Algorithms for High-Performance Computing Platforms (2020-2021)

Course 2: Tasks

Laércio LIMA PILLA
pilla@lri.fr

Agenda

What is a task?
Parallel Patterns
Algorithmic Structures
Implementation Concepts

But first...


YouTube FR
credit song
COPY MACHINE
0:00 / 1:15
Credit Is Due (The Attribution Song)
112,659 views • 27 Jun 2011
1.5K
39
SHARE
SAVE
Question Copyright
4.68K subscribers
SUBSCRIBE

But first...

- Introduction to Parallel Computing by Allen Malony et al. from the University of Oregon: https://ipcc.cs.uoregon.edu/curriculum.html
- Rodric Rabbah, 6.189 Multicore Programming Primer, January (IAP) 2007. (Massachusetts Institute of Technology: MIT OpenCourseWare). http://ocw.mit.edu (accessed Oct 02, 2020). License: Creative Commons Attribution-Noncommercial-Share Alike.
- "HPC - from applications to tasks" by Francieli Zanon Boito.
- Bull, J. Mark. "A hierarchical classification of overheads in parallel programs." In Software Engineering for Parallel and Distributed Systems, pp. 208-219. Springer, Boston, MA, 1996. https://link.springer.com/content/pdf/10.1007/978-0-387-34984-8_18.pdf
- Parallel Program Engineering by Michael Gerndt et al., http://wwwi10.lrr.in.tum.de/~gerndt/home/Teaching/PPE/PPE.html

What is a task?

What is a task?

Loose definition:

Concurrent/parallel unit of work.

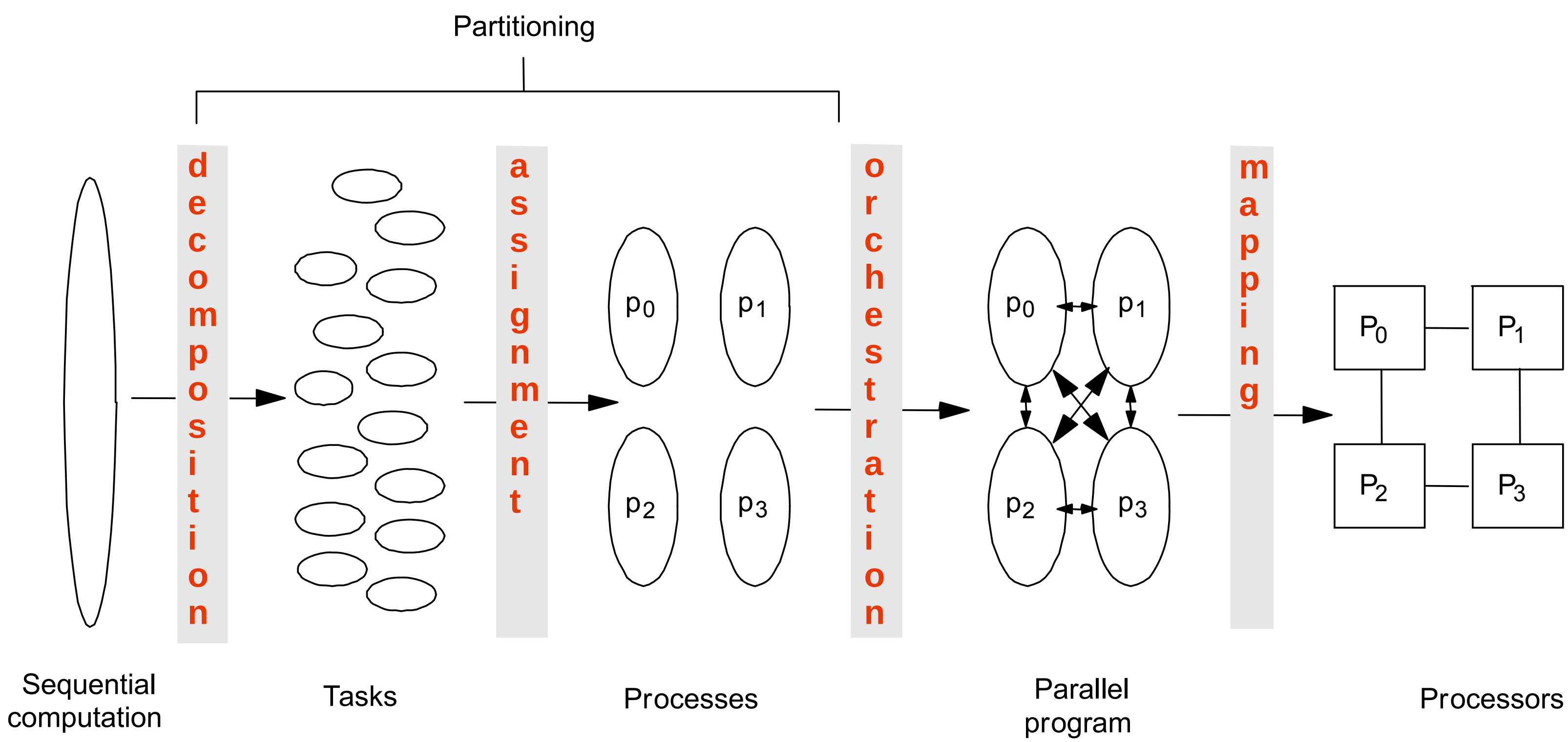
The meaning of a "task" is context-dependent.

We can have tasks within tasks within tasks…

« des tortues jusqu'en bas »

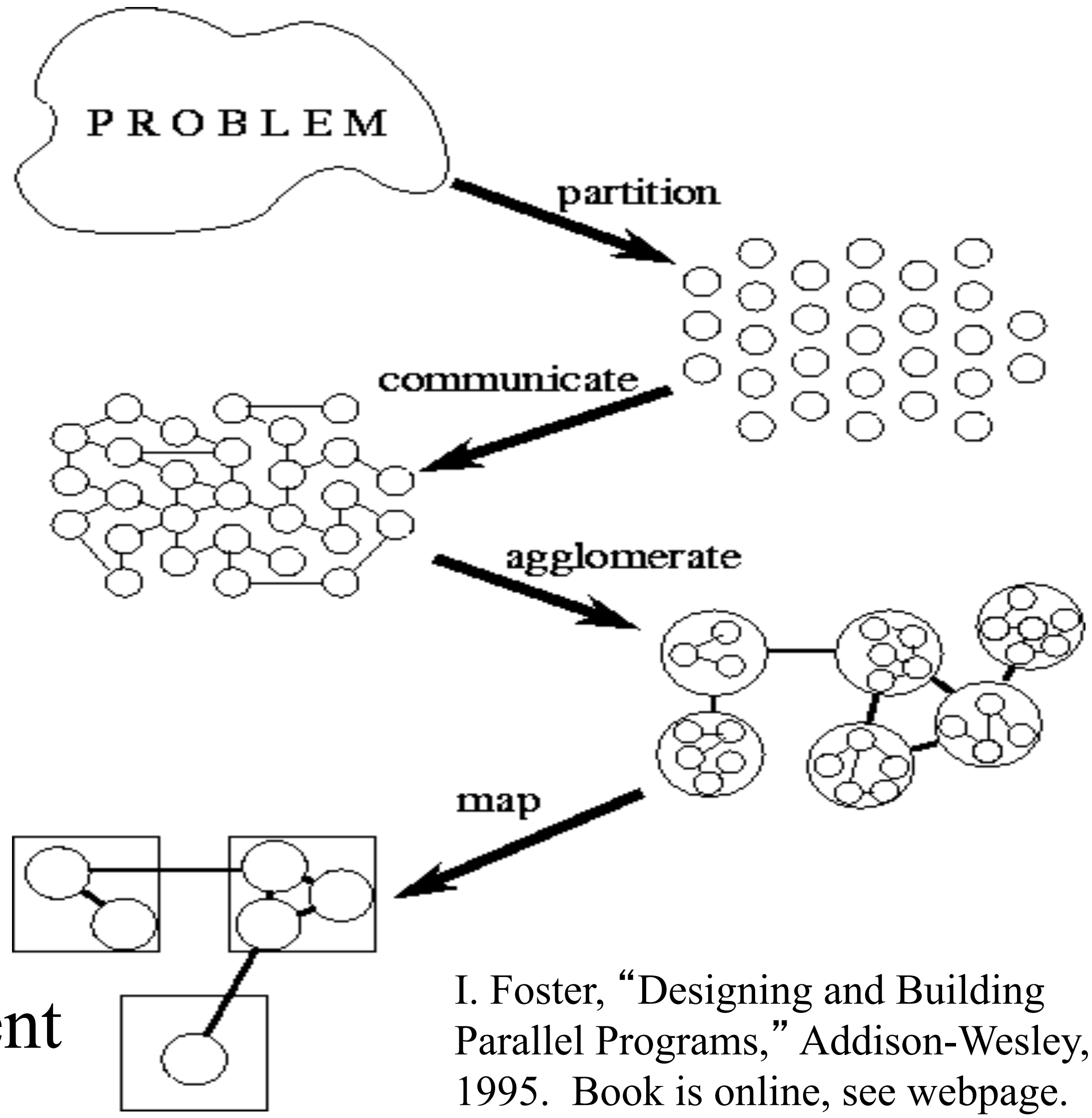
By Pelf at en.wikipedia - Originally from en.wikipedia; description page is/was here., Public Domain, https://commons.wikimedia.org/w/index.php?curid=2747463

4 Common Steps to Creating a Parallel Program



Methodological Design

- Partition
  - Task/data decomposition
- Communication
  - Task execution coordination
- Agglomeration
  - Evaluation of the structure
- Mapping
  - Resource assignment



I. Foster, "Designing and Building Parallel Programs," Addison-Wesley, 1995. Book is online, see webpage.

What is a task?

Examples of what a task means for parallelism/scheduling

For ...	… tasks are/can be thought as ...
A Supercomputer/Cluster	Jobs (application instances)
An Operating System	Threads, Processes
A Processor	Instructions
A Game	AI for NPCs, rendering, physics simulation
Finances	A Model with different inputs
Distributed Machine Learning	Mini-batches
A Scientific Workflow	Applications or scripts
A Climate Model	An Atmospheric Model, Ocean Model, ...

What is a task?

Parallelism means tasks.

Tasks mean we have to manage tasks.

How difficult can it be?

What is a task?

Bull, J. Mark.

"A hierarchical classification of overheads in parallel programs."

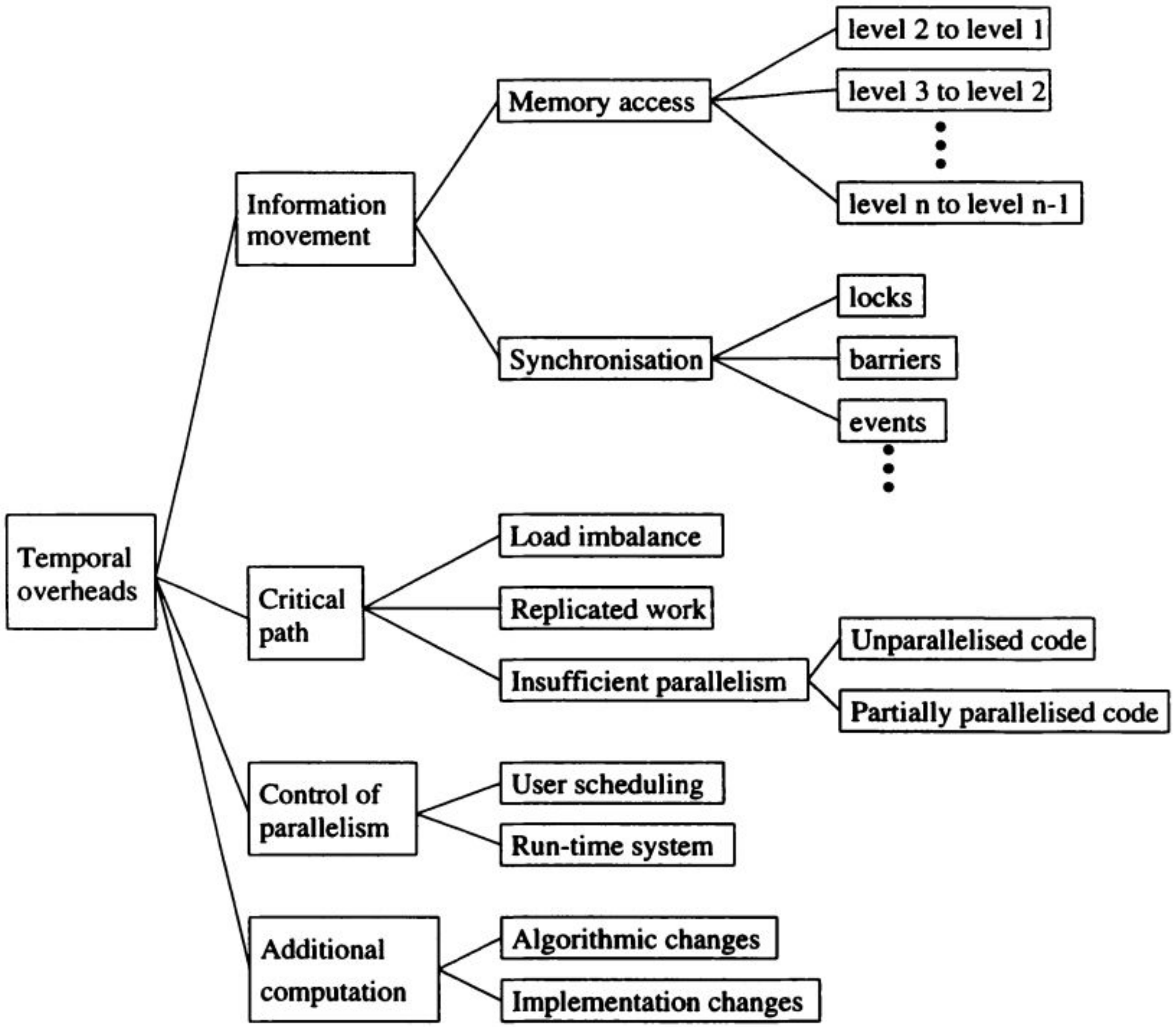


Figure 1 Classification of temporal overheads.

What is a task?

Bull, J. Mark.

"A hierarchical classification of overheads in parallel programs."

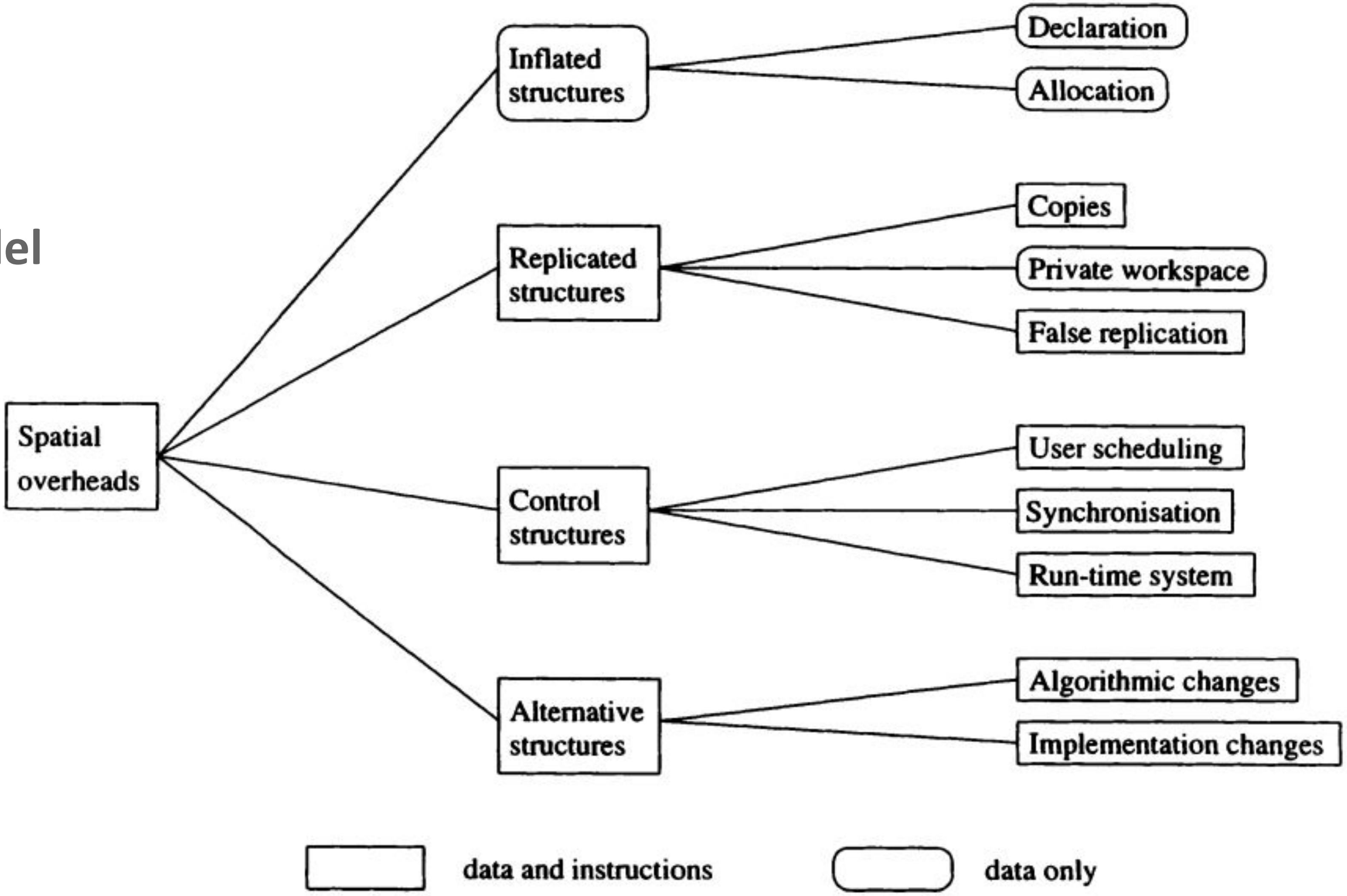
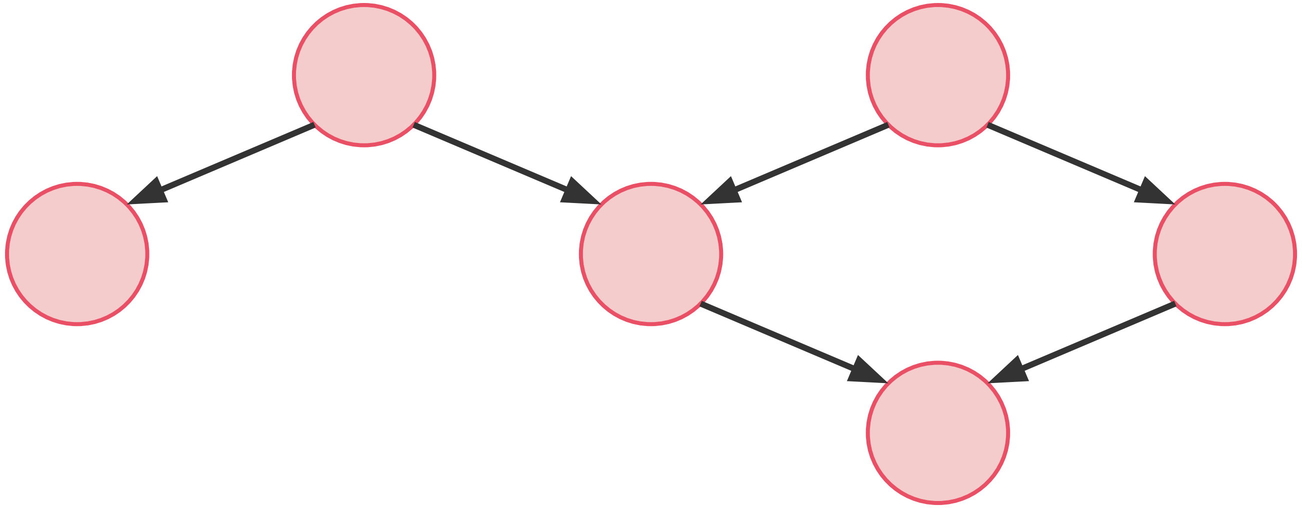


Figure 2 Classification of spatial overheads.

What is a task?

Representation of tasks as a Direct Acyclic Graph (DAG)

- **Tasks with dependencies**

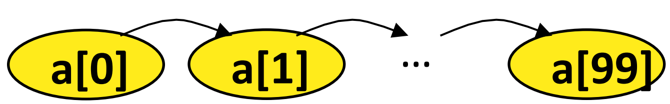


- **Embarrassingly parallel (EP) problems**

Directed Acyclic Graphs (DAG)

- Captures data flow parallelism
- Nodes represent operations to be performed
  - Inputs are nodes with no incoming arcs
  - Output are nodes with no outgoing arcs
  - Think of nodes as tasks
- Arcs are paths for flow of data results
- DAG represents the operations of the algorithm and implies precedent constraints on their order

```
for (i=1; i<100; i++)
        a[i] = a[i-1] + 100;
```



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

$R_1 = \{ x, y \}$
$W_1 = \{ a \}$

$R_2 = \{ x, z \}$
$W_2 = \{ b \}$

$$R_1 \cap W_2 = \phi$$

$$R_2 \cap W_1 = \phi$$

$$W_1 \cap W_2 = \phi$$

Independent versus Dependent

- In other words the execution of

  statement1;

  statement2;

  must be equivalent to

  statement2;

  statement1;

- Their order of execution must not matter!
- If true, the statements are *independent* of each other
- Two statements are *dependent* when the order of their execution affects the computation outcome

Examples

- Example 1
  S1: a=1;
  S2: b=1;
- Statements are independent
- Example 2
  S1: a=1;
  S2: b=a;
- Dependent (*true (flow) dependence*)
  - Second is dependent on first
  - Can you remove dependency?
- Example 3
  S1: a=f(x);
  S2: a=b;
- Dependent (*output dependence*)
  - Second is dependent on first
  - Can you remove dependency? How?
- Example 4
  S1: a=b;
  S2: b=1;
- Dependent (*anti-dependence*)
  - First is dependent on second
  - Can you remove dependency? How?

