MIT OpenCourseWare
http://ocw.mit.edu

6.189 Multicore Programming Primer, January (IAP) 2007

Please use the following citation format:

> Rodric Rabbah, *6.189 Multicore Programming Primer, January (IAP) 2007*. (Massachusetts Institute of Technology: MIT OpenCourseWare). http://ocw.mit.edu (accessed MM DD, YYYY). License: Creative Commons Attribution-Noncommercial-Share Alike.

Note: Please use the actual date you accessed this material in your citation.

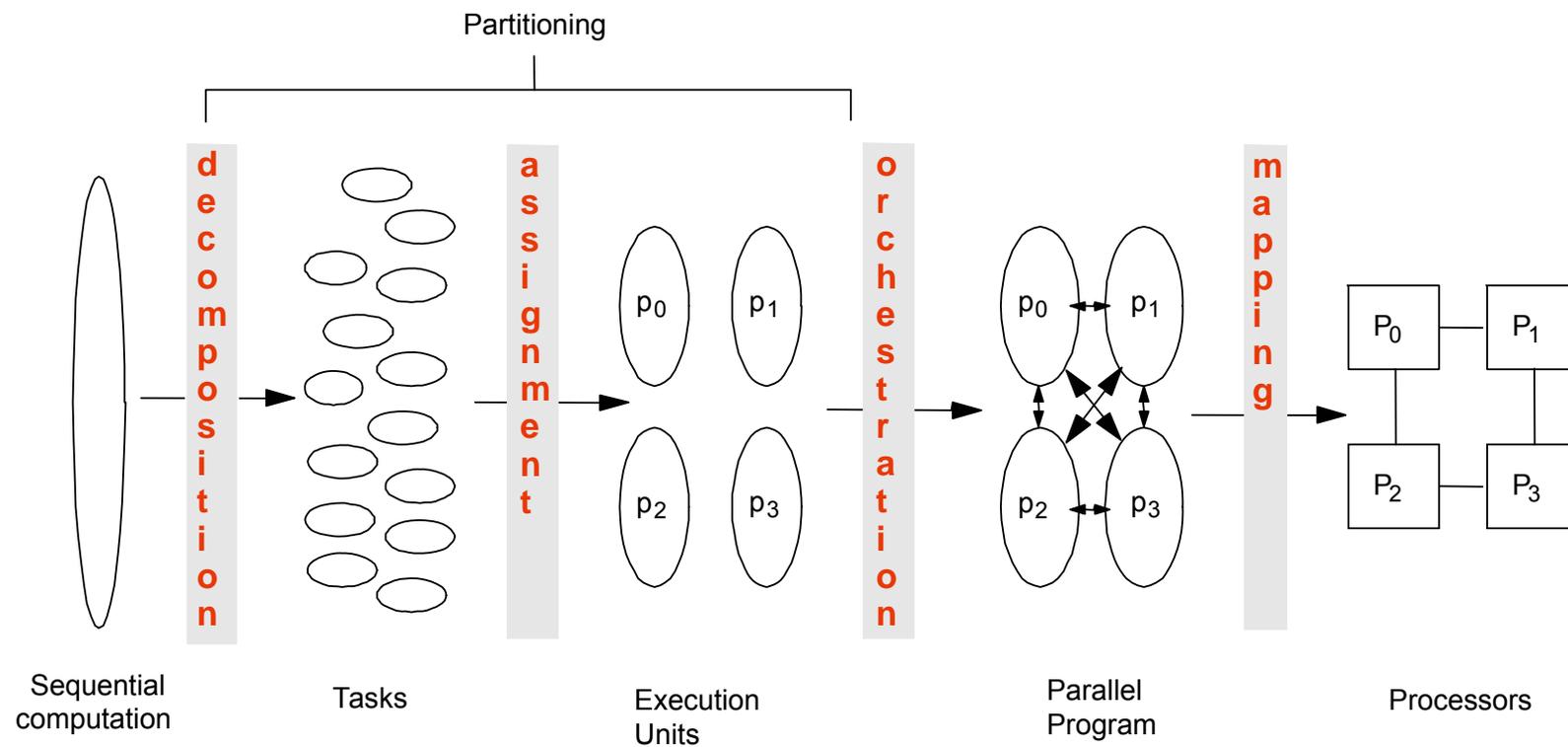For more information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms
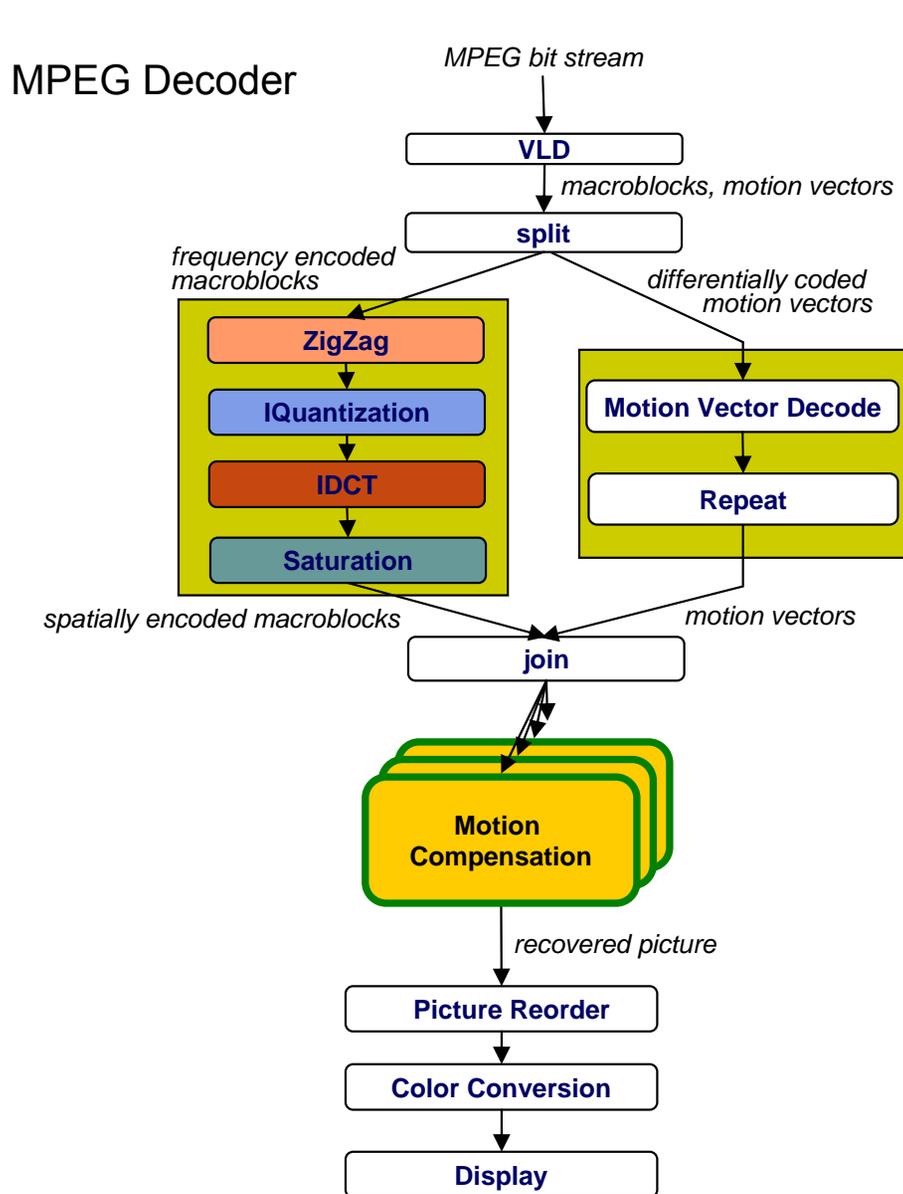
6.189 IAP 2007

