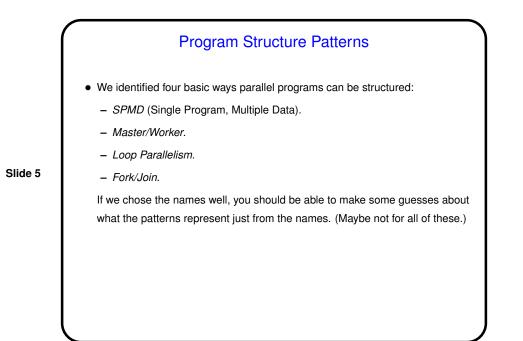
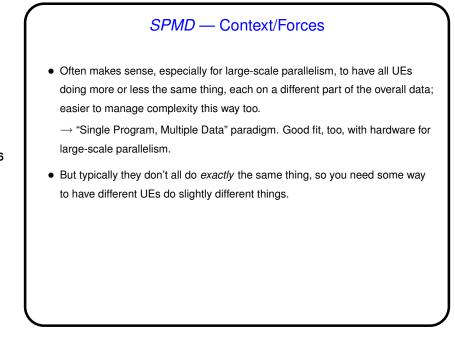
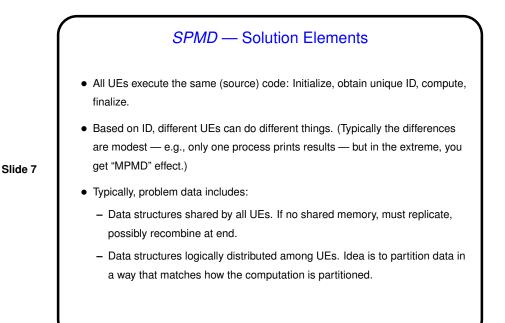


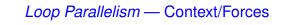
	Forces
	Part of the "design pattern" idea is that a good pattern represents a good trade-off between "forces" pulling in different directions.
•	For the patterns in this chapter, common set of forces:
	 Clarity of abstraction — is structure clear from code?
	 Scalability — does program "scale" well to large numbers of PEs (processing elements)?
	 Efficiency — does it make good use of resources?
	 Maintainability — can humans understand it?
	 Environmental affinity — does it work well in the likely target environment(s)?
	- Sequential equivalence - same results no matter how many processes?







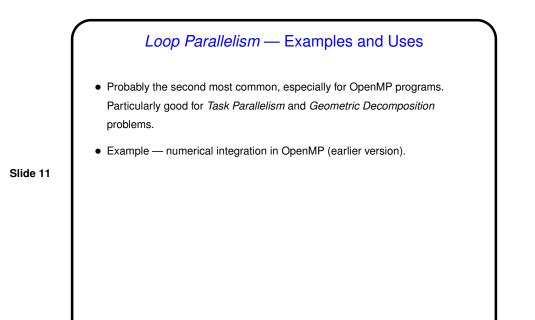
SPMD — Examples and Uses • Very, very common, especially for MPI programs. Particularly good for Task Parallelism and Geometric Decomposition problems. • Example — numerical integration: - Logical choice for MPI. One choice we could make, though, is how to partition data (loop iterations) among UEs - by blocks, or cyclically? Only one "shared" variable - sum being computed. Notice that in effect we replicate the variable, and recombine at the end. - Can do something in similar in OpenMP (another version of example).



- Programs in traditional application areas for parallel programming science and engineering — mostly loop-based. Optimizing loops has a long history first vectorizing, then parallelizing.
- Particularly appealing approach when a sequential program already exists, and you want to convert ("parallelize") it. Sometimes conversion can be done one loop at a time — easier to develop/test/debug.

Slide 10

Loop Parallelism — Solution Elements Find computationally intensive loops. (No point, for example, in spending a lot of time parallelizing initialization code.) Eliminate loop-carried dependencies (e.g., replicating variables so each UE has a copy). Parallelize loops — arrange for iterations to be distributed among UEs. Optimize loop "schedule" (how iterations are mapped to UEs).



Master/Worker — Context/Forces

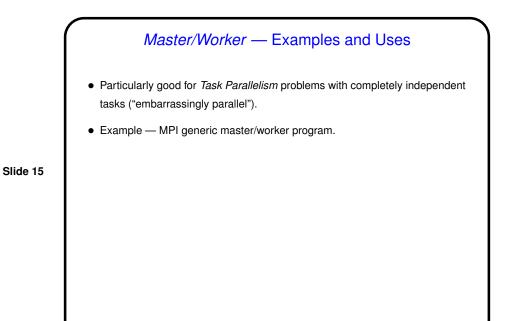
- For applications where it's easy to tell how to split up the computational load to get "good load balance", previous two patterns usually work well.
- But for some applications, it's not so obvious how to do this maybe not really possible, if work per task varies a lot and is not predictable, or if target platform includes PEs with different capabilities.