True Dependence and Anti-Dependence

- Given statements S1 and S2,

  S1;

  S2;

- S2 has a *true (flow) dependence* on S1

  if and only if

  S2 reads a value written by S1

  X =
  ⋮
  = X

  δ

- S2 has an *anti-dependence* on S1

  if and only if

  S2 writes a value read by S1

  = X
  ⋮
  X =

  δ^{-1}

Output Dependence

- Given statements S1 and S2,

  S1;

  S2;

- S2 has an *output dependence* on S1

  if and only if

  S2 writes a variable written by S1

X =
⋮
X =
δ^0

- Anti- and output dependences are "name" dependencies
  - Are they "true" dependences?
- How can you get rid of output dependences?
  - Are there cases where you can not?

Statement Dependency Graphs

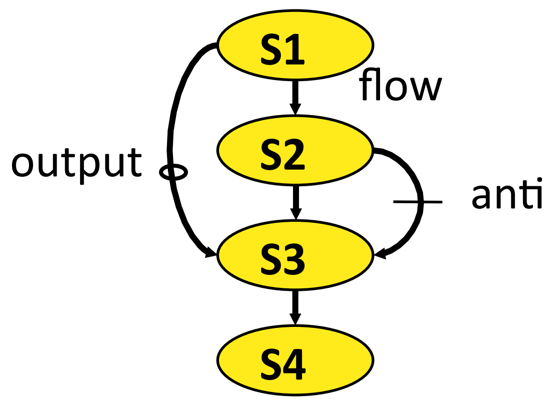
- Can use graphs to show dependence relationships
- Example

  S1: a=1;

  S2: b=a;

  S3: a=b+1;

  S4: c=a;



- $S_2 \;\delta\; S_3$: S_3 is flow-dependent on S_2
- $S_1 \;\delta^0\; S_3$: S_3 is output-dependent on S_1
- $S_2 \;\delta^{-1}\; S_3$: S_3 is anti-dependent on S_2

When can two statements execute in parallel?

- Statements S1 and S2 can execute in parallel if and only if there are *no dependences* between S1 and S2
  - True dependences
  - Anti-dependences
  - Output dependences
- Some dependences can be remove by modifying the program
  - Rearranging statements
  - Eliminating statements

How do you compute dependence?

- Data dependence relations can be found by comparing the IN and OUT sets of each node
- The IN and OUT sets of a statement S are defined as:
  - IN(S) : set of memory locations (variables) that may be used in S
  - OUT(S) : set of memory locations (variables) that may be modified by S
- Note that these sets include all memory locations that may be fetched or modified
- As such, the sets can be conservatively large

Parallel Patterns

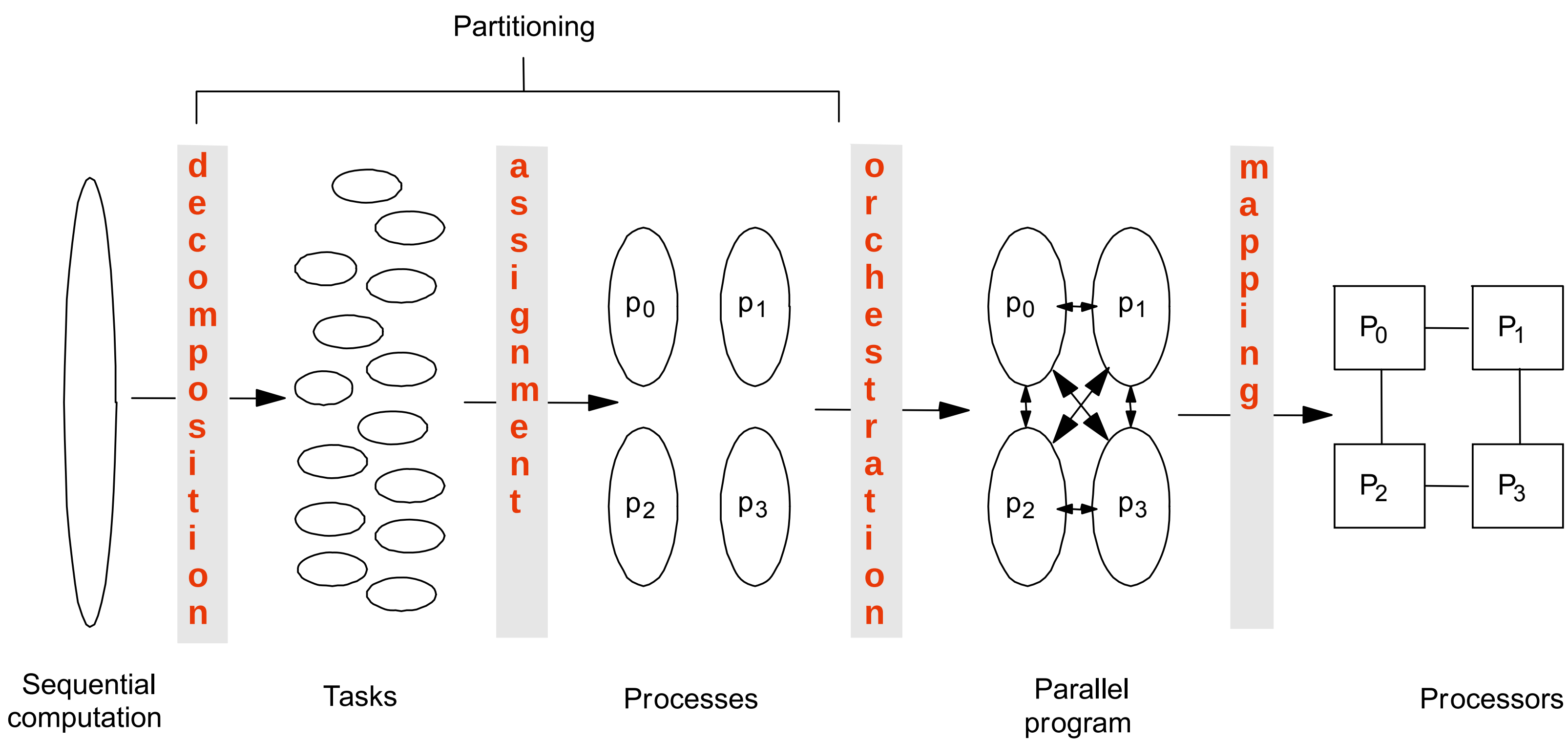
Parallel Patterns

Two main ways to think about partitioning an application

- **Task decomposition**
  - **Also known as *Functional decomposition***
  - **Different computations -> different tasks**

- **Data decomposition**
  - **Also known as *Domain decomposition***
  - **Same computation applied to different data**
  - **Different parts of the data -> different tasks**

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

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.

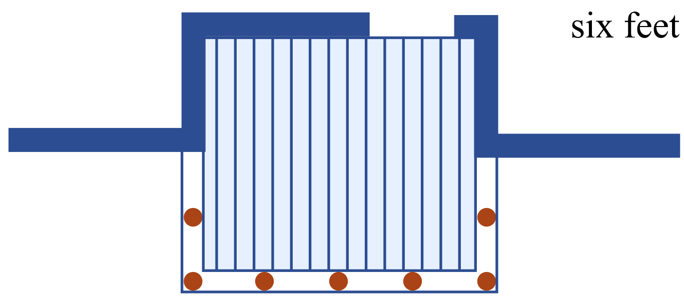


Image by MIT OpenCourseWare.

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

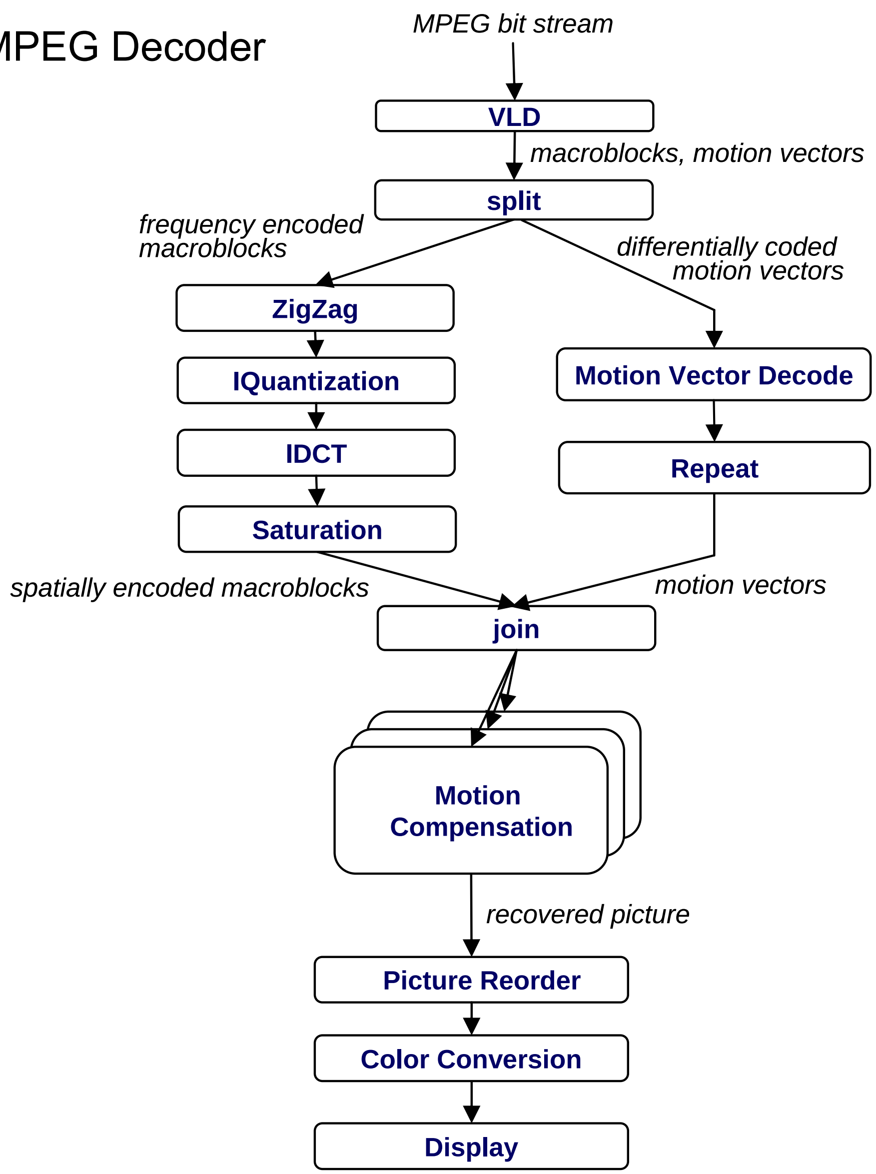
Software Construction

- Supporting Structures
  - Code and data structuring patterns
- Implementation Mechanisms
  - Low level mechanisms used to write parallel programs

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

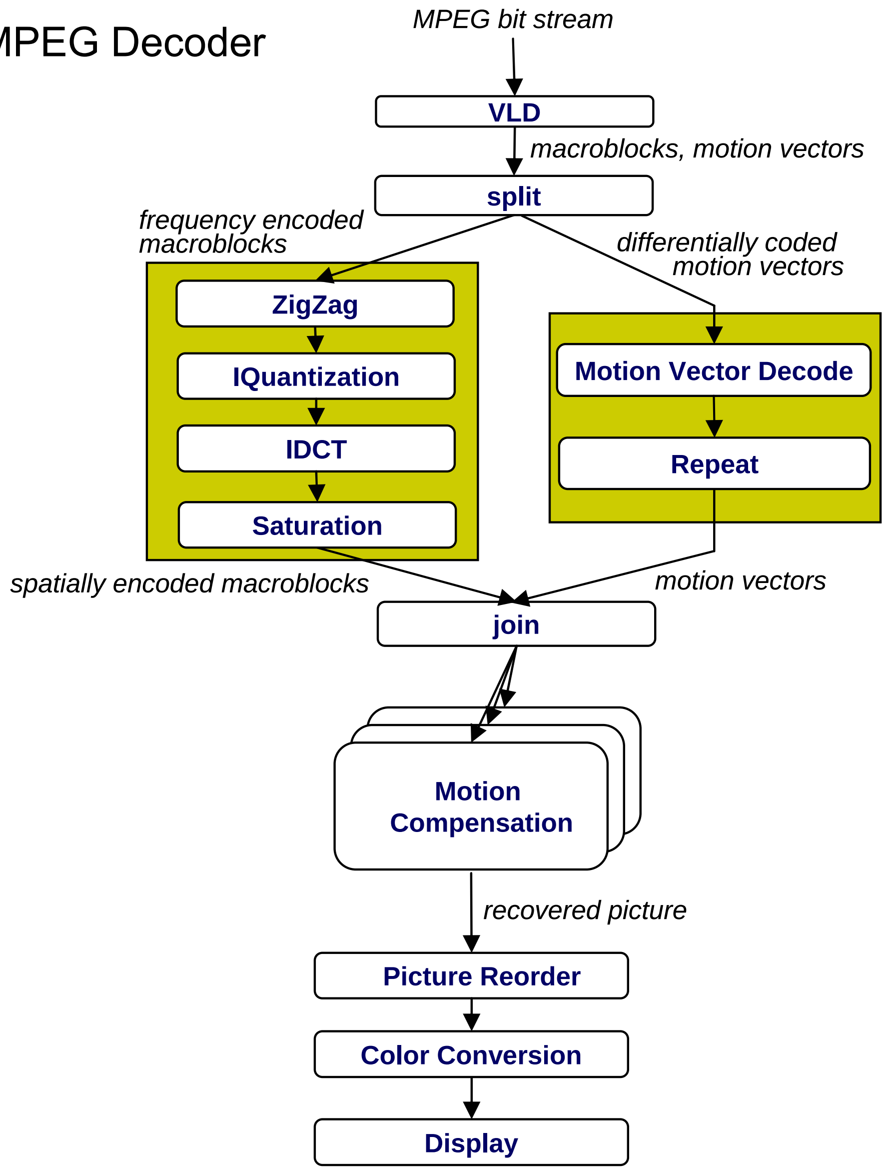
Here’s my algorithm. Where’s the concurrency?

MPEG Decoder



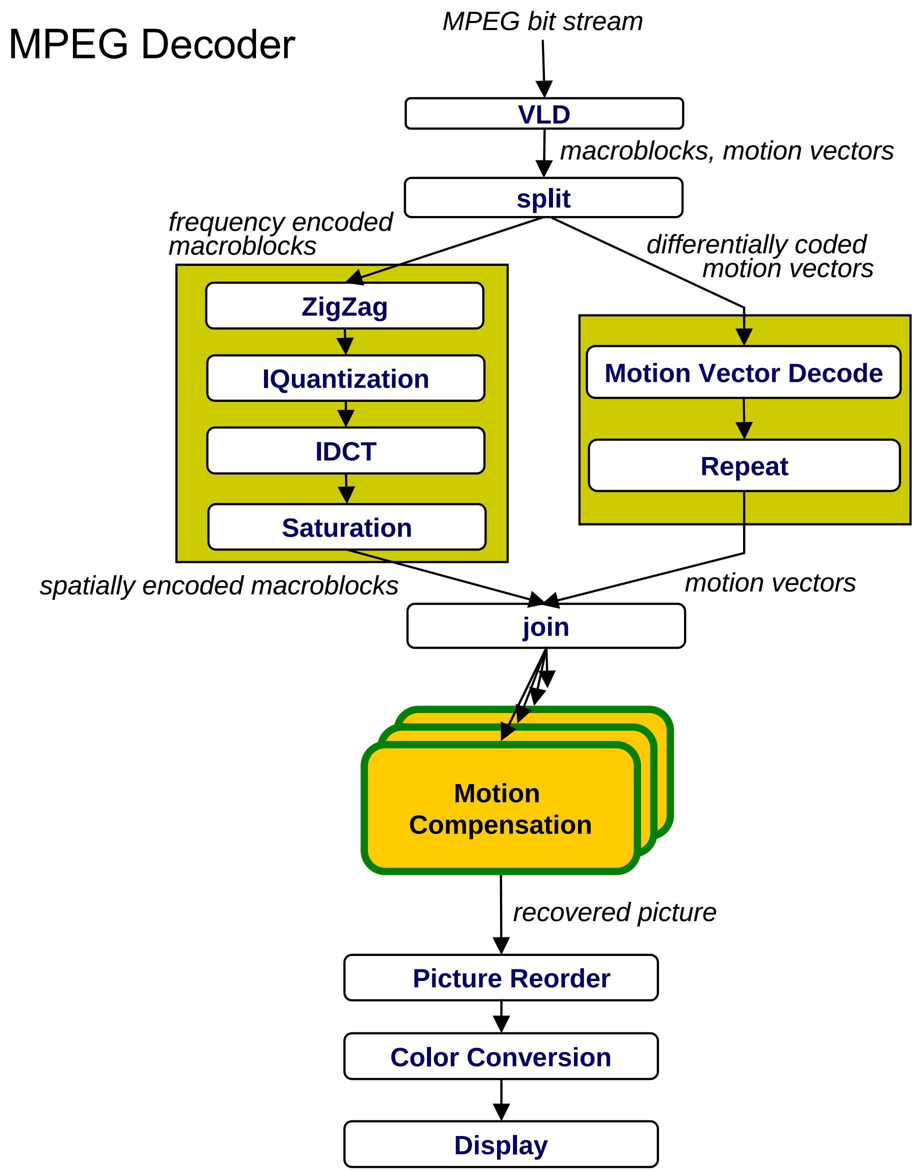
Here's my algorithm. Where's the concurrency?

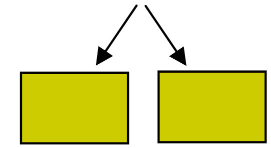
MPEG Decoder



- 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

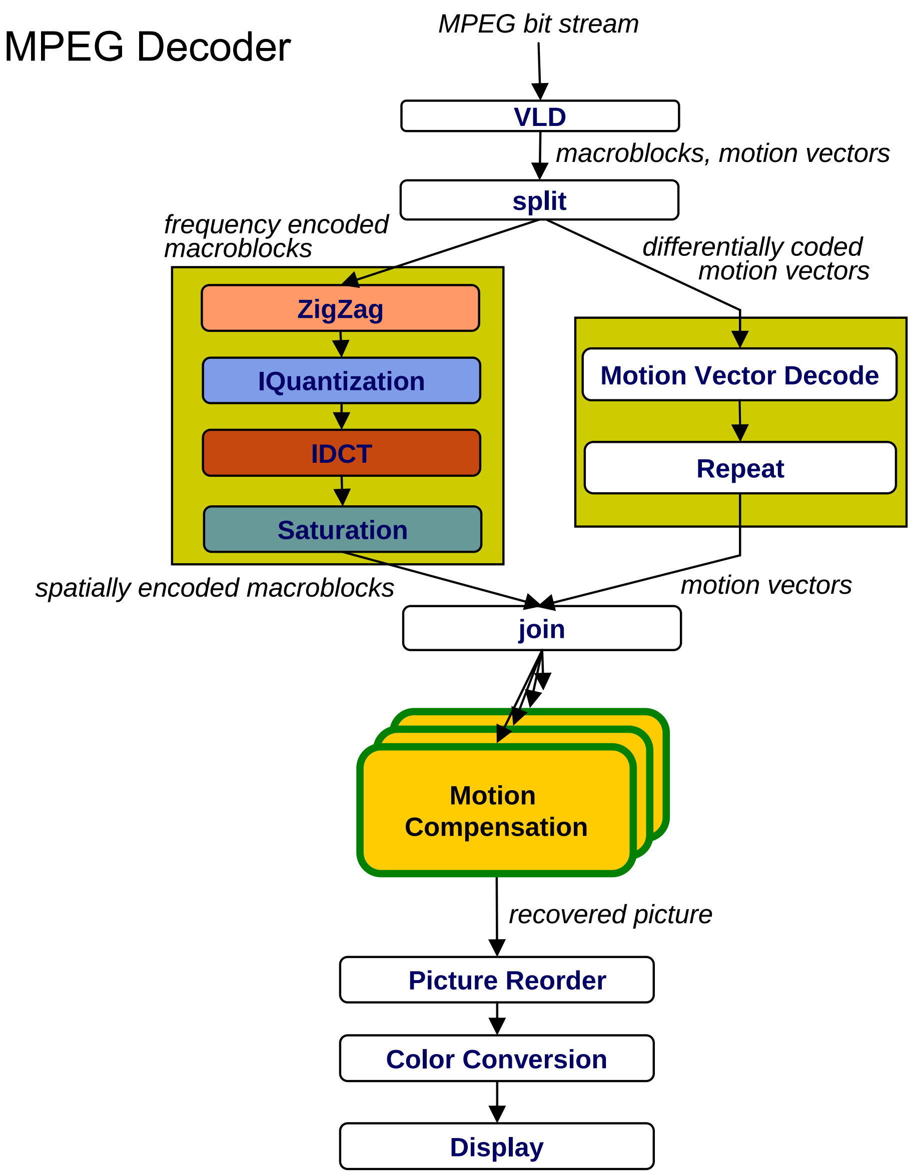
Here's my algorithm.
Where's the concurrency?



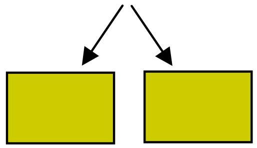


- Task decomposition
  - Parallelism in the application
- Data decomposition
  - Same computation is applied to small data chunks derived from large data set

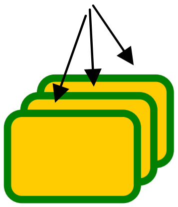
Here's my algorithm. Where's the concurrency?



