Slide 1

## Administrivia

- Reminder: Homework 3 due today. (How many are done, or close? Could extend due date.)

Slide 2

## *Algorithm Structure* Patterns — Review/Recap

- Basic frameworks for parallel applications; our idea was that most applications would look like one of these.
- Four of them discussed last time; two more . . .

**Slide 3**

## *Pipeline*

- Problem statement:

  Suppose that the overall computation involves performing a calculation on many sets of data, where the calculation can be viewed in terms of data flowing through a sequence of stages. How can the potential concurrency be exploited?

- Key idea in solution — set up "assembly line" (pipeline).
- Canonical example is signal/image processing application, where you have a sequence of incoming images and want to apply same sequence of transformations to each one.

**Slide 4**

## *Event-Based Coordination*

- Problem statement:

  Suppose the application can be decomposed into groups of semi-independent tasks interacting in an irregular fashion. The interaction is determined by the flow of data between them which implies ordering constraints between the tasks. How can these tasks and their interaction be implemented so they can execute concurrently?

- Key idea in solution — structure computation in terms of semi-independent entities, interacting via "events".
- Canonical example is discrete event simulation — simulating many semi-independent entities that interact in irregular/unpredictable ways.

**Slide 5**

## 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 6**

## *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 7

## Forces

- Part of the "design pattern" idea: 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 8

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

Slide 9

## *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 10

## *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 11

## *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. Note 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 12

## *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 13**

## *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 14**

## *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 15

## *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 16

## *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 17

## *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 18

## *Master/Worker* — Examples and Uses

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

Slide 19

## *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 20

## *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 21

## *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 22

## *Supporting Structures* Data Structure Patterns

- (Next time.)

**Slide 23**

## Minute Essay

- How scalable are *Pipeline* and *Event-Based Coordination*? if not very, can you think of a way to fix that?

**Slide 24**

## Minute Essay Answer

- Neither pattern is very scalable, since they're based on a task decomposition that has one task per pipeline stage or one task per entity. Sometimes additional concurrency can be exposed by further decomposing these stages or entities.