Slide 1

Administrivia

- If you haven't already looked at the syllabus, please do review it. I think you've all taken at least one course with me, so you're probably reasonably familiar with how I like to do courses. The only things to note are that while we won't have exams, I do plan to use the final-exam period for project presentations.
- And I'll repeat my usual request:

  If you spot something wrong with course material on the Web, please let me know!

Slide 2

"Text"book

- Why are we using "my" book when there are books that are more textbook-like, and also more recent? because I think it comes closer than any other book I know to covering the material I think is perhaps best learned from a book.
- (And no, I'm not getting rich off the royalties! not even when the book first came out, and for sure not any more.)
- If I can believe what Amazon.com tells me, the book is back in print, so you should not have trouble obtaining a copy (aside from the usual price pain, but I think as textbooks go this one is not bad?).

  Also I've put a copy on 1-day reserve at the library too.

Slide 3

Other Readings

- The part of our book that's becoming more and more dated is the appendices, giving introductions to various "programming environments" (more about those next time).
- My co-authors and I produced revised versions of all of these appendices, and while there are potential copyright issues with making them broadly available, we're agreed that it's okay to make available a small number of printed copies. So . . . Priscilla Riojas in the ASO has a copy for each of you.

Slide 4

What is Parallel/Distributed Computing?

- The explanation for the non-technical:
- Some computational jobs are just too much for one processor — no way to get them done in reasonable time.
- For jobs done by people, what do you do when the job is too much for one person?

Slide 5

What is Parallel/Distributed Computing? Continued

- For jobs done by people, if too much for one person you assign a team — but you have to figure out
  - How to divide up work among team members.
  - How to coordinate activities of team members.
- Same idea applies to computing:

  If too much for one processor, use multiple processors.

  Issues are similar — how to divide up work, how to coordinate.

Slide 6

Simple Examples

- “People job” examples:
  - Digging a hole.
  - Building a house.
  - Baby.

  For some, easy to see how to get it done faster with more people. With others, not so much. What do you notice about the last one in particular?
- Trivial computer examples:
  - Adding up a lot of numbers.
  - Computing Fibonacci numbers.

  Here too, some problems seem to lend themselves better to using multiple processors.

Slide 7

Application Areas: Traditional HPC

- Traditionally, parallel computing of interest mostly to people with a need for big computation and funds to buy expensive high-end hardware.
- Examples:
  - Oil and exploration.
  - Simulated testing of nuclear weapons.
  - Fluid dynamics.
  - Rendering for computer-generated animation.
- `http://www.top500.org` tracks fastest computers; for many years now all have been “massively parallel”.

Slide 8

Application Areas: Recent

- As hardware has become more available / affordable, it’s within range for more people, and more and more interesting work involves processing on scale that’s not feasible without parallel computing.
- Examples:
  - Astrophysics simulations (e.g,, Dr. Lewis’s work on Saturn’s rings).
  - “Big data” applications.
  - AI; machine learning.

The Need for Speed

Slide 9

- One reason is to solving the same problems faster (reduce wall-clock time). Important for, e.g., weather forecasting.
- Another reason is to solve larger problems, or solve the same problem more exactly (better answers may require more detail, hence more processing).

Can't You Just Get a Faster Computer?

Slide 10

- For a long time, yes (up to a point):

  Moore's law predicts that number of transistors on a die roughly doubles every 1.5 years. Until recently, that meant doubling processor speed and memory.
- But for many years (since at least 1989 or so, which is when my personal involvement with parallel computing began), people had been saying . . .

Slide 11

Can't You Just Get a Faster Computer? Continued

- However fast processors are, it's never fast enough, and faster is more expensive.
- Eventually we'll run into physical limitations on hardware — speed of light limits how fast we can move data along wires (in copper, light moves 9 cm in a nanosecond — one "cycle" for a 1GHz processor), other factors limit how small/fast we can make chips.
- Maybe we can switch to biological computers or quantum computers, but those are pretty big paradigm shifts . . .

Slide 12

Can't You Just Get a Faster Computer? Continued

- And then about 2004(?), Intel kind of shook things up . . .

  Their chip designers could still put more transistors on a chip, but they were no longer able to use them to get more speed(!).

  Their solution: Use all those extra transistors to provide multiple processing elements on a chip ("cores" for computational chips, other elements in GPUs).
- And at that point, "the future" when parallel computing would be needed had arrived!

