Slide 1

Administrivia

- About the textbook: Apparently Amazon.com sells a Kindle edition, and it *appears* that there are free applications that can read it on most platforms (Linux, alas, not included). If that doesn't work for you and getting print copies isn't feasible, I can check about the legality of printing a paper copy.

Slide 2

Minute Essay From Last Lecture

- Not a lot to report? except book status.

Slide 3

Recap — Overview of Hardware / Software Models

- Hardware models in current use include shared-memory MIMD, distributed-memory MIMD, and now SIMD.
- Each has a corresponding programming model (though current SIMD platforms are still evolving).

Slide 4

What Programming Languages Support This?

- A regular sequential language, with a parallelizing compiler.
- A language designed to support parallel programming (Java, Scala, Ada, PCN).
- A regular sequential language plus calls to parallel library functions (PVM, MPI, Pthreads).
- A regular sequential language with some added features (OpenMP, CC++).
- For each of these categories: How attractive is it for programmers? How easy is it to implement?

Slide 5

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, Scala, 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 (OpenMP, CC++): Also familiar for users, can be difficult to implement.

Slide 6

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) — a semi-standard for message-passing programming.
  - OpenMP (language extension) — an emerging standard for shared-memory programming.
  - Java — widely available and might be many people’s first exposure to parallel programming.

  (If writing it now, would probably include OpenCL — possible emerging standard for GPGPU.)
- Other popular programming environments include C++ threads, POSIX threads (Pthreads), Win32 API, PVM, . . .

Slide 7

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 8

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 9

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 10

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 11

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 12

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 13

Performance, Continued

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

Slide 14

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 15

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 four programming environments we will use for writing programs (C-with-OpenMP, C-with-MPI, Scala/Java, and C-with-OpenCL).

Slide 16

OpenMP

- Early work on message-passing programming resulted in many competing programming environments — but eventually, MPI emerged as a standard.
- Similarly, initially many different programming environments for shared-memory programming, but OpenMP emerged as a standard.
- In both cases, idea was to come up with a single standard, then allow many implementations. For MPI, standard defines concepts and library. For OpenMP, standard defines concepts, library, *and compiler directives*.
- First release 1997 (for Fortran, followed in 1998 by version for C/C++).
- Production-quality commercial compilers appeared first. At one point, only no-cost compilers were "research software" or in work. Support then added to GNU compilers.

Slide 17

What's an OpenMP Program Like?

- Fork/join model — "master thread" spawns a "team of threads", which execute in parallel until done, then rejoin main thread. Can do this once in program, or multiple times.
- Source code in C/C++/Fortran, with OpenMP compiler directives (`#pragma` — ignored if compiling with a compiler that doesn't support OpenMP) and (possibly) calls to OpenMP functions.

  Compiler must translate compiler directives into calls to appropriate functions (to start threads, wait for them to finish, etc.)
- A plus — can start with sequential program, add parallelism incrementally — usually by finding most time-consuming loops and splitting them among threads.
- Number of threads controlled by environment variable or from within program.

Slide 18

Simple Example / Compiling and Executing

- Look at simple program — `hello.c` on sample programs page.
- Compile with compiler supporting OpenMP.
- Execute like regular program. Can set environment variable `OMP_NUM_THREADS` to specify number of threads. Default value seems to be one thread per processor.
- (To be continued.)

Slide 19

Minute Essay

- Any questions? otherwise just sign in.