1 and T2, if all the data that's consumed by T1, so all the data elements that are read by T1 are different from the ones that are read by T2, then you have -- you know, if T2 is running in parallel, there's really no problem with T1 because it's updating the orthogonal data set. Similarly for T2 and T1, any outputs are different.

So as an example, let's say you have two tasks. In T1 you're doing some basic statements. And these could be essentially more coarse grained. There could be a lot more computation in here. I just simplified it for the illustration. So you have task a equals x plus y. And task two does b equals x plus z. So if we look at the read set for T1, these are all the variables or data structures or addresses these that are read by the first task. So that's x and y here. And all the data that's written or produced by T1. So here we're just producing one data value. And that's going into location A.

Similarly we can come up with the read set and write set for T2. And so that's shown on here. So we have -- task T2 has x plus z in its read set. And it produces one data value, b. If we take the intersection of the read and write sets for the different tasks, then they're empty. I read something completely different than what's produced in this task and vice versa. And they write to two completely different memory locations. So I can essentially parallelize these or run these two tasks in parallel.

So you can extend this analysis. And compilers can actually use this condition to determine when two tasks can be parallelized if you're doing automatic parallelization. And you'll probably hear more about these later on.

And so what I focused on last time were the finding concurrency patterns. And I had identified sort of four design spaces based on the work that's outlined in the book by Mattson, Sanders, and Massingill. And so starting with two large sort of concepts. The first helps you figure out how you're going to actually express your algorithm. So first you find your concurrency and then you organize in some way. And so we're going to talk about that in more detail.

And then once you've organized your tasks in some way that actually expresses your overall computation, you need some software construction utilities or data structures or mechanisms for actually orchestrating computations for which they have also abstracted out some common patterns. And so I'll briefly talk about these as well.

And so on your algorithm expression side, these are essentially conceptualization steps that help you abstract out your problem. And you may in fact think about your algorithm expression in different ways to expose different kinds of concurrency or to be able to explore different ways of mapping the concurrency to hardware. And so for construction it's more about actual engineering and implementation. So here you're actually thinking about what

do the data structures look like? What is the communication pattern going to look like? Am I but going to use things like MPI or OpenMP? What does that help me with in terms of doing my implementation?

So given a collection of concurrent tasks -- so you've done your first step in your four design patterns. You know, what is your next step? And that's really mapping those tasks that you've identified down to some sort of execution units. So threads are very common. This is essentially what we've been using on Cell. We take our computation and we wrap it into SPE threads and then we can execute those at run time.

So some things to keep in mind -- although you shouldn't over constrain yourself in terms of these considerations. What is the magnitude of your parallelism that you're going to get? You know, do you want hundreds or thousands of threads? Or do you want something on the order of tens? And this is because you don't want to overwhelm the intended system that you're going to run on. So we talked about yesterday on Cell processor, if you're creating a lot more than six threads, then you can create problems or you essentially don't get extra parallelism because each thread is running to completion on each SPE. And contact switch overhead is extremely high.

So you don't want to spend too much engineering cost to come up with an algorithm implementation that's massively scalable to hundreds or thousands of threads when you can't actually exploit it. But that doesn't mean that you should over constrain your implementation to where if now I want to take your code and run it on a different machine, I essentially have to redesign or re-engineer the complete process. So you want to avoid tendencies to over constrain the implementation. And you want to leave your code in a way that's malleable so that you can easily make changes to sort of factor in new platforms that you want to run on or new machine architecture features that you might want to exploit.

So there are three major organization principles I'm going to talk about. And none of these should be sort of foreign to you at this point because we've talked about them in different ways in the recitations or in previous lectures. And it's really, what is it that determines sort of the algorithm structure based on the set of tasks that you're actually carrying out in your computation?

And so there's the principle that says, organize things by tasks. I'm going to talk to that. And then there's a principle that says, well, organize things by how you're doing the data decomposition. So in this case how you're actually distributing the data or how to the data is laid out in memory, or how you're partitioning the data to actually compute on it dictates how you should actually organize your actual computation.

And then there's organize by flow of data. And this is something you'll hear about more in the next lecture where we're talking about streaming. But in this pattern if there are specific sort of computations that take advantage of high bandwidth flow of data between computations, you might want to exploit that for concurrency. And we'll talk about that as well.

