Slide 1

Administrivia

- Homework 3 on the Web.

Slide 2

Supporting Structures Patterns — Review/Recap

- Two groups of patterns:
- Patterns representing program structure (last time).
- Patterns representing frequently-used data structures (this time).

Slide 3

Supporting Structures Data Structure Patterns

- By no means not a complete list, but some examples of frequently-used ways of sharing data:
  - *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).
- Programming environment / library may provide support (e.g., Java has library class(es) for shared queues).

Slide 4

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 5

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 6

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 and understand; code can get messy.

Slide 7

Distributed Array, Continued

- Commonly used approaches (“distributions”):
  - 1D block.
  - 2D block.
  - Block-cyclic.
- For some problems (such as heat-diffusion problem), makes sense to extend each “local section” with “ghost boundary” containing values needed for update.

Slide 8

Example Application: Heat-Diffusion Problem

- We’ve talked about this problem in general terms. Now look at code . . .
- OpenMP version is fairly straightforward — parallelize two inner loops, only somewhat-tricky part is the reduction operation to compute `maxdiff`.
- MPI version is less straightforward — applying *Distributed Array* pattern to the two big arrays is straightforward in principle but in practice full of messy details.
- (Look at code.)

Slide 9

Homework 3 Background

- (Look at homework writeup briefly.)

Slide 10

Minute Essay

- None — sign in.