Administrivia

Slide 1

- Reminder: Quiz 6 Wednesday. More questions from Chapter 4.
- Reminder: Exam 2 next Monday. Review sheet on the Web. We'll do some review in class Wednesday.

Minute Essay From Last Lecture

Slide 2

- Most people seemed to think the first problem on Homework 6 did help them understand that initially-inscrutable diagram for the single-cycle implementation.
- Several people however thought the second problem wasn't very clear (and admittedly it was a lot less well-defined).

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 a few years back. 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 (Overview)

- 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, but conceptually quite similar to multiprocessors.
- “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 (Overview)

- 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

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

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

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

Shared-Memory Hardware, Revisited

- Figure 6.7 sketches basic idea — multiple processing elements (call them processors, cores, whatever) connected to a single memory.
- Synchronization (locking) *can* be done with no hardware support, but it’s tricky. Simple approach is something such as:

  ```
  while (lock != 0) {};
  lock = 1;
  ```

  which doesn’t work because test and set are separate instructions. (What goes wrong?)
- Somewhat-tricky algorithms exist for solving this problem in software, but . . .

Slide 11

Shared-Memory Hardware — Locking

- Locking is much easier if ISA provides some support, in the form of an instruction that allows . . . Well, essentially allows both read and write access to a location *as a single atomic operation*.
- Some architectures implement this directly, via a “compare and swap” or “test and set” instruction. But for MIPS that might be challenging (why?).
- So MIPS defines two instructions, “load linked” (`ll`) and “store conditional” (`sc`). Tricky, but an example may help some.

Slide 12

Shared-Memory Hardware — Locking, Continued

- Example from text, p. 122, to exchange a memory location with register contents, slightly modified (first line is wrong?!):

```
again:  add     $t0, $zero, $s2 # $t0 <- $s2
        ll      $t1, 0($s1) # $t1 <- 0($s1)
        sc      $t0, 0($s1) # $t0 -> 0($s1) IF unchanged
        beq     $t0, $zero, again  # try again if changed
        add     $s2, $zero, $t1 # $s2 <- old value of 0($s2)
```

- How this works: The `ll` “remembers” the load-from address. The following `sc` “succeeds” only if the value at that address hasn’t been changed (e.g., by another processor). Tricky!

Shared-Memory Hardware — Memory

Slide 13

- Access to RAM can be reasonably straightforward — only one processor at a time. Caches complicate things (next slide).
- “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 can affect performance — and may be non-trivial to accomplish, especially if programming environment doesn’t give you appropriate tools.

Shared-Memory Hardware — Caches

Slide 14

- As noted, even if access to RAM is one-processor-at-a-time, if each processing element has its own cache, things may get tricky. Typically hardware provides some way to keep them all in synch (the “cache coherency” problem discussed in Chapter 5).
- Further, application programs may have to deal with “false sharing” — multiple threads access distinct data in the same “cache line”. Cache coherency guarantees correctness of result, but performance may well be affected. (Example — multithreaded program where each thread computes a partial sum. Having the partial sums as “thread-local” variables can be much faster than having a shared array of partial sums.)

Distributed-Memory Hardware, Revisited

- 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.

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.

Other Hardware Support for Parallelism

Slide 17

- 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").

Minute Essay

Slide 18

- I hear that most of you have had some exposure to multi-threaded programming in CS1 and/or CS2 — what? And I'd be interested in hearing about any other experiences you've had with any kind of parallel programming.
- I'm planning to spend some of Wednesday reviewing for Exam 2, but that's not likely to take the whole hour. Also we'll have a bit of time in the last class. Any other topics you'd like to hear more about? (I have some ideas but am open to suggestions.)