PROFESSOR: So in this second lecture we're going to talk about some design patterns for parallel programming. And to tell you a little bit about what a design pattern is and why is it useful. And some of you, if you've taken object oriented programming you've probably already have seen design patterns before. So I ended the last lecture with OK, so I understand some of the performance implications, how do I go about parallelizing my program? This is a flyer I found quite often in books and talks on parallel programming. It essentially lays out 4 common steps for parallelizing your program. So often, you start out with sequential programs. This shouldn't be surprising since for a long time as you've heard in earlier lectures, people just coded sequential code and that was just good enough. So now the problem is you want to take that sequential code or you want to still write sequential code just because it's conceptually easier and you want to be able to parallelize it so you can map it down to your parallel architecture, which in this example has 4 processors.

So the first step is you take your sequential program and you divide it up into tasks. So during the project reviews, for example, yesterday when I talked to each team individually we sort of talked about this and you sort of stumbled on these 4 steps whether you realized it or not. And so you come up with these tasks and then each one essentially incapsulates some computation. And then you group them together, so this is some granularity adjustment and you map them down to processes. These are things you can pose into threads, for example. And then you have to essentially plot these down onto actual processors and they have to talk with each other, so you have to orchestrate to communication and then finally do the execution. So step through each one of these at a time. Sort of composition and recall that just really effects or just really is effected by Amdahl's Law. And that if there's not a whole lot of parallels in the application your decomposition is a waste of time. There's not really a whole lot to get. But what you're trying to do is identify concurrency in your application and figure out at what level to exploit it. So what you're trying to do is divide up your computation into tasks and eventually these are going to be distributed among processors and you want to find enough of them so that you can keep all the processors busy. And remember that the more of these that you have this gives you sort of an upper bound on your potential speed up. And as in the rate tracing example that I showed, the number of tasks that you have may vary run time. So sometimes you might have lot of arrays bouncing off a lot of things, and sometimes you might not have a whole lot of reflection going on so number of arrays will change over time.

And in other applications, interactions, for example, between molecules might change in a molecular dynamic simulator. The assignment really effects granularity. This is where you've partitioned your tasks, you're trying to

group them together so that you're taking into account, what is the communication cost going to be? What kind of locality am I going to deal with? And what kind of synchronization mechanisms do I need and how often do I need to synchronize? You adjust your granularity such that you end up with things that are load balanced and you try to reduce communication as much as possible. And structured approaches might work well. You might look at the code, do some inspection, you might understand the application, but there are some well-known design patterns which is essentially the thing we're going to get to try to help you with this.

As programmers really, I think, we worry about partitioning first. This is really independent of an architecture programming model. Just taking my application and figuring out well, what are different parts that I need to compose together to build my application? So I'm going to show you an example of that. And one thing to keep in the back of your mind is that the complexity of how much partitioning work you actually have to do really effects your decision. So if you start out with some piece of code or you wrote your code in one way and you realize that to actually parallelize it requires so much more work, in some user studies we've done on sort of trying to get performance from code, it really effects how much work you actually do. And if something requires a lot of work, you might not do it even though it might have really hot payoff. So you want to be able to keep complexity down, so it pays off to really think well about your algorithm, how you structure it ahead of time.

And finally, the last two stages I've lumped together. It's really orchestration and mapping. I have my task, they need to communicate, so what kind of computation primitives do I need? What kind of communication primitives do I need? So am I packaging things up into threads? And they're talking together over DMAs or shared memories. And what you want to do is try to preserve locality and then figure out how to come up with a scheduling order that preserves overall dependence of the computation.

Parallel program by patterns is meant to essentially give you a cookbook or set of recipes you can follow to help you with different steps: decompose, assign, orchestrate and map. This could lead to really high quality solutions in some domains. So in the scientific computations there's a lot of problems that are well understood and well studied and some of the frequently occurring things have been abstracted out and sort of recorded in patterns. And there's another purpose to patterns too, in that they provide you with the vocabulary for-- two programmers can talk to each other and use the right terminology and that conveys a whole lot of information without having to actually go through and understand all the details. You instantaneously know if I use a particular model. It can also help with software reusability, malleability, and modularity. All of those things that are software engineer perspective from an engineering perspective are important.

