Slide 1

Administrivia

- Reminder: Homework 4 due today.
- Homework 5 on the Web. Due a week from Thursday. Two programming problems. Sample programs page has some code for simple I/O procedures that I hope will reduce some of the tedium. If you can't figure out how to use it, or if you have other problems making programs work, please ask for help! it may have to be by e-mail.

Slide 2

More Administrivia

- Midterms almost ready to return, at last! All problems graded, and I just need to add up points and record scores. I should be sending out grades tomorrow by e-mail and returning actual papers, and a sample solution, Thursday.
- Reminder: Final next week, Wednesday or Thursday, depending on section. (Times on "lecture topics and assignments" page.) Okay to come to the other section's time if you prefer. I will post a review sheet, probably by Thursday, but briefly: Format similar to midterm. Comprehensive but with slight emphasis on later material. Solutions to all homeworks to be available.

Slide 3

Parallel Computing — Overview

- Support for "things happening at the same time" goes back to early mainframe days, in the sense of having more than one program loaded into memory and available to be worked on. If only one processor, "at the same time" actually means "interleaved in some way that's a good fake". (Why? To "hide latency".)
- Support for actual parallelism goes back almost as far, though mostly of interest to those needing maximum performance for large problems. Somewhat controversial, and for many years "wait for Moore's law to provide a faster processor" worked well enough. Now, however . . .

Slide 4

Parallel Computing Overview, Continued

- Improvements in "processing elements" (processors, cores, etc.) seem to have stalled. Instead hardware designers are coming up with ways to provide more processing elements.
- One result is that multiple applications can execute really at the same time.
- Another result is that individual applications *could* run faster by using multiple processing elements.

  Non-technical analogy: If the job is too big for one person, you hire a team. But making this effective involves some challenges (how to split up the work, how to coordinate).
- In a perfect world, maybe compilers could be made smart enough to convert programs written for a single processing element to ones that can take advantage of multiple PEs. Some progress has been made, but goal is elusive.

Slide 5

Parallel Computing — Hardware Platforms

- Clusters — multiple processor/memory systems connected by some sort of interconnection (could be ordinary network or fast special-purpose hardware). Examples go back many years.
- Multiprocessor systems — single system with multiple processors sharing access to a single memory. Examples also go back many years.
- Multicore processors — single "processor" with multiple independent PEs sharing access to a single memory. Relatively new.
- "SIMD" platforms — hardware that executes a single stream of instructions but operates on multiple pieces of data at the same time. Popular early on (vector processors, early Connection Machines) and now being revived (GPUs used for general-purpose computing).

Slide 6

Parallel Programming — Software

- Key idea is to split up application's work among multiple "units of execution" (processes or threads) and coordinate their actions as needed. Non-trivial in general, but not too difficult for some special cases ("embarrassingly parallel") that turn out to cover a lot of ground.
- Two basic models, shared-memory and distributed-memory. Shared-memory has two variants, SIMD ("single instruction, multiple data" and MIMD ("multiple instruction, multiple data"). SPMD ("single program, multiple data") can be used with either one, and often is, since it simplifies things.

Slide 7

Parallel Programming — Shared-Memory Model (MIMD)

- "Units of execution" are (typically) threads, all with access to common memory space, potentially executing different code.
- Convenient in a lot of ways, but sharing variables makes "race conditions" possible. (Now that you know more about how hardware works you may understand the issues better! A single line of HLL code may translate to multiple instructions . . .)
- Typical programming environments include ways to start threads, split up work, synchronize. OpenMP extensions (C/C++/Fortran) somewhat low-level standard.

Slide 8

Parallel Programming — Distributed-Memory Model

- "Units of execution" are processes, each with its own memory space, communicating using message passing, potentially executing different code.
- Less convenient, and performance may suffer if too much communication relative to amount of computation, but race conditions much less likely.
- Typical programming environments include ways to start processes, pass messages among them. MPI library (C/C++/Fortran) somewhat low-level standard.

Slide 9

Parallel Programming — SIMD Model

- "Units of execution" term may not make sense. Parallelism comes from all processing elements executing the same program in lockstep, but with different processing elements operating on different data elements.
- Excellent fit for some problems ("data-parallel"), not for others. Very convenient when it fits, pretty inconvenient when not.
- Typical programming environments feature ways to express data parallelism. OpenCL (C/C++) may emerge as somewhat low-level standard, especially suited for GPGPU.

Slide 10

Parallel Programming — Shared-Memory Hardware

- Figure 6.7 sketches basic idea — multiple processing elements (call them processors, cores, whatever) connected to a single memory.
- Synchronization (locking) *can* (in theory?) be done with no hardware support, but much easier if ISA includes instruction(s) for locking. MIPS does (briefly described in chapter 2).
- Access to RAM can be reasonably straightforward — only one processor at a time — but if each processing element has its own cache, things may get tricky. Typically hardware provides some way to keep them all in synch.
- "Single memory" may actually be multiple memories, with each processing element having access to all memory, but faster access to one section ("NUMA" (Non-Uniform Memory Access). Making good use of this also can affect performance — and may be non-trivial to accomplish, especially if programming environment doesn't give you appropriate tools.

Slide 11

Parallel Programming — Distributed-Memory Hardware

- Figure 6.13 sketches basic idea — multiple systems (processor(s) plus memory) communicating over a network.
- No special hardware required, though really high-end systems may provide a fast special-purpose network.

Slide 12

Parallel Programming — SIMD Hardware

- Various ways to implement this idea in hardware.
- One approach: multiple processing elements sharing access to memory and all executing the same instruction stream,

  This is more or less how GPUs work. A complication — they often have a separate memory, so data must be copied to/from RAM. Potential performance problem, may be cumbersome for programmers.
- Another approach: “vector processing units” that stream/pipeline operation on data elements to get the data-parallelism effect.

Slide 13

Other Hardware Support for Parallelism

- Instruction-level parallelism (discussed in not-assigned section(s) of chapter 4) allows executing instructions from a single instruction stream at the same time, if it's safe to do so. Requires hardware and compiler to cooperate, and (sometimes?) involves duplicating parts of hardware (functional units).
- Hardware multithreading (discussed in chapter 6) includes several strategies for speeding up execution of multiple threads by duplicating parts of processing element (as opposed to duplicating full PE, as happens with "cores").

Slide 14

Minute Essay

- None — quiz.