Lecture 14 – Parallelism in Software I

Mattan Erez

The University of Texas at Austin
Credits

• Most of the slides courtesy Dr. Rodric Rabbah (IBM)
  – Taken from 6.189 IAP taught at MIT in 2007
• Parallel Scan slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  – Taken from EE493-AI taught at UIUC in Spring 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 → Tasks → Units of Execution → Parallel Program → Processors

Partitioning
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.
Amdahl's Law

- Speedup = \( \frac{\text{old running time}}{\text{new running time}} \)
  
  \[
  = \frac{100 \text{ seconds}}{60 \text{ seconds}}
  = 1.67
  
  (\text{parallel version is 1.67 times faster})
  
Use 5 processors for parallel work

- 25 seconds sequential
- 50 seconds parallel
- 25 seconds sequential

Total time: 100 seconds

- 25 seconds sequential
- 10 seconds parallel
- 25 seconds sequential
- 25 seconds sequential

Total time: 60 seconds
Amdahl’s Law

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

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

- \((1-p)\): fraction of time to complete sequential work
- \(\frac{p}{n}\): 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.

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

Fine

Coarse

Expensive sync → coarse granularity
Few units of exec + time disparity → 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.
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?

MPEG Decoder

MPEG bit stream

VLD

macroblocks, motion vectors

split

- frequency encoded macroblocks
- differently coded motion vectors

ZigZag

- IQuantization
- IDCT
- Saturation

- motion vectors
- spatially encoded macroblocks

join

Motion Compensation

- recovered picture

Picture Reorder

Color Conversion

Display
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

MPEG Decoder

- VLD: macroblocks, motion vectors
- Split: frequency encoded macroblocks, differentially coded motion vectors
- ZigZag
- IQuantization
- IDCT
- Saturation
- Join: spatially encoded macroblocks, motion vectors
- Motion Vector Decode
- Repeat
- Motion Compensation
- Recovered picture
- Picture Reorder
- Color Conversion
- Display
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 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
Common Data Decompositions

• Geometric data structures
  – Decomposition of arrays along rows, columns, blocks
  – Decomposition of meshes into domains
Common Data Decompositions

• Geometric data structures
  – Decomposition of arrays along rows, columns, blocks
  – Decomposition of meshes into domains

• Recursive data structures
  – Example: decomposition of trees into sub-trees
Guidelines for Data Decomposition

• Flexibility
  – Size and number of data chunks should support a wide range of executions

• Efficiency
  – Data chunks should generate comparable amounts of work (for load balancing)

• Simplicity
  – Complex data compositions can get difficult to manage and debug
Data Decomposition Examples

• Molecular dynamics
  – Compute forces
  – Update accelerations and velocities
  – Update positions

• Decomposition
  – Baseline algorithm is $N^2$
    • All-to-all communication
  – Best decomposition is to treat mols. as a set
  – Some advantages to geometric discussed in future lecture
Data Decomposition Examples

• Molecular dynamics
  – Geometric decomposition

• Merge sort
  – Recursive decomposition