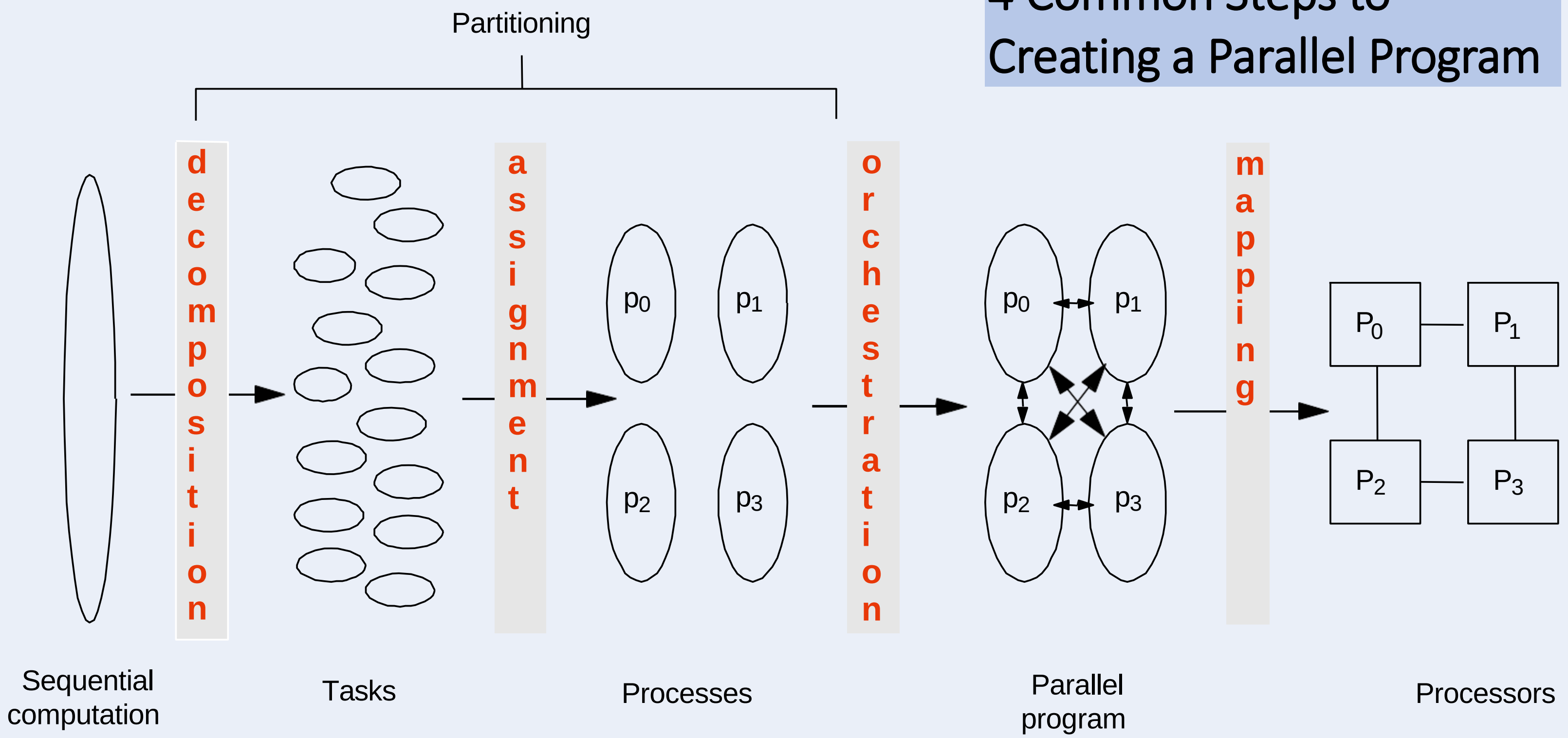
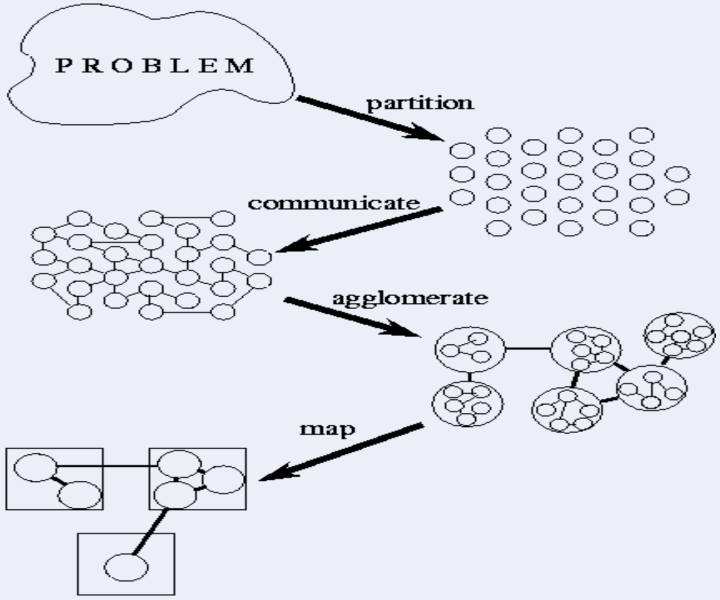
Lecture 12

Design for Parallel Programming

4 Common Steps to Creating a Parallel Program



4 steps - Ian Foster (1995)
Designing and Building Parallel Programs



Decomposition/ Partition

- 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

Agglomeration (Granularity)

- Balance work and reduce communication

- Structured approaches
  - Well-known design patterns

- Partitioning first
  - Independent of architecture or programming model

Orchestration and Mapping (Locality)

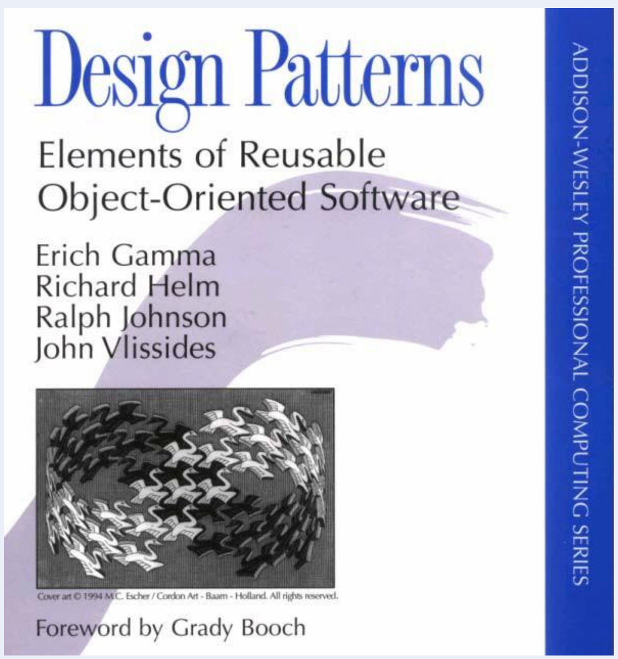
- Computation and communication concurrency
- Preserve locality of data
- Schedule tasks to satisfy dependences early