Lecture7

Design Patterns for
Parallel Programming II

Recap: Common Steps to Parallelization

Partitioning

| decomposition | | assignment | | orchestration | | mapping |

Sequential computation → Tasks → Execution Units → Parallel Program → Processors

Recap: Decomposing for Concurrency

MPEG Decoder

MPEG bit stream

VLD

macroblocks, motion vectors

split

frequency encoded macroblocks

differentially coded motion vectors

ZigZag

IQuantization

IDCT

Saturation

Motion Vector Decode

Repeat

spatially encoded macroblocks

motion vectors

join

Motion Compensation

recovered picture

Picture Reorder
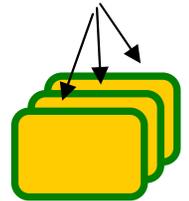
Color Conversion

Display
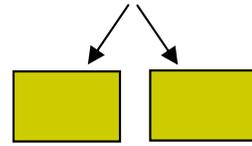
- **Task decomposition**
  - Parallelism in the application
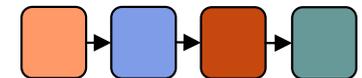
- **Data decomposition**
  - Same computation many data

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

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

T₁

$$a = x + y$$

T₂

$$b = x + z$$

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

$R_2 = \{ x, z \}$
$W_2 = \{ b \}$
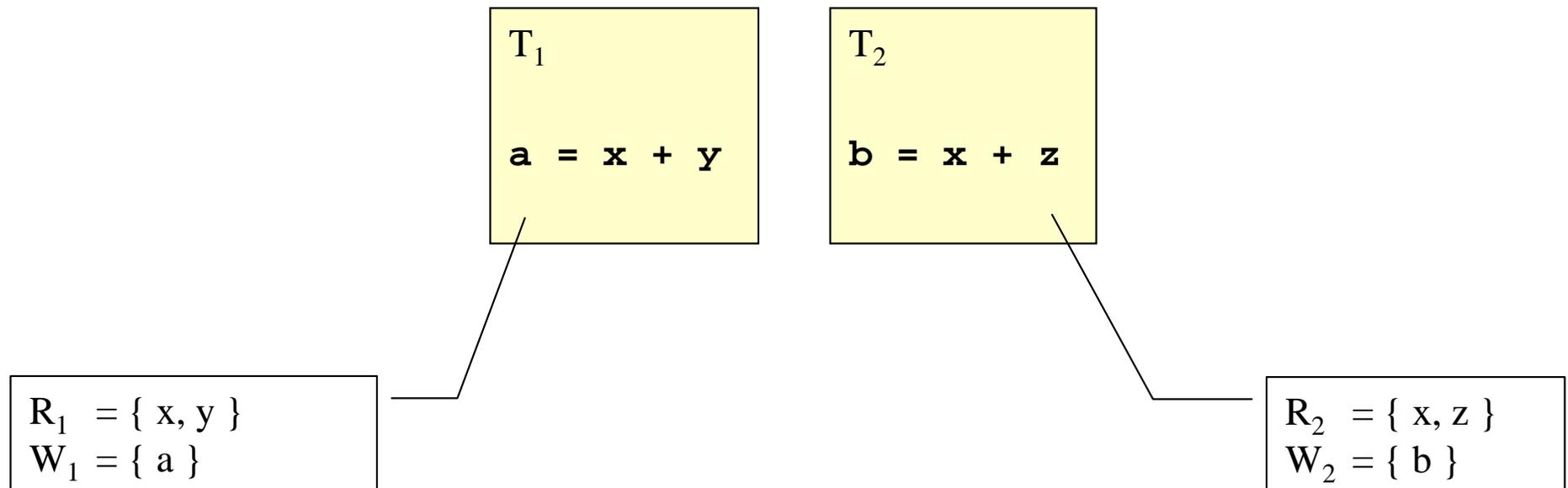
$$R_1 \cap W_2 = \phi$$
$$R_2 \cap W_1 = \phi$$
$$W_1 \cap W_2 = \phi$$

