Slide 1

Administrivia

- Reminder: Homework 2 nominally due Thursday. Accepted without penalty through Monday after spring break. Grades and comments on Homework 1 coming soon.

Slide 2

Recap — *Algorithm Structure* Patterns

- If decomposition/analysis reveals organization in terms of tasks — *Task Parallelism* (probably most common strategy) or *Divide and Conquer*.
- If decomposition/analysis reveals organization in terms of data — *Geometric Decomposition* (second most common strategy) or *Recursive Data*.
- If organization is in terms of flow of data — (*Pipeline* and *Event-Based Coordination*).
- Last time we talked briefly about *Task Parallelism* and *Geometric Decomposition*. Review, and look at the other four. (Notice the common "pattern format" — problem statement, solution, key ideas.)

Slide 3

Task Parallelism

- Problem statement:

  When the problem is best decomposed into a collection of tasks that can execute concurrently, how can this concurrency be exploited efficiently?

- Key ideas in solution — managing tasks (getting them all scheduled), detecting termination, managing any data dependencies.

- Many, many examples, including:
  - Numerical integration example (next slide).
  - Molecular dynamics example (after that).
  - Mandelbrot set computation.
  - Branch-and-bound computations: Maintain list of "solution spaces". At each step, pick item from list, examine it, and either declare it a solution, discard it, or divide it into smaller spaces and put them back on list. Tasks consist of processing items from list.

Slide 4

Numerical Integration and *Finding Concurrency* Patterns

- A task decomposition probably makes sense here, with the tasks being the iterations of the main loop.

- There's only one group of tasks, and the tasks in the group can execute concurrently.

- Data shared among tasks includes a read-only variable (`step`), a variable that could be made task-local (`x`), and an "accumulate data" variable (`sum`).

Slide 5

Numerical Integration and *Task Parallelism*

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

  One task per loop iteration is really too many, since each task is so small, but we can get away with it if we keep the overhead of managing the tasks small — as all our solutions do.

- How to manage data dependencies (if any)?

  Dependency involving `x` can be managed by just giving each UE its own copy.

  Dependency involving `sum` can be managed by giving each UE a local copy and combining all copies at end.

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

  All tasks are the same size, so static assignment will work and probably be most efficient.

Slide 6

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 7

Geometric Decomposition

- Problem statement:

  How can an algorithm be organized around a data structure that has been decomposed into concurrently updatable "chunks"?

- Key ideas in solution — distributing data, arranging for needed communication.

- Probably second most common pattern. Examples include:
  - Heat-diffusion problem previously discussed (next slide).
  - Matrix multiplication using blocks.

Slide 8

Heat Diffusion and *Geometric Decomposition*

- 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. More in *Distributed Array*, next chapter.

- How to synchronize/communicate?

  With shared memory, just need barrier synchronization.

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

Slide 9

Divide and Conquer

- Problem statement:

  Suppose the problem is formulated using the sequential divide and conquer strategy. How can the potential concurrency be exploited?
- Key idea in solution — create new task(s) every time we split (sub)problem, recombine when we merge.
- Examples include mergesort and some non-naive algorithms for N-body problem.
- Straightforward if you already have a sequential divide-and-conquer solution, but scalability is somewhat limited.

Slide 10

Recursive Data

- Problem statement:

  Suppose the problem involves an operation on a recursive data structure (such as a list, tree, or graph) that appears to require sequential processing. How can operations on these data structures be performed in parallel?
- Key idea in solution — “out of the box” thinking to expose concurrency.
- Probably least-used structure currently (because it doesn’t map well to current architectures); included for completeness and because examples are interesting — e.g. “roots in forest” example.

Slide 11

Pipeline

- Problem statement:

  Suppose that the overall computation involves performing a calculation on many sets of data, where the calculation can be viewed in terms of data flowing through a sequence of stages. How can the potential concurrency be exploited?

- Key idea in solution — set up “assembly line” (pipeline).

- Canonical example is signal/image processing application, where you have a sequence of incoming images and want to apply same sequence of transformations to each one.

Slide 12

Event-Based Coordination

- Problem statement:

  Suppose the application can be decomposed into groups of semi-independent tasks interacting in an irregular fashion. The interaction is determined by the flow of data between them which implies ordering constraints between the tasks. How can these tasks and their interaction be implemented so they can execute concurrently?

- Key idea in solution — structure computation in terms of semi-independent entities, interacting via “events”.

- Canonical example is discrete event simulation — simulating many semi-independent entities that interact in irregular/unpredictable ways.

Minute Essay

- How scalable are *Pipeline* and *Event-Based Coordination*? if not very, can you think of a way to fix that?

Slide 13

Minute Essay Answer

- Neither pattern is very scalable, since they're based on a task decomposition that has one task per pipeline stage or one task per entity. Sometimes additional concurrency can be exposed by further decomposing these stages or entities.

Slide 14