Lecture 4
Patterns for Parallel Programming

John Cavazos

Dept of Computer & Information Sciences

University of Delaware

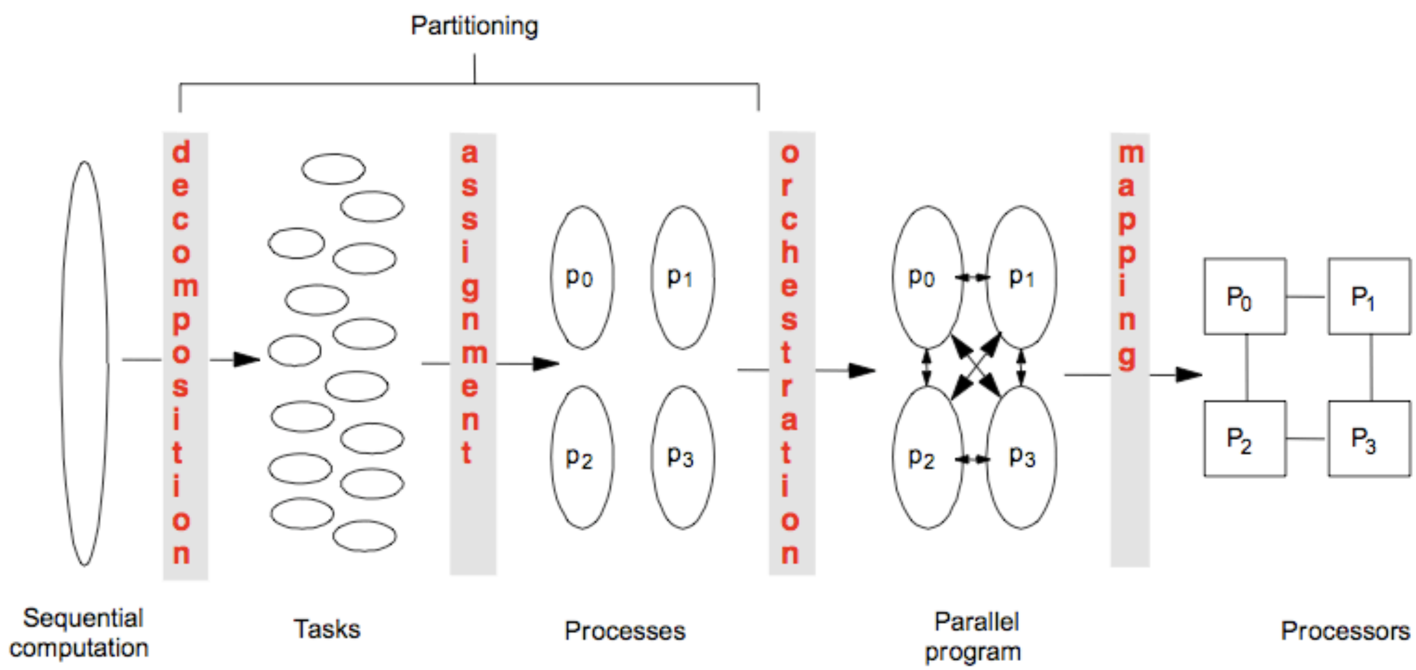
www.cis.udel.edu/~cavazos/cisc879

Lecture Overview

- Writing a Parallel Program
- Design Patterns for Parallel Programs
  - Finding Concurrency
  - Algorithmic Structure
  - Supporting Structures
  - Implementation Mechanisms

Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007

Parallelization Common Steps



1. Study problem or code

2. Look for parallelism opportunities

3. Try to keep all cores busy doing useful work

Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007

Decomposition

- Identify concurrency
  - Decide level to exploit it
  - Requires understanding the algorithm!
  - May require restructing program/algorithm
    - May require entirely new algorithm
- Break computation into tasks
  - Divided among processes
  - Tasks may become available dynamically
  - Number of tasks can vary with time

Want enough tasks to keep processors busy.

Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007

Assignment

- Specify mechanism to divide work
  - Balance of computation
  - Reduce communication
- Structured approaches work well
  - Code inspection and understanding algorithm
  - Using design patterns (second half part of lecture)

Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007

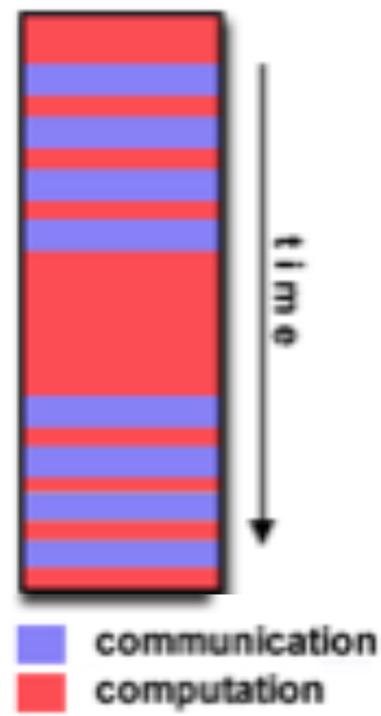
Granularity

- Ratio of computation and communication
- **Fine-grain parallelism**
- **Coarse-grain parallelism**

Most efficient granularity depends on algorithm/hardware.

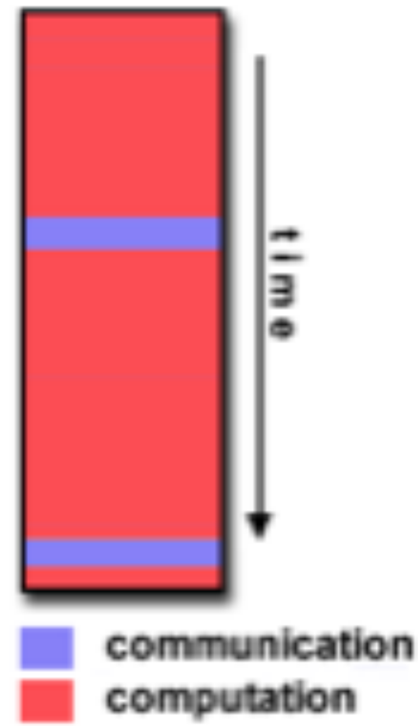
Fine-grain Parallelism

- Tasks execute little comp. between comm.
- Easy to load balance
- If ***too fine***, comm. may take longer than comp.



Coarse-grain Parallelism

- Long computations between communication
- More opportunity for performance increase
- Harder to load balance



Orchestration and Mapping

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

Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007

Lecture Overview

- Parallelizing a Program
- Design Patterns for Parallelization
  - Finding Concurrency
  - Algorithmic Structure
  - Supporting Structures
  - Implementation Mechanisms

Patterns for Parallelization

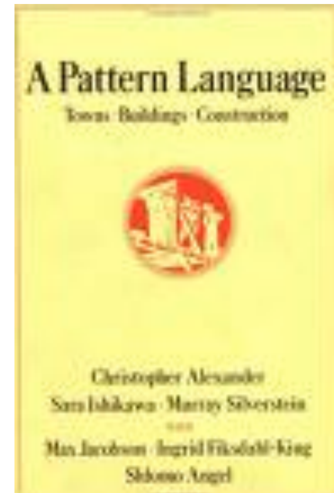
- Parallelization is a difficult problem
  - Hard to fully exploit parallel hardware
- Solution: ***Design Patterns***

What are Design Patterns?

- Cookbook for parallel programmers
  - Can lead to high quality solutions
- Provides a common vocabulary
  - Each pattern has a name and associated vocabulary for discussing solutions
- Helps with software reusability and modularity

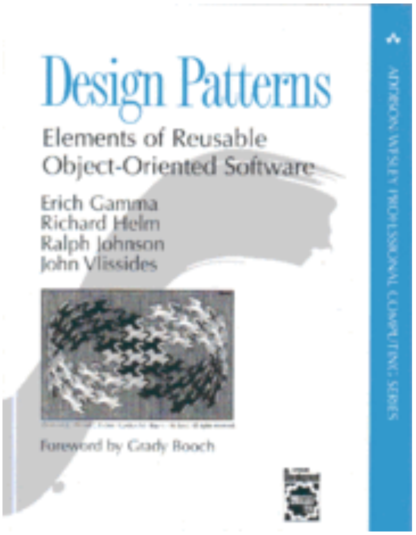
Architecture Patterns

- Christopher Alexander
  - Berkeley architecture professor
- Developed patterns for architecture
  - City planning
  - Layout of windows in a room
- Attempt to capture principles for “living” designs



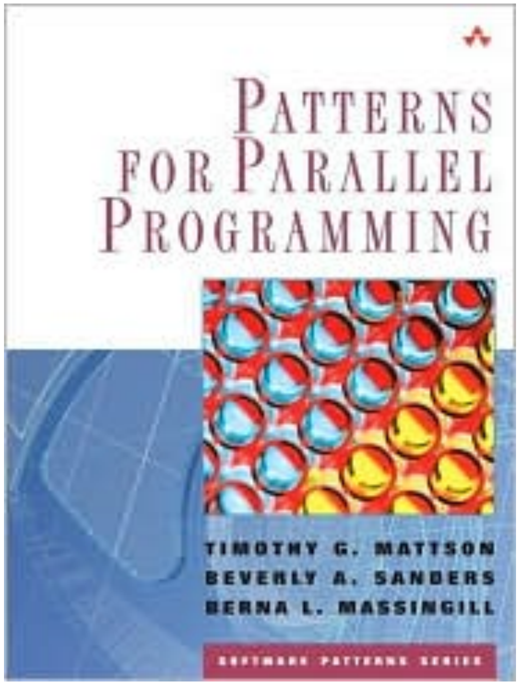
Patterns for OOP

- First to bring pattens to CS
- Design Patterns: Elements of Reusable Object-Oriented Software (1994)
  - Gamma et al. (Gang of Four)
- Catalogue of “patterns”
  - Solutions to common problems in software design
- Not a finished solution!
  - Rather a template for how to solve a problem



Patterns Parallelization Book

- Patterns for Parallel Programming.
  - Mattson et al. (2005)
- Four Design Spaces
  - Finding Concurrency
    - Expose concurrent task or data
  - Algorithm Structure
    - Map tasks to processes
  - Supporting Structures
    - Code and data structuring patterns
  - Implementation Mechanisms
    - Low-level mechanisms for writing programs



Finding Concurrency

- Decomposition
  - Data, Task, Pipeline
- Dependency Analysis
  - Control dependences
  - Data dependences
- Design Evaluation
  - Suitability for target platform
  - Design quality

Decomposition

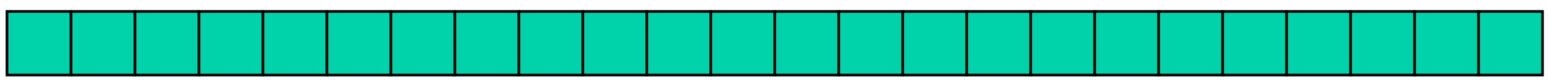
- Data (domain) decomposition
  - Break data up independent units
- Task (functional) decomposition
  - Break problem into parallel tasks
- Case for Pipeline decomposition
  - Special case of task decomposition

Data (Domain) Decomposition

- Also known as Domain Decomposition
- Implied by Task Decomposition
  - ***Which decomposition more natural to start with***:
  - 1) Decide how data elements divided among cores
  - 2) Decide which tasks each core should be performing
- Data decomposition is good starting point when
  - Main computation manipulating a large data structure
  - Similar operations applied to different parts of a data structure (SPMD)
- Example : Vector operations

Data Decomposition

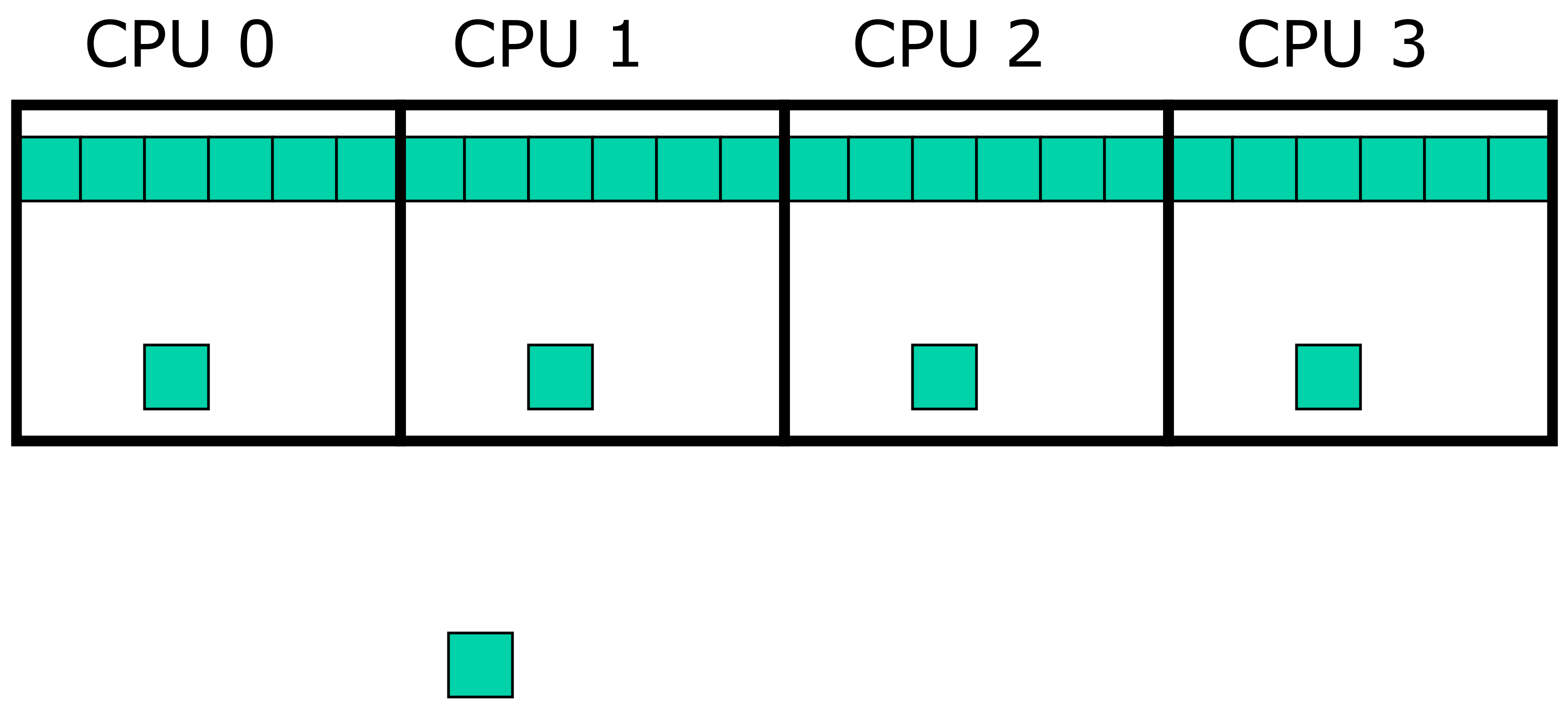
Find the largest element of an array



Slide Source: Intel Software College, Intel Corp.

Data Decomposition

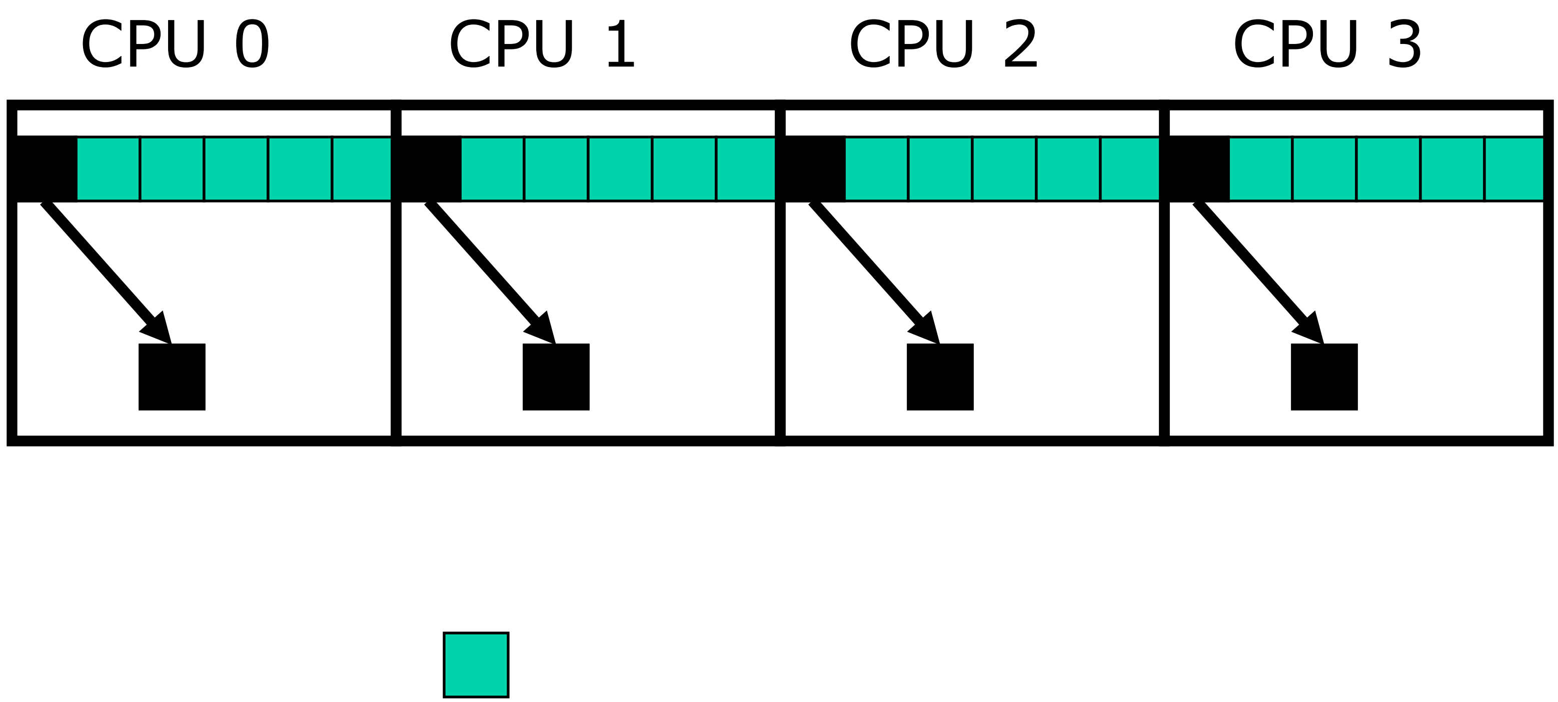
Find the largest element of an array



Slide Source: Intel Software College, Intel Corp.

