Parallel & Distributed Computing

Basic Parallelization Methodology

Lecture 3, Spring 2014

Instructor: 罗国杰

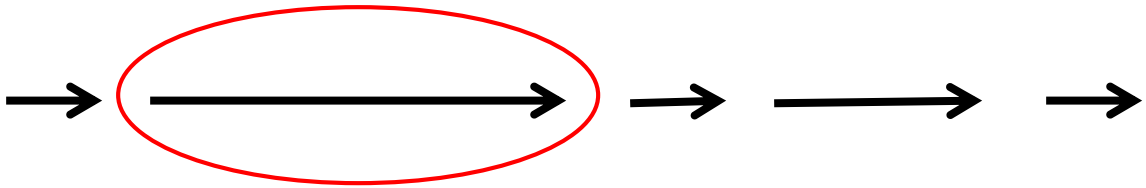
gluo@pku.edu.cn

Outline

- Incremental parallelization
  - Study a sequential program (or code segment)
  - Look for bottlenecks & opportunities for parallelism
  - Try to keep all processors busy doing useful work
- Design methodology for parallel algorithm, and
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping
- Use of dependency graph

Incremental parallelization

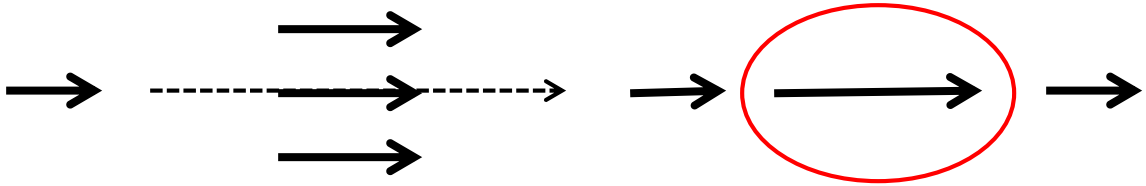
- Study a sequential program (or code segment)
- Look for bottlenecks & opportunities for parallelism
- Try to keep all processors busy doing useful work



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Incremental parallelization

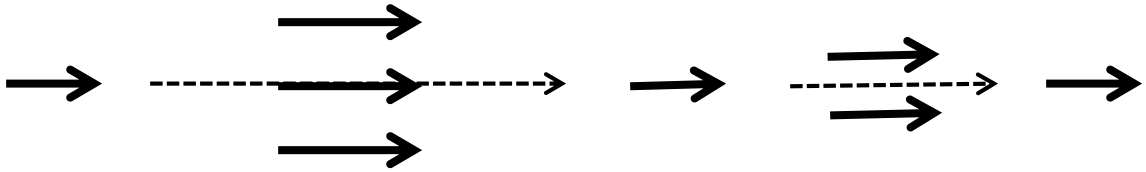
- Study a sequential program (or code segment)
- Look for bottlenecks & opportunities for parallelism
- Try to keep all processors busy doing useful work



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Incremental parallelization

- Study a sequential program (or code segment)
- Look for bottlenecks & opportunities for parallelism
- Try to keep all processors busy doing useful work

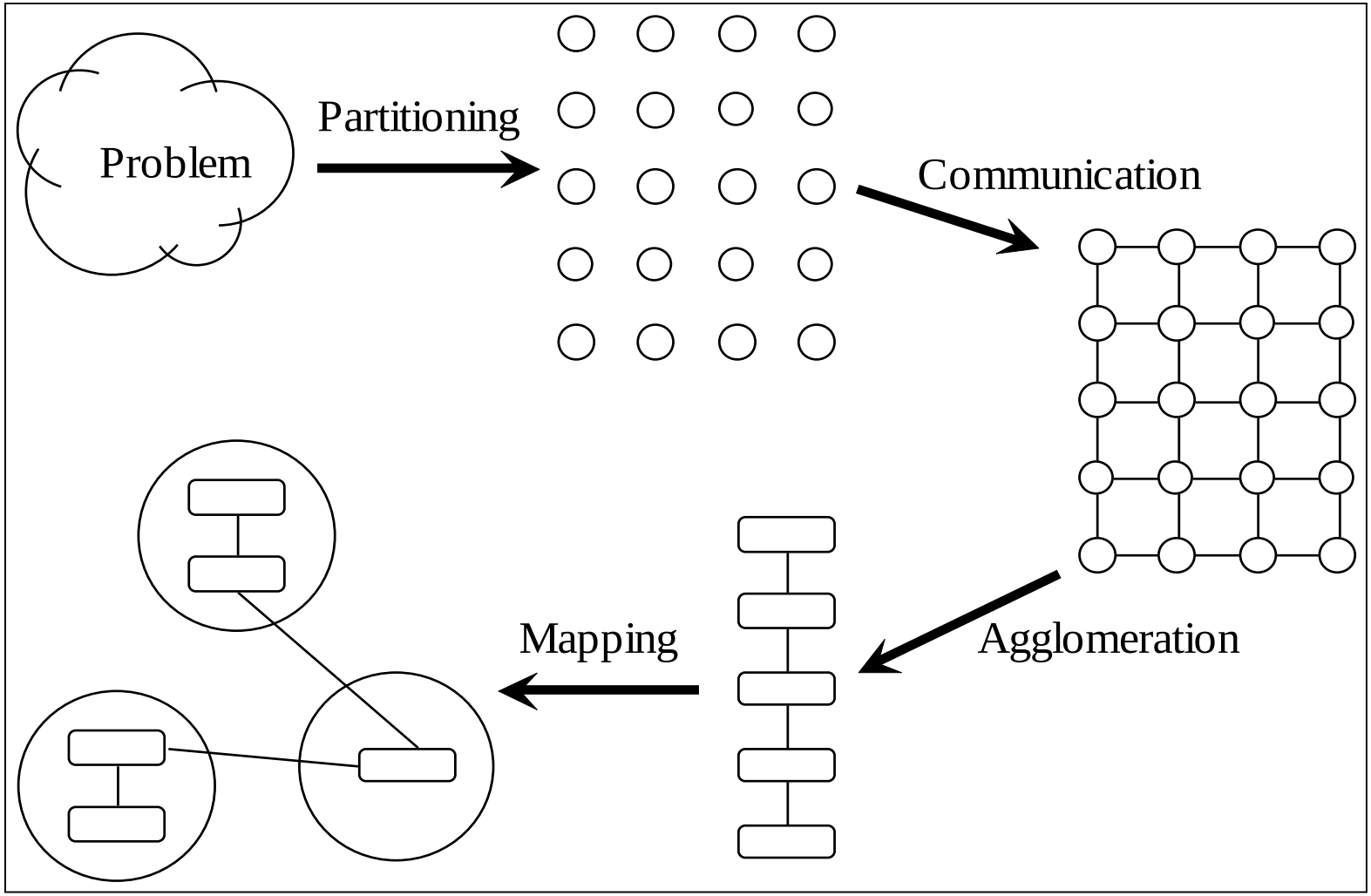


Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Foster's Design Methodology

- **Partitioning**
- **Communication**
- **Agglomeration**
- **Mapping**

Foster's Methodology



Partitioning

- Dividing computation and data into pieces
- Exploit data parallelism
  - (Data/domain partitioning/decomposition)
  - Divide data into pieces
  - Determine how to associate computations with the data
- Exploit task parallelism
  - (Task/functional partitioning/decomposition)
  - Divide computation into pieces
  - Determine how to associate data with the computations
- Exploit pipeline parallelism
  - (to optimize loops)

Domain Decomposition

- First, decide how data elements should be divided among processors
- Second, decide which tasks each processor should be doing

- Example: find maximum element in a vector

Source: CS133 Spring 2010 at UCLA (Kaplan)

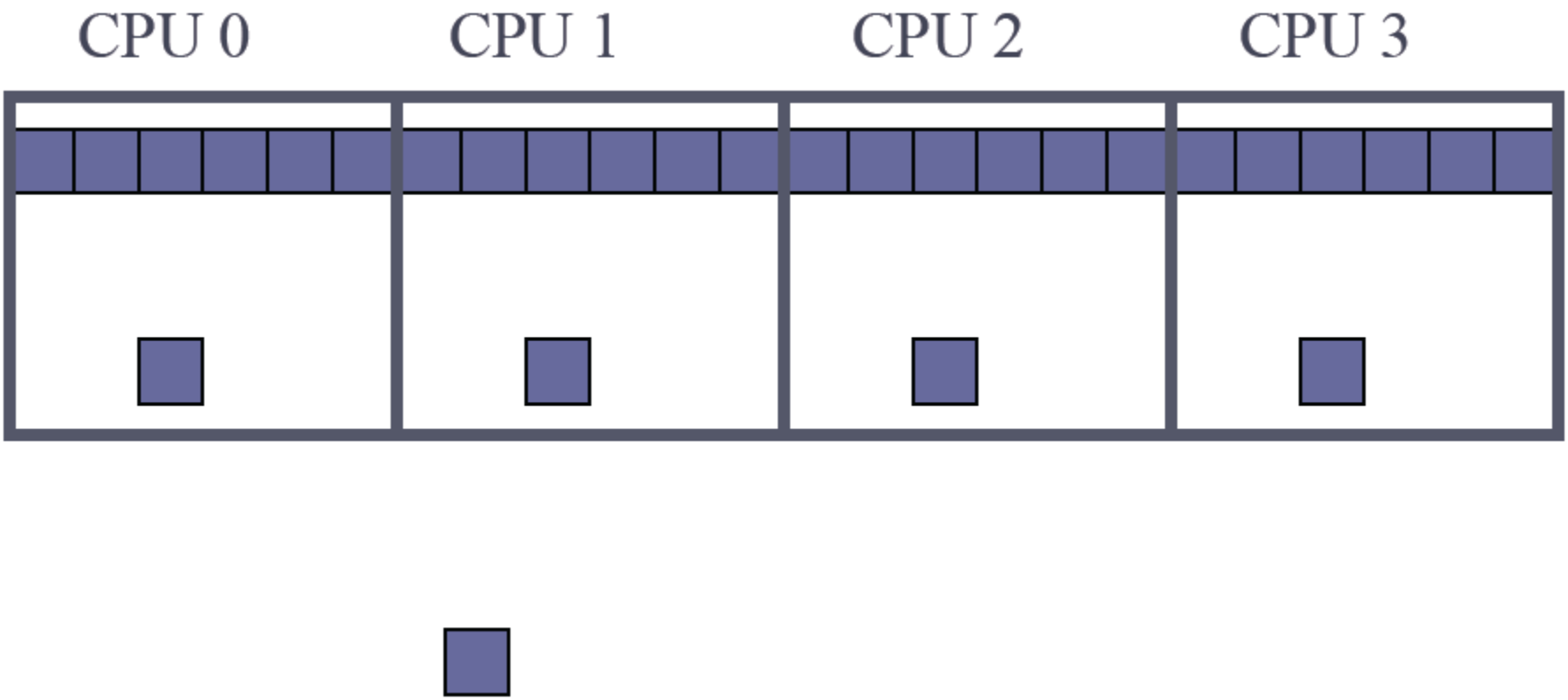
Domain Decomposition Example

Find the largest element of an array

Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Domain Decomposition Example

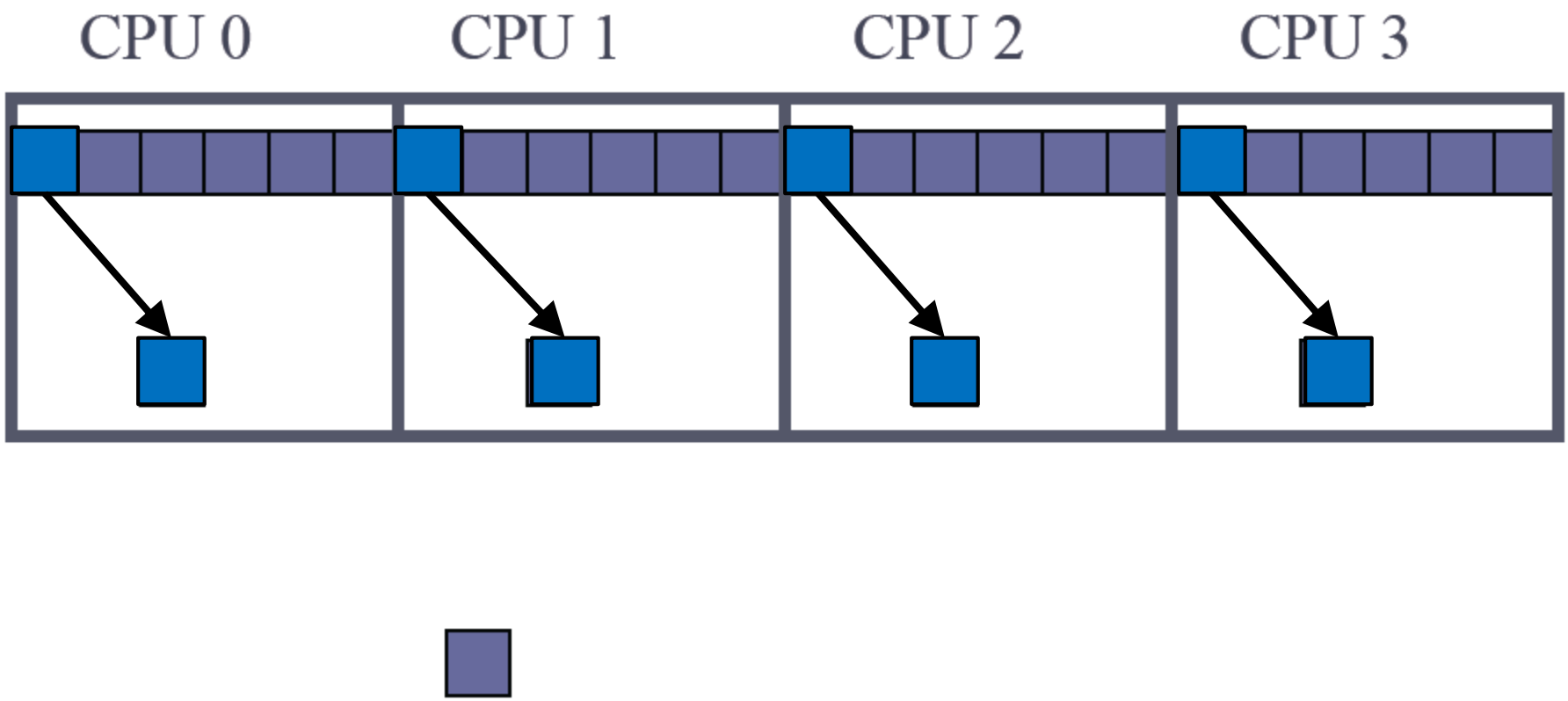
Find the largest element of an array



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Domain Decomposition Example

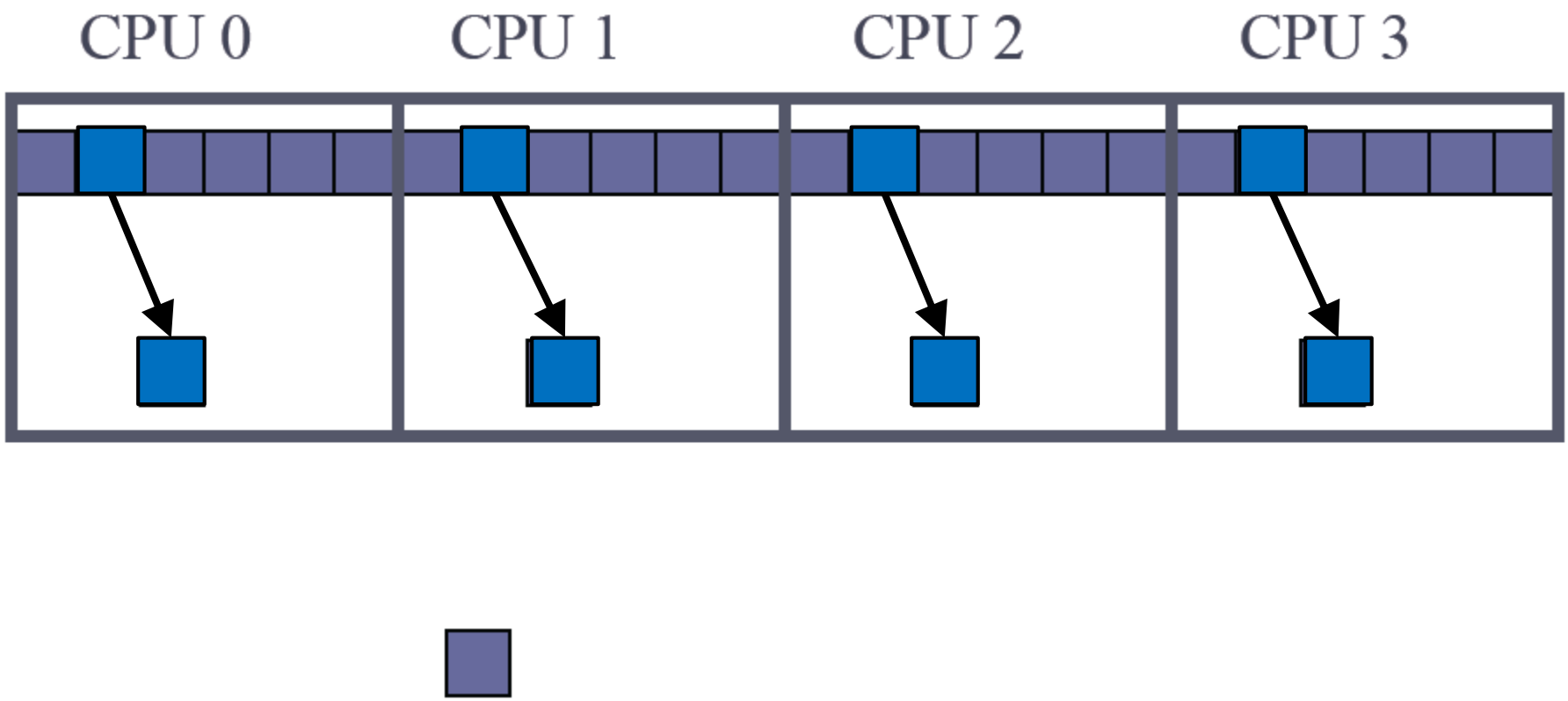
Find the largest element of an array



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Domain Decomposition Example

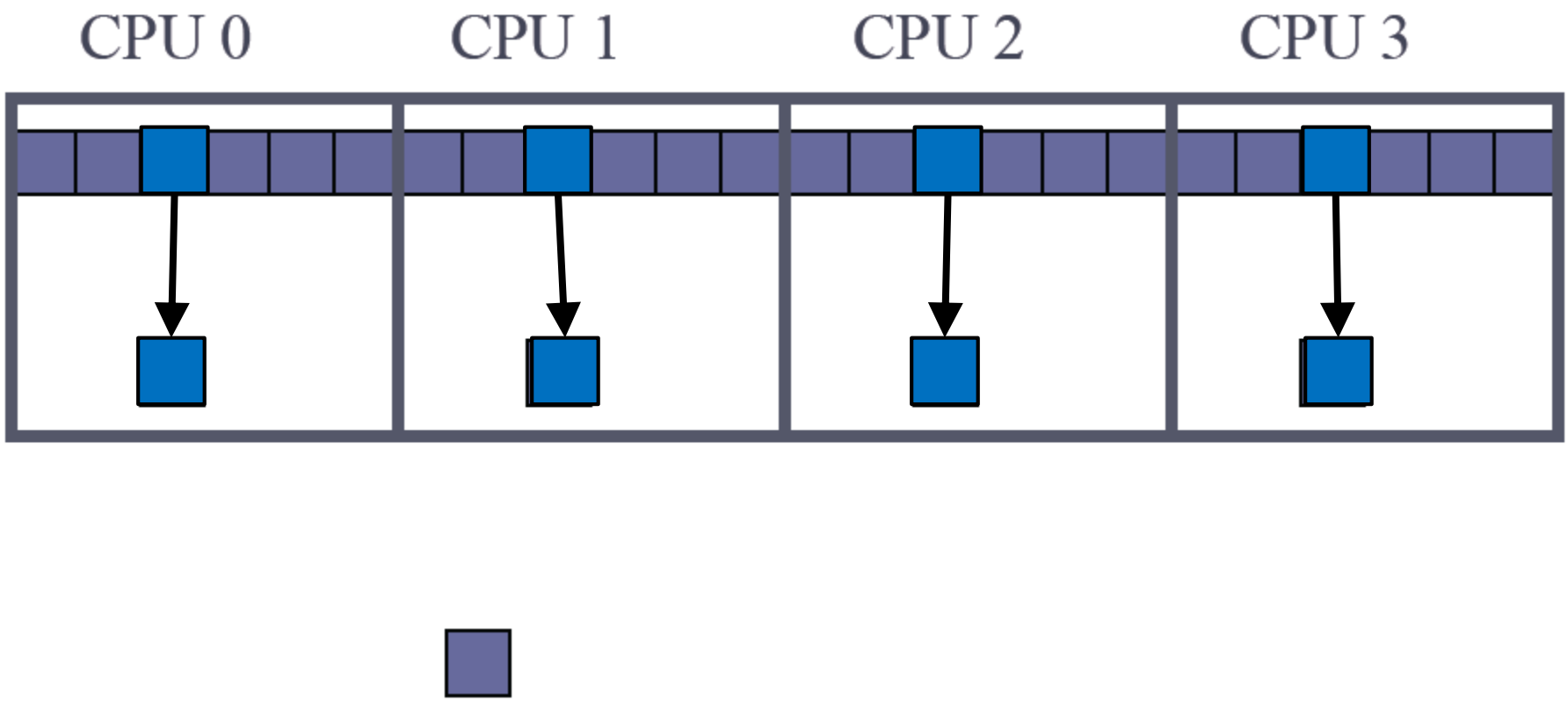
Find the largest element of an array



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Domain Decomposition Example