- Basic idea one or more workers that execute tasks, master that manages things.
- "Bag of tasks" represents tasks yet to be done. Typically created by master process; often implemented as shared queue. Workers can pull elements from it directly, or can communicate with master to get new tasks.
- Typical approach shown in Fig. 5.14.

Master/Worker --- Solution Elements, Continued

- Several potential complications:
 - All tasks may be known initially, or new ones may be generated during computation.
 - Usually computation isn't done until all tasks are done, but sometimes can stop early.
- Several variations/optimizations:
 - Master can turn into a worker after creating tasks. (Obviously more efficient if it has nothing to do.)
 - Master can be implicit, if tasks are loop iterations and dynamic scheduling of loop iterations is possible.
- Implementation normally involves, to some extent, one of the other patterns in this chapter.



Fork/Join — Context/Forces

- For applications where the number of concurrent tasks is more or less constant, and relationships among them are simple and regular, previous patterns usually work well.
- Slide 16
- But for some applications, tasks are created dynamically ("forked") and later terminated ("joined" with forking task) as program runs. Sometimes you can still use one of the previous patterns, but sometimes not — if relationships among tasks are recursive (e.g., *Divide and Conquer*) or irregular, or if different tasks represent different functions (i.e., you need to do two or more different things concurrently).
- In that case, it may make more sense to create a UE for each task potentially expensive, but easier to understand.



- Simple approach one task per UE. As new tasks are created, a new UE is created for each; when the task finishes, the UE is destroyed. Typically the UE that created the new task/UE waits for it to finish. Simple to understand, but potentially inefficient.
- Slide 17
- More complicated approach pool of UEs and queue of tasks, with UEs grabbing new tasks out of the queue as they finish their old tasks. Potentially more efficient, but more complicated to program and understand.



• Particularly good for *Divide and Conquer* and *Recursive Data* problems. One-task-per-UE version is OpenMP's standard programming model (expressed implicitly). Also matches (pre-1.5) Java's support for multithreading.

(Curiously enough, though, most OpenMP programs really use the simpler *Loop Parallelism.*)

• Example — mergesort.

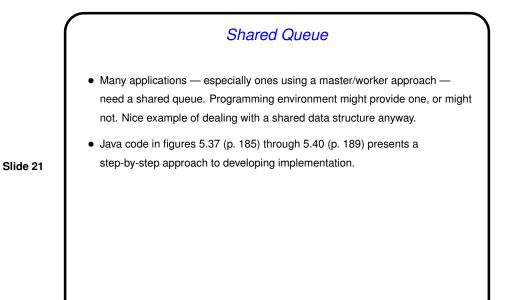


- Basic ways parallel programs can be structured:
 - SPMD (Single Program, Multiple Data) "like an MPI program" (but could use same strategy in OpenMP, e.g.).

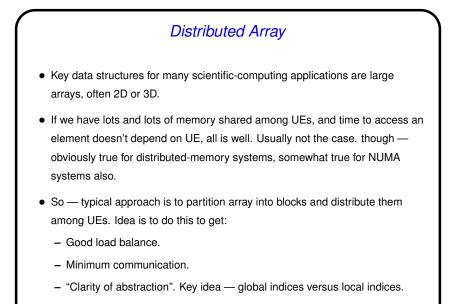
- Loop Parallelism "like an OpenMP program".
- Master/Worker as the name suggests. Look briefly at MPI example.
- Fork/Join if you need to be able to create / wait for UEs in any arbitrary way.
- How to choose one? usually based on combination of programming environment (MPI, OpenMP, etc.) and overall strategy (*Algorithm Structure* pattern).

Supporting Structures Data Structure Patterns

- Probably not a complete list, but some examples of frequently-used ways of sharing data:
 - Shared Data (generic advice for dealing with data dependencies).
 - Shared Queue (what the name suggests mostly included as example of applying Shared Data).
 - Distributed Array (what the name suggests).
- Programming environment / library may provide support (e.g., Java has library class(es) for shared queues).



Simplest approach to managing a shared data structure where concurrent modifications might cause trouble — one-at-a-time execution. Shown in figures 5.37 (nonblocking) and 5.38 (block-on-empty). Only tricky bits are use of dummy first node and details of take. Reasons to become clearer later. Usually a good idea to try simplest approach first, and only try more complex ones if better performance is needed. ("Premature optimization is the root of all evil." Attributed to D. E. Knuth; may actually be C. A. R. Hoare.)
Here, next thing to try is concurrent calls to put and take. Not too hard for nonblocking queue — figure 5.39. Tougher for block-on-empty queue — figure 5.40. In both cases, must be very careful.
If still too slow, or a bottleneck for large numbers of UE, explore distributed queue.



Pictures are easy to draw and understand; code can get messy.

Distributed Array, Continued

- Commonly used approaches ("distributions"):
 - 1D block.
 - 2D block.
 - Block-cyclic.

- For some problems (such as heat distribution problem), makes sense to extend each "local section" with "ghost boundary" containing values needed for update.
- Look at some versions of code for the heat-distribution problem. (MPI code in book as Figures 4.14 and 4.15 (pp. 90–91).)