So brief history and I found this in some of the talks that I was researching. There's a book by Christopher Alexander from Berkeley in 1977 that actually looked at classifying patterns or really listing patterns from an architectural perspective. He tried to look at what are some patterns that occur in living designs and recording

those. So as an example, for example, there's a 6 foot balcony pattern. So if you're going to build a balcony you should build it 6 foot deep and you should have it slightly recessed and so on because this is what's commonly used and these are the kinds of balconies that have good properties architecturally. Now I don't know whether this book actually had a whole lot of impact on how people designed architectures. Certainly not probably for the Stata Center, but some patterns from object oriented programming, I think, many of you have already seen these by the Gang of Four in 1995-- really sort of organized and classified and came up with different ways of-- or captured different ways of programming that people had been using. You know, things like the visitor pattern, for example, some of you might know.

So in 2005, not too long ago there was a new book, which I'm using to create some of these slides. Really came up with or recorded patterns for parallel programming. And they identified really 4 design spaces. I think these are sort of structured to express or capture different elements. So some elements are for the algorithm expression, I've listed those here and some are for the actual software construction or the actual implementation. So under algorithm expression it's really the thing of decomposition; finding concurrency. Where are my tasks? In the algorithm structure, well, you might need some way of packaging those tasks together so that they can talk to each other and be able to use parallel architecture. On the software construction side you're dealing slightly more low level details. So what are some things you might need at a slightly lower level of implementation to actually get all the computation that's expressed at the algorithm level to work and run well? So I'm going to essentially talk about the latter part in the next lecture and I'll cover much of the algorithm expression stuff here-- at least the fine concurrency part in this talk. And if there's time I'll do algorithm structure. Otherwise, just talk about it next time.

So let's say you're working with MPEG decoding. This is a pipeline picture of an MPEG 2 decoder or rather a blocked level diagram of an MPEG 2 decoder. And you have this algorithm and you say, OK, I want to parallelize this. Where's my parallelism? Where's my concurrency? You know, in MPEG 2 you have some bit screen, you do some decoding on it and you end up with two things. You end up with motion vectors that tell you here's somebody's head, in the next scene it's moved to this particular location. So these are captured by the motion vectors. So this captures or recovers temporal information. In here you cover spatial information. So in somebody's head you might have discovered some redundancies so that redundancy is eliminated out so you need to know essentially, uncompress or undo that compression. So you go through some stages. You combine the two together. Combine the motion estimation and now the recovered pictures to reconstruct the image and then you might do some additional stages. This particular stage here is indicated to be data parallel in that I can do different scenes for example in parallel or I might be able to do different slices of the picture in parallel. So I can essentially take advantage of data parallelism in the concept of taking a loop and breaking it up as I showed in lecture 5.

So in task decomposition what we're looking for is really independent coarse-grain computation. And these often are inherent to the algorithm. so here I've outlined these in yellow here. You know, so this is one particular task. I can have one thread of execution doing all the spatial decomposition. I can have another thread decoding all my motion vectors. And in general, you're looking for sequences of statements that operate together as a group. These could be loops or they could be functions. And usually you want these to essentially just fall out of your algorithm as it's expressed. And a lot of cases it does, so depending on how you think about the program you might be able to find these quicker or easier.

Data decompositions, which I've highlighted here essentially says you have the same computation applied to lots of small data element. You know, you can take your large data element, partition it into smaller chunks and do the computation over and over in parallel and so that allows you to essentially get that kind of data parallelism, expansion of space.

And finally, I'm going to make a case for pipeline parallelism, which essentially says, well, I can recognize that I have a lot of stages in my computation and it does help to actually have this kind of decomposition just because you're familiar with pipelining concepts from other domains. So this type of producer consumer chain is actually beneficial. So it does help to sort of expose these kinds of relationships. So what are some guidelines for actually coming up with your task decomposition? Where do you start? You have your algorithm, you understand the problem really well, you're writing some code and the hope is that in fact, as I've pointed out, it does happen that you can look for natural code regions that incapsulate your computation. So function calls, distinct loop iterations are pretty good places to start looking. And it's easier as a general rule, it's easier to start with as many tasks as possible and then fuse them to make the more coarse-grained than to go the other way around. It impacts your software engineering decisions, it impacts software implementation, it impacts how you incapsulate things at low level details of implementation. So it's always easier to fuse than to fizz.

And you want to consider three things. You want to keep three things in mind: flexibility, efficiency, and simplicity. So flexibility says if you made some decisions, is that really going to scale well or is it going to allow you to sort of make the decisions, changes? So you might want to have fixed tasks versus parameterized tasks, for example. So the loops that I showed in the previous talk, each loop that I parallelized had a hard coded number that said, you're going to do 4 iterations. That may work well or it may not work well. You know, I can't reuse that code now if I want to essentially use that kind of data decomposition and work sharing if I have a longer loop and I want a longer array and I want each thread to do more work. So you might want to parameterize more things in your tasks.

