Outline

• Parallel programming
  – Start from scratch
  – Reengineering for parallelism

• Parallelizing a program
  – Decomposition (finding concurrency)
  – Assignment (algorithm structure)
  – Orchestration (supporting structures)
  – Mapping (implementation mechanisms)

• Patterns for Parallel Programming
Credits

• Most of the slides courtesy Dr. Rodric Rabbah (IBM)
  – Taken from 6.189 IAP taught at MIT in 2007.
Parallel programming from scratch

• Start with an algorithm
  – Formal representation of problem solution
  – **Sequence** of steps

• Make sure there is parallelism
  – In each algorithm step
  – Minimize synchronization points

• Don’t forget locality
  – Communication is costly
    • Performance, Energy, System cost

• More often start with existing sequential code
Reengineering for Parallelism

• Define a testing protocol

• Identify program hot spots: where is most of the time spent?
  – Look at code
  – Use profiling tools

• Parallelization
  – Start with hot spots first
  – Make sequences of small changes, each followed by testing
  – Patterns provide guidance
4 Common Steps to Creating a Parallel Program

Sequential Computation \rightarrow \text{Partitioning} \rightarrow \text{Assignment} \rightarrow \text{Orchestration} \rightarrow \text{Mapping} \rightarrow \text{Processors}

- **Decomposition**
- **Tasks**
- **Units of Execution**
- **Parallel Program**
- **Processors**
Decomposition

• Identify concurrency and decide at what level to exploit it

• Break up computation into tasks to be divided among processes
  – Tasks may become available dynamically
  – Number of tasks may vary with time

• Enough tasks to keep processors busy
  – Number of tasks available at a time is upper bound on achievable speedup

Main consideration: coverage and Amdahl’s Law
Coverage

• **Amdahl's Law**: The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
  
  – Demonstration of the law of diminishing returns
Amdahl’s Law

- Potential program speedup is defined by the fraction of code that can be parallelized.

Use 5 processors for parallel work:

25 seconds sequential
+ 10 seconds parallel
+ 25 seconds sequential
+ 25 seconds sequential
= 60 seconds
Amdahl’s Law

- Speedup = old running time / new running time
  = 100 seconds / 60 seconds
  = 1.67
  (parallel version is 1.67 times faster)
Amdahl’s Law

- $p = \text{fraction of work that can be parallelized}$
- $n = \text{the number of processor}$

\[
\text{speedup} = \frac{\text{old running time}}{\text{new running time}} = \frac{1}{(1-p) + \frac{p}{n}}
\]

- $\text{fraction of time to complete sequential work}$
- $\text{fraction of time to complete parallel work}$
Implications of Amdahl’s Law

- Speedup tends to tend to infinity as number of processors tends to infinity

\[
\frac{1}{1 - p}
\]

Super linear speedups are possible due to registers and caches

Typical speedup is less than linear

Parallelism only worthwhile when it dominates execution
Assignment

- Specify mechanism to divide work among PEs
  - Balance work and reduce communication

- Structured approaches usually work well
  - Code inspection or understanding of application
  - Well-known design patterns

- As programmers, we worry about partitioning first
  - Independent of architecture or programming model?
  - Complexity often affects decisions
  - Architectural model affects decisions

Main considerations: granularity and locality
Fine vs. Coarse Granularity

- **Fine-grain Parallelism**
  - Low computation to communication ratio
  - Small amounts of computational work between communication stages
  - High communication overhead
    - Potential HW assist

- **Coarse-grain Parallelism**
  - High computation to communication ratio
  - Large amounts of computational work between communication events
  - Harder to load balance efficiently
Load Balancing vs. Synchronization

Fine

Coarse

PE₀  PE₁

PE₀  PE₁
Load Balancing vs. Synchronization

Expensive sync $\rightarrow$ coarse granularity
Few units of exec + time disparity $\rightarrow$ fine granularity
Orchestration and Mapping

• Computation and communication concurrency
• Preserve locality of data
• Schedule tasks to satisfy dependences early
• Survey available mechanisms on target system

Main considerations: locality, parallelism, mechanisms (efficiency and dangers)
Parallel Programming by Pattern

• Provides a cookbook to systematically guide programmers
  – Decompose, Assign, Orchestrate, Map
  – Can lead to high quality solutions in some domains

• Provide common vocabulary to the programming community
  – Each pattern has a name, providing a vocabulary for discussing solutions

• Helps with software reusability, malleability, and modularity
  – Written in prescribed format to allow the reader to quickly understand the solution and its context

• Otherwise, too difficult for programmers, and software will not fully exploit parallel hardware