EE382N: Computer Architecture
Parallelism and Locality
Fall 2009

Lecture 8 - Parallelism in Software
(Patterns for Parallel Programming)

Mattan Erez

The University of Texas at Austin
Announcements

• I won’t be able to teach next Monday
• Option 1: Derek Chiou will give a lecture on dataflow architectures
• Option 2: Re-schedule class to later in the week. Maybe Thursday evening or Friday during the day

• I’ll post a survey
Credits

• Most of the slides courtesy Dr. Rodric Rabbah (IBM)
  - Taken from 6.189 IAP taught at MIT in 2007.
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
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
4 Common Steps to Creating a Parallel Program

Partitioning

Sequential Computation → Tasks → Units of Execution → Parallel program → Processors

- Decomposition
- Assignment
- Orchestration
- Mapping
Reengineering for Parallelism

- Parallel programs often start as sequential programs
  - Easier to write and debug
  - Legacy codes

- How to reengineer a sequential program for parallelism:
  - Survey the landscape
  - Pattern provides a list of questions to help assess existing code
  - Many are the same as in any reengineering project
  - Is program numerically well-behaved?

- Define the scope and get user's acceptance
  - Required precision of results
  - Input range
  - Performance expectations
  - Feasibility (back of envelope calculations)
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
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

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>Code Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>sequential</td>
</tr>
<tr>
<td>+</td>
<td>parallel</td>
</tr>
<tr>
<td>+ 10</td>
<td>sequential</td>
</tr>
<tr>
<td>+ 25</td>
<td>sequential</td>
</tr>
</tbody>
</table>

Result: 60 seconds
Amdahl’s Law

- Speedup = \frac{\text{old running time}}{\text{new running time}}

= \frac{100 \text{ seconds}}{60 \text{ seconds}}

= 1.67

(parallel version is 1.67 times faster)
Amdahl’s Law

- $p =$ fraction of work that can be parallelized
- $n =$ the number of processors

\[
\text{speedup} = \frac{\text{old running time}}{\text{new running time}}
\]

\[
= \frac{1}{(1 - p) + \frac{p}{n}}
\]

- fraction of time to complete sequential work
- fraction of time to complete parallel work
Implications of Amdahl’s Law

• Speedup tends to \( \frac{1}{1-p} \) as number of processors tends to infinity

Super linear speedups are possible due to registers and caches

Linear speedup (100% efficiency)

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 \text{coarse granularity}

Few units of exec + time disparity \rightarrow \text{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
History

• Berkeley architecture professor Christopher Alexander

• In 1977, patterns for city planning, landscaping, and architecture in an attempt to capture principles for “living” design
Example 167 (p. 783): 6ft Balcony

Therefore:

Whenever you build a balcony, a porch, a gallery, or a terrace always make it at least six feet deep. If possible, recess at least a part of it into the building so that it is not cantilevered out and separated from the building by a simple line, and enclose it partially.

six feet deep
Patterns in Object-Oriented Programming

• Design Patterns: Elements of Reusable Object-Oriented Software (1995)
  - Gang of Four (GOF): Gamma, Helm, Johnson, Vlissides
  - Catalogue of patterns
  - Creation, structural, behavioral
Patterns for Parallelizing Programs

4 Design Spaces

Algorithm Expression

- Finding Concurrency
  - Expose concurrent tasks

- Algorithm Structure
  - Map tasks to processes to exploit parallel architecture

Software Construction

- Supporting Structures
  - Code and data structuring patterns

- Implementation Mechanisms
  - Low level mechanisms used to write parallel programs

Here’s my algorithm.
Where’s the concurrency?
Here’s my algorithm. Where’s the concurrency?

- **Task decomposition**
  - Independent coarse-grained computation
  - Inherent to algorithm

- **Sequence of statements (instructions) that operate together as a group**
  - Corresponds to some logical part of program
  - Usually follows from the way programmer thinks about a problem
Here's my algorithm.
Where's the concurrency?

- **Task decomposition**
  - Parallelism in the application

- **Pipeline task decomposition**
  - Data assembly lines
  - Producer-consumer chains
Here’s my algorithm. Where’s the concurrency?

- **Task decomposition**
  - Parallelism in the application

- **Pipeline task decomposition**
  - Data assembly lines
  - Producer-consumer chains

- **Data decomposition**
  - Same computation is applied to small data chunks derived from large data set
Guidelines for Task Decomposition

• Algorithms start with a good understanding of the problem being solved

• Programs often naturally decompose into tasks
  - Two common decompositions are
    • Function calls and
    • Distinct loop iterations

• Easier to start with many tasks and later fuse them, rather than too few tasks and later try to split them
Guidelines for Task Decomposition

• Flexibility
  - Program design should afford flexibility in the number and size of tasks generated
    • Tasks should not be tied to a specific architecture
    • Fixed tasks vs. Parameterized tasks

• Efficiency
  - Tasks should have enough work to amortize the cost of creating and managing them
  - Tasks should be sufficiently independent so that managing dependencies doesn’t become the bottleneck

• Simplicity
  - The code has to remain readable and easy to understand, and debug
Case for Pipeline Decomposition

• Data is flowing through a sequence of stages
  – Assembly line is a good analogy

• What’s a prime example of pipeline decomposition in computer architecture?
  – Instruction pipeline in modern CPUs

• What’s an example pipeline you may use in your UNIX shell?
  – Pipes in UNIX: cat foobar.c | grep bar | wc

• Other examples
  – Signal processing
  – Graphics
Guidelines for Data Decomposition

• Data decomposition is often implied by task decomposition

• Programmers need to address task and data decomposition to create a parallel program
  – Which decomposition to start with?

• Data decomposition is a good starting point when
  – Main computation is organized around manipulation of a large data structure
  – Similar operations are applied to different parts of the data structure