Slide 1

Administrivia

- (None?)

Slide 2

Recap — Overview of Hardware / Software Models

- Hardware models in current use include shared-memory MIMD, distributed-memory MIMD, and now SIMD (as implemented by GPUs).
- Each has a corresponding programming model (though current SIMD/GPU platforms are still evolving).
- (Aside: I haven't done a good job yet of talking about GPGPU. My plan is to remedy that when we come to the appendix on OpenCL.)

Slide 3

Aside: Terminology

- Often want to talk in generic terms about something that executes a stream of instructions. No generic terms seemed to exist, so we made some up:
- Unit of execution (UE): A software entity that executes a stream of instructions independent of others. Could be a (single-threaded) “heavyweight process”, a thread, etc.
- Processing element (PE): A hardware entity that executes a stream of instructions. Could be a single-core processor, one core on a multi-core processor, etc.
- (We hoped the terms would catch on, but no.)

Slide 4

What Programming Languages Support This?

- A regular sequential language, with a parallelizing compiler.
- A language designed to support parallel programming (e.g., Java, Scala, Ada).
- A regular sequential language plus calls to parallel library functions (e.g., MPI, POSIX threads, both callable from C and others).
- A regular sequential language with some added features (e.g., OpenMP extensions to C and others).
- Which is best? no surprise, “it depends”, maybe . . .

Slide 5

What Programming Languages Support This?, Continued

- A regular sequential language with a parallelizing compiler: Attractive, but such compilers not easy.
- A language designed to support parallel programming: Perhaps the most expressive, but not all programmers are willing to learn new languages, and implementation from scratch not trivial.
- A regular sequential language plus calls to parallel library functions: Easier for programmers to learn, easier to implement.
- A regular sequential language with some added features: Also easier for programmer to learn, but implementation can be tough (consider making *any* addition to C++!).

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, not to mention that the list was compiled in 2004.)
- For our book we chose one of each:
  - MPI (library) — a semi-standard for message-passing programming.
  - OpenMP (language extension) — an emerging (at the time) standard for shared-memory programming.
  - Java — widely available and might be many people's first exposure to parallel programming.

  (If writing it now, we'd almost surely include something for GPGPU, possibly OpenCL since it was meant to be cross-platform, though it may not be catching on.)

Slide 7

Parallel Programming Environments, Continued

- Other popular programming environments include C++ threads, POSIX threads (Pthreads), Win32 API, . . .

Slide 8

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 9

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” —

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)}$$

  Example: If a program takes 10 seconds on one processor and 5 seconds on four processors, $S(4)$ is 2.
- What’s the best possible value you can imagine for $S(P)$?

Slide 10

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 11

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 12

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/computers interact?)

Slide 13

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 the processors you’re using!)

Slide 14

Performance, Continued

- Even without overhead, though, why wouldn’t we always get “perfect” speedup (P)?
- Well . . .

Slide 15

Amdahl's Law

- 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 16

Amdahl's Law, Continued

- Example: 1/10 of program is not "parallelizable". Then best speedup is 10, i.e., wall-clock time decreases by a factor of 10.
- Nothing to sneeze at, but clearly not very satisfactory if you need more speedup and have lots of processors to use.

Slide 17

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 18

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. ("And there was much rejoicing.")

Slide 19

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 20

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.

Slide 21

Sidebar — GNU Compilers on Classroom/Lab Machines

- At least two versions of GNU compiler collection installed on most machines.
  - Most-recent version available in standard Scientific Linux repositories (4.8.5).
  - More-recent version directly from project Web site. Versions vary among builds.
- To get the newest version, type

  `module load gcc-latest`

  (`module avail` if you don't remember the name)

  and then standard command names (`gcc`, `g++`, etc.) should give you the latest available version. Also sets up other needed environment.

  (If you always want to do this, put in `.bash_profile` in your home directory.)

Slide 22

Sidebar — `make` and makefiles

- Compiling with non-default options (as you must do to compile OpenMP programs with `gcc`) can become tedious.
- `make` can help. Briefly — it's a very old UNIX tool intended to help automate building large programs. Can be used in different ways, but one of them is simply to make it easy to compile with non-default options.
- To use `make`, set up `Makefile` (example linked from "Sample programs" Web page), and then type `make foo` to compile `foo.c` to `foo`.

Slide 23

Sidebar — Environment Variables (in `bash`)

- To set environment variable `FOO` for the rest of the session:

  `export FOO=fooval`

- To just run `bar` with a value for `FOO`:

  `FOO=fooval ./bar`

Slide 24

How Do Threads Interact?

- With OpenMP, threads share an address space, so they communicate by sharing variables. (Contrast with MPI, to be discussed next, in which processes don't share an address space, so to communicate they must use messages.)
- Sharing variables is more convenient, may seem more natural.
- However, "race conditions" are possible — program's outcome depends on scheduling of threads, often giving wrong results.

  What to do? use synchronization to control access to shared variables. Works, but takes (execution) time, so good performance depends on using it wisely.

  To be continued . . .

Slide 25

Minute Essay

- (We're not doing these for live classes, but for this recording it will be a way to track who's viewed it.)
- Questions?