Slide 1

Administrivia

- Reminder (or notice, for those not subscribed to the CSMajors mailing list): "Research Opportunities Fair" today at 5pm in the Great Hall.

Slide 2

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".

Slide 3

Hardware for Parallel Computing — Overview

- Hardware for sequential computing pretty much all builds on the same model — “von Neumann architecture”.
- 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 4

SIMD Architectures

- Basic idea sort of implied by name (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 processor — addition(s) to CPU meant to speed up operations on arrays (vectors) by using pipelining and/or multiple AUs. Can be thought of as a special case of (pipelined) SIMD.
- Both used more widely in early supercomputers than now, except in special-purpose hardware (though the latter may be usable for general-purpose computing — e.g., “GPGPU”).

Slide 5

MIMD Architectures

- Again, basic idea implied by name (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 6

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 (ccNUMA).
- Attractive from programming point of view, but not very scalable.
- 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 7

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 8

“Parallel Hardware is Becoming Mainstream”?

- It’s been an article of faith for a long time that eventually we’d hit physical limits on speed of single CPUs, despite interpretation of Moore’s law as “CPU speed doubles every 1.5 years.”
- But — strictly speaking, Moore’s law says that the number of transistors that can be placed on a die doubles every 1.5 years.
- Historically that has meant — more or less — doubling speed and memory size. That seems to be at an end (for now?) — tricks hardware designers use to get more speed require higher power density, generate more heat, etc.
- So, what to do with all those transistors? Provide hardware support for parallelism! current buzzphrases are “multicore chip” and “Hyper Threading”.

Slide 9

One Approach — Multicore Chips

- Key idea here — chip includes several (currently usually two or four) “cores”, all sharing one connection to memory.
- Each “core” is a CPU in the sense we talk about in Computer Design; each typically has its own first-level cache.
- To fully exploit this for a single application, probably need multiple threads.

Slide 10

Another Approach — Hyper Threading

- Key idea here — chip includes hardware support for having more than one thread at a time “active”, but strictly speaking only a single processing core. Replicated components include program counter, ALU.
- What this allows is very fine-grained concurrency (“more than one logical CPU”), which can hide latency. (Note, though, that performance improvements range from zero to about 30 percent.)
- To fully exploit this for a single application, probably need multiple threads.

"Parallel Hardware is Becoming Mainstream"?, Continued

- In addition to hardware support for shared-memory parallelism — Ubiquity of networking makes almost any PC part of a "cluster".

Slide 11

Programming Models

- Two broad categories of currently popular hardware (shared-memory MIMD and distributed-memory MIMD).
- Analogously, two basic programming models: shared memory and message passing. 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) emulated shared memory on distributed-memory hardware.

Slide 12

Slide 13

One 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.
- (How would the "add up a lot of numbers" example work here?)

Slide 14

Another 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 15

What Programming Languages Support This?

- A regular sequential language, with a parallelizing compiler.
- A language designed to support parallel programming (Java, Ada, PCN).
- A regular sequential language plus calls to parallel library functions (PVM, MPI, Pthreads).
- A regular sequential language with some added features (CC++, OpenMP).
- For each of these categories: How attractive is it for programmers? How easy is it to implement?

Slide 16

What Programming Languages Support This?, Continued

- 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 17

Parallel Programming Environments

- By "programming environments" we mean languages / libraries / extensions. There are many! (Table 2.1 in book has a list — and we might have missed a few.)
- 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 18

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 19

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 $S(P)$?

Slide 20

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 21

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 22

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 23

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 24

Performance, Continued

- Even without overhead, though, why wouldn’t we always get “perfect” speedup (P)?

Slide 25

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 26

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.

Minute Essay

Slide 27

- Was there anything in today's lecture that was particularly unclear, or you want to hear more about?