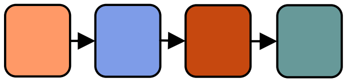
- Task decomposition
  - Parallelism in the application



- 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 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

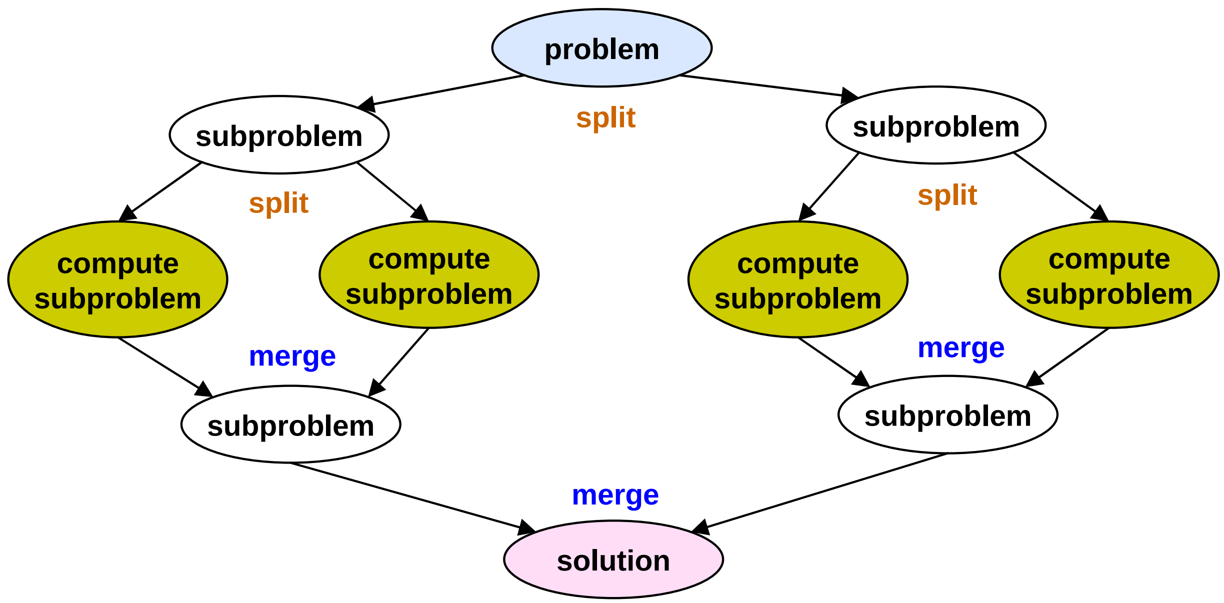
Guidelines for Data Decomposition

- 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

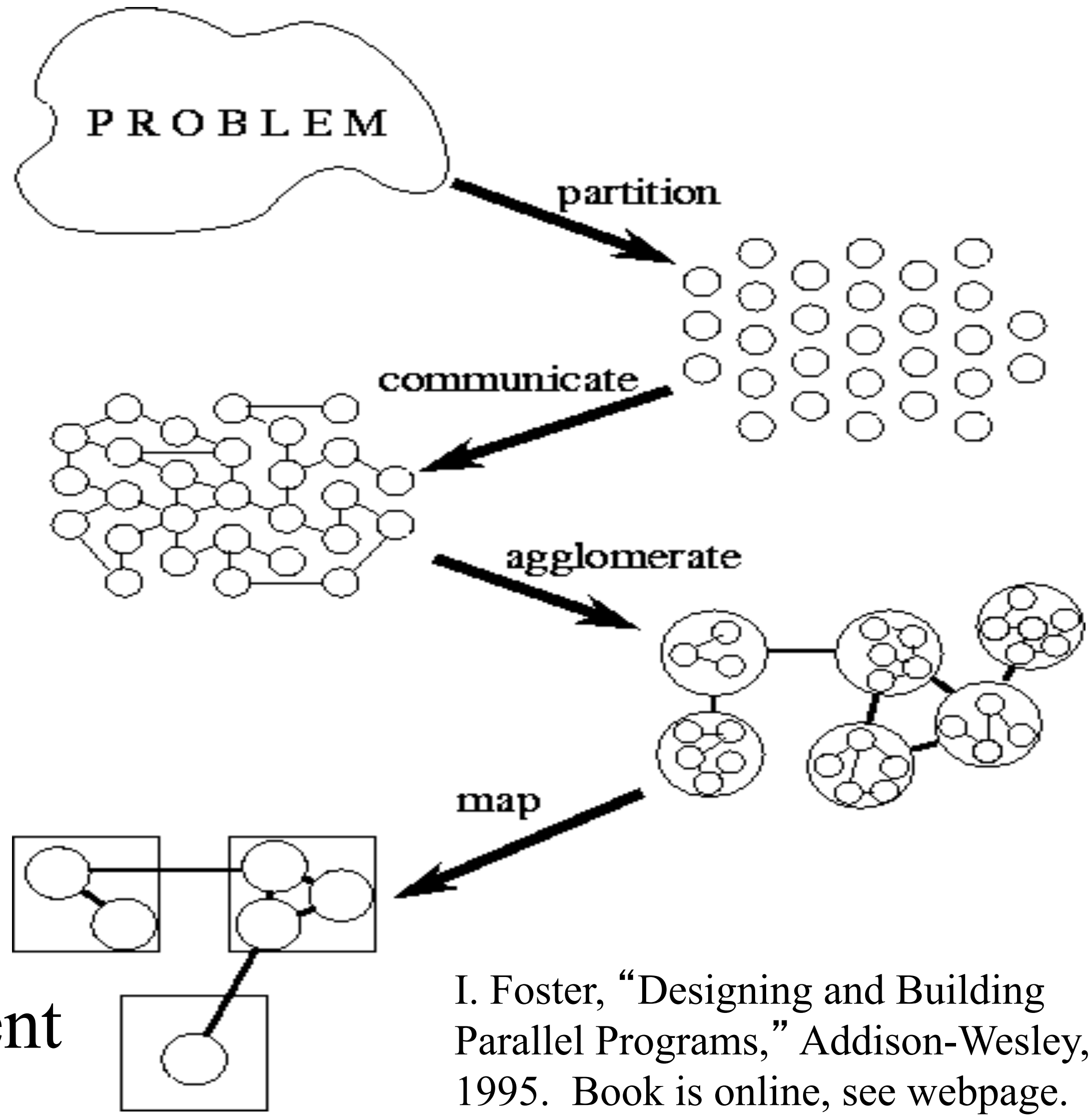


Guidelines for Data Decomposition

- 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

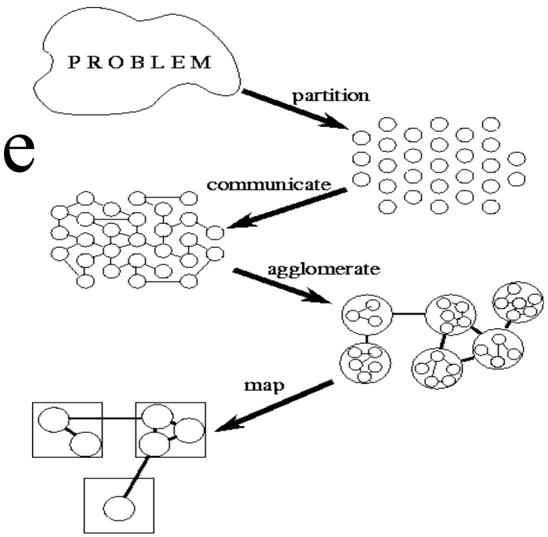
Methodological Design

- Partition
  - Task/data decomposition
- Communication
  - Task execution coordination
- Agglomeration
  - Evaluation of the structure
- Mapping
  - Resource assignment



I. Foster, "Designing and Building Parallel Programs," Addison-Wesley, 1995. Book is online, see webpage.

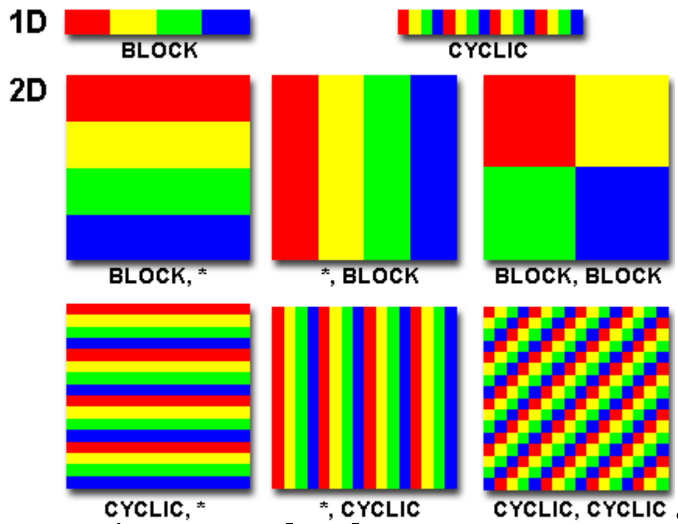
Partitioning

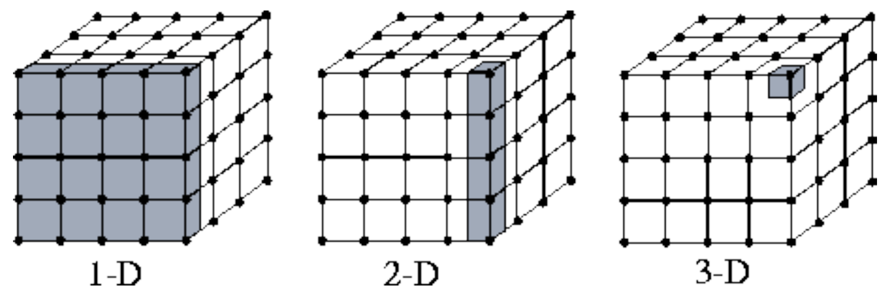


- Partitioning stage is intended to expose opportunities for parallel execution
- Focus on defining large number of small task to yield a fine-grained decomposition of the problem
- A good partition divides into small pieces both the computational *tasks* associated with a problem and the *data* on which the tasks operates
- *Domain decomposition* focuses on computation data
- *Functional decomposition* focuses on computation tasks
- Mixing domain/functional decomposition is possible

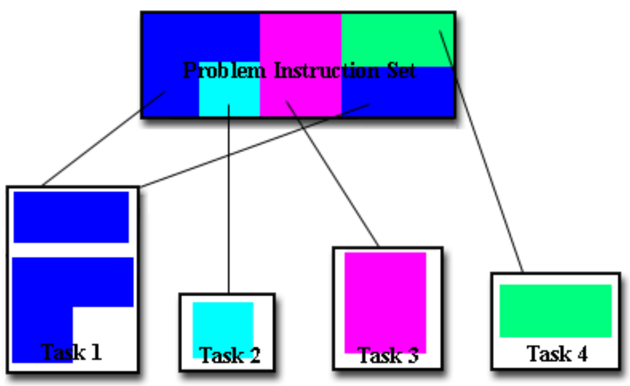
Domain and Functional Decomposition

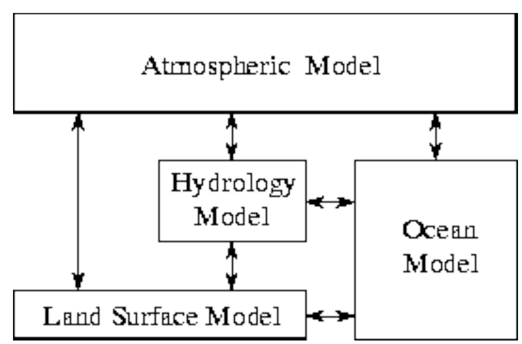
- Domain decomposition of 2D / 3D grid





- Functional decomposition of a climate model

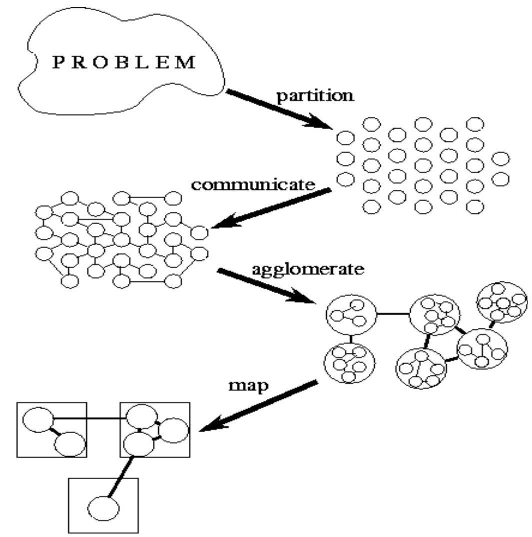




Partitioning Checklist

- Does your partition define at least an order of magnitude more tasks than there are processors in your target computer? If not, may loose design flexibility.
- Does your partition avoid redundant computation and storage requirements? If not, may not be scalable.
- Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.
- Does the number of tasks scale with problem size? If not may not be able to solve larger problems with more processors
- Have you identified several alternative partitions?

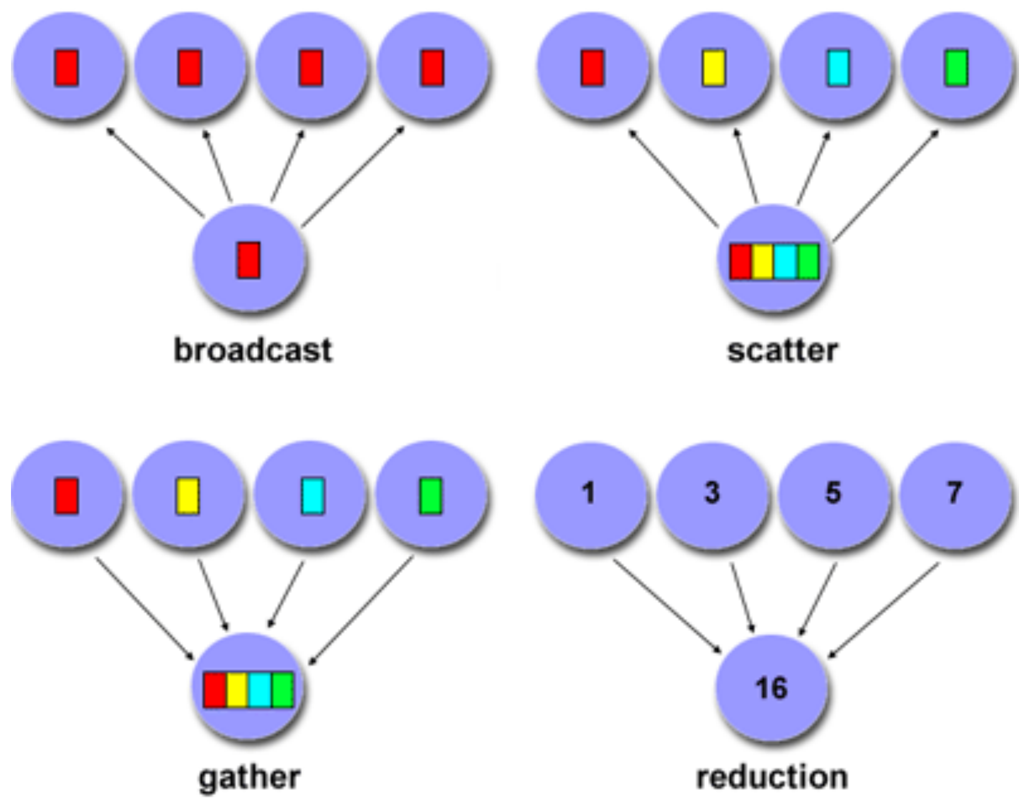
Communication (Interaction)



- Tasks generated by a partition must interact to allow the computation to proceed
  - Information flow: data and control
- Types of communication
  - *Local* vs. *Global*: locality of communication
  - *Structured* vs. *Unstructured*: communication patterns
  - *Static* vs. *Dynamic*: determined by runtime conditions
  - *Synchronous* vs. *Asynchronous*: coordination degree
- Granularity and frequency of communication
  - Size of data exchange
- Think of communication as interaction and control
  - Applicable to both shared and distributed memory parallelism

Types of Communication

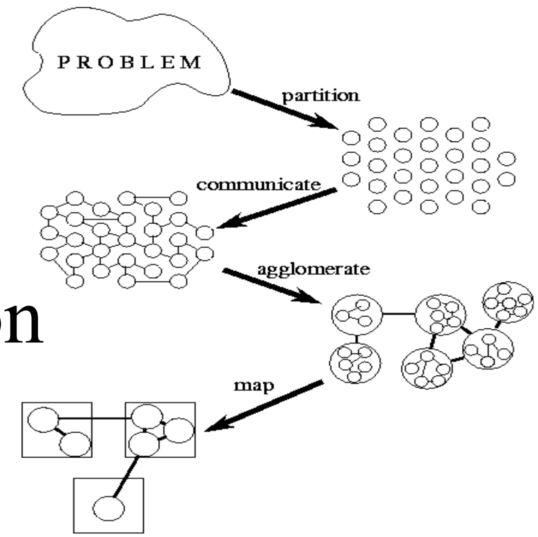
- Point-to-point
- Group-based
- Hierachical
- Collective



Communication Design Checklist

- Is the distribution of communications equal?
  - Unbalanced communication may limit scalability
- What is the communication locality?
  - Wider communication locales are more expensive
- What is the degree of communication concurrency?
  - Communication operations may be parallelized
- Is computation associated with different tasks able to proceed concurrently? Can communication be overlapped with computation?
  - Try to reorder computation and communication to expose opportunities for parallelism

Agglomeration

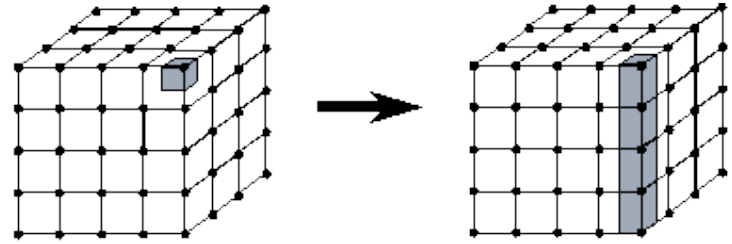


- Move from parallel abstractions to real implementation
- Revisit partitioning and communication
  - View to efficient algorithm execution
- Is it useful to *agglomerate*?
  - What happens when tasks are combined?
- Is it useful to *replicate* data and/or computation?
- Changes important algorithm and performance ratios
  - *Surface-to-volume*: reduction in communication at the expense of decreasing parallelism
  - *Communication/computation*: which cost dominates
- Replication may allow reduction in communication
- Maintain flexibility to allow overlap

Types of Agglomeration

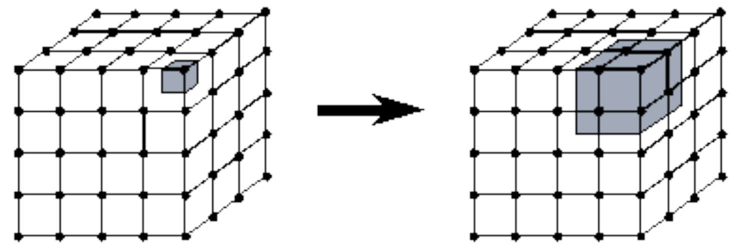
- Element to column

(a)



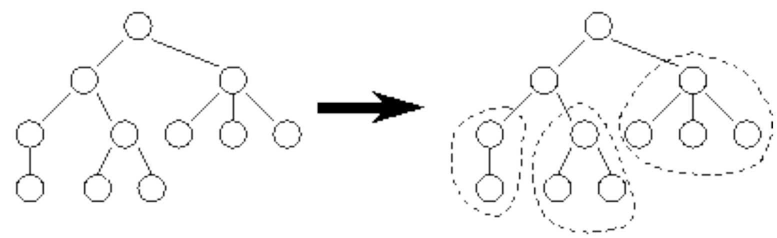
- Element to block
  - Better surface to volume

(b)



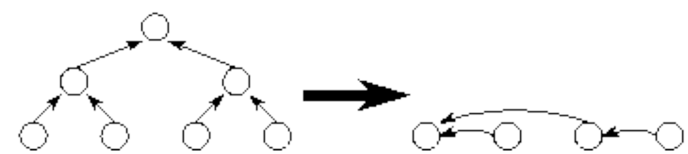
- Task merging

(c)



- Task reduction
  - Reduces communication

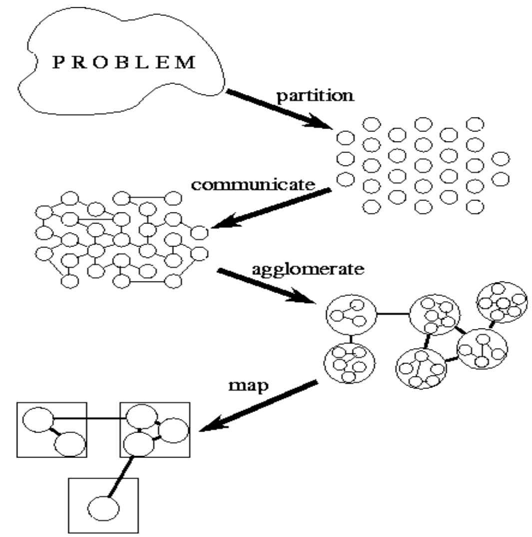
(d)



Agglomeration Design Checklist

- Has increased locality reduced communication costs?
- Is replicated computation worth it?
- Does data replication compromise scalability?
- Is the computation still balanced?
- Is scalability in problem size still possible?
- Is there still sufficient concurrency?
- Is there room for more agglomeration?
- Fine-grained vs. coarse-grained?

Mapping



- Specify where each task is to execute
  - Less of a concern on shared-memory systems
- Attempt to minimize execution time
  - Place concurrent tasks on different processors to enhance physical concurrency
  - Place communicating tasks on same processor, or on processors close to each other, to increase locality
  - Strategies can conflict!
- Mapping problem is *NP-complete*
  - Use problem classifications and heuristics
- Static and dynamic load balancing

Mapping Algorithms

- Load balancing (partitioning) algorithms
- Data-based algorithms
  - Think of computational load with respect to amount of data being operated on
  - Assign data (i.e., work) in some known manner to balance
  - Take into account data interactions
- Task-based (task scheduling) algorithms
  - Used when functional decomposition yields many tasks with weak locality requirements
  - Use task assignment to keep processors busy computing
  - Consider centralized and decentralize schemes

Mapping Design Checklist

- Is static mapping too restrictive and non-responsive?
- Is dynamic mapping too costly in overhead?
- Does centralized scheduling lead to bottlenecks?
- Do dynamic load-balancing schemes require too much coordination to re-balance the load?
- What is the tradeoff of dynamic scheduling complexity versus performance improvement?
- Are there enough tasks to achieve high levels of concurrency? If not, processors may idle.