Slide 13

But I Don’t Care About Solving Big Problems

- What if you aren’t interested in solving problems that involve too much computation for one processor? Is there still a reason to be interested in parallel computing?
- The hardware is there, and it’s becoming mainstream — multicore chips, general-purpose computing GPUs, etc.

  To get the best use of it for single applications, will probably need parallelism.
- More and more the “hot” areas of computing seem to involve a fair amount of computation (AI, machine learning, “big data”).
- Also, for some applications, thinking of them as parallel/multithreaded can lead to a solution that lets you do something useful while waiting for I/O, etc. (“hiding latency”).

Slide 14

About the Course

- Can think of this course as the equivalent of CS1 for parallel (and to some extent concurrent and distributed) programming. As with CS1, many things to learn all at once (next slide).
- Also as with CS1, the idea will be to teach a mix of technical skills and basic concepts, with emphasis on learning by doing.

Slide 15

About the Course — Things to Learn

- A new “box of tools” — or several boxes of tools (different languages/libraries/paradigms). Must learn syntax/functions, plus tools such as compilers and runtime systems.
- How to use the stuff in the box of tools to solve interesting problems — from low-level “what is this syntax good for?” to algorithm design.
- How to think about “does it work?” (Testing is of less value for parallel programs because programs may not run exactly the same way every time!)
- How to think about “how fast is it?” (Is there anything analogous to “big-O” analysis?)

Slide 16

Terminology — Parallel Versus Distributed Versus Concurrent

- Key idea in common — more than one thing happening “at the same time”. Distinctions among terms (in my opinion) not as important, but:
- “Parallel” connotes processors working more or less in synch. Examples include multiple-processor systems. Analogous to team of people all in the same room/building, working same hours.
- “Distributed” connotes processors in different locations, not necessarily working in synch. Example is `SETI@home` project. Analogous to geographically distributed team of people.
- “Concurrent” includes apparent concurrency. Example is multitasking operating systems. Analogous to one person “multitasking”. Can be useful for “hiding latency”. Goes back to first operating systems!

Slide 17

Hardware for Parallel Computing — Overview

- Hardware for sequential computing pretty much all builds on the same model — “von Neumann architecture” (as discussed in CSCI 2321 — processor, memory containing instructions and data, etc.).
- Hardware for parallel computing is more diverse. Some major categories (using classification scheme proposed by Flynn in 1972(!)):
  - SIMD / vector architectures.
  - MIMD with shared memory.
  - MIMD with distributed memory.
- All of these have a long history, going back to early days of computing (1960-something — see history link on class “useful links” page).

Slide 18

SIMD Architectures

- “Single Instruction, Multiple Data”:

  Many identical arithmetic units all executing the same instruction stream in lockstep (via single control unit), each on its own data. Can have separate memory for each AU or all can share.
- “Vector processors” are sort of a pipelined special case of SIMD: Addition(s) to CPU meant to speed up operations on arrays (vectors) by using pipelining and/or multiple arithmetic units.
- Both used fairly extensively early on and then abandoned, except for special-purpose hardware such as graphics cards. *But* the latter are emerging as a computing platform (“GPGPU”). Here as in other things fashion(?) is cyclical?

Slide 19

MIMD Architectures

- “Multiple Instruction, Multiple Data”:

  Many processors, each executing its own stream of instructions on its own data.

- Category is broad enough, and popular enough, to consider two subcategories (shared and distributed memory).

Slide 20

Shared-Memory MIMD Architectures

- Basic idea here: Multiple processors, all with access to a common (shared) memory.
- Details of access to shared memory vary: Shared bus versus crossbar switch, management of caches, etc. Textbook for CSCI 2321 has (some) details. Access to memory can be “constant-time” (SMP) or can vary (NUMA).
- Attractive from programming point of view, but not as scalable as distributed memory. (Up until recently, not scalable enough to be really attractive.)
- Many, many examples, from early mainframes to dual-processor PCs to multicore chips.
- Conceptually, each processor has access to all memory locations via normal memory-access instructions (e.g., load/store). Convenient, but has some potential drawbacks (“race conditions”). Hardware and/or programming environment must provide “synchronization mechanism(s)”.