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

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?
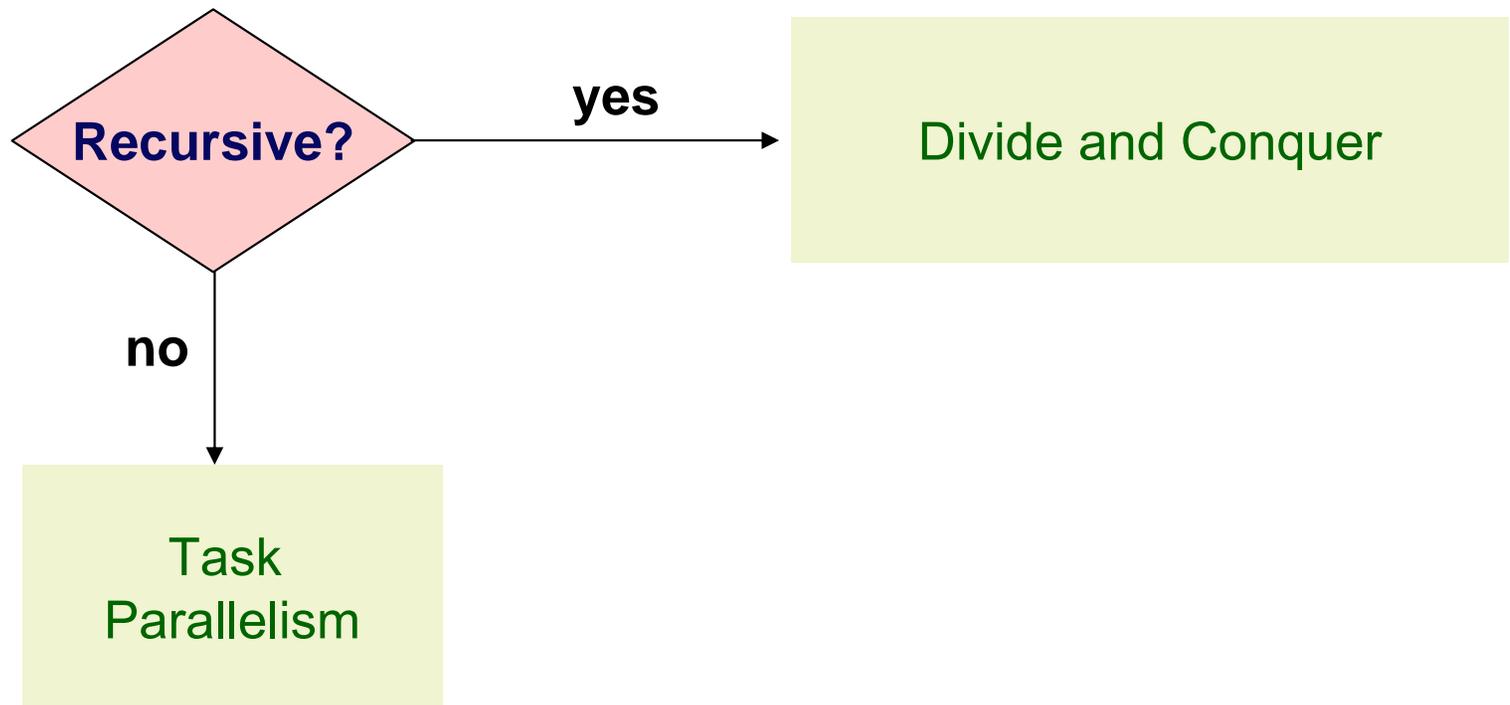
Task Parallelism

- ## 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
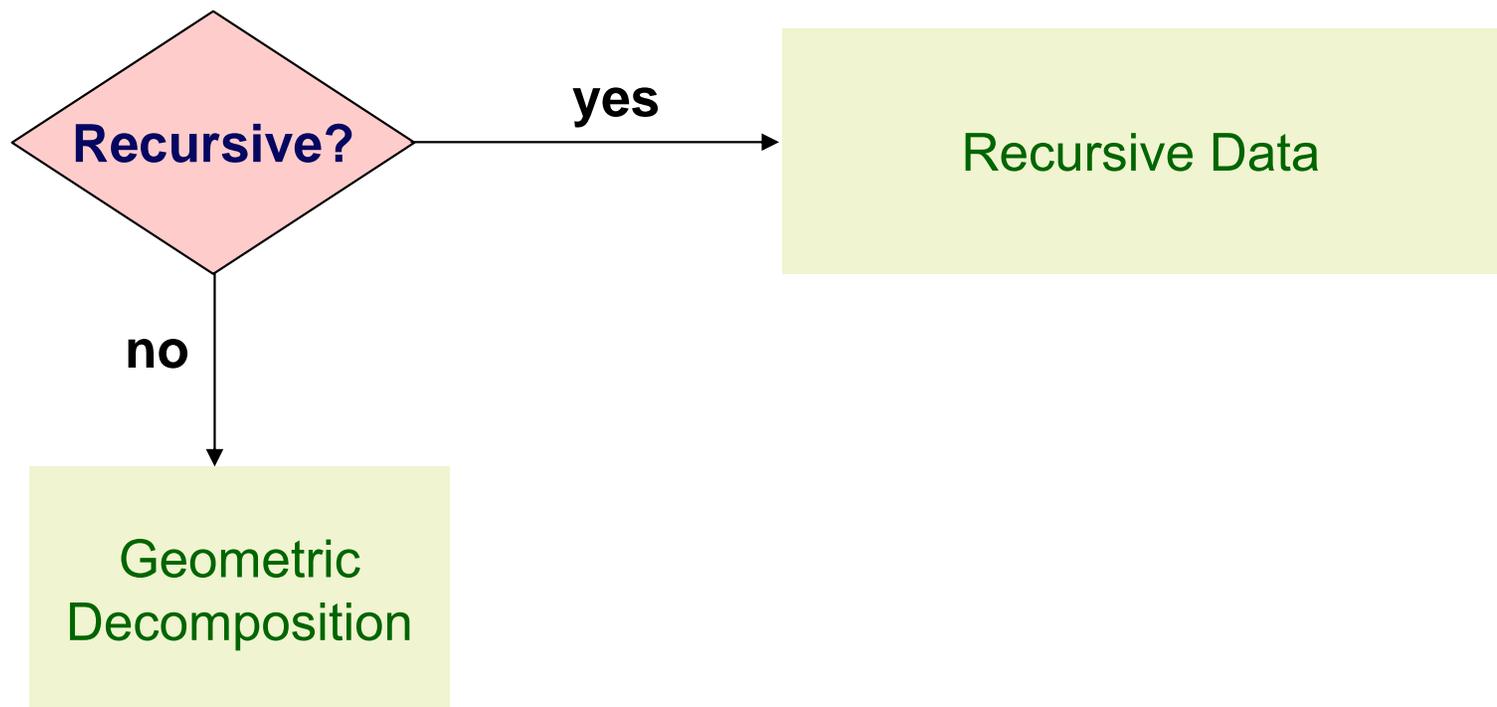 - All tasks may not need to complete to arrive at a solution