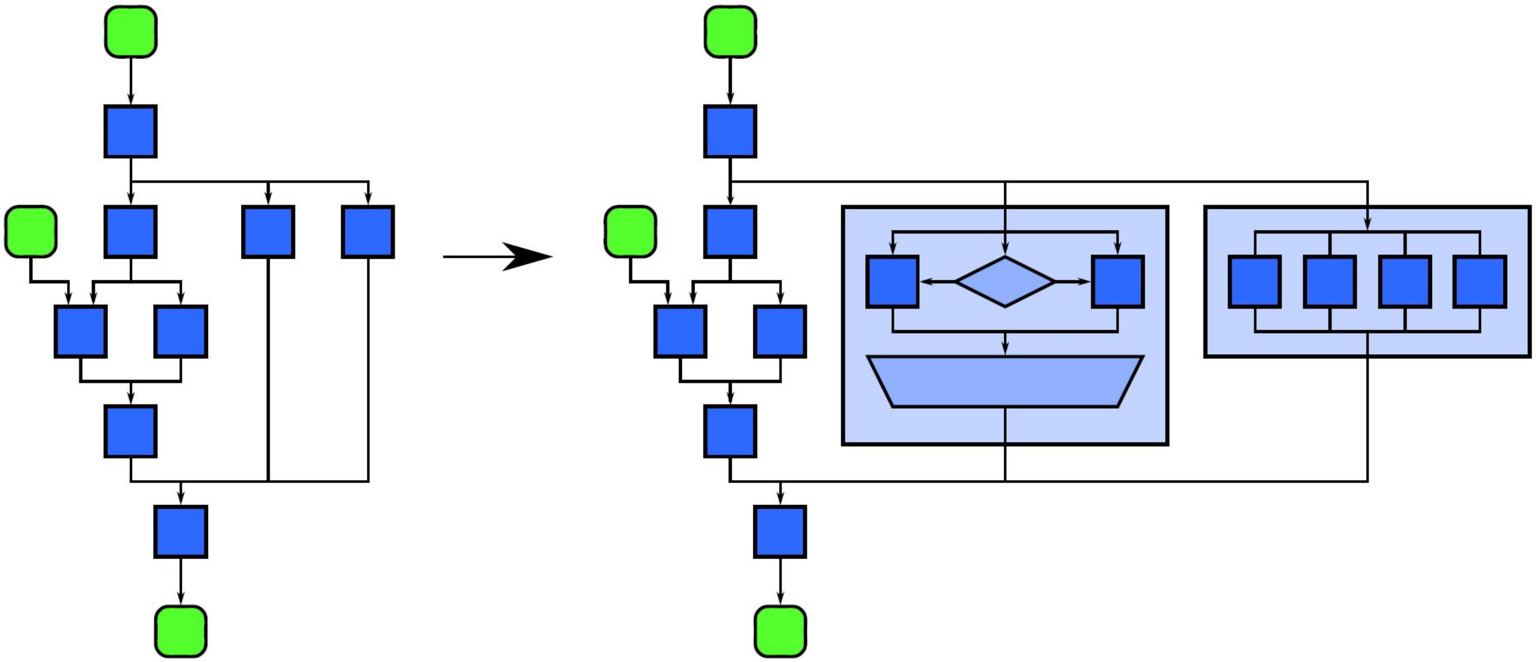
Parallel Patterns

- **Parallel Patterns**: A recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design.
- Patterns provide us with a "vocabulary" for algorithm design
- It can be useful to compare parallel patterns with serial patterns
- Patterns are universal – they can be used in *any* parallel programming system

Nesting Pattern

- **Nesting** is the ability to hierarchically compose patterns
- This pattern appears in both serial and parallel algorithms
- "Pattern diagrams" are used to visually show the pattern idea where each "task block" is a location of general code in an algorithm
- Each "task block" can in turn be another pattern in the **nesting pattern**

Nesting Pattern



Nesting Pattern: A compositional pattern. Nesting allows other patterns to be composed in a hierarchy so that any task block in the above diagram can be replaced with a pattern with the same input/output and dependencies.

Parallel Control Patterns

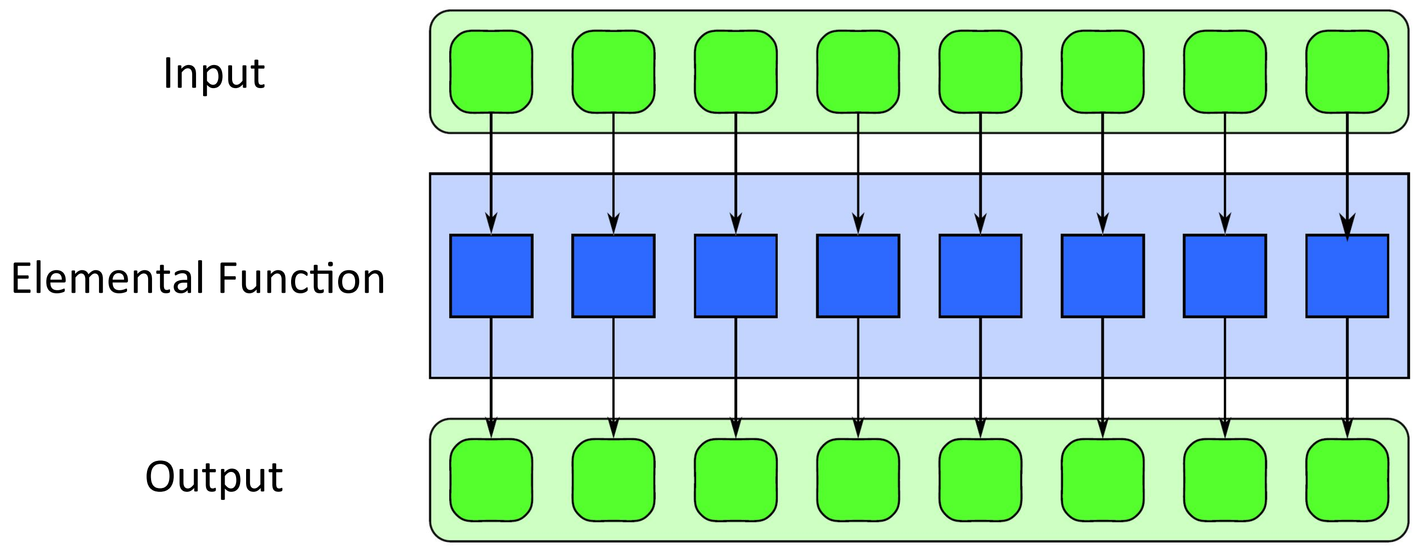
- Parallel control patterns extend serial control patterns
- Each parallel control pattern is related to at least one serial control pattern, but relaxes assumptions of serial control patterns
- Parallel control patterns: **fork-join**, **map**, **stencil**, **reduction**, **scan**, **recurrence**

Parallel Control Patterns: Fork-Join

- **Fork-join**: allows control flow to fork into multiple parallel flows, then rejoin later
- Cilk Plus implements this with **spawn** and **sync**
  - The call tree is a parallel call tree and functions are spawned instead of called
  - Functions that spawn another function call will continue to execute
  - Caller *syncs* with the spawned function to join the two
- A "join" is different than a "barrier
  - Sync – only one thread continues
  - Barrier – all threads continue

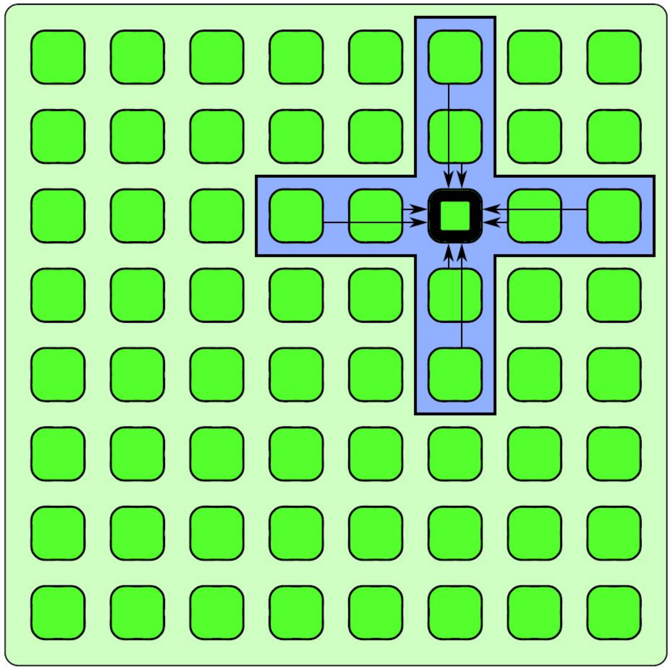
Parallel Control Patterns: Map

- **Map**: performs a function over every element of a collection
- Map replicates a serial iteration pattern where each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection
- The replicated function is referred to as an "elemental function"



Parallel Control Patterns: Stencil

- **Stencil**: Elemental function accesses a set of "neighbors", stencil is a generalization of map
- Often combined with iteration – used with iterative solvers or to evolve a system through time



- Boundary conditions must be handled carefully in the stencil pattern
- See stencil lecture…

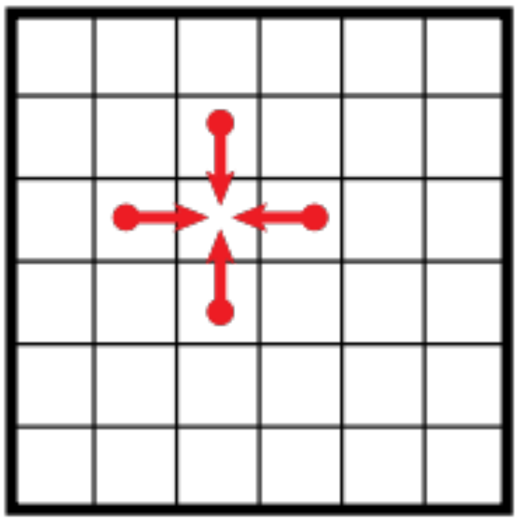
Stencil Pattern

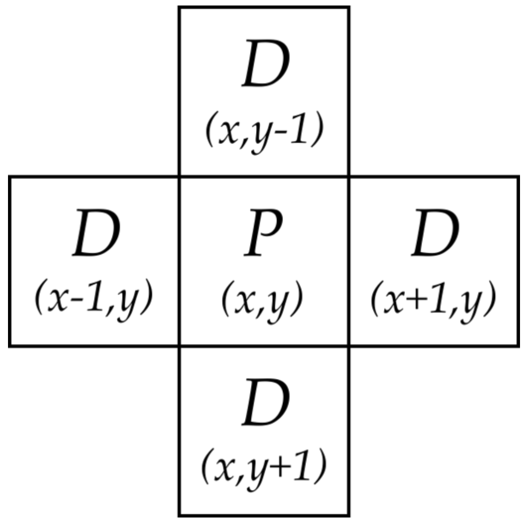
- A stencil pattern is a map where each output depends on a “neighborhood” of inputs
- These inputs are a set of fixed offsets relative to the output position
- A stencil output is a function of a “neighborhood” of elements in an input collection
  - Applies the stencil to select the inputs
- Data access patterns of stencils are regular
  - Stencil is the “shape” of “neighborhood”
  - Stencil remains the same

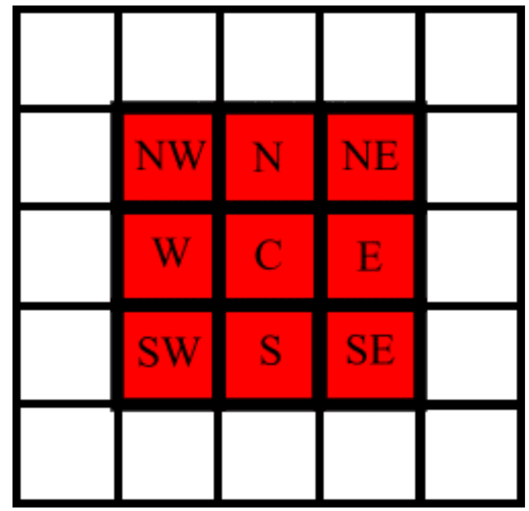
Stencil Patterns

- Stencils can operate on one dimensional and multidimensional data
- Stencil neighborhoods can range from compact to sparse, square to cube, and anything else!
- It is the pattern of the stencil that determines how the stencil operates in an application

2-Dimensional Stencils







4-point stencil

Center cell (P) is not used

5-point stencil

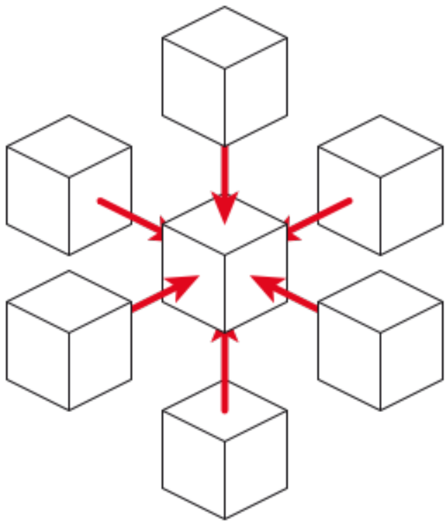
Center cell (P) is used as well

9-point stencil

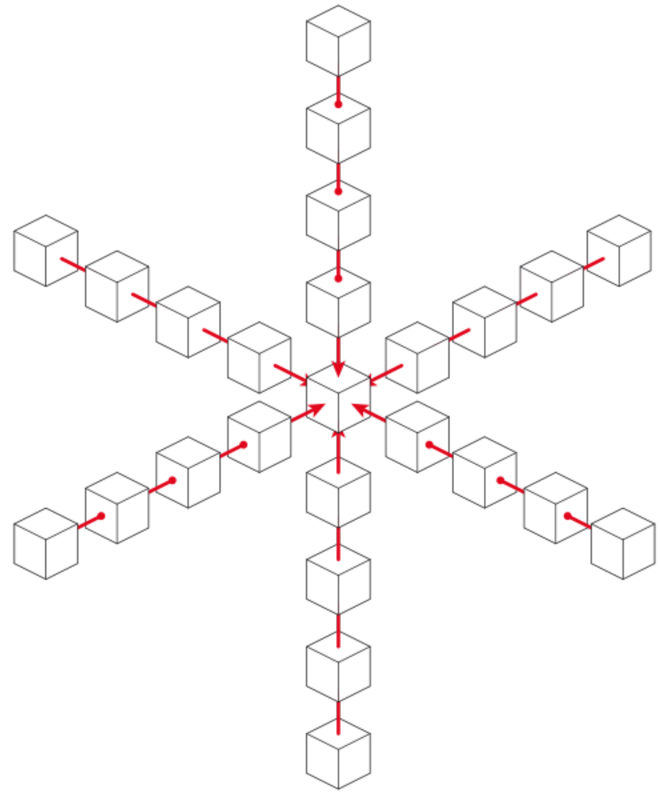
Center cell (C) is used as well

Source: http://en.wikipedia.org/wiki/Stencil_code

3-Dimensional Stencils



6-point stencil
(7-point stencil)



24-point stencil
(25-point stencil)

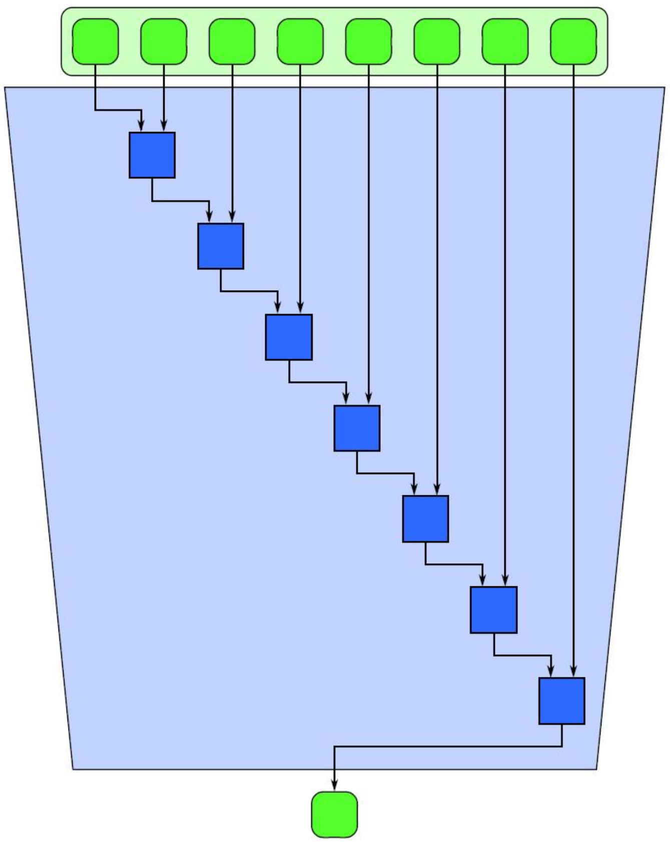
Source: http://en.wikipedia.org/wiki/Stencil_code

Parallel Control Patterns: Reduction

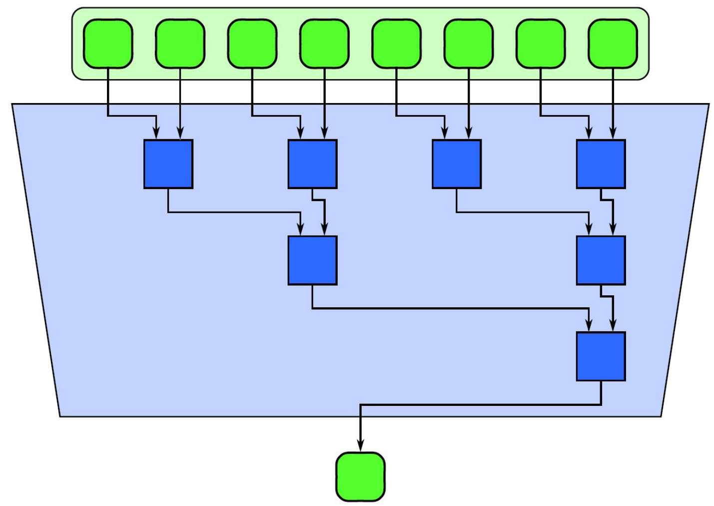
- **Reduction**: Combines every element in a collection using an associative "combiner function"
- Because of the associativity of the combiner function, different orderings of the reduction are possible
- Examples of combiner functions: addition, multiplication, maximum, minimum, and Boolean AND, OR, and XOR

Parallel Control Patterns: Reduction

Serial Reduction



Parallel Reduction

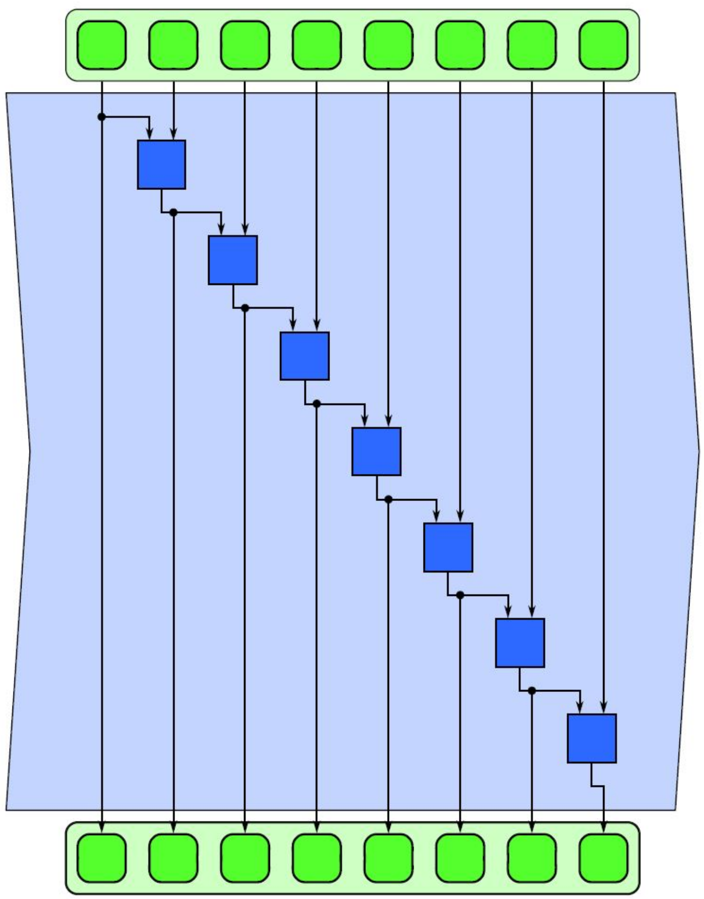


Parallel Control Patterns: Scan

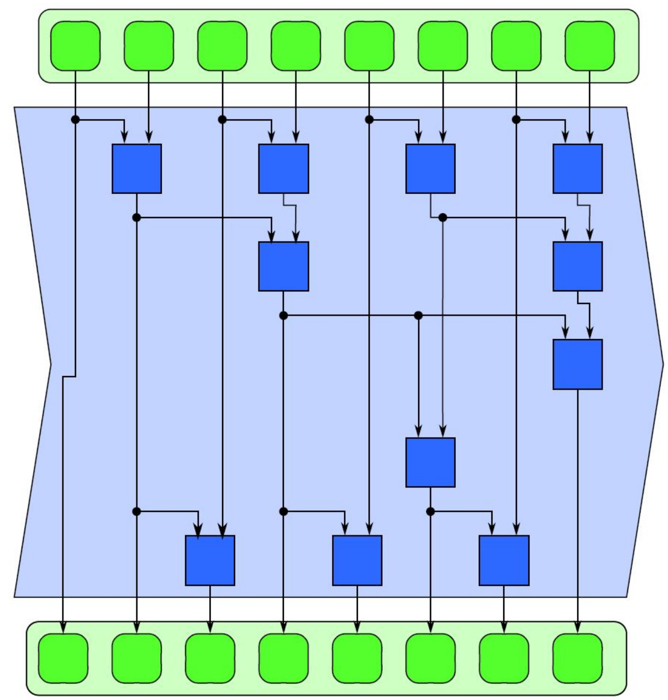
- **Scan**: computes all partial reduction of a collection
- For every output in a collection, a reduction of the input up to that point is computed
- If the function being used is associative, the scan can be parallelized
- Parallelizing a scan is not obvious at first, because of dependencies to previous iterations in the serial loop
- A parallel scan will require more operations than a serial version

Parallel Control Patterns: Scan

Serial Scan



Parallel Scan



Algorithmic Structures

Algorithmic Structures

Organized by task, data, or dataflow

- **By tasks**: Independent Task Execution, Aggregation of Tasks, Recursive Tasks, Static Task Scheduling, Dynamic Task Scheduling, Dataflow Task Scheduling ...
- **By data decomposition**: Static distribution pattern, Redistribution pattern, Irregular Distribution Pattern, Oversubscription pattern ...
- **By dataflow**: Pipeline pattern, Event-based coordination pattern

Algorithm Structure Design Space

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

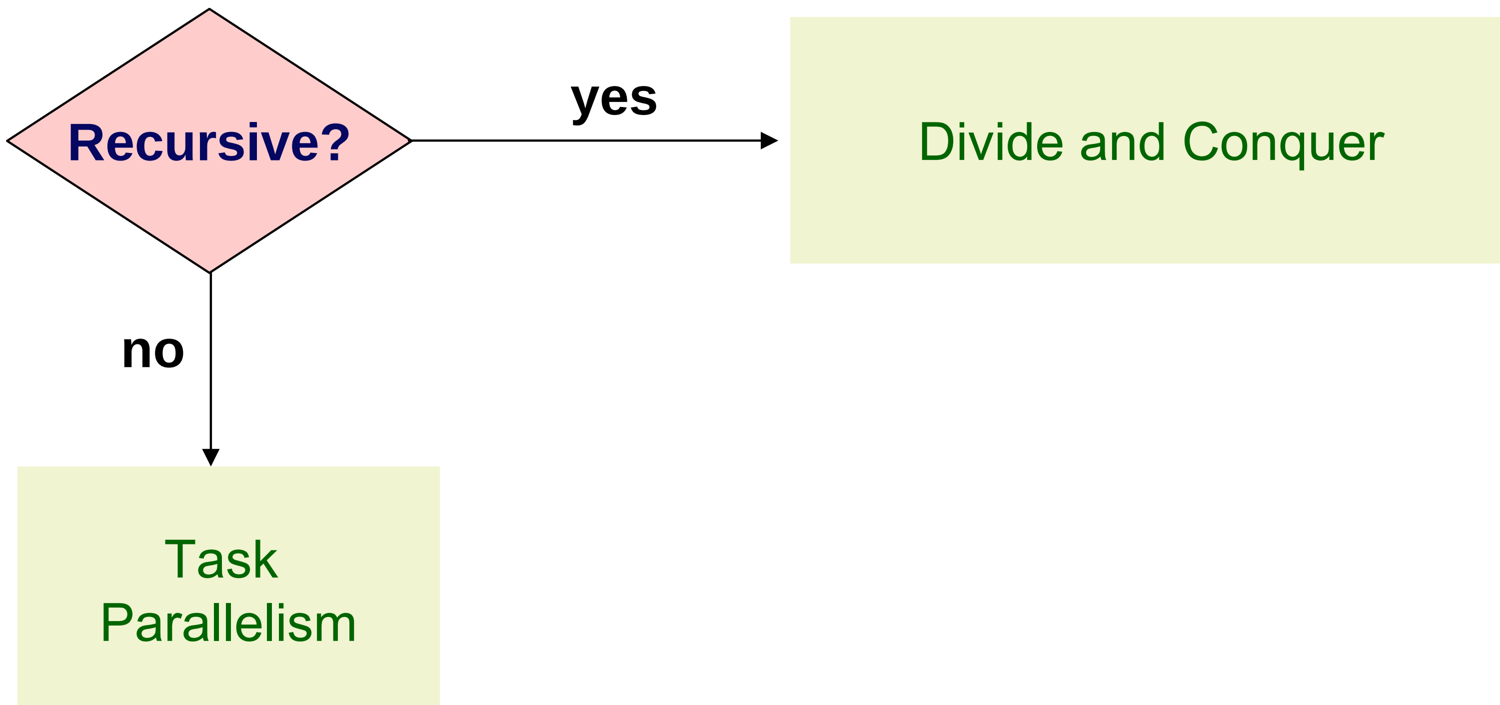
- Important considerations
  - Magnitude of number of execution units platform will support
  - Cost of sharing information among execution units
  - Avoid tendency to over constrain the implementation
    - Work well on the intended platform
    - Flexible enough to easily adapt to different architectures

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?

