Slide 1

Administrivia

- Reminder: OpenMP program for Homework 1 due today. MPI program due Tuesday.

Slide 2

Sidebar: Timing Parallel Programs

- "How long did it take?" often of interest. Can use system tools (e.g., `time` command) to check total elapsed time. Or can time "interesting" parts of program:

  `MPI_Wtime` returns elapsed time; call twice and subtract to find out how long something takes (`time_msg.c` on "sample programs" page).

  (In OpenMP, use `omp_get_wtime`.)

  I like to print time plus enough identifying info (number of processes/threads, problem size) that you can "collect performance data" by capturing program output.

- How meaningful output is depends — e.g., on whether the system is otherwise idle. Probably best to repeat observations a few times, and do some sort of averaging.

Slide 3

Review of Useful Unixisms

- Remember that you can capture output of program `foo` in file foo.out by typing:

  `foo >foo.out` (to overwrite)

  `foo >>foo.out` (to append)

  Append `2>&1` to capture standard error too.

- You can have output go both to the screen and a file by typing:

  `foo | tee foo.out` (to overwrite)

  `foo | tee -a foo.out` (to append)

- Remember that if you have commands you want to execute multiple times, you can make a script — e.g., put the commands in file `bar` and execute by typing `sh bar`.

Slide 4

Review of Useful Unixisms, Continued — Text Editors

- If you're using `vim` to edit code and finding it painful — try spending half an hour with the tutorial (command `vimtutor`), and/or try the graphical version `gvim`.
- If you haven't imprinted on a Unix editor, also spend half an hour with the tutorial for `emacs` (`emacs`, then follow instructions on screen for accessing tutorial).
- If neither one appeals to you, there are others (`gedit` and `pico` come to mind).

Slide 5

Numerical Integration, Revisited

- Recall numerical integration example, sequential version.
- Before talking about how to parallelize using MPI, let's try to be explicit about what we did to parallelize with OpenMP, as an example of how to think about designing a parallel application . . .

Slide 6

Numerical Integration, Continued

- Starting point is an understanding of the problem/computation. Pretty simple here, no?
- First step in developing a parallel version is to break the computation down into the smallest "tasks" that can execute concurrently. Here, that's the iterations of the main computation loop.
- Next step is to consider how these tasks interact — are there logic/control dependencies? data dependencies? shared data? Here, the tasks are all independent except that they share some variables — so if we can manage the shared data, we can execute them in any order we want — including concurrently. We just found some "exploitable concurrency".

Slide 7

Numerical Integration, Continued

- Next step is to develop a strategy for taking advantage of this potential for concurrent execution.
- For that, it can help to try to use one of a few very common strategies (which our book captures as patterns). This example fits the simplest one (*Task Parallelism*).

Slide 8

Numerical Integration, Continued

- Key elements of (*Task Parallelism*) strategy, as they apply here:
  - Split "tasks" (loop iterations) among UEs as evenly as possible, since they're all the same size.
  - Make sure every UE has its own copy of work variable `x`.
  - Manage the shared variable `sum` as for "reduction operations" — give each UE its own local variable, combine at the end.
- Final step is to turn the strategy into code — which we already did in OpenMP.

Slide 9

Numerical Integration in MPI

- Now figure out how to apply the overall strategy using MPI. Key difference is lack of shared memory — means we don't have problems with threads stepping on shared work variables, but we have to work harder to combine partial results.
- Sample program `num-int-par.c`.

Slide 10

Parallel Programming in Java

- Java supports multithreaded (shared-memory parallel) programming as part of the language — `synchronized` keyword, `wait` and `notify` methods of `Object` class, `Thread` class. Programs that use the GUI classes (AWT or Swing) multithreaded under the hood. Justification probably has more to do with hiding latency than HPC, but still useful, and recent versions (5.0 and beyond) includes much useful library stuff.
- Java also provides support for forms of distributed-memory programming, through library classes for networking, I/O (`java.nio`), and Remote Method Invocation (RMI).

Slide 11

What Does A Multithreaded Java Program Look Like?

- Easy answer: Like a regular Java program. (In fact, any program with a GUI . . .)
- Programming model: All threads share a common address space. Programmer is responsible for creating threads, providing synchronization, etc.

Slide 12

Creating Threads in Java

- Threads are all instances of `Thread` class (or a subclass). Pre-5.0, two ways to create threads:
  - Create a subclass of `Thread` (frowned on by o-o purists).
  - Create a `Thread` using an object that implements `Runnable` (preferable).

  Either way, `run` method (of subclass of `Thread`, or of `Runnable`) contains code for thread to execute.
- Start thread with `start` method. Can wait for it to finish with `join`.
- "Hello world" example (`Hello1.java` and `Hello2.java` on sample programs page).

Slide 13

Minute Essay

- None — sign in.