Parallel Patterns

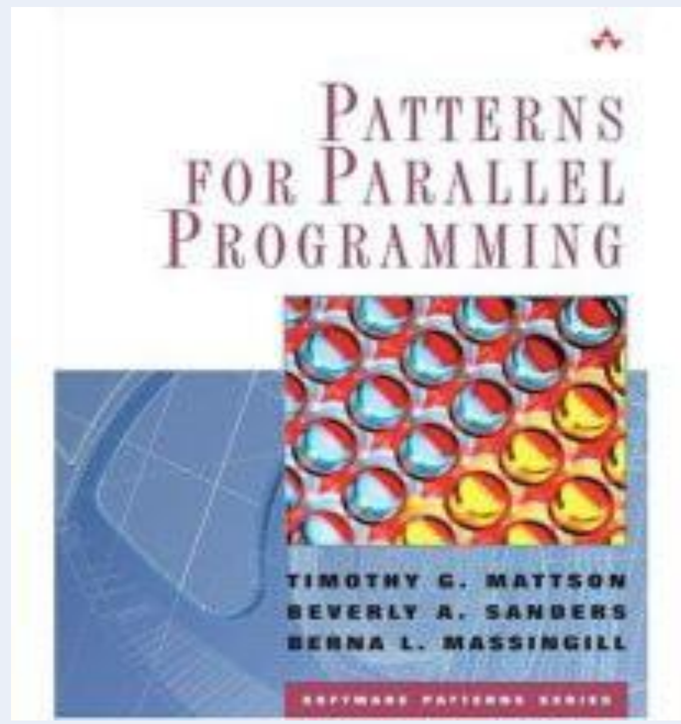
- Provides a recipes to systematically guide programmers
  - Can lead to high quality solutions in some domains
- Provide common vocabulary to the programming community
  - Each pattern has a name, providing a vocabulary for discussing solutions
- Helps with software reusability, malleability, and modularity
  - 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

Patterns in Programming

- *Design Patterns: Elements of Reusable Object-Oriented Software*
- Gamma, Helm, Johnson, Vlissides (1995)
  - Gang of Four (GOF)



- *Patterns for Parallel Programming*
- Mattson, Sanders, and Massingill
(2005)



Patterns for Parallelizing Programs

Design Space

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

Guidelines for Task Decomposition

- Algorithms start with a good understanding of the problem being solved
- Programs often naturally decompose into tasks
  - common decompositions depends on:
    - data
    - functions
- 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

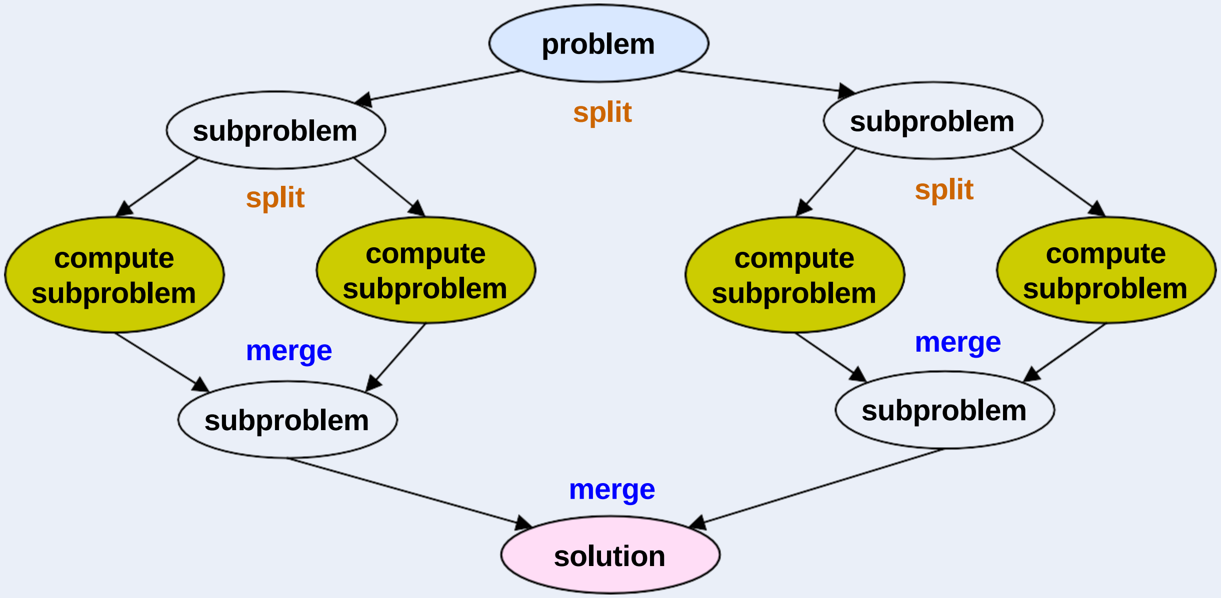
- Flexibility
  - Program design should afford flexibility in the number and size of tasks generated
    - Tasks should not tied to a specific architecture
    - Fixed tasks vs. Parameterized tasks
- Efficiency
  - **Tasks should have enough work to amortize the cost of creating and managing them**
  - 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

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/ Refactorization 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

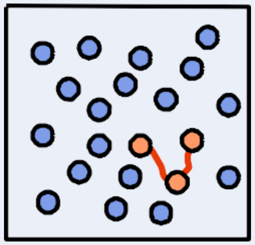
- Define a testing protocol

- 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"

