



• There's been increasing interest lately in "virtual machines" / "virtualization". Some are purely software (e.g., Java Virtual Machine) but others involve or at least rely on hardware.

Slide 3

Idea actually goes back a long time — supported in 1970s by IBM's VM/370, which was (or "is"?) in some sense a stripped-down O/S that allowed running multiple "guest O/S"es side by side. Very useful in its time — physical machines often needed to be shared among people with very different needs w.r.t. O/S. Current versions include the VMware ESX server (other examples in textbook, but this name I recognize).

### Sidebar: Dual-Mode Operation

 For simple single-user machines it's more or less reasonable to allow any application to do anything the hardware can do (access all memory, do I/O, etc.) — though even there it means whatever O/S there is is vulnerable to malicious or buggy programs.

Slide 4

• For anything less simple, useful to have a notion of two modes, regular and privileged, where in regular mode some instructions are not permitted (attempts to execute them cause hardware exceptions) (e.g., instructions to do I/O). Normally at least some O/S code runs in privileged mode, and applications in regular mode. Makes it possible for the O/S to defend itself from malicious or buggy applications, and also avoids applications interfering with each other.

# Virtual Machines — Semi-Executive-Level Summary

 What the real hardware is running is a "Virtual Machine Monitor", a.k.a.
 "hypervisor" (term analogous to "supervisor", a term for O/S). Interrupts and exceptions transfer control to this hypervisor, which then decides which guest O/S they're meant for and does the right thing.

Slide 5

- This all works better with hardware support for dual-mode operation: Guest O/S's run in regular mode, and when they execute privileged instructions (as they more or less have to), the hypervisor gets control and then can simulate ...
- Other than than, programs run as they do without this extra layer of abstraction they're just executing instructions, after all?

### Virtual Machines — Semi-Executive-Level Summary, Continued

- Some architectures make this easier than others they're "virtualizable".
- Interestingly enough(?), IBM's rather old 370 does, but for many newer architectures the needed support has had to be added on, not always neatly. "Hm!"?

Slide 6

• (Textbook has a few more details, in section 5.8.)

## 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".)

Slide 7

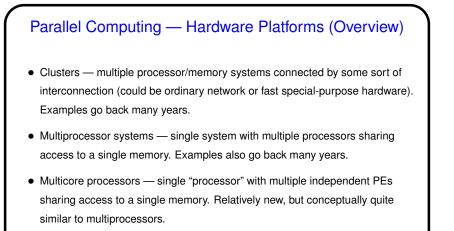
 Support for actual parallelism goes back almost as far, though mostly of interest to those needing maximum performance for large problems.
 Somewhat controversial, and for many years "wait for Moore's law to provide a faster processor" worked well enough. Now, however ...

### Parallel Computing — Overview, Continued

- Improvements in "processing elements" (processors, cores, etc.) seem to have stalled some years ago. 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).

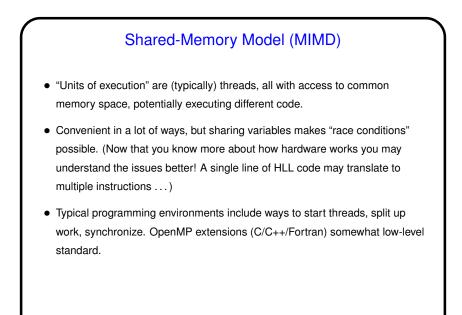
 In a perfect world, maybe compilers could be made smart enough to convert programs written for a single processing element to ones that can take advantage of multiple PEs. Some progress has been made, but goal is elusive.



 "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).

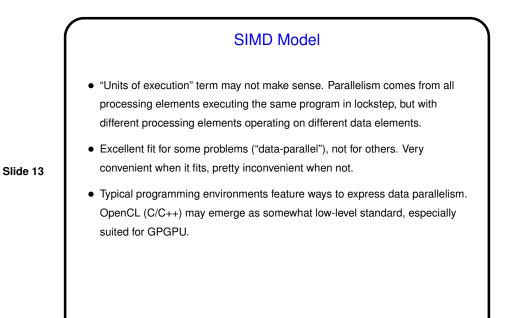
### Parallel Programming — Software (Overview)

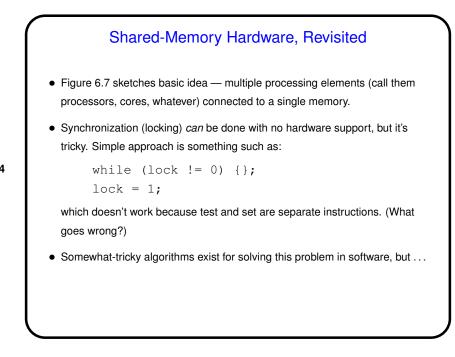
- Key idea is to split up application's work among multiple "units of execution" (processes or threads) and coordinate their actions as needed. Non-trivial in general, but not too difficult for some special cases ("embarrassingly parallel") that turn out to cover a lot of ground.
- Two basic models, 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.