Data Decomposition

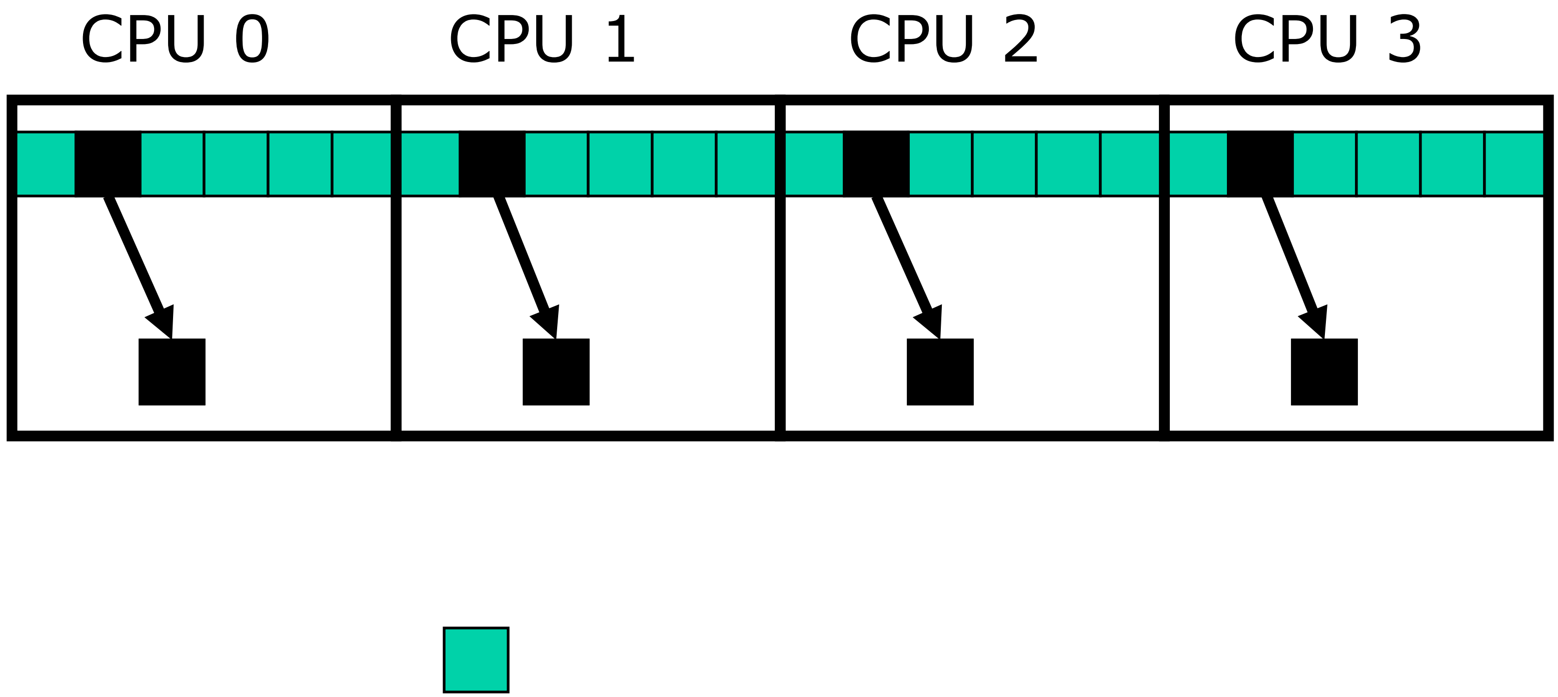
Find the largest element of an array



Slide Source: Intel Software College, Intel Corp.

Data Decomposition

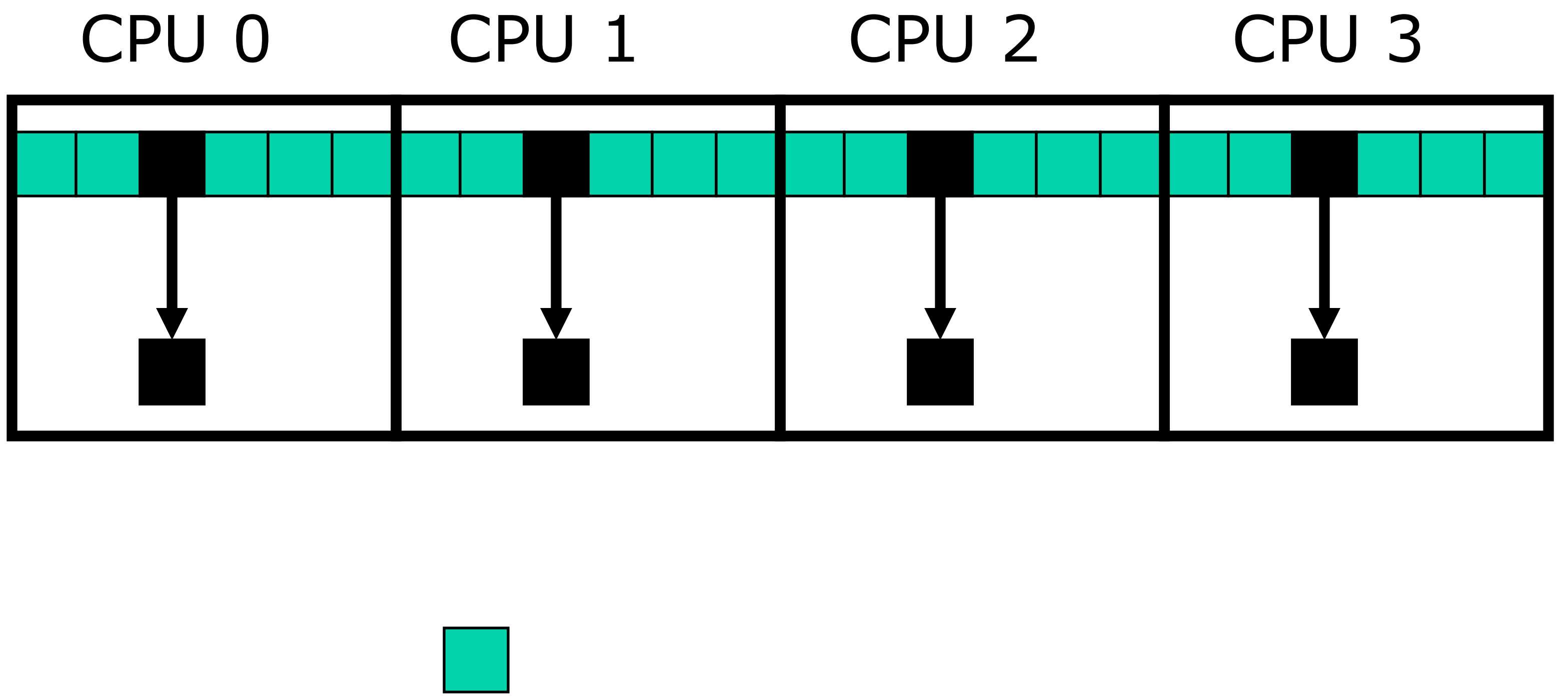
Find the largest element of an array



Slide Source: Intel Software College, Intel Corp.

Data Decomposition

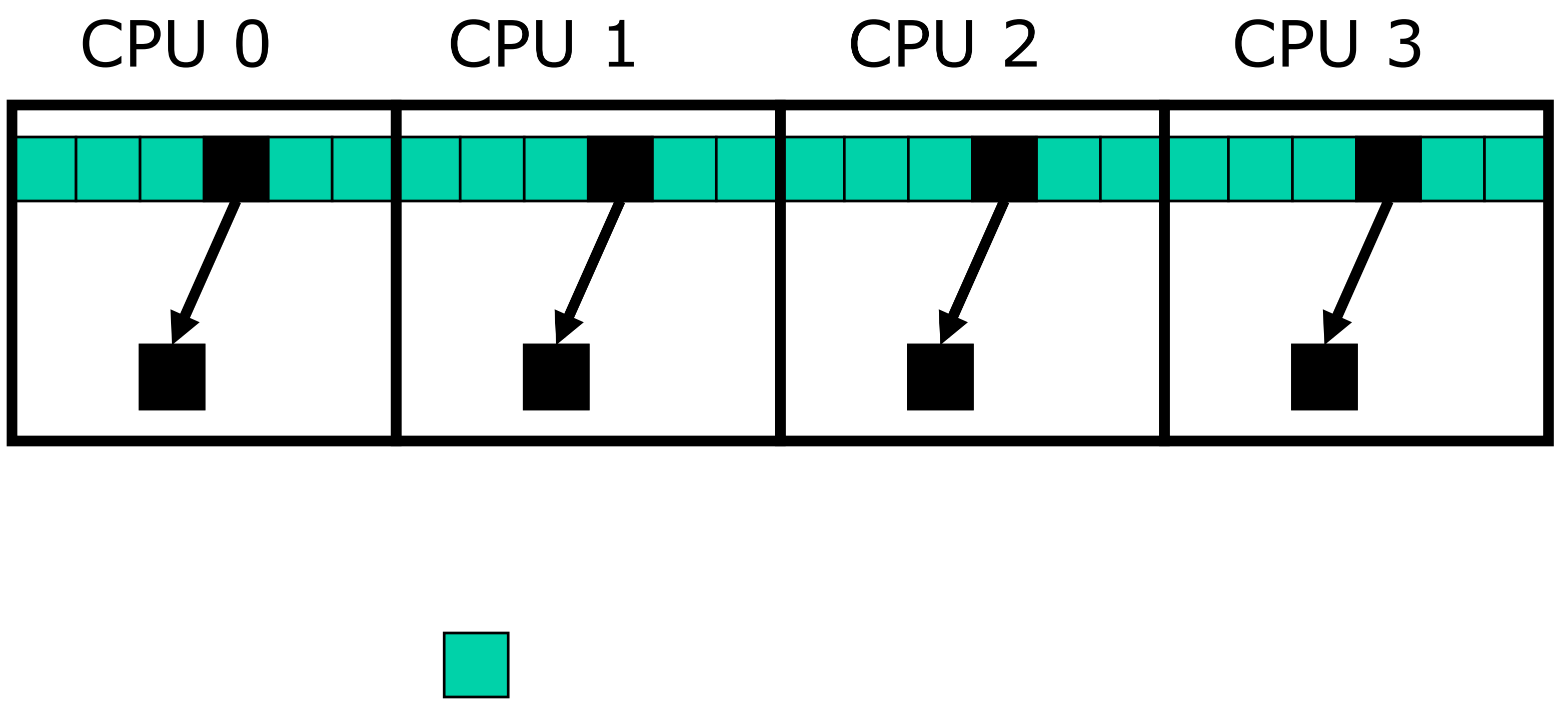
Find the largest element of an array



Slide Source: Intel Software College, Intel Corp.

Data Decomposition

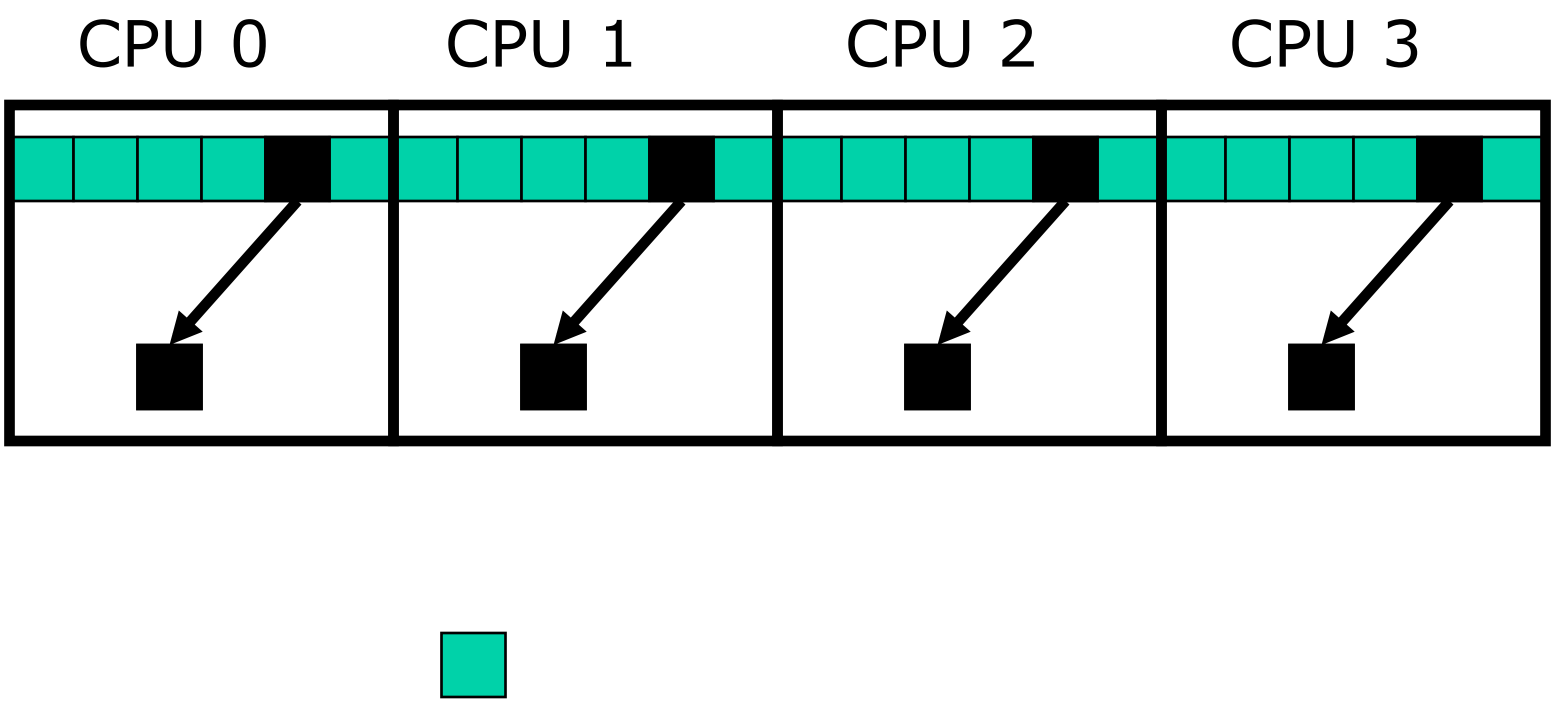
Find the largest element of an array



Slide Source: Intel Software College, Intel Corp.

Data Decomposition

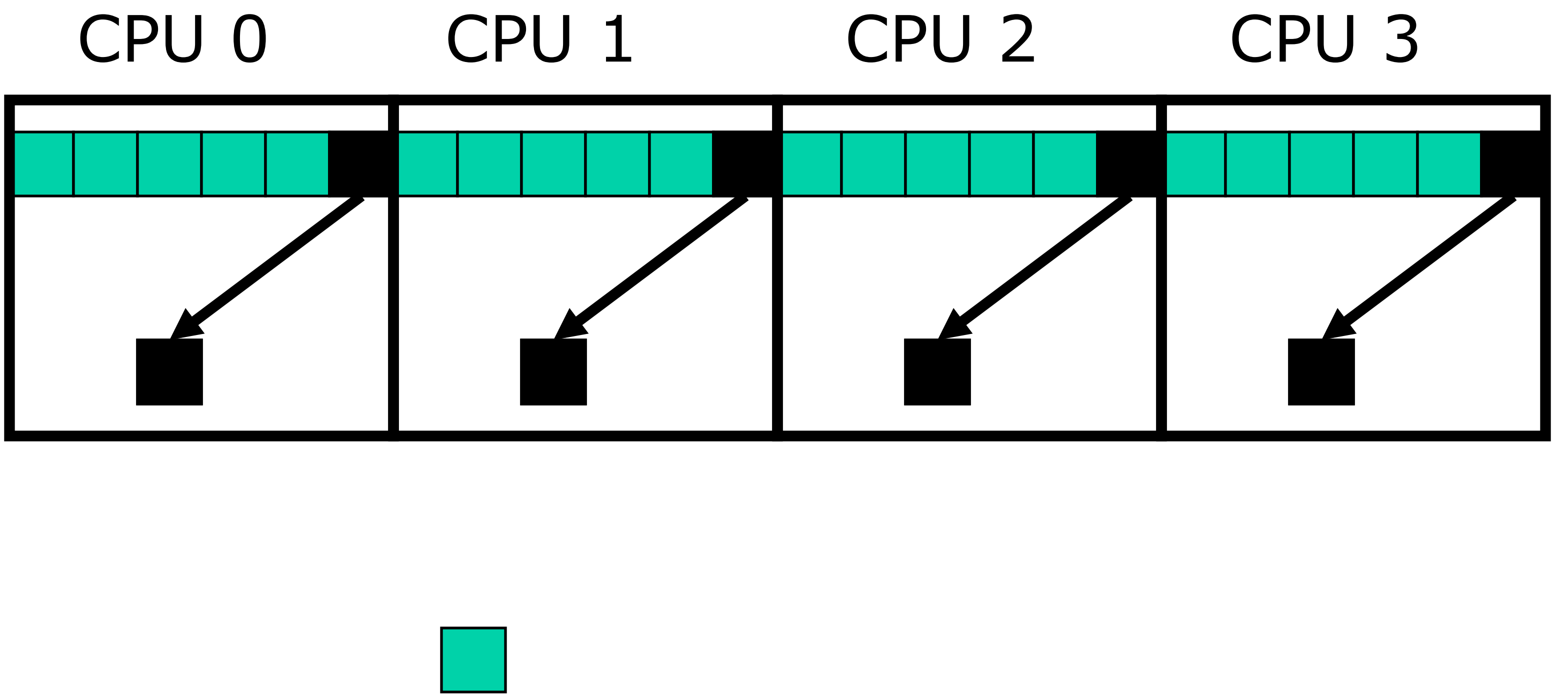
Find the largest element of an array



Slide Source: Intel Software College, Intel Corp.