Divide and Conquer

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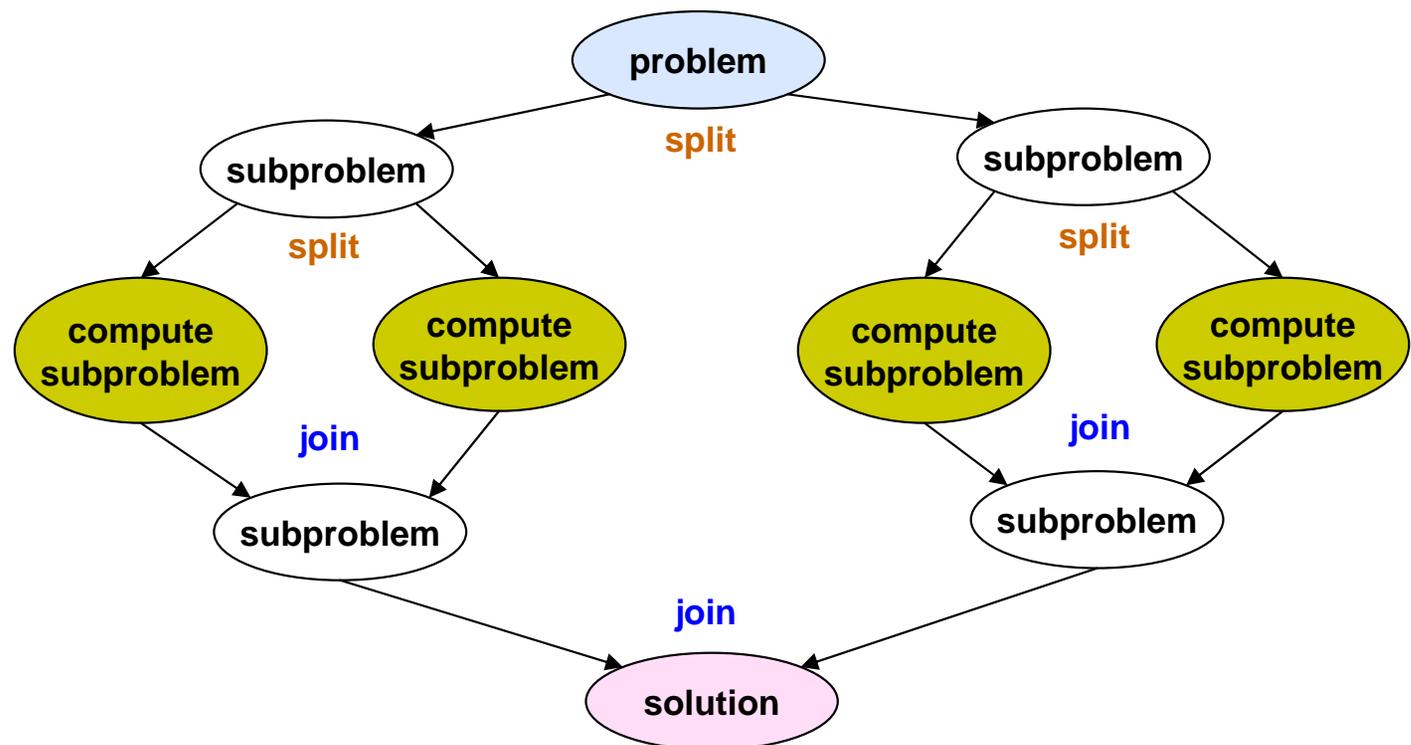
Organize by Data?

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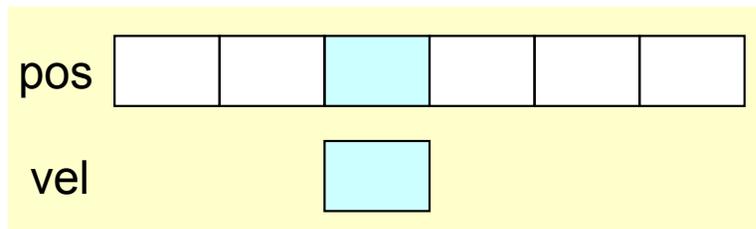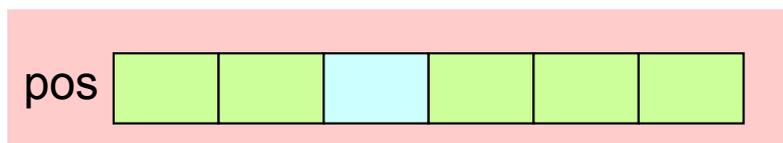 - Recursive data structures