OK. So a design diagram for how can you actually go through process. So you can ask yourself a set of questions. if I want to organize things by tasks, then there are essentially two main clusters or two main computations, two main patterns. If the code is recursive, then you essentially want to apply a divide and conquer pattern or divide and conquer organization. If it's not recursive. then you essentially want to do task parallelism.

So in task parallelism -- you know, I've listed two examples here. But really any of the things that we've talked about in the past fit. Ray computation, ray tracing. So here you're shooting rays through a scene to try to determine how to render it. And really each ray is a separate and independent computation step. In molecular dynamics you're trying to determine the non-bonded force calculations. There are some dependencies, but really you can do each calculation for one molecule or for one atom independent of any other. And then there are sort of the global dependence of having to update or communicate across all those molecules that sort of reflect new positions in the system.

So the common factors here are your tasks are associated with iterations of a loop. And you can distribute, you know, each process -- each processor can do a different iteration of the loop. And often you know sort of what the tasks are before you actually start your computation. Although in some cases, like in ray tracing, you might generate more and more threads as you go along, or more and more computations because as the ray is shooting off, you're calculating new reflections. And that creates sort of extra work. But largely you have these independent tasks that you can encapsulate in threads and you run them.

And this is sort of -- it might appear subtle, but there are algorithm classes where not all tasks essentially need to complete for you to arrive at a solution. You know, in some cases you might convert to an acceptable solution. And you don't actually need to go through and exercise all the computation that's outstanding for you to say the program is done. So there will be a tricky issue -- I'll revisit this just briefly later on -- is how do you determine if your program has actually terminated or has completed?

In divide and conquer, this is really for recursive programs. You know, you can think of a well-known sorting algorithm, merge sort, that classically fits into this kind of picture, where you have some really large array of data that you want to sort. You keep subdividing into smaller and smaller chunks until you can do local reorderings. And then you start merging things together. So this gives you sort of a way to take a problem, divide it into subproblems. And then you can split the data at some point and then you join it back together. You merge it. You might see things like fork and merge or fork and join used instead of split and join. I've used the terminology that sort of melds well with some of the concepts we use in streaming that you'll see in the next lecture.

And so in these kinds of programs, it's not always the case that each subproblem will have essentially the same amount of work to do. You might need more dynamic load balancing because each subproblem -- how you

distribute the data might lead you to do more work in one problem than in the other. So as opposed to some of the other mechanisms where static load balancing will work really well -- and to remind you, static load balancing essentially says, you have some work, you assign it to each of the processors. And you're going to be relatively happy with how each processor's sort of utilization is going to be over time. Nobody's going to be too overwhelmed with the amount of work they have to do. In this case, you might end up with needing some things for dynamic load balancing that says, I'm unhappy with the work performance or utilization. Some processors are more idle than the others, so you might want to essentially redistribute things.

So what we'll talk about -- you know, how does this concept of divide and conquer parallelization pattern work into the actual implementation? You know, how do I actually implement a divide and conquer organization?

The next organization is organized by data. So here you have some computation -- not sure why it's flickering.

AUDIENCE: Check your -- maybe your VGA cables aren't in good.

PROFESSOR: So in the organize by data, you essentially want to apply this if you have a lot of computation that's using a shared global data structure or that's going to update a central data structure. So in molecular dynamics, for example, you have a huge array that records the position of each of the molecules. And while you can do the coarse calculations independently, eventually all the parallel tasks have to communicate with the central data structure and say, here are the new locations for all the molecules. And so that has to go into a central repository.

And there are different kinds of sort of decompositions within this organization. If your data structure is recursive, so a link list or a tree or a graph, then you can apply the recursive data pattern. If it's not, if it's linear, like an array or a vector, then you apply geometric decomposition. And you've essentially seen geometric decomposition. These were some of the labs that you've already done.

And so the example from yesterday's recitation, you're doing an end body simulation in terms of who is gravitating towards who, you're calculating the forces between pairs of objects. And depending on the force that each object feels, you calculate a new motion vector. And you use that to update the position of each body in your, say, galaxy that you're simulating. And so what we talked about yesterday was given an array of positions, each processor gets a sub-chunk of that position array. And it knows how to calculate sort of locally, based on that. And then you might also communicate local chunks to do more scalable computations.