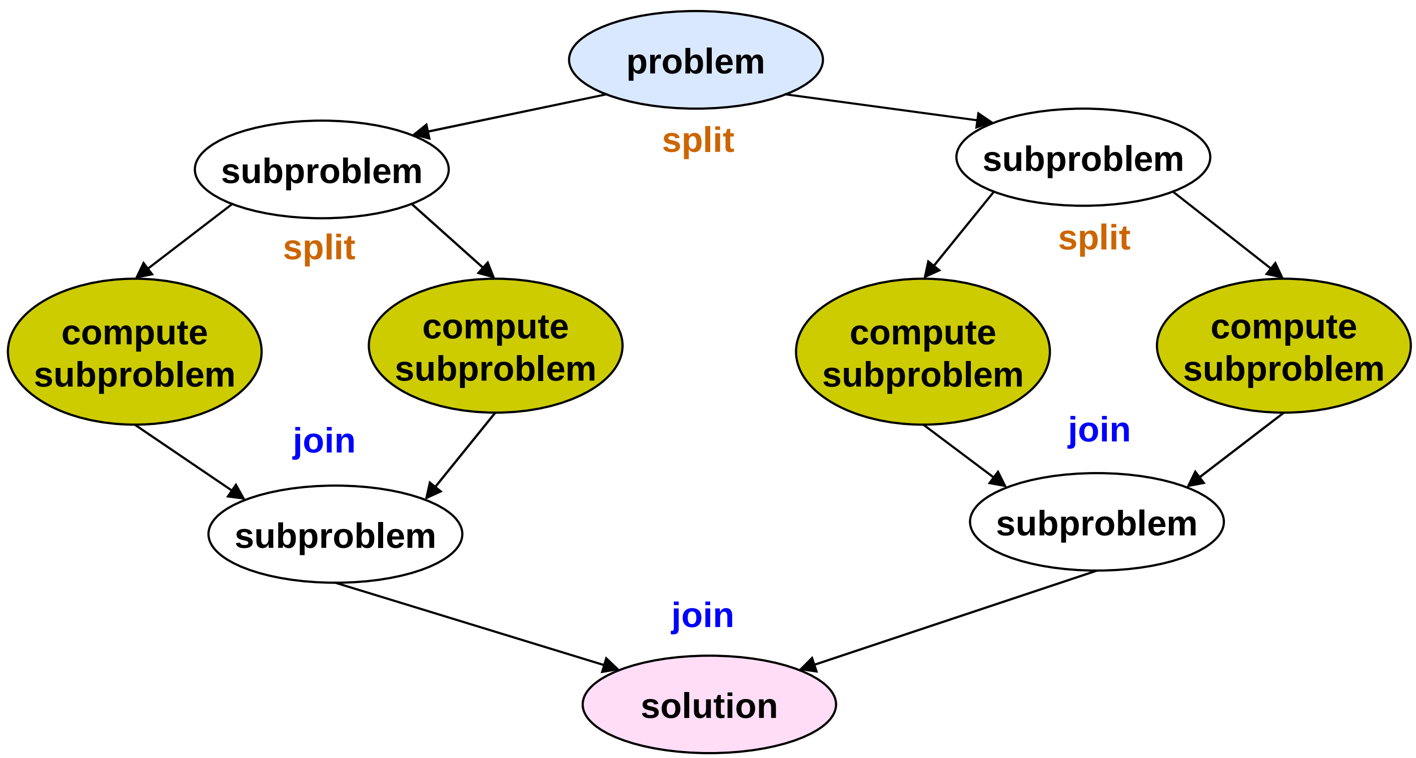


Task Parallelism

- Ray tracing
  - Computation for each ray is a separate and independent
- 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

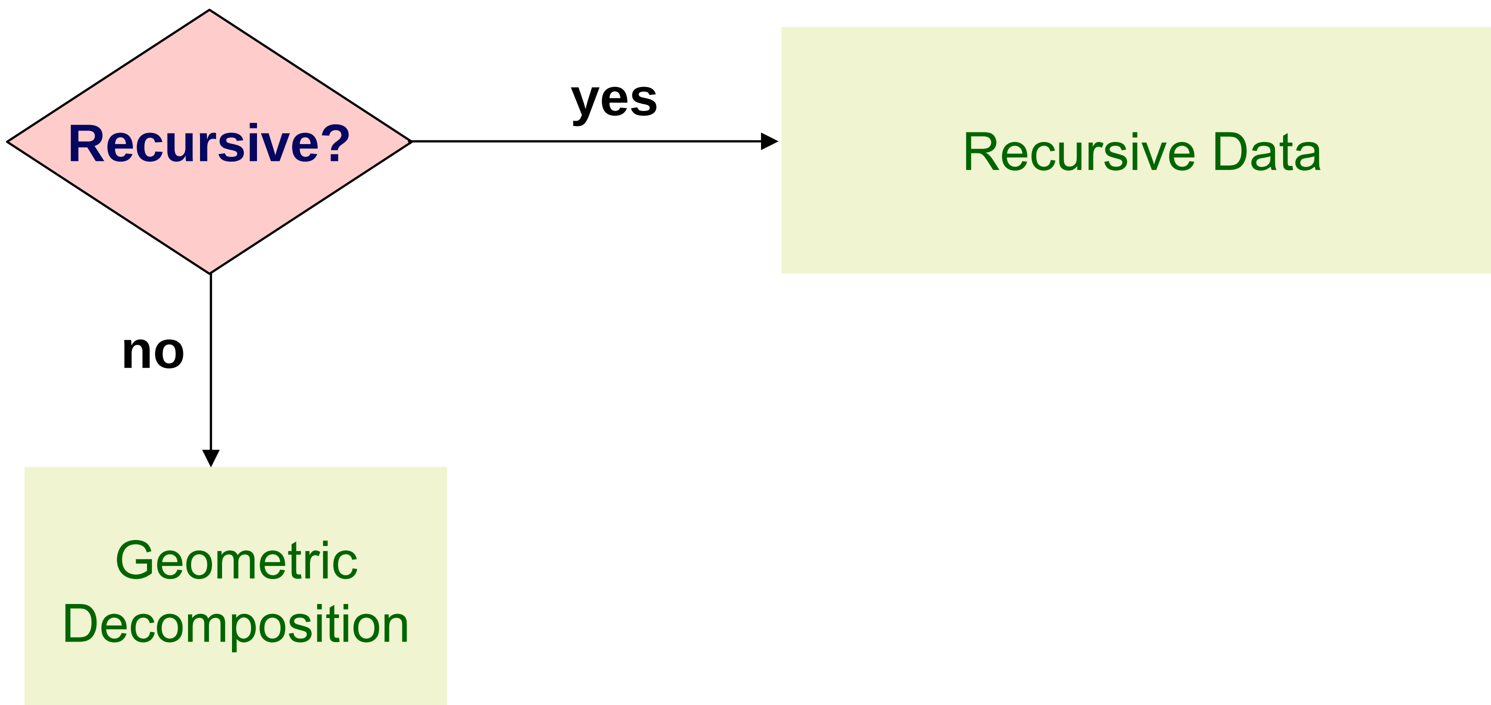
Divide and Conquer

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



Organize by Data?

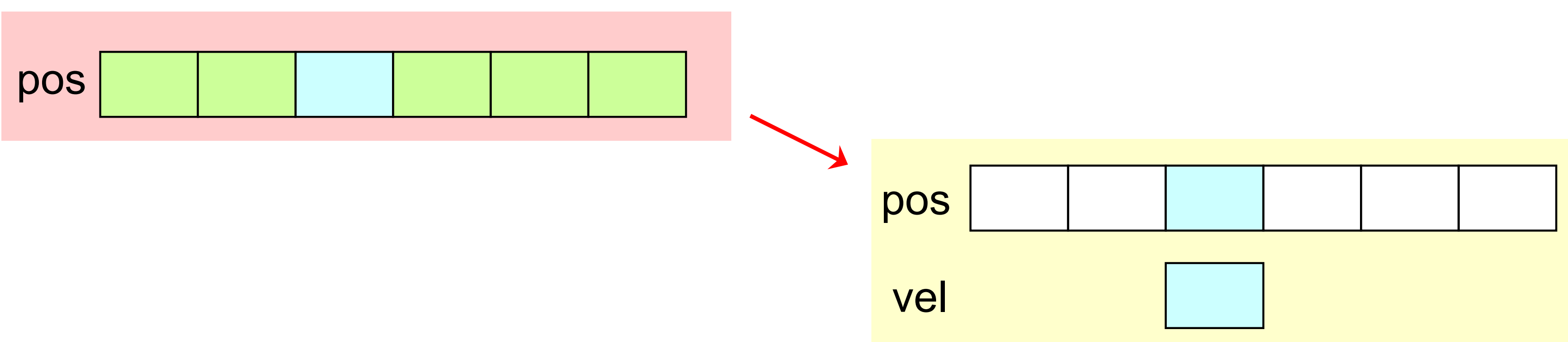
- Operations on a central data structure
  - Arrays and linear data structures
  - Recursive data structures



Geometric Decomposition

- Gravitational body simulator
  - Calculate force between pairs of objects and update accelerations

```
VEC3D acc[NUM_BODIES] = 0;

for (i = 0; i < NUM_BODIES - 1; i++) {
  for (j = i + 1; j < NUM_BODIES; j++) {
    // Displacement vector
    VEC3D d = pos[j] – pos[i];
    // Force
    t = 1 / sqr(length(d));
    // Components of force along displacement
    d = t * (d / length(d));

    acc[i] += d * mass[j];
    acc[j] += -d * mass[i];
  }
}
```



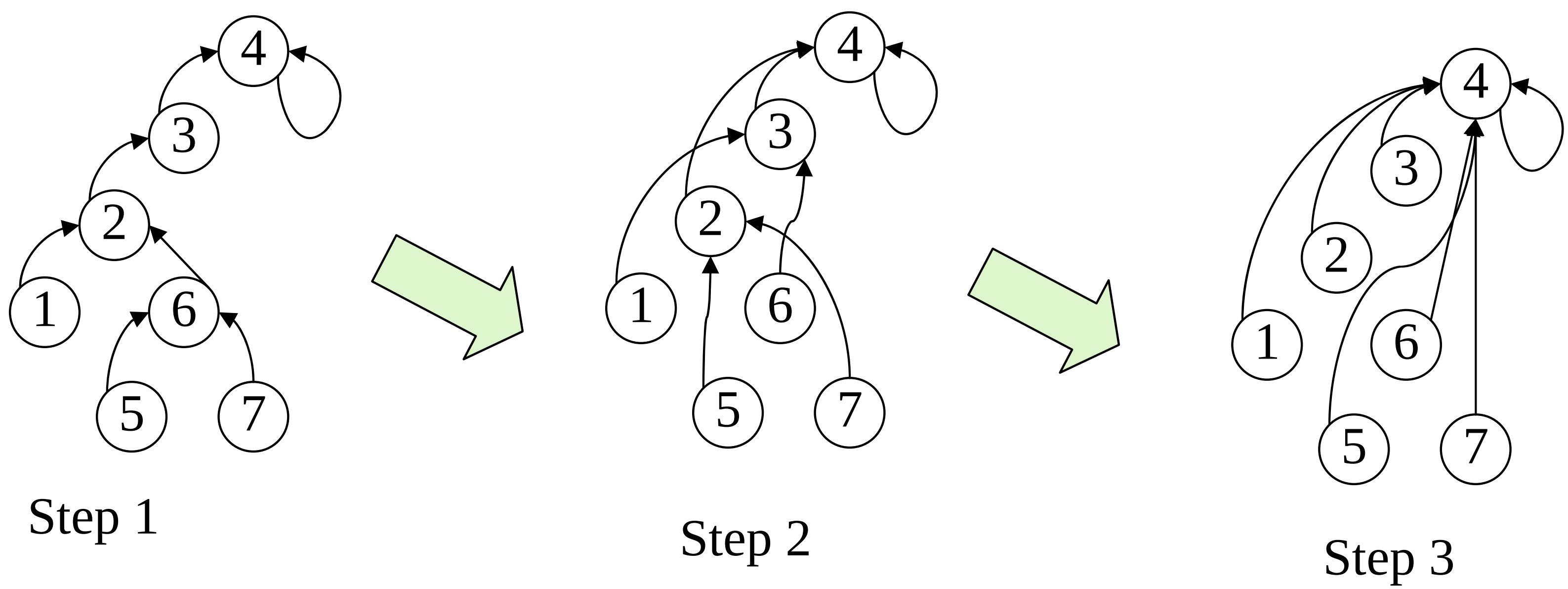
Recursive Data

- 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)



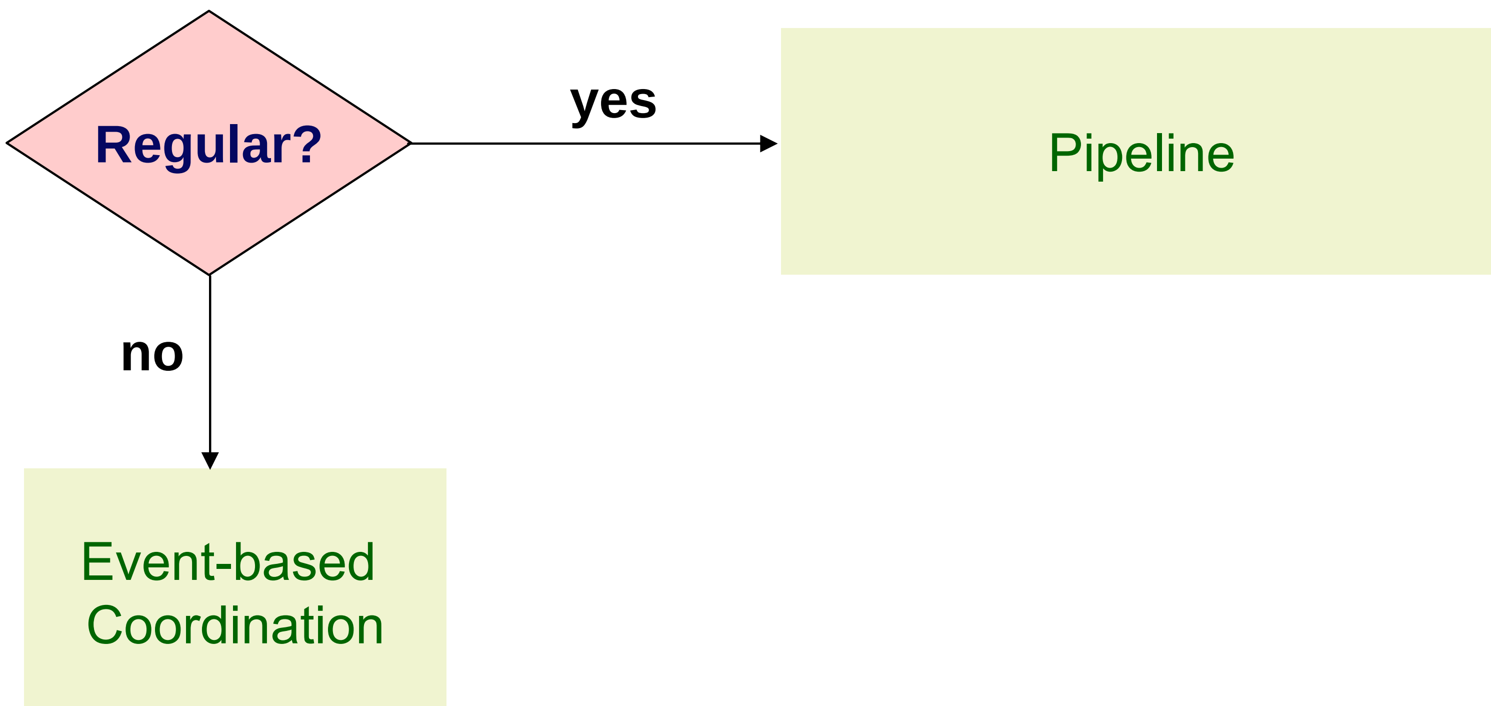
Work vs. Concurrency Tradeoff

- 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

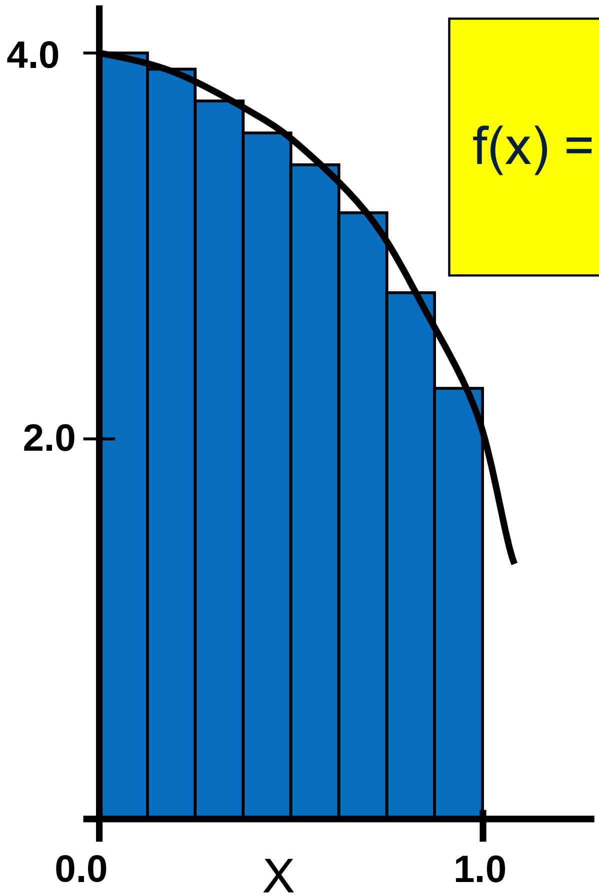
- In this pattern, interaction of tasks to process data can vary over unpredictable intervals
- Deadlocks are likely for applications that use this pattern

Implementation Concepts

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

Example: Parallel Numerical Integration

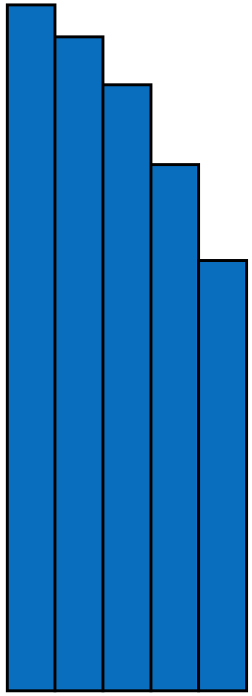


$$f(x) = \frac{4.0}{(1+x^2)}$$

```
static long num_steps = 100000;

void main()
{
    int i;
    double pi, x, step, sum = 0.0;

    step = 1.0 / (double) num_steps;
    for (i = 0; i < num_steps; i++){
        x = (i + 0.5) ∗ step;
        sum = sum + 4.0 / (1.0 + x∗x);
    }

    pi = step ∗ sum;
    printf("Pi = %f\n", pi);
}
```

