Slide 1

Administrivia

- Homework 5 on the Web. Due last day of class (12/02).

Slide 2

Example Application: Mergesort

- Mergesort should be familiar from other courses. Sequential algorithm is divide-and-conquer, and solution of subproblems is independent, so:
- We could pretty much skip the whole *Finding Concurrency* step and go directly to . . .
- *Algorithm Structure* pattern *Divide and Conquer* seems to fit.

Mergesort, Continued

Slide 3

- One important consideration is whether to every call to the recursive function should be a task. Probably not — way too much overhead.
- For this problem, at each level both subproblems are roughly the same size, so what probably makes sense is to have at most one task per UE. (If the subproblems were of different sizes, we'd want to consider having more tasks and mapping them to UEs in a way that would produce good load balance.)
- (Look at code, timing data.)

Review — Organization of Our Pattern Language

Slide 4

- *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., distributed array).
- *Implementation Mechanisms* — no patterns, but generic discussion of "building blocks" provided by programming environments.

Slide 5

Implementation Mechanisms Design Space

- We talked early in the semester about nuts and bolts of four specific programming environments.
- Recap that in a more general way, as a discussion of "implementation mechanisms". Why do this? good review, and also background if (when?) you later want to learn other parallel programming languages/libraries.
- Think about learning a new procedural language: You ask how to write assignments, if/then/else, loops, etc.
- Are there there analogous "building blocks" for parallel programming? we say there are . . .

Slide 6

Parallel Programming Basics

- UE management.
- Synchronization.
- Communication.
- (These may need to be reworked to apply well to OpenCL. Open question for now.)

Slide 7

UE Management

- "UE"? In MPI we have processes. In OpenMP we have (implicit) threads. In Java we have threads. In OpenCL we have — not clear, but I say something related to work groups. 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 . . . (Here and in the remaining discussion, I'll omit OpenCL because it's not clear how it fits into the overall scheme.)

Slide 8

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 is how threads are created, destroyed.
- In OpenMP, threads 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 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.util.concurrent` provides interfaces/classes that hide some of these details.

Managing Processes

Slide 9

- 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 MPI 2.x and later, can 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.

Synchronization

Slide 10

- “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 11

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 12

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. (Something to check: `single` does, while `master` does not.)

- In early releases of Java, if you wanted a barrier you had to construct one. Java 1.5 added `java.util.concurrent`, whichh includes `CyclicBarrier`, etc.

Slide 13

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 14

Mutual Exclusion, Continued

- 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*. (Beginners often seem to get this wrong!)
- 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 15

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 16

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. Examples 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 17

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 (as far as I know), 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 18

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” — e.g., “tuple space” in Linda.

Slide 19

Minute Essay

- We only have two real classes left. Thoughts on how we should use them? Things that would be easily doable for me: more multithreading in Java, more MPI functions, quick tour of some other programming environments (e.g., C++ threads, or POSIX threads).
- Are you planning to attend class next Monday?