And recursive data structure are a little bit more tricky. So at face value you might think that there's really no kind of parallelism you can get out of a recursive data structure. So if you're iterating over a list and you want to get the sum, well, you know, I just need to go through the list. Can I really parallelize that? There are, however, opportunities where you can reshape the computation in a way that exposes the concurrency. And often what this

comes down to is you're going to do more work, but it's OK because you're going to finish faster. So this kind of work/concurrency tradeoff, I'm going to illustrate with an example.

So in this application we have some graphs. And for each node in a graph, we want to know what is its root? So this works well when you have a forest where not all the graphs are connected and given a node you want to know who is the root of this graph. So what we can do is essentially have more concurrency by changing the way we actually think about the algorithm.

So rather than starting with each node and then, in a directed graph, following its successor -- so this is essentially order n, because for each node we have to follow n links -- we can think about it slightly differently. So what if rather than finding the successor and then finding that successor's successor, at each computational step we start with a node and we say who is your successor's successor?

So we can converge in this example in three steps. So from five to six we can say who is this successor? So who is the successor's successor of five? And that would be two. And similarly you can do that for seven and so on. And so you keep asking the question. So you can distribute all these data structures, repeatedly ask these questions out of all these end nodes, and it leads you to an order log n solution versus an order n solution.

But what have I done in each step? Well, I've actually created myself and I've sort of increased the amount of work that I'm doing by order n. Right there. Right. Yes. Because I've essentially for each node doing n queries, you know, who's your successor's successor? Whereas in a sequential case, you know, I just need to do it once for each node. And that works really well.

So most strategies based on this pattern of actually decomposing your computation according to recursive pattern lead you to doing much more work or some increase in the amount of work. But you get this back in because you can decrease your execution. And so this is a good tradeoff that you might want to consider.

AUDIENCE: In the first one order n was sequential?

PROFESSOR: Yeah, yeah. It's a typo. Yeah. So organize by flow or organize by flow of data. And this is essentially the pipeline model. And we talked about this again in some of the recitations in terms of SPE to SPE communication. Or do you want to organize based on event-based mechanisms? So what these really come down to is, well, how regular is the flow of data in your application? If you have regular, let's say, one-way flow through a stable computation path -- so I've set up my sort of algorithm structure. Data is flowing through it at a regular rate. The computation graph isn't changing very much. Then I can essentially pipeline things really well. And this could be a linear chain of computation or it could be sort of nonlinear. There could be branches in the graph. And I can use that in a way to exploit pipeline parallelism.

If I don't have sort of this nice, regular structure, it could be events that are created at run time. So, for example, you're a car wash attendant and a new car comes in. So you have to find a garage to assign to it and then turn on the car wash machine. So the dynamic threads are created based on sensory input that comes in, then you might want to use an events-based coordination. You have irregular computation. The computation might vary based on the data that comes into your system. And you might have unpredictable data flow.

So in the pipeline model, the things to consider is the pipeline throughput versus the pipeline latency. So the amount of concurrency in a pipeline is really limited by the number of stages. This is nothing new. You've seen this, for example, in super scaled pipelines. And just as in this case, as in the case of an architecture pipeline, the amount of time it takes you to fill the pipeline and the amount of time it takes you to drain the pipeline can essentially limit your parallelism. So you want those to be really small compared to the actual computation that you spend in your pipeline.

And the performance metric is usually the throughput. How much data can you pump through your pipeline per unit time? So in video encoding, you know, it's the frames per second that you can produce. And the pipeline latency, though, is important, especially in a real-time application where you need a result every 10 milliseconds. You know, your pacemaker for example has to produce a beep or a signal to your heart at specific rates. So you need to consider what is your pipeline throughput versus your pipeline latency? And that can actually determine how many stages you might want to actually decompose or organize your application in.

And in the event-based coordination, these are interactions of tasks over unpredictable intervals. And you're more prone to sort of deadlocks in these applications. Because you might have cyclic dependencies where one event can't proceed until it gets data from another event. But it can't proceed until it gets data from another event. You can create sort of these complex interactions that often lead to deadlock. So you have to sort of be very careful in structuring things together so you don't end up with feedback loops that block computation progress.

