Slide 1

Administrivia

- Reminder: Homework 2 due Tuesday.

Slide 2

Minute Essay From Last Lecture

- Several sensible suggestions. Discuss.

Slide 3

Homework 2 — “Homework 1 Revisited”

- Write your own “thread-safe” RNG — one that can be called from multiple threads concurrently without ill effects. (Think about how to do that in Java versus C.)

  (References to some reasonable choices to be included in writeup on Web.)
- Modify your programs to use your RNG. Do something you think is reasonable so that each UE starts with a different seed.
- Measure your programs’ accuracy and performance and plot results.
- Make a second version of your code in which all UEs start from the same seed but each works on a different part of the problem space (domain decomposition). Measure its accuracy and performance.

Slide 4

Shell Scripts — Basics

- “Shell script” is just a (text) file containing commands you could type to a shell. It should be fairly easy to “mass produce” multiple executions of the same command with different parameters using cut-and-paste in a text editor. (Or you could use the shell’s constructs for looping, parameters, etc.)
- Execute commands in `scriptname` two ways:
  - `sh scriptname`
  - Make the file executable (`chmod u+x scriptname`) and execute directly (`scriptname` or `./scriptname`).

Slide 5

Capturing Output — Basics

- To capture output of a command in a file:
  - `cmd >outfile` to overwrite `outfile`.
  - `cmd >>outfile` to append.
- To capture output in a file but also view it as it's generated:
  - `cmd | tee outfile` to overwrite `outfile`. (That | is the "pipe symbol", a vertical bar.)
  - `cmd | tee -a outfile` to append.
- Once you have the output, you can edit as needed . . .

Slide 6

A Few Words About Design Patterns

- Idea originated with architect Christopher Alexander (first book 1977). Briefly — look for problems that have to be solved over and over, and try to come up with "expert" solution, write it in a form accessible to others. Usually this means adopting "pattern format" to use for all patterns. Characteristics of a good pattern:
  - Neat balancing of competing "forces" (tradeoffs).
  - Name either tells you what it's about, or is a good addition to vocabulary.
  - "Aha!" aspect.
- First used in CS in OOD/OOP, about 1987. Really started to take off in OO community with "Gang of Four" book (Gamma, Helms, Johnson, and Vlissides; 1995). Now can find people writing patterns in many, many areas.
- To give you the idea — look at some simple patterns (links on course "Useful links" page).

Slide 7

“A Pattern Language for Parallel Programming”?

- Goal of our book (and preceding work) — apply this idea in parallel computing.
- We started out looking for patterns representing high-level structures for parallel programs, thinking there might be a dozen of them.
- At some point we realized we also wanted to talk about how you get from the original problem to one of these structures — i.e., how do expert parallel programmers think about how to decompose a problem, etc.? and also about commonly-occurring data structures and program structures, and how to map high-level designs/structures into real programming environments.
- Eventually — four-layer “pattern language”. (Notice that “pattern language” connotes common vocabulary more than grammatical structure. Not a programming language!)

Slide 8

Overall Organization of Our Pattern Language

- Four “design spaces” corresponding to phases in design.
  - *Finding Concurrency* — how to decompose problems, analyze decomposition.
  - *Algorithm Structure* — high-level program structures.
  - *Supporting Structure* — program structures, data structures.
  - *Implementation Mechanisms* — generic discussion of programming environment “building blocks”.
- Idea is that you start at the top, work your way down, possibly with some backtracking.

Slide 9

Finding Concurrency — Preview

- Decomposition patterns (*Task Decomposition*, *Data Decomposition*): Break problem into tasks that maybe can execute concurrently.
- Dependency analysis patterns (*Group Tasks*, *Order Tasks*, *Data Sharing*): Organize tasks into groups, analyze dependencies among them.
- *Design Evaluation*: Review what you have so far, possibly backtrack.

