Slide 1

Administrivia

- Reminder: Homework 2 due today. (Accepted without penalty through 5pm tomorrow.)

Slide 2

Finding Concurrency Design Space

- Starting point in our grand strategy for developing parallel applications. Overall idea — capture how experienced parallel programmers think about initial design of parallel applications. Might not be necessary if clear match between application and an *Algorithm Structure* pattern.
- Idea is to work through three groups of patterns in sequence (possibly with backtracking):
  - 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.
- Keep in mind — best to focus attention on computationally intensive parts of problem.

Slide 3

Task-Based Versus Data-Based Decomposition

- Two basic approaches to decomposing a problem — task-based and data-based. Usually one will seem more logical than the other, but may need to think through both.
- Either way, you'll look at both tasks and data; difference is in which you look at first, and then the other follows.

Slide 4

Task Decomposition

- Goal here is to break up (some of) computation into "tasks" — logical elements of overall computation that might be independent enough to do concurrently.
- At this stage, try to stay abstract and portable; also try to identify lots of tasks (can always recombine them later if too many), as independent of each other as possible.
- Places to look for tasks include groups of function calls (e.g., in divide-and-conquer strategy), loop iterations (e.g., many examples we've discussed).
- Simple example — matrix multiplication.
- Once you have this, consider data related to each task (*Data Decomposition*).

Data Decomposition

Slide 5

- Goal here is to break up (some of) problem data into parts ("chunks") that can be operated on concurrently. Good choice if most computation consists of updates to big data structure(s).
- Again, try to stay abstract and portable; also try to "parameterize" decomposition so you can easily try various choices at runtime.
- Data structures to look at include arrays, recursive structures such as trees.
- Simple example — matrix multiplication.
- Once you have this, consider computation related to each chunk of data (*Task Decomposition*).

Decomposition — Examples

Slide 6

- Next slides will show working through our two examples. For purposes of illustration, we'll do one starting with a *Task Decomposition* and inferring a *Data Decomposition*, the other one the other way around.

Slide 7

Molecular Dynamics Example — Task Decomposition

- Tasks that find the vibrational forces on an atom.
- Tasks that find the rotational forces on an atom.

  (Together, these are tasks to compute "bonded forces" — those due to chemical bonds.)
- Tasks that find the non-bonded forces on an atom (the ones due to electrical charges).
- Tasks that update the position and velocity of an atom.
- Tasks that update the neighbor list for an atom. (Or we could consider updating all the neighbor lists as one task, as in the book, if we think it won't be done very often and therefore is not worthwhile to parallelize.)

Slide 8

Molecular Dynamics Example — Data Decomposition

- Key data structures:
  - An array of atom coordinates, one element per atom.
  - An array of atom velocities, one element per atom.
  - An array of lists, one per atom, each defining the neighborhood of atoms considered to be "close".
  - An array of forces on atoms, one element per atom.
- Decompose each of these to match task decomposition — into elements corresponding to individual atoms.

Slide 9

Heat Diffusion Example — Data Decomposition

- Key data structures:
  - Array for “old values” (time step k).
  - Array for “new values” (time step $k+1$).
- Most computation involves updating or otherwise operating on these two arrays. Think of partitioning into “chunks”.

Slide 10

Heat Diffusion Example — Task Decomposition

- Tasks to compute new values from old values, one per chunk.
- Tasks to compute maximum difference between new and old values, one per chunk.
- Task to swap pointers (to fake copying new values to old values).

Group Tasks

- Once you've broken down problem into tasks / data chunks, need to put it back together as design for parallel algorithm.
- First step — look for "groups of tasks" — logically related, or interdependent, or all with same constraints, etc. Often just one group.

Slide 11

Order Tasks

- Next step — identify constraints on groups of tasks. Possibilities:
  - "First this, then that."
  - "All of these together."

Slide 12

Slide 13

Molecular Dynamics Example — Group Tasks, Order Groups

- Task groups based on list of a few slides back — each type of task (e.g., compute rotational forces) defines a task group.
- Ordering constraints, for each timestep:
  - Task group to compute neighbor list must run before task group to compute non-bonded forces.
  - Task groups to compute bonded and non-bonded forces must run before task group to update positions and velocities.
  - Task group to update positions and velocities must run before next timestep.
- (Also see Figure 3.4 in the book.)