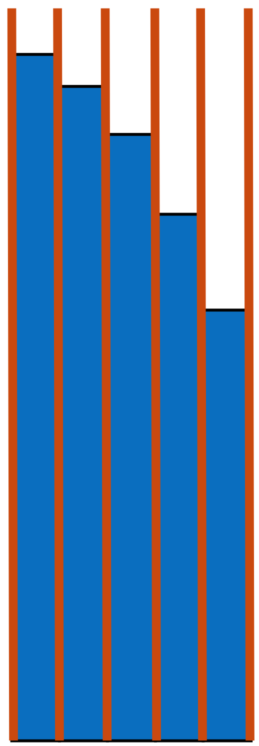
Computing Pi With Integration (MPI)

```
static long num_steps = 100000;

void main(int argc, char* argv[])
{
   int i_start, i_end, i, myid, numprocs;
   double pi, mypi, x, step, sum = 0.0;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);

   MPI_BCAST(&num_steps, 1, MPI_INT, 0, MPI_COMM_WORLD);

   i_start = my_id * (num_steps/numprocs)
   i_end = i_start + (num_steps/numprocs)

   step = 1.0 / (double) num_steps;

   for (i = i_start; i < i_end; i++) {
        x = (i + 0.5) * step
        sum = sum + 4.0 / (1.0 + x*x);
   }
   mypi = step * sum;

   MPI_REDUCE(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   if (myid == 0)
        printf("Pi = %f\n", pi);

   MPI_Finalize();
}
```



Block vs. Cyclic Work Distribution

```
static long num_steps = 100000;

void main(int argc, char* argv[])
{
   int i_start, i_end, i, myid, numprocs;
   double pi, mypi, x, step, sum = 0.0;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);

   MPI_BCAST(&num_steps, 1, MPI_INT, 0, MPI_COMM_WORLD);

   i_start = my_id * (num_steps/numprocs)
   i_end = i_start + (num_steps/numprocs)

   step = 1.0 / (double) num_steps;

   for (i = myid; i < num_steps; i += numprocs) {
        x = (i + 0.5) * step
        sum = sum + 4.0 / (1.0 + x*x);
   }
   mypi = step * sum;

   MPI_REDUCE(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   if (myid == 0)
        printf("Pi = %f\n", pi);

   MPI_Finalize();
}
```



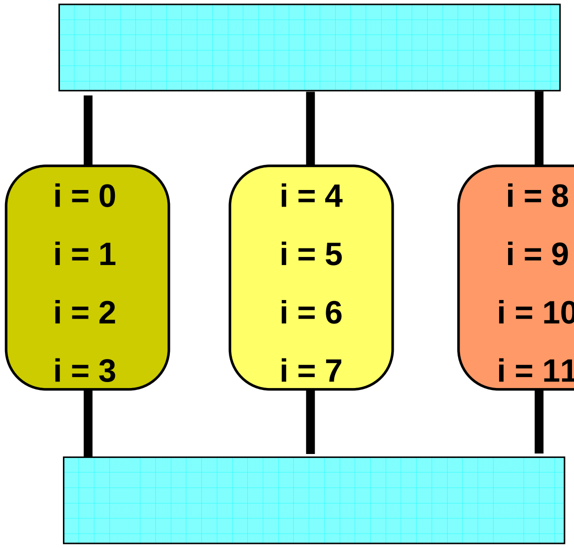
SPMD Challenges

- 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

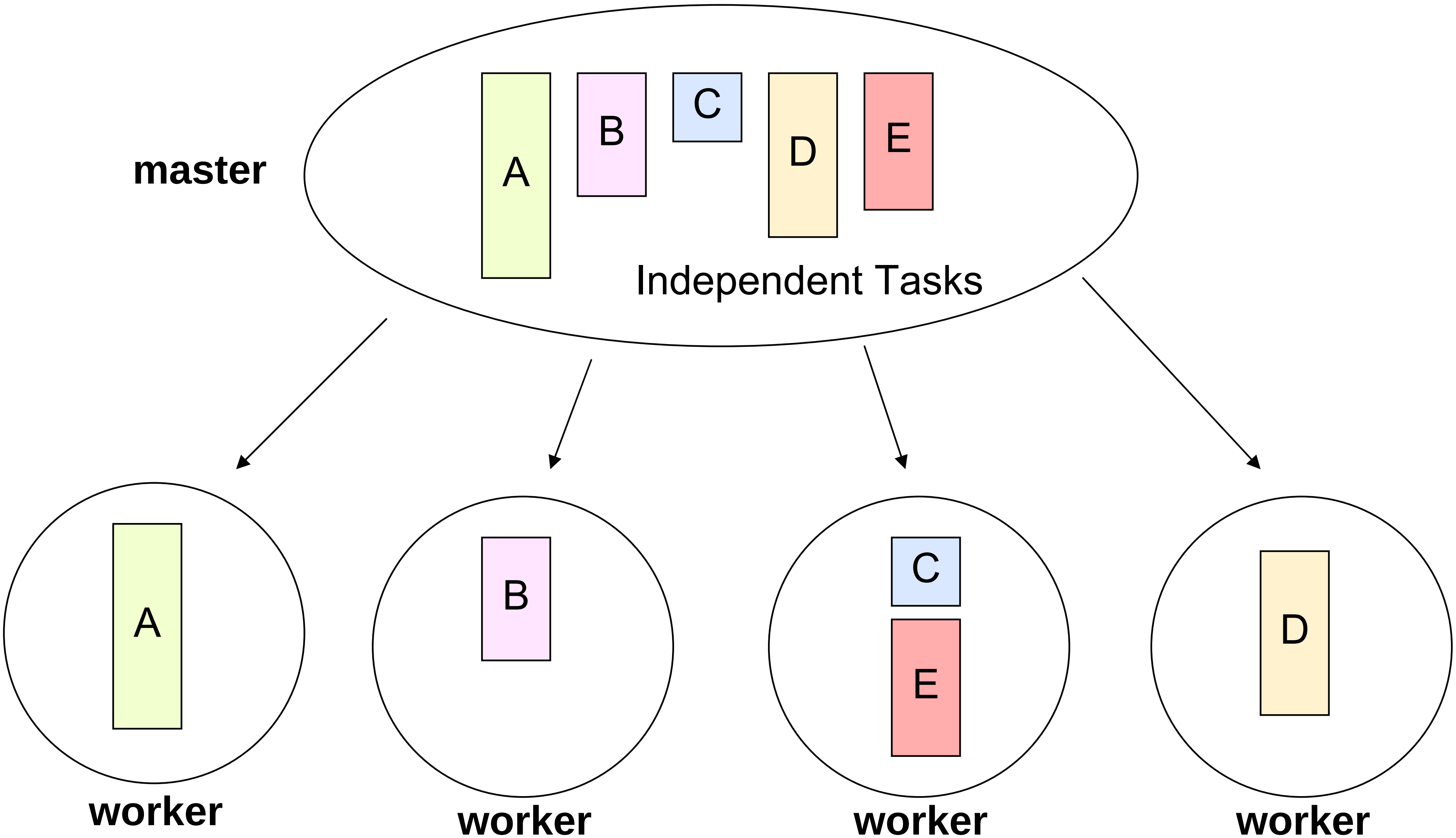
Loop Parallelism Pattern

- 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

Types of Parallel Programs

- Flavors of parallelism
  - Data parallelism
    - all processors do same thing on different data
  - Task parallelism
    - processors are assigned tasks that do different things
- Parallel execution models
  - Data parallel
  - Pipelining (Producer-Consumer)
  - Task graph
  - Work pool
  - Master-Worker

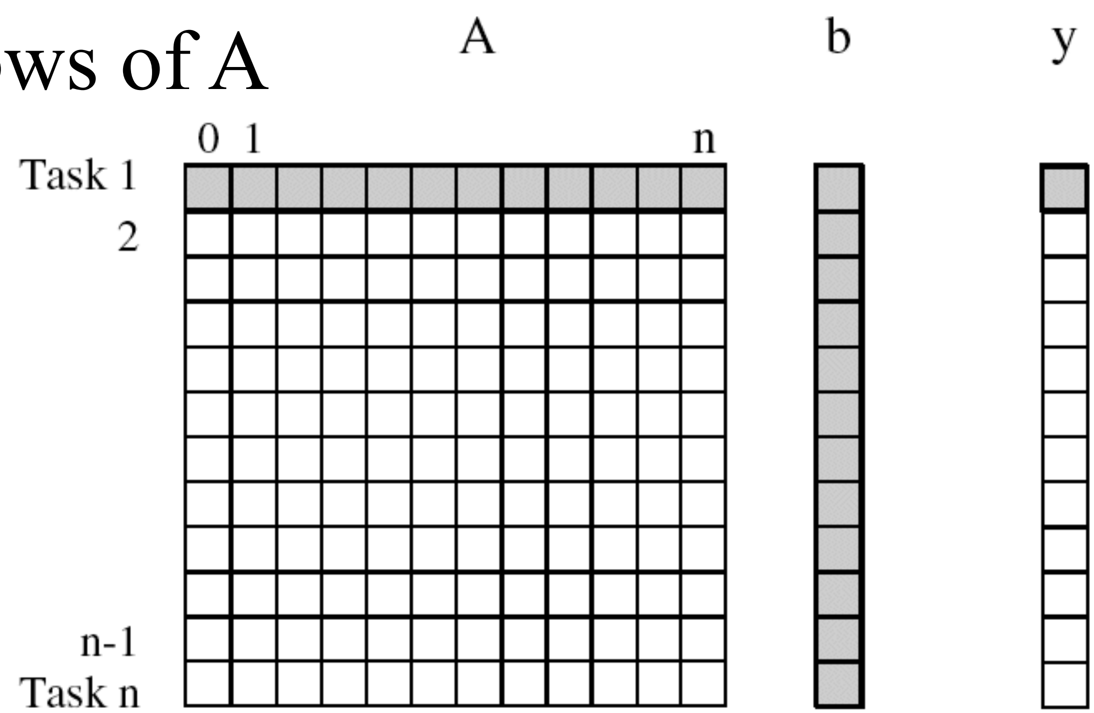
Data Parallel

- Data is decomposed (mapped) onto processors
- Processors performance similar (identical) tasks on data
- Tasks are applied concurrently
- Load balance is obtained through data partitioning
  - Equal amounts of work assigned
- Certainly may have interactions between processors
- Data parallelism scalability
  - Degree of parallelism tends to increase with problem size
  - Makes data parallel algorithms more efficient
- Single Program Multiple Data (SPMD)
  - Convenient way to implement data parallel computation
  - More associated with distributed memory parallel execution

Matrix - Vector Multiplication

- A x b = y
- Allocate tasks to rows of A

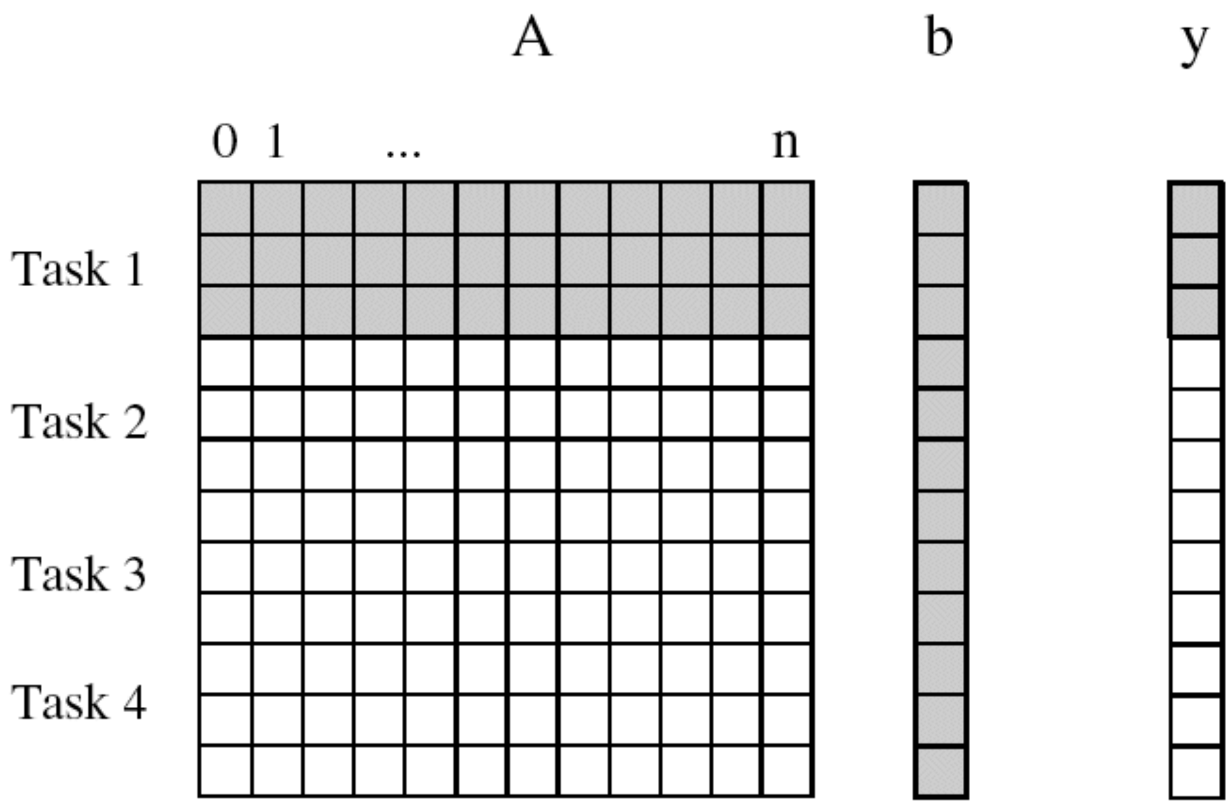
  $$y[i] = \sum_j A[i,j]*b[j]$$



- Dependencies?
- Speedup?
- Computing each element of y can be done independently

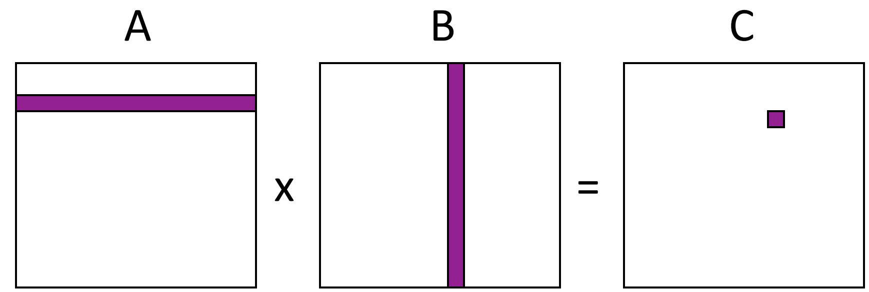
Matrix-Vector Multiplication (Limited Tasks)

- Suppose we only have 4 tasks
- Dependencies?
- Speedup?

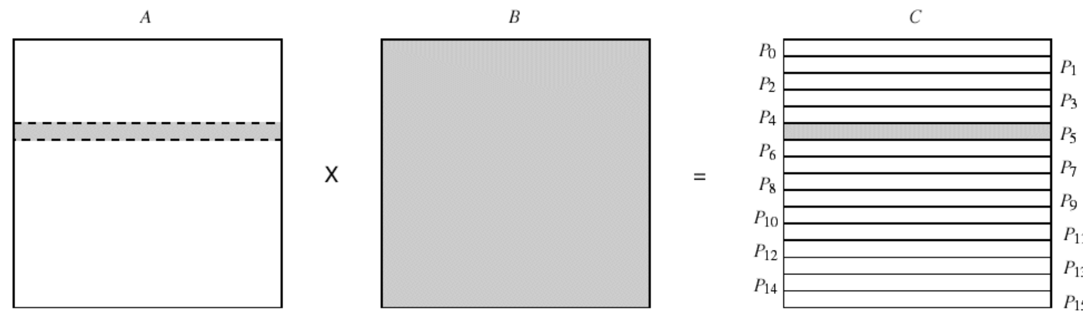


Matrix Multiplication

- A x B = C
- A[i,:] • B[:,j] = C[i,j]

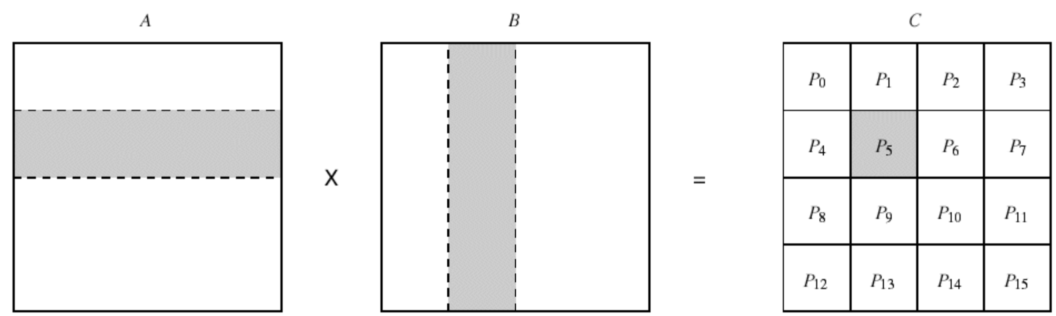


- Row partitioning
  - *N* tasks



(a)

- Block partitioning
  - *N*N/B* tasks



(b)

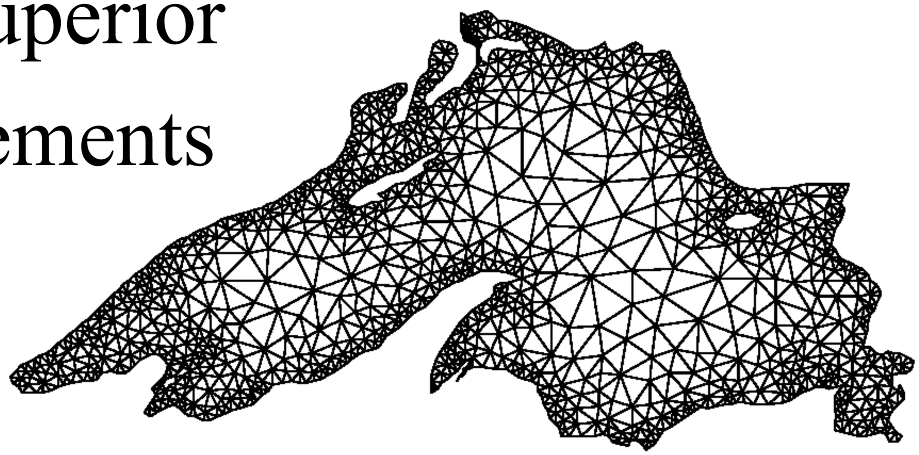
- Shading shows data sharing in B matrix

Granularity of Task and Data Decompositions

- Granularity can be with respect to tasks and data
- Task granularity
  - Equivalent to choosing the number of tasks
  - Fine-grained decomposition results in large # tasks
  - Large-grained decomposition has smaller # tasks
  - Translates to data granularity after # tasks chosen
    - consider matrix multiplication
- Data granularity
  - Think of in terms of amount of data needed in operation
  - Relative to data as a whole
  - Decomposition decisions based on input, output, input-output, or intermediate data

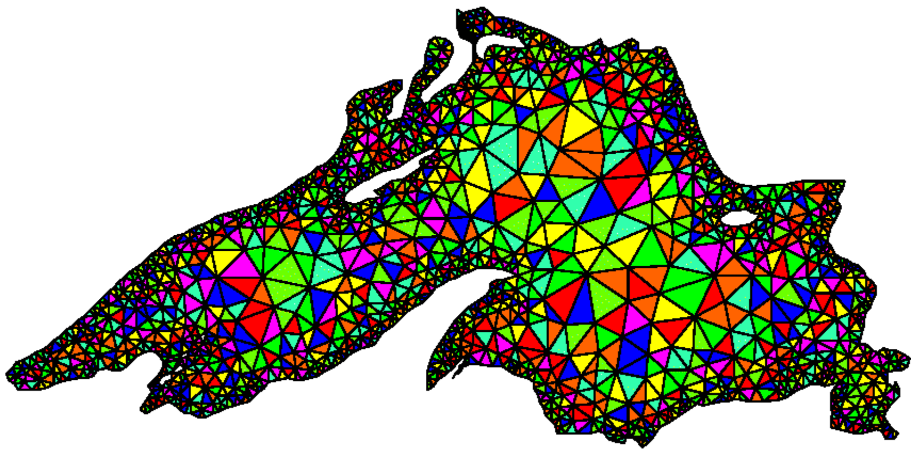
Mesh Allocation to Processors

- Mesh model of Lake Superior
- How to assign mesh elements to processors

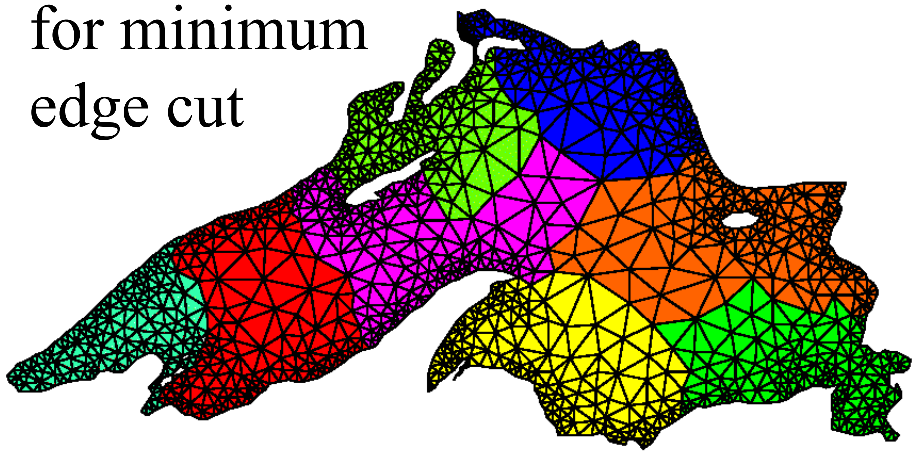


- Distribute onto 8 processors

randomly



graph partitioning for minimum edge cut

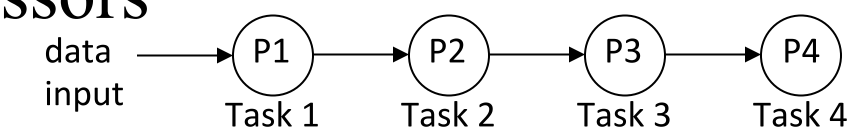


Pipeline Model

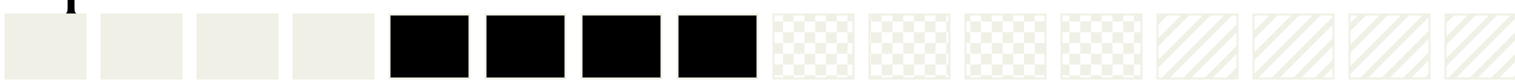
- Stream of data operated on by succession of tasks