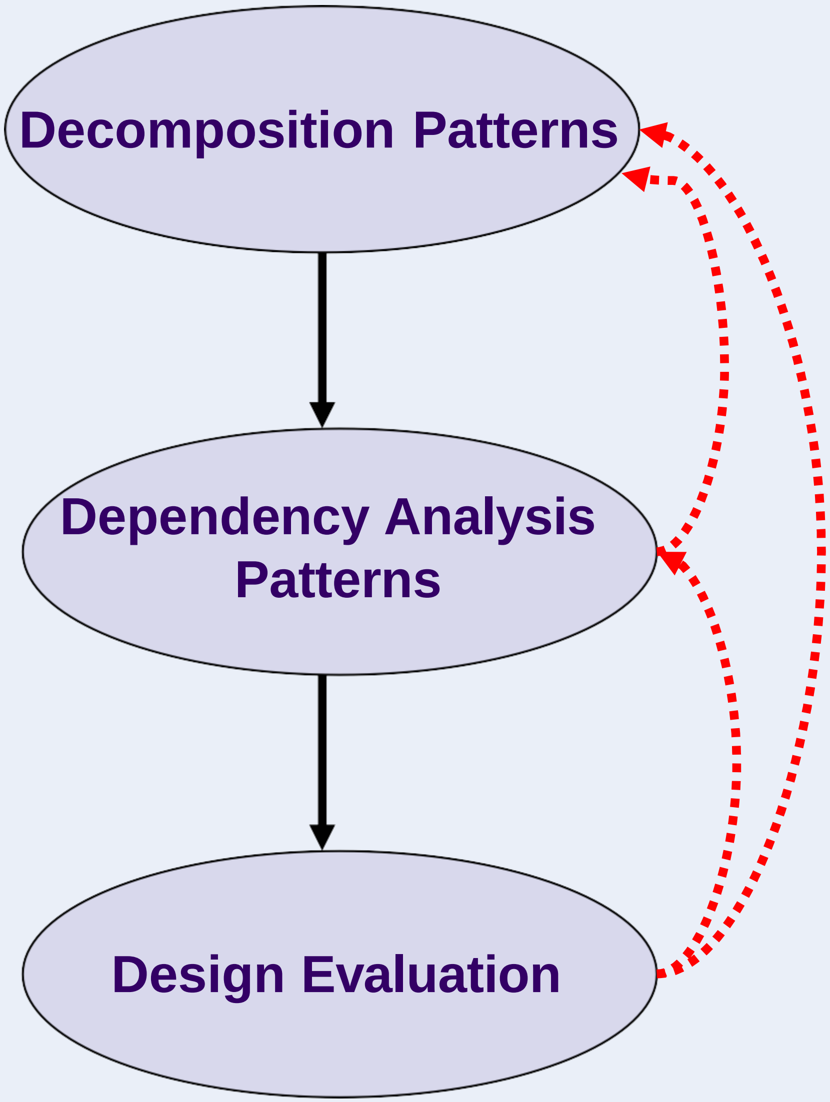


Sequential Molecular Dynamics Simulator

```
// 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

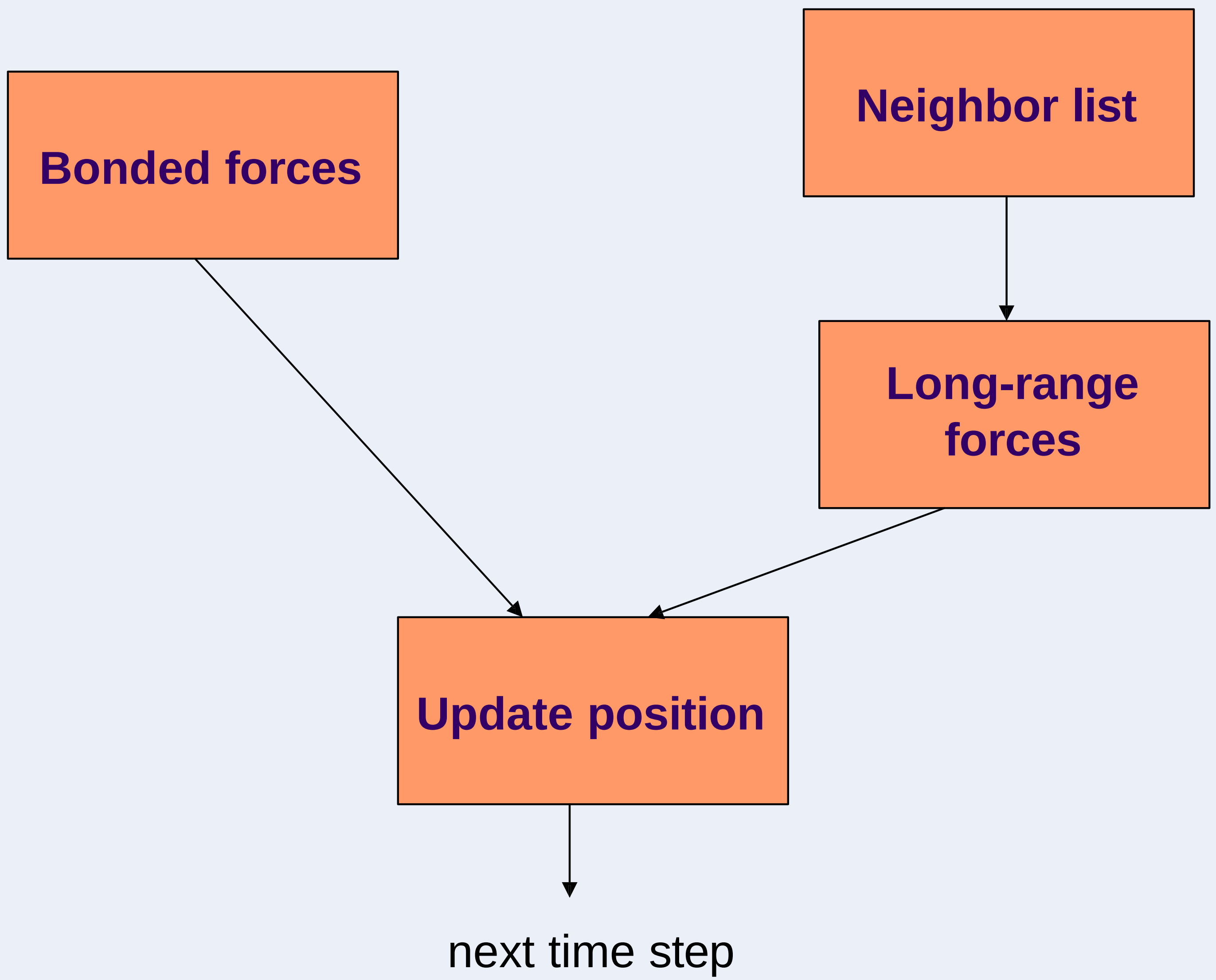
- Main computation is a loop over atoms
- Suggests task decomposition
  - Task corresponds to a loop iteration
    - Update a single atom
  - Additional tasks
    - Calculate bonded forces
    - Calculate long range forces
  - Find neighbors
  - Update position
- There is data shared between the tasks

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

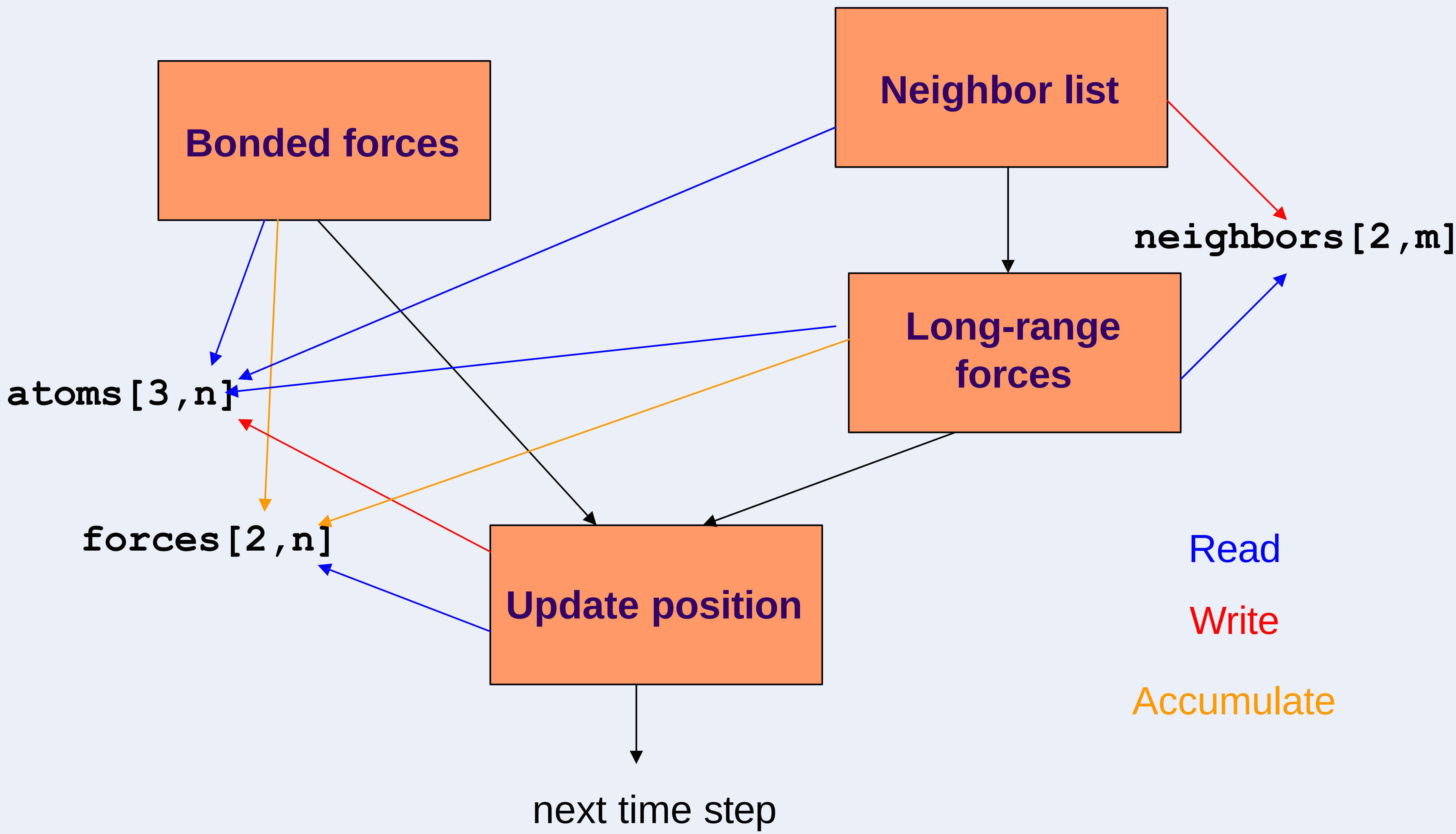
and

Task Dependency Graph

Understand Control Dependences



Understand Data Dependences



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

