Slide 1

Administrivia

- Reminder: Reading assignments will be on the “lecture topics and assignments” Web page. Ideally you will read before class!

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 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”).

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.

Slide 11

“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”.
- Also, high-end graphics cards are emerging as a platform for “GPGPU” parallel computing.

Slide 12

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) emulate shared memory on distributed-memory hardware.
- (It’s not clear where GPGPU fits in here. More about it later in the semester if possible.)

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) — a semi-standard for message-passing programming.
  - OpenMP (language extension) — an emerging standard for shared-memory programming.
  - Java — widely available and might be many people's first exposure to parallel programming.

  (If we writing it now, we would include OpenCL — possible emerging standard for GPGPU.)
- Other popular programming environments include POSIX threads (Pthreads), Win32 API, PVM, . . .

Slide 18

Minute Essay

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