Data Decomposition

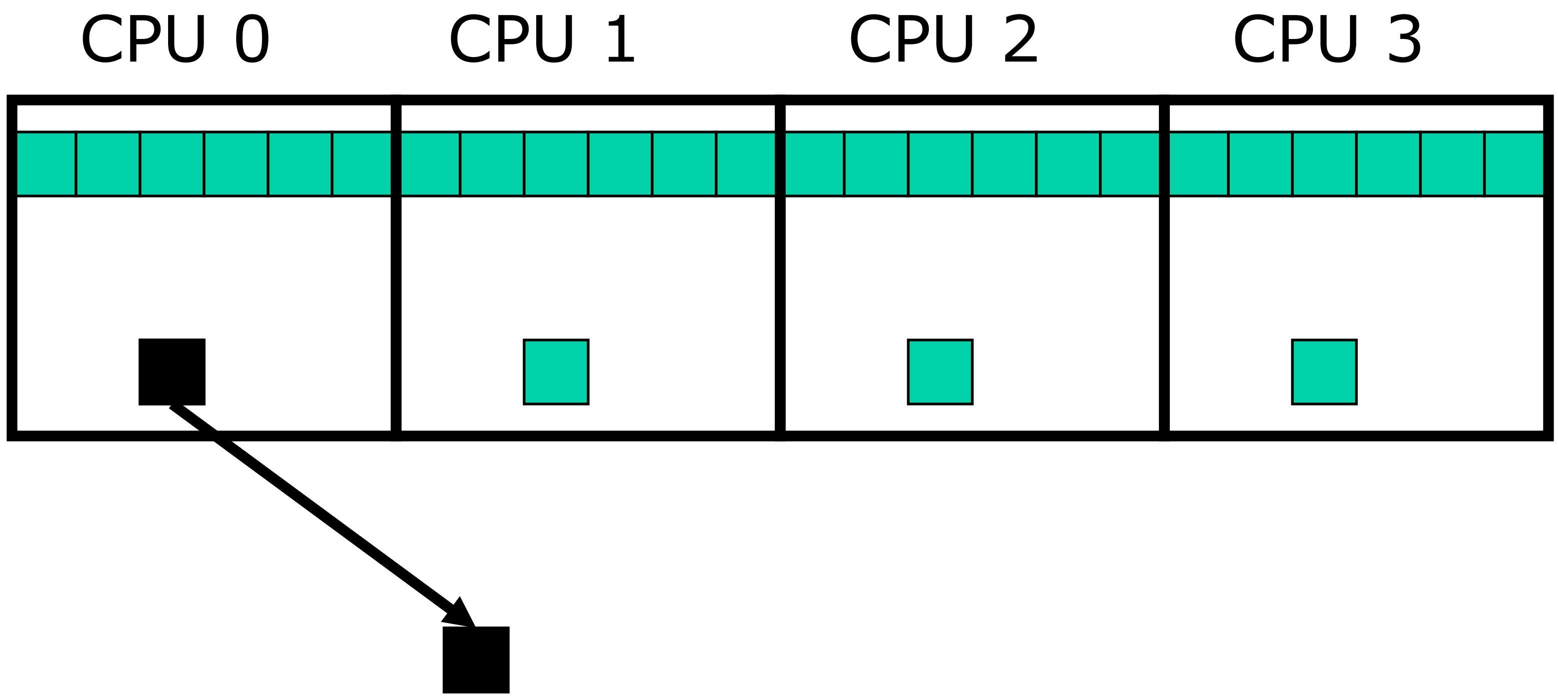
Find the largest element of an array



Slide Source: Intel Software College, Intel Corp.

Data Decomposition

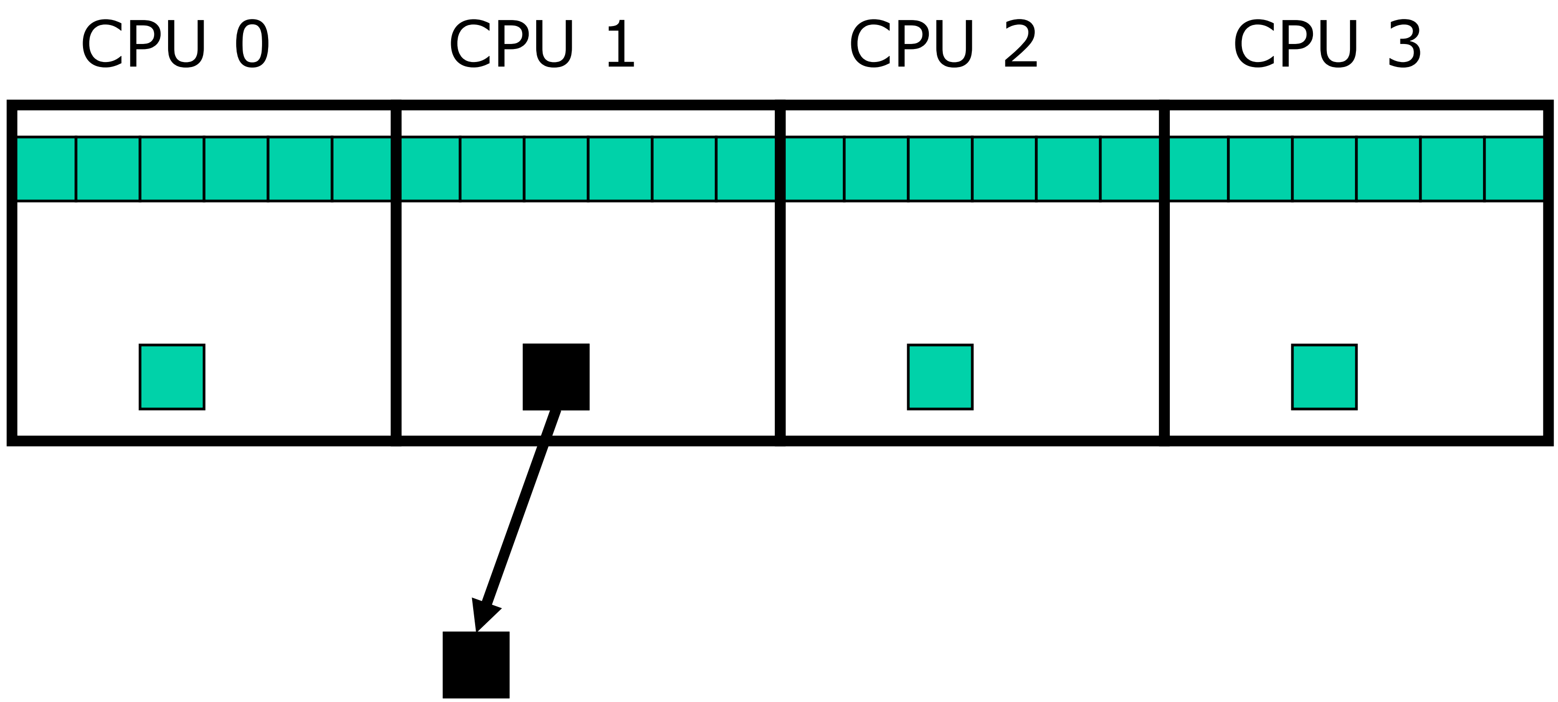
Find the largest element of an array



Slide Source: Intel Software College, Intel Corp.

Data Decomposition

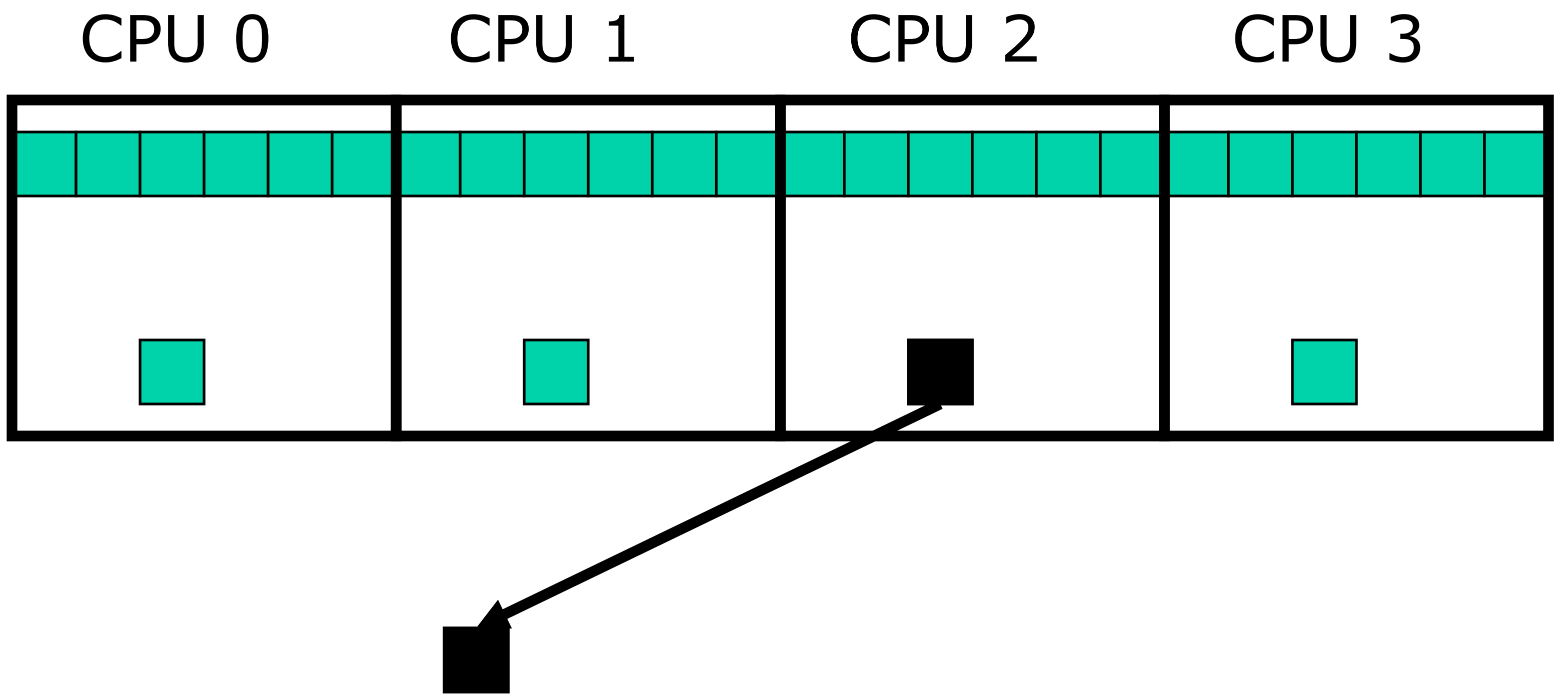
Find the largest element of an array



Slide Source: Intel Software College, Intel Corp.

Data Decomposition

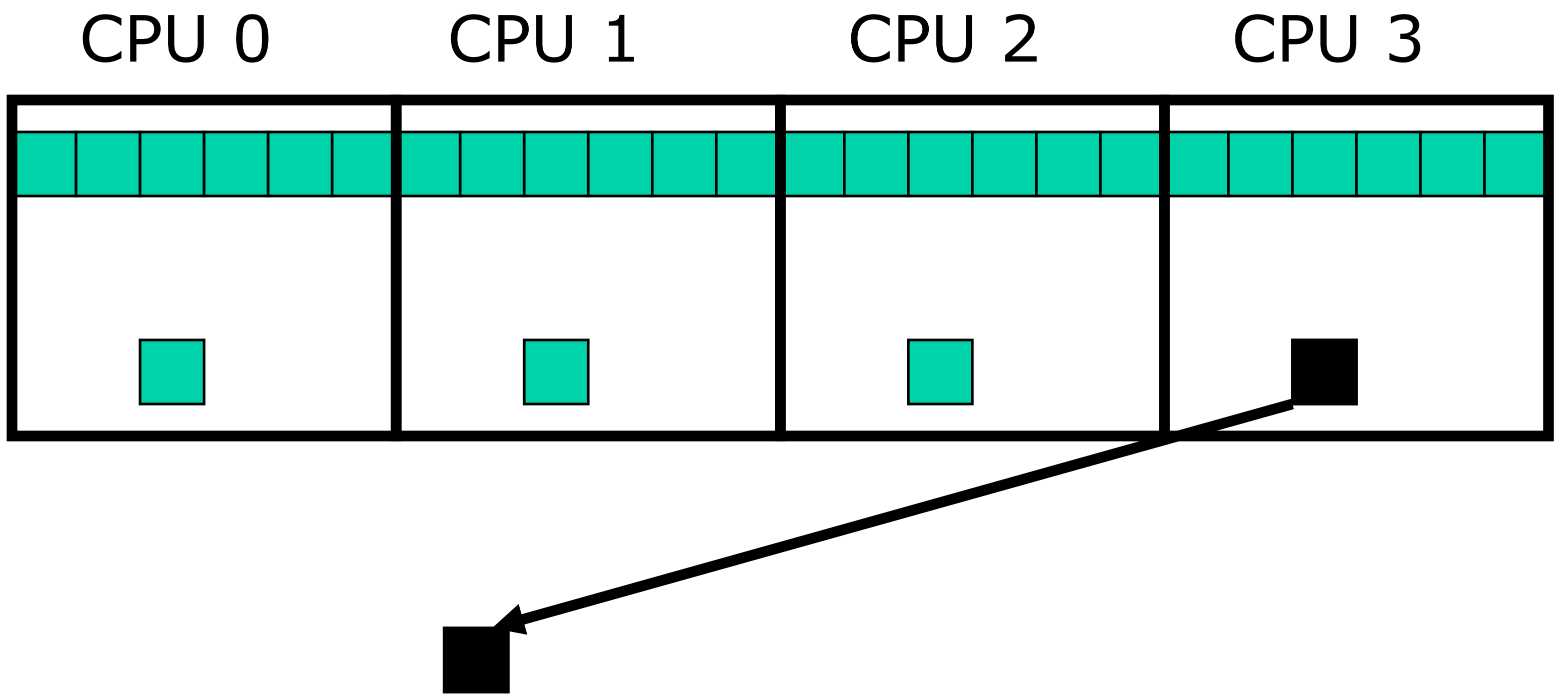
Find the largest element of an array



Slide Source: Intel Software College, Intel Corp.

Data Decomposition

Find the largest element of an array



Slide Source: Intel Software College, Intel Corp.

Data Decomposition Forces

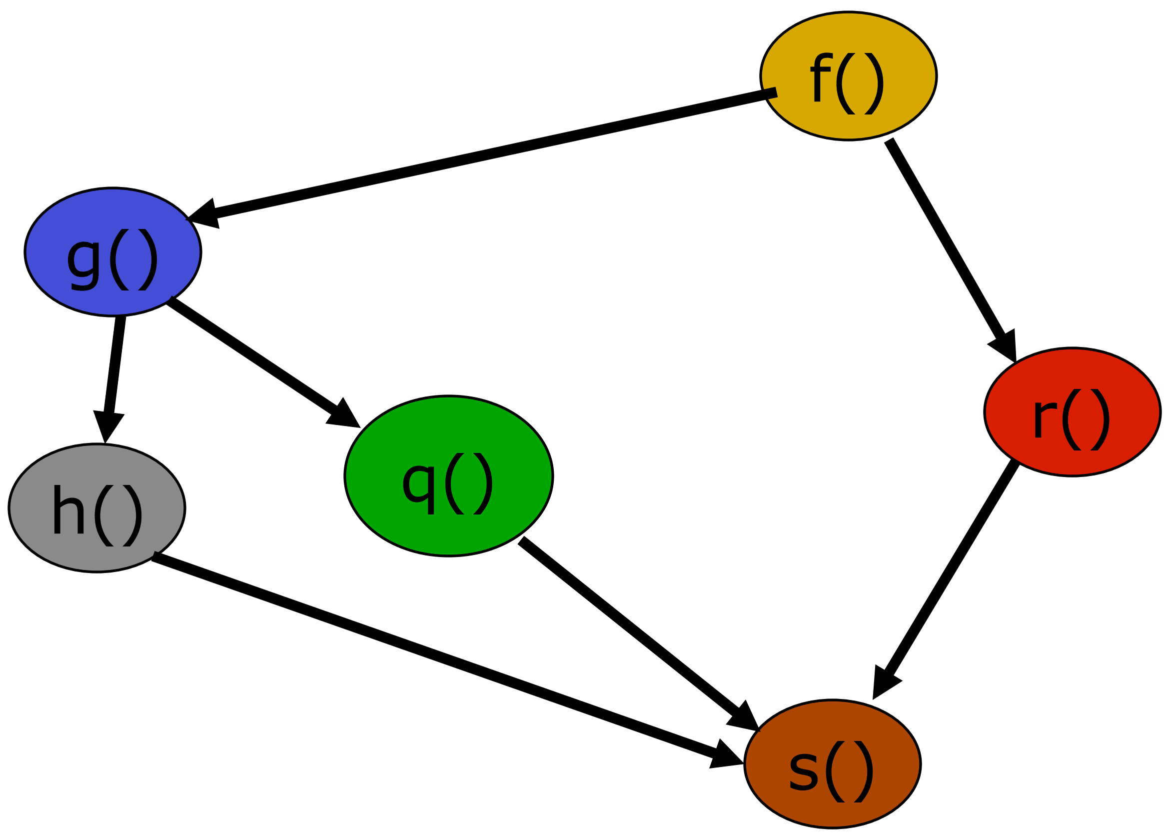
- Flexibility
  - Size of data chunks should support a range of systems
    - Granularity knobs
- Efficiency
  - Data chunks should have comparable computation (load balancing)
- Simplicity
  - Complex data decomposition difficult to debug

Task (Functional) Decomposition

- Programs often naturally decompose into tasks
  - Functions
  - Distinct loop iterations
    - Loop splitting algorithms
- Divide tasks among cores
  - Easier to start with too many tasks and fuse some later
- Decide data accessed (read/written) by each core
- Example: Event-handler for a GUI

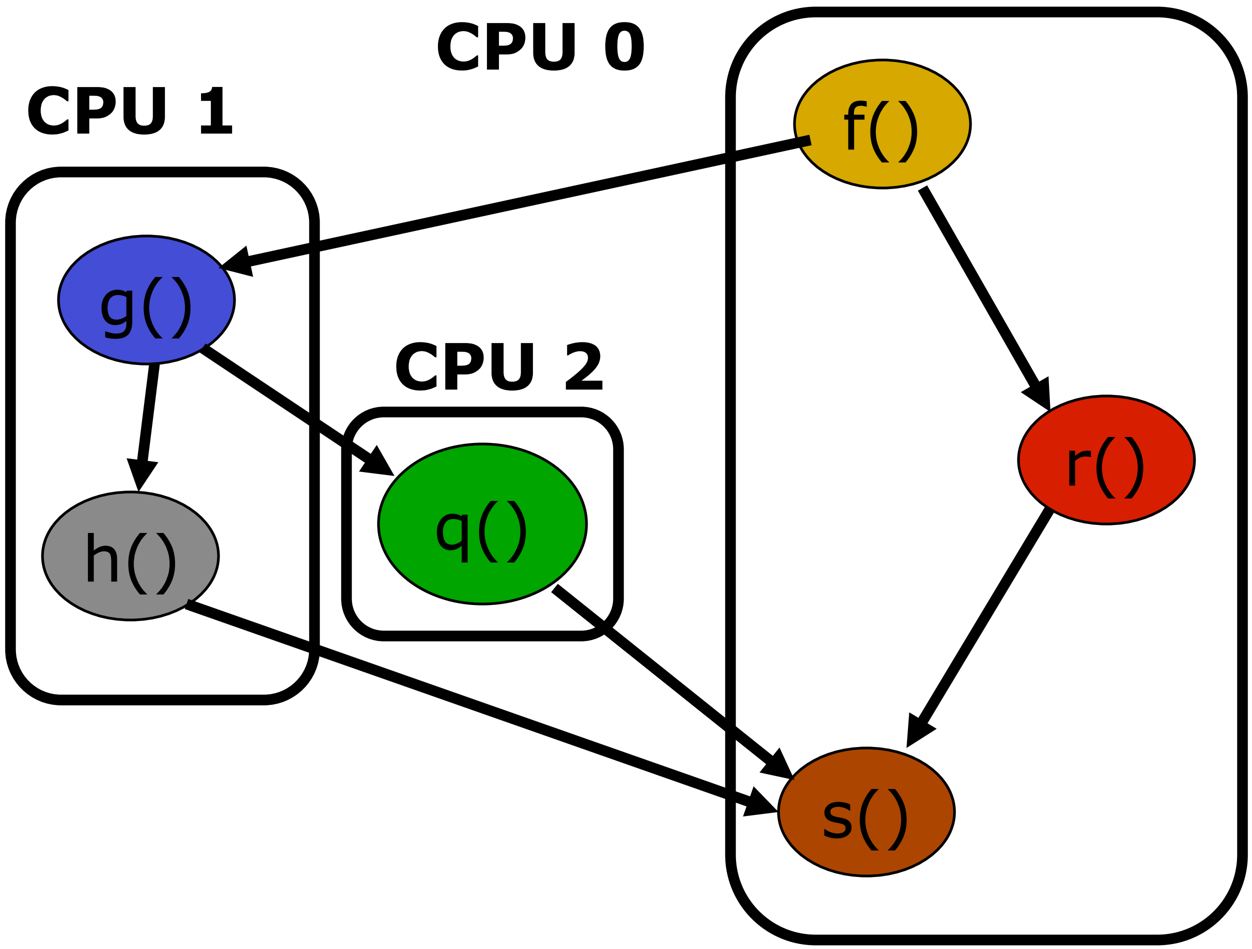
Slide Source: Intel Software College, Intel Corp.