Patterns for Parallelizing Programs

4 Design Spaces

Algorithm Expression

- Finding Concurrency
  - Expose concurrent tasks
- Algorithm Structure
  - Map tasks to units of execution 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

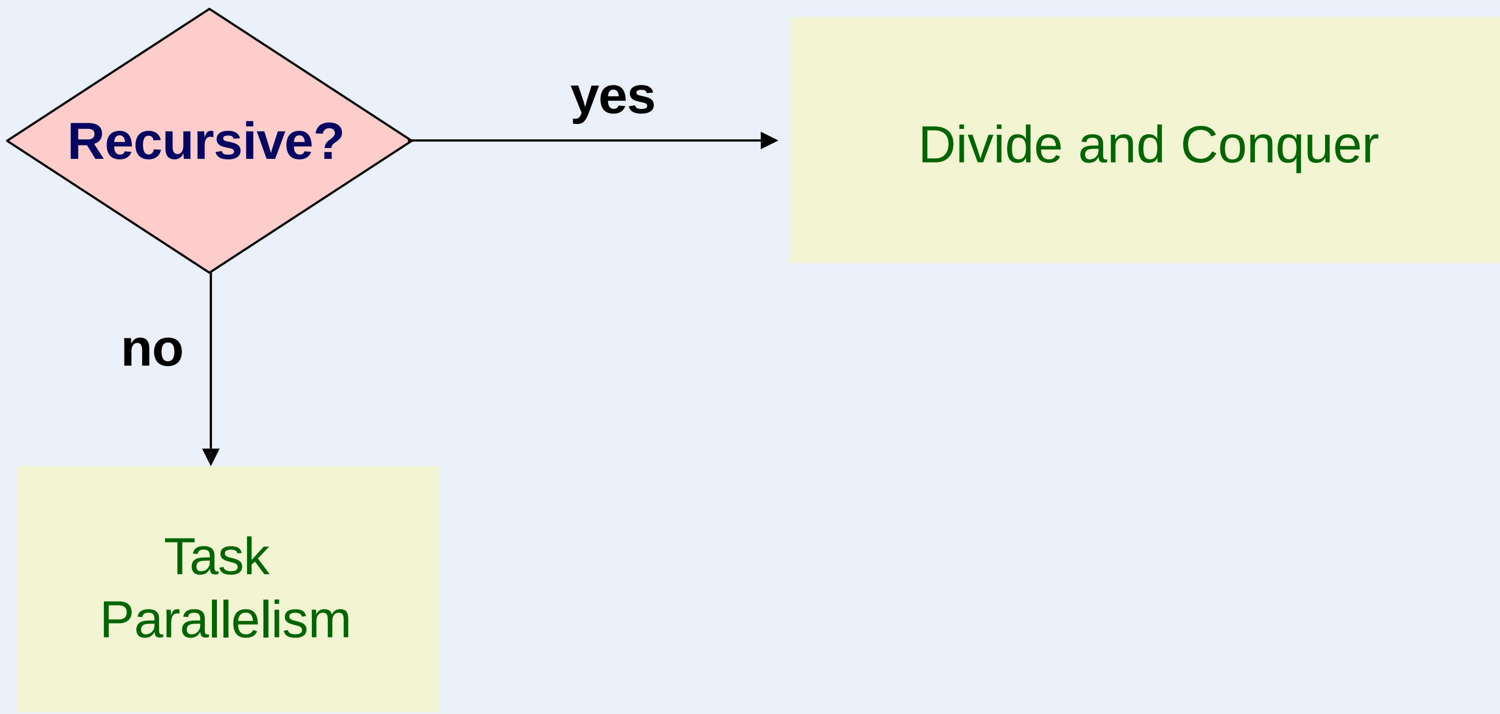
- Given a collection of concurrent tasks, what’s the next step?
- Map tasks to units of execution (e.g., threads)

- 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

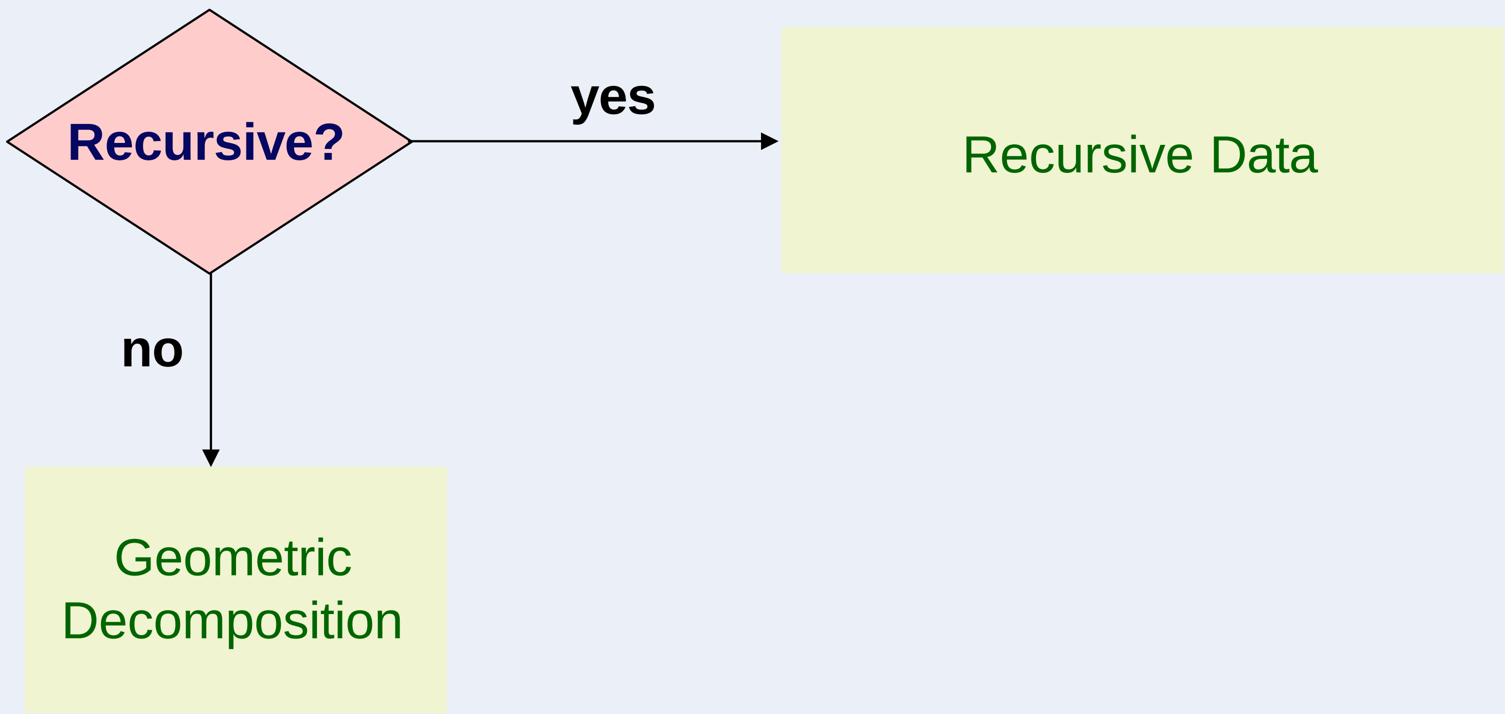
- How to determine the algorithm structure that represents the mapping of tasks to units of execution?
- Concurrency usually implies major organizing principle
  - Organize by tasks
