Lecture 6

Design Patterns for Parallel Programming I
4 Common Steps to Creating a Parallel Program
Decomposition (Amdahl’s Law)

- 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
Assignment (Granularity)

- Specify mechanism to divide work among core
  - 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
  - But complexity often affect decisions!
Orchestration and Mapping (Locality)

- Computation and communication concurrency
- Preserve locality of data
- Schedule tasks to satisfy dependences early
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

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

MPEG Decoder

- MPEG bit stream
  - VLD
    - macroblocks, motion vectors
      - split
        - ZigZag
        - frequency encoded macroblocks
        - IQuantization
        - motion vectors
          - Motion Vector Decode
            - Repeat
              - motion vectors
                - Spatially encoded macroblocks
                  - join
                    - motion vectors
                      - motion vectors
                        - IDCT
                          - Saturation
                            - Spatially encoded macroblocks
                              - join
                                - recovered picture
                                  - Motion Compensation
                                    - Motion Compensation
                                      - 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
- Data decomposition
  - Same computation is applied to small data chunks derived from large data set
Here’s my algorithm. Where’s the concurrency?

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

- **Data decomposition**
  - Same computation many data

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

- Array data structures
  - Decomposition of arrays along rows, columns, blocks

- 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
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
Re-engineering for Parallelism
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 users 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
  - Pattern provides guidance
Example: Molecular dynamics

● Simulate motion in large molecular system
  ■ Used for example to understand drug-protein interactions

● Forces
  ■ Bonded forces within a molecule
  ■ Long-range forces between atoms

● Naïve algorithm has $n^2$ interactions: not feasible

● Use cutoff method: only consider forces from neighbors that are “close enough”
// pseudo code
real[3,n] atoms
real[3,n] force
int [2,m] neighbors

function simulate(steps)
    for time = 1 to steps and for each atom
        Compute bonded forces
        Compute neighbors
        Compute long-range forces
        Update position
    end loop
end function
Finding Concurrency Design Space

- Decomposition Patterns
- Dependency Analysis Patterns
- Design Evaluation
Decomposition Patterns

- Main computation is a loop over atoms
- Suggests task decomposition
  - Task corresponds to a loop iteration
    - Update a single atom
  - Additional tasks
    - Calculate bonded forces
    - Calculate long range forces
  - Find neighbors
  - Update position
- There is data shared between the tasks

```python
for time = 1 to steps and
  for each atom
    Compute bonded forces
    Compute neighbors
    Compute long-range forces
    Update position
end loop
```
Understand Control Dependences

- Bonded forces
- Neighbor list
- Long-range forces

Update position

next time step
Understand Data Dependences

- Bonded forces
- Neighbor list
- Long-range forces
- Update position

Next time step

Atoms $[3,n]$
Forces $[2,n]$
Neighbors $[2,m]$

Read
Write
Accumulate
Evaluate Design

● What is the target architecture?
  ■ Shared memory, distributed memory, message passing, …

● Does data sharing have enough special properties (read only, accumulate, temporal constraints) that we can deal with dependences efficiently?

● If design seems OK, move to next design space