  Task 1 Task 2 Task 3 Task 4

  - Tasks are assigned to processors

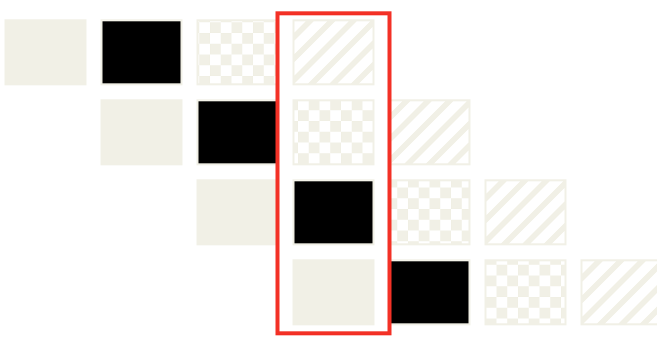


- Consider *N* data units
- Sequential



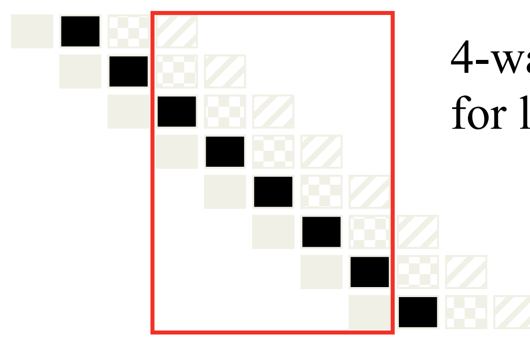
- Parallel (each task assigned to a processor)

4 data units



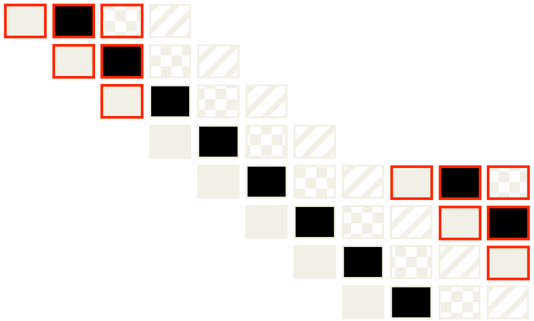
4-way parallel

8 data units



4-way parallel, but for longer time

Pipeline Performance

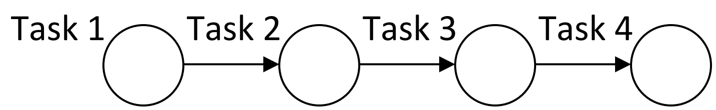


- *N* data and *T* tasks
- Each task takes unit time *t*
- Sequential time = *N*T*t*
- Parallel pipeline time = *start + finish + (N-2T)/T * t*

  = O(N/T)   (for N>>T)
- Try to find a lot of data to pipeline
- Try to divide computation in a lot of pipeline tasks
  - More tasks to do (longer pipelines)
  - Shorter tasks to do
- Pipeline computation is a special form of *producer-consumer* parallelism
  - Producer tasks output data input by consumer tasks

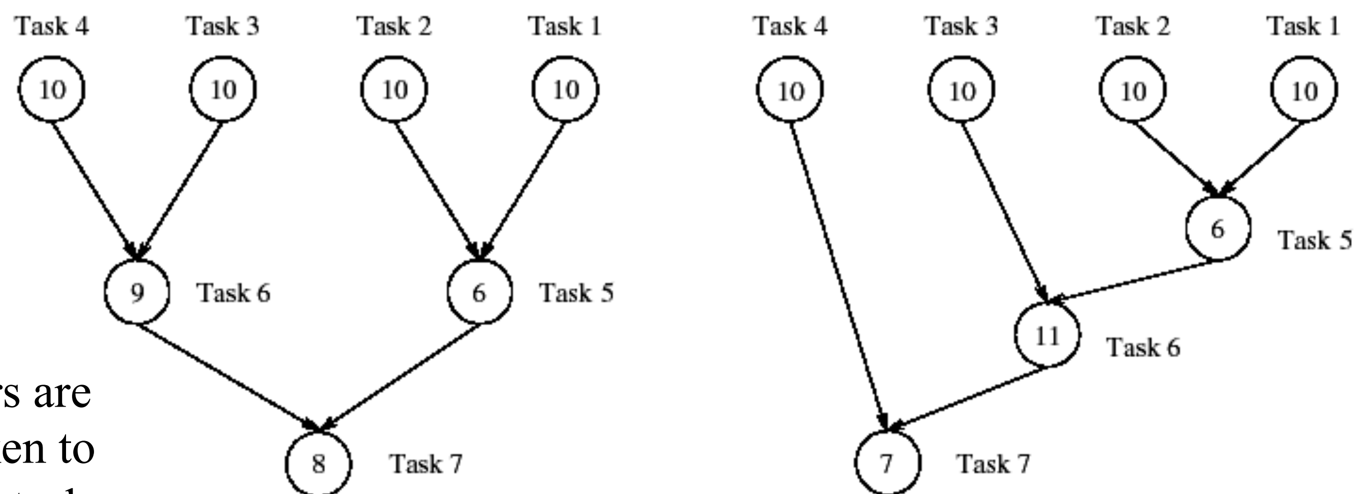
Tasks Graphs

- Computations in any parallel algorithms can be viewed as a task dependency graph
- Task dependency graphs can be non-trivial
  - Pipeline



  - Arbitrary (represents the algorithm dependencies)

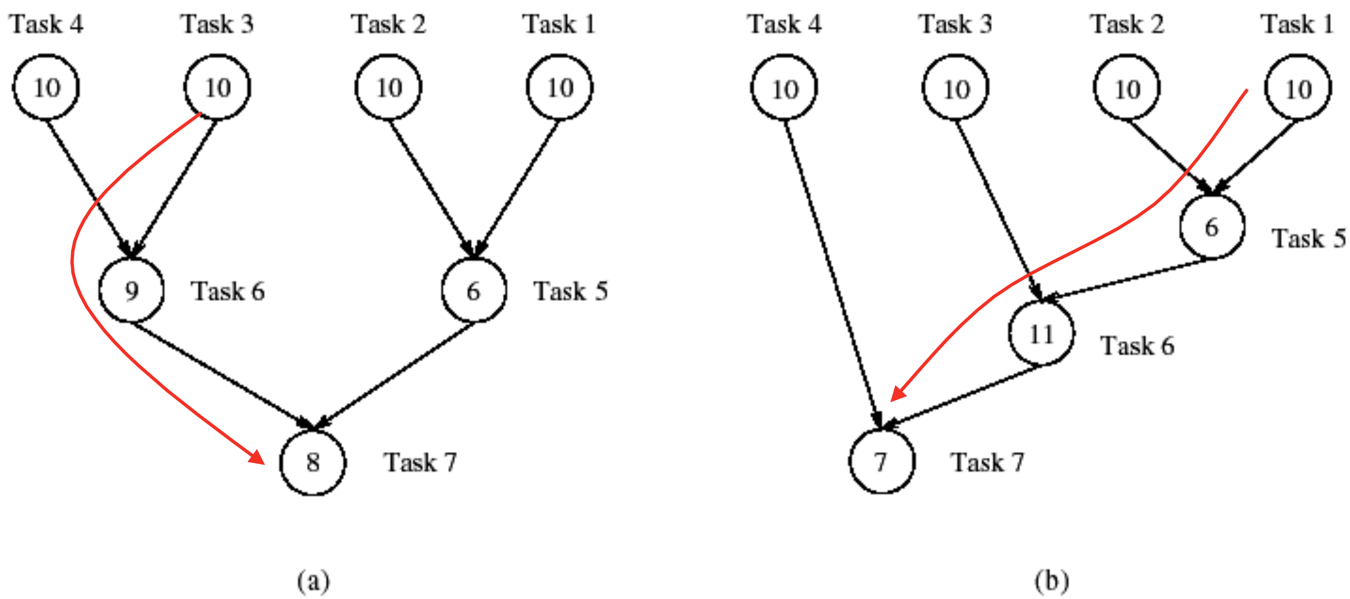
Numbers are time taken to perform task



(a) (b)

Task Graph Performance

- Determined by the *critical path (span)*
  - Sequence of dependent tasks that takes the longest time

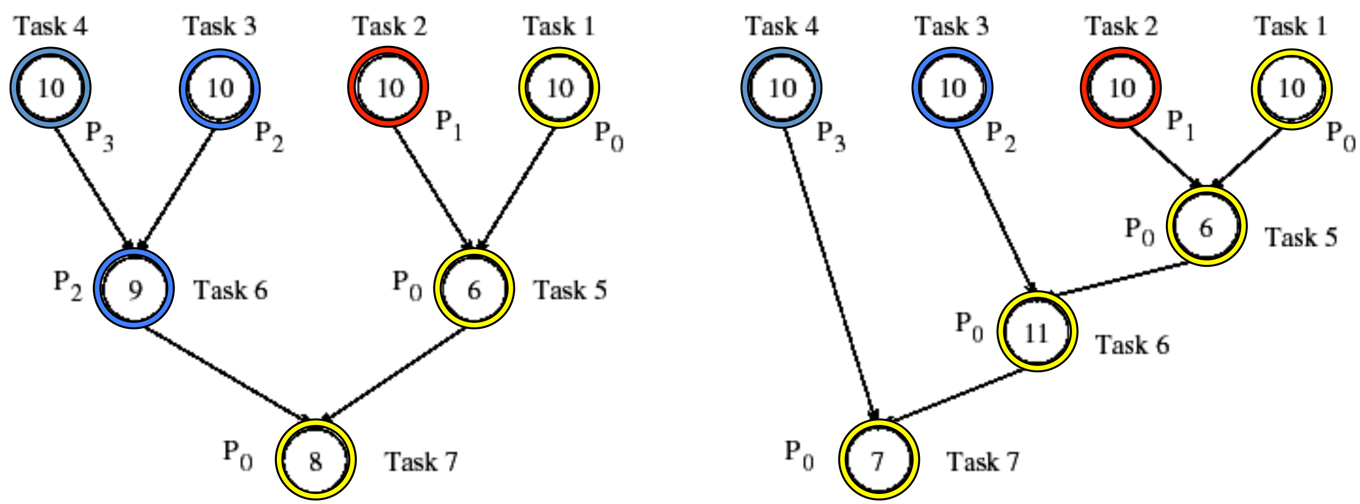


Min time = 27 Min time = 34

  - *Critical path length* bounds parallel execution time

Task Assignment (Mapping) to Processors

- Given a set of tasks and number of processors
- How to assign tasks to processors?
- Should take dependencies into account
- Task mapping will determine execution time

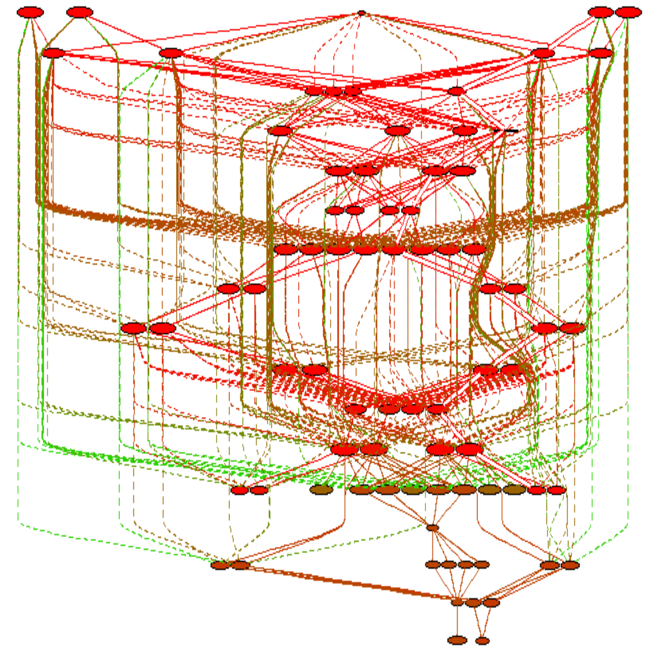


(a) Total time = ?

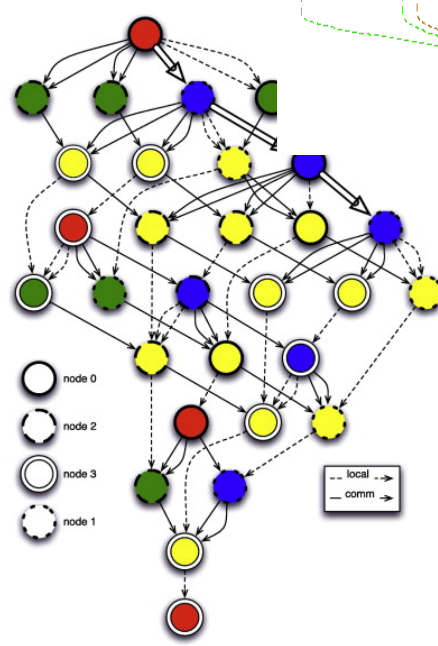
(b) Total time = ?

Task Graphs in Action

Task graph for PDE solver



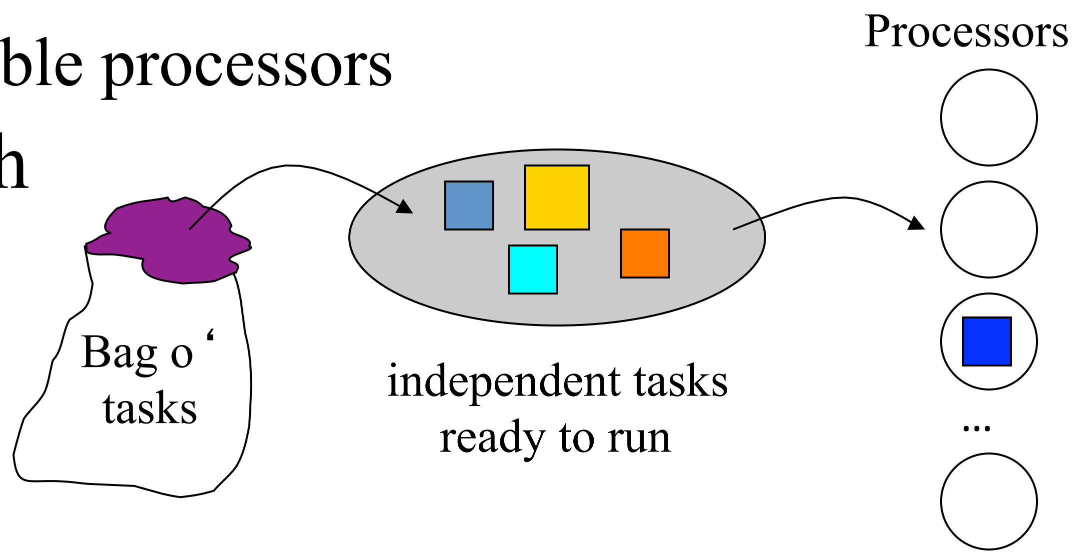
- Uintah task graph scheduler
  - C-SAFE: Center for Simulation of Accidental Fires and Explosions, University of Utah
  - Large granularity tasks
- PLASMA
  - DAG-based parallel linear algebra
  - DAGuE: A generic distributed DAG engine for HPC



DAG of QR for a 4 × 4 tiles matrix on a 2 × 2 grid of processors.

Bag o' Tasks Model and Worker Pool

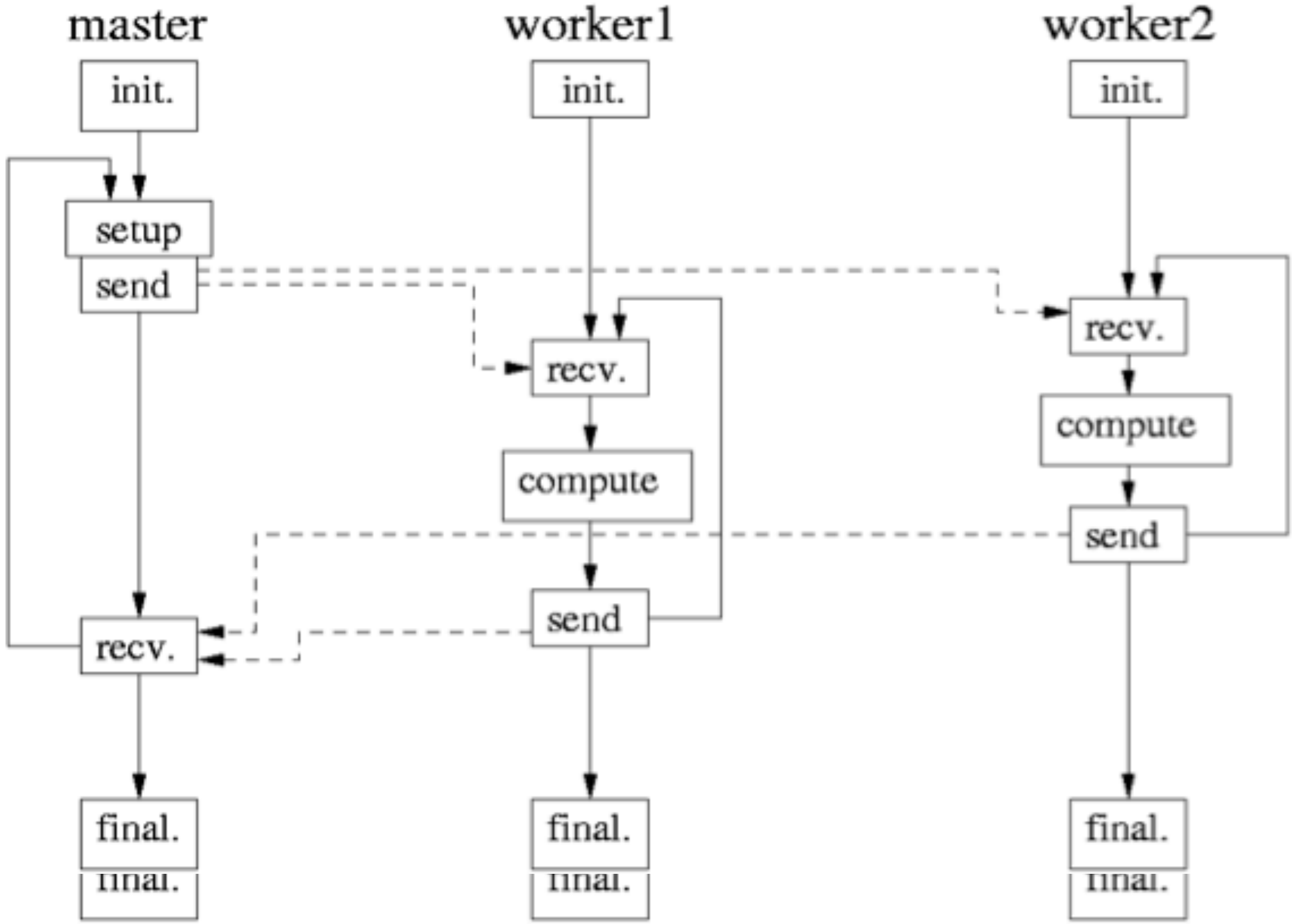
- Set of tasks to be performed
- How do we schedule them?
  - Find independent tasks
  - Assign tasks to available processors
- Bag o' Tasks approach
  - Tasks are stored in a bag waiting to run
  - If all dependencies are satisified, it is moved to a ready to run queue
  - Scheduler assigns a task to a free processor
- Dynamic approach that is effective for load balancing



Master-Worker Parallelism

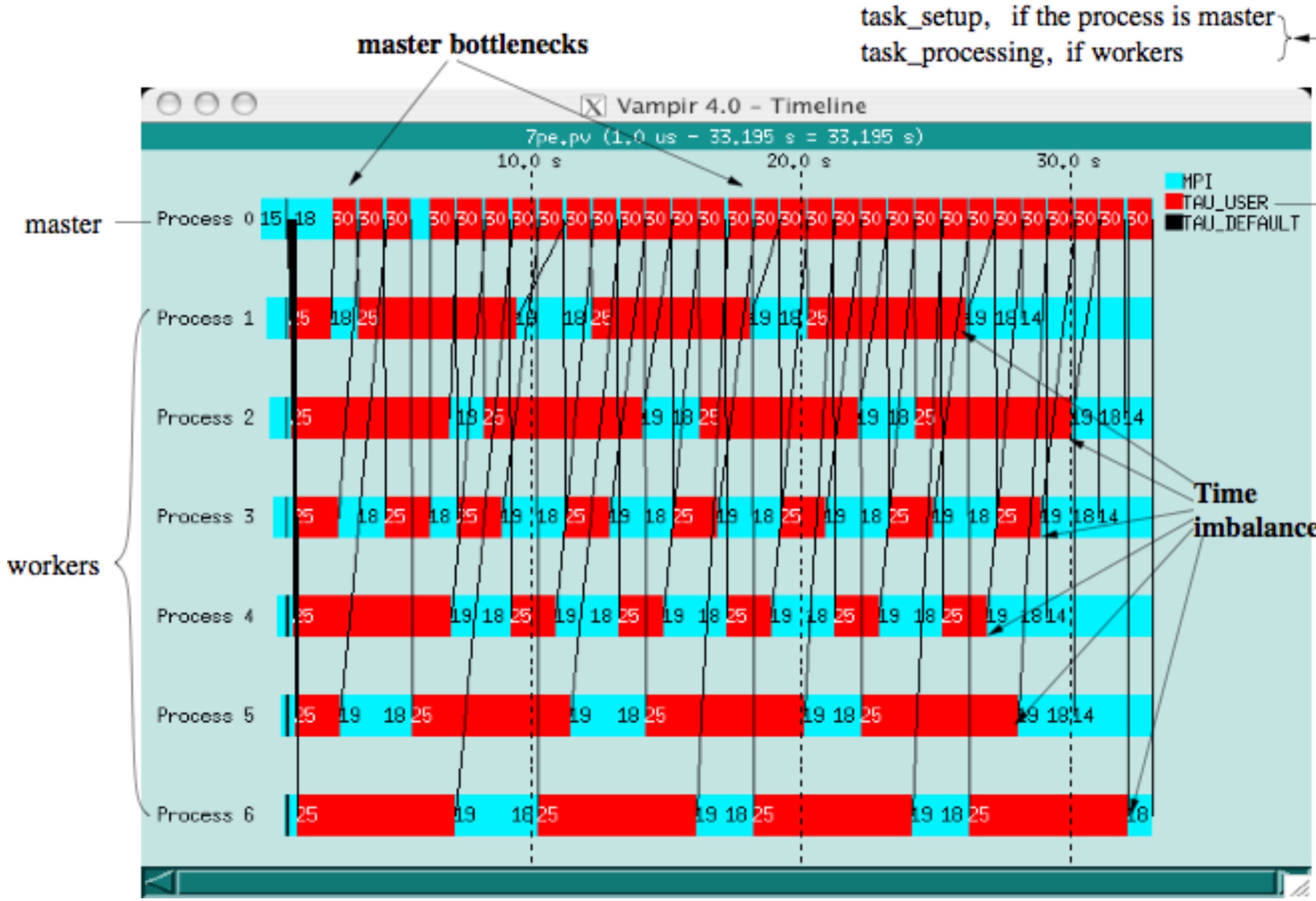
- One or more master processes generate work
- Masters allocate work to worker processes
- Workers idle if have nothing to do
- Workers are mostly stupid and must be told what to do
  - Execute independently
  - May need to synchronize, but most be told to do so
- Master may become the bottleneck if not careful
- What are the performance factors and expected performance behavior
  - Consider task granularity and asynchrony
  - How do they interact?