  - Organize by data decomposition
  - Organize by flow of data

Organize by Tasks?



Organize by Data?

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



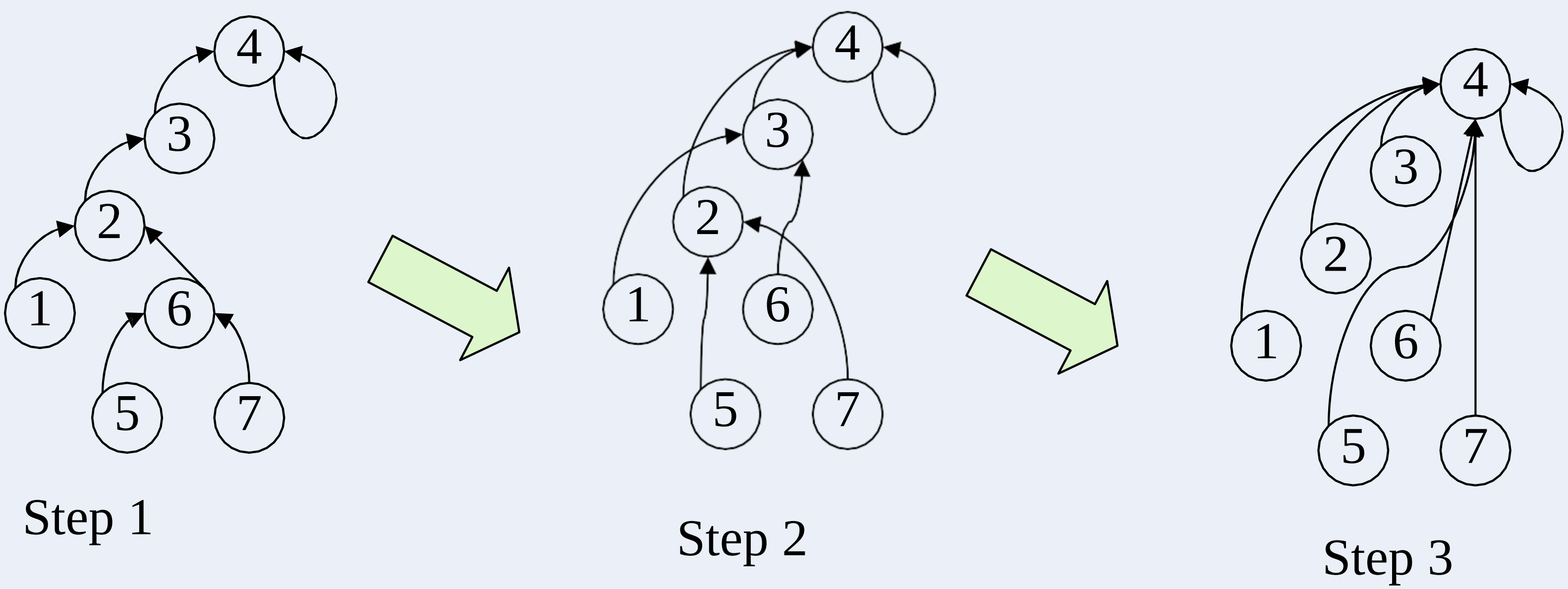
Recursive Data

- 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

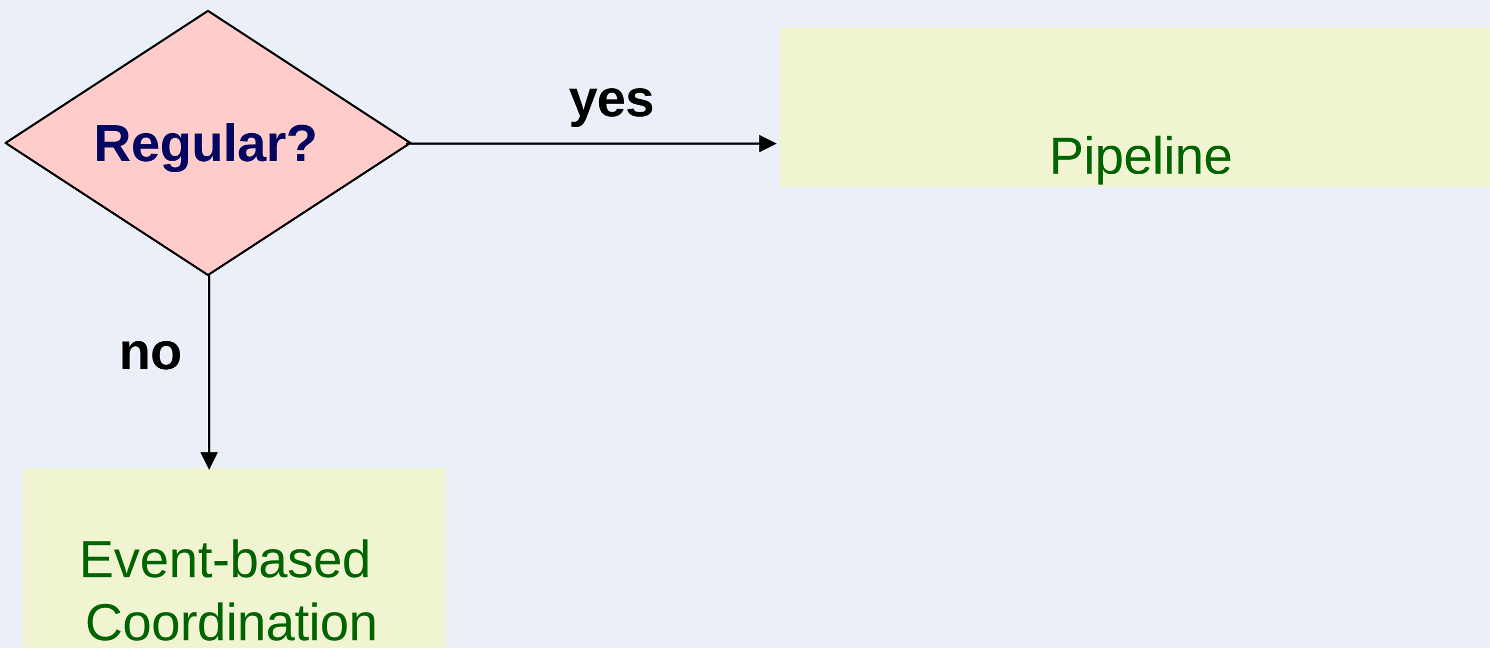
Recursive Data Example: Find the Root

- 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)



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 likely for applications that use this pattern

Supporting Structures

- SPMD
- Loop parallelism
- Master/Worker
- Fork/Join

SPMD Pattern

- Single Program Multiple Data: create a single source-code image that runs on each processor
  - Initialize