The efficiency, in that you have to keep in mind that each of these tasks will eventually sort of have to talk with other tasks. There's communication costs that have to be taken into account, synchronization. So you want these

tasks to actually amortize the communication costs or other overheads over to computation. And you want to keep in mind that there's going to be dependencies between these tasks and you don't want these dependencies to get out of hand. So you want to keep things under control. And lastly, which is probably as important as the other two: simplicity. And that if you start decomposing your code into different chunks and you can't then understand your code in the end, it doesn't help you from debugging perspective, doesn't help you from a software engineering perspective or not being able to reuse you code or other people being able to understand your code.

Guidelines for data decomposition are sort of similar. And you essentially have to do task and data parallelism to sort of complete your process. And often your task decomposition dictates your data partitioning. So if I've split up a loop into two different processes I've essentially implied how data should be distributed between these two threads. And data composition is a good starting point as opposed to task parallelism as a good starting point. If you're doing the same computation over and over and over again, over really, really large data sets, so you can essentially use that as your stick to decide whether you do task decomposition first or data decomposition first.

I've just listed two common data decompositions. I'll talk about more of these later on when we talk about actual performance optimizations. So you want to decompose arrays for example, along rows or columns. You can compose arrays into blocks, you decompose them into blocks. You have recursive data structures. So a tree, you might partition it into left and right sub-trees in a binary tree. And the thing you're trying to get to is actually start with a problem, and then recursively subdivide it until you can get to a manageable part. Do the computation and figure out a way to do the integration. You know, it's like merge sort, classic example -- tries to capture this really well. So again, the three theme, key concepts to keep in mind when you're doing data decomposition: flexibility, efficiency, and simplicity. The first two are really just meant to suggest that the size of the data that you've allocated actually leads to enough work. Because you want to amortize the cost of communication or synchronization, but you also want the amount of work that's generated by each data chunk to generate about the same amount of work, so load balancing. And simplicity, just for same reason as task decomposition can get out of hand, data decomposition can get out of hand. You don't want data moving around throughout and then it becomes again, hard to debug or manage or make changes and track dependencies.

Pipeline parallelism, this is actually classified somewhere else in the book. Actually, lifted it up and tried to make a case for it in that it's just good nature I think, to expose producer consumer relationships in your code. So if I have a function that's producing data that's going to be used by another function as with the spatial decomposition or in different stages of classic rate tracing algorithms you want to maintain that producer consumer relationship or that asembly line analogy. And what are some prime examples of pipelines in computer architecture? It's like the instruction pipeline and your super scalar. But there might be some other examples of pipelines, things that you might have used in say, UNIX shell. So cat processor, pipe it to another-- to a grep word and then word count that.

So I think it's a natural concept. We use it in many different ways and it's good to sort of practice that at the software level as well. And there are some computations in specific domains, like in signal processing and graphics that have really sort of-- where the pipeline model is really important part of how computation gets carried out. You know, you have your graphics pipeline for example, in signal processing. How much time do I have? How am I doing on time?

PROFESSOR: OK, should I stop here?

AUDIENCE: About how much more?

PROFESSOR: 10 slides.

PROFESSOR: OK, so this is sort of a brief summary, which will lead into a much larger talk at the next lecture on how you actually go about re-engineering your code for parallelism. And this could come into play if you start with sequential code and you're parallelizing it. Some of you are doing that for your projects. Or if you're actually writing code from scratch and you want to engineer that for parallelism as well. So I think it's important to sort of understand the problem that you're working with. So you want to survey your landscape, understand what other people might have done, and look for well-known solutions and common pitfalls. And the patterns that I'm going to talk about in more detail really provide you with a list of questions to sort of help you assess the existing code that you're working with or the problem that you're trying to solve. There's something that you need to keep in mind that sort of effect your overall correctness.

So for example, is your computation numerically stable? You might know if you have a floating point computation you might not be able to reorder all the operations because that might effect your actual precision. And so your overall output might be different and that may or may not be acceptable. So a lot of scientific codes for example, are things that have to deal with a lot of precision, might have to be cognizant of that fact. You want to also define the scope of, what are you trying to do and will it be good enough? You want to do back of the hand, back of the envelope calculations to make sure that things that you're suggesting of doing are actually feasible, that they're actually practical and that will give you the sort of performance expectations that you've set out. You also want to understand your input range. You might be able to specialize if there are some cases for example, that you're allowed to ignore. So these are good things to keep in mind.