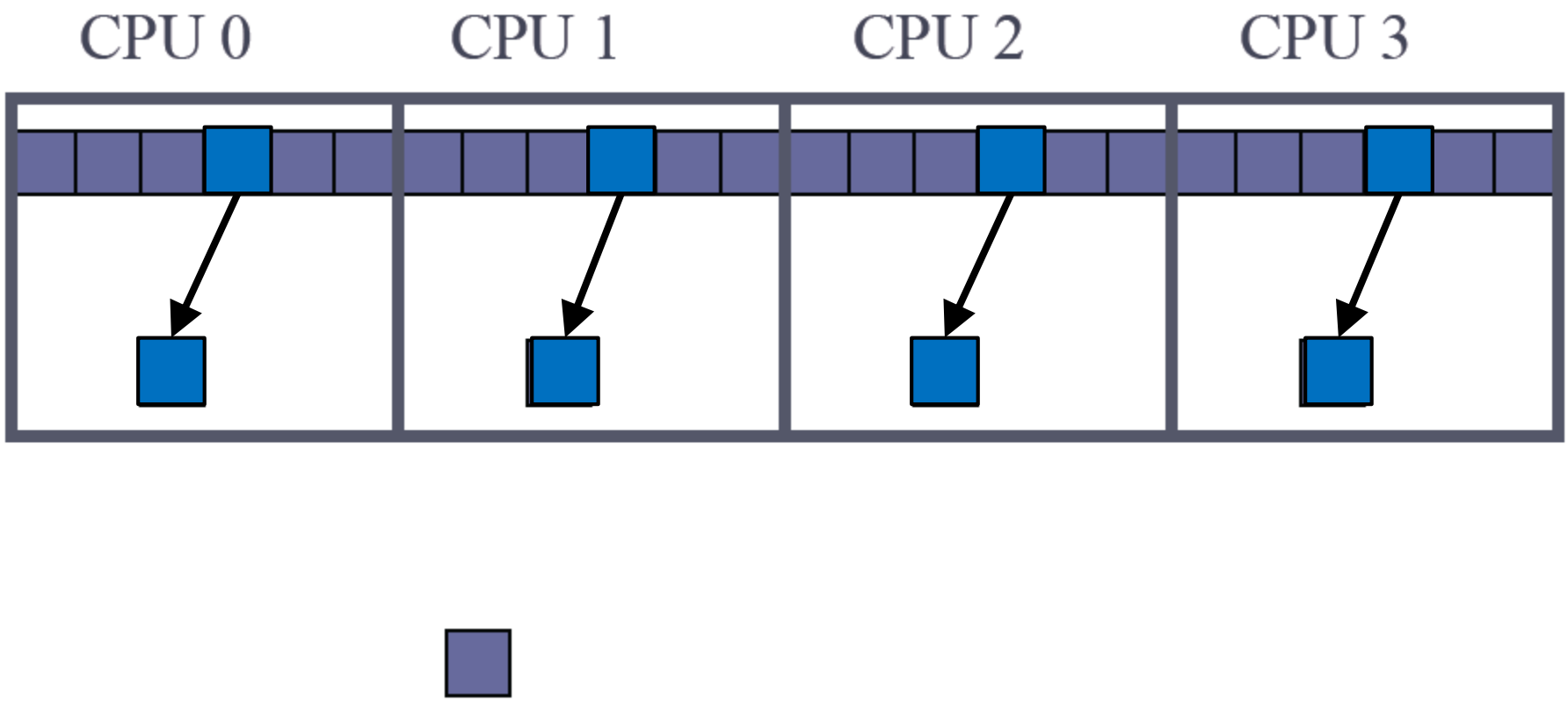
Find the largest element of an array



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Domain Decomposition Example

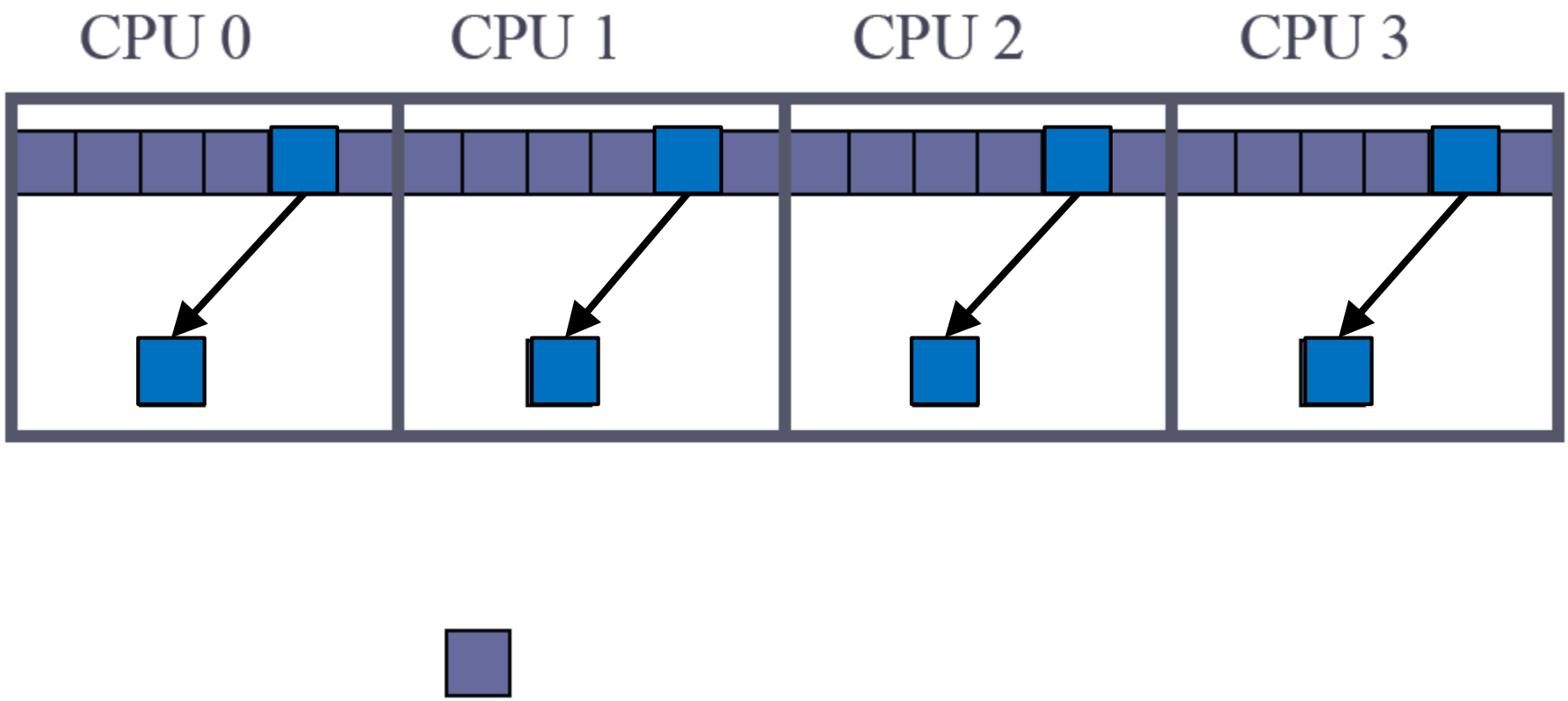
Find the largest element of an array



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Domain Decomposition Example

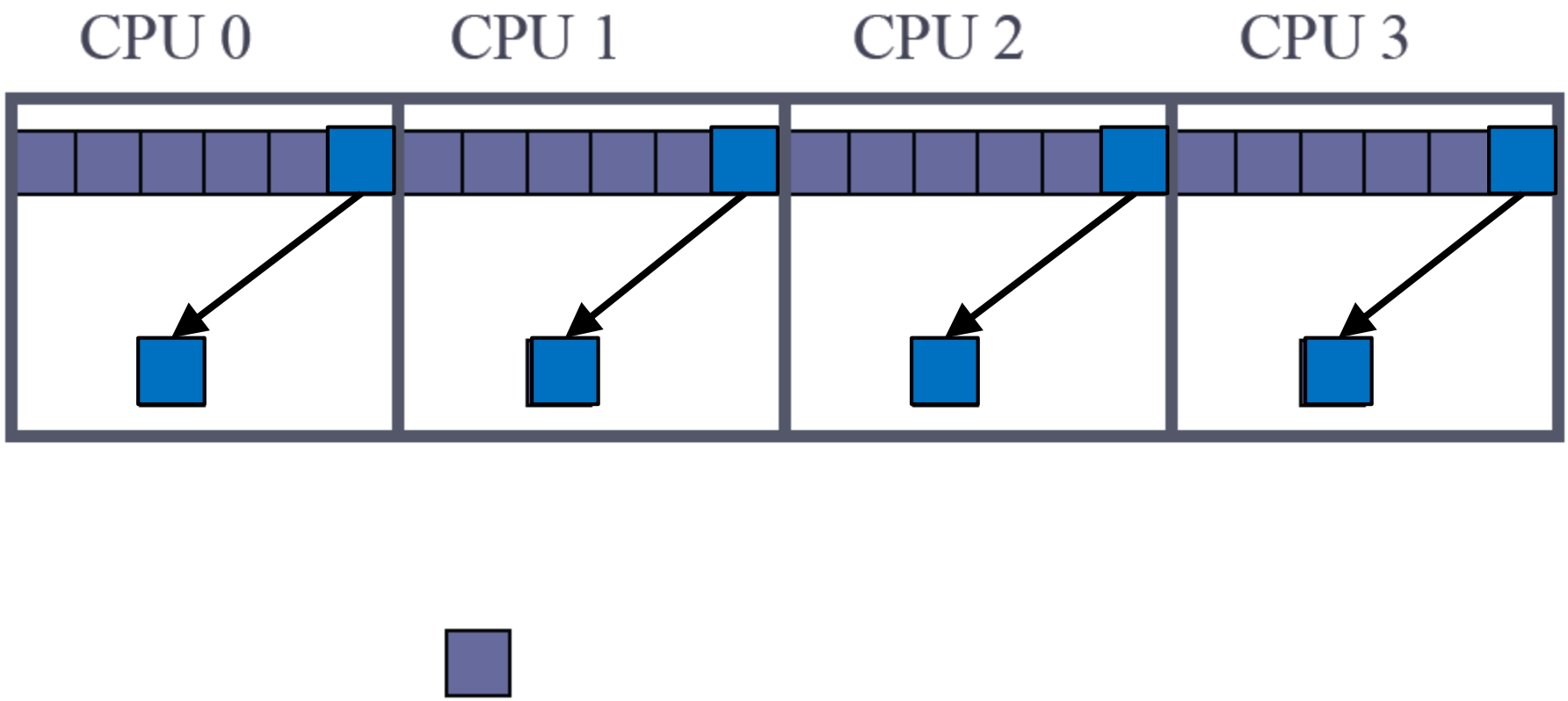
Find the largest element of an array



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Domain Decomposition Example

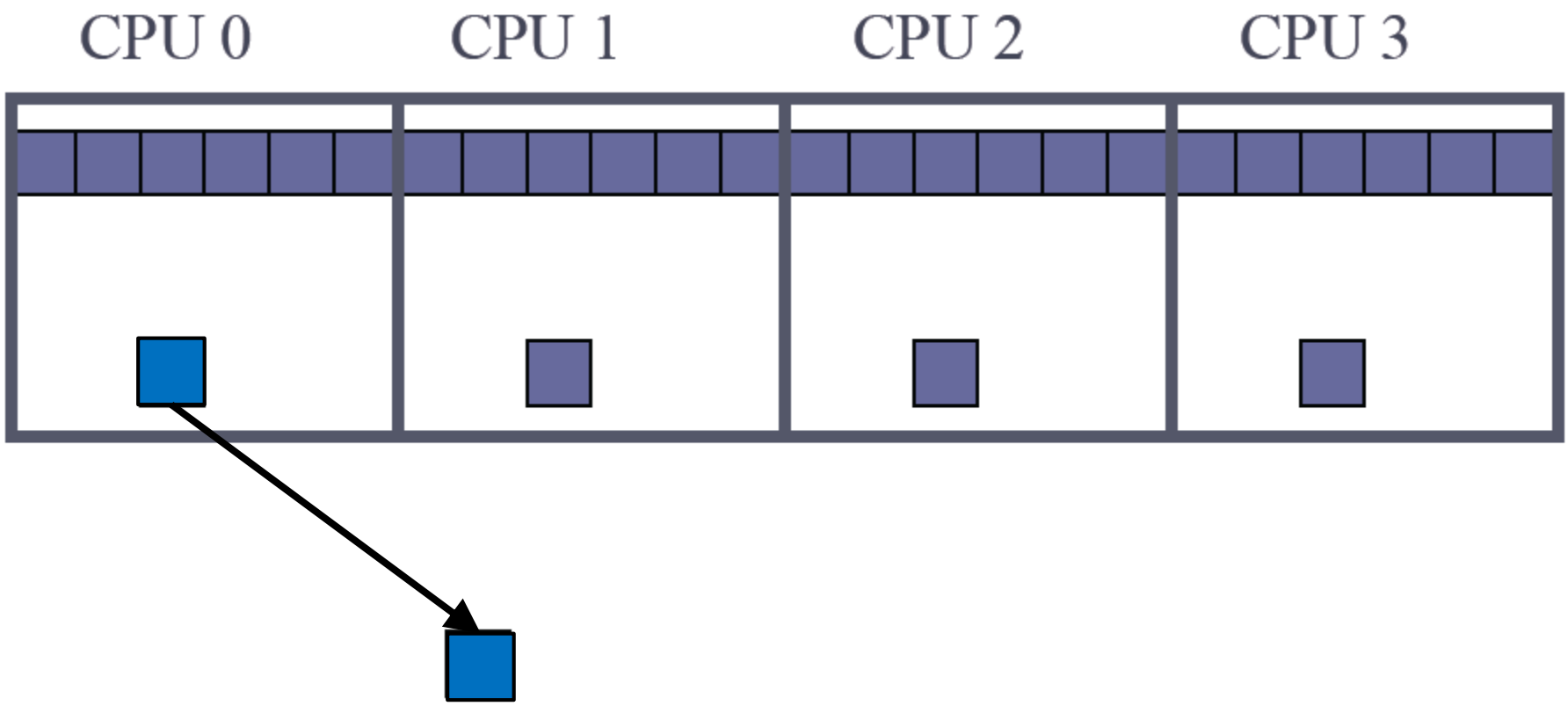
Find the largest element of an array



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Domain Decomposition Example

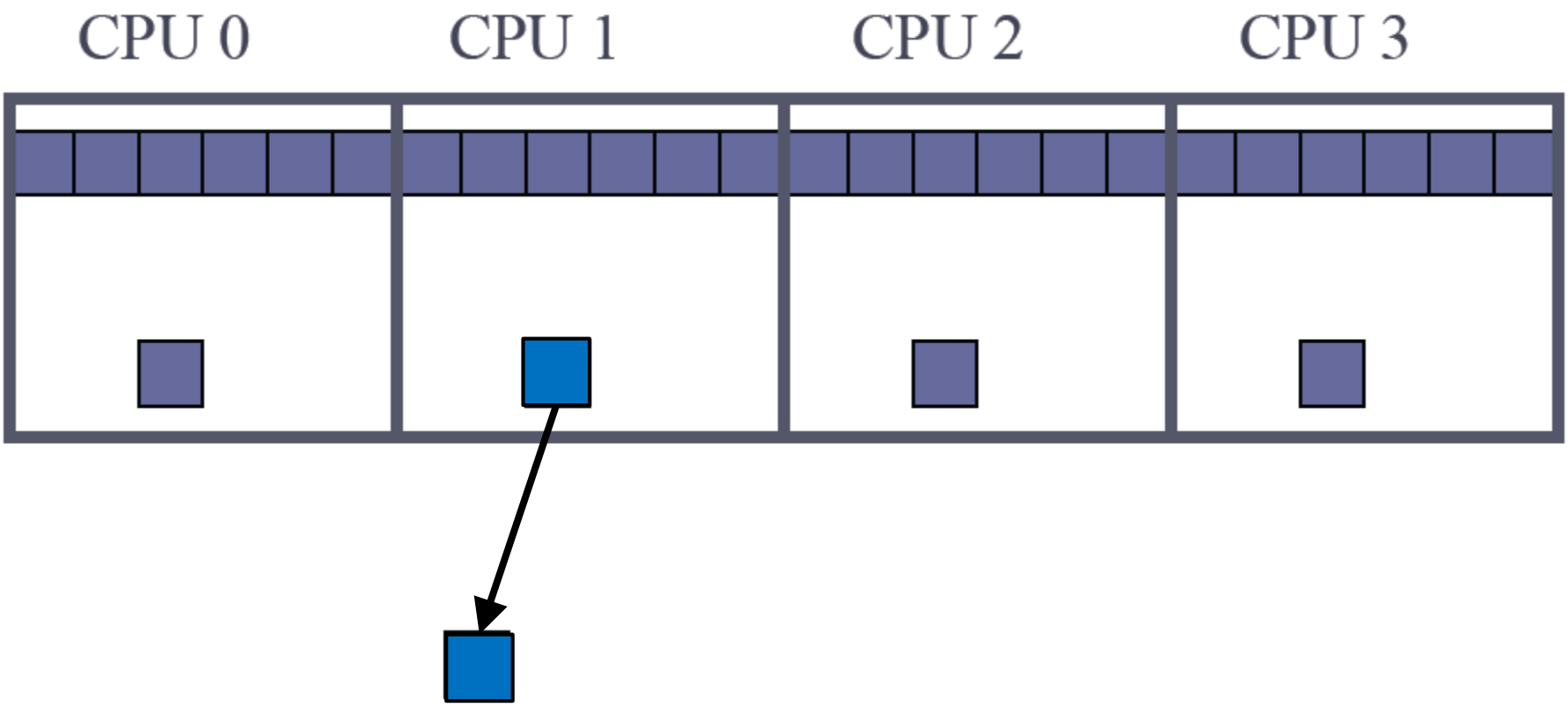
Find the largest element of an array



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Domain Decomposition Example

