Administrivia

- Classes next week: I plan to be at a conference but may be able to find a guest lecturer. I'll notify you via the course Web page and e-mail.
- Homework 1 to be on Web by next week. Due the following week.

Slide 1

Recap — Current Hardware for Parallel Programming

- One category — multiple CPUs sharing access to a common memory.
- Another category — multiple CPUs, each with separate memory, communicating over interconnection network.

Slide 2

Slide 3

Recap — Programming Models

- Shared-memory model — concurrently-executing threads sharing address space. Various ways to communicate / synchronize.
- Distributed-memory model — concurrently-executing processes, each with separate address space, communicating by sending / receiving messages.

Slide 4

What Programming Languages Support These Models?

- A regular sequential language, with a parallelizing compiler. Attractive, but such compilers are not easy.
- A language designed to support parallel programming (Java, Ada, PCN). Perhaps the most expressive, but more work for programmers and implementers.
- A regular sequential language plus calls to parallel library functions (PVM, MPI, Pthreads). More familiar for users, easier to implement.
- A regular sequential language with some added features (CC++, OpenMP). Also familiar for users, can be difficult to implement.
- (For a list of “programming environments”, see Table 2.1 in book.)

Slide 5

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 6

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 7

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 8

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 9

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 10

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 11

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 12

Simple Examples / Compiling and Executing

- Look at sample program `hello.c`. (All sample programs from class should be on the Web, linked from course "sample programs" page, with short instructions on how to use MPI.)
- We'll use the LAM/MPI that comes with FC4. 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 13

Simple (Blocking) Point-to-Point Communication in MPI

- Send with `MPI_Send` — returns as soon as data has been copied to system buffer, buffer in program can be reused.
- Receive with `MPI_Recv` — waits until message has been received.
- Can use "tags" to distinguish between kinds of messages. Can receive selectively or not (`MPI_ANY_TAG`). Received tag is in returned `MPI_Status` variable (e.g., `status.MPI_TAG`).
- Can receive from specific sender or from any sender. (`MPI_ANY_SOURCE`). Sender is in returned `MPI_Status` variable (e.g., `status.MPI_SOURCE`).
- For `MPI_Recv`, "length" parameter specifies buffer length. Use `MPI_Get_count` to get actual count.
- Look at sample program and `send-recv.c`.

Slide 14

Not-So-Simple Point-to-Point Communication in MPI

- For not-too-long messages and when readability is more important than performance, `MPI_Send` and `MPI_Recv` are probably fine.
- If messages are long, however, buffering can be a problem, and can even lead to deadlock. Also, sometimes it's nice to be able to overlap computation and communication.
- Therefore, MPI offers several other kinds of send/receive functions — "synchronous" (blocks both sender and receiver until communication can take place), "non-blocking" (doesn't block at all, program must later test/wait for communication to take place).

Slide 15

Collective Communication in MPI

- “Collective communication” operation — one that involves many processes (typically all, or all in MPI “communicator”).
- Could implement using point-to-point message passing, but some operations are common enough to be library functions — broadcast (`MPI_Bcast`), “reduction” (`MPI_Reduce`), etc.

Slide 16

Timing MPI 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).
- 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.

Minute Essay

- None — sign in.