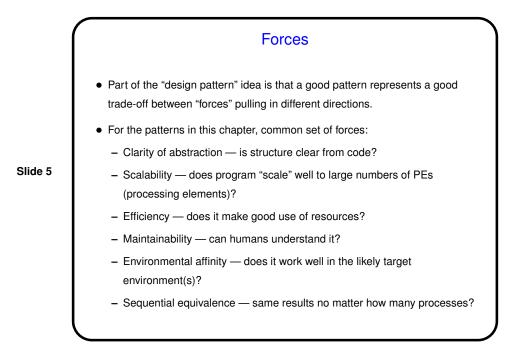
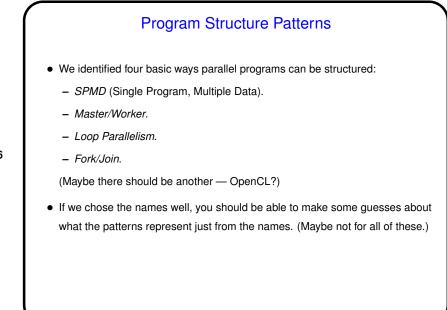


Supporting Structures Design Space
Key idea here — represent (and talk about in general terms) two classes of commonly-used things:

Program structures — e.g., SPMD (think "like MPI programs").
Frequently-used data structures — e.g., shared queue.





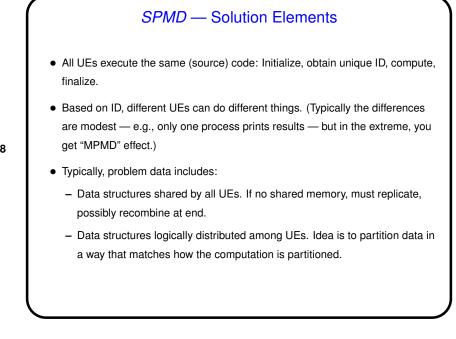


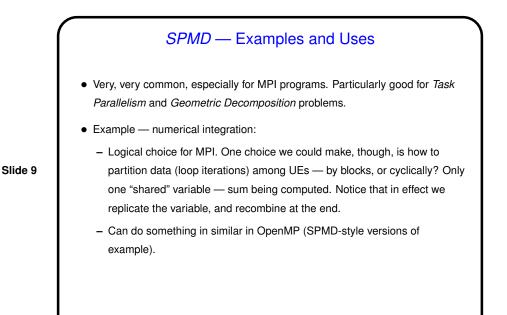
• Often makes sense, especially for large-scale parallelism, to have all UEs doing more or less the same thing, each on a different part of the overall data; easier to manage complexity this way too.

 \rightarrow "Single Program, Multiple Data" paradigm. Good fit, too, with hardware for large-scale parallelism.

• But typically they don't all do *exactly* the same thing, so you need some way to have different UEs do slightly different things.

Slide 7



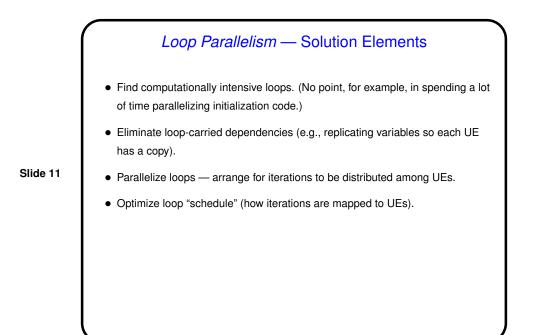


Loop Parallelism — Context/Forces

 Programs in traditional application areas for parallel programming — science and engineering — mostly loop-based. Optimizing loops has a long history first vectorizing, then parallelizing.

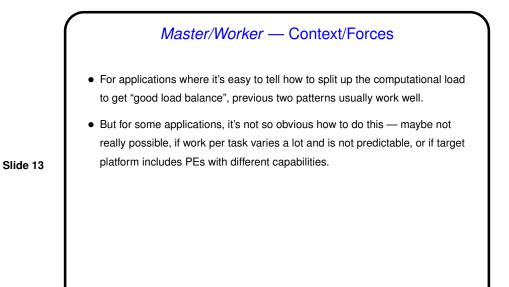
Slide 10

 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.



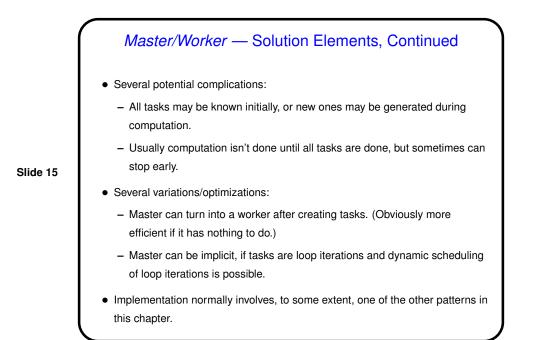
Loop Parallelism — Examples and Uses

- Probably the second most common, especially for OpenMP programs. Particularly good for *Task Parallelism* and *Geometric Decomposition* problems.
- Example numerical integration in OpenMP (earlier version).



Master/Worker — Solution Elements

- 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 — Examples and Uses

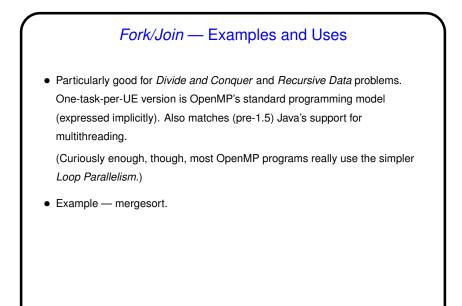
- Particularly good for *Task Parallelism* problems with completely independent tasks ("embarrassingly parallel").
- Example MPI generic master/worker program. (Next time?)



- 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.
- 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.

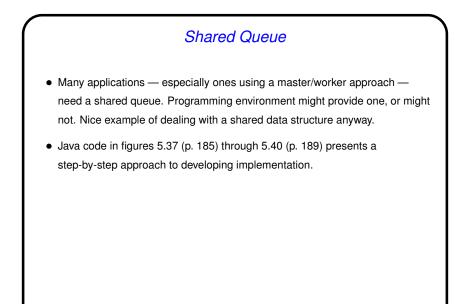
Fork/Join --- Solution Elements

- 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.
- 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.





- 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).

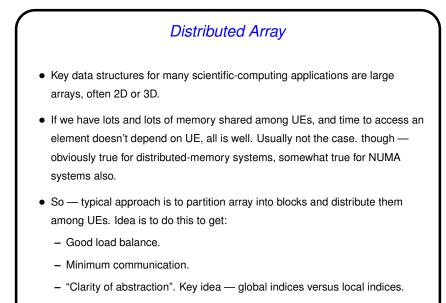


Shared Queue, Continued

• 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.

Slide 24

• For some problems (such as heat-diffusion problem), makes sense to extend each "local section" with "ghost boundary" containing values needed for update.