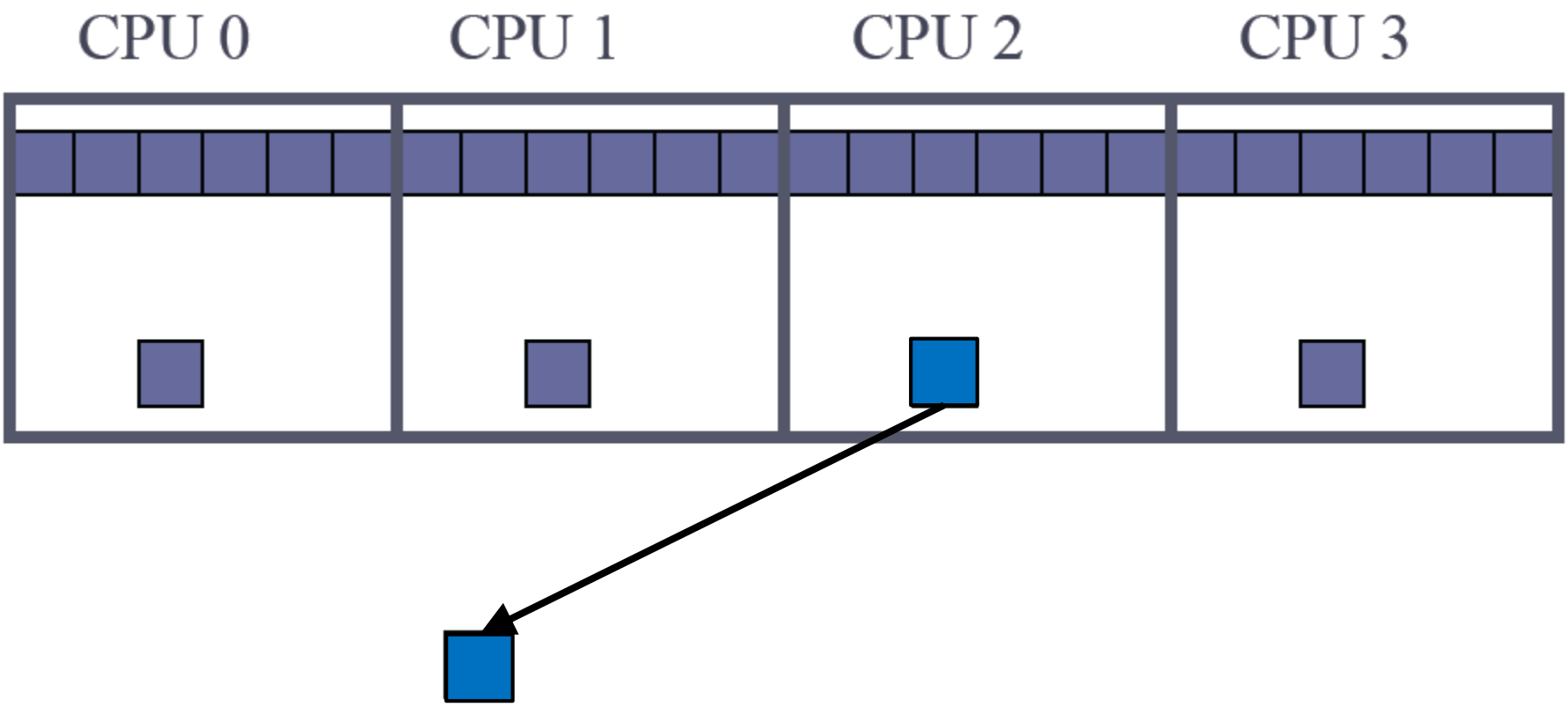
Find the largest element of an array



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Domain Decomposition Example

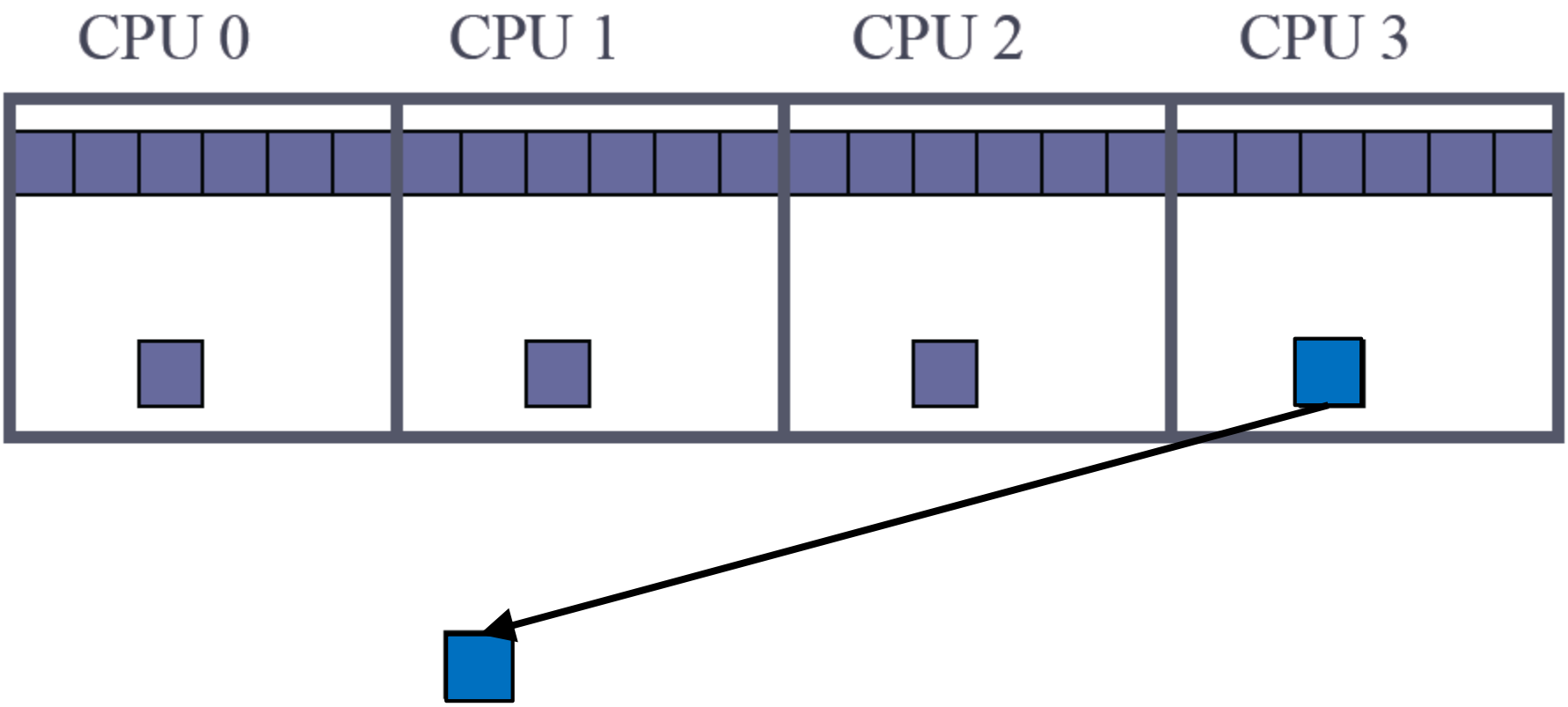
Find the largest element of an array



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

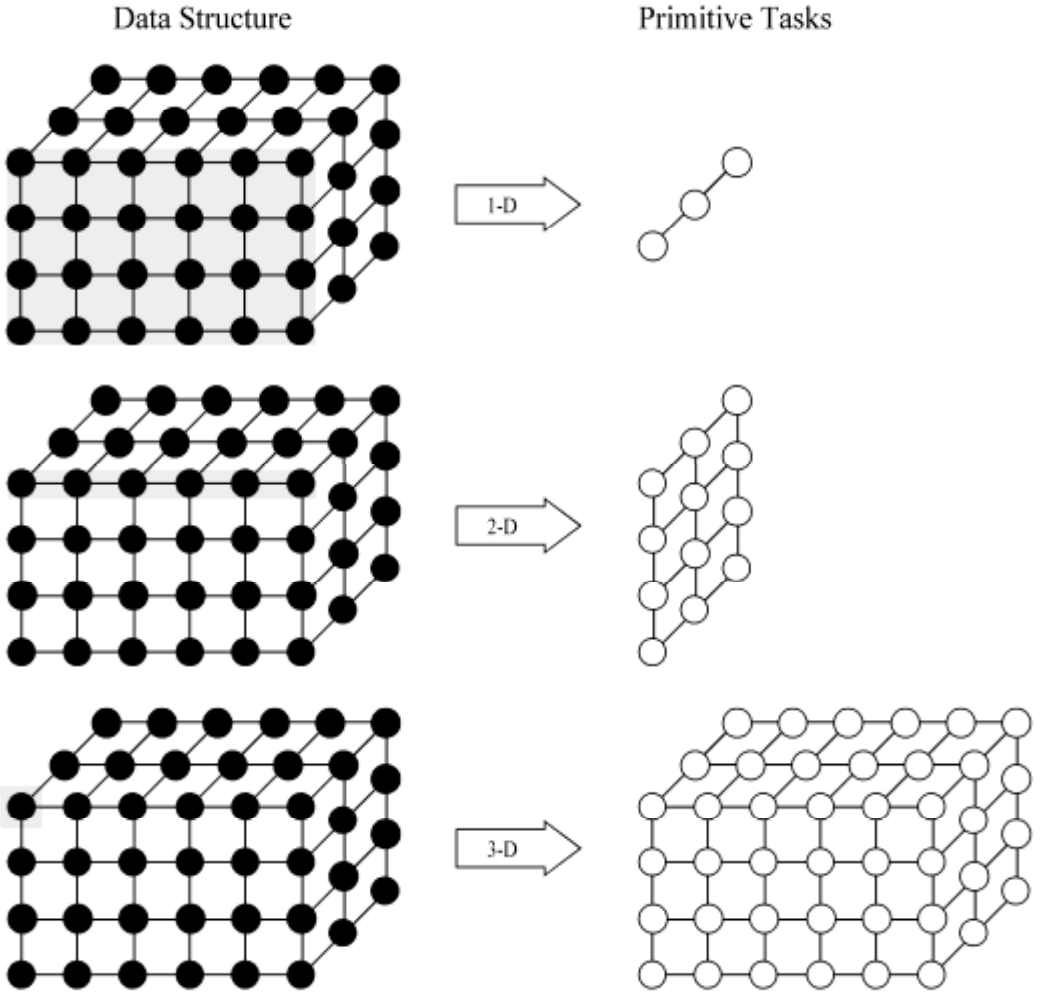
Domain Decomposition Example

Find the largest element of an array



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Domain Decomposition: Another Example

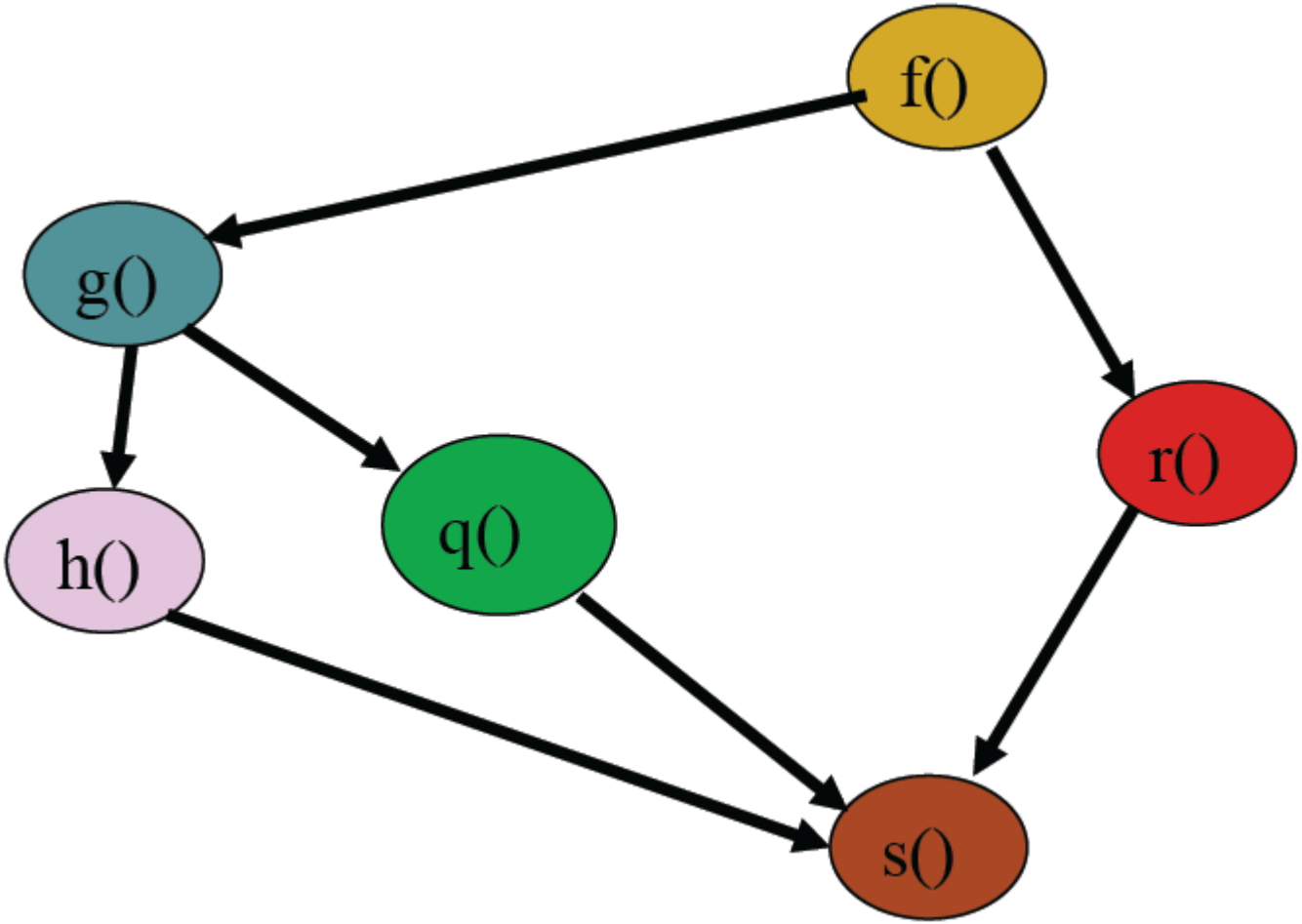


Functional (Task) Decomposition

- First, divide tasks among processors
- Second, decide which data elements are going to be accessed (read and/or written) by which processor
- Example: event-handler for GUI

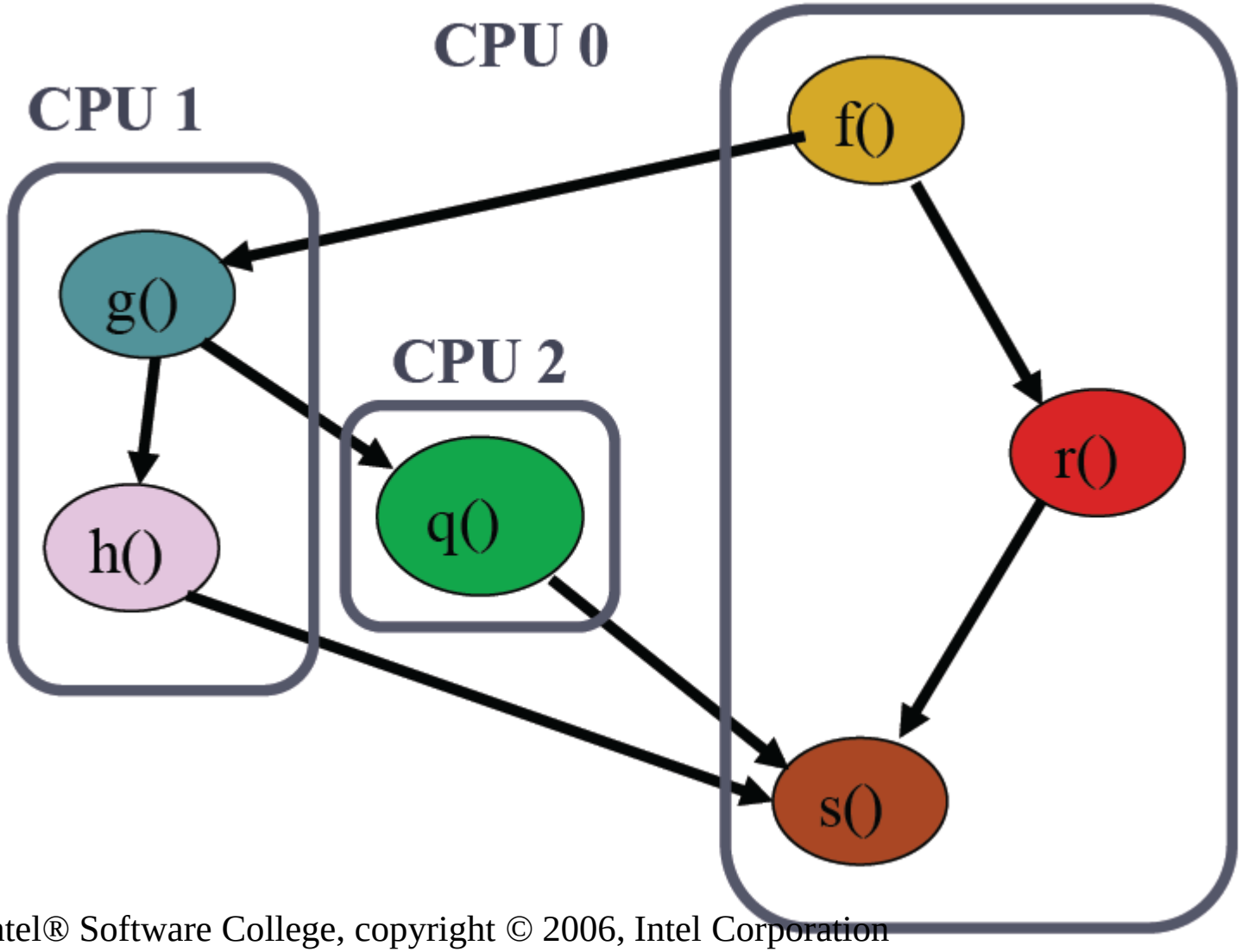
Source: CS133 Spring 2010 at UCLA (Kaplan)

Functional Decomposition Example



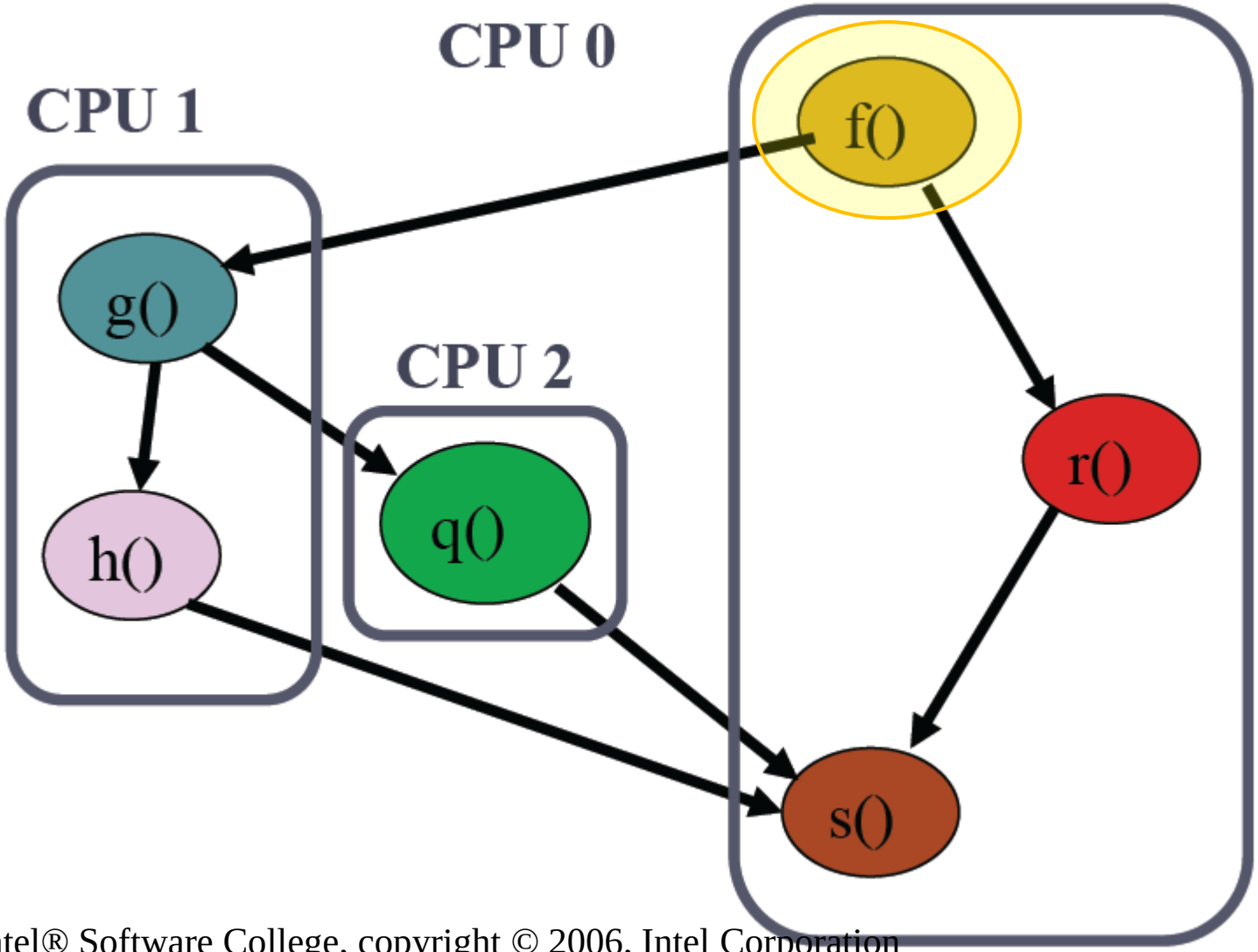
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Functional Decomposition Example