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. $O(n)$

Step 1   Step 2   Step 3

16
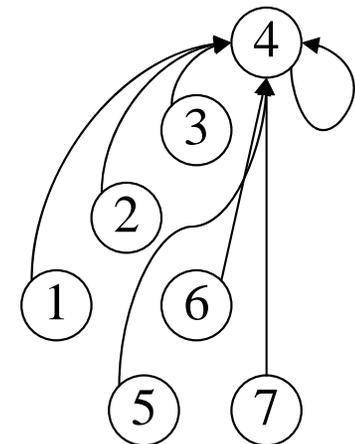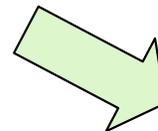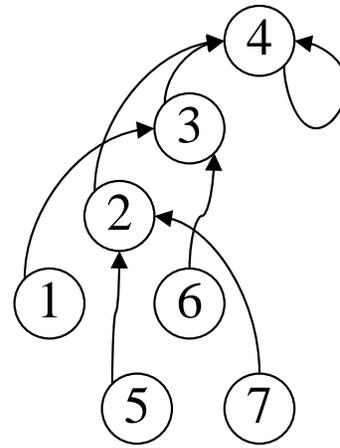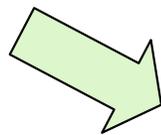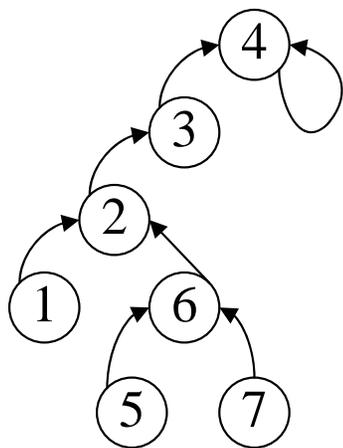
Work vs. Concurrency Tradeoff
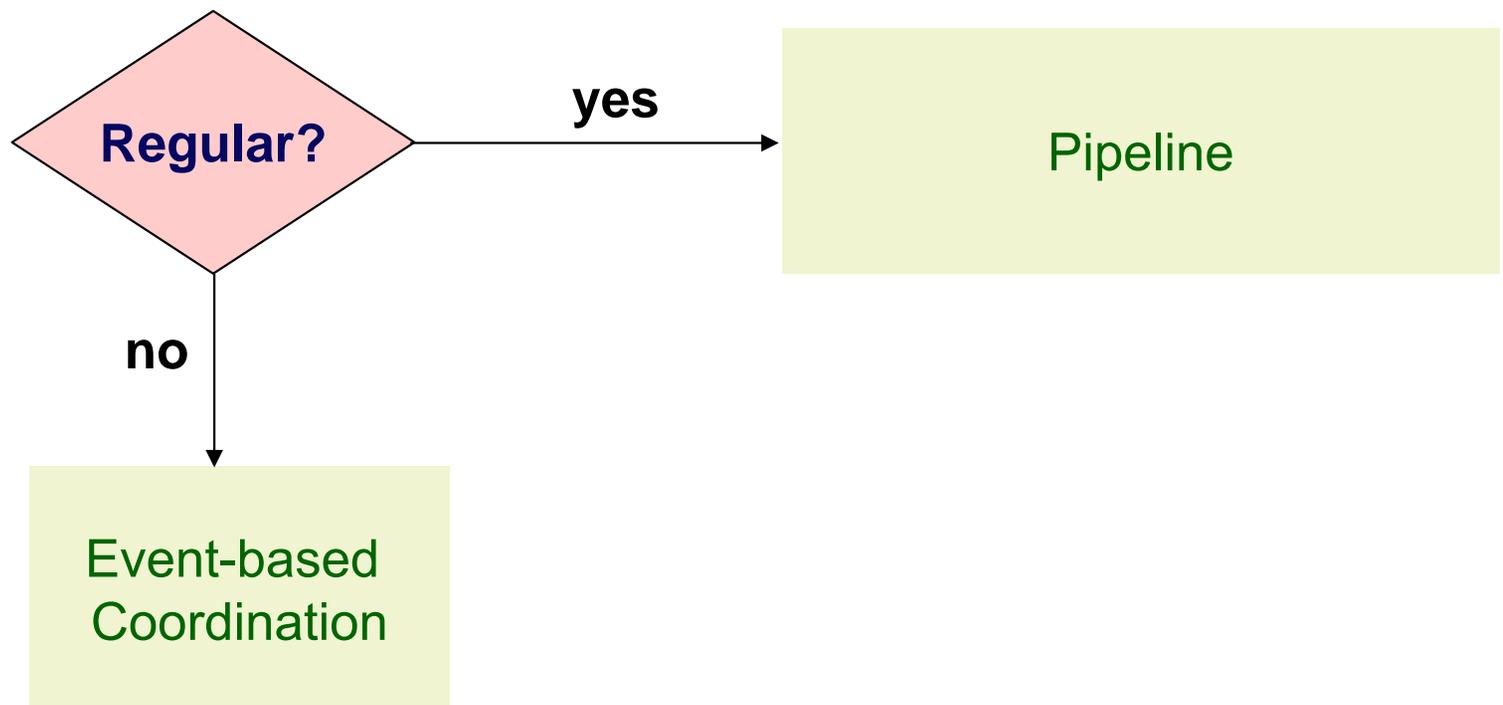
- Parallel restructuring of find the root algorithm leads to $O(n \log n)$ work vs. $O(n)$ with sequential approach

- Most strategies based on this pattern similarly trade off increase in total work for decrease in execution time due to concurrency

 6.189 IAP 2007 MIT

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

```
        ┌─────────┐
        │ Regular? │ ──── yes ────►  ┌──────────────┐
        └─────────┘                  │   Pipeline   │
            │                        └──────────────┘
            no
            │
            ▼
     ┌──────────────┐
     │ Event-based  │
     │ Coordination │
     └──────────────┘
```

