Slide 1

Administrivia

- Reminder: Homework 6 due Friday. Talk Friday about review for the final (May 10)?

Slide 2

A Few Words About Caching and Memory

- As you may know (or not!) access to RAM is slow compared to processor speeds.
- So many/most hardware platforms define a hierarchy of forms of memory, from fast-but-expensive to slow-but-cheap: registers, “cache”, RAM, “virtual memory” (on disk). Some managed by hardware, some by software (operating system). Idea is to use each level to hold frequently-used values from next level down.
- What makes this work is “locality” — set of variables in frequent use at one point in time is likely to be also in frequent use at nearby points in time, and also set of variables is likely to be somewhat compact (as opposed to spread over the program’s whole range of data).
- Making good use of cache turns out to sometimes have a big effect on performance! (Example.)

Slide 3

Parallel Programming — Overview/Recap

- Hardware to support "doing more than one thing at a time" has existed for a long time. Mostly used to run more than one application at a time, or by people who need maximum performance.
- Even with single processors one can fake simultaneous execution, so understanding of programming issues also goes back a long time (mostly in the context of operating systems).
- Now, however, "buy a faster (single) processor" isn't feasible as a way of getting better performance. Instead it seems that one must do parallel programming . . .

  (Which turns out to be a useful mental model for some kinds of problems as well.)

Slide 4

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.

Slide 5

Parallel Programming — Shared-Memory Model

- "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.

Slide 6

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.

Slide 7

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.

Slide 8

Parallel Programming — Shared-Memory Hardware

- Figure 7.2 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 9

Parallel Programming — Distributed-Memory Hardware

- Figure 7.4 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 10

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 (as I understand things!) is how GPUs work. One complication is that 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 11

Other Hardware Support for Parallelism

- Instruction-level parallelism (discussed in section 4.10, not assigned) 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 7) 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 12

Minute Essay

- None — quiz.