Slide 14

Heat Diffusion Example — Group Tasks, Order Groups

- Task groups based on list of a few slides back — each type of task (e.g., compute new values from old values) defines a task group.
- Ordering constraints, for each timestep:
  - Three tasks groups must run in sequence.
  - All task groups must run before next timestep.

Slide 15

Data Sharing

- Sometimes tasks are totally independent, each executes on totally separate data, etc. Usually not, though. Point here is to think through dependencies.
- Useful to think in terms of:
  - “Task-local” data — variables used only/mainly by single task, particularly the ones being updated. Example — chunks in heat diffusion problem.
  - Globally shared data — variables not associated with any particular task(s). Example — sum in numerical integration problem.
  - Data shared among smaller groups of tasks. Example — “boundary” points in heat diffusion problem.

Slide 16

Data Sharing, Continued

- Potential problems different in different environments; goal is to ensure correctness without adding too much overhead:
  - With shared memory, all UEs (can) have access to all data, but must use synchronization to prevent “race conditions”.
  - With distributed memory, each UE has its own data, so race conditions not possible, but must use communication to (in effect) share data.
- Basic approach — first identify what data is shared, then figure out how it’s used.

Slide 17

Data Sharing — Categories of Shared Data

- Read-only: Easiest case. If shared memory, don't need to do anything. If distributed memory, consider giving each process a copy. Examples include global constants.
- Effectively-local (large data structure, but each element accessed by only one UE): Also easy. If distributed memory, give each process "its" data.

Slide 18

Data Sharing — Categories of Shared Data, Continued

- Read-write (accessed by more than one task, at least one changing it): Can be arbitrarily complicated, but some common cases aren't too bad:
  - "Accumulate" (variable(s) used to accumulate result — usually a reduction). Example — sum in numerical integration problem. Give each task (or each UE) a copy and combine at end.
  - "Multiple-read/single-write" (multiple tasks need initial value, one task computes new value). Example — points near boundaries of chunks in heat diffusion problem. Create at least two copies, one for task that computes new value, other(s) to hold initial value for other tasks.

Slide 19

Molecular Dynamics Example — Analyze Task/Data Dependencies

- Arrays of atom positions, velocities:
  - Read-only for most groups of tasks — but tasks may need access to many elements, so for distributed memory might want to duplicate.
  - Updated by one group of tasks, but each task updates its own element(s) — "effectively local".
- Array of forces:
  - Read-only for group of tasks that update positions and velocities, and each task needs access only to "local" data.
  - Updated by several groups of tasks, but updates fit "accumulate data" model.

Slide 20

Molecular Dynamics Example — Task/Data Dependencies, Continued

- Array of neighbor lists:
  - Read-only for group of tasks that compute "non-bonded" forces, and each task needs access only to local data.
  - Updated by one group of tasks, but each task updates its own element(s).
- (Also see Figure 3.5 in book.)

Slide 21

Heat Diffusion Example — Analyze Task/Data Dependencies

- Arrays of old, new values:
  - Old values read-only for all groups of tasks, and each task needs access mostly to local data — plus "boundary values" for neighboring tasks.
  - New values updated by one group of tasks, and each task computes values only for "its" elements.

  For distributed memory, could distribute among processes, with extra variable(s) to hold copy of boundary values.
- Maximum difference between old, new values is "accumulate data" in one group of tasks, read-only elsewhere.
- Pointers to old/new values — changed at end of time step by one task, read-only elsewhere. Could duplicate for distributed memory.

Slide 22

Design Evaluation

- Idea of this pattern — questions to ask yourself about design/analysis before going further, to reduce odds of costly mistakes.
- Ideal design is easy to implement/maintain and produces a fast program suitable for target architecture. (But keep in mind old saying from engineering: "Good, fast, cheap. Pick any two.")
- (To be continued.)

Minute Essay

- None — sign in.

Slide 23