Slide 1

Administrivia

- None.

Slide 2

Minute Essay From Last Lecture

- Question(s): What did you find most difficult about Homework 1? What did you find most interesting?
- Most difficult? many people said “getting my head around (something)”, “re-learning C”, or “setting things up”.

  Most interesting? one person said “the verbose but unhelpful error messages MPI produces.”

Slide 3

Basics of Multithreaded (Shared-Memory Parallel) Programming

- Recall idea of message-passing programming:

  An executing program consists of a bunch of "processes" running concurrently. They communicate by sending/receiving messages.

- In contrast, multithreaded programs typically start out with one "thread" and then create others as they execute. All threads have access to the same address space, so variables can be shared. Hence, no need for messages — instead, a need to "synchronize" among threads, e.g., making sure only thread at a time can access a particular variable.

Slide 4

OpenMP

- As with MPI, idea was to come up with a single standard for shared-memory parallel programming, 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 fr C/C++).
- Unlike MPI, there's not (to my knowledge) a freely-available implementation. (There's a project underway to develop one — see `http://gcc.gnu.org/projects/gomp`.) Intel's latest C++ compiler provides support and is free for non-commercial use on Linux.

Slide 5

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 (roughly analogous to "number of processes" parameter for `mpirun`), or from within program.

Slide 6

How Do Threads Interact?

- With MPI, processes don't share an address space, so to communicate they must use messages. With OpenMP, threads do share an address space, so they communicate by sharing variables.
- 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 7

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 8

OpenMP Constructs — Basic Categories

- Parallel regions ("replicate the following in all threads").
- Worksharing ("divide the following among threads").
- Data environment (shared variables versus per-thread variables).
- Synchronization.
- Runtime functions / environment variables.

Slide 9

Parallel Regions in OpenMP

- `#pragma omp parallel` tells compiler to do following block in all threads (starting team of threads if necessary). Execution doesn't proceed in main thread until all are done. Example — "hello world" shown earlier.
- Block must be a "structured block" — block with one point of entry (at top) and one point of exit (at bottom). In C/C++, this is a statement or statements enclosed in brackets (with no `goto`s into / out of block).

Slide 10

Worksharing Constructs in OpenMP

- `#pragma omp parallel for` tells compiler to split iterations of following `for` loop among threads. By default, main thread doesn't continue until all are done, but can override that (might be useful if you have two consecutive such loops).
- How loop iterations are mapped onto threads — controlled by `schedule` clause. More about this later.
- To make different threads do different things — `#pragma parallel sections`, etc. (More in standard.)

Slide 11

A Little About Variables in OpenMP

- Most variables are shared by default, including any global variables.
- Some things, though, aren't — variables within a statement block, stack (local) variables in subprograms called from parallel region.
- Can specify that each thread gets its own copy with `private` clause.
- Can specify that each thread gets its own copy, and copies are combined at the end, with `reduction` clause.

Slide 12

Numerical Integration, Revisited

- Recall numerical integration program from a couple of classes ago. Let's try parallelizing with OpenMP.
- One approach — use parallel region to create an SPMD program, conceptually identical to MPI program except for details of computing/combining partial sums. Look at code ... (`num-int-par-spmd-1.c` on sample programs page).
- Another approach — use worksharing construct to split loop iterations, `private` and `reduce` clauses to compute/combine partial sums. Look at code ... (`num-int-par.c` on sample programs page).

Slide 13

Minute Essay

- None — sign in.