Slide 1

Administrivia

- Grades for Homework 1 mailed — at last. I hope not so long a delay on Homework 2 grades!

Slide 2

Minute Essay From Last Lecture

- About turning in Homework 2 in pieces or all at once — some people were fine with all at once, but more would have preferred to turn things in in pieces. Maybe for the next homework?
- About what was noteworthy, people mentioned how it was interesting to compare same problem in different programming environments (part of the point!), others about this difficulty or that (OpenCL was the most troublesome), etc.

Slide 3

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.

Slide 4

Supporting Structures Design Space

- Key idea here — represent (and talk about in general terms) two classes of commonly-used things:
  - Program structures — e.g., SPMD (think "like MPI programs").
  - Frequently-used data structures — e.g., shared queue.

Slide 5

Forces

- Part of the "design pattern" idea is that a good pattern represents a good trade-off between "forces" pulling in different directions.
- For the patterns in this chapter, common set of forces:
  - Clarity of abstraction — is structure clear from code?
  - Scalability — does program "scale" well to large numbers of PEs (processing elements)?
  - Efficiency — does it make good use of resources?
  - Maintainability — can humans understand it?
  - Environmental affinity — does it work well in the likely target environment(s)?
  - Sequential equivalence — same results no matter how many processes?

Slide 6

Program Structure Patterns

- We identified four basic ways parallel programs can be structured:
  - *SPMD* (Single Program, Multiple Data).
  - *Master/Worker*.
  - *Loop Parallelism*.
  - *Fork/Join*.

  (Maybe there should be another — OpenCL?)
- 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 7

SPMD — Context/Forces

- Often makes sense, especially for large-scale parallelism, to have all UEs doing more or less the same thing, each on a different part of the overall data; easier to manage complexity this way too.

  → "Single Program, Multiple Data" paradigm. Good fit, too, with hardware for large-scale parallelism.
- But typically they don't all do *exactly* the same thing, so you need some way to have different UEs do slightly different things.

Slide 8

SPMD — Solution Elements

- All UEs execute the same (source) code: Initialize, obtain unique ID, compute, finalize.
- Based on ID, different UEs can do different things. (Typically the differences are modest — e.g., only one process prints results — but in the extreme, you get "MPMD" effect.)
- Typically, problem data includes:
  - Data structures shared by all UEs. If no shared memory, must replicate, possibly recombine at end.
  - Data structures logically distributed among UEs. Idea is to partition data in a way that matches how the computation is partitioned.

Slide 9

SPMD — Examples and Uses

- Very, very common, especially for MPI programs. Particularly good for *Task Parallelism* and *Geometric Decomposition* problems.
- Example — numerical integration:
  - Logical choice for MPI. One choice we could make, though, is how to partition data (loop iterations) among UEs — by blocks, or cyclically? Only one "shared" variable — sum being computed. Notice that in effect we replicate the variable, and recombine at the end.
  - Can do something in similar in OpenMP (SPMD-style versions of example).

Slide 10

Loop Parallelism — Context/Forces

- Programs in traditional application areas for parallel programming — science and engineering — mostly loop-based. Optimizing loops has a long history — first vectorizing, then parallelizing.
- Particularly appealing approach when a sequential program already exists, and you want to convert ("parallelize") it. Sometimes conversion can be done one loop at a time — easier to develop/test/debug.

Slide 11

Loop Parallelism — Solution Elements

- Find computationally intensive loops. (No point, for example, in spending a lot of time parallelizing initialization code.)
- Eliminate loop-carried dependencies (e.g., replicating variables so each UE has a copy).
- Parallelize loops — arrange for iterations to be distributed among UEs.
- Optimize loop "schedule" (how iterations are mapped to UEs).

Slide 12

Loop Parallelism — Examples and Uses

- Probably the second most common, especially for OpenMP programs. Particularly good for *Task Parallelism* and *Geometric Decomposition* problems.
- Example — numerical integration in OpenMP (earlier version).

Slide 13

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 14

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 15

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 16

Master/Worker — Examples and Uses

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

Slide 17

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 18

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 19

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 20

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 21

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 22

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 23

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 24

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 25

Minute Essay

- None — sign in.