  - Obtain a unique identifier
  - Run the same program each processor
    - Identifier and input data differentiate behavior
  - Distribute data
  - Finalize

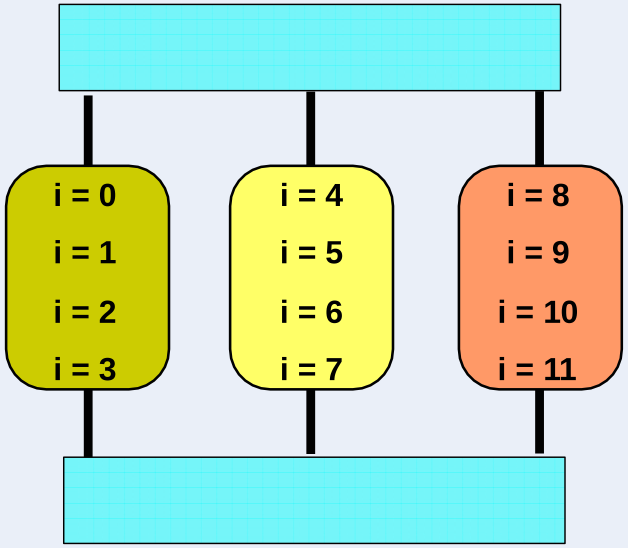
SPMD Challenges

- Split data correctly
- Correctly combine the results
- Achieve an even distribution of the work
- For programs that need dynamic load balancing, an alternative pattern is more suitable

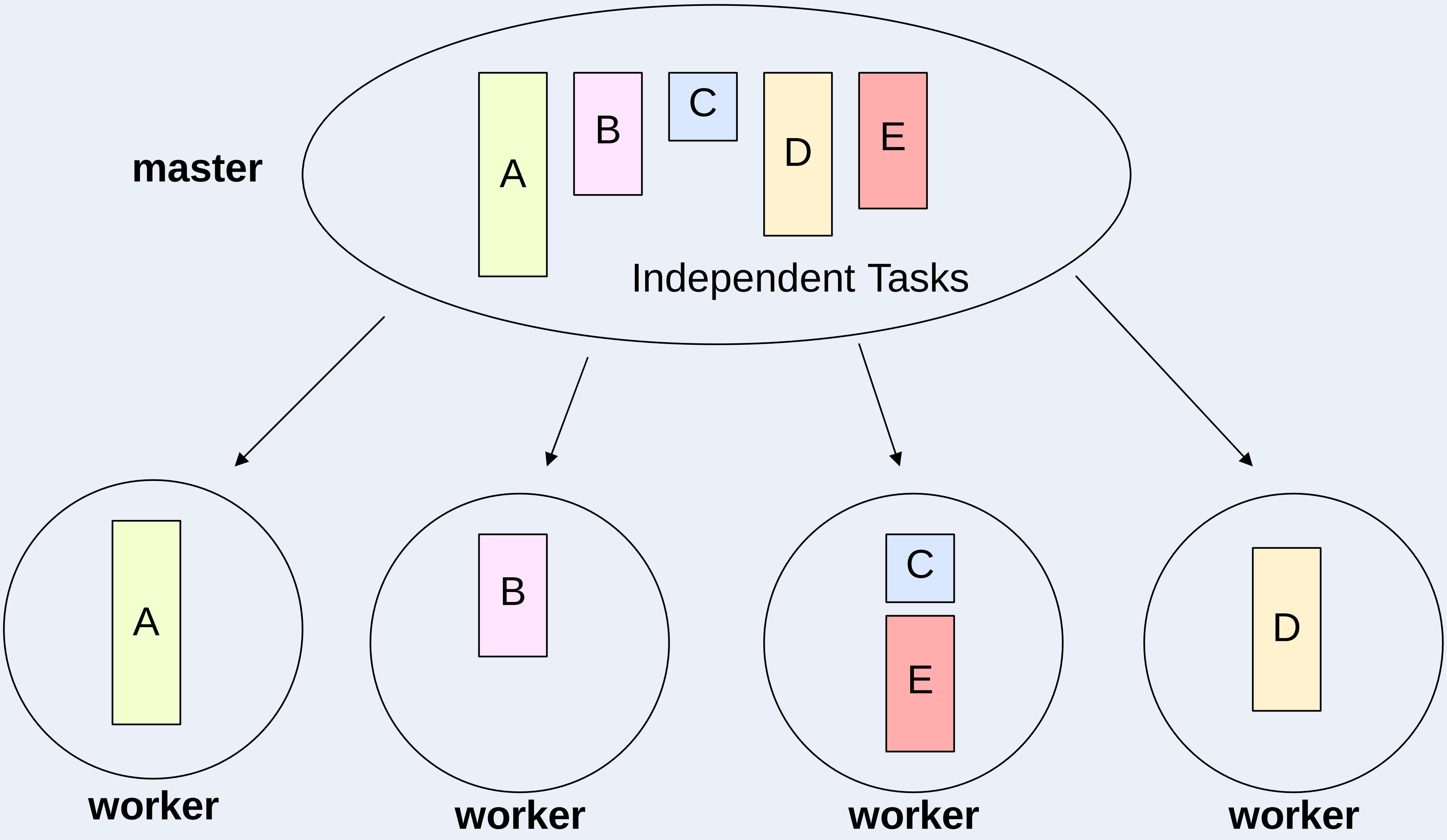
Loop Parallelism Pattern

- Many programs are expressed using iterative constructs
  - Programming models like OpenMP provide directives to automatically assign loop iteration to execution units
  - Especially good when code cannot be massively restructured