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

6.189 IAP 2007

Supporting Structures

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

 6.189 IAP 2007 MIT

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

$$f(x) = \frac{4.0}{(1+x^2)}$$
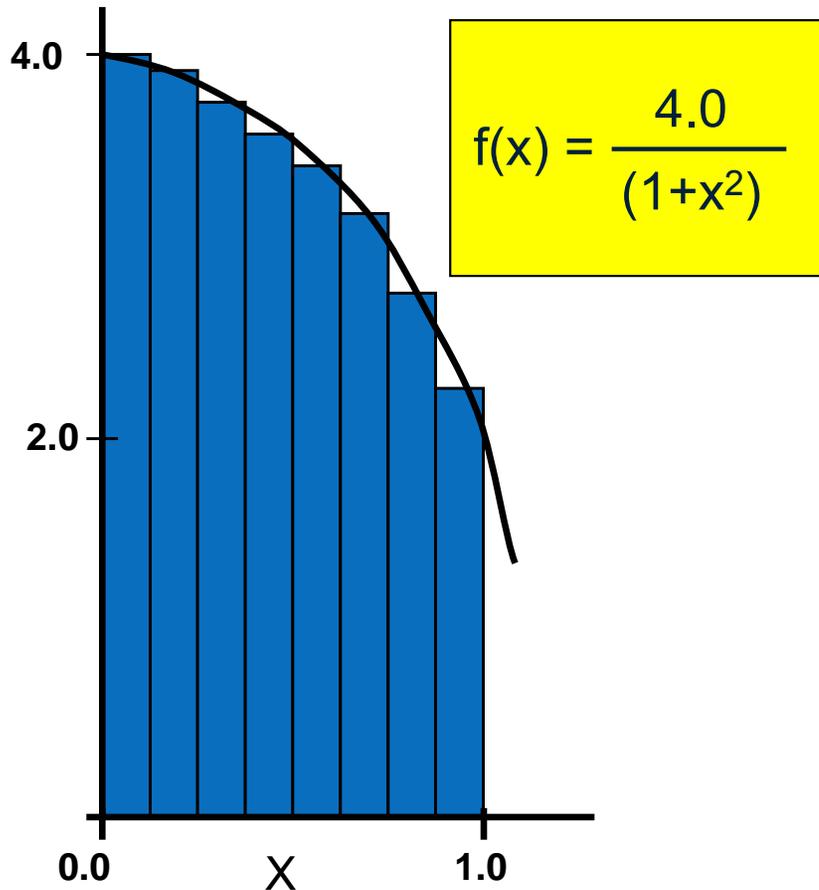
```
static long num_steps = 100000;

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();
}
```
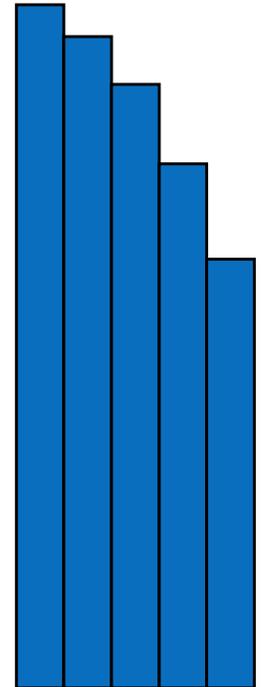
Block vs. Cyclic Work Distribution

```
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();
}
```
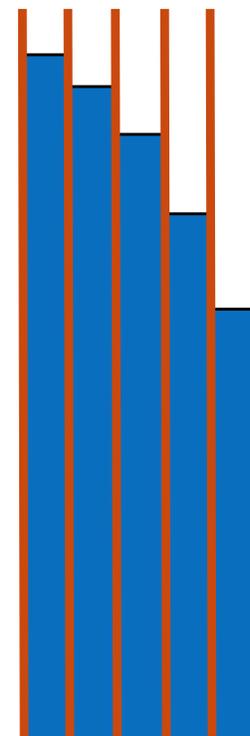
SPMD Challenges

- 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
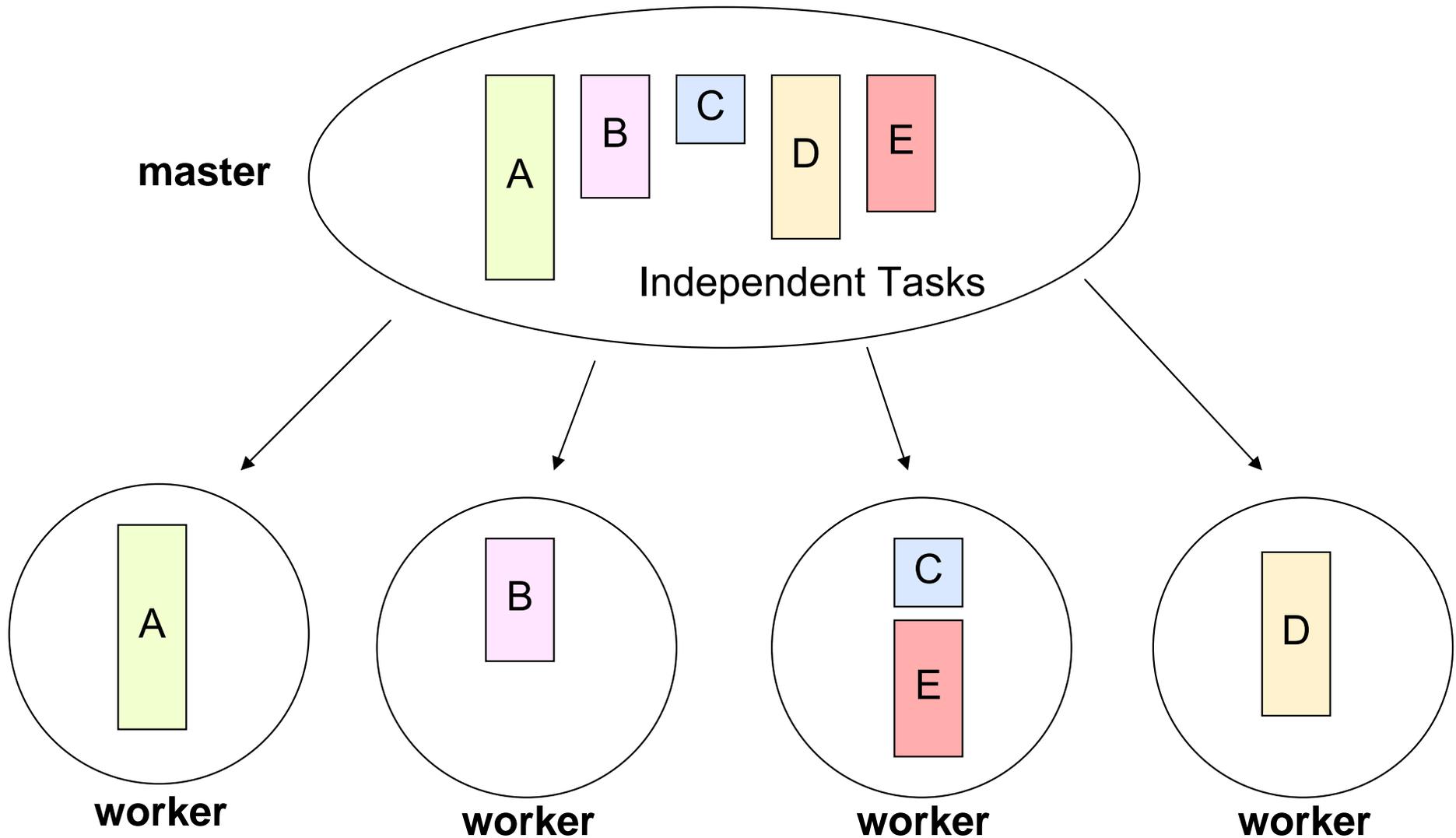
Loop Parallelism Pattern

