Slide 1

Administrivia

- (How many will be here Thursday?)
- Homework 2 officially due today at midnight. If that's not possible for you — as long as it's in by class time Thursday, okay. Don't worry if performance is not great. This is a first pass — goal is to get something that produces more or less the same results as a sequential program, experiment a bit with different seeds for RNG.

Slide 2

More Administrivia

- About Exam 1: Would anyone object if I proposed just dropping it? both exams? Grade then would be determined by homeworks (of which there would be slightly more), the to-be-discussed project, and attendance/participation.

  (This is a big enough change that I won't do it if at least one student strongly objects. If you do but don't want to speak up now, send me e-mail. If I get no such e-mail by midnight tonight, no exams.)

Slide 3

Review — Organization of Our Pattern Language

- Four "design spaces" corresponding to phases in design:
  - *Finding Concurrency* patterns — how to decompose problems, analyze decomposition.
  - *Algorithm Structure* patterns — high-level program structures.
  - *Supporting Structure* patterns — program structures (e.g., SPMD, fork/join), data structures (e.g., shared queue).
  - *Implementation Mechanisms* — no patterns, but generic discussion of "building blocks" provided by programming environments.
- Idea is that you start at the top, work your way down, possibly with some backtracking. We'll discuss starting at the bottom . . .

Slide 4

Implementation Mechanisms Design Space

- So far we've talked about some "big picture of parallel computing" stuff, plus nuts and bolts of three specific programming environments — the low-level stuff you need in order to be able to write programs.
- Now we're going to recap that in a more general way — as a discussion of "implementation mechanisms". Why do this? Could express in design pattern terms:

  Problem: Where do you start in learning how to use a new parallel programming "environment" (language, language extension, or library)?

  Context: In learning a new sequential programming language, ask about writing assignment, if/then/else, loops, etc. Are there analogous "building blocks" for parallel programming?

  Solution: General discussion of (1) UE management, (2) synchronization, and (3) communication.

Slide 5

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

Slide 6

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 also provides interfaces/classes that hide some of these details — allow you to define a "pool" of threads, specify tasks to be assigned to threads in the pool, etc. To learn more: Start with documentation of `Executor`, `Executors`.

Slide 7

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, I think), 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 8

Synchronization

- "Synchronization" — very generic term, idea is to enforce constraints on order in which things execute in different UEs. Examples:
  - If one thread holds a particular lock, all other threads wanting the lock must wait.
  - A process executing a blocking "receive a message" operation must wait until the message arrives (which implies that it's been sent, etc.).
- Different systems/environments provide different ways of doing this — locks, message-passing, other "synchronization mechanisms" discussed in operating systems courses/texts (semaphores, monitors, etc.). What you want to know, when learning a new language/library, is what it provides along these lines. Look at categories of mostly-commonly-needed functionality . . .

Slide 9

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 10

Barriers

- Idea is much like what you might guess from the name — point that all UEs must reach before any can proceed.
- MPI has `MPI_Barrier` function — all processes (or all in a "communicator" group) call it, and then the ones that arrive early wait until all have arrived. Mostly useful in timing things.
- OpenMP has explicit `barrier` pragma and also inserts implicit barriers at ends of many constructs.
- In pre-1.5 Java, if you wanted a barrier you had to construct one. Java 1.5 has `CyclicBarrier`. etc.

Slide 11

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"). More detail later (*Shared Data* pattern) about when this is needed.
- OpenMP has `critical section` pragma. If more flexibility needed, locks also available.
- Java has synchronized methods/blocks. Synchronization is with regard to some particular object — and of course, if you want to ensure mutual exclusion, all participating threads must synchronize *on the same object*.
- MPI doesn't provide explicit functions/constructs for mutual exclusion — generally no need to manage shared resources because there aren't any. If needed, "roll your own" — assign all operations on shared resource to a single process, implement some sort of token scheme, etc.

Slide 12

Communication

- In the shared-memory model, communication (sharing information) among UEs is easy (trivial, really) but synchronization is difficult.
- In the distributed-memory model, other way around.
- Look at two basic categories of functionality . . .

Slide 13

Message Passing

- Basic ideas as discussed earlier — idea is that UEs communicate by "sending messages", each with a sender and a receiver and containing any desired data.
- MPI provides explicit support through library functions, as discussed earlier.
- OpenMP doesn't, of course — and yet in some cases it can make sense, and it's not hard to "fake it" by using shared variables as buffers. Nice example in book.
- Java also doesn't explicitly support message passing, exactly, but `java.net` and `java.io` packages provide support for communication over sockets, and RMI allows a program running on one computer to invoke methods on another (with parameters and return values communicated as necessary). `java.nio` package may also be of interest — allows one thread to monitor multiple connections (previously required multiple threads).

Slide 14

Collective Communication

- Basic idea as discussed earlier — communication events that involve more than two UEs. Frequently all UEs involved. Common examples: broadcast, barrier, reduction. (Review — reduction means repeatedly applying a binary operator to "reduce" a set of data items to a single data item. Examples include sum, product, max, min.)
- MPI provides explicit support through library functions, as discussed earlier.
- OpenMP also provides explicit support for some collective operations, also as discussed earlier — barriers, reduction via `reduction` clause.
- Java doesn't (unless in classes added in 1.5?), but these operations can all be coded in terms of point-to-point message passing.
- As an example of "roll your own" — discussion of various ways to implement reduction.

Slide 15

Other Communication Constructs

- "One-sided" communication — two UEs communicate, but only one of them explicitly does anything (e.g., one UE puts something into a buffer on another node).
- Various schemes for "virtual shared memory" — "tuple space" in Linda, e.g.

Slide 16

Minute Essay

- Tell me about your experiences doing Homework 2. What was difficult? What was easy? What was interesting? Which of the three parallel programming environments (MPI, OpenMP, Java) do you feel most comfortable with at this point?