You also want to define a testing protocol. I think it's important to understand-- you started out with some piece of code, you're going to make some changes to it, how you going to go about testing it? How you might go about debugging it and that could be essentially where you spend a lot of your time. And then having these things in mind, I think, the parts that are worth looking at are the parts that make the most sense. Where is your

computation spending most of its time? Is there hot spots in your code? And you can use profiling tools for that and in fact, you'll see some of that for cell in some of the recitations later in the course. So a simple example of molecular dynamics simulator.

What you're trying to do is, you have some space of molecules, which I'm just going to represent in 2D. You know, look, they have water molecules and I have some protein that I'm trying to understand how the different atoms in that molecule are moving around so that I can determine the shape of the protein. So there are forces, there are bonded forces between the molecules. So I've just shown for example, bonded forces among my protein and then there are non-bonded forces. So how are different atoms sort of interacting with each other because of electrostatic forces, for example. So what you try to do is figure out, on each atom, what are all the forces that are affecting it and what is its current position and then you try to estimate where it's going to move based on Newtonian, in the simplest case, a Newtonian f equals m a type projection. So in a naive algorithm you have n squared interactions. You have to calculate all the forces on one molecule from all others. By understanding your problem you know that you can actually exploit the properties of forces that essentially decrease exponentially, so you can use a cutoff distance. So if a molecule is way too far away you can ignore this. And for people who do galaxy calculations, you know you can ignore geometric forces between constellations or clusters that are too far apart. So in the sequential code, some pseudo code for doing a molecular dynamic simulator, you have your atoms array, your force array, your set of neighbors in a two-dimensional space and you're going to go through and sort of simulate different time steps. And for each time step you're going to do-- for each atom-- compute the bonded forces, compute who are my neighbors for those neighbors, compute-- so these are things that essentially incapsulate distance. Compute the forces between them, update the position and end. So since this is a loop then that might suggest essentially where to start looking for concurrency.

So you can start with the decomposition patterns and they'll be more in depth details about those next. I'm going to give you some intuition and then you would try to figure out whether your decomposition has to abide by certain dependencies and what are those dependencies? How do you expose them? And then, how can you design and evaluate? How can you evaluate your design? Screwed up again. I just fixed this.

OK, so this is the pseudo code again from the previous slide. And since all you have is a simple loop that essentially says, this is where to look for the computation. And since you're essentially doing the same computation for each atom then that again, gives you the type of parallelism that we've talked about before. So you can look for splitting up each iteration and parallelizing those so that each processor for example, does one atom. Or each processor does a collection of atoms. But there are additional tasks. So data level parallelism versus sort of control parallelism. For each atom you also want to calculate the forces. You want to calculate long range interactions, find neighbors, update position and so on. And some of these have shared data, some of them

do not. So you have to factor that in. So understanding there control dependencies essentially tells you how you need to it lay out your orchestration. So you have your bonded forces, you have your neighbor list, that effects your long range calculations. But to do this update position I need both of these tasks to have completed. And in each one of these tasks there's different data structures that they need.

So everybody essentially reads the location of the items. So this is good because we want time step that essentially says, I can really distribute this really well, but then there's a right synchronization problem because eventually I have to update this array so I have to be careful about who goes first. There's an accumulation, which means I can potentially do a reduction on these. There's some write on the other end, but that seems to be a localized data structure. So for partitioning example, neighbors might have to be just locals at a different processor. So coming up with this structure and sort of block level diagram helps you essentially figure out where are your tasks? Helps you figure out what kind of synchronization mechanisms you need and it can also help you suggest the data distribution that you might need to reduce synchronization costs and problems. And lastly, you want to essentially evaluate your design. And you want to keep in mind, what is your target architecture. Are you trying to really run on shared memory and distributed memory and message passing or are you just doing this for one architecture?

So for your project, you're doing this for self, so you can be very self specific. But if you're doing this in other contexts, the architecture actually might influence some of your decisions. Does data sharing have enough special properties like a read only? There are data structures that are read only. Are there enough accumulations that you can exploit by reductions? Are there temporal constraints on data sharing that you can exploit? And can you deal with those efficiently? If you can't then you have a problem. So you need to resolve that. If the designs OK then you move on to the next design space. So at the next lecture I'll go through these in a lot more detail. That's it.