Task Decomposition



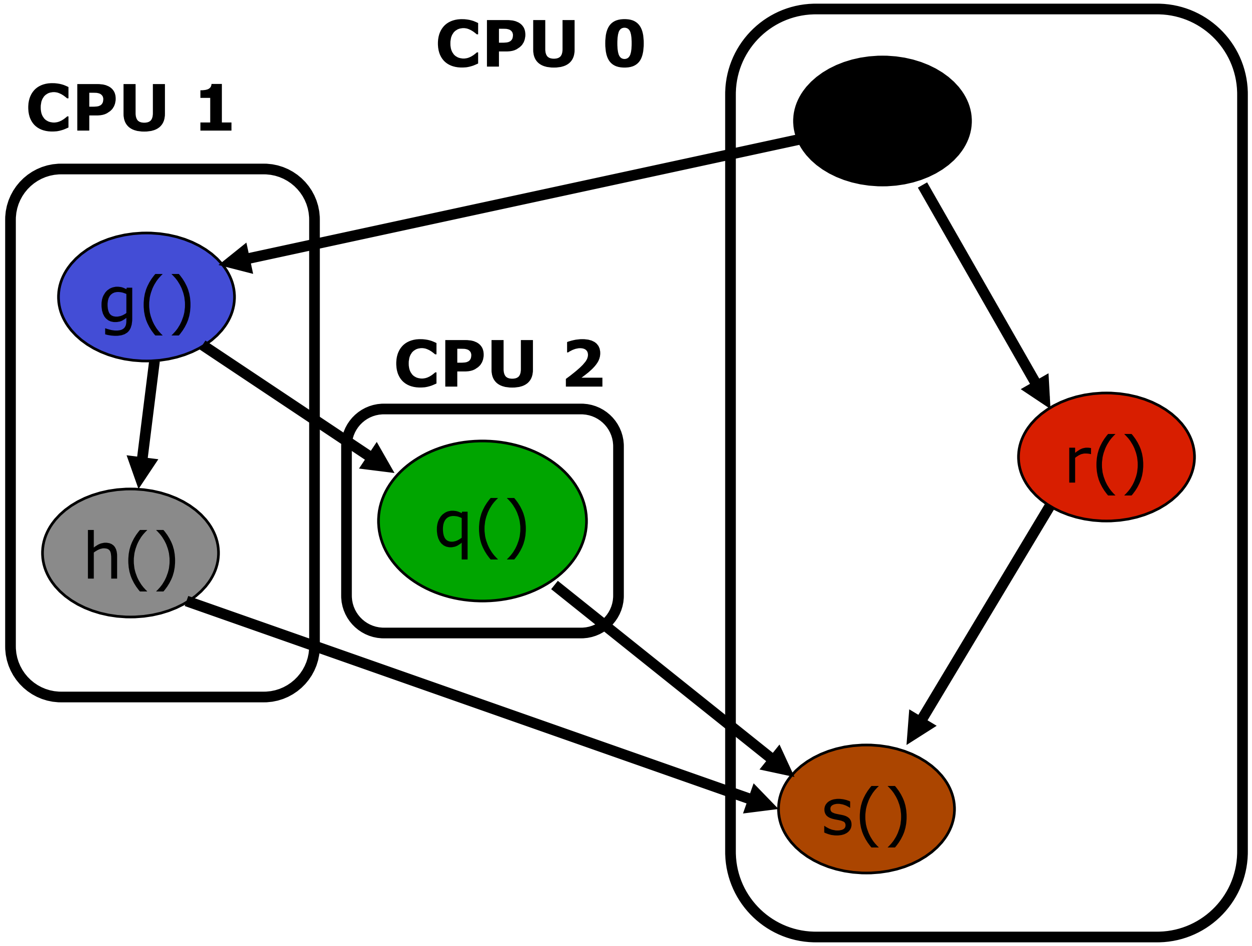
Slide Source: Intel Software College, Intel Corp.

Task Decomposition



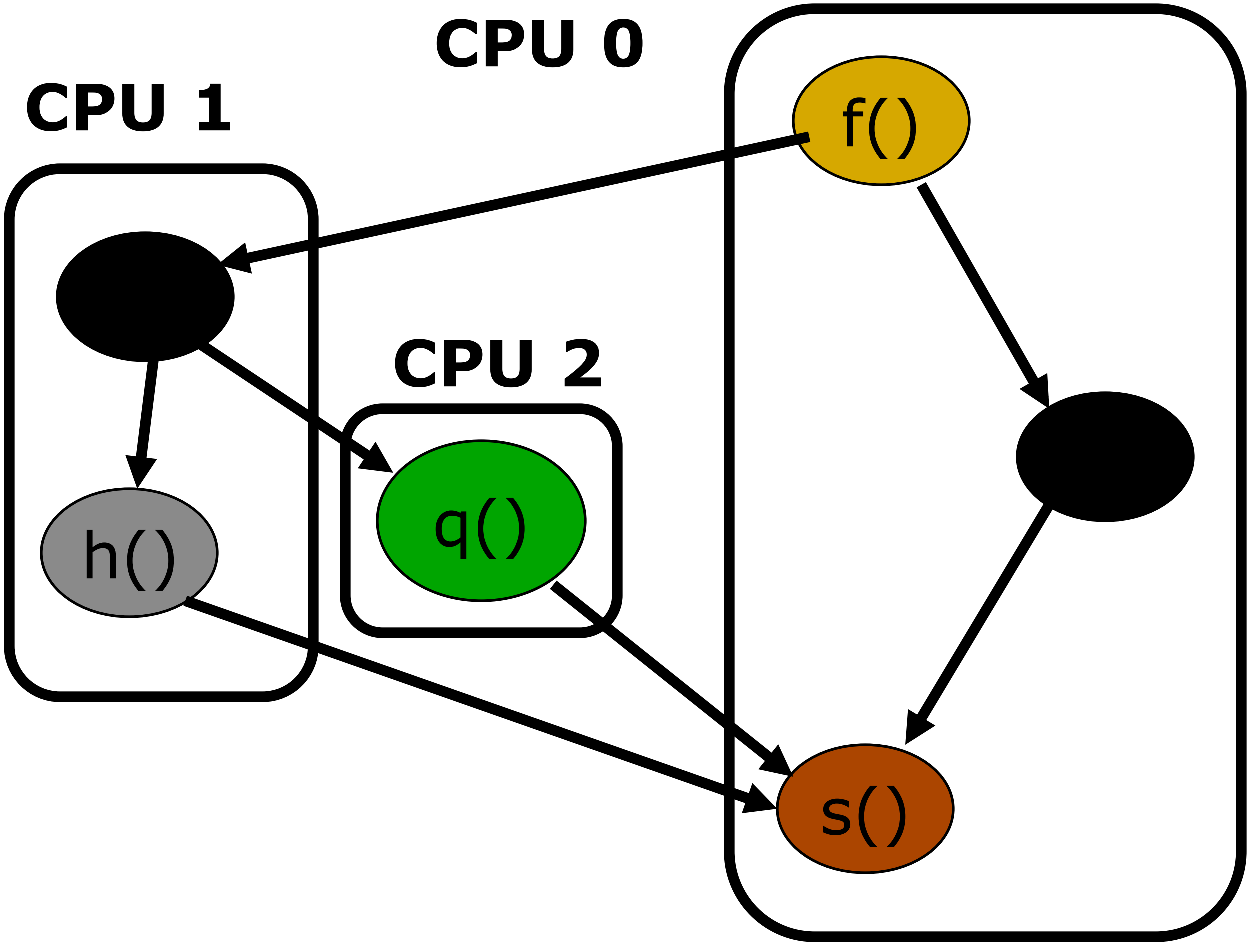
Slide Source: Intel Software College, Intel Corp.

Task Decomposition



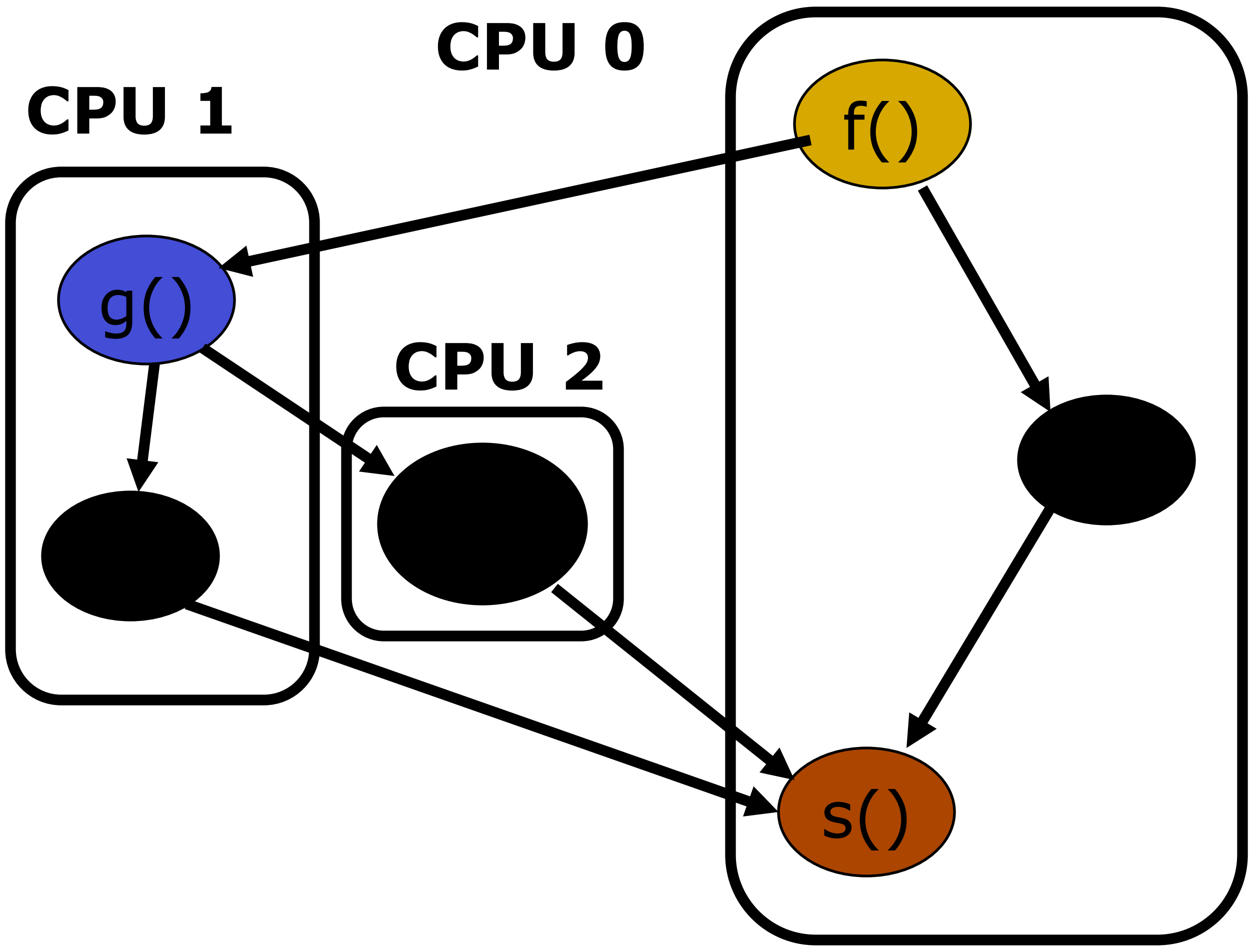
Slide Source: Intel Software College, Intel Corp.

Task Decomposition



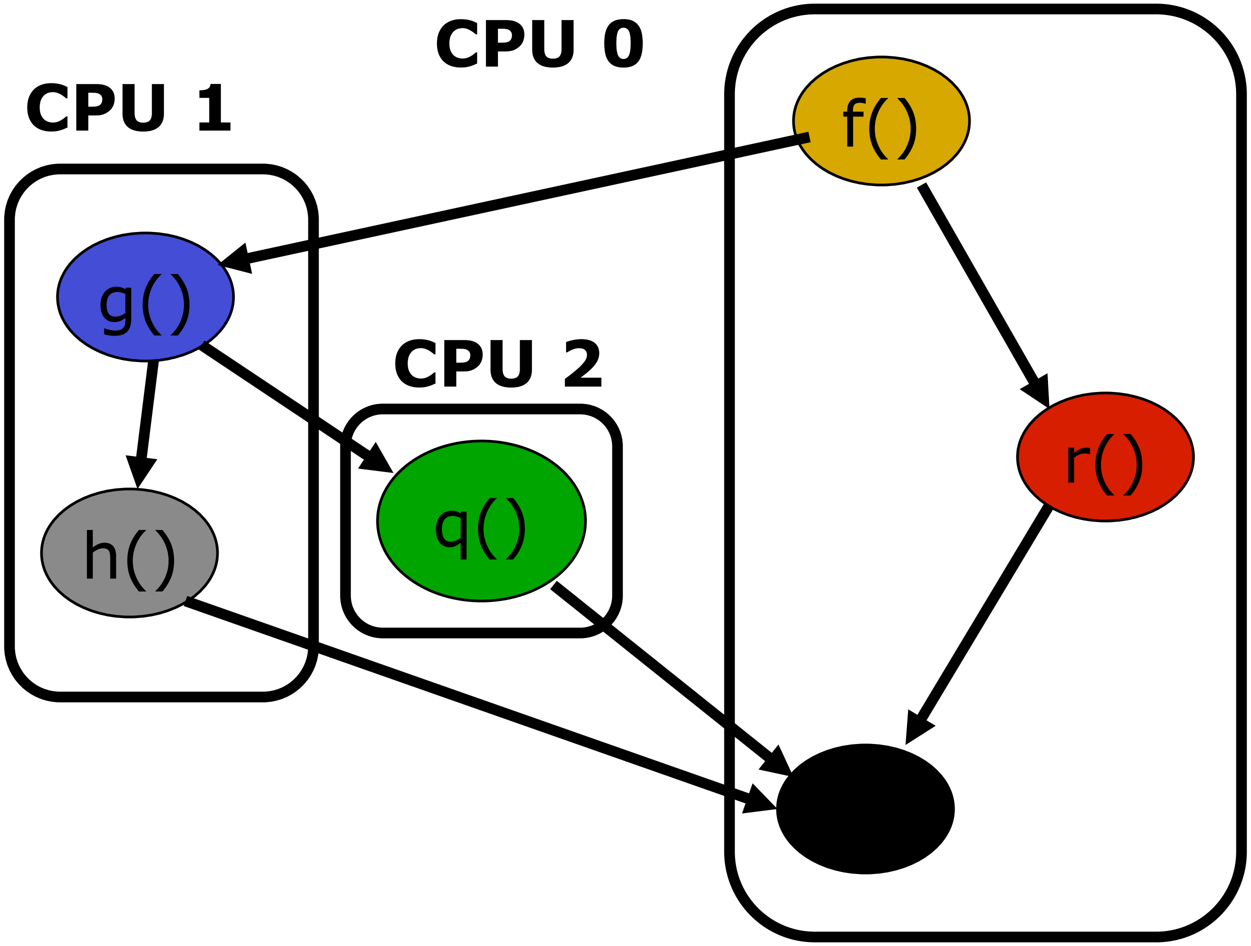
Slide Source: Intel Software College, Intel Corp.

Task Decomposition



Slide Source: Intel Software College, Intel Corp.

Task Decomposition



Slide Source: Intel Software College, Intel Corp.