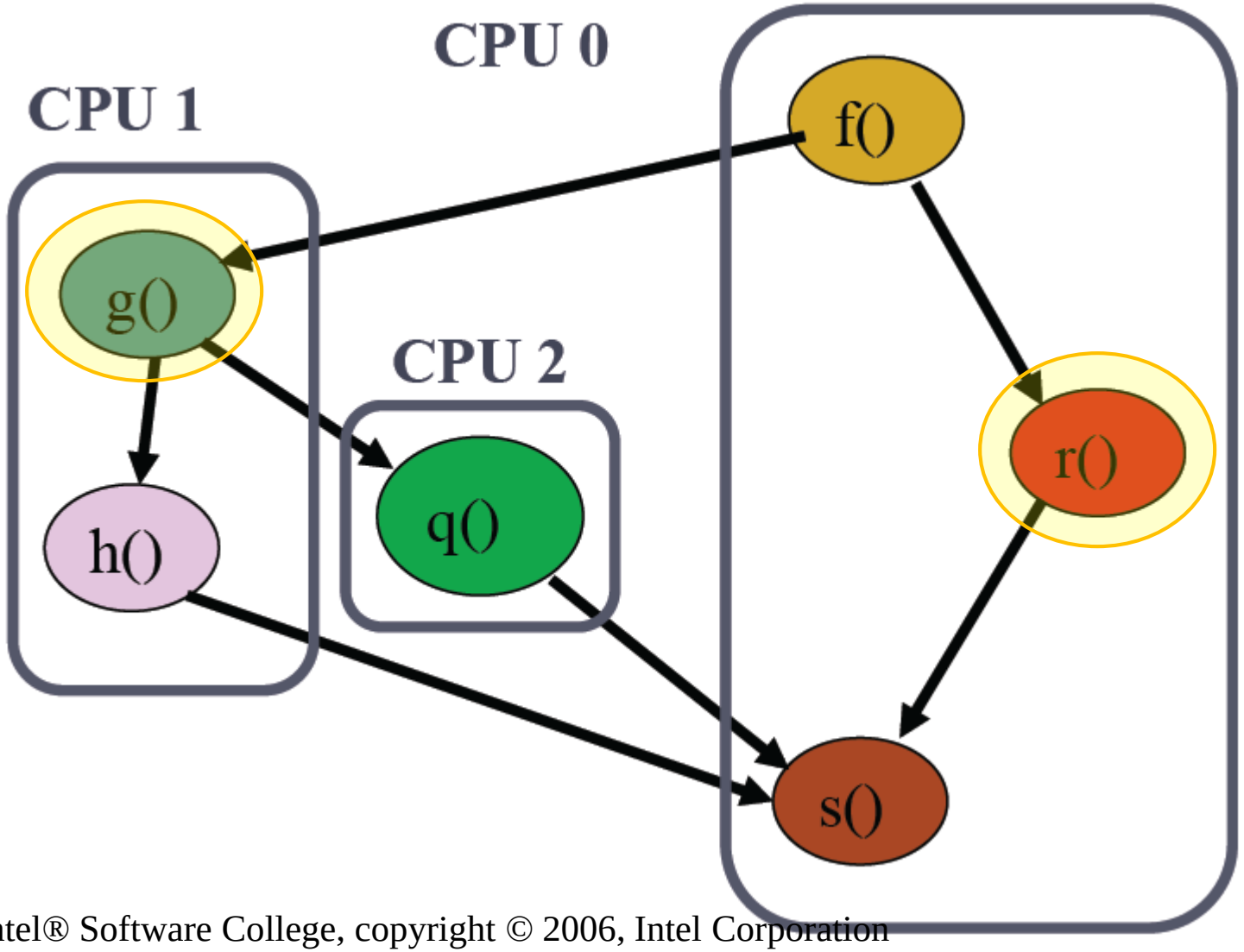
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Functional Decomposition Example



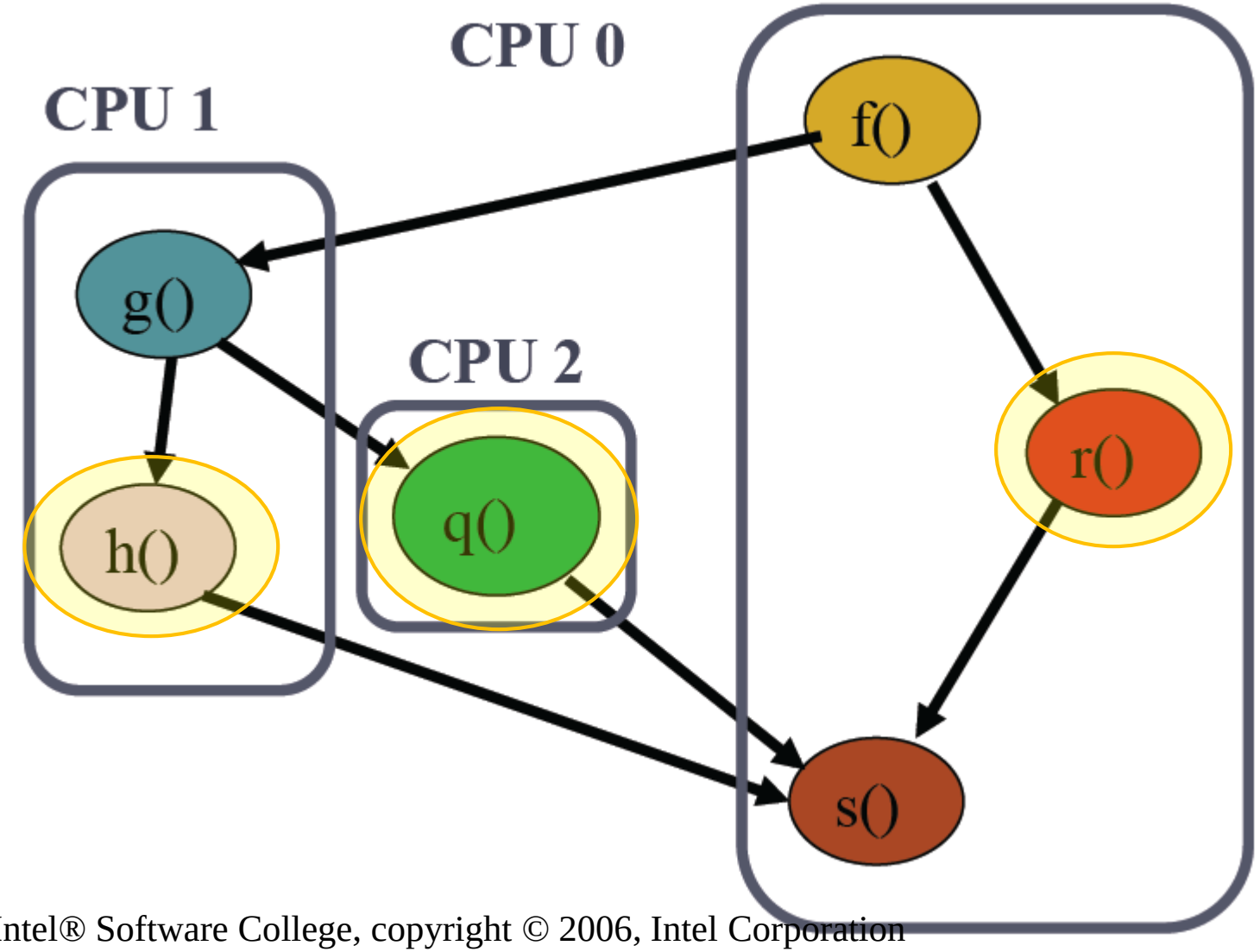
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Functional Decomposition Example



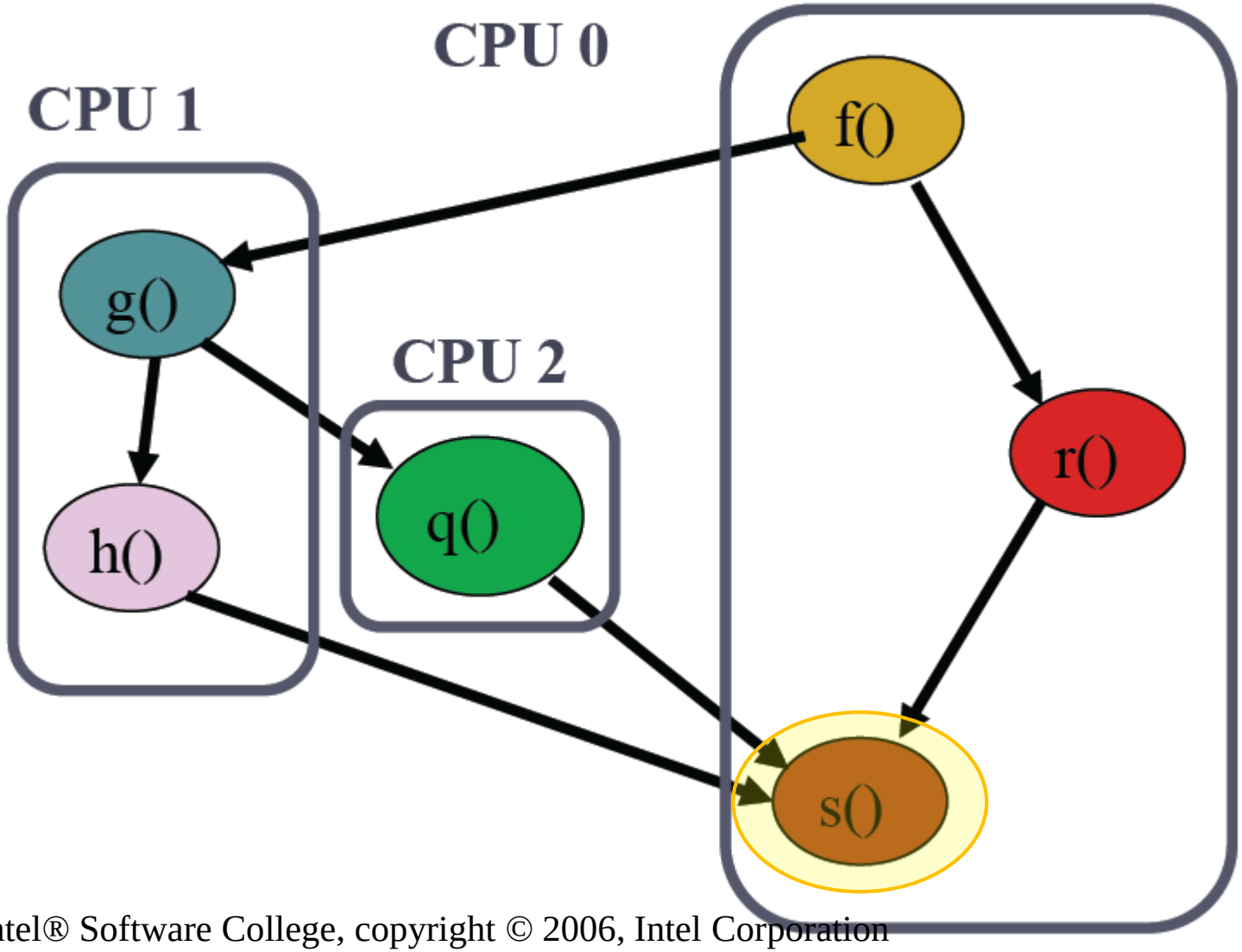
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Functional Decomposition Example



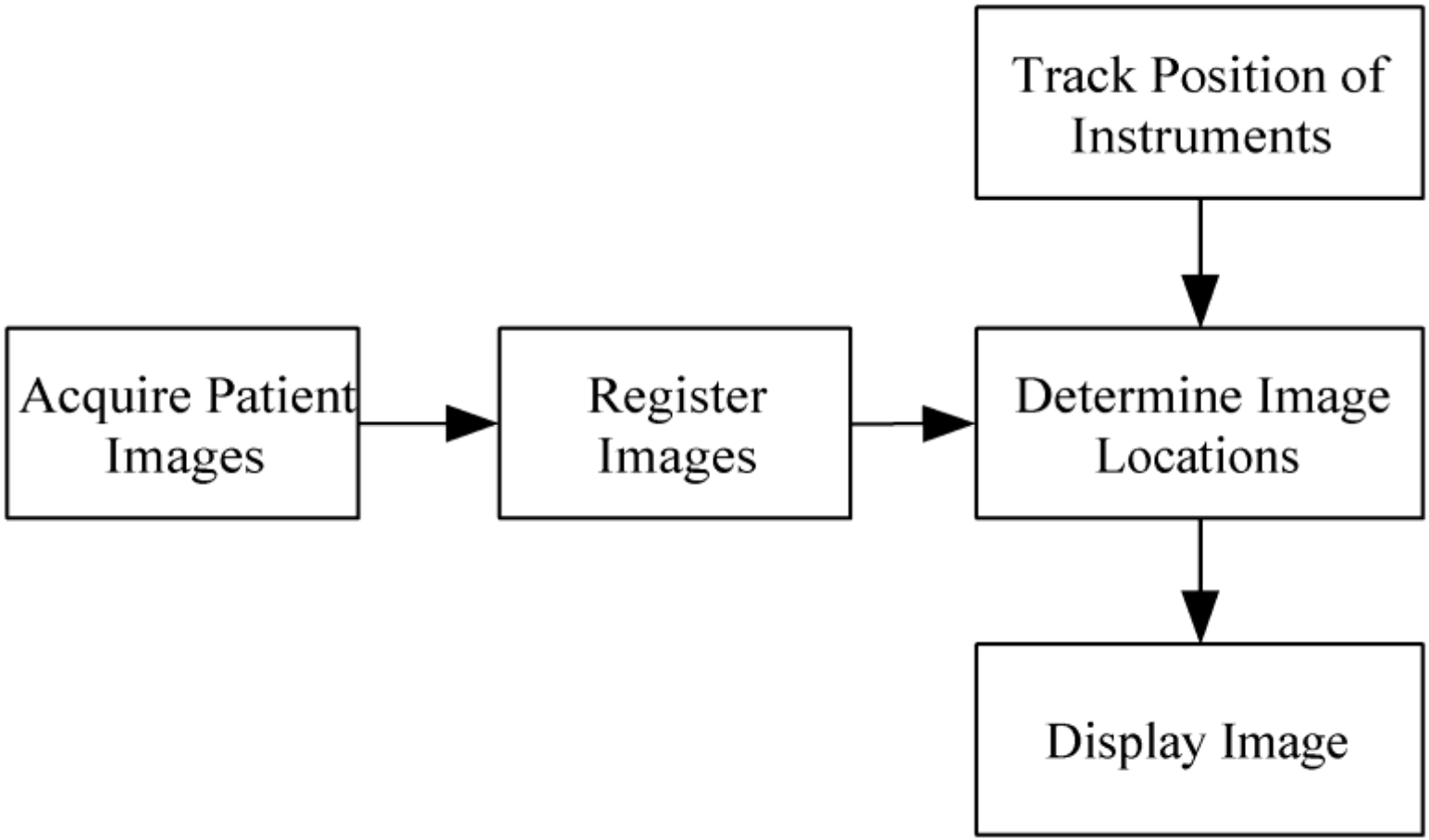
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Functional Decomposition Example

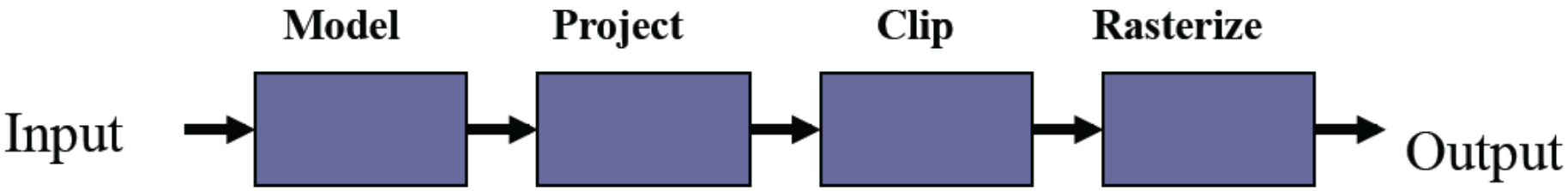


Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Functional Decomposition: Another Example



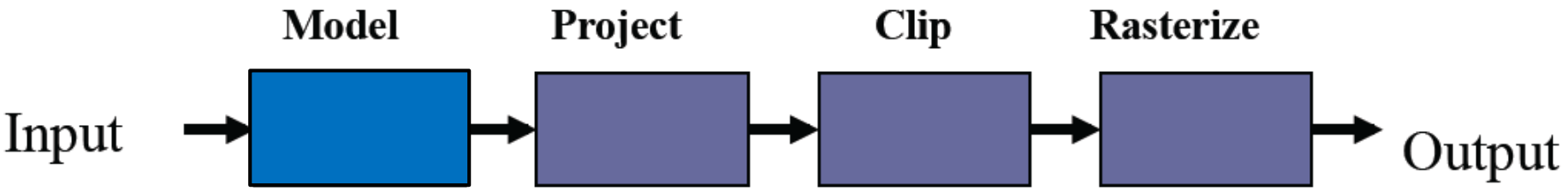
Pipelining



- **Special kind of task decomposition**
- **"Assembly line" parallelism**
- **Example: 3D rendering in computer graphics**

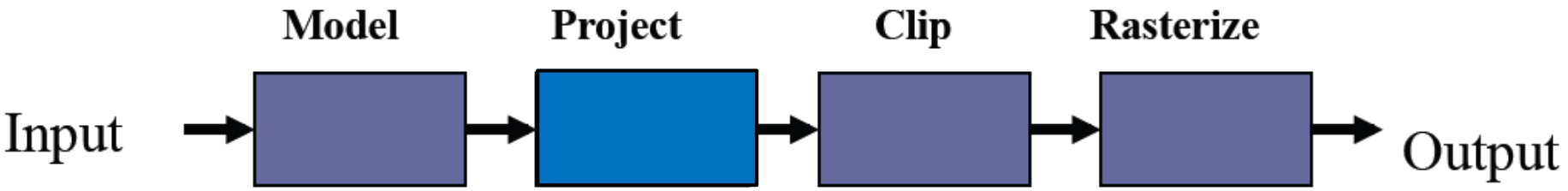
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Processing One Data Set (Step 1)



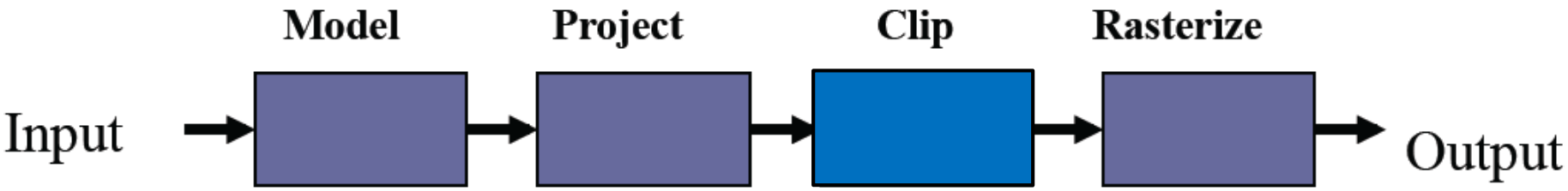
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Processing One Data Set (Step 2)



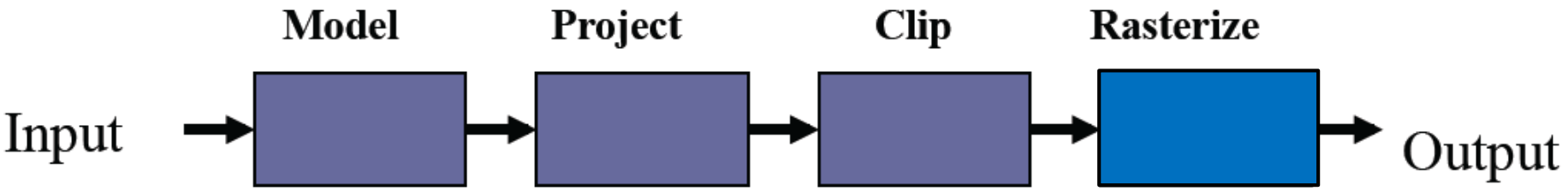
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Processing One Data Set (Step 3)



