Slide 1

Administrivia

- Reminder: Homework 3 assigned. Not due soon, but lengthy, so I say start soon. Several programs to write. “Lecture topics etc.” page has suggested completion dates for individual pieces to encourage you . . .

Slide 2

A Few Words About Design Patterns

- Title of our book includes the word “patterns”.
- What do we mean? “Design patterns”.

Slide 3

A Few (More) Words About Design Patterns

- Idea originated with architect Christopher Alexander (first book 1977). Basic idea: Look for problems that have to be solved over and over, and try to come up with "expert" solution, then 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.
- Simple low-level example — iterator.

Slide 4

"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.
- After much thought and discussion . . .

Slide 5

“A Pattern Language for Parallel Programming”, Continued

- Eventually: Four-layer “pattern language”. (Note that “pattern language” connotes common vocabulary more than grammatical structure. Not a programming language!)
- We figured it would be a starting point but might need to revised and extended. Indeed, that’s so, especially (IMO) to adapt to changing state of the world.
- Much work has been done on that, primarily by Mattson and Sanders and a group at UC Berkeley. Project seems to be somewhat stalled at this point, but maybe someday?

Slide 6

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 Structures* — 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 7

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 8

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 9

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 Queue* — example of applying *Shared Data*.
  - *Distributed Array*.

Slide 10

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 11

Example Applications

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

Slide 12

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.

  At each step use classical mechanics to figure out new positions for atoms based on current positions and forces among them.

  In more detail . . .

Slide 13

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 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 in each, so don't have to do this every step.

Slide 14

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
```

Slide 15

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 16

Example — Heat Diffusion

- A simple example, representative of a big class of scientific-computing applications: “Heat distribution problem”.
- Goal is to simulate what happens when two ends of a pipe are put in contact with things at different (constant) temperatures — pipe conducts heat, its temperature changes over time, eventually converging on a smooth gradient.
- Can model mathematically how temperature in pipe changes over time using partial differential equations.
- Can approximate solution by “discretizing” — spatially and with regard to time.

Slide 17

Heat Diffusion Code

```
double *uk = malloc(sizeof(double) * NX);
double *ukp1 = malloc(sizeof(double) * NX);
double *temp;
double dx = 1.0/NX; double dt = 0.5*dx*dx;
double maxdiff, diff;

initialize(uk, ukp1);

for (int k = 0; (k < NSTEPS) && (maxdiff >= threshold); ++k) {

  /* compute new values */
  for (int i = 1; i < NX-1; ++i) {
    ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
  }

  /* check for convergence */
  maxdiff = 0.0;
  for (int i = 1; i < NX-1; ++i) {
    diff = fabs(uk[i] - ukp1[i]);
    if (diff > maxdiff) maxdiff = diff;
  }

  /* "copy" ukp1 to uk by swapping pointers */
  temp = ukp1; ukp1 = uk; uk = temp;

  printValues(uk, k);
}
```

Slide 18

Minute Essay

- Have you encountered design patterns previously? maybe in another course, or in other work? (Where?)