Slide 1

Administrivia

- (None.)

Slide 2

Finding Concurrency Design Space, Review

- Recall two running examples.
- First step is to decompose problem, using both *Task Decomposition* and *Data Decomposition* (but one of them will "drive" the overall decomposition).
- Next step is to collect tasks into groups (*Group Tasks*) and think about any ordering constraints (*Order Tasks*).
- Next step is to analyze how data is shared among tasks (*Data Sharing*). (Review slides from last time.)
- Finally, *Design Evaluation* . . .

Slide 3

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.")

Slide 4

Design Evaluation — Suitability for Target Platform

- How many processing elements (PEs) are available? Need at least one task per PE, often want many more — unless we can easily get exactly one task per PE at runtime, with good load balance.
- How are data structures shared among PEs? If there's a lot of shared data, or sharing is very "fine-grained", implementing for distributed memory will likely not be easy or fast.
- How many UEs are available and how do they share data? Similar to previous questions, but in terms of UEs — with some architectures, can have multiple UEs per PE, e.g., to hide latency. For this to work, "context switching" must be fast, and problem must be able to take advantage of it.
- How does time spent doing computation compare to overhead of synchronization/communication, on target platform? May be a function of problem size relative to number of PEs/UEs.

Slide 5

Design Evaluation — Design Quality

- Is it flexible? Will it adapt well to a range of platforms (if appropriate), differing numbers of UEs/PEs, different problem sizes? Does it deal gracefully with "boundary cases"?
- Is it efficient? Can you get good load balance? Is overhead minimal? consider UE creation and scheduling, communication, and synchronization.
- Is it (paraphrasing Einstein) "as simple as possible, but not simpler"? Is it reasonable to think mortals can produce working code relatively quickly? which can later be ported and/or enhanced?

Slide 6

Design Evaluation — Preparation for Next Phase

- How regular are tasks and their data dependencies?
- Are interactions between tasks (or groups of tasks) synchronous or asynchronous?
- Are tasks grouped in the best way?

Slide 7

Molecular Dynamics Example — Design Evaluation

- Major phases of computation seem to involve a lot of tasks, so we can take advantage of many processors.
- Data sharing seems more suited to shared memory than distributed memory, but the latter could work if we just duplicate data (have to think about how well that would "scale").
- Tasks and data are fairly regular, with one exception: how many neighbors an atom has might vary a lot. Probably will affect how we split up work among UEs.
- Interaction among tasks is synchronous.

Slide 8

Heat Diffusion Example — Design Evaluation

- Major phases of computation seem to involve a lot of tasks, so we can take advantage of many processors.
- Data sharing seems suitable for either shared or distributed memory.
- Tasks and data are very regular, interaction is synchronous.

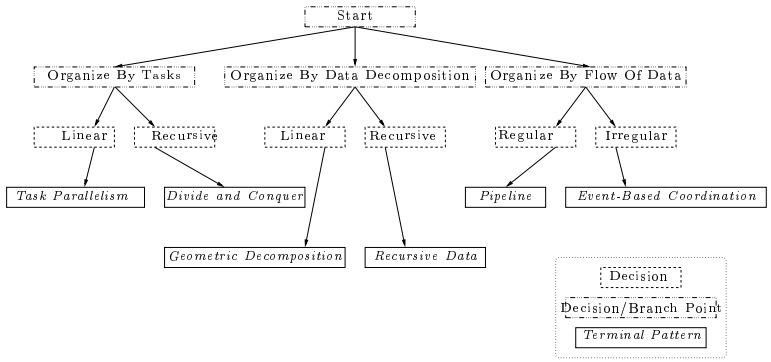
Slide 9

Algorithm Structure Design Space

- Historical note: These are the patterns with the longest history. We started out trying to identify commonly-used overall structures for parallel programs (these patterns), and then at some point added the other "design spaces".
- After much thought, writing, revision, and arguing, we came up with . . .

Slide 10

Algorithm Structure Decision Tree (Fig. 4.2)



Slide 11

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:
  - Molecular dynamics example (next slide).
  - 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 12

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 13

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 14

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 15

Minute Essay

- Does the overall strategy for the two examples make sense? Do you think you could (almost?) turn them into code?