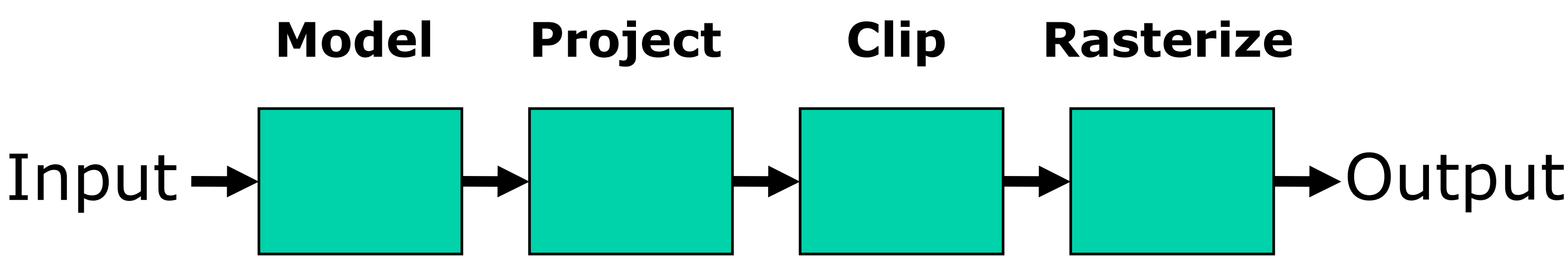
Task Decomposition Forces

- Flexibility in number and size of tasks
  - Task **size** should not be tied to a specific architecture
  - Parameterize number of tasks
    - Flexible to any architecture topology
- Efficiency
  - Task have enough computation to amortize creation costs
  - Sufficiently independent so dependencies are manageable
- Simplicity
  - Easy to understand and debug

Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

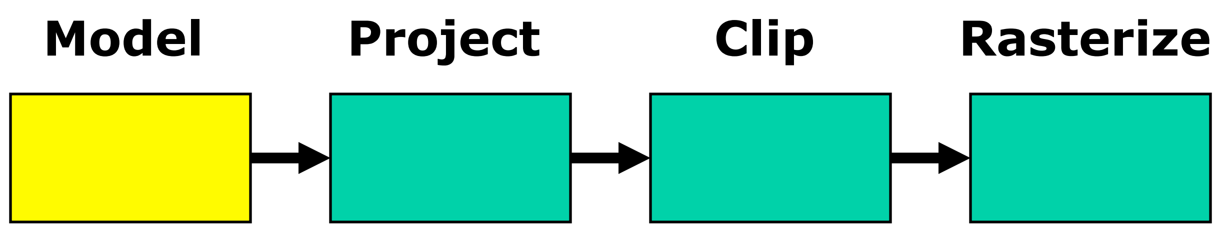
- Special kind of task decomposition
  - Data flows through a sequence of tasks
- "Assembly line" parallelism
- Example: 3D rendering in computer graphics



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

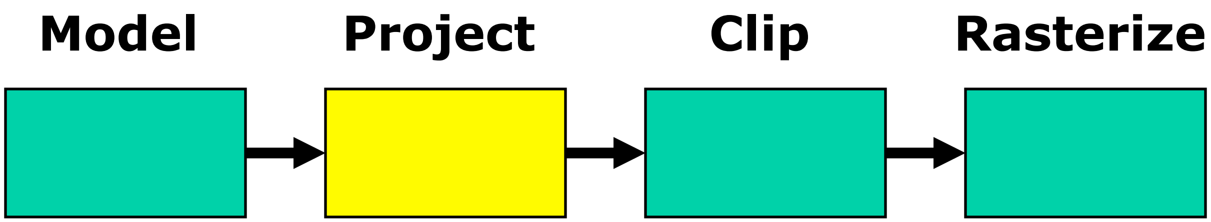
- Processing one data set (Step 1)



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

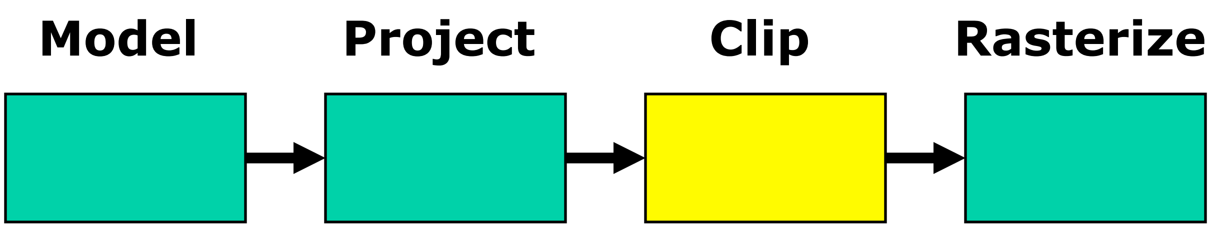
- Processing one data set (Step 2)



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

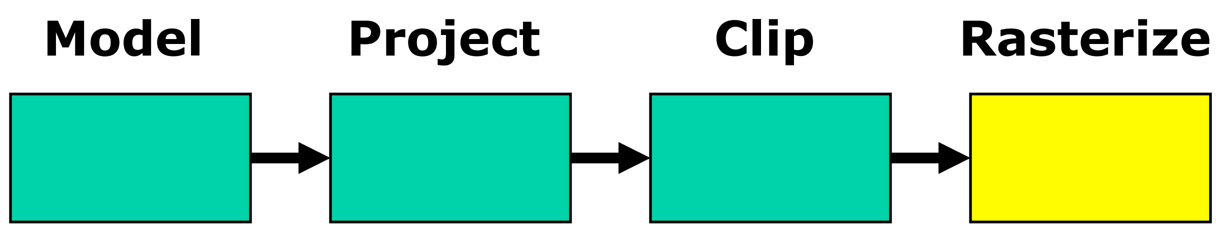
- Processing one data set (Step 3)



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

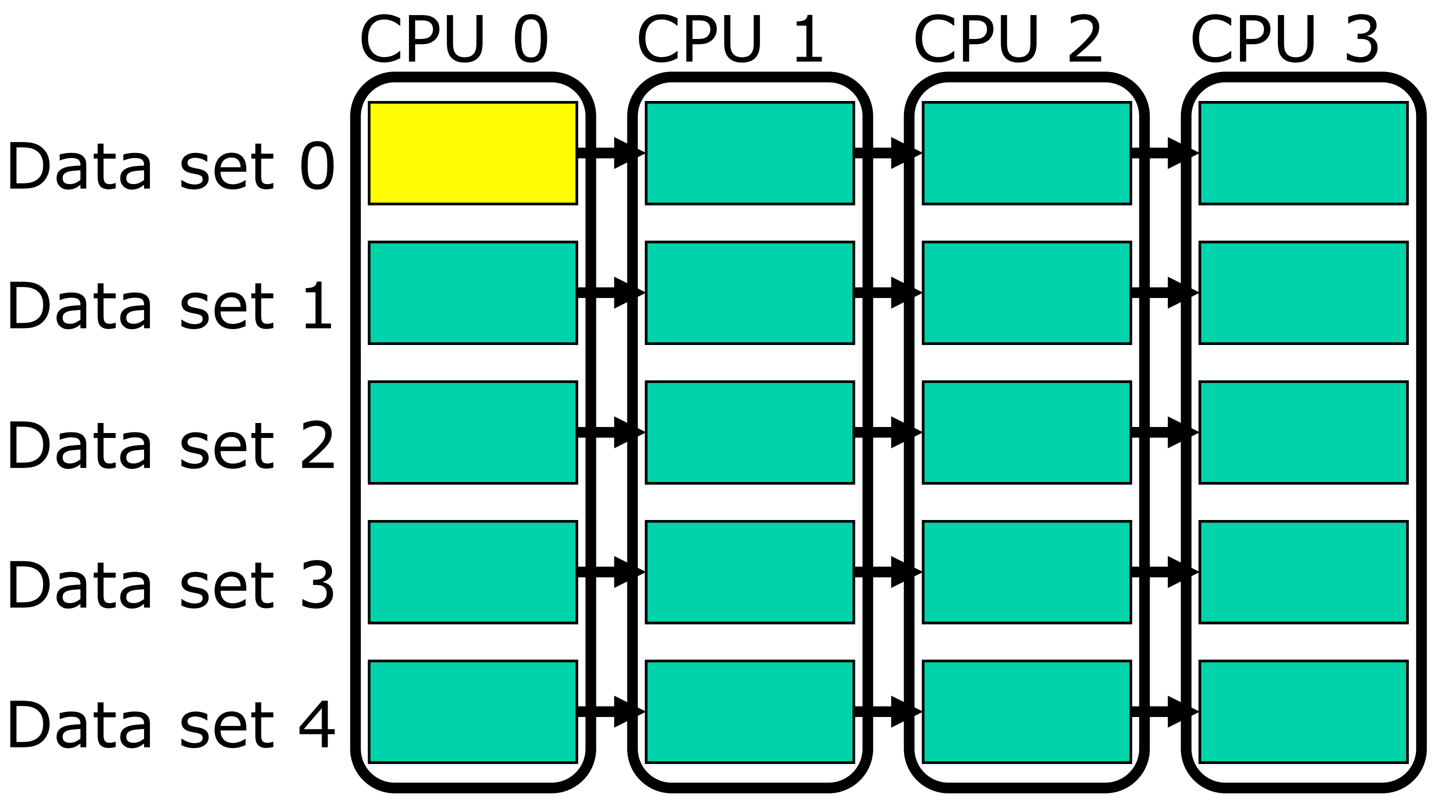
- Processing one data set (Step 4)
  - Pipeline processes 1 data set in 4 steps



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

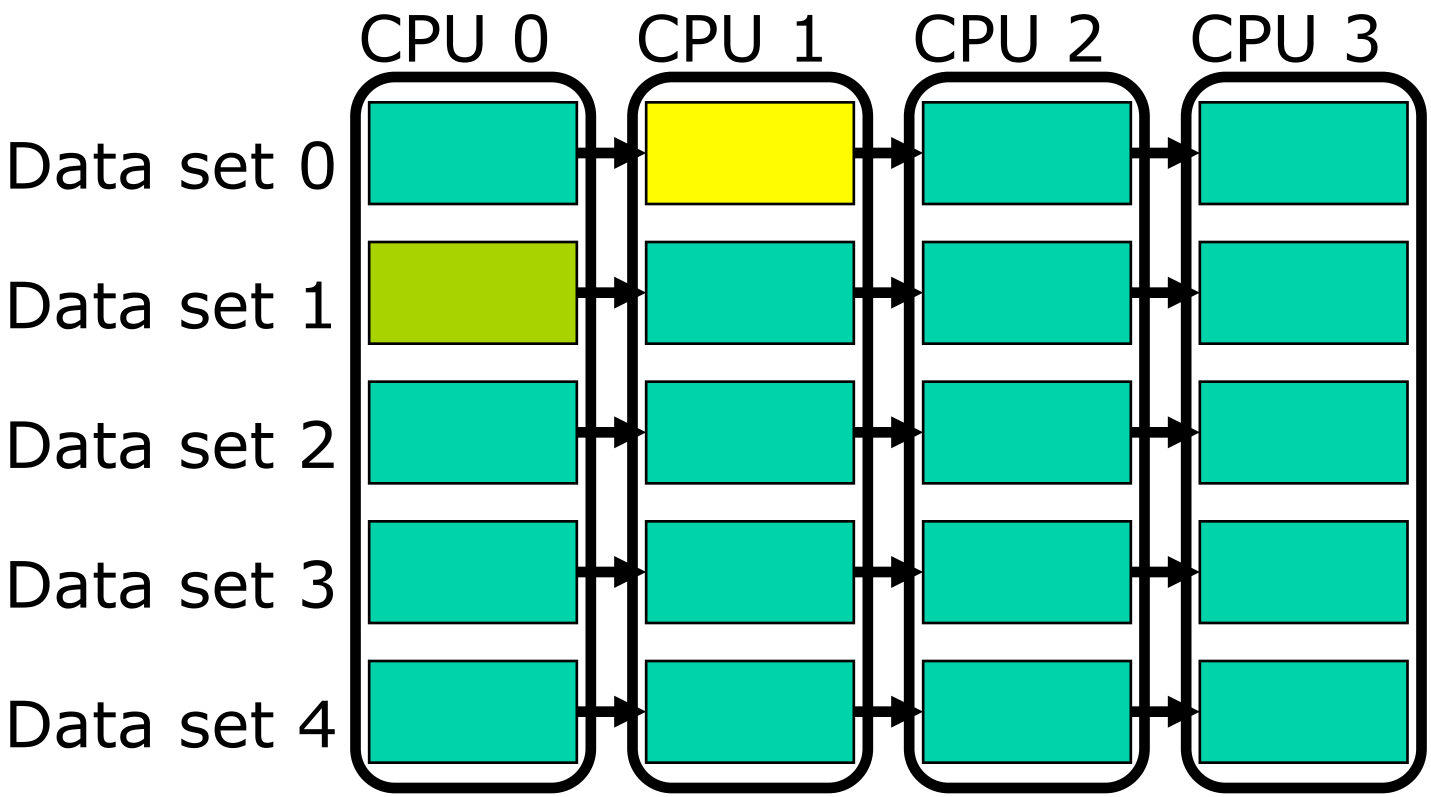
- Processing five data set (Step 1)



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

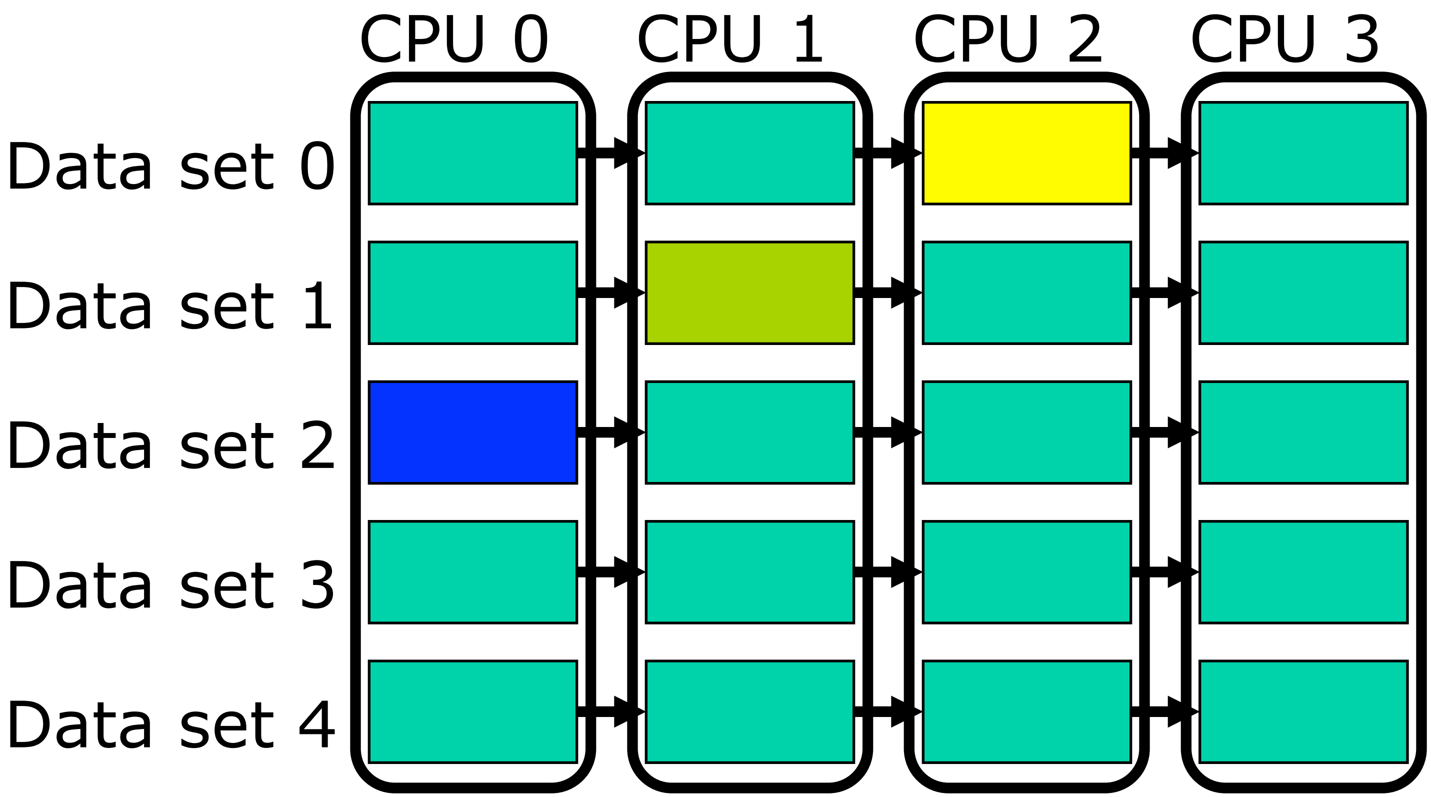
- Processing five data set (Step 2)



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

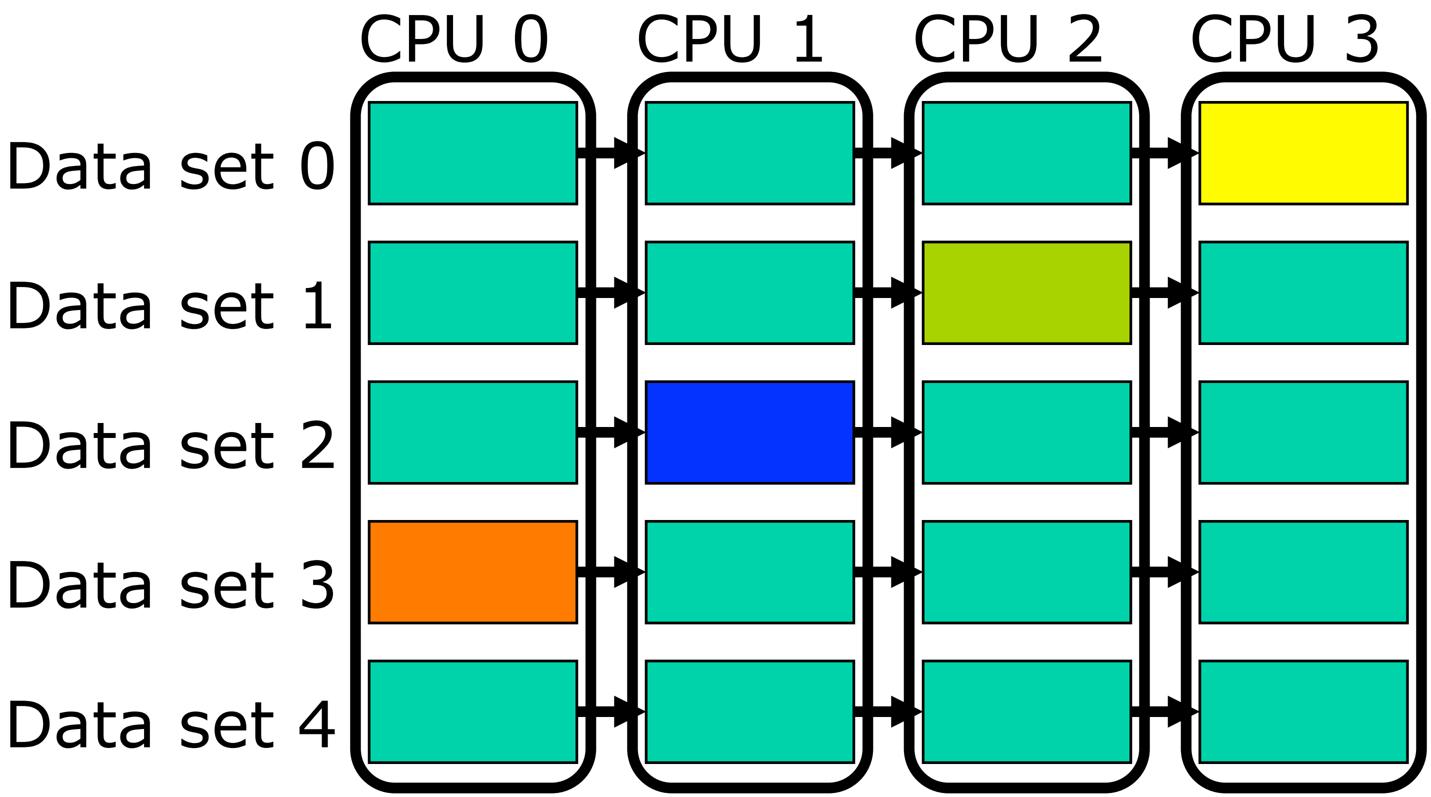
- Processing five data set (Step 3)