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

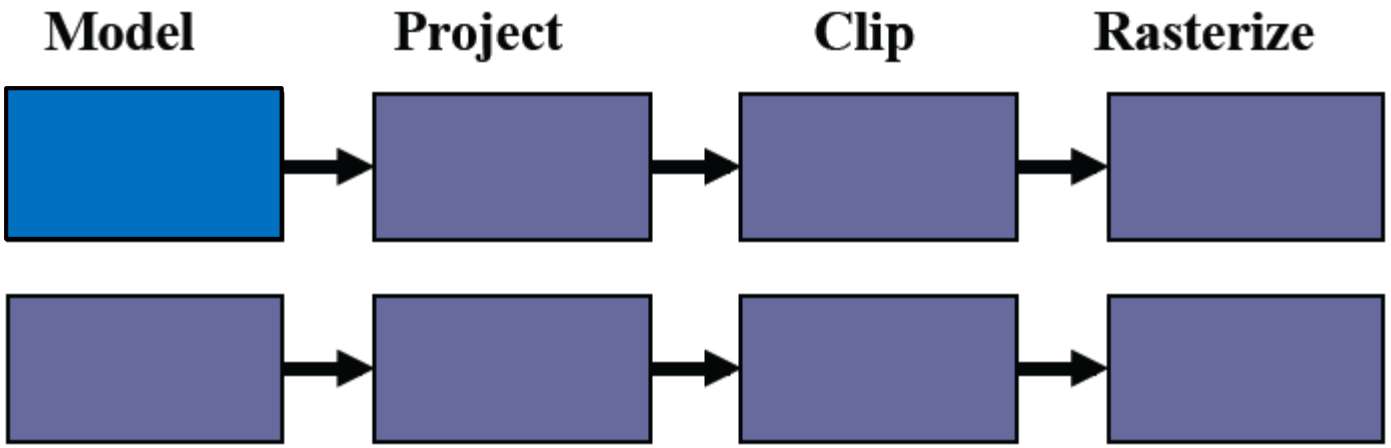
Processing One Data Set (Step 4)



The pipeline processes 1 data set in 4 steps

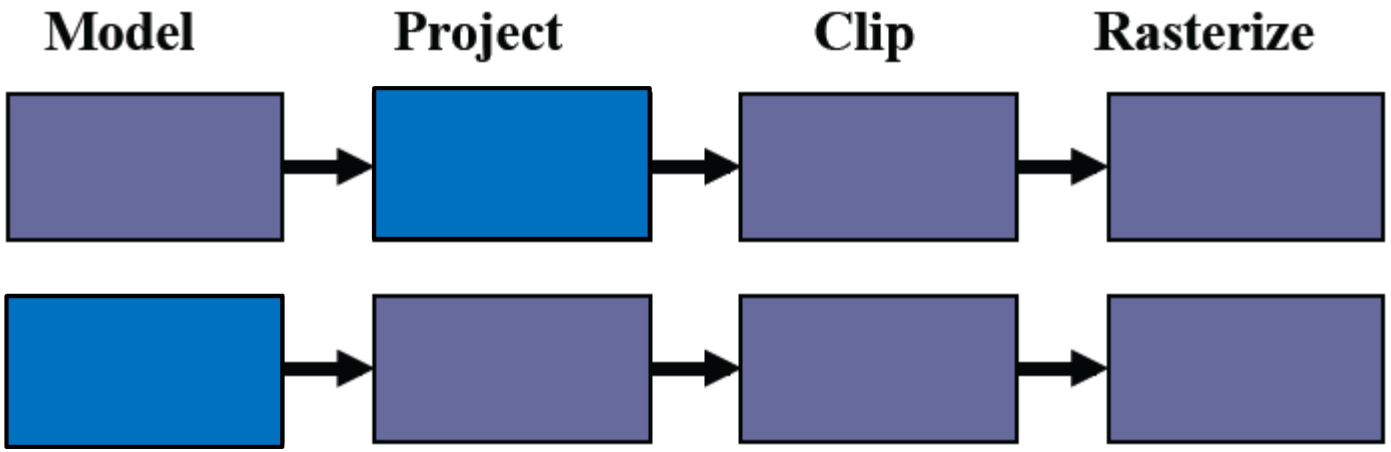
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Processing Two Data Sets (Step 1)



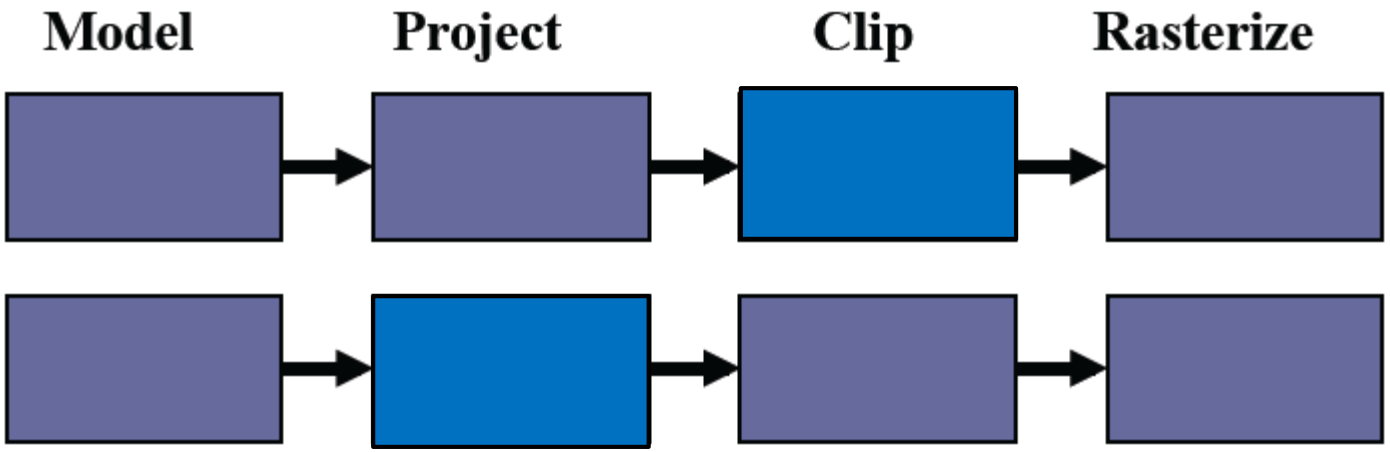
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Processing Two Data Sets (Step 2)



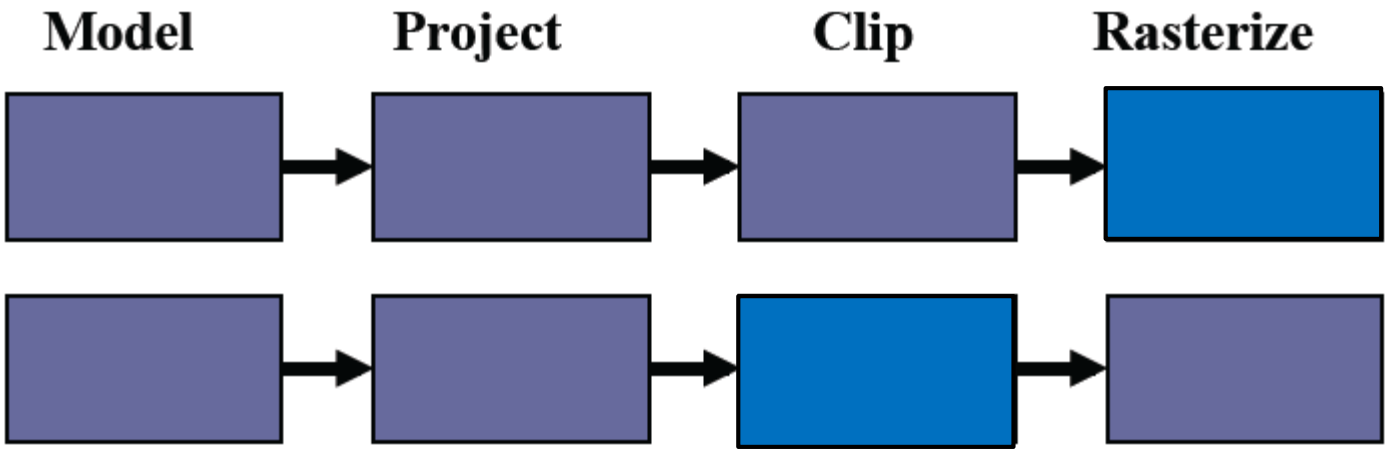
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Processing Two Data Sets (Step 3)



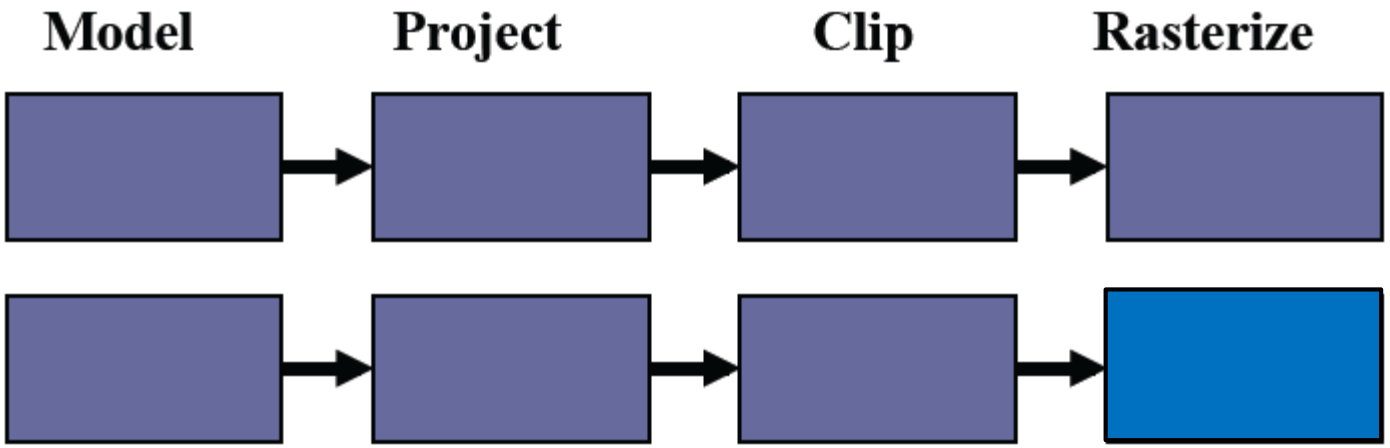
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Processing Two Data Sets (Step 4)



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

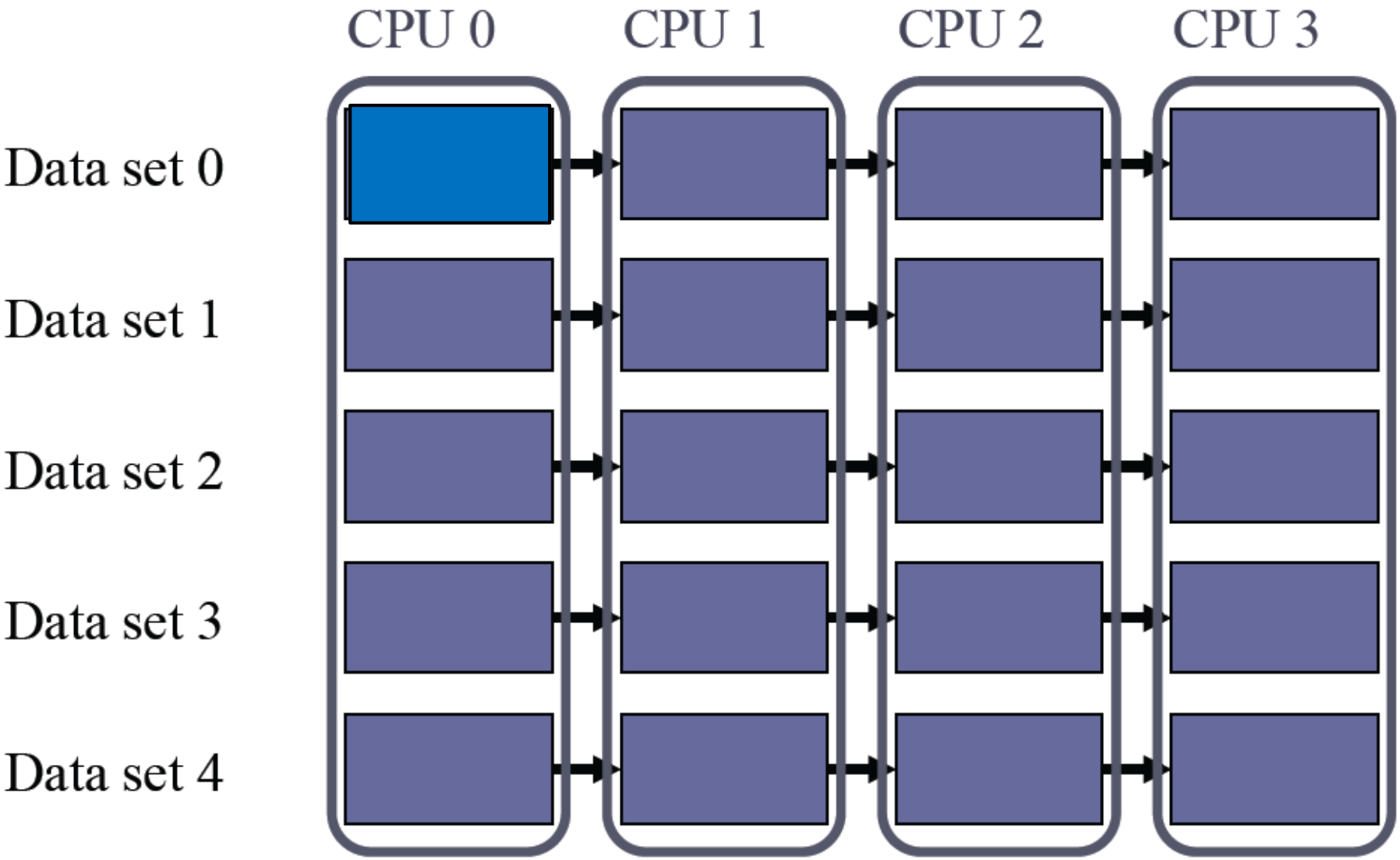
Processing Two Data Sets (Step 5)



The pipeline processes 2 data sets in 5 steps

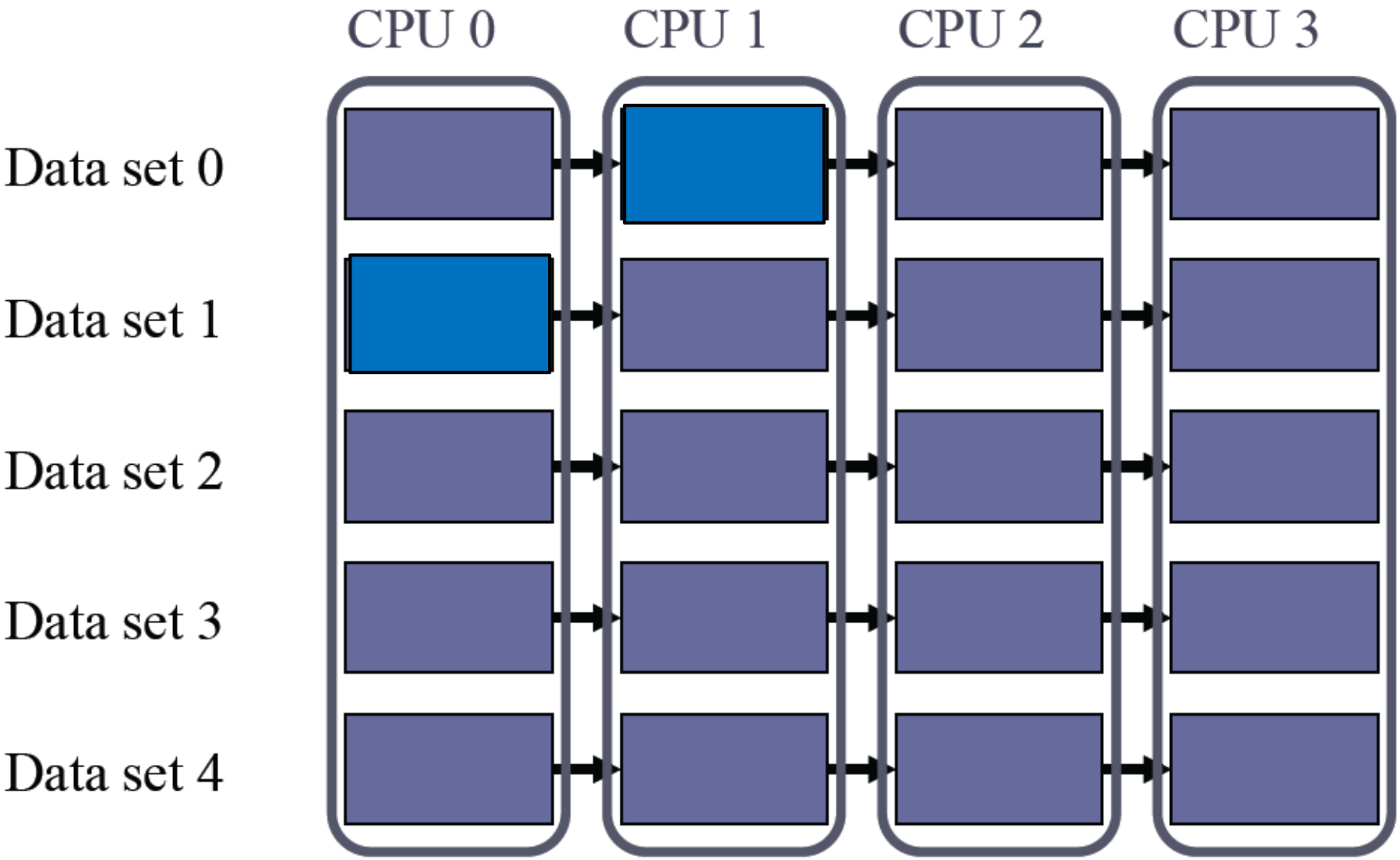
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Pipelining Five Data Sets (Step 1)



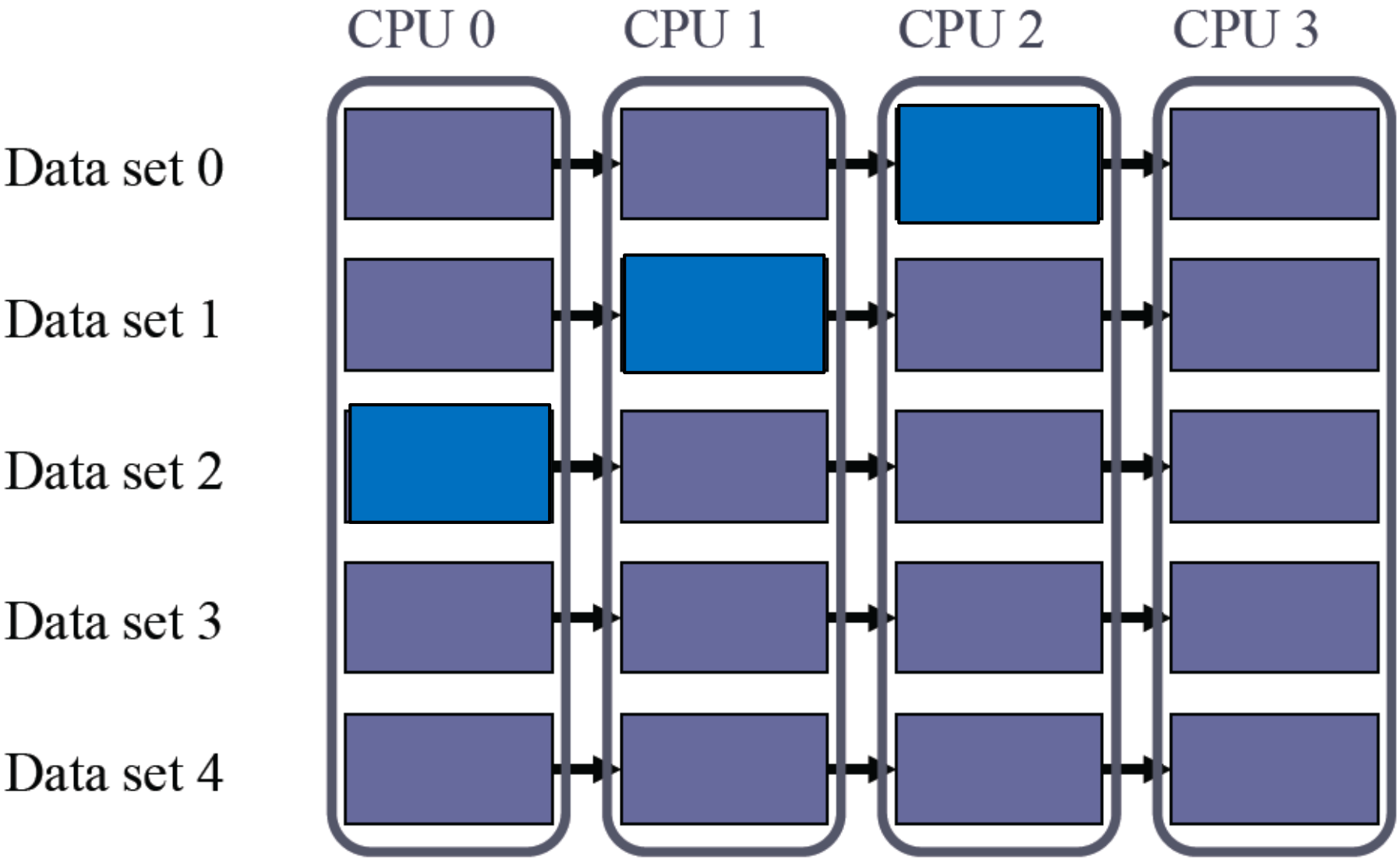
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Pipelining Five Data Sets (Step 2)



