Slide 1

## Administrivia

- (None?)

Slide 2

## *Supporting Structures* Design Space, Continued

- Two categories of patterns in this space — program structure and data structure. We talked about program structure patterns last week and before.
- Now look at patterns for data structures:
  - *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).

Slide 3

## *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 4

## *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 5

## Motivating Example for *Distributed Array*

- A simple example, representative of a big class of scientific-computing applications — "heat distribution problem".
- Goal is to simulate what happens when two ends of a pipe are put in contact with things at different (constant) temperatures — pipe conducts heat, its temperature changes over time, eventually converging on a smooth gradient.
- Can model mathematically how temperature in pipe changes over time using partial differential equations.
- Can approximate solution by "discretizing" — spatially and with regard to time. Sequential code in figure 4.11 (p. 86).

Slide 6

## Motivating Example, Continued

- How to parallelize? Obvious places to look for lots of tasks are loops. Time-step loop doesn't look promising — values at each step depend on values at previous step. Loop over points seems more likely.
- Could consider each iteration as a task. Here, though, makes at least as much sense to focus on decomposing large data structures (arrays) and operating on elements concurrently.
- Dependencies among tasks / data elements: At each step, we first update `ukp1` using `uk` (where for each point we need values from neighbors too), then update `uk` using `ukp1`. All elements can be updated concurrently.
- Since main source of concurrency appears to involve updating a large data structure, with tasks that aren't completely independent — algorithm structure is *Geometric Decomposition*.

Slide 7

## *Geometric Decomposition* — Key Elements

- How to decompose data (into "chunks"): "Granularity" can affect performance.

  Straightforward for heat distribution problem,
- Consider how update of each chunk depends on other chunks; may need "exchange operation" before update.

  For heat distribution problem, need access to "boundary values" in "neighbor chunks".
- How to update data.

  Straightforward for heat distribution problem.
- How to distribute chunks among UEs: Load balance, communication can affect performance.

  Straightforward for heat distribution problem. On distributed-memory system — "distributed arrays".

Slide 8

## *Distributed Array*

- Key data structures for many scientific-computing applications are large arrays, often 2D or 3D.
- If we have lots and lots of memory shared among UEs, and time to access an element doesn't depend on UE, all is well. Usually not the case. though — obviously true for distributed-memory systems, somewhat true for NUMA systems also.
- So — typical approach is to partition array into blocks and distribute them among UEs. Idea is to do this to get:
  - Good load balance.
  - Minimum communication.
  - "Clarity of abstraction". Key idea — global indices versus local indices.

  Pictures are easy to draw; code can get messy.

Slide 9

## *Distributed Array*, Continued

- Commonly used approaches ("distributions"):
  - 1D block.
  - 2D block.
  - Block-cyclic.
- For some problems (such as heat distribution problem), makes sense to extend each "local section" with "ghost boundary" containing values needed for update.
- MPI version of heat distribution code — figures 4.14 and 4.15 (pp. 90–91).

Slide 10

## Minute Essay

- None — sign in.

Slide 11

## Minute Essay Answer

- FIX THIS