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

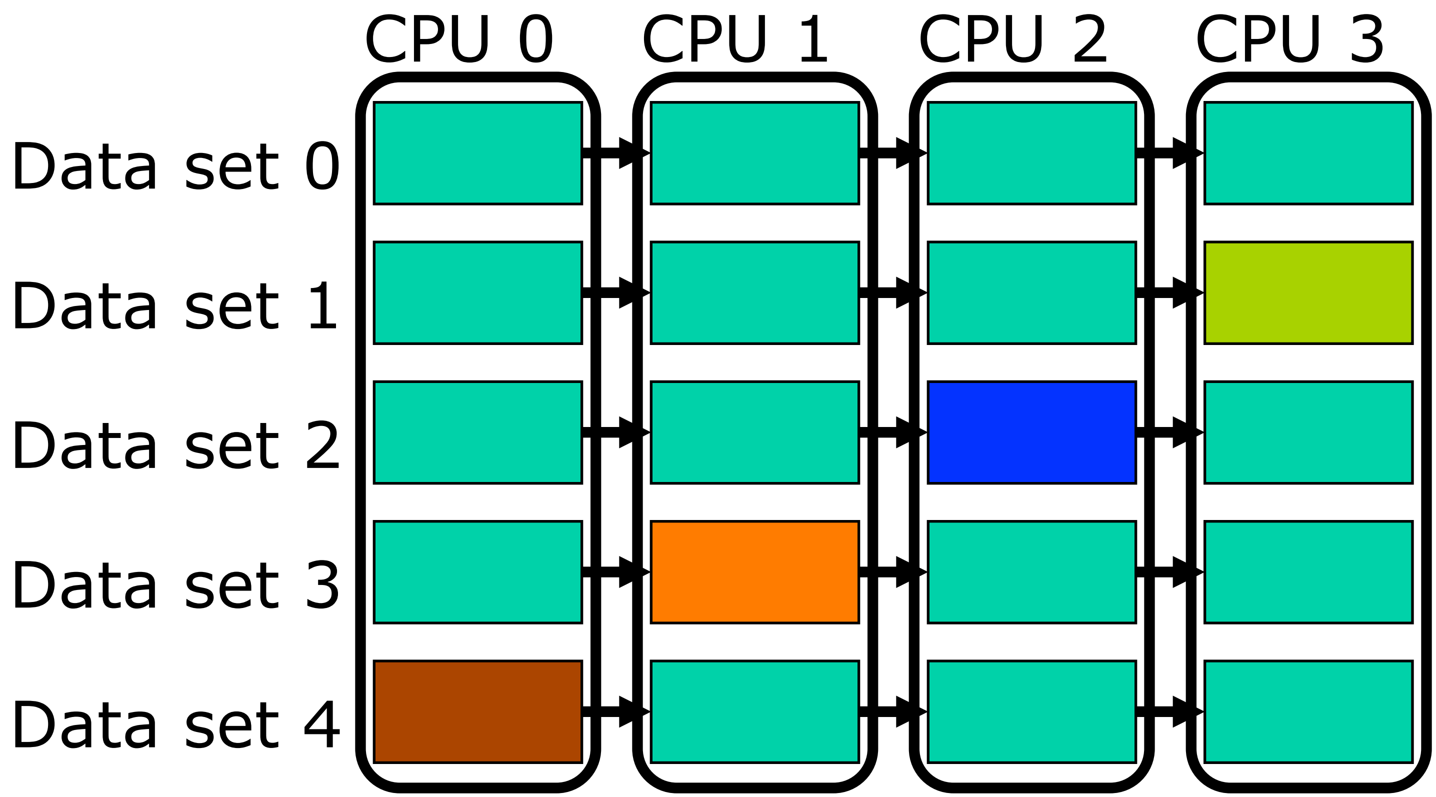
- Processing five data set (Step 4)



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

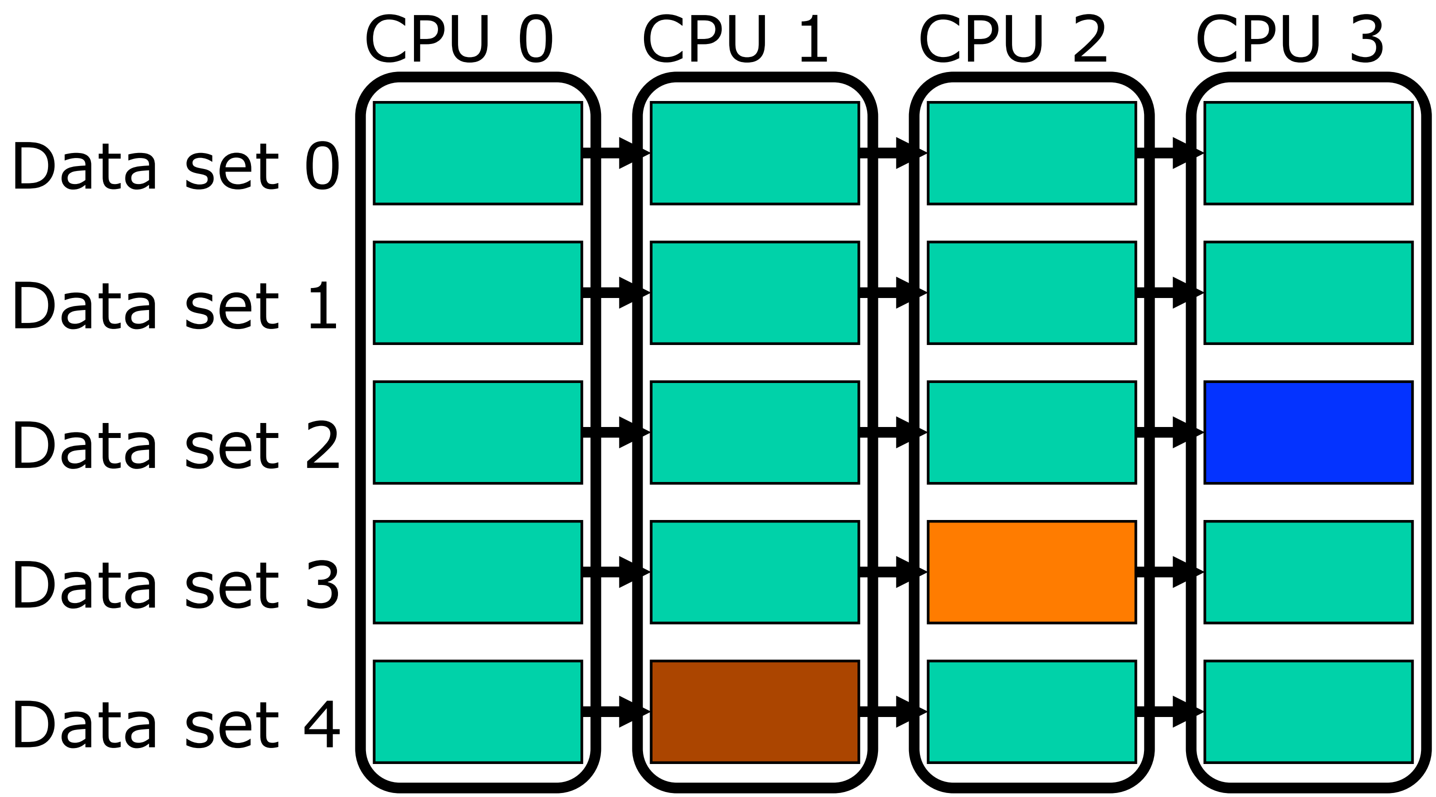
- Processing five data set (Step 5)



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

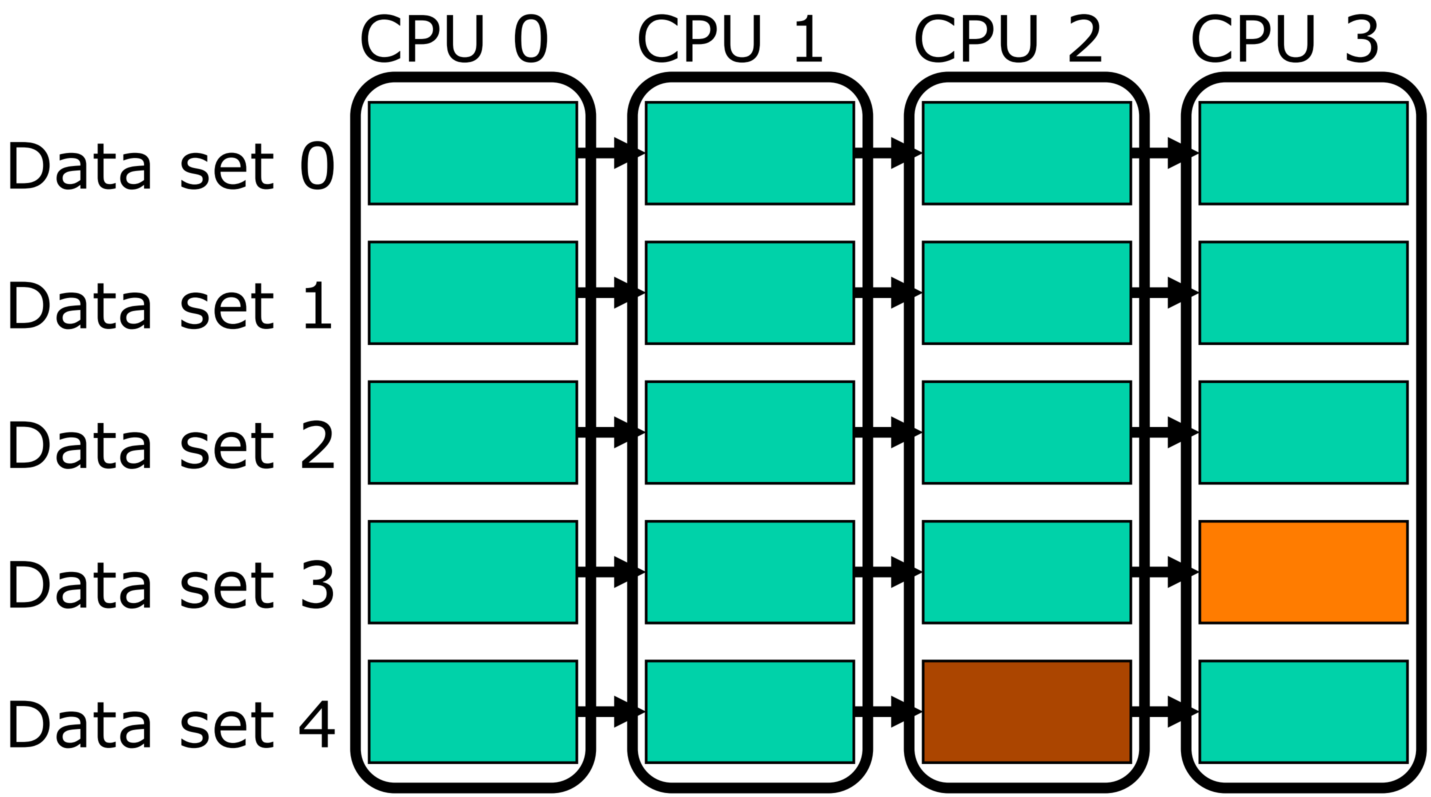
- Processing five data set (Step 6)



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

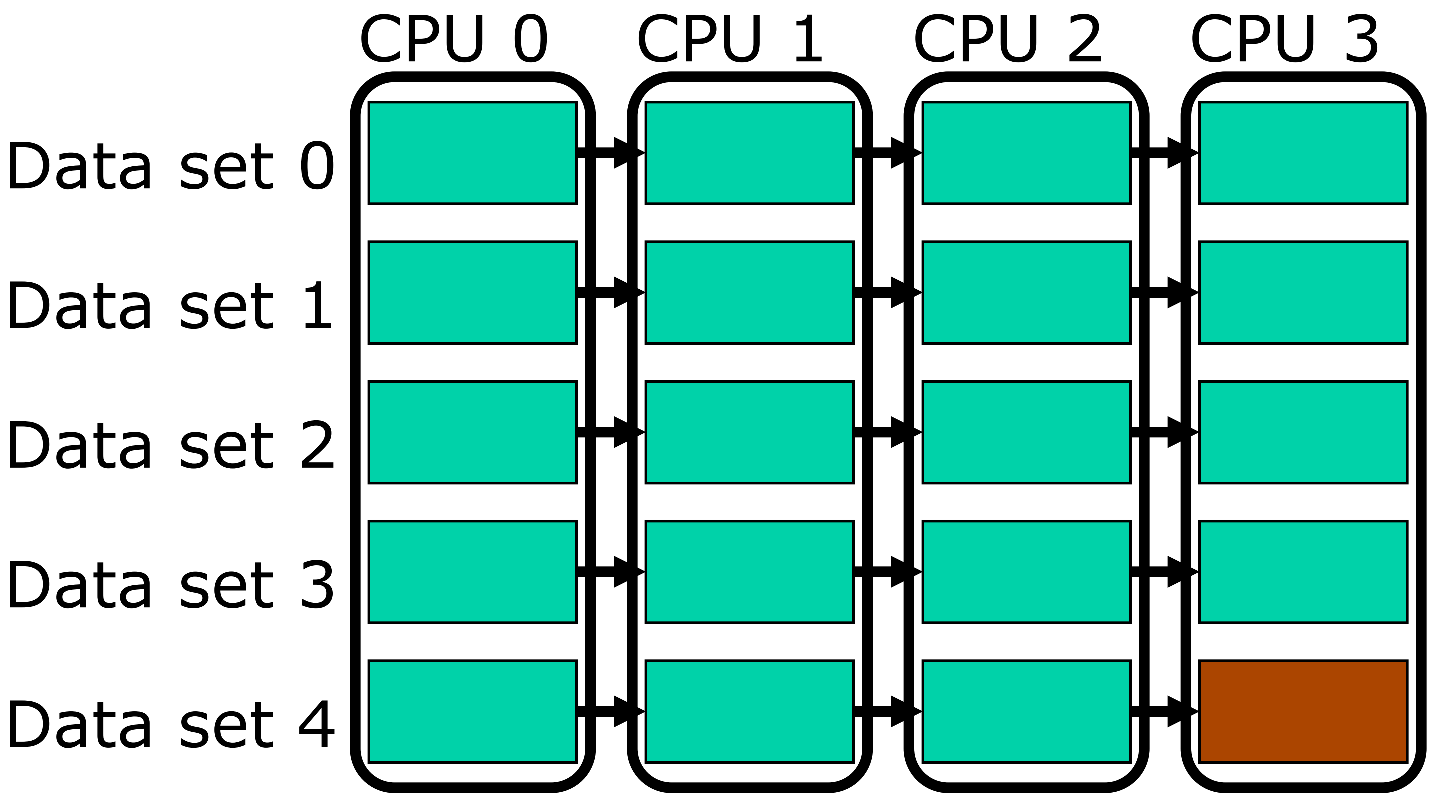
- Processing five data set (Step 7)



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition

- Processing five data set (Step 8)



Slide Source: Intel Software College, Intel Corp.

Pipeline Decomposition Forces

- Flexibility
  - Deeper pipelines are better
    - Will scale better and can later merge pipeline stages
- Efficiency
  - Stages of pipeline should not cause bottleneck
  - Even amount of work in each stage
- Simplicity
  - More pipeline stages break down problem into more manageable chunks of code

Slide Source: Intel Software College, Intel Corp.

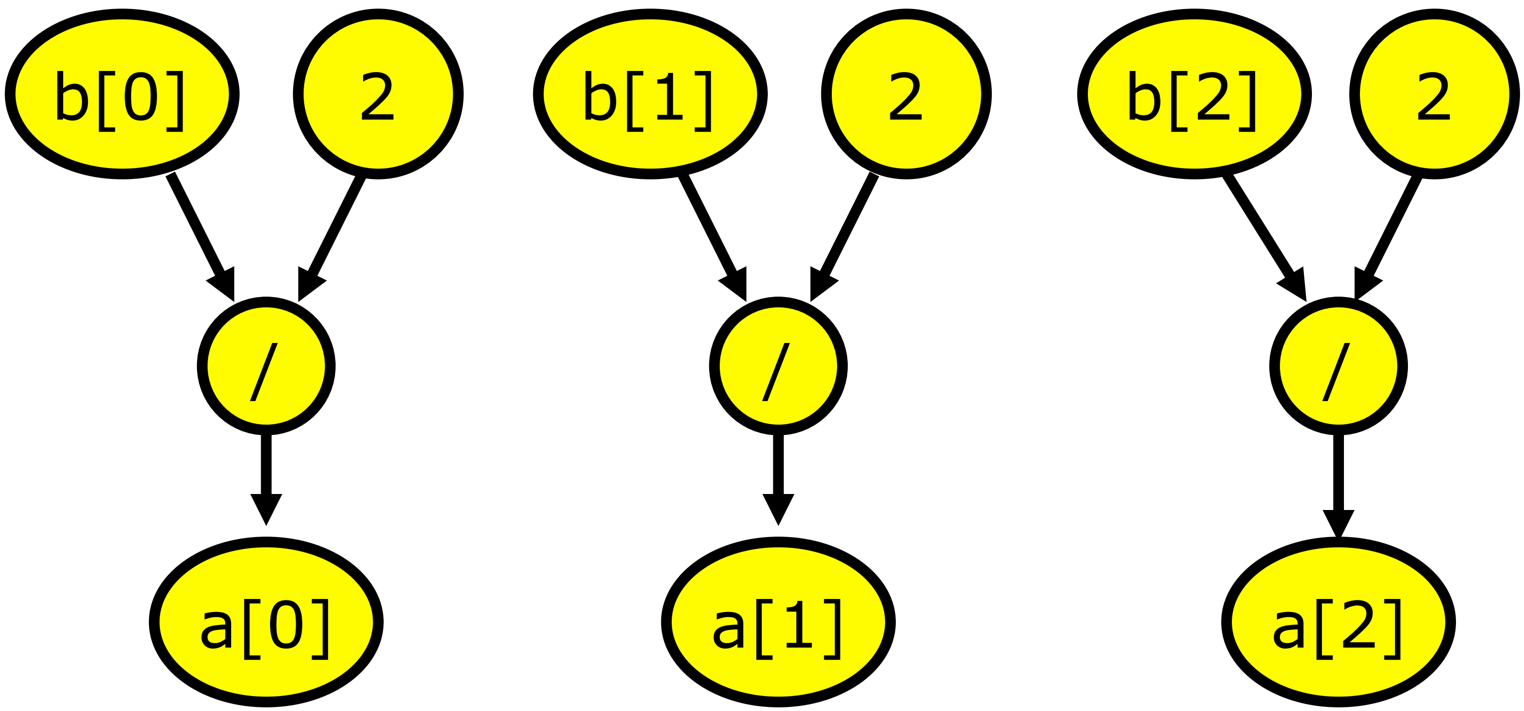
Dependency Analysis

- Control and Data Dependences
- Dependence Graph
  - Graph = (nodes, edges)
  - Node for each
    - Variable assignment
    - Constant
    - Operator or Function call
  - Edge indicates use of variables and constants
    - Data flow
    - Control flow

Slide Source: Intel Software College, Intel Corp.