Master-Worker Execution Model (Li Li)



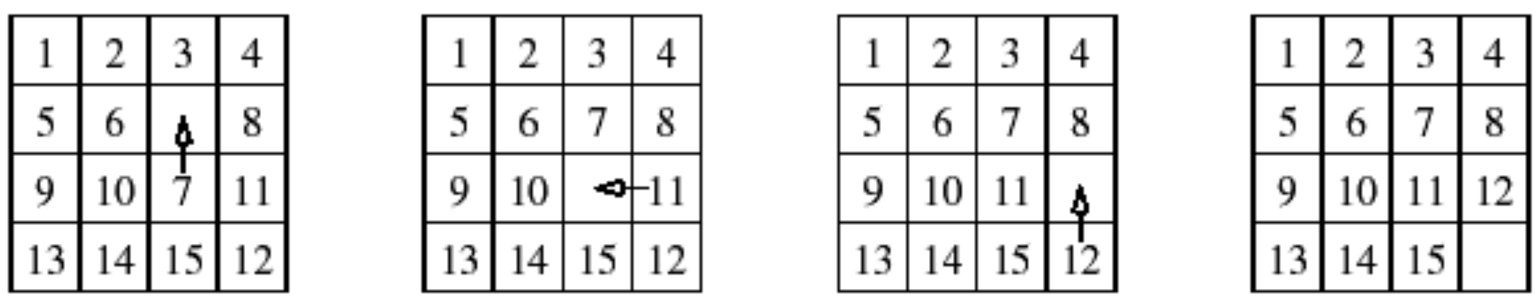
Li Li, “Model-based Automatics Performance Diagnosis of Parallel Computations,” Ph.D. thesis, 2007.

M-W Execution Trace (Li Li)



Search-Based (Exploratory) Decomposition

- 15-puzzle problem
- 15 tiles numbered 1 through 15 placed in 4x4 grid
  - Blank tile located somewhere in grid
  - Initial configuration is out of order
  - Find shortest sequence of moves to put in order



(a) (b) (c) (d)

- Sequential search across space of solutions
  - May involve some heuristics

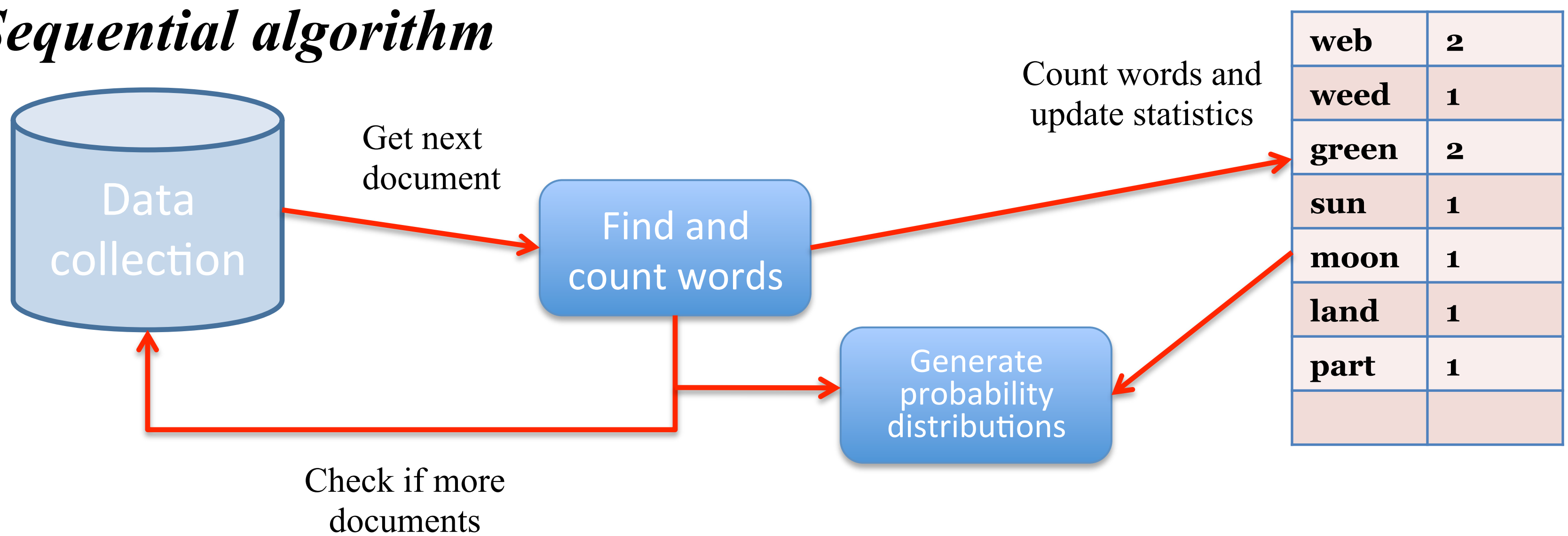
Big-Data and Map-Reduce

- Big-data deals with processing large data sets
- Nature of data processing problem makes it amenable to parallelism
  - Looking for features in the data
  - Extracting certain characteristics
  - Analyzing properties with complex data mining algorithms
- Data size makes it opportunistic for partitioning into large # of sub-sets and processing these in parallel
- We need new algorithms, data structures, and programming models to deal with problems

A Simple Big-Data Problem

- Consider a large data collection of text documents
- Suppose we want to find how often a particular word occurs and determine a probability distribution for all word occurrences

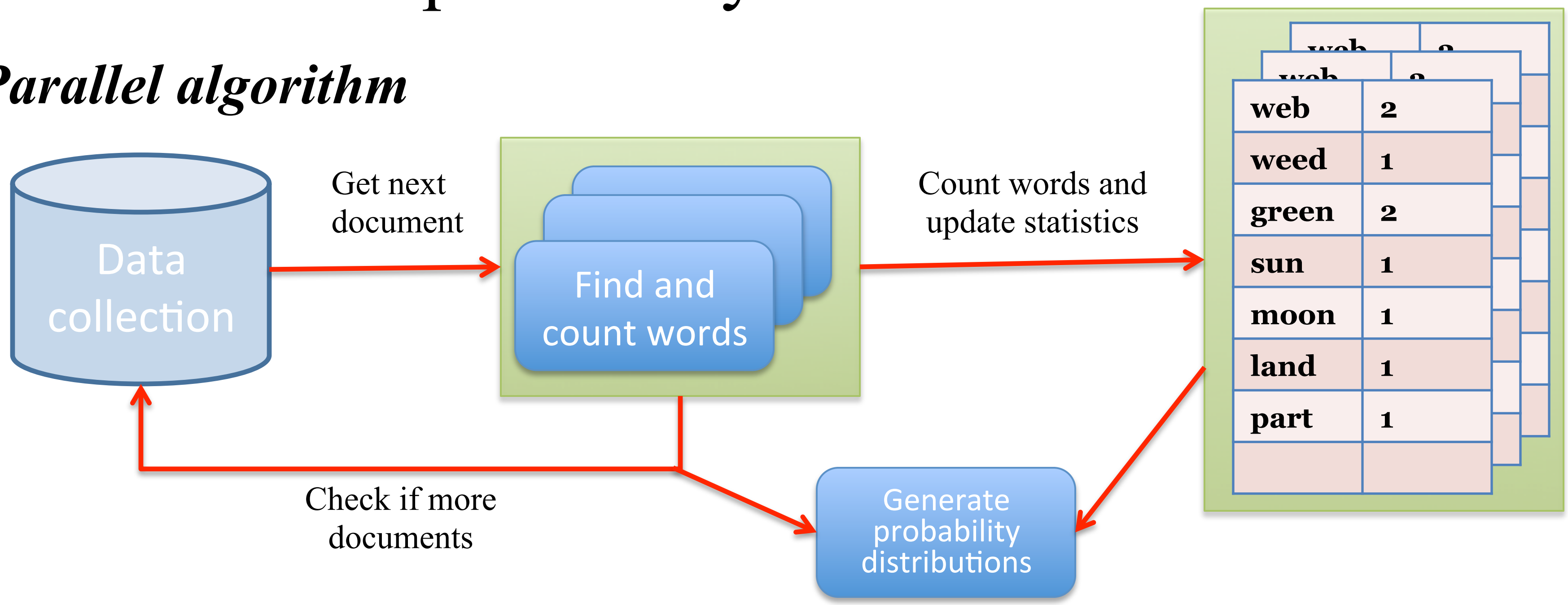
Sequential algorithm



Parallelization Approach

- *Map*: partition the data collection into subsets of documents and process each subset in parallel
- *Reduce*: assemble the partial frequency tables to derive final probability distribution

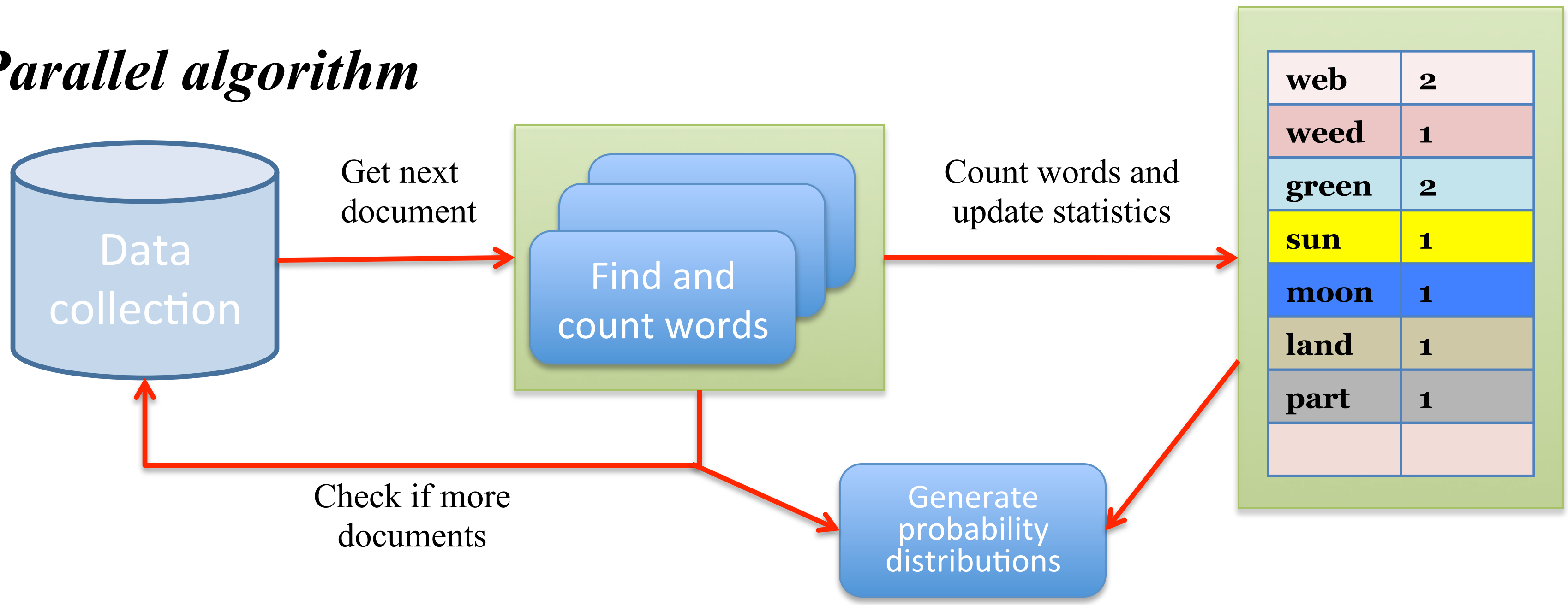
Parallel algorithm



Parallelization Approach

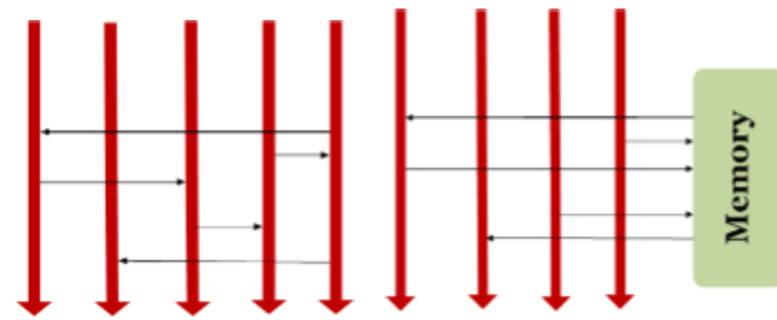
- *Map*: partition the data collection into subsets of documents and process each subset in parallel
- *Reduce*: assemble the partial frequency tables to derive final probability distribution

Parallel algorithm



Actually, it is not easy to parallel….

Different programming models
Message Passing   Shared Memory



Fundamental issues
Scheduling, data distribution, synchronization, inter-process communication, robustness, fault tolerance, …

Architectural issues
Flynn's taxonomy (SIMD, MIMD, etc.), network topology, bisection bandwidth, cache coherence, …

Different programming constructs
Mutexes, conditional variables, barriers, …
masters/slaves, producers/consumers, work queues,. …

Common problems
Livelock, deadlock, data starvation, priority inversion, …dining philosophers, sleeping barbers, cigarette smokers, …

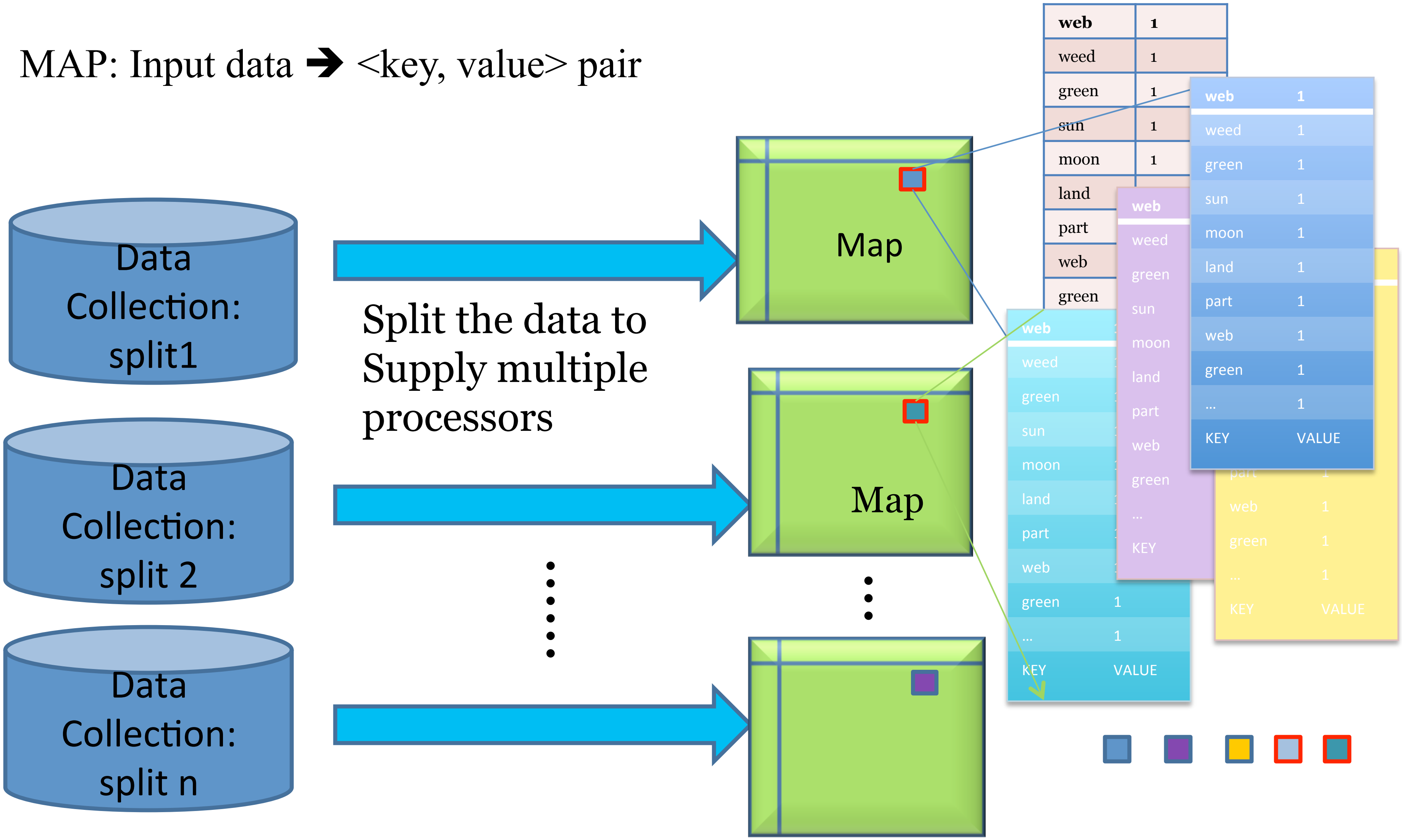
Actually, Programmer's Nightmare….

Map-Reduce Parallel Programming

- Become an important distributed parallel programming paradigm for large-scale applications
  - Also applies to shared-memory parallelism
  - Becomes one of the core technologies powering big IT companies, like Google, IBM, Yahoo and Facebook.
- Framework runs on a cluster of machines and automatically partitions jobs into number of small tasks and processes them in parallel
- Can capture in combining Map and Reduce parallel patterns

Map-Reduce Example

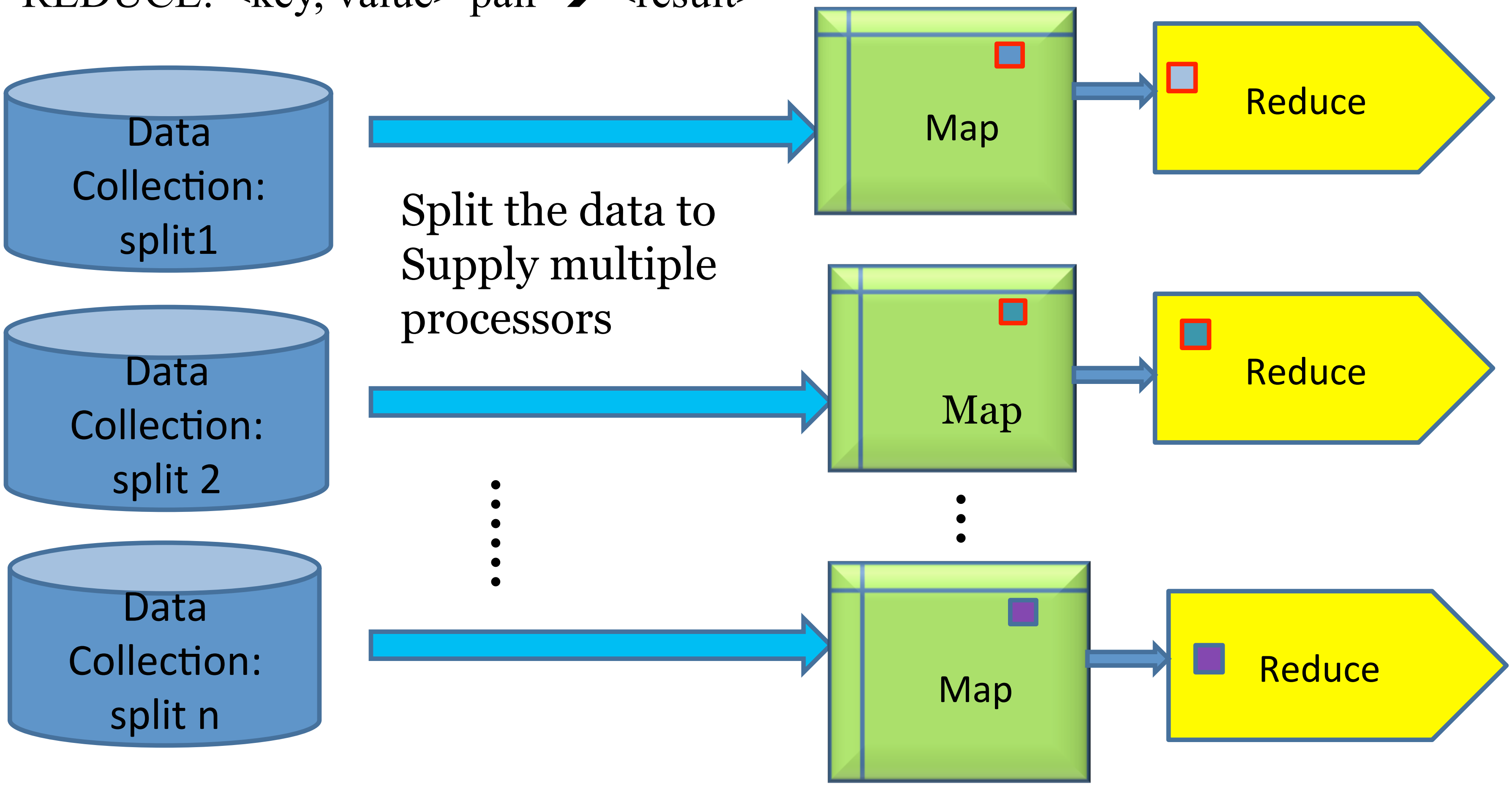
MAP: Input data ➔ <key, value> pair



MapReduce

MAP: Input data ➔ <key, value> pair

REDUCE: <key, value> pair ➔ <result>



Agenda

What is a task? - unit of parallelism
Parallel Patterns - decompose parallelism
Algorithmic Structures - organize parallelism
Implementation Concepts - code parallelism

Algorithms for High-Performance Computing Platforms (2020-2021)

Course 2: Tasks

Laércio LIMA PILLA
pilla@lri.fr