So given sort of these three organizational structures that say, you know, I can organize my computation by task or by the flow of data or by sort of the pipeline nature of the computation, what are the supporting structures? How do I actually implement these? And so there are many different supporting structures. I've identified sort of four that occur most often in literature and in books and common terminology that's used. And so those are SPMD, loop parallelism, the master/worker pattern, and the fork/join pattern.

In the SPMD pattern, you're talking about a single program, multiple data concept. So here you just have one program. You write it once and then you assign it to each of your processors to run. So it's the same program. It just runs on different machines. Now each program or each instance of the code can have different control flow that it takes. So just because they're running the same program doesn't mean the computation is happening in

lock step. That would be a sort of a SIMD or vector-like computation.

In this model you can actually take independent control flow. It could be different behavior in each instance of the code. But you're running the same code everywhere. So this is slightly different, for example, from what you've seen on Cell, where you have the PPE thread that creates SPE threads. Sometimes the SPE threads are the same, but it's not always the case that the PPE threads and the SPE threads are the same.

So in the SPMD model there are really five steps that you do. You initialize sort of your computation in the world of sort of code instances that you're going to run. And for each one you obtain a unique identifier. And this usually helps them being able to determine who needs to communicate with who or ordering dependencies. And you run the same program on each processor. And what you need to do in this case is also distribute your data between each of the different instances of your code. And once, you know, each program is running, it's computing on its data, eventually you need to finalize in some way. And so that might mean doing a reduction to communicate all the data to one processor to actually output the value.

And so we saw in SPMD an example for the numerical integration for calculating pi. And if you remember, so we had this very simple c loop. And we showed the MPI implementation of the c loop. And so in this code, what we're doing is we're trying to determine different intervals. And for each interval we're going to calculate a value and then in the MPI program we're essentially deciding how big an interval each process should run. So it's the same program. It runs on every single machine or every single processor. And each processor determines based on its ID which interval of the actual integration to do. And so in this model we're distributing work relatively evenly. Each processor is doing a specific chunk that starts at say some index i. And if I have 10 processors, I'm doing 100 steps. Then you're doing i, i plus 10, i plus 20 and so on.

But I can do a different distribution. So the first is a block distribution. I can do something called a cyclic distribution. So in a cyclic distribution, I distribute work sort of in a round robin fashion or some other mechanism. So here, you know, each processor -- sorry. In the block distribution I sort of start at interval i and I go -- sorry. So each processor gets one entire slice here. So I start here and I go through to completion. I start here and go through to completion. In a cyclic distribution I might do smaller slices of each one of those intervals. And so I greyed out the components for the block distribution to show you that for a contrast here.

There are some challenges in the SPMD model. And that is how do you actually split your data correctly? You have to distribute your data in a way that, you know, doesn't increase contention on your memory system, where each actual processor that's assigned the computation has data locally to actually operate on. And you want to achieve an even work distribution. You know, do you need a dynamic load balancing scheme or can you use an alternative pattern if that's not suitable?

So the second pattern, as opposed to the SPMD pattern is loop parallelism pattern. In this case, this is the best suited when you actually have a programming model or a program that you can't really change a whole lot or that you don't really want to change a whole lot. Or you have a programming model that allows you to sort of identify loops that take up most of the computation and then insert annotations or some ways to automatically parallelize those loops.

So we saw in the OpenMP example, you have some loops you can insert these pragmas that say, this loop is parallel. And the compiler in the run-time time system can automatically partition this loop into smaller chunks. And then each chunk can compute in parallel. And you might apply this scheme in different ways depending on how well you understand your code. Are you running on a shared memory machine? You can't afford to do a whole lot of restructuring. Communication costs might be really expensive.

In the master/worker pattern, this is really starting to get closer to what we've done with the Cell recitations in the Cell labs. You have some world of independent tasks and the master essentially running and distributing each of these tasks to different processors. So in this case you'd get several advantages that you can leverage. If each of your tasks are varied in nature -- and they might finish at different times or they require different kinds of resources, you can use this model to sort of view your machine as sort of a non-symmetric processor. Not everybody is the same. And you can use this model really well for that.

So you can distribute these and then you can do dynamic load balancing. Because as processors -- as workers finish you can ship them more and more data. So it has some particularly relevant properties for heterogeneous computations, but it's also really good for when you have a whole lot of parallelism in your application. So something called embarrassingly parallel problems. So ray tracing, molecular dynamics, a lot of scientific applications have these massive levels of parallelism. And you can use this essentially work-queue based mechanism that says I have all these tasks and I'll just dispatch them to workers and compute.

