Slide 1

## Administrivia

- (What were you doing on this date in 2001?)
- Reminder: Homework 1 OpenMP program due Wednesday.
- Homework 1 updated to include instructions for MPI program. Due a week from today.
- I've taken a copy of the book to the library and asked that they put it on reserve. So in a few days maybe?

Slide 2

## Minute Essay From Last Lecture

- Many people are going with the Kindle version of the book, but not all. I'll keep trying to find good options for getting a print copy.

**Slide 3**

## Running OpenMP Programs, Revisited

- For most if not all programs we write for this class, we'll be interested in finding out how they "scale" with varying numbers of threads. To make this interesting you need a machine with as many cores as possible. Classroom machines have eight, which is not bad, but `Dione` has more, though it's running an older build (should not matter for us).
- Probably smart to re-run experiments at least twice, and if times vary a lot, run more as get — average?

**Slide 4**

## 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. First standard MPI 1.1, followed by MPI 2.x and 3.x. 1.1 specification is 500+ pages, and later standards even bigger.
- Original reference implementation — MPICH (Argonne National Lab). LAM/MPI (Local Area Multicomputer) is another free implementation. Latest / most popular may be OpenMPI (installed on department machines).

Slide 5

## 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 (first) standard! (for historical/political reasons — some early target platforms were very restrictive, would not have supported what academic-CS types wanted).
- (Compare and contrast all of the above with OpenMP.)

Slide 6

## 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.
- More . . .

Slide 7

## 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 8

## 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. You will need to do some setup before MPI programs will run.)
- We'll use OpenMPI as installed on the lab machines. There should be `man` pages for all commands and functions. (*Note* that before using these commands you need to use `module` to add them to your search path. `module load openmpi-default` adds the default version.
- Compile with `mpicc`.
- Execute with `mpirun`. (*Note* that for the default version you will need to specify `--prefix $MPI_HOME`.)

Slide 9

## 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 `send-recv.c`.

Slide 10

## 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).

  (More about these later.)

Slide 11

## 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 12

## 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 13

## 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 14

## 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 15

## 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 16

## 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`.

## Minute Essay

- How are you doing on Homework 1? any preliminary thoughts about OpenMP?
- Any questions?

**Slide 17**