- 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];
```

i = 0	i = 4	i = 8
i = 1	i = 5	i = 9
i = 2	i = 6	i = 10
i = 3	i = 7	i = 11
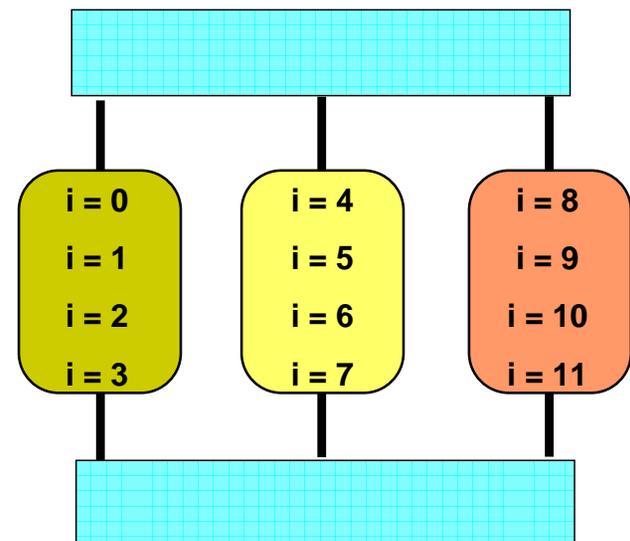
Master/Worker Pattern

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
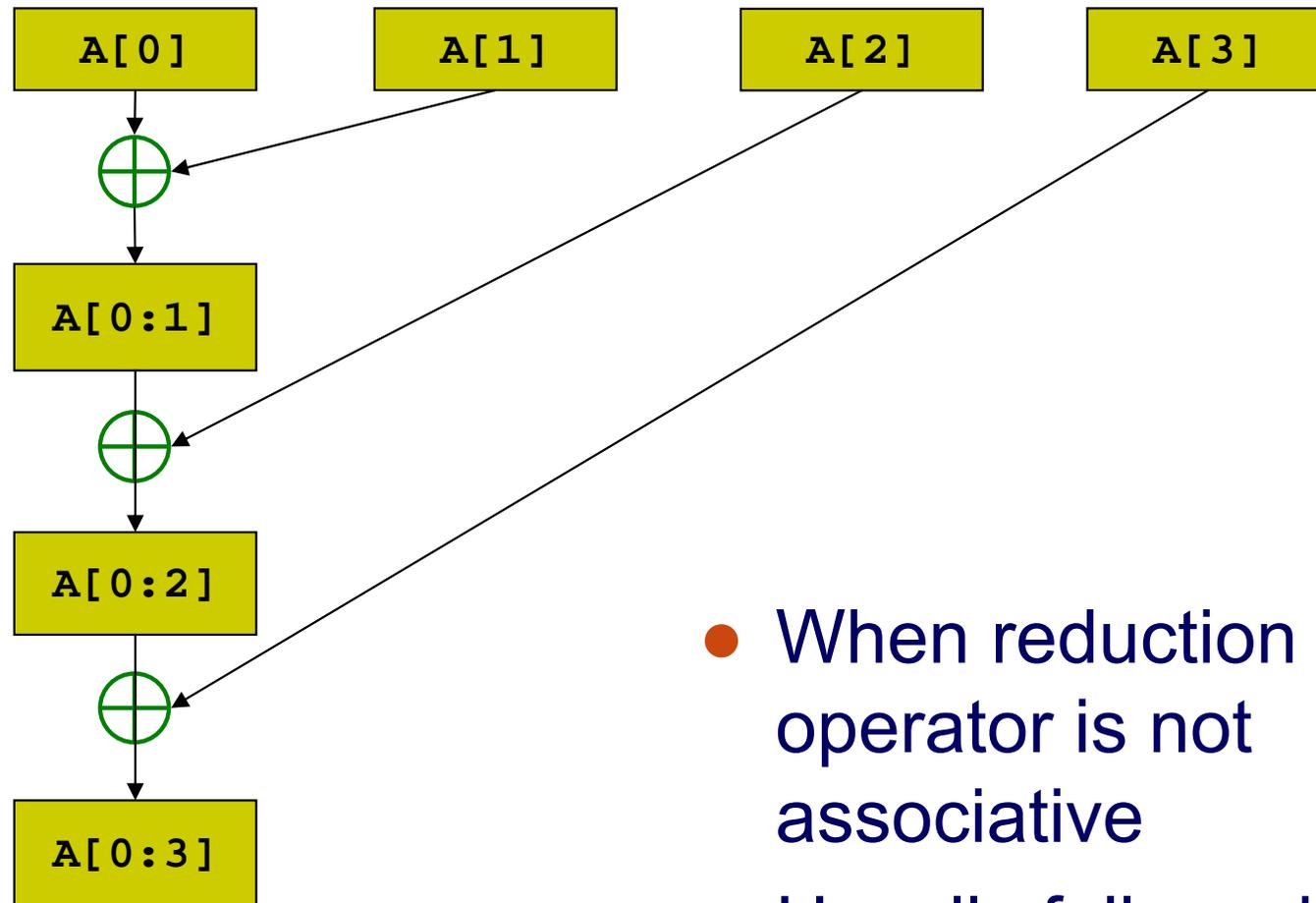
Fork/Join Pattern

