Slide 1

Administrivia

- (None.)

Slide 2

Supporting Structures Program Structure Patterns — Review

- We identified four basic ways parallel programs can be structured:
  - *SPMD* (Single Program, Multiple Data) ("like an MPI program").
  - *Master/Worker* (what the name suggests).
  - *Loop Parallelism* ("like an OpenMP program").
  - *Fork/Join* (what the name suggests, and also a catchall).

  If we chose the names well, you should be able to make some guesses about what the patterns represent just from the names. (Maybe not for all of these.)

Slide 3

Master/Worker — Context/Forces

- For applications where it's easy to tell how to split up the computational load to get "good load balance", previous two patterns usually work well.
- But for some applications, it's not so obvious how to do this — maybe not really possible, if work per task varies a lot and is not predictable, or if target platform includes PEs with different capabilities.

Slide 4

Master/Worker — Solution Elements

- Basic idea — one or more workers that execute tasks, master that manages things.
- "Bag of tasks" represents tasks yet to be done. Typically created by master process; often implemented as shared queue. Workers can pull elements from it directly, or can communicate with master to get new tasks.
- Typical approach shown in Fig. 5.14.

Slide 5

Master/Worker — Solution Elements, Continued

- Several potential complications:
  - All tasks may be known initially, or new ones may be generated during computation.
  - Usually computation isn't done until all tasks are done, but sometimes can stop early.
- Several variations/optimizations:
  - Master can turn into a worker after creating tasks. (Obviously more efficient if it has nothing to do.)
  - Master can be implicit, if tasks are loop iterations and dynamic scheduling of loop iterations is possible.
- Implementation normally involves, to some extent, one of the other patterns in this chapter.

Slide 6

Master/Worker — Examples and Uses

- Particularly good for *Task Parallelism* problems with completely independent tasks ("embarrassingly parallel").
- Example — MPI generic master/worker program.

Slide 7

Fork/Join — Context/Forces

- For applications where the number of concurrent tasks is more or less constant, and relationships among them are simple and regular, previous patterns usually work well.
- But for some applications, tasks are created dynamically ("forked") and later terminated ("joined" with forking task) as program runs. Sometimes you can still use one of the previous patterns, but sometimes not — if relationships among tasks are recursive (e.g., *Divide and Conquer*) or irregular, or if different tasks represent different functions (i.e., you need to do two or more different things concurrently).
- In that case, it may make more sense to create a UE for each task — potentially expensive, but easier to understand.

Slide 8

Fork/Join — Solution Elements

- Simple approach — one task per UE. As new tasks are created, a new UE is created for each; when the task finishes, the UE is destroyed. Typically the UE that created the new task/UE waits for it to finish. Simple to understand, but potentially inefficient.
- More complicated approach — pool of UEs and queue of tasks, with UEs grabbing new tasks out of the queue as they finish their old tasks. Potentially more efficient, but more complicated to program and understand.

Slide 9

Fork/Join — Examples and Uses

- Particularly good for *Divide and Conquer* and *Recursive Data* problems. One-task-per-UE version is OpenMP's standard programming model (expressed implicitly). Also matches (pre-1.5) Java's support for multithreading.

  (Curiously enough, though, most OpenMP programs really use the simpler *Loop Parallelism*.)

- Example — mergesort.

Slide 10

Supporting Structures Data Structure Patterns

- Probably 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 11

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 12

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 13

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 14

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.
- Look at some versions of code for the heat-distribution problem. (MPI code in book as Figures 4.14 and 4.15 (pp. 90–91).)

Slide 15

Minute Essay

- None — sign in.