Slide 21

Distributed-Memory MIMD Architectures

- Basic idea here: Multiple processors, each with its own memory, communicating via some sort of interconnect network.
- Details of interconnect network vary: Can be custom-built “backplane” or standard network. Various “topologies” possible. Textbook for CSCI 2321 has (some) details.
- Not initially as attractive from a programming point of view, but very scalable.
- Examples include “massively parallel” supercomputers, Beowulf clusters, networks of PCs/workstations, etc.
- Conceptually, each processor has access only to its own memory via normal memory-access instructions (e.g., load/store). Communication between processors is via “message passing” (details depending on type of interconnect network). Not so convenient, but much less potential for race conditions.

Slide 22

“Parallel Hardware is Becoming Mainstream”?

- As noted earlier, for many many years there was those who said that soon parallel computing would become necessary for everyone.

  And “soon” didn’t come, and didn’t come . . . And then it did(!).
- And now, the hardware is mainstream, so might as well learn to use it well?

Slide 23

“Parallel Hardware is Becoming Mainstream”?, Continued

- Ubiquity of networking makes almost any PC part of a “cluster”, hence suitable for distributed-memory parallel computing.
- Most processor chips now have multiple “cores” (processing elements).
- High-end graphics cards are emerging as a platform for general-purpose computing (“GPGPU”).

Slide 24

Programming Models

- Three broad categories of hardware currently popular:
  - Shared-memory MIMD.
  - Distributed-memory MIMD.
  - SIMD as implemented by GPUs. (May be worth noting a complication: GPUs typically have their own memory distinct from main memory, and using them involves some tedious transfer of data between the two.)
- Analogously, three basic programming models: shared memory, message passing, and — I’m not sure what name to give to the model for GPUs!

  Obviously shared-memory model works well with shared-memory hardware, etc., but can also do message-passing on shared-memory hardware, or (with more difficulty) emulate shared memory on distributed-memory hardware.

Slide 25

Programming Model: Shared Memory

- Key idea — threads executing concurrently, all sharing one memory.
- Maps well onto hardware platforms for smaller-scale parallel computing, can be implemented on other platforms too (with some work).
- Challenge for programmers is to break up the work, figure out how to get threads to interact *safely* — sharing variables has its pitfalls. (I think you all have talked about that already?)
- (How would the “add up a lot of numbers” example work here?)

Slide 26

Shared Memory Programming — Example

- Seems kind of obvious how to use multiple processors to add up a lot of numbers: Split the numbers among processors and . . .
- Well, it doesn’t really work to have all them add into a global sum, does it? (potential for race conditions)
- Instead can have each processor compute a partial sum and then combine them. Still potential for race conditions but much more manageable.

Slide 27

Programming Model: Distributed Memory With Message Passing

- Key idea — processes executing concurrently, each has its own memory, all interaction is via messages.
- Maps well onto most-common hardware platforms for large-scale parallel computing, can be implemented on others too.
- Challenge for programmers is to break up the work, figure out how to get separate processes to interact *by message-passing* — no shared memory.
- (How would the “add up a lot of numbers” example work here?)

Slide 28

Distributed Memory Programming — Example

- Basic strategy for using multiple processors to add up a lot of numbers seems the same as for shared memory. But . . .
- First you have to decide whether to store all numbers in one processor’s memory, or distribute them (fairly common), or even give each processor a copy!
- If you distribute the numbers among processors, then what to do next seems somewhat straightforward? have each processor compute a partial sum of “its” values and then combine them by using message-passing to communicate these partial sums.

Slide 29

Programming Model: SIMD on Graphics Card

- Key idea — define computation as sequence of operations on arrays, where "operation on an array" means performing the same operation on all elements concurrently.
- Maps fairly well onto platforms for GPGPU, and can be simulated on CPUs as well. Computation is often defined in terms of "computational kernels" to be applied to all elements of an array concurrently.
- Several challenges: Just figuring out how to map applications to this somewhat restricted view of parallelism can be a challenge (fairly easy for some problems — e.g., vector addition — less so for others). And typically computation requires moving data between regular computer memory and GPU's own memory.
- More about this model / platform later! I've worked with it less and am still thinking about best conceptual view!

Slide 30

Minute Essay

- How much of this was review for you? (I'm told that CS2 instructors usually introduce students to multithreaded programming, but I've tried to put that in a larger context here.)