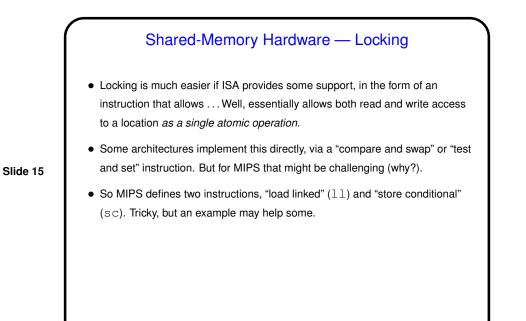


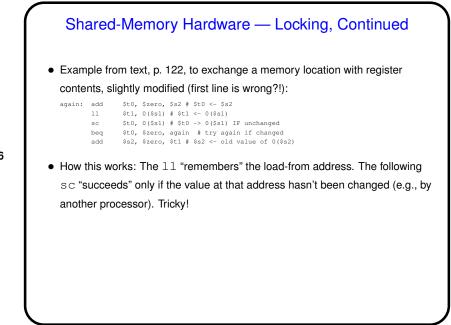
Slide 12

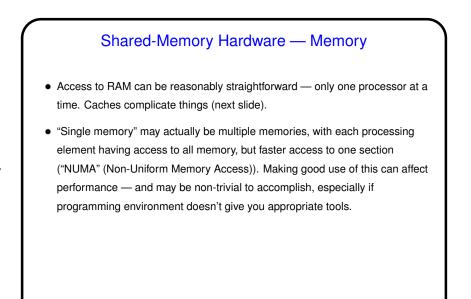
# Distributed-Memory Model "Units of execution" are processes, each with its own memory space, communicating using message passing, potentially executing different code. Less convenient, and performance may suffer if too much communication relative to amount of computation, but race conditions much less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





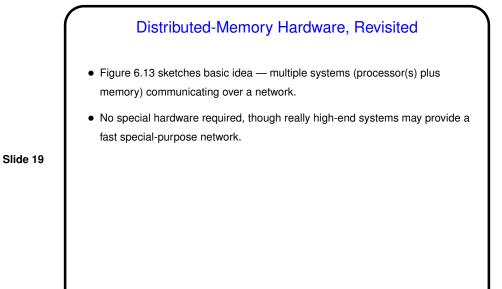


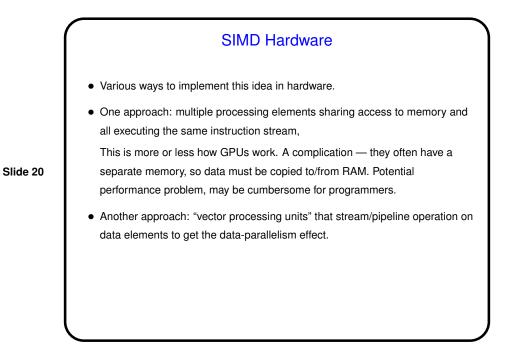




Slide 18

# Shared-Memory Hardware — Caches 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.)



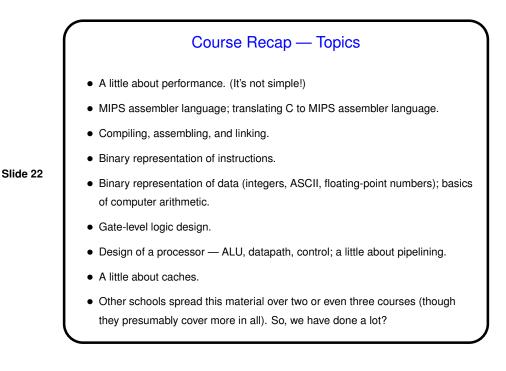


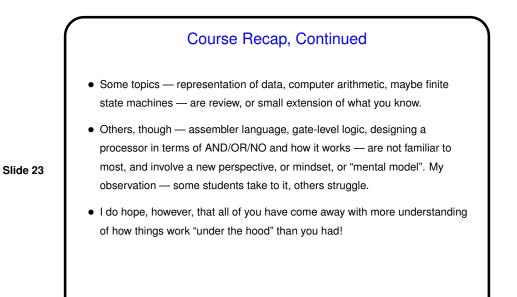
### Other Hardware Support for Parallelism

 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).

Slide 21

 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").





How did you like having "class" via video lecture? my thinking is that the videos can be replayed, which could be a plus, but they don't offer an easy way to ask questions.
If you've taken another course in which lectures were all on video and class time was used for something else, how did that work for you? If you haven't, does it sound like something you'd like? (Now that I know how to make video lectures I'm speculating about how to use them as a supplement rather than substitute.)
And best wishes for a good summer break!