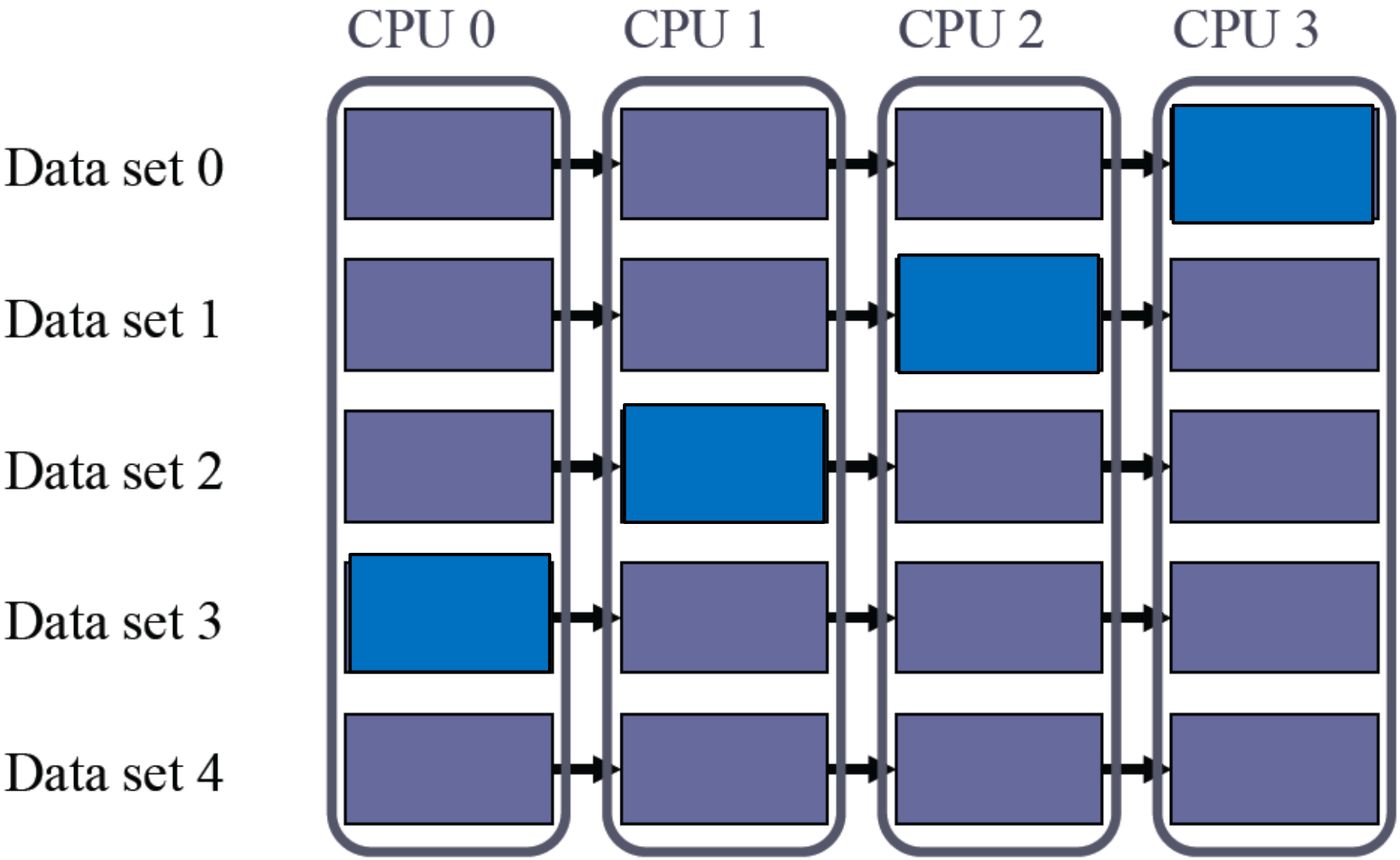
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Pipelining Five Data Sets (Step 3)



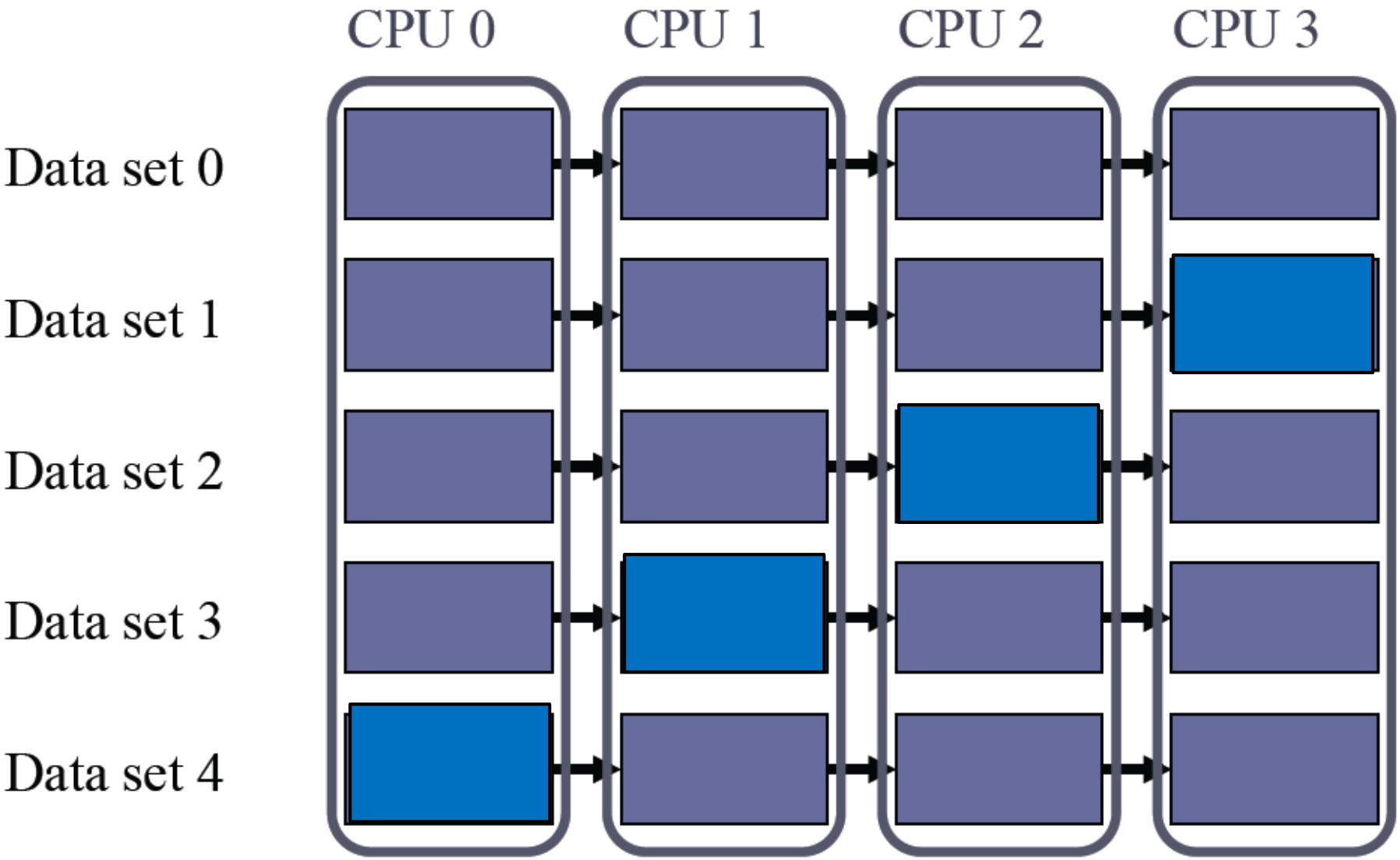
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Pipelining Five Data Sets (Step 4)



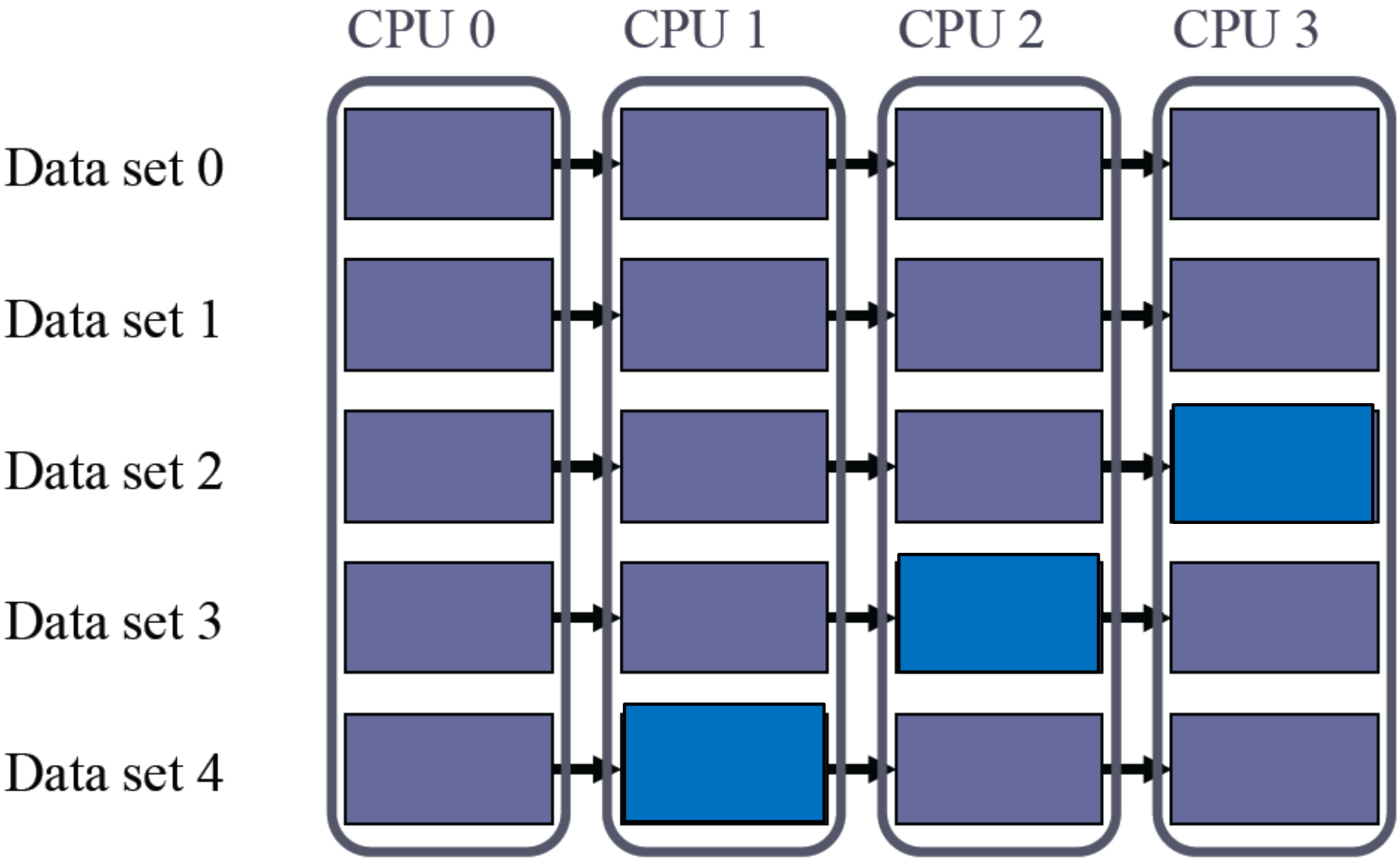
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Pipelining Five Data Sets (Step 5)



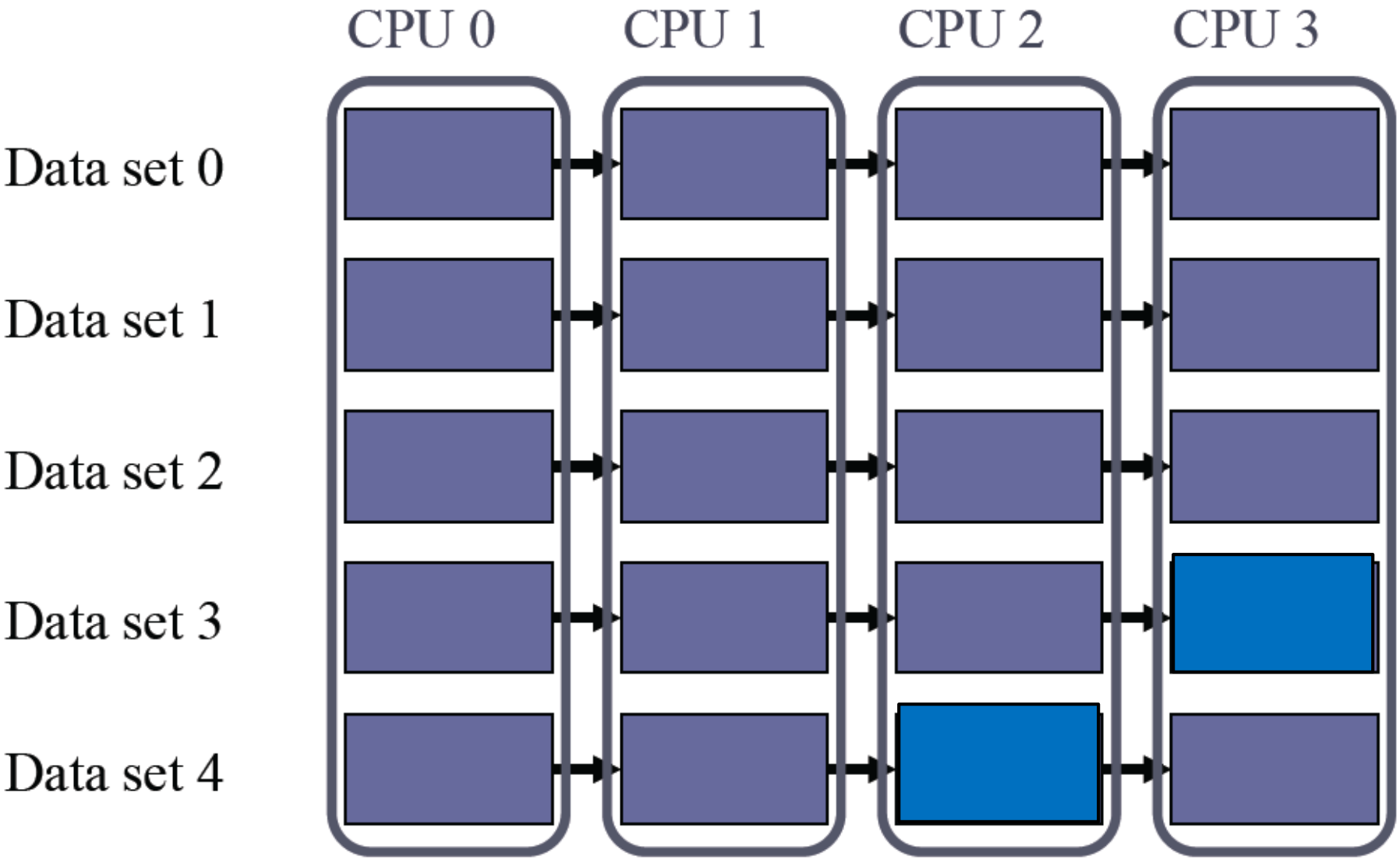
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Pipelining Five Data Sets (Step 6)



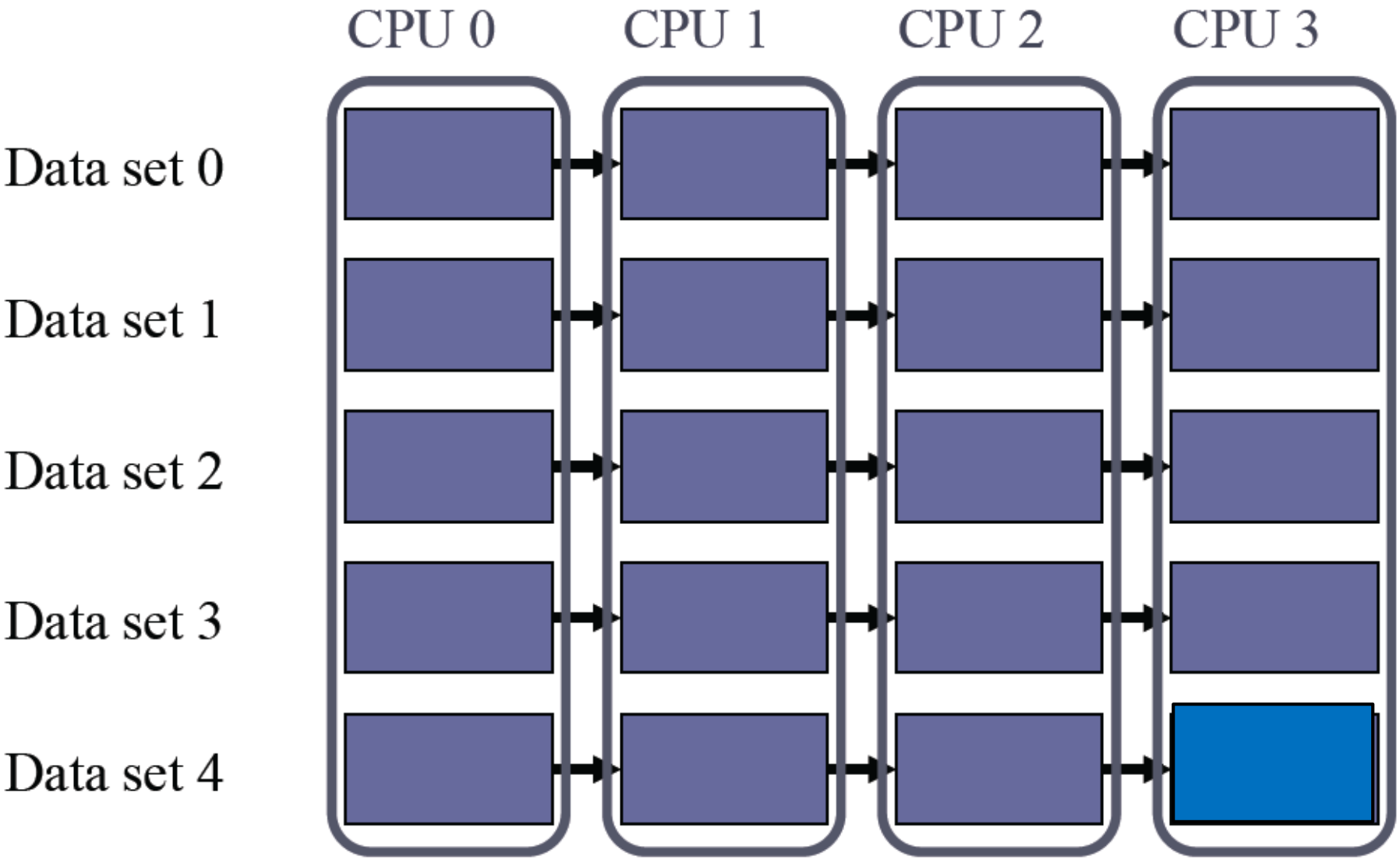
Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Pipelining Five Data Sets (Step 7)



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Pipelining Five Data Sets (Step 8)