Dependency Analysis

```
for (i = 0; i < 3; i++)
   a[i] = b[i] / 2.0;
```

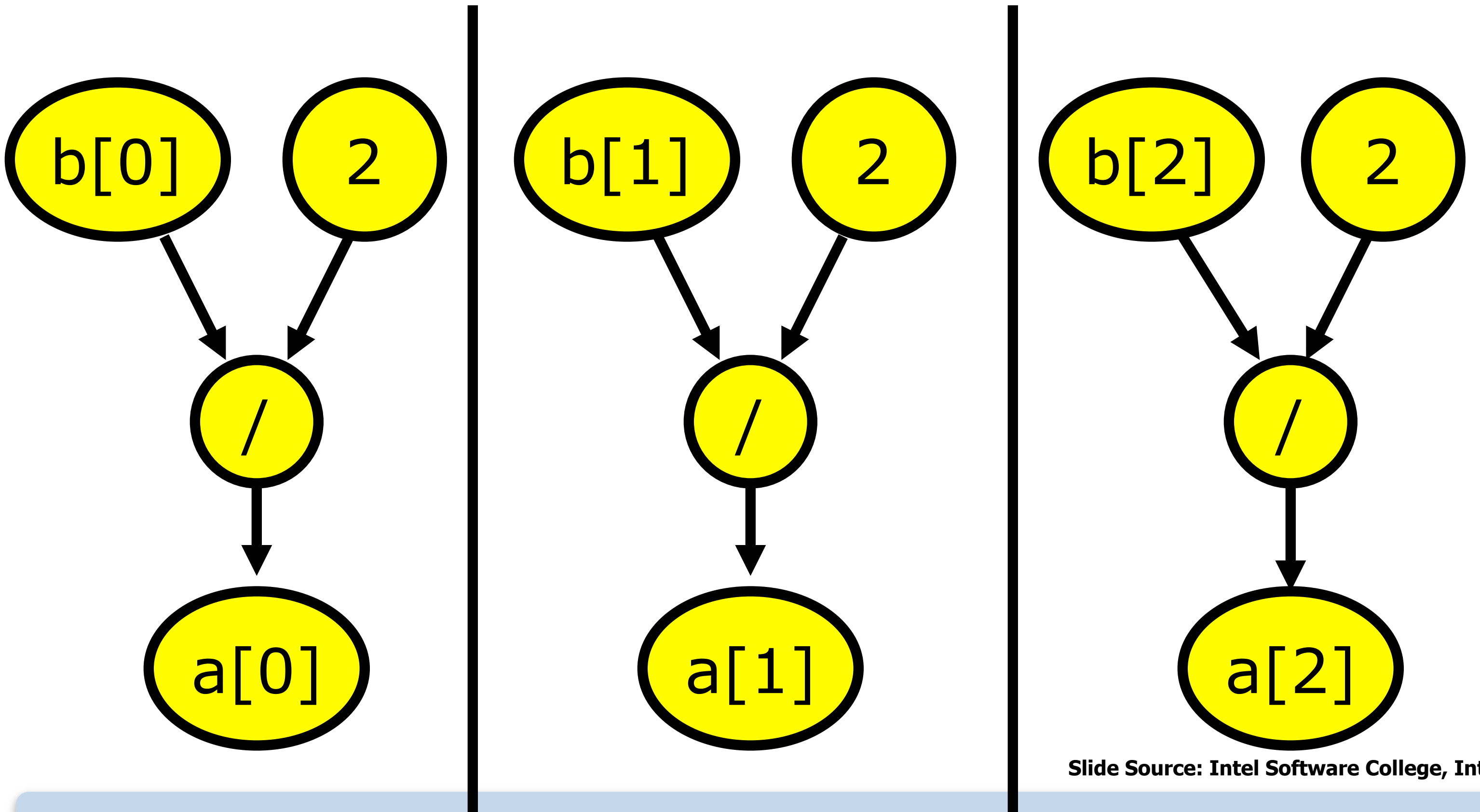


Slide Source: Intel Software College, Intel Corp.

Dependency Analysis

```
for (i = 0; i < 3; i++)
  a[i] = b[i] / 2.0;
```

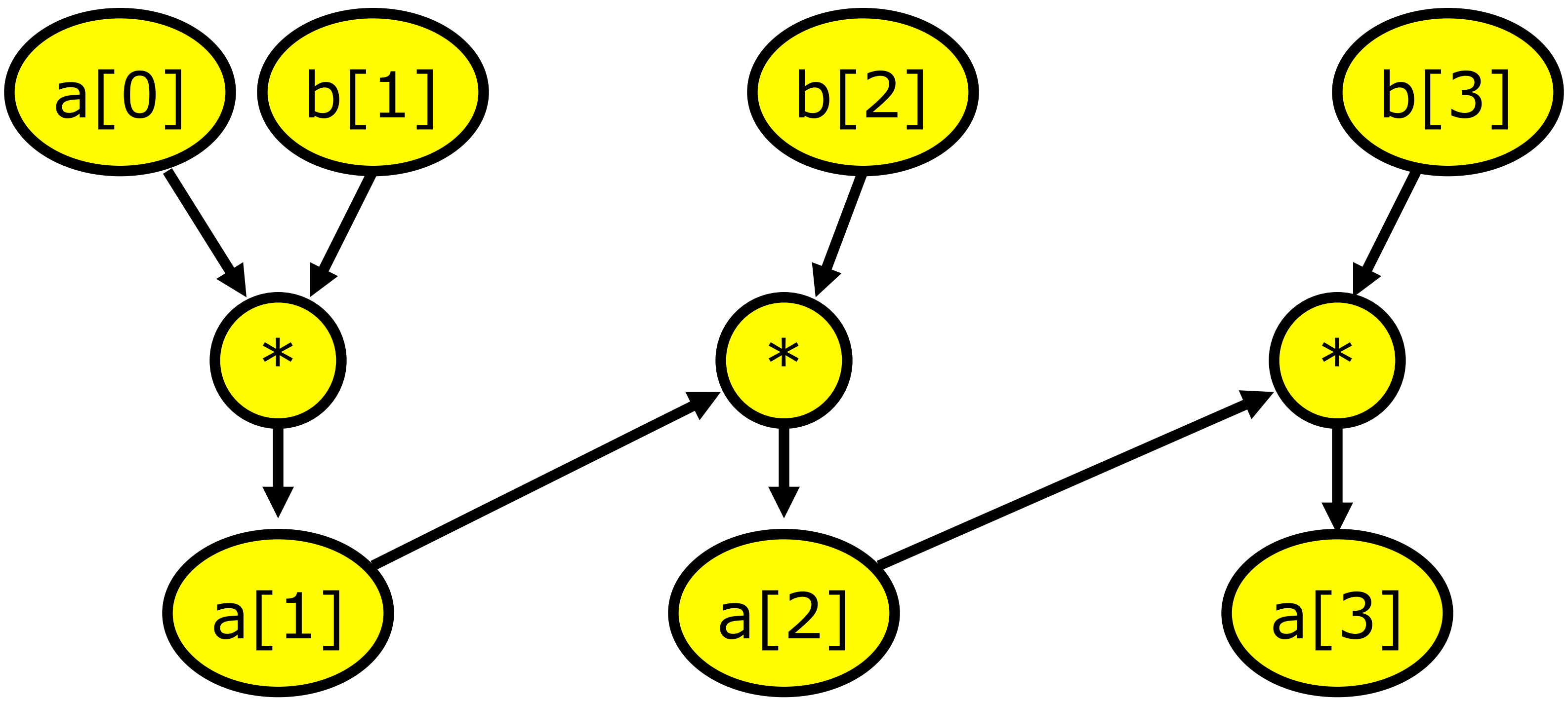
Domain decomposition possible



Slide Source: Intel Software College, Intel Corp.

Dependency Analysis

```
for (i = 1; i < 4; i++)
   a[i] = a[i-1] * b[i];
```

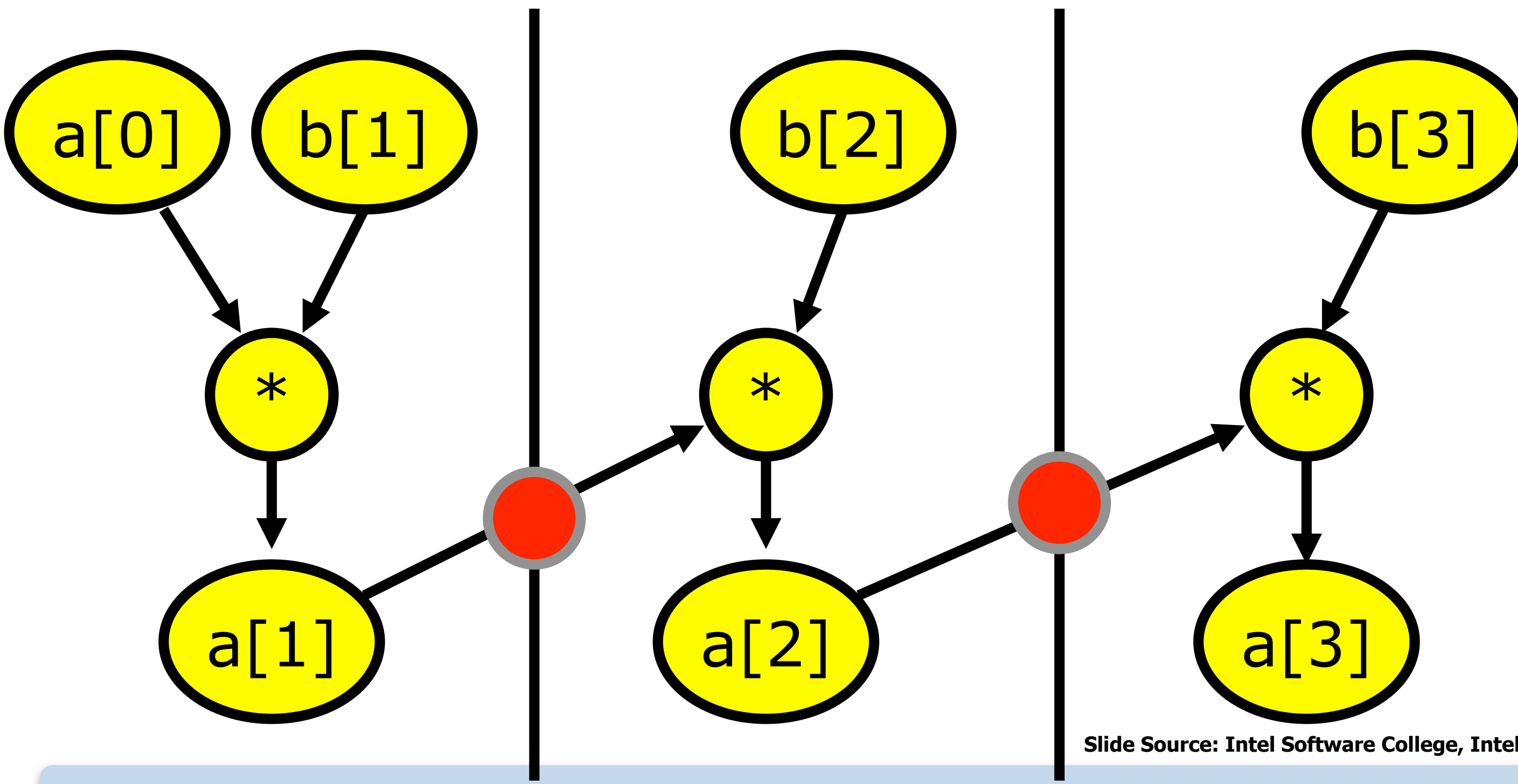


Slide Source: Intel Software College, Intel Corp.

Dependency Analysis

```
for (i = 1; i < 4; i++)
   a[i] = a[i-1] * b[i];
```

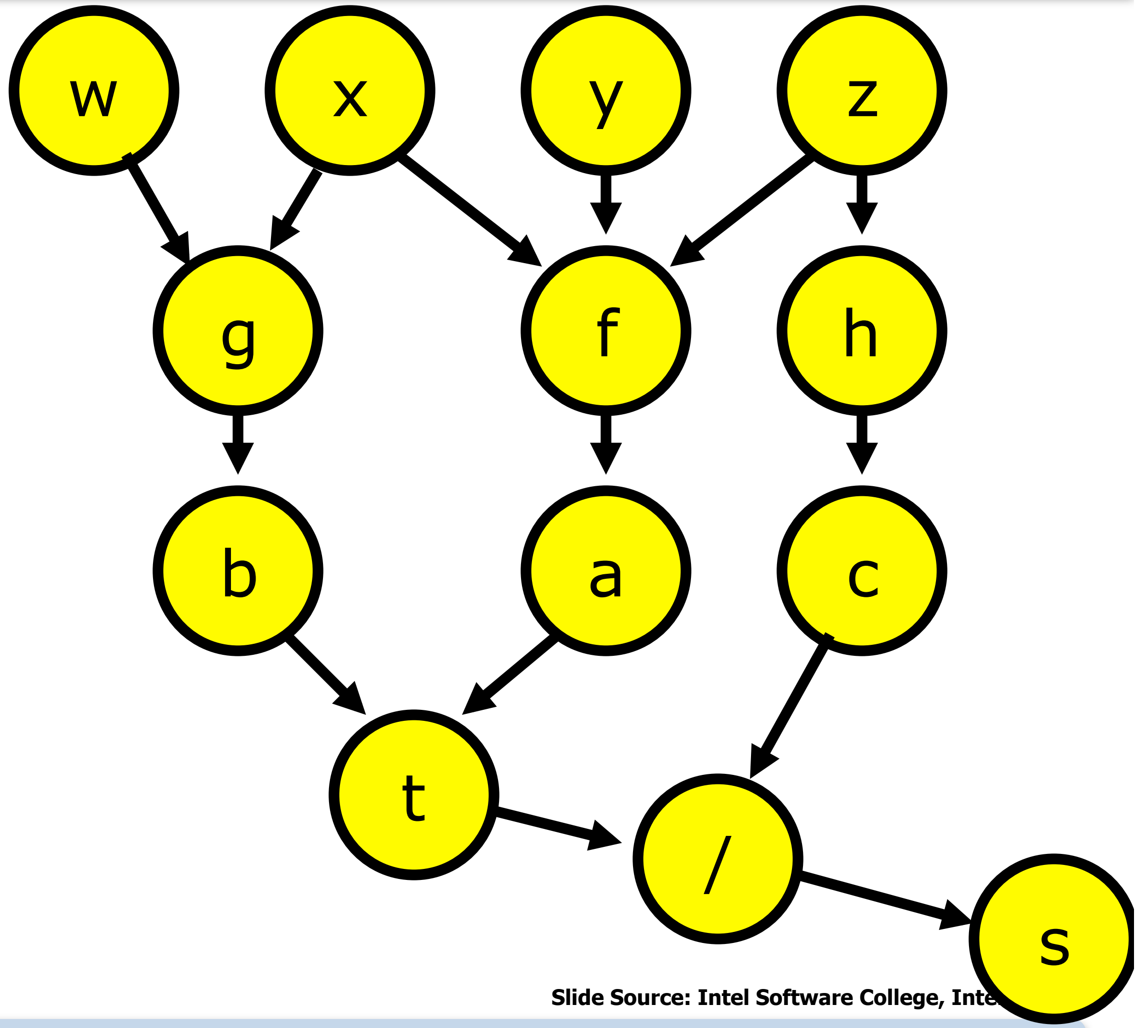
No domain decomposition



Slide Source: Intel Software College, Intel Corp.

Dependency Analysis

```
a = f(x, y, z);
b = g(w, x);
t = a + b;
c = h(z);
s = t / c;
```

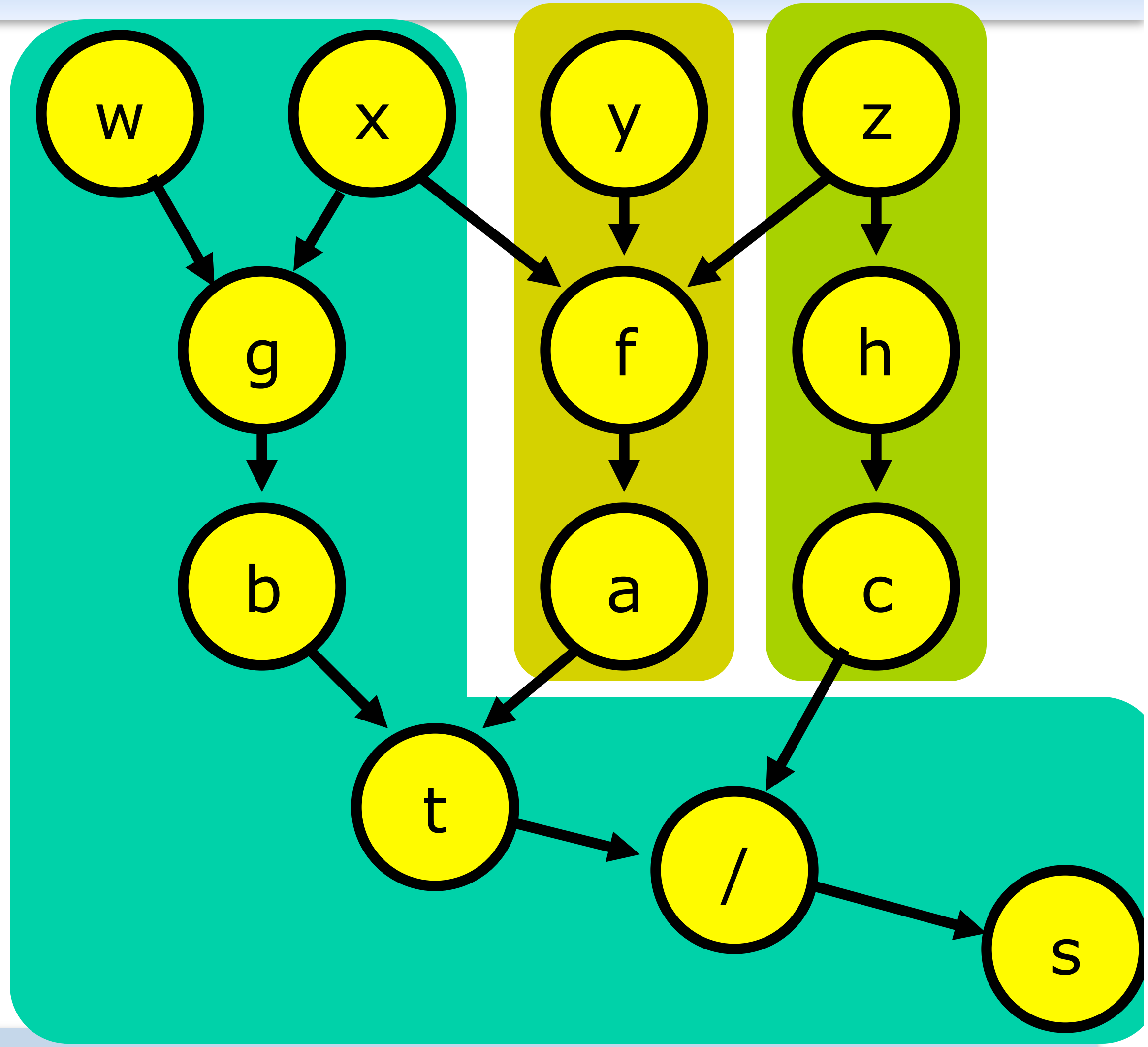


Slide Source: Intel Software College, Inte

Dependency Analysis

a = f(x, y, z);
b = g(w, x);
t = a + b;
c = h(z);
s = t / c;

Task decomposition with 3 CPUs.