- 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

6.189 IAP 2007

Communication Patterns
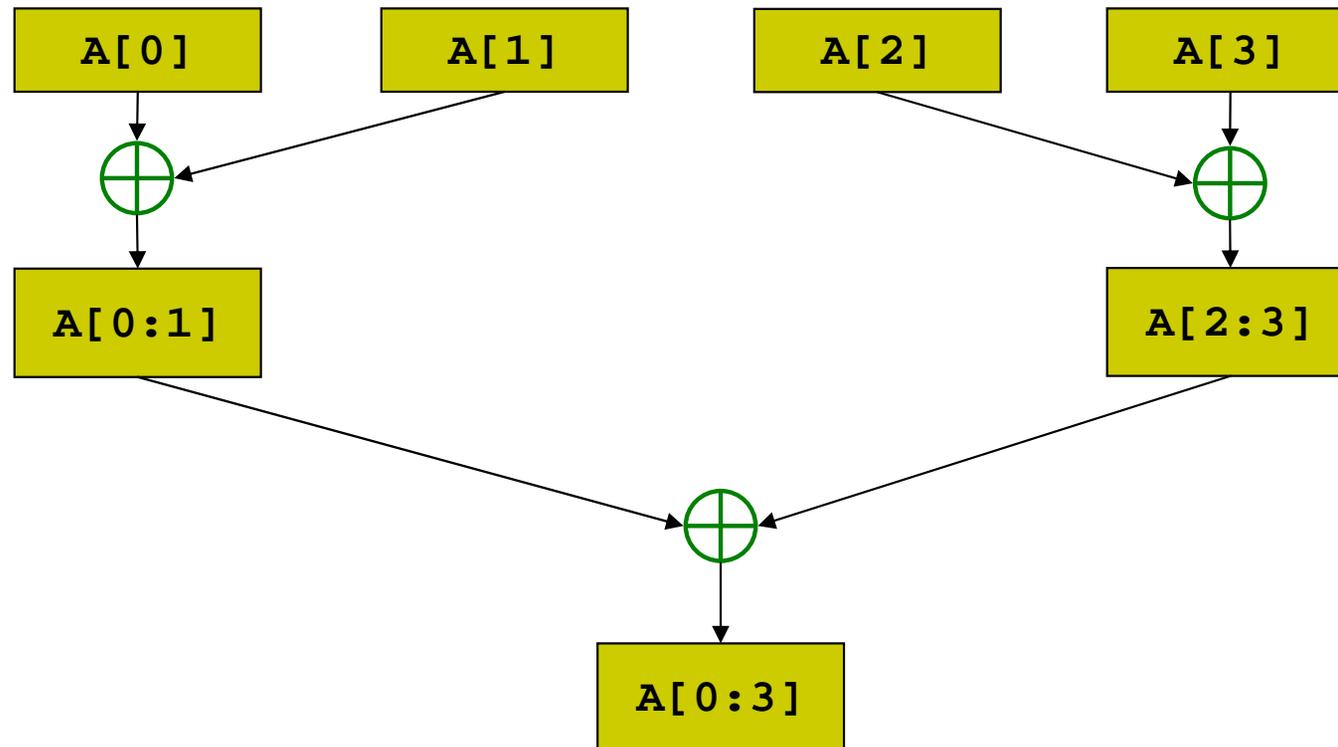
- Point-to-point
- Broadcast
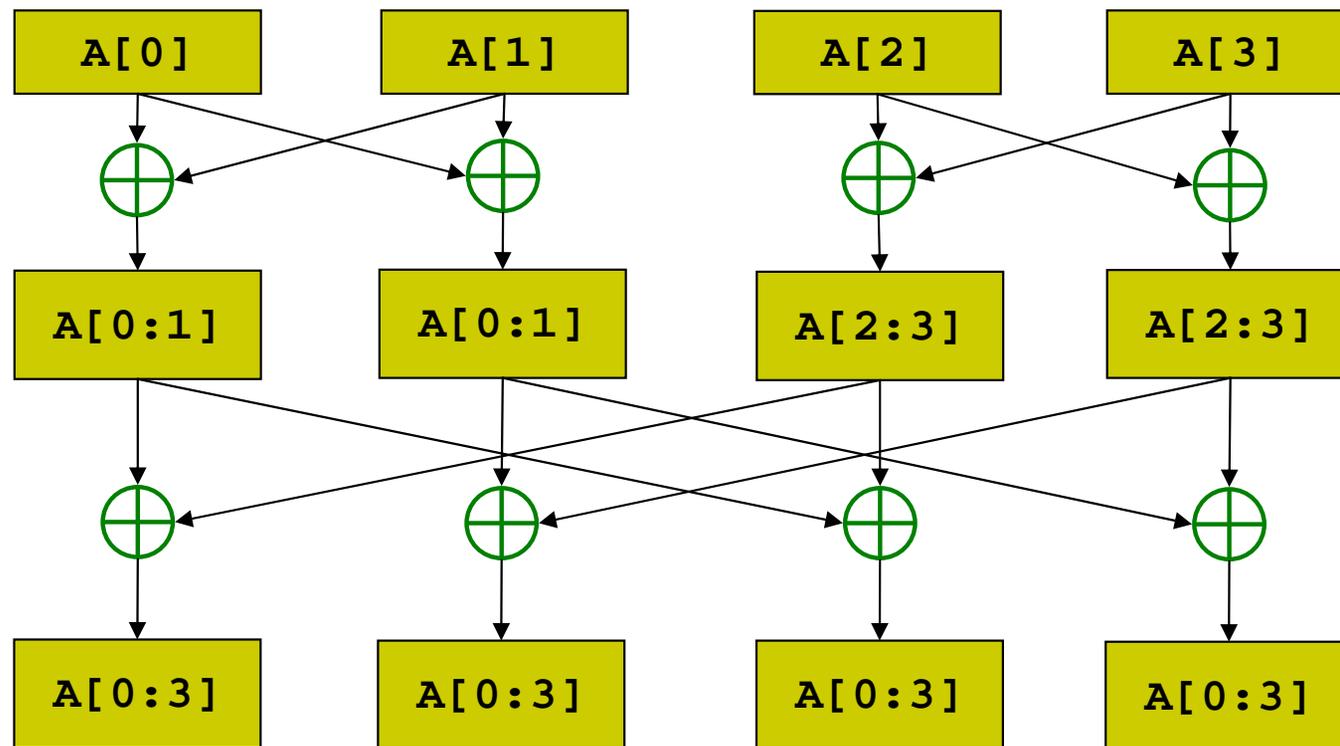- Reduction

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. $O(2n)$

6.189 IAP 2007

Summary

36

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