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

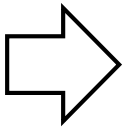
Speedup from Pipelining

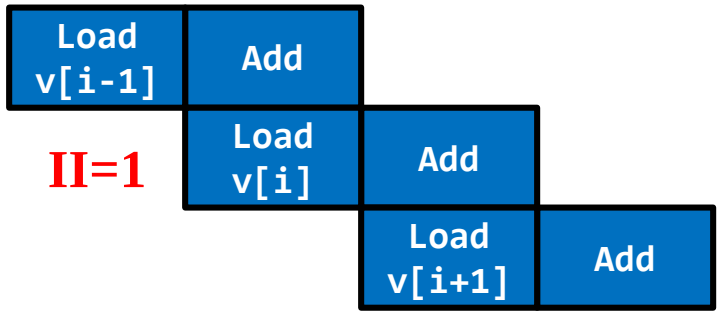
- In the previous examples,
  - Process 1 data set in 4 steps
  - Process 2 data sets in 5 steps
  - Process 5 data sets in 8 steps
  - Process N data sets in ?? steps
- Pipelining improves throughput, but not latency

Throughput and Initiation Interval

- **Initiation Interval (II)**
  - *Number of cycles that must elapse between two loop iterations*
- **Throughput**
  - *Amount of work done over a period of time (often per unit time)*
  - *II=1 ➔ Throughput is maximized*

```
for (i=0; i<SIZE; i++) {
    r += v[i];
}
```





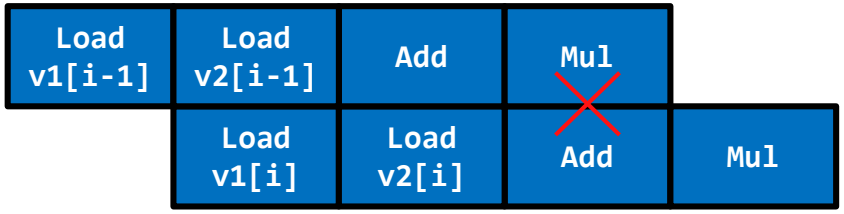
Limitation of Initiation Interval

- **II=1 is not always possible**
  - *Resource conflict*
  - *Data dependency*

Assume one ALU

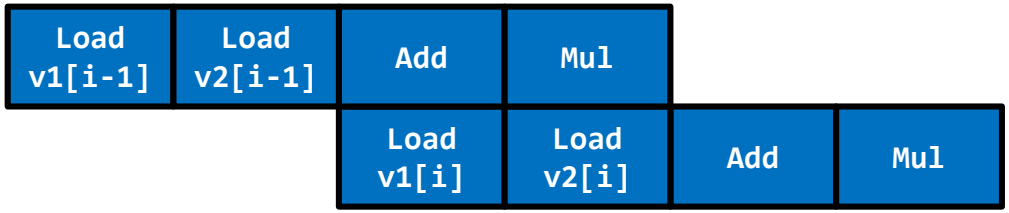
```
for (i=0; i<SIZE; i++) {
    r += v1[i];
    r *= v2[i];
}
```

II=1



Not possible

II=2

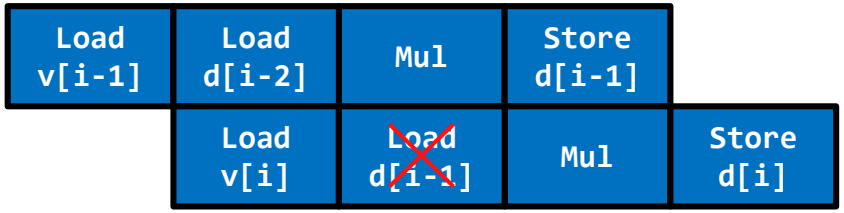


Limitation of Initiation Interval

- **II>1 because of Data dependency**

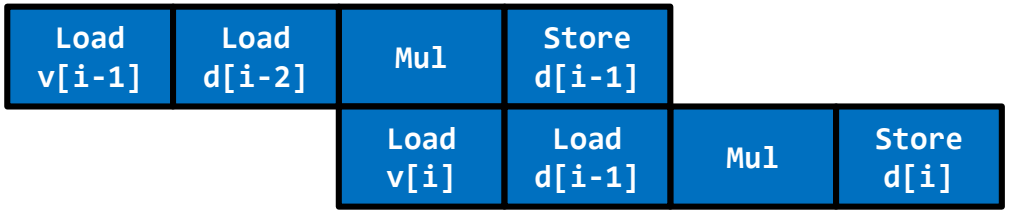
```
for (i=0; i<SIZE; i++) {
    d[i] = d[i-1]*v[i];
}
```

II=1



Not possible

II=2



Partitioning Checklist

- At least 10x more primitive tasks than processors in target computer
  - If not, later design options may be too constrained
- Minimize redundant computations and redundant data storage
  - If not, the design may not work well when the size of the problem increases
- Primitive tasks roughly the same size
  - If not, it may be hard to balance work among the processors
- Number of tasks an increasing function of problem size
  - If not, it may be impossible to use more processors to solve large problem instances

Communication

- **Determine values passed among tasks**
  - ***Task-channel graph***
- **Local communication**
  - **Task needs values from a small number of other tasks**
  - **Create channels illustrating data flow**
- **Global communication**
  - **Significant number of tasks contribute data to perform a computation**
  - **Don't create channels for them early in design**

Communication Checklist

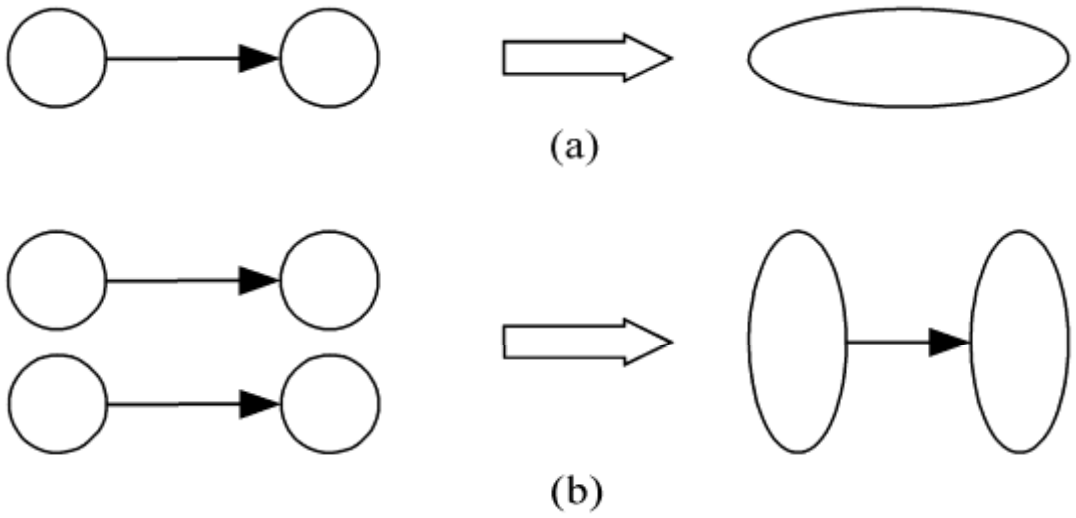
- Communication operations balanced among tasks
- Each task communicates with only small group of neighbors
- Tasks can perform communications concurrently
- Task can perform computations concurrently

Agglomeration

- Grouping tasks into larger tasks
- Goals
  - Improve performance
  - Maintain scalability of program
  - Simplify programming
- In message-passing programming, goal often to create one agglomerated task per processor

Agglomeration Can Improve Performance

- Eliminate communication between primitive tasks agglomerated into consolidated task
- Combine groups of sending and receiving tasks



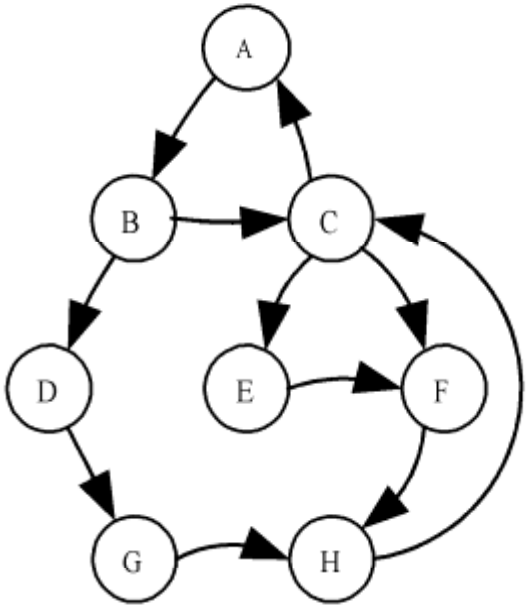
Agglomeration Checklist

- Locality of parallel algorithm has increased
- Replicated computations take less time than communications they replace
- Data replication doesn't affect scalability
- Agglomerated tasks have similar computational and communications costs
- Number of tasks increases with problem size
- Number of tasks suitable for likely target systems
- Tradeoff between agglomeration and code modifications costs is reasonable

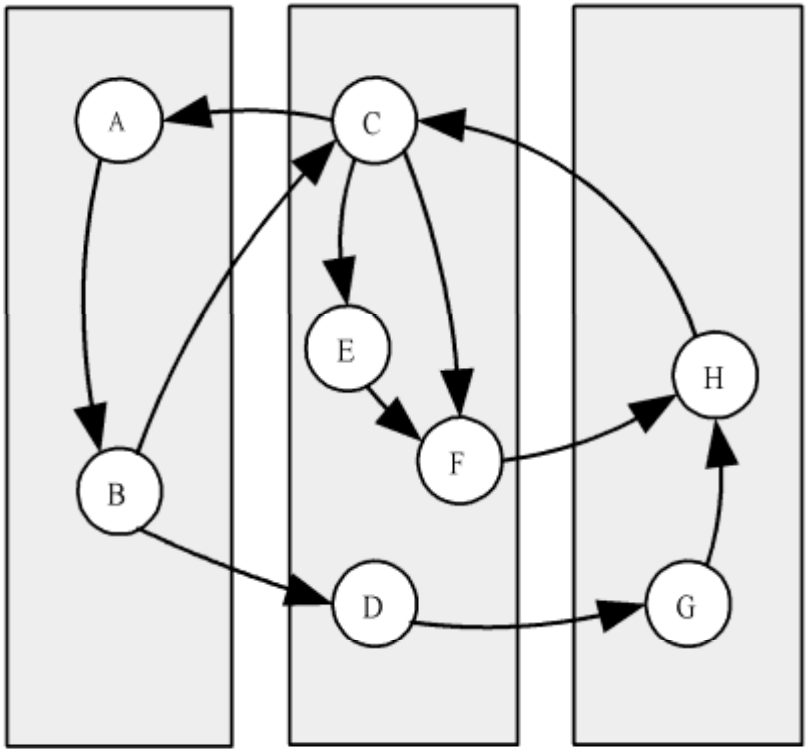
Mapping

- Process of assigning tasks to processors
  - Centralized multiprocessor: mapping done by operating system
  - Distributed memory system: mapping done by user
- Conflicting goals of mapping
  - Maximize processor utilization
  - Minimize interprocessor communication

Mapping Example



(a)



(b)

Optimal Mapping

- Finding optimal mapping is NP-hard
- Often rely on heuristics

Mapping Decision Tree

- **Static number of tasks**
  - **Structured communication**
    - **Constant computation time per task**
      - Agglomerate tasks to minimize comm
      - Create one task per processor
    - **Variable computation time per task**
      - Cyclically map tasks to processors
  - **Unstructured communication**
    - Use a static load balancing algorithm
- **Dynamic number of tasks**
  - **…**

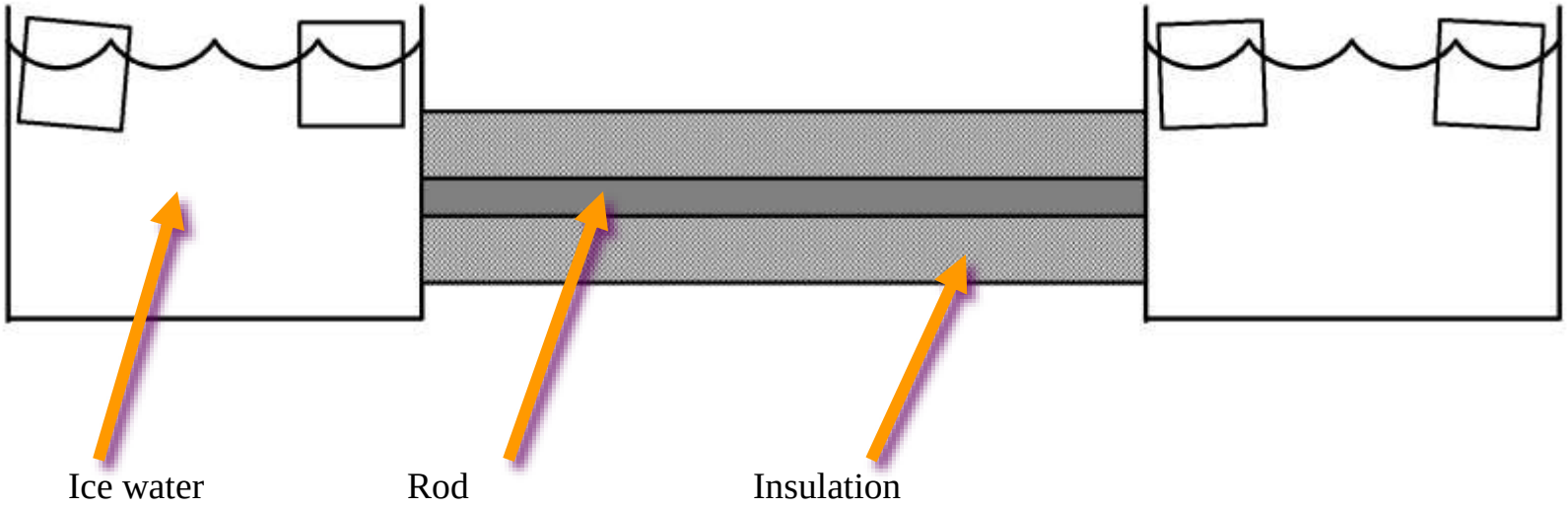
Mapping Strategy

- Static number of tasks
  - …
- Dynamic number of tasks
  - Frequent communications between tasks
    - Use a dynamic load balancing algorithm
  - Many short-lived tasks
    - Use a runtime task-scheduling algorithm

Mapping Checklist

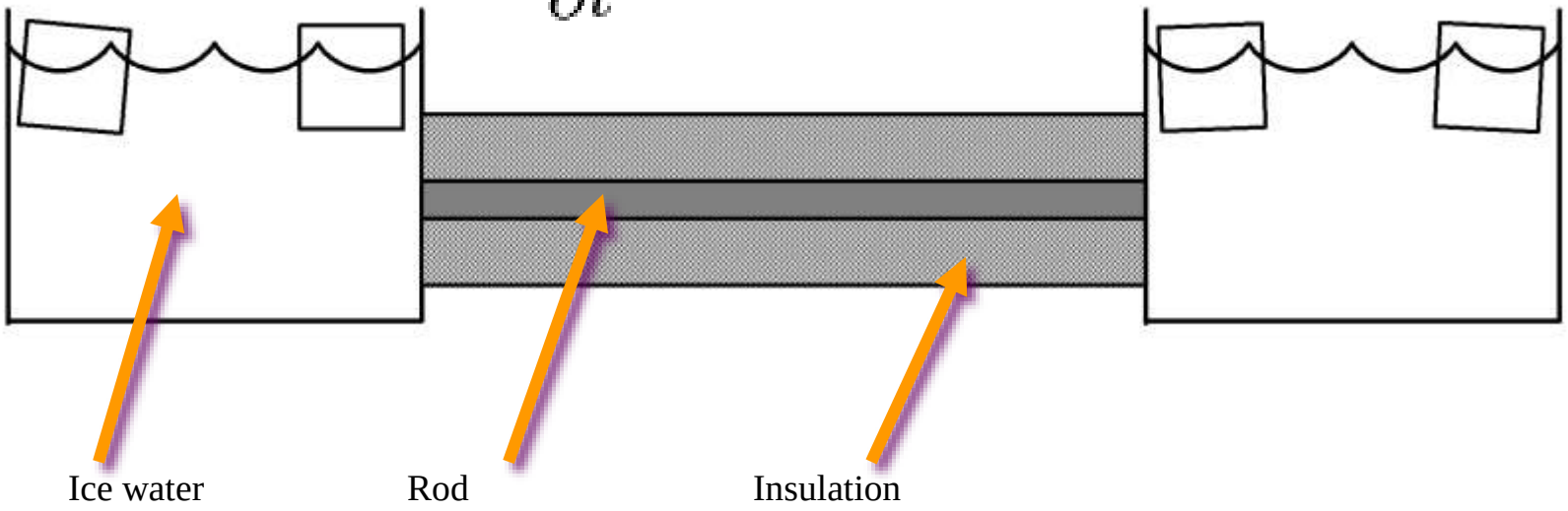
- Considered designs based on one task per processor and multiple tasks per processor
- Evaluated static and dynamic task allocation
- If dynamic task allocation chosen, task allocator is not a bottleneck to performance
- If static task allocation chosen, ratio of tasks to processors is at least 10:1

