Slide 1

Administrivia

- One-paragraph project proposals due today by 5pm. Okay to slip until tomorrow, but not much longer. Even if you've said something to me in person, please send me a short e-mail telling me what you plan to do.

Slide 2

Review — Organization of Our Pattern Language

- Four "design spaces" corresponding to phases in design:
  - *Finding Concurrency* patterns — how to decompose problems, analyze decomposition.
  - *Algorithm Structure* patterns — high-level program structures.
  - *Supporting Structure* patterns — program structures (e.g., SPMD, fork/join), data structures (e.g., shared queue).
  - *Implementation Mechanisms* — no patterns, but generic discussion of "building blocks" provided by programming environments.
- We've looked now at everything except the top level, so . . .

Slide 3

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 4

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 5

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).
- Examples — molecular dynamics (previous lecture(s)), matrix multiplication.
- Once you have this, consider data related to each task (*Data Decomposition*).

Slide 6

Data Decomposition

- 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.
- Examples — heat diffusion problem (previous lecture(s)), matrix multiplication.
- Once you have this, consider computation related to each chunk of data (*Task Decomposition*).

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 7

Order Tasks

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

Slide 8

Slide 9

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 10

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 11

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 12

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 that 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 13

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 14

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 15

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 balace? 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?

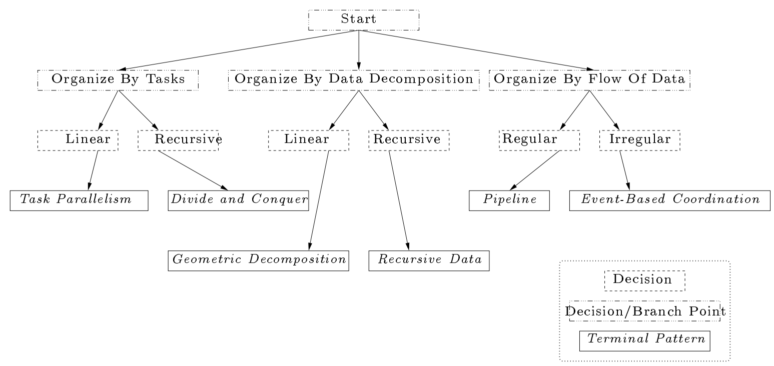
Slide 16

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?

Algorithm Structure Decision Tree, Revisited

(Figure 4.2):



Slide 17

Minute Essay

- Do you plan to be in class Thursday?
- Any requests for the next two lectures?

Slide 18