And as I pointed out earlier, you know, when do you define your entire computation to have completed? You know, sometimes you're computing a result until you've reached some result. And often you're willing to accept a result within some range of error. And you might have some more threads that are still in flight. Do you terminate your computation then or not? What are some issues with synchronization? If you have so many threads that are running together, you know, does the communication between them to send out these control messages say, I'm done, start to overwhelm you?

In the fork/join pattern -- this is really not conceptually too different in my mind from the master/worker model, and also very relevant to what we've done with Cell. The main difference might be that you have tasks that are dynamically created. So in the embarrassingly parallel case, you actually know the world of all your potential task

that you're going to run in parallel. In the fork/join join model some new computation might come up as a result of, say, an event-based mechanism. So a task might be created dynamically and then later terminated or they might complete. And so new ones come up as a result.

AUDIENCE: It almost seems like you are forking the task in the forking model. And then keep assigning tasks to that. The fork/join model you just keep forking at first virtual box. Might not be completely matched to a number of processor available. fork them out.

PROFESSOR: So the process that's equating all these threads or that's doing all the forking is often known as the parent and the tasks that are generated are the children. And eventually essentially the parent can't continue or can't resume until its children have sort of completed or have reached the join point. And so those are really some of the models that we've seen already, in a lot of cases in the recitations and labs for how you run your computations. And some of you have already discovered these and actually are thinking about how your projects should be sort of parallelized for your actual Cell demos.

Some of the other things that I'm just going to talk about are communication patterns. So two lectures ago you saw, for example, that you have point to point communication or you have broadcast communication. So in point to point communication, you have two tasks that need to communicate. And they can send explicit messages to each other. These could be control messages that say I'm done or I'm waiting for data. Or they could be data messages that actually ships you a particular data element that you might need. And again we've seen this with Cell.

Broadcast says, you know, I have some result that everybody needs. And so I send that out to everybody by some mechanism. There is no real broadcast mechanism on Cell. The concept I'm going to talk about though is the reduction mechanism, which really is the inverse of the broadcast. So in the broadcast I have a data element I need to send to everybody else. In the reduction, all of you have data that I need or all of us have data that each somebody else needs. So what we need to do is collectively bring that data together or group it together and generate an end result.

So a simple example of a reduction, you have some array of elements that you want to add together. And sort of the result of the collective operation is the end sum. So you have an array of four elements, A0, A1, A2, and A3. And you can do a serial reduction. I can take A0 and add it to A1. And that gives me a result. And I can take A2 and add that to it. And I can take A3 and add that to it. And so at the end I'll have sort of calculated the sum from A0 to A3.

So this is essentially -- the serial reduction applies when your operation is an associative. So the addition is associative. So in this case I can actually do something more intelligent. And I think we talked about that last time.

I'm going to show you some more examples. And often sort of the end result follows a broadcast. It says, here is the end result. Who are all the people that need it? I'll sort of broadcast that out so that everybody has the result. If your operation isn't associative, then you're essentially limited to a serial process. And so that's not very good from a performance standpoint.

Some of the tricks you can apply for actually getting performance out of your reduction is to go to a tree-based reduction model. So this might be very obvious. Rather than doing A0 and A1 and then adding A2 to that result, I can do A0 and A1 together. In parallel I can do A2 and A3. And then I can get those results and add them together. So rather than doing n steps I can do log n steps. So this is particularly attractive when only one task needs the result. So in the MPI program when we're doing the integration to calculate pi, you know, one processor needs to print out that value of pi.

But if you have a computation where more than one process actually needs the result of the reduction, there's actually a better mechanism you can use that's sort of a better alternative to the tree-based reduction followed by a broadcast. So you can do a recursive doubling reduction. So at the end here, every process will have the result of the reduction without having done the broadcast. So we can start off as with the tree-based and add up A0 and A1 together. But what we do is for each process that has a value, we sort of do a local exchange. So from here we communicate the value to here. And from here we communicate the value to here. And so now these two processors that had the value independently now both have a local sum, A0 to A1. And similarly we can sort of make the similar symmetric computation on the other side.

