Outline

• Parallelism in SW
  – ILP/DLP/TLP?

• 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
ILP/ DLP/ TLP in Software

• Does software also have ILP, DLP, and TLP?
TLP or DLP?
Converting Between ILP, TLP, and DLP?

- HW finally determines what parallelism mechanisms were used.

- Easy: DLP → TLP → ILP

- Harder/inefficient: ILP → TLP → DLP
  - Requires significant analysis
  - Often need to speculate
Converting Between ILP, TLP, and DLP

• Examples for conversion:

• SW:

• HW:
Outline

• Parallelism in SW
  – ILP/DLP/TLP?

• 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
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 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.
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 processor

$$speedup = \frac{\text{old running time}}{\text{new running time}}$$

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

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.

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
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

Outline

- Parallelism in SW
  - ILP/DLP/TLP?
- 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
Here’s my algorithm.
Where’s the concurrency?

MPEG Decoder

MPEG bit stream

VLD
macroblocks, motion vectors

split

frequency encoded macroblocks
differentially coded motion vectors

ZigZag

I Quantization

IDCT

Saturation

Motion Vector Decode

Repeat

motion vectors

join

spatially encoded macroblocks

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
  - differently coded motion vectors
- **ZigZag**
- **IQuantization**
- **IDCT**
- **Saturation**
- **Motion Vector Decode**
  - **Repeat**
- **Motion Compensation**
- **Picture Reorder**
- **Color Conversion**
- **Display**

**MPEG bit stream**
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
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
Data Decomposition Examples

- **Molecular dynamics**
  - Geometric decomposition

- **Merge sort**
  - Recursive decomposition
Dependence Analysis

• Given two tasks how to determine if they can safely run in parallel?
Bernstein’s Condition

- $R_i$: set of memory locations read (input) by task $T_i$
- $W_j$: set of memory locations written (output) by task $T_j$

- Two tasks $T_1$ and $T_2$ are parallel if
  - input to $T_1$ is not part of output from $T_2$
  - input to $T_2$ is not part of output from $T_1$
  - outputs from $T_1$ and $T_2$ do not overlap
Example

\[ \begin{align*}
T_1 & \quad a = x + y \\
T_2 & \quad b = x + z \\
\end{align*} \]

- \( R_1 = \{ x, y \} \)
- \( W_1 = \{ a \} \)
- \( R_2 = \{ x, z \} \)
- \( W_2 = \{ b \} \)

\[ \begin{align*}
R_1 \cap W_2 &= \emptyset \\
R_2 \cap W_1 &= \emptyset \\
W_1 \cap W_2 &= \emptyset
\end{align*} \]
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

Patterns for Parallel Programming. Mattson, Sanders, and Massingill (2005)
Algorithm Structure Design Space

• Given a collection of concurrent tasks, what’s the next step?
• Map tasks to units of execution (e.g., threads)

• Important considerations
  – Magnitude of number of execution units platform will support
  – Cost of sharing information among execution units
  – Avoid tendency to over constrain the implementation
    • Work well on the intended platform
    • Flexible enough to easily adapt to different architectures
Major Organizing Principle

• How to determine the algorithm structure that represents the mapping of tasks to units of execution?

• Concurrency usually implies major organizing principle
  - Organize by tasks
  - Organize by data decomposition
  - Organize by flow of data
Organize by Tasks?

Recursive?

yes → Divide and Conquer

no → Task Parallelism
Task Parallelism

- Molecular dynamics
  - Non-bonded force calculations, some dependencies

- Common factors
  - Tasks are associated with iterations of a loop
  - Tasks largely known at the start of the computation
  - All tasks may not need to complete to arrive at a solution
Divide and Conquer

- For recursive programs: divide and conquer
  - Subproblems may not be uniform
  - May require dynamic load balancing
Organize by Data?

- Operations on a central data structure
  - Arrays and linear data structures
  - Recursive data structures
Recursive Data

• Computation on a list, tree, or graph
  – Often appears the only way to solve a problem is to sequentially move through the data structure

• There are however opportunities to reshape the operations in a way that exposes concurrency
Recursive Data Example: Find the Root

- Given a forest of rooted directed trees, for each node, find the root of the tree containing the node
  - Parallel approach: for each node, find its successor’s successor, repeat until no changes
- $O(\log n)$ vs. $O(n)$
Work vs. Concurrency Tradeoff

- Parallel restructuring of find the root algorithm leads to $O(n \log n)$ work vs. $O(n)$ with sequential approach.

- Most strategies based on this pattern similarly trade off increase in total work for decrease in execution time due to concurrency.
Organize by Flow of Data?

- In some application domains, the flow of data imposes ordering on the tasks
  - Regular, one-way, mostly stable data flow
  - Irregular, dynamic, or unpredictable data flow
Pipeline Throughput vs. Latency

• Amount of concurrency in a pipeline is limited by the number of stages

• Works best if the time to fill and drain the pipeline is small compared to overall running time

• Performance metric is usually the throughput
  – Rate at which data appear at the end of the pipeline per time unit (e.g., frames per second)

• Pipeline latency is important for real-time applications
  – Time interval from data input to pipeline, to data output
Event-Based Coordination

- In this pattern, interaction of tasks to process data can vary over unpredictable intervals.

- Deadlocks are a danger for applications that use this pattern:
  - Dynamic scheduling has overhead and may be inefficient.
  - Granularity a major concern.