Credits

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

**Diagram:**
- MPEG bit stream → VLD → macroblocks, motion vectors → split → ZigZag → IQuantization → IDCT → Saturation → frequency encoded macroblocks → differentially encoded motion vectors → Join → motion vectors → Motion Vector Decode → Repeat → Motion Compensation → recovered picture → Picture Reorder → Color Conversion → Display

- **Tasks:**
  - VLD
  - Split
  - Motion Vector Decode
  - Motion Compensation
  - Picture Reorder
  - Color Conversion
  - Display

- **Decomposition:**
  - Task decomposition
  - Pipeline task decomposition
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

\[ T_1 \]
\[ a = x + y \]

\[ T_2 \]
\[ b = x + z \]

\[ R_1 = \{ x, y \} \]
\[ W_1 = \{ a \} \]

\[ R_2 = \{ x, z \} \]
\[ W_2 = \{ b \} \]

\[ R_1 \cap W_2 = \emptyset \]
\[ R_2 \cap W_1 = \emptyset \]
\[ W_1 \cap W_2 = \emptyset \]
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

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?

yes

Recursive Data

no

Geometric Decomposition
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. $O(n)$

```
Recursive Data Example: Find the Root

\begin{itemize}
  \item Given a forest of rooted directed trees, for each node, find the root of the tree containing the node
  \begin{itemize}
    \item Parallel approach: for each node, find its successor’s successor, repeat until no changes
    \item $O(\log n)$ vs. $O(n)$
  \end{itemize}
\end{itemize}
```
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