And now we can communicate data from these two processors here to come up with the end --

PROFESSOR: It was there. All right. Must have been lost in the animation. So you actually do that the other way as well so that you have the sum A0 to A3 on all the different processors. Sorry about the lost animation. OK. So this is better than the tree-based approach with a broadcast because you end up with local results of your reduction. And rather than doing the broadcast following the tree-based reduction which takes n steps, you end up with an order n. Everybody has a result in order n versus an order 2n process for the tree-based plus broadcast.

AUDIENCE: On the Cell processor but not in general.

PROFESSOR: Not in general. It depends on sort of the architectural mechanism that you have for your network. If you actually do need to sort of, you know, if you have a broadcast mechanism that has bus-based architecture where you can deposit a local value, everybody can pull that value, then, yeah, it can be more efficient. Or on optical networks, you can broadcast the data and everybody can just fuse it out. OK.

So summarizing all the different patterns, so here these are the actual mechanisms that you would use for how

you would implement the different patterns. So in the SPMD you would write the same program. In loop parallelism you have your program and you might annotate sort of some pragmas that tell you how to parallelize your computation. In the master/worker model you might have sort of a master that's going to create threads and you actually know -- you might sort of have a very good idea of what is the kind of work you're going to have to do in each thread. In the fork/join model you have more dynamism. So you might create threads on the fly.

And you apply these sort of based on appeal or what is more suited in terms of implementation to each of the different patterns for how you actually organize your data. So in the task parallelism model, this is where you have a world of threads that you know you're going to calculate or that you're going to use for your computation. And really you can use largely any one of these models. So I used a ranking system where four stars is really good. One star is sort of bad or no star means not well suited.

AUDIENCE: Sort of in Cell because the inherit master there. Sometimes master/worker might get a little bit of a biasing than this one.

PROFESSOR: Right, so --

AUDIENCE: You don't have to pay a cost of having master

PROFESSOR: Right. Right. Although you could use the Cell master to do regular computations as well. But, yes. So and the divide and conquer model, you know, might be especially well suited for a fork and join because you're creating all these recursive subproblems They might be heterogeneous. In the nature of the computation that you do, you might have more problems created dynamically. Fork/join really works well for that. And the fact, you know, the subproblem structure that I showed, the graph of sort of division. And then merging works really well with the fork/join model.

In the recursive, in the geometric decomposition -- this is essentially your lab one exercise and the things we went over yesterday in the recitation. You're taking data and you're partitioning over multiple processors to actually compute in parallel. So this could be SPMD implementation or it could be a loop parallelism implementation, which we didn't do. Less suitable, the master/worker and fork/join, often because the geometric decomposition applied some distribution to the data which has static properties that you can exploit in various ways. So you don't need to pay the overhead of master/worker or fork/join.

Recursive data structures sort of have very specific models that you can run with. Largely master/worker is a decent implementation choice. SPMD is another. And you're going to hear more about sort of the pipeline mechanism in the next talk so I'm not going to talk about that very much. Event-based coordination, largely dynamic. So fork/join works really well. So one --

AUDIENCE: When you're buffering them you could do master/worker with pipelining?

PROFESSOR: Yes, so next slide. So sort of these choices or these tradeoffs aren't really orthogonal. You can actually combine them in different ways. And in a lot of applications what you might find is that the different patterns compose hierarchically. And you actually want that in various ways -- for various reasons. So in the MPEG example, you know, we had tasks here within each task and identified some pipeline stages. You know, here I have some data parallelism so I can apply the loop pattern here.

And what I want to do is actually in my computation sort of express these different mechanisms so I can understand sort of different tradeoffs. And for really large applications, there might be different patterns that are well suited for the actual computation that I'm doing. So I can combine things like pipelining with a task-based mechanism or data parallelism to actually get really good performance speedups.

And one of the things that might strike you as well, heck, this is a whole lot of work that I have to do to actually get my code in the right way so that I can actually take advantage of my parallel architecture. You know, I have to conceptually think about the question the right way. I have to maybe restructure my computation in different ways to actually exploit parallelism. Data distribution is really hard. I have to get that right. Synchronization issues might be a problem. And how much buffering do I need to do between different tasks?

So the thing you're going to hear about in the next talk is, well, what if these things really fall out naturally from the way you actually write the program, and if the way you actually write your program matches really well with the intuitive, sort of natural conceptualization of the problem. And so I'll leave Bill to talk about that. And I'm going to stop here. Any questions?

