Slide 1

Administrivia

- Minor(?) syllabus correction: An alert student pointed out that 50 points for class attendance/participation is about a sixth of your grade. I didn't have in mind for it to count that much, so I'm changing that 50 points to 20 points.

Slide 2

Minute Essay From Last Lecture

- One person asked about hardware and O/S involvement in parallel processing . . .
- Hardware manages lowest level of sharing memory among processors/cores (only one at a time can access a location) and is involved in transferring data and/or routing messages (though not in a way that matters for application programming?).
- O/S is responsible for scheduling processes/threads. Reasonable to assume that it will usually make "good" decisions (i.e., if there are enough processors/cores to run all non-blocked processes/threads, then all will run).

Slide 3

Recap — Overview of Hardware / Software Models

- Hardware models in current use include shared-memory MIMD, distributed-memory MIMD, and (very recently) SIMD.
- Each has a corresponding programming model (though current SIMD platforms are still evolving).

Slide 4

Recap — Parallel Programming Environments

- 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.

Slide 5

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”, corresponding to (we think) steps in program design / development.

Slide 6

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’s the best possible value you can imagine for $S(P)$?

Performance, Continued

- Best possible value for $S(P)$? would seem to be P, right?
- Can you think of circumstances in which you could do better (“superlinear speedup”)?

Slide 7

Performance, Continued

- “Superlinear speedup” could happen if dividing up the computation among processors means more of the program’s code/data can fit into memory, or cache. Could also happen in searches, if you can stop after finding one solution.
- What’s the worst value you can imagine for $S(P)$?

Slide 8

Slide 9

Performance, Continued

- Worst possible value would seem to be 1, right?
- Can you think of circumstances in which you'd do worse? (Hint: What do you know so far about how the parts of the program running on different cores/processors/machines interact?)

Slide 10

Parallel Overhead

- Many reasons why a "real" parallel program might be slower than hoped for — even, possibly, slower than the sequential program!
- 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. (Sometimes can speed things up by "overlapping computation and communication".)
- Also, "poor load balance" may slow things down.
- (And we're not even mentioning what happens if you don't have exclusive access to all processors.)

Slide 11

Performance, Continued

- Even without overhead, though, why wouldn't we always get "perfect" speedup (P)?

Slide 12

Amdahl's Law

- And 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 13

What’s Next — Nuts and Bolts

- So we can start writing programs as soon as possible, next topic will be a fast tour through the three programming environments we will use for writing programs. (Some possibility of also including OpenCL.)

Slide 14

OpenMP

- Early work on message-passing programming resulted in many competing programming environments — but eventually, MPI emerged as a standard.
- Similarly, many different programming environments for shared-memory programming, but OpenMP may be emerging as a standard.
- In both cases, idea was to come up with a single standard, 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 for C/C++).
- Several production-quality commercial compilers available. Up until very recently, free compilers were, um, “research software” or in work. Latest versions of GNU compilers, though, offer support. !!

Slide 15

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 or from within program.

Slide 16

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 17

Sidebar — Environment Variables (in bash)

- To set environment variable FOO for the rest of the session:

  ```
  export FOO=fooval
  ```

  (To set every time you log in, put in .bash_profile.)

- To run bar with a value for FOO:

  ```
  FOO=fooval bar
  ```

Slide 18

How Do Threads Interact?

- With OpenMP, threads share an address space, so they communicate by sharing variables. (Contrast with MPI, to be discussed next, in which processes don't share an address space, so to communicate they must use messages.)
- 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 19

Example — Numerical Integration

- Compute π by integrating $\int_0^1 \frac{4}{1+x^2}\,dx$.
- Do this numerically by approximating area under curve by many small rectangles, computing their area, adding results.
- Sequential program fairly straightforward. (`num-int-seq.c` on “sample programs” page).
- “Parallelize” how? next time . . .

Slide 20

Minute Essay

- Based on the possibly-not-much you know at this point about parallel programming, what parts of the sequential numerical integration program look like they could be divided up among different processes/threads? Are there obvious potential pitfalls with how those processes/threads might share data?
- (We will talk about all of this next time.)