

Program Structure Patterns

- We identified four basic ways parallel programs can be structured:
 - SPMD (Single Program, Multiple Data).
 - Master/Worker.
 - Loop Parallelism.
 - Fork/Join.

If we chose the names well, you should be able to make some guesses about what the patterns represent just from the names. (Maybe not for all of these.)

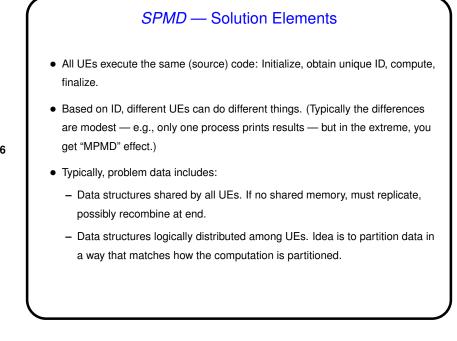


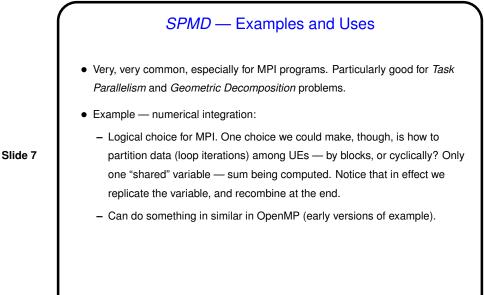
• Often makes sense, especially for large-scale parallelism, to have all UEs doing more or less the same thing, each on a different part of the overall data; easier to manage complexity this way too.

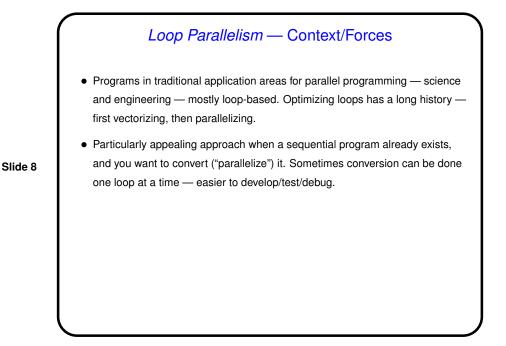
 \rightarrow "Single Program, Multiple Data" paradigm. Good fit, too, with hardware for large-scale parallelism.

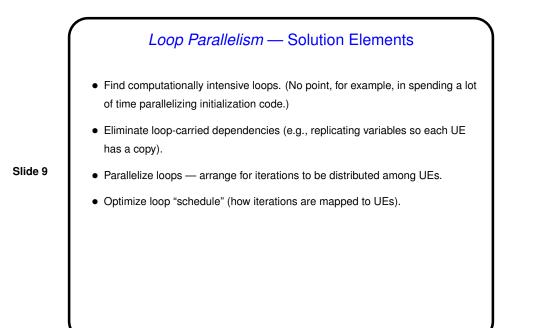
• But typically they don't all do *exactly* the same thing, so you need some way to have different UEs do slightly different things.

Slide 5



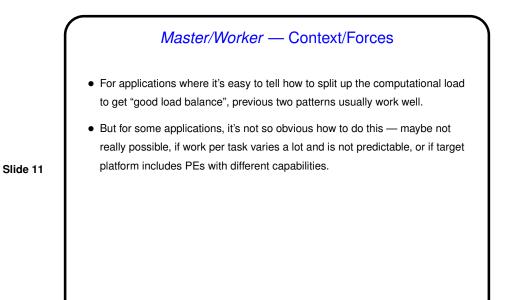






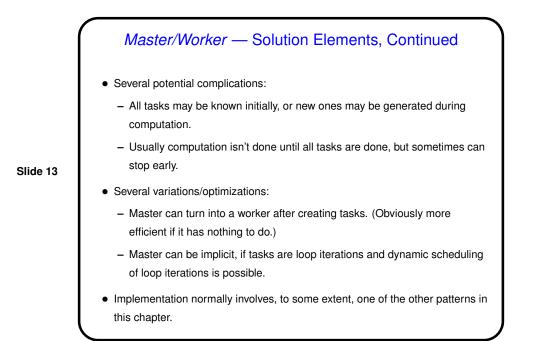
Loop Parallelism — Examples and Uses

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

Slide 15



- 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.