Evaluate Design

- Is the design good enough
  - YES - move to next design space
  - NO - re-evaluate previous patterns
- Forces
  - Suitability to target platform
    - Should not depend on underlying architecture
  - Design quality
    - Trade-offs between simplicity, flexibility, and efficiency
      - Pick any two!
  - Preparation for next phase
    - Understand design to help in next phase: Algorithm Structure

Lecture Overview

- Design Patterns for Parallel Programs
  - Finding Concurrency
  - **Algorithmic Structure**
  - Supporting Structures
  - Implementation Mechanisms

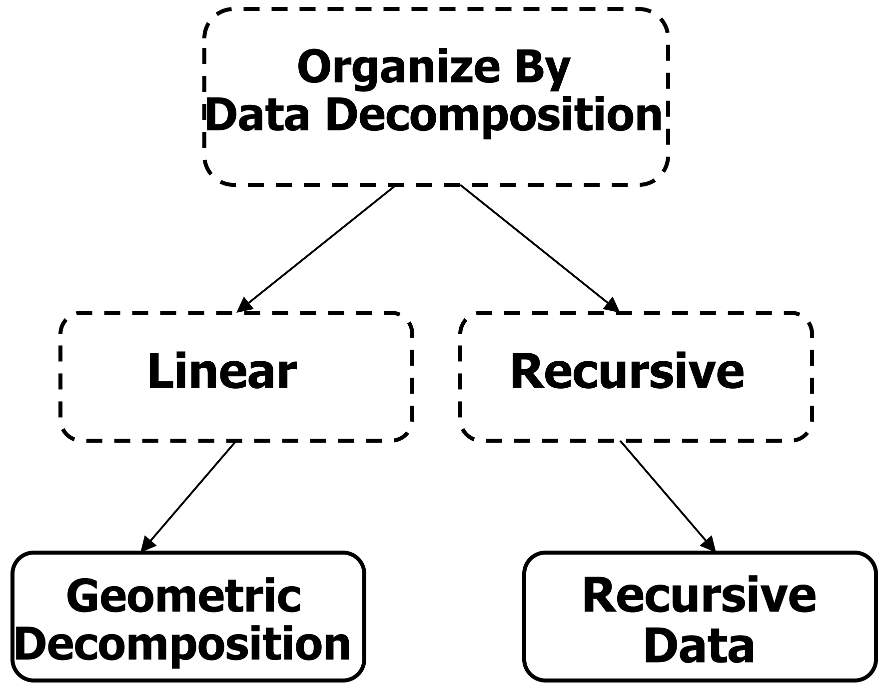
Algorithm Structure Patterns

- Given a set of concurrent tasks, what's next?
- Important questions based on target platform:
  - How many cores will your algorithm support?
    - Consider the order of magnitude
  - How expensive is sharing?
    - Architectures have different communication costs
  - Is design constrained to hardware?
    - Software typically outlives hardware
    - Flexible to adapt to different architectures
  - Does algorithm map well to programming environment?
    - Consider language/library available

How to Organize Concurrency

- ***Major organizing principle*** implied by concurrency
- Organization by data decomposition
  - Geometric Decomposition
  - Recursive Data
- Organization by task decomposition
  - Task Parallelism
  - Divide and Conquer
- Organization by flow of data
  - Pipeline
  - Event-Based coordination

Organize by Data

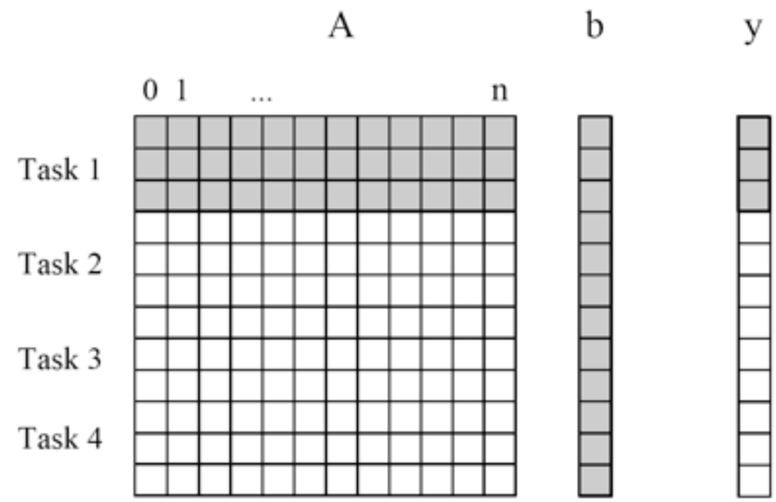


Organize by Data

- Operations on core data structure
- Geometric Decomposition
- Recursive Data

Geometric Deomposition

- Arrays and other linear structures
  - Divide into contiguous substructures
- Example: Matrix multiply
  - Data-centric algorithm and linear data structure (array) implies geometric decomposition

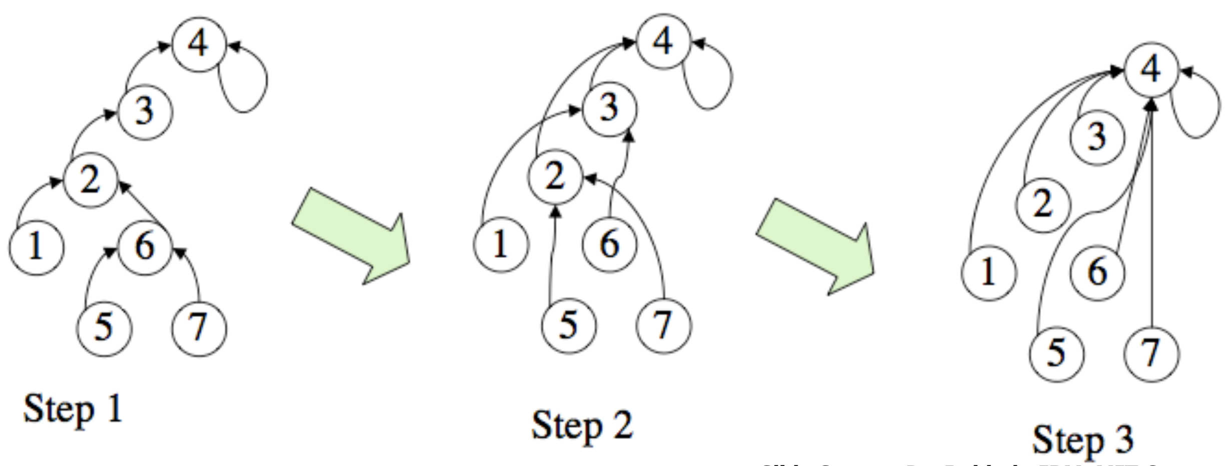


Recursive Data

- Lists, trees, and graphs
  - Structures where you would use divide-and-conquer
- May seem that can only move sequentially through data structure
  - But, there are ways to expose concurrency

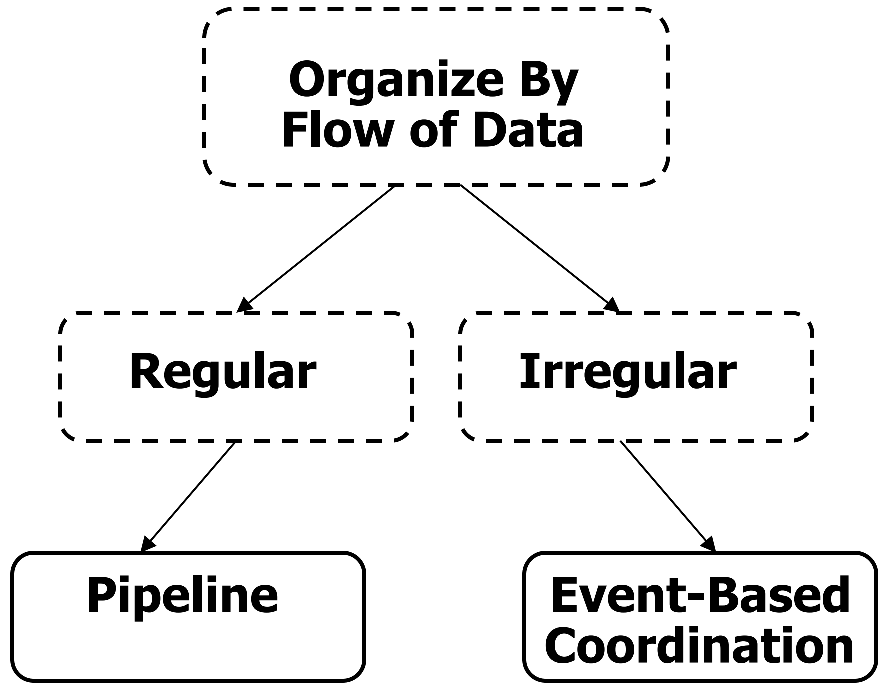
Recursive Data Example

- Find the Root: Given a forest of directed trees find the root of each node
  - Parallel approach: For each node, find its successor's successor
  - Repeat until no changes
    - O(log n) vs O(n)



Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007

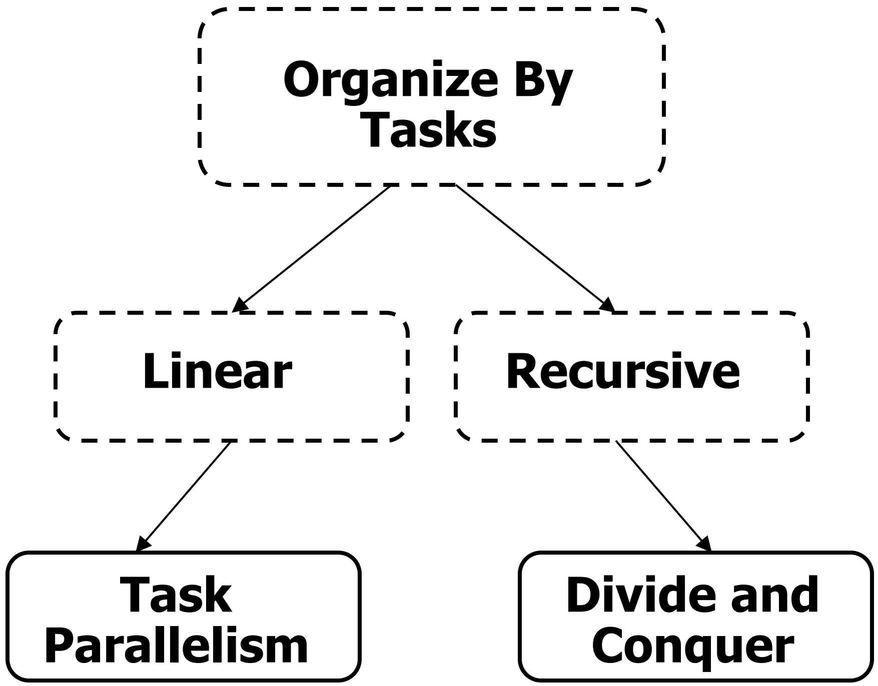
Organize by Flow of Data



Organize by Flow of Data

- Computation can be viewed as a flow of data going through a sequence of stages
- Pipeline: one-way predictable communication
- Event-based Coordination: unrestricted unpredictable communication

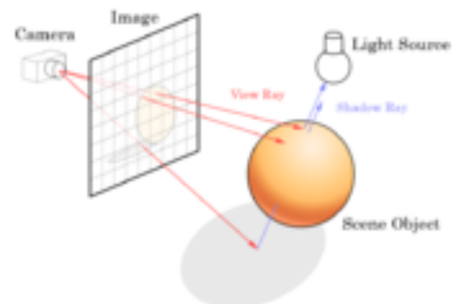
Organize by Tasks



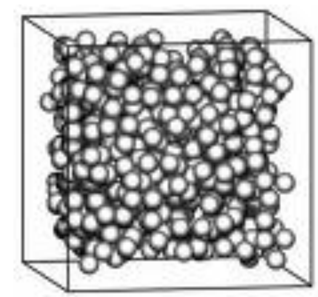
Task Parallelism

- Tasks are linear (no structure or hierarchy)
- Can be completely independent
  - Embarrassingly parallel
- Can have some dependencies
- Common factors
  - Tasks are associated with loop iterations
  - All tasks are known at beginning
  - All tasks must complete
  - However, there are exeptions to all of these

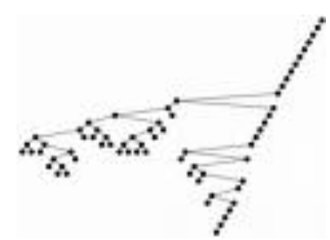
Task Parallelism (Examples)



- Ray Tracing
    - Each ray is separate and independent



- Molecular Dynamics
    - Vibrational, rotational, nonbonded forces are independent for each atom

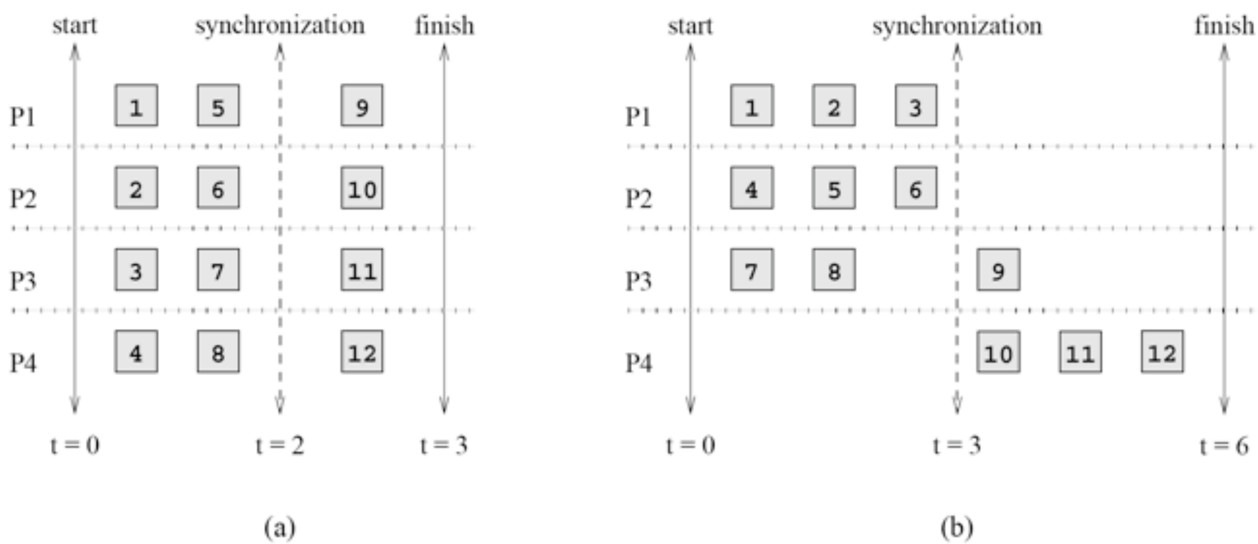


- Branch-and-bound computations
    - Repeatedly divide into smaller solution spaces until solution found
    - Tasks weakly dependent through queue

Task Parallelism

- Three Key Elements
- Is Task definition adequate?
  - Number of tasks and their computation
- Schedule
  - Load Balancing
- Dependencies
  - Removable
  - Separable by replication

Schedule

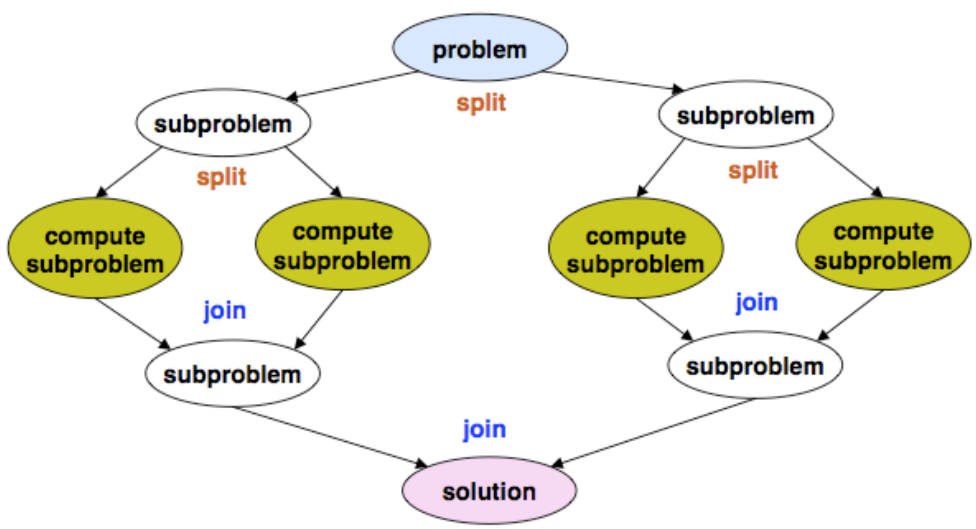


Not all schedules of task equal in performance.

Slide Source: Introduction to Parallel Computing, Grama et al.

Divide and Conquer

- Recursive Program Structure
  - Each subproblem generated by split becomes a task
- Subproblems may not be uniform
  - Requires load balancing



Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007

Pipeline performance

- Concurrency limited by pipeline depth
  - Balance computation and communication (architecture dependent)
- Stages should be equally computationally intensive
  - Slowest stage creates bottleneck
  - Combine lightly loaded stages or decompose heavily-loaded stages
- Time to fill and drain pipe should be small

Read for More Information

- Reengineering for Parallelism: An Entry Point for PLPP (Pattern Language for Parallel Programming) for Legacy Applications http://www.cise.ufl.edu/research/ParallelPatterns/plop2005.pdf