Slide 1

Administrivia

- Reading assignments updated on Web. Plan is for assignment to match what I will cover in lecture.
- Homework 1 will be on Web soon. Probably not due until a week from Tuesday, but I may post it "early" for those who are eager?

Slide 2

Minute Essay From Last Lecture

- Have you taken / are you taking CSCI 2321 (Computer Design)? How much do you remember from it?

  Most people have — but not all — and most people say they remember *something*. If I assume too much, speak up please!

Slide 3

Parallel Programming Environments

- By "programming environments" we mean languages / libraries / extensions. There are many!
- For our book we chose one of each:
  - MPI (library) because it's something of a standard for message-passing programming.
  - OpenMP (language extension) because it's emerging as a standard for shared-memory programming.
  - Java because it's widely available and might be many people's first exposure to parallel programming.
- Other popular programming environments — POSIX threads (Pthreads), Win32 API, PVM, ...

Slide 4

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". For this course, we'll start at the bottom and work up, so we can start writing code now!

Slide 5

But First, 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 would you guess is the best possible value for $S(P)$?

Slide 6

Amdahl's Law

- Of course, 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 7

Parallel Overhead

- As we will find out — many reasons why a "real" parallel program might be slower than Amdahl's Law predicts.
- 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.
- Also, "poor load balance" may slow things down.
- But sometimes we can speed things up by "overlapping computation and communication".

Slide 8

Basics of Message-Passing Programming

- Idea of message-passing programming is simple:

  An executing program consists of a bunch of "processes" running concurrently. Usually one per processor (PE), but could be more. (Why?)

  They communicate by sending/receiving messages. Simplest form is "point to point" — process A sends a message (with some data) to process B, which receives it. (Can also define "collective communication".)
- And then there are many interesting details — can sending process proceed without waiting? what happens if you try to receive a message and it hasn't been sent? etc., etc.

Slide 9

MPI — the Message Passing Interface

- Idea was to come up with a single standard (concepts and library) for message-passing programs, then allow many implementations. Similar to language standards (C, C++, etc.). Good for portability.
- MPI Forum — international consortium — began work in 1992. MPI 1.1 and MPI 2.0 standards defined. Huge! 1.1 specification is 500+ pages.
- Reference implementation — MPICH (Argonne National Lab). Another popular and free implementation (installed here) — LAM/MPI (Local Area Multicomputer).

Slide 10

What's an MPI Program Like?

- "SPMD" (Single Program, Multiple Data) model — many processes, all running the same source code, but each with its own memory space and each with a different ID. Could take different paths through the code depending on ID.
- Source code in C/C++/Fortran, with calls to MPI library functions.
- How programs get started isn't specified by the standard! (for historical/political reasons — some early target platforms were very restrictive, would not have supported what academic-CS types wanted).

Slide 11

What's in the MPI Library?

- Setup and bookkeeping — initialization, cleanup, environment query, etc.
- Data management — pack/unpack, derived data types.
- Point-to-point communication — several varieties, differing mostly in how much synchronization.
- Collective operations — e.g., broadcast.

Slide 12

MPI "Communicators"

- (One more thing to define before we can write simple code.)
- MPI allows grouping processes; group plus associated context called a "communicator". Makes it easier to write "safe" parallel libraries.
- Predefined communicator `MPI_COMM_WORLD` includes all processes. Programmers can create additional ones.

Slide 13

Simple Examples / Compiling and Executing

- Let's look at some simple programs — `hello.c` and `send-recv.c`. (These are on the Web, linked from sample programs page, with short instructions on how to use MPI.)
- We'll use the LAM/MPI that comes with FC2. There should be `man` pages for all commands and functions.
- Compile with `mpicc`.
- Before running, must "boot" (`lamboot` command) — start MPI background processes on all machines to be used.
- Execute with `mpirun`.
- Shut down with `lamhalt`. (Otherwise background processes continues to run.)

Slide 14

Minute Essay Answer

- No change in what process 1 does/prints, but process 0 would now (misleadingly) print the changed values.