AUDIENCE: We can take in some questions and then everybody --

AUDIENCE: You talked about fork and join. When you have a parent thread that spawns off to a child thread, how do you keep your parent thread from using up the SPE?

PROFESSOR: So you have a fork/join where you have --

AUDIENCE: Most of the parents it might be the PPE. And so if you just do fork/join, might not really use PPE unless you can, you know, you have some time and you let it do some of the task and come back.

AUDIENCE: So for our purposes we shouldn't spawn off new threads by the SPEs?

PROFESSOR: So, yeah. So most of the threads that are spawned off are done by the PPE. So you have these -- in fact a good walk through in recitation yesterday. You have the PPE. Essentially it sends messages to the SPEs

that says, create these threads and start running them. Here's the data for them. And then these threads run on the SPEs. And they just do local computation. And then they send messages back to the PPE that says, we're done. So that essentially implements the join mechanism.

AUDIENCE: On the other hand, if you are doing something like master slave way, and then the SPE can send a message and deliver another job into the PPE who feeds the master. If SPE see there's some more computers, you can say, OK, look, put this into your keybord and keep sending messages and so the master can look at that and update it. So, you know, it's not only master who has to fork off but the slaves also. They still can send information back. So you can think about something like very confident that way. There are eight -- like if six SPE is running and you first get something in there and SPE says divide it will take one task and run that until the other one to the master will finish it and here's my ID. Send me the message when it's done. And so you fork that end and wait. So you can assume you can do something like that.

So it's almost master/slave but the coordination is there. The trouble with normally fork/join is if you create too many threads. You are in like a thread hell because there are too many things to run. I don't know, can you SPE?

PROFESSOR: No.

AUDIENCE: So you can't even do that because of some physical limitation. You can't get take up 1000 threads you run another master/slave thing yourself is because 1000 threads on top of your SPEs. And that's going to be locked threads.

PROFESSOR: Yeah. Contact switching on the SPEs is very expensive. So on the PlayStation 3 you have six SPEs available to you. So if you have a lot more than six threads that you've created, essentially each one runs to completion. And then you swap that out and you bring in -- well, that terminates. You deallocate it from the SPE and you bring in a new thread. If you actually want to do more thread-like dynamic load balancing on the SPEs, it's not well suited for that. Just because the --

AUDIENCE: The best model there is master/slave. Because the PPE [UNINTELLIGIBLE PHRASE] the master part. It will run more sequential code. And when there's parallel send -- it will give it to you and produce the work queue model type and send stuff into SPE and feed that. So work queue type models can be used there.

PROFESSOR: Yeah. And the SPMD model might not work really well because you have this heterogeneity in the actual hardware, right. So if I'm taking the same program running on the SPE versus the PPE, that code might not be -- so I essentially have to specialize the code. And that starts to deviate away from the SPMD model.

AUDIENCE: Something I think most of the code you write for Cell will probably be master/worker. And if you try to do something other than you should think hard why that's the case.

PROFESSOR: You can do fork/join but you know, it's --

AUDIENCE: I mean you can't -- because you don't have virtualization. If you fork too much where are you going to put those?

PROFESSOR: Right. Sometimes you fork -- AUDIENCE: Yeah. but in that sense you -- should you fork too much? To keep work you want the master. You can fork things -- you can do virtual fork and send the work to the master and say, here, I forked something. Here's the work. I mean, the key thing is do the simplest thing. I mean, you guys have two weeks left. And if you try doing anything complicated, you might end up with a big mess that's undebuggable. Just do simple things.

And I can vouch, parallelism is hard. Debugging parallel code is even harder. So you're sort of trying to push the limits on the complexity of messages going all over the world and the three different types of parallelism all trying to compete in there. Just do the simple thing. Just get the simple thing working. First get the sequential code working and keep adding more and more story. And then make sure that each level it works. The problem with parallelism is because things that determine if some bugs that might show up. Data might be hard. But design absolutely matters.

Another thing I think, especially for doing demos and stuff would be nice, would be to have a knob that basically you can tune. So you can say, OK, no SPEs. Everything running in PPE one is two is. So you can actually see hopefully in your code how how things move for the demo part.

PROFESSOR: You had a question? All right. We'll take a brief break and do the quizzes in the meantime.