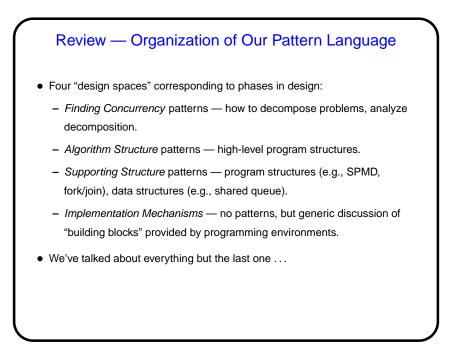
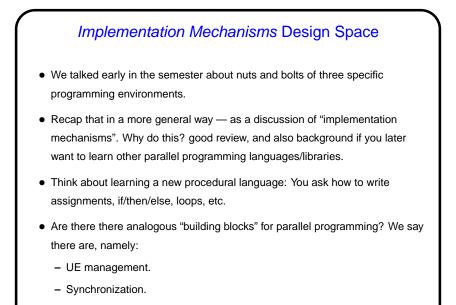


Slide 1





- Communication.

UE Management

 "UE"? In MPI we have processes. In OpenMP we have (implicit) threads. In Java we have threads. Common theme — something that carries out computations. Generally have several of these running concurrently. Our generic term — "unit of execution" (UE).

Slide 4

- In general, what you want to know is how these are created and destroyed.
- $\bullet\,$ Discuss separately for processes and threads \ldots

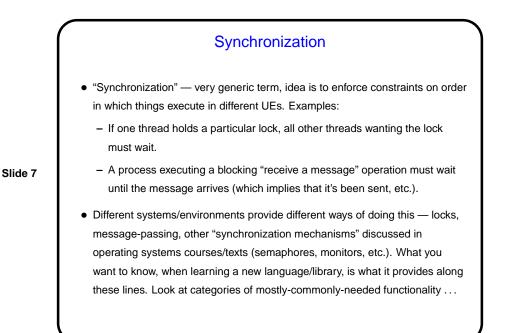
Managing Threads

- Threads typically lightweight, so creating/destroying them during computation is reasonable (though one wouldn't want to go overboard). What you want to know — how are threads created? how are they destroyed?
- In OpenMP, threads are created by parallel pragma (which applies to a "structured block"). All but master thread end and are destroyed at end of block to which pragma applies. (Actually, implementation may reuse them for subsequent parallel block. But it's as if they're created new each time.)
- In Java, threads are created by creating instances of Thread class, or subclass. Must also invoke start. A thread terminates when its run method ends; it's destroyed by the garbage collector in the usual way. Java 1.5 provides interfaces/classes that hide some of these details.

Managing Processes

- Processes are "heavier" than threads, so creating and destroying them during computation isn't done much. Again, though, what you want to know is how they're created, how they're destroyed.
- In PVM (and in newest version of MPI), could explicitly "spawn" a process.
- In MPI 1.1, creating processes is external to the API. Why? Historical reasons, basically. Processes end when the code they run terminates.
 Possible for them to hang around ("orphan processes") if code doesn't end cleanly.
- In Java, there's some support for creating processes, but it's mostly for interfacing with underlying system. Support for distributed-memory computing is via sockets (low-level version of message-passing, in a way) and RMI.

Slide 5



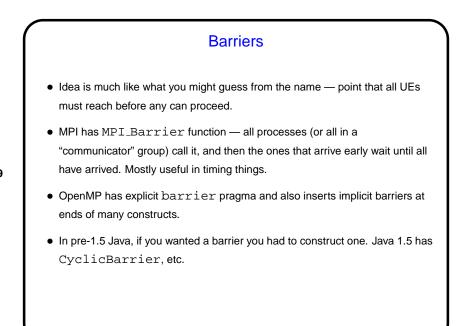
Memory Synchronization and Fences

Additional complication in shared-memory systems:

In the simple classical model of how things work, reads/writes to memory are "atomic" (execute without interference from other UEs).

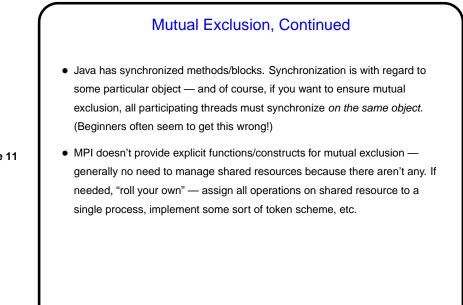
Reality these days is somewhat different — hardware may cache values, compiler may do interesting optimizations, etc., etc.

- How to know when there's a consistent view of memory all UEs share?
 "Memory fence" idea writes before the fence visible to reads after it, etc.
- Memory fences usually implicit in higher-level constructs, but you could need to know about them if threads share variables that change during execution, and access to the variable isn't controlled by some sort of synchronization (OpenMP critical section, Java synchronized block, etc.).
- More details in chapter 6, with examples ...

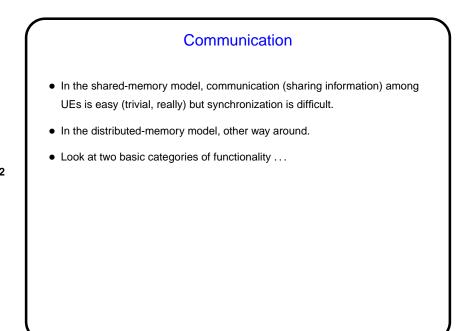


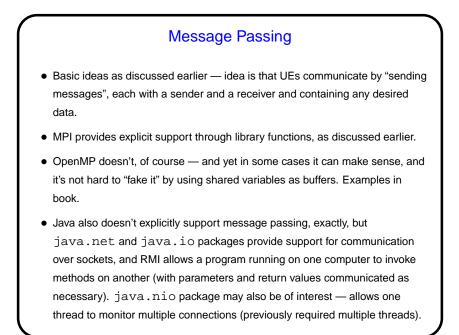
Slide 9

Mutual Exclusion
Idea is again what you might guess from the name, and as we've discussed – only one UE at a time can have access to some "critical section" of code (to prevent "race conditions"). Shared Data talks more about when this is needed.
OpenMP has critical section pragma. If more flexibility needed, locks also available. (Idea is that before entering a critical section you obtain the relevant lock, and then release it on exit.)



Slide 11





Slide 14 Slide 14 A san example of "roll your own" — discussion of various ways to implement reduction.

