Slide 1

Administrivia

- Reminder: Homework 3 due Tuesday.

Slide 2

Supporting Structures Program Structure patterns — Recap

- Basic ways parallel programs can be structured:
  - *SPMD* (Single Program, Multiple Data) — "like an MPI program" (but could use same strategy in OpenMP, e.g.).
  - *Loop Parallelism* — "like an OpenMP program".
  - *Master/Worker* — as the name suggests.
  - *Fork/Join* — if you need to be able to create / wait for UEs in any arbitrary way.
- (We've seen examples of most. Look also at MPI master/worker.)

Slide 3

Supporting Structures Data Structure Patterns

- Probably not a complete list, but some examples of frequently-used ways of sharing data:
  - *Shared Data* (generic advice for dealing with data dependencies).
  - *Shared Queue* (what the name suggests — mostly included as example of applying *Shared Data*).
  - *Distributed Array* (what the name suggests).
- Programming environment / library may provide support (e.g., Java has library class(es) for shared queues).

Slide 4

Shared Queue

- Many applications — especially ones using a master/worker approach — need a shared queue. Programming environment might provide one, or might not. Nice example of dealing with a shared data structure anyway.
- Java code in figures 5.37 (p. 185) through 5.40 (p. 189) presents a step-by-step approach to developing implementation.

Slide 5

Shared Queue, Continued

- Simplest approach to managing a shared data structure where concurrent modifications might cause trouble — one-at-a-time execution. Shown in figures 5.37 (nonblocking) and 5.38 (block-on-empty). Only tricky bits are use of dummy first node and details of `take`. Reasons to become clearer later.

  Usually a good idea to try simplest approach first, and only try more complex ones if better performance is needed. ("Premature optimization is the root of all evil." Attributed to D. E. Knuth; may actually be C. A. R. Hoare.)
- Here, next thing to try is concurrent calls to `put` and `take`. Not too hard for nonblocking queue — figure 5.39. Tougher for block-on-empty queue — figure 5.40. In both cases, must be very careful.
- If still too slow, or a bottleneck for large numbers of UE, explore distributed queue.

Slide 6

Distributed Array, Recap

- Ideas are fairly straightforward, easy to draw. Code is apt to be messy — but good use of functions and macros can help.
- Example — heat-distribution problem (later slides).

Slide 7

Molecular Dynamics Example — Recap

- Previously discussed the problem (what we're computing and how) and sketched out how to decompose/analyze it.
- Also decided on overall algorithm structure of *Task Parallelism*. Pseudocode in next slide, again.

Slide 8

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 9

Molecular Dynamics and *Task Parallelism*

- How to define tasks so we get "enough but not too many"?

  One task per atom pair is too many; one task per atom is probably right.

- How to manage data dependencies (if any)?

  Dependency involving `forces` array — potentially any UE can write to any element, if we exploit symmetry resulting from Newton's third law. But computation is accumulation/reduction, so just give each UE a local copy and combine all copies at end.

- How to assign tasks to UEs? statically (at compile time) or dynamically (at runtime)?

  Work per task can vary, since how many atoms are "close" varies. Decide at next level.

Slide 10

Design of Program for Molecular Dynamics

- Finally, we turn the design into code, probably using patterns from *Supporting Structures* design space, and possibly some information/understanding from *Implementation Mechanisms* (to be discussed later).

- Based on previous design steps, consider *Loop Parallelism* and/or *SPMD*. Decide based mostly on target platform.

Slide 11

Molecular Dynamics and *Loop Parallelism* — Key Design Decisions

- Parallelize computationally intensive loop only (the one for non-bonded forces).
- Figure out what to do about shared variables:
  - Make temporary variables used inside loop private.
  - Make forces array a reduction variable.
- Decide how to map iterations onto UEs. Dynamic schedule works well if available (as it is in OpenMP).
- OpenMP-based pseudocode as shown in figure 5.25 (p. 161) and following `pragma omp` directives). Compare to pseudocode in figure 4.4 (p. 72).

Slide 12

Molecular Dynamics and *SPMD* — Key Design Decisions

- Only parallelize computation of non-bonded forces, since that's most of the computational load.
- Keep a copy of the full force and coordinate arrays on each node.
- Have each UE redundantly update positions and velocities for the atoms (i.e., assume it's cheaper to redundantly compute these terms than to do them in parallel and communicate the results).
- Have each UE compute its contributions to the force array and then combine (or reduce) the UEs' contributions into a single global force array copied onto each UE.

Slide 13

Molecular Dynamics and *SPMD* — Code

- Slightly more detailed sequential pseudocode in figure 5.7 (p. 134).
- MPI main pseudocode in figure 5.8 (p. 135). Compare to figure 5.7.
- Pseudocode for computation of non-bonded forces in figure 5.9 (p. 136). Compare to sequential pseudocode in figure 4.4 (p. 72).
- Pseudocode for computation of neighbor list in figure 5.10 (p. 137). Notice that we exploit the symmetry resulting from Newton's third law.
- A remaining decision — how to distribute atoms among UEs. Cyclic distribution is easy and will probably work okay. If not, could do something more complex — define "owner-computes filter" — boolean function of ID and loop iteration.
- Notice that we could do this in OpenMP too.

Slide 14

Heat Diffusion Example — Recap

- Previously discussed the problem (what we're computing and how) and sketched out how to decompose/analyze it.
- Also decided on overall algorithm structure of *Geometric Decomposition*.

Heat Diffusion and *Geometric Decomposition*, Again

- How to distribute data?

  One chunk per UE will probably work well. (Note that for other problems it might not.) Might be nice to include in data structure a place to store values from neighboring chunks, as described in *Distributed Array*.

Slide 15

- How to synchronize/communicate?

  With shared memory, just need barrier synchronization.

  With distributed memory, need to exchange values with neighbor UEs, also perform reduction.

Design of Program for Heat Diffusion

- Finally, we turn the design into code, probably using patterns from *Supporting Structures* design space, and possibly some information/understanding from *Implementation Mechanisms* (to be discussed later).
- Based on previous design steps, consider *Loop Parallelism* and/or *SPMD*. Decide based mostly on target platform.

Slide 16

Slide 17

Heat Diffusion and *Loop Parallelism*

- Key design decision: Parallelize both computationally intensive loops (to compute new values, find maximum difference between old and new values). How to deal with shared variables is fairly straightforward.
- Code on "Sample programs" page . . .

Slide 18

Heat Diffusion and *SPMD*

- Key design decision: Distribute both of the large arrays (`uk`, `ukp1`). This more or less forces/implies how to divide up the computation:
  - Have each UE initialize its local section.
  - Have each UE compute new values for points in its local section of `ukp1`.
  - Have each UE compute a local maximum for differences between old and new values, and then use reduction to get a global maximum.
- Code on "Sample programs" page . . .
- Notice that we could do this in OpenMP too.

Slide 19

Minute Essay

- Which of the *Algorithm Structure* patterns we talked about seems like a good fit for the "game of life" program as described? (Choices include *Task Parallelism* (like the numerical integration example), *Divide and Conquer*, *Geometric Decomposition* (like the heat equation), *Recursive Data*, *Pipeline*, and *Event-Based Coordination*.)
- What other pattern(s) we've talked about recently seem like they might be useful?

Slide 20

Minute Essay Answer

- *Geometric Decomposition* seems like a good fit.
- *Distributed Array* should also be useful, for the distributed-memory version anyway.