```
#pragma omp parallel for
for(i = 0; i < 12; i++)
      C[i] = A[i] + B[i];
```



Master/Worker Pattern



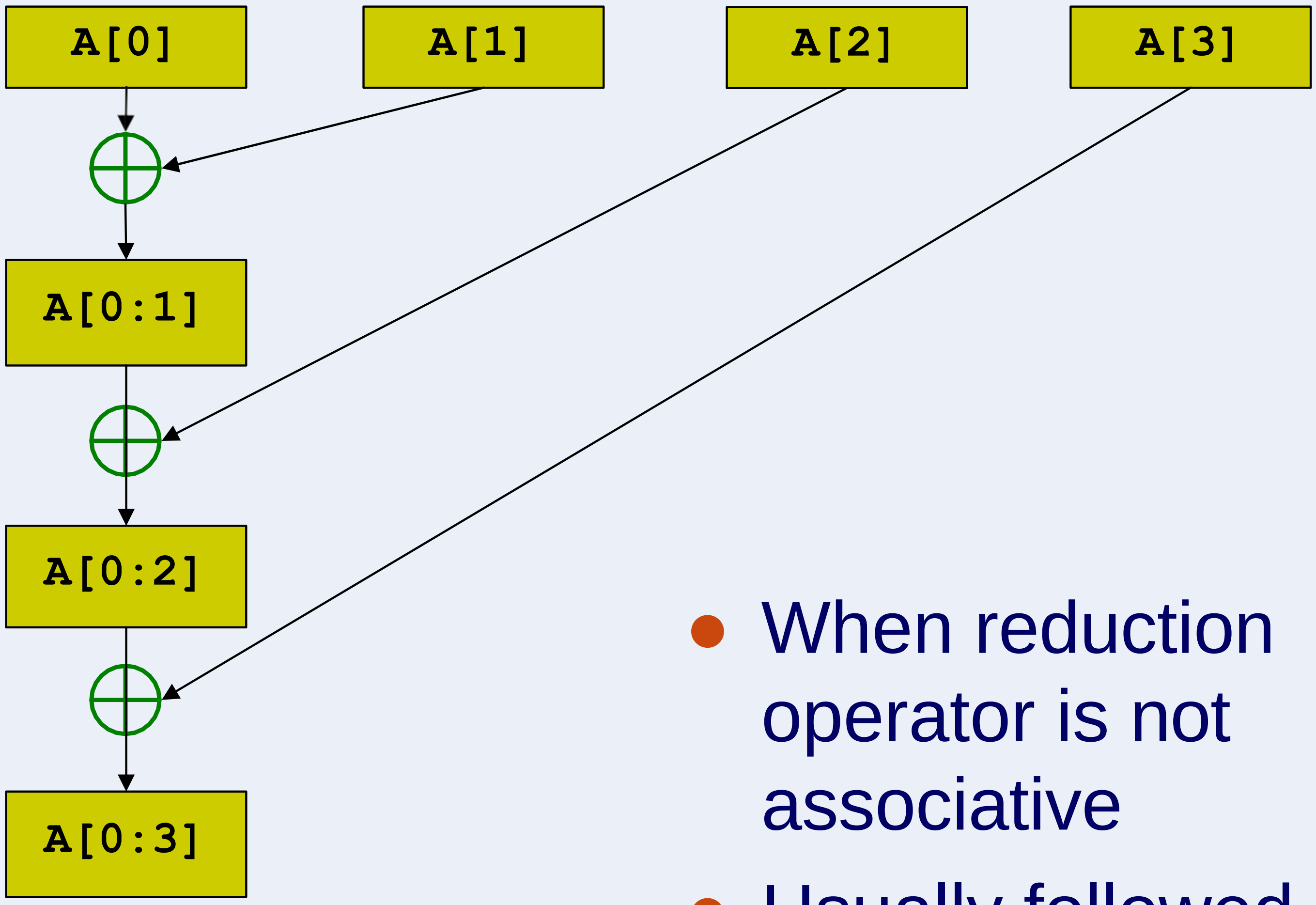
Master/Worker Pattern

- Particularly relevant for problems using task parallelism pattern where task have no dependencies
  - Embarrassingly parallel problems
- Main challenge in determining when the entire problem is complete

Fork/Join Pattern

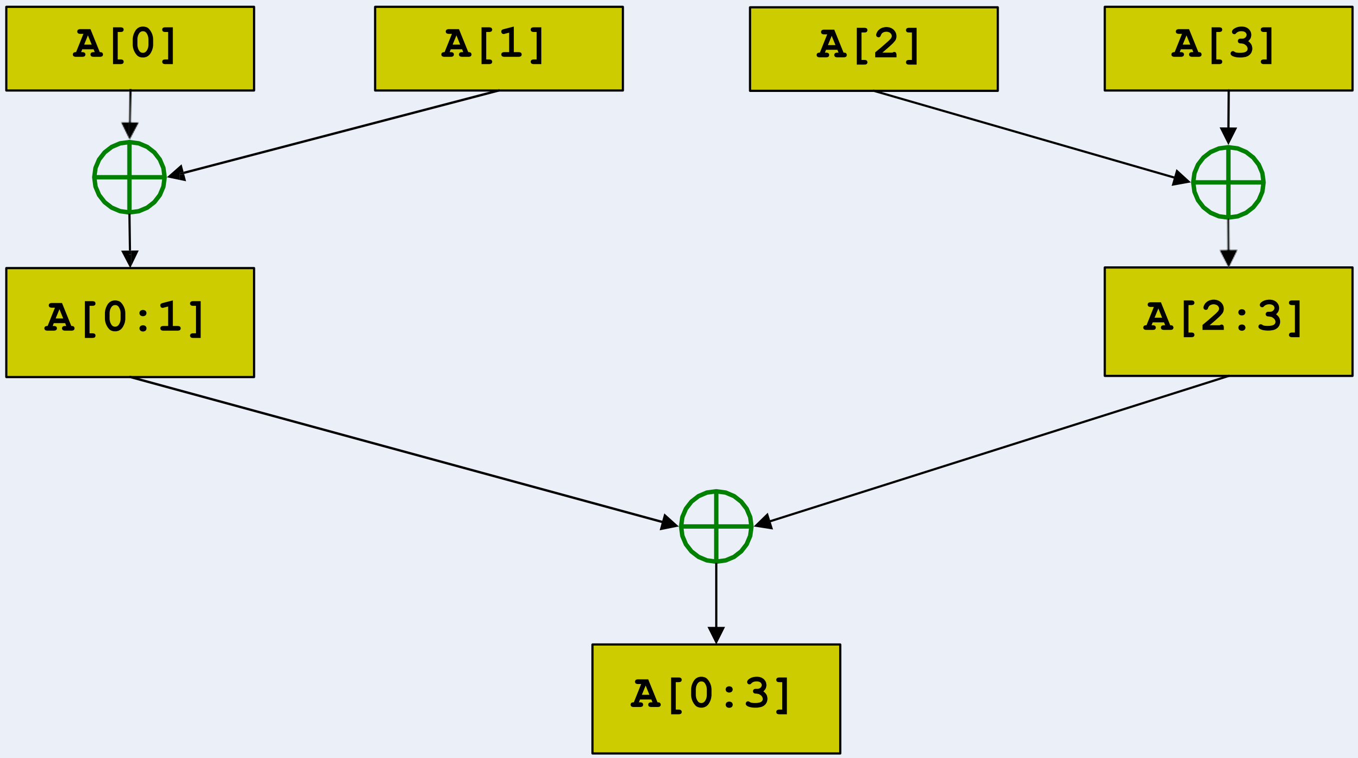
- Tasks are created dynamically
  - Tasks can create more tasks
- Manages tasks according to their relationship
- Parent task creates new tasks (fork) then waits until they complete (join) before continuing on with the computation

Serial Reduction



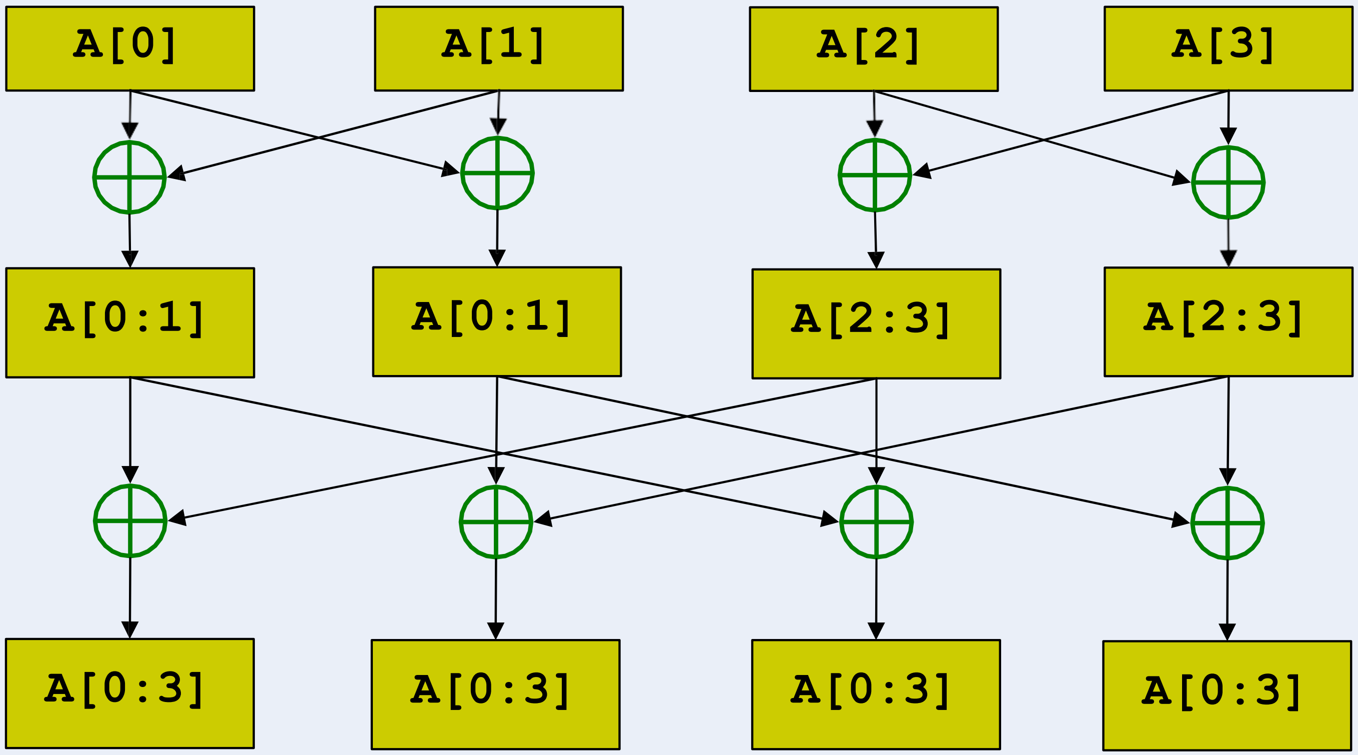
- When reduction operator is not associative
- Usually followed by a broadcast of result

Tree-based Reduction



- n steps for 2^n units of execution
- When reduction operator is associative
- Especially attractive when only one task needs result

Recursive-doubling Reduction



- n steps for 2^n units of execution
- If all units of execution need the result of the reduction

Recursive-doubling Reduction

- Better than tree-based approach with broadcast
  - Each units of execution has a copy of the reduced valut at the end of n steps
  - In tree-based approach with broadcast
    - Reduction takes n steps
    - Broadcast cannot begin until reduction is complete
    - Broadcast takes n steps (architecture dependent)
    - O(n) vs. O(2n)

