Slide 1

Administrivia

- Homework 1 on the Web. First installment due next Wednesday.

Slide 2

Minute Essay From Last Lecture

- One person asked about using C++ even though examples are in C. For now I'm going to say I'd rather you didn't: This course tries to focus on programming environments in which programmers have more control over details that could influence performance, and my sense is that C++ isn't so much one of those. (Yes, we'll do Java/Scala, but . . .) But I should think about it more.

Slide 3

OpenMP Programs — Recap/Review

- OpenMP defines some concepts and a set of extensions to three base languages (C, C++, Fortran). These extensions include compiler directives and library functions.
- So, OpenMP programs look like programs in the base language, plus the directives, which are defined in a way that the code still compiles as sequential code even without support for the directives.

Slide 4

Simple Example / Compiling and Executing

- Look at simple programs — `hello.c`, `hello++.cpp` 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 processing element.

Slide 5

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 (4.4.7).
  - 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.

Slide 6

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 7

Sidebar — Environment Variables (in `bash`)

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

  ```
  export FOO=fooval
  ```

  (To set every time you log in, put in `.bash_profile`. To be sure it's set in terminal-window sessions may need to set it in `.bashrc`.)

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

  ```
  FOO=fooval bar
  ```

Slide 8

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.

Slide 9

Example — Numerical Integration

- Compute π by integrating $\int_0^1 \frac{4}{1+x^2}\,dx$.
- Do this numerically by approximating area under curve by many small rectangles, computing their area, adding results.
- Sequential program fairly straightforward. (`num-int-seq.c` on "sample programs" page).
- "Parallelize" how? (Discuss.)

Slide 10

Parallel Version of Numerical Integration — Strategy

- Basic strategy seems sort of obvious? most of the processing consists of adding up items computed in a `for` loop, so "parallelize" that: Parcel out iterations of loop among threads, have each thread compute a partial sum, and then combine partial sums.
- But it seems like there might be some issues: How to split iterations among threads? What about shared variables?

Slide 11

Basic OpenMP Constructs

- `#pragma omp parallel` before a block launches a "team" of threads, which continue until the end of the block. Code after the block executes only after all threads have completed the block.
- `#pragma omp master` or `#pragma omp single` within a parallel block says only one thread will do following block.
- `#pragma omp for` (within parallel block) says iterations of the following `for` loop are split among threads. Sort of the workhorse construct for OpenMP; many options.

Slide 12

Basic OpenMP Constructs — Parallel `for`

- By default, variables are shared, and semantics of initial, final values are a little complicated.
- `private` can be used to give each thread its own copy of a variable.
- `reduction` can be used to give each thread its own copy of a variable and have them combined ("reduced") at end.
- `schedule` lets you choose how iterations are split among threads — statically/evenly or at runtime.

Slide 13

Parallel Version of Numerical Integration — Code

- (See example code.)

Slide 14

Homework 1 — Overview

- This assignment asks you to parallelize a sequential program fairly similar to the numerical integration example:

  The sequential program estimates the value of π by simulating throwing "darts" at a square board and counting how many fall within an inscribed circle. (Picture?)

- The assignment will eventually ask you to do this in each of the programming environments we'll use, as a way of getting started with them. We'll do it twice, once just to get started and to discover some possibly-subtle pitfalls, and again to address those pitfalls.

Slide 15

Synchronization Constructs

- `critical` — only one thread at a time executes this block of code. (Example — `synch-2.c` on sample programs page.)
- `barrier` — threads wait here until all have arrived. Implicit barrier at end of parallel region.
- `single` — only one thread executes this block.
- Several others — `atomic`, `flush`, `ordered`, `master`. More about them in the specification.

Slide 16

Locks

- `omp_lock_t` — declares a lock variable.
- `omp_init_lock`, `omp_destroy_lock` — create and destroy.
- `omp_set_lock` — acquire lock (wait if necessary).
- `omp_unset_lock` — release lock.
- Other functions described in specification.
- Example — `synch-3.c` on sample programs page.

Slide 17

Minute Essay

- Any questions?
- Have you been able to get access to a copy of the textbook?
- If you did an internship this past summer and you are free Tuesdays at 3:35pm, *please* consider responding to Dr. Zhang's request for student speakers. 10 minutes, no slides . . .