Slide 1

Administrivia

- A reminder: Please do not reboot the machines in this room (HAS 340). People depend on their being available for remote access.

Slide 2

Recap — Current Hardware for Parallel Programming

- One category — multiple CPUs sharing access to a common memory.
- Another category — multiple CPUs, each with separate memory, communicating over interconnection network.

Slide 3

Recap — Programming Models

- Shared-memory model — concurrently-executing threads sharing address space. Various ways to communicate / synchronize.
- Distributed-memory model — concurrently-executing processes, each with separate address space, communicating by sending / receiving messages.

Slide 4

What Programming Languages Support This?, Continued

- A regular sequential language with a parallelizing compiler: Attractive, but such compilers are not easy.
- A language designed to support parallel programming (Java, Ada, PCN): Perhaps the most expressive, but more work for programmers and implementers.
- A regular sequential language plus calls to parallel library functions (PVM, MPI, Pthreads): More familiar for users, easier to implement.
- A regular sequential language with some added features (CC++, OpenMP): Also familiar for users, can be difficult to implement.

Slide 5

Parallel Programming Environments

- By "programming environments" we mean languages / libraries / extensions. There are many! (Table 2.1 in book has a list — and we might have missed a few.)
- For our book we chose one of each:
  - MPI (library) because it's something of a standard for message-passing programming.
  - OpenMP (language extension) because it's emerging as a standard for shared-memory programming.
  - Java because it's widely available and might be many people's first exposure to parallel programming.
- Other popular programming environments — POSIX threads (Pthreads), Win32 API, PVM, ...

Slide 6

Sketch of Parallel Algorithm Development

- Start with understanding of problem to be solved / application.
- Decompose computation into "tasks" — snippets of sequential code that you might be able to execute concurrently.
- Analyze tasks and data — how do tasks depend on each other? what data do they access (local to task and shared)?
  (Or start with decomposition of data and infer tasks from that.)
- Plan how to map tasks onto "units of execution" (threads/processes) and coordinate their execution. Also plan how to map these onto "processing elements".
- Translate this design into code.
- Our book organizes all of this into four "design spaces", corresponding to (we think) steps in program design / development.

Slide 7

A Few Words About Performance

- If the point is to "make the program run faster" — can we quantify that?
- Sure. Several ways to do that. One is "speedup" —

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)}$$

- What's the best possible value you can imagine for $S(P)$?

Slide 8

Performance, Continued

- Best possible value for $S(P)$? would seem to be P, right?
- Can you think of circumstances in which you could do better ("superlinear speedup")?

Slide 9

Performance, Continued

- "Superlinear speedup" could happen if dividing up the computation among processors means more of the program's code/data can fit into memory, or cache. Could also happen in searches, if you can stop after finding one solution.
- What's the worst value you can imagine for $S(P)$?

Slide 10

Performance, Continued

- Worst possible value would seem to be 1, right?
- Can you think of circumstances in which you'd do worse? (Hint: What do you know so far about how the parts of the program running on different cores/processors/machines interact?)

Slide 11

Parallel Overhead

- Many reasons why a "real" parallel program might be slower than hoped for — even, possibly, slower than the sequential program!
- For shared-memory programming — if we need to synchronize use of shared variables, that takes time.
- For message-passing programming — sending messages takes time. Typically time to send a message involves a fixed cost plus a per-byte cost.

  (Sometimes can speed things up by "overlapping computation and communication".)
- Also, "poor load balance" may slow things down.
- (And we're not even mentioning what happens if you don't have exclusive access to all processors.)

Slide 12

Performance, Continued

- Even without overhead, though, why wouldn't we always get "perfect" speedup (P)?

Slide 13

Amdahl's Law

- And most "real programs" have some parts that have to be done sequentially. Gene Amdahl (principal architect of early IBM mainframe(s)) argued that this limits speedup — "Amdahl's Law":

  If γ is the "serial fraction", speedup on P processors is (at best — this ignores overhead)

  $$S(P) = \frac{1}{\gamma + \frac{1-\gamma}{P}}$$

  and as P increase, this approaches $\frac{1}{\gamma}$ — upper bound on speedup.

  (Details of math in chapter 2.)

Slide 14

What's Next — Nuts and Bolts

- So we can start writing programs as soon as possible, next topic will be a fast tour through the three programming environments we will use for writing programs.

Slide 15

Minute Essay

- None — sign in.