Algorithm Structure and Organization

	Task parallelism	Divide and conquer	Geometric decomposition	Recursive data	Pipeline	Event-based coordination
SPMD	****	***	****	**	***	**
Loop Parallelism	****	**	***			
Master/ Worker	****	**	*	*	****	*
Fork/ Join	**	****	**		****	****

- Patterns can be hierarchically composed so that a program uses more than one pattern

Applications

Structural Patterns

- Pipe-and-Filter
- Agent-and-Repository
- Process-Control
- Event-Based/Implicit-Invocation
- Arbitrary-Static-Task-Graph
- Model-View-Controller
- Iterative-Refinement
- Map-Reduce
- Layered-Systems
- Puppeteer

Computational Patterns

- Graph-Algorithms
- Dynamic-Programming
- Dense-Linear-Algebra
- Sparse-Linear-Algebra
- Unstructured-Grids
- Structured-Grids
- Graphical-Models
- Finite-State-Machines
- Backtrack-Branch-and-Bound
- N-Body-Methods
- Circuits
- Spectral-Methods
- Monte-Carlo

Parallel Algorithm Strategy Patterns

- Task-Parallelism
- Divide and Conquer
- Data-Parallelism
- Pipeline
- Discrete-Event
- Geometric-Decomposition
- Speculation

Implementation Strategy Patterns

Program structure

- SPMD
- Fork/Join
- Kernel-Par.
- Loop-Par.
- Vector-Par.
- Actors
- Work-pile

Data structure

- Shared-Queue
- Shared-Map
- Shared-Data
- Partitioned-Array
- Partitioned-Graph

Parallel Execution Patterns

- Coordinating Processes
- Stream processing
- Shared Address Space Threads
- Task Driven Execution

Slides references

- Rodric Rabbah, IBM, MIT Lectures in Parallel programming, 2007