Slide 10

Algorithm Structure — Preview

- *Task Parallelism* — decompose problem into lots of tasks, independent or nearly so. Example: numerical integration.
- *Divide and Conquer* — decompose recursively as in divide-and-conquer algorithms. Examples: quicksort, mergesort.
- *Geometric Decomposition* — decompose based on data (by rows, by columns, etc.). Example: Mesh-based computation.
- *Recursive Data* — rethink computation to expose unexpected concurrency. Ignore for now.
- *Pipeline* — decompose based on assembly-line analogy.
- *Event-Based Coordination* — decompose problem into entities interacting asynchronously.

Slide 11

Supporting Structures — Preview

- Program structure patterns:
  - *SPMD* (Single Program, Multiple Data) — “like an MPI program”.
  - *Loop Parallelism* — “like an OpenMP program”.
  - *Master/Worker* — like the name suggests.
  - *Fork/Join* — when none of the others fits.
- Data structure patterns:
  - *Shared Data* — generic advice for dealing with data dependencies.
  - *Shared Queue* — example of applying *Shared Data*).
  - *Distributed Array*.

Slide 12

Implementation Mechanisms — Preview

- Generic discussion of “building blocks” for parallel programming — analogous to assignment, if/then/else, loops in procedural programming languages. (Can think of this as “what basic questions do I ask about a new parallel programming environment?”)
- Three basic categories:
  - UE management.
  - Synchronization.
  - Communication.

Slide 13

Example Applications

- Before starting on *Finding Concurrency* patterns — two example applications to be used as running examples.

Slide 14

Example — Molecular Dynamics

- Goal is to simulate what happens to large molecule. Of interest, e.g., in modeling how a drug interacts with a protein.
- Approach is to treat molecule as a collection of balls (atoms) connected by springs (chemical bonds). Then do “standard time-stepping” — divide time into discrete steps, and at each step use classical mechanics to figure out new positions for atoms based on current positions and forces among them.

  In more details . . .

Slide 15

Molecular Dynamics — Computation

- At each time step:
  - Compute forces (vibrational and rotational) on atoms caused by chemical bonds between them. Short-range interaction, so not too much computation here.
  - Compute forces on atoms caused by their electrical charges. Potentially must consider all pairs of atoms, so lots of computation here.
  - Use forces to update atoms' positions and velocities.
  - Compute other physical properties of the system — e.g., energies.
- To reduce the computational load, can limit computation of electrical-charge-induced forces to atoms that are "close". To do this, calculate for each atom a list of "neighbors". If time steps are short, atoms don't move much, and we don't have to do this every step.

Slide 16

Molecular Dynamics Pseudocode

```
Int const N   // number of atoms
Array of Real :: atoms  (3,N) //3D coordinates
Array of Real :: velocities (3,N) //velocity vector
Array of Real :: forces (3,N) //force in each dimension
Array of List :: neighbors(N) //atoms in cutoff volume

loop over time steps
    vibrational_forces (N, atoms, forces)
    rotational_forces (N, atoms, forces)
    neighbor_list (N, atoms, neighbors)
    non_bonded_forces (N, atoms, neighbors, forces)
    update_atom_positions_and_velocities
        (N, atoms, velocities, forces)
    physical_properties ( ... Lots of stuff  ... )
end loop
```

Pseudocode for Non-Bonded Force Computation

```
function non_bonded_forces (N, Atoms, neighbors, Forces)
    Int const N  // number of atoms
    Array of Real :: atoms  (3,N) //3D coordinates
    Array of Real :: forces (3,N) //force in each dimension
    Array of List :: neighbors(N) //atoms in cutoff volume
    Real :: forceX, forceY, forceZ

    loop [i] over atoms
        loop [j] over neighbors(i)
             forceX = non_bond_force(atoms(1,i), atoms(1,j))
             forceY = non_bond_force(atoms(2,i), atoms(2,j))
             forceZ = non_bond_force(atoms(3,i), atoms(3,j))
             force(1,i) += forceX;    force(1,j) -= forceX;
             force(2,i) += forceY;    force(2,j) -= forceY;
             force(3,i) += forceZ;    force(3,j) -= forceZ;
        end loop [j]
    end loop [i]
end function non_bonded_forces
```

Slide 17

Example — Heat Diffusion

- (Next time.)

Slide 18

Slide 19

Minute Essay

- What did you find most interesting about Homework 1? most difficult?