Case Study: Boundary Value Problem



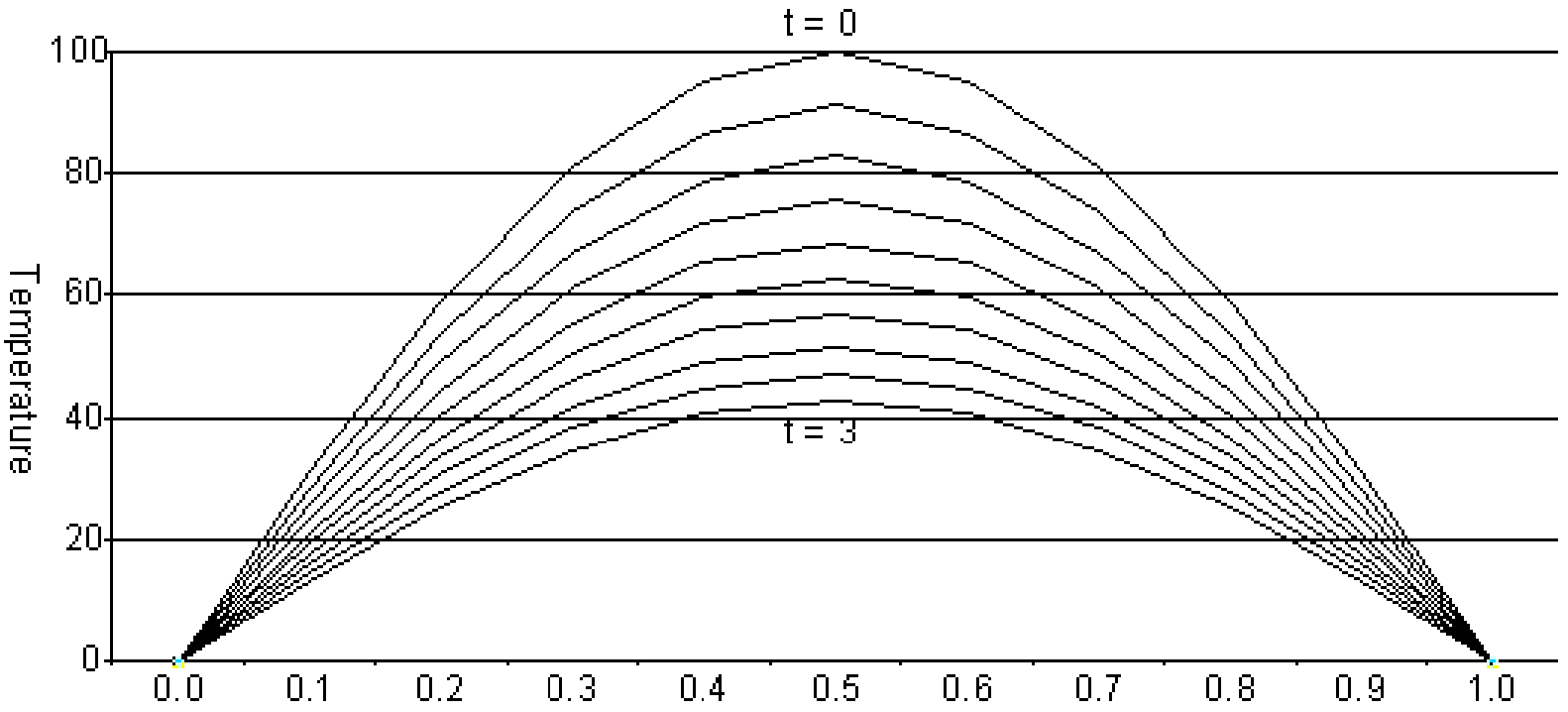
Problem Modeling

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$$

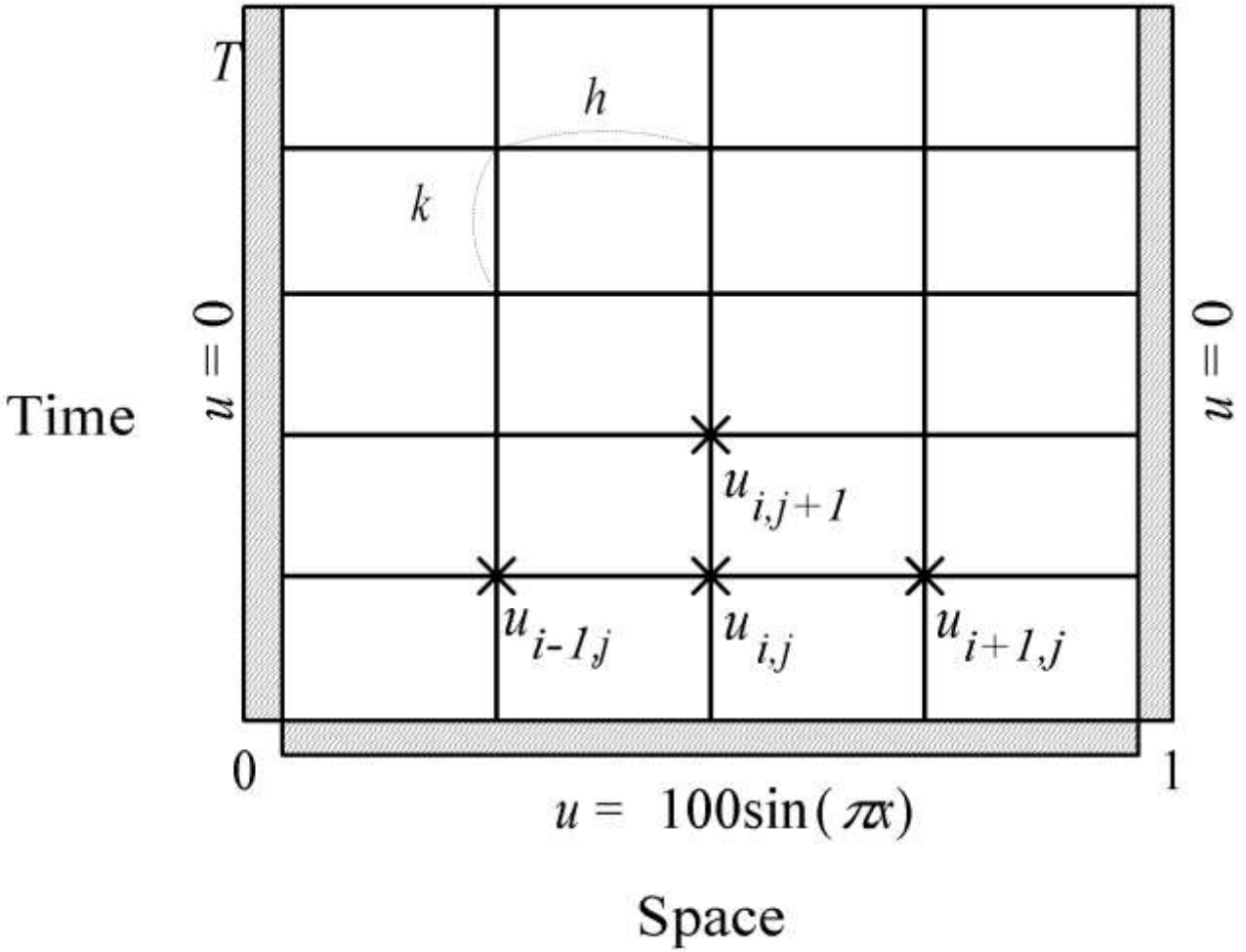


A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Rod Cools as Time Progresses



Finite Difference Approximation



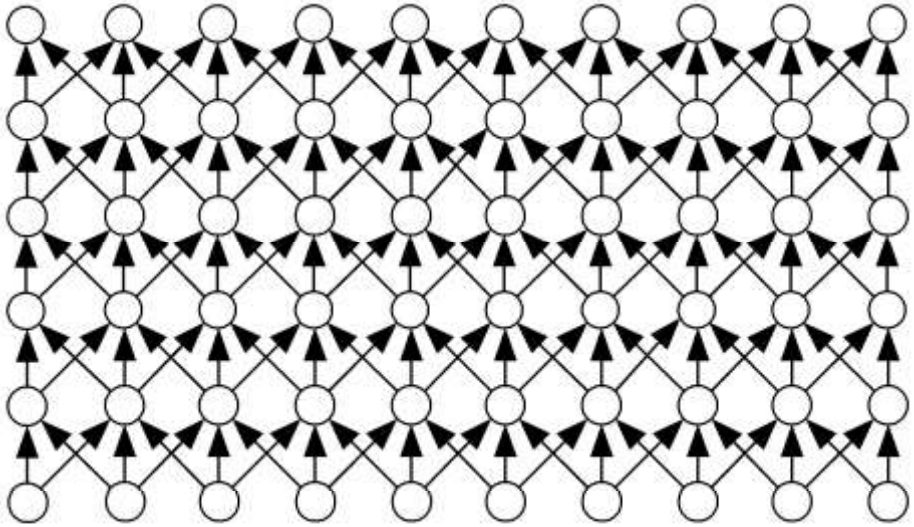
Partitioning

- One data item per grid point
- Associate one primitive task with each grid point
- Two-dimensional domain decomposition

Communication

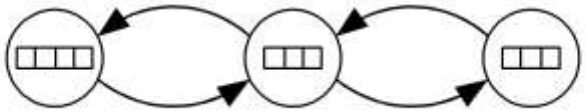
- Identify communication pattern between primitive tasks
- Each interior primitive task has three incoming and three outgoing channels

Agglomeration and Mapping



(a)

(b)



(c)

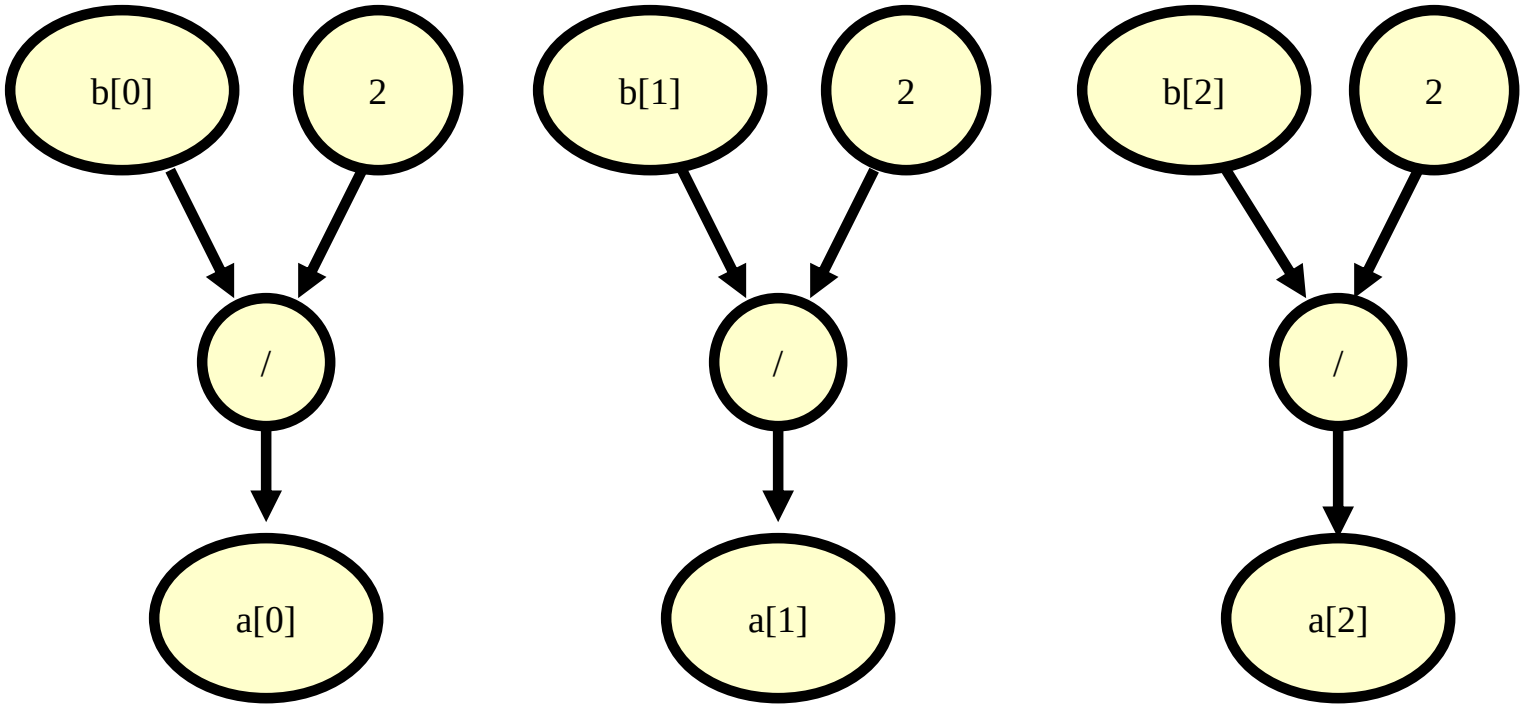
Agglomeration

Dependence Graph

- Directed graph = (nodes, edges)
- Node for each…
  - Variable assignment (except index variables)
  - Constant
  - Operator or function call
- Edges indicate data/control dependences
  - Data flow
    - New value of variable depends on another value
  - Control flow
    - New value of variable cannot be computed until condition is computed

Dependence Graph Example #1

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

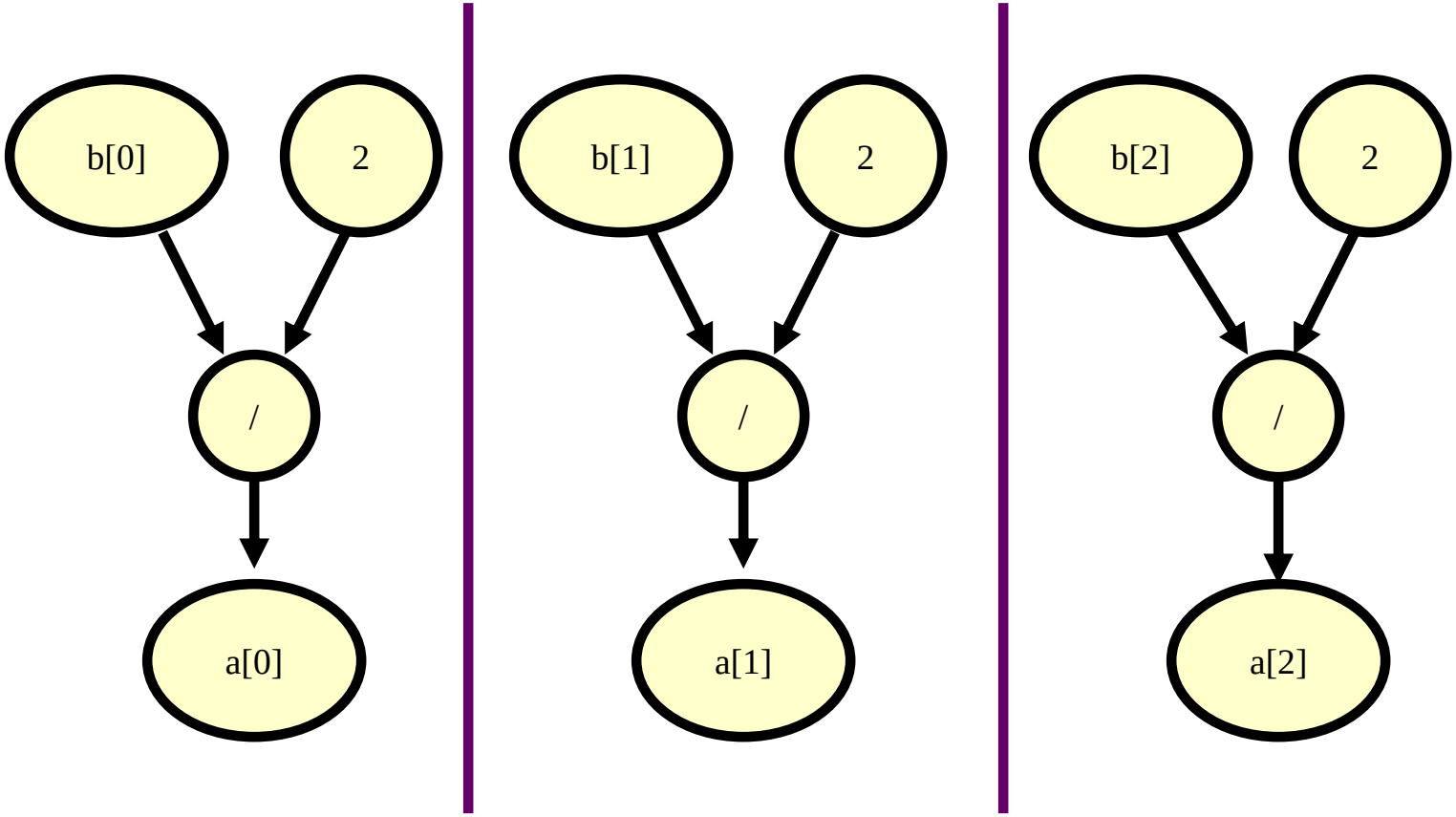


Dependence Graph Example #1

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

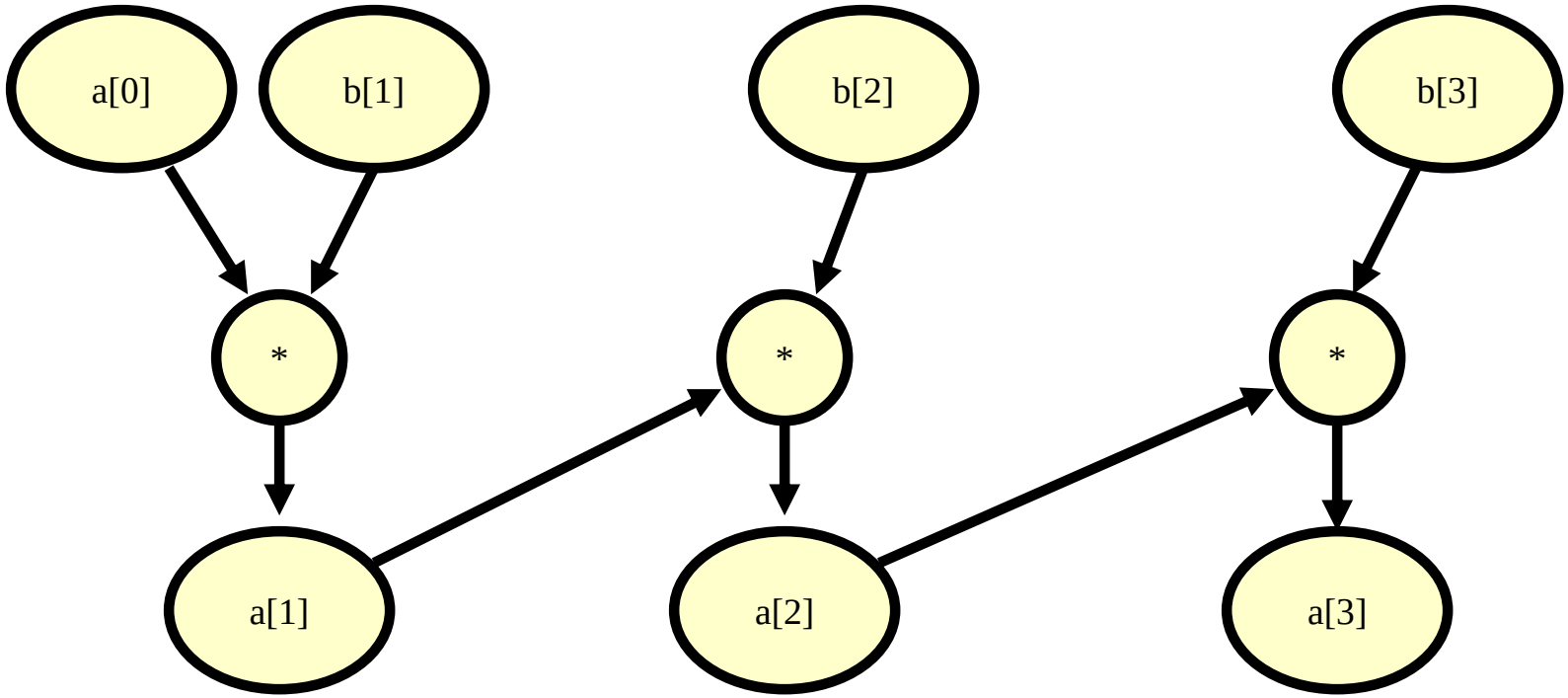
Domain decomposition

possible



Dependence Graph Example #2

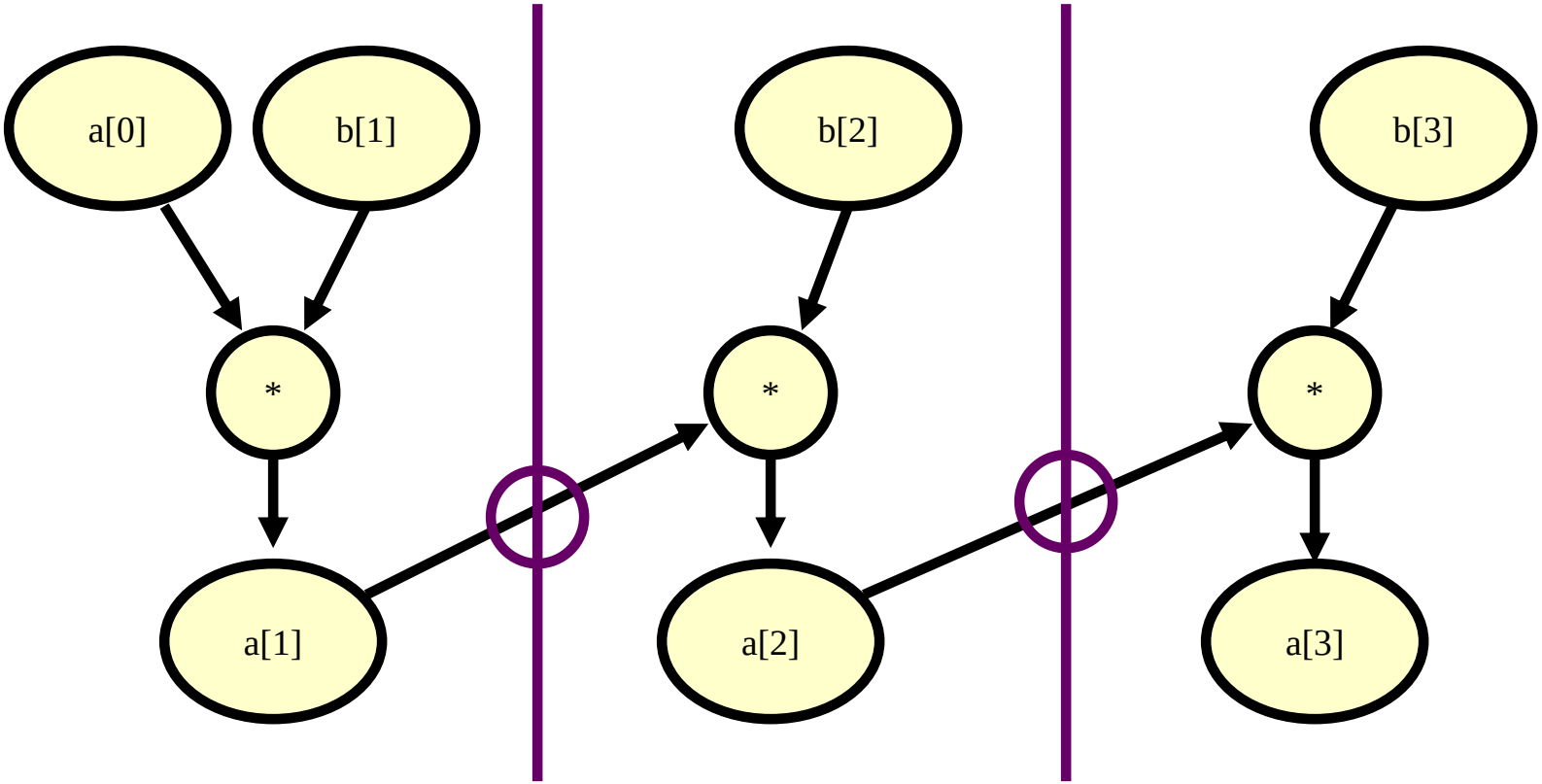
```
for (i = 1; i < 4; i++)
  a[i] = a[i-1] * b[i];
```



Dependence Graph Example #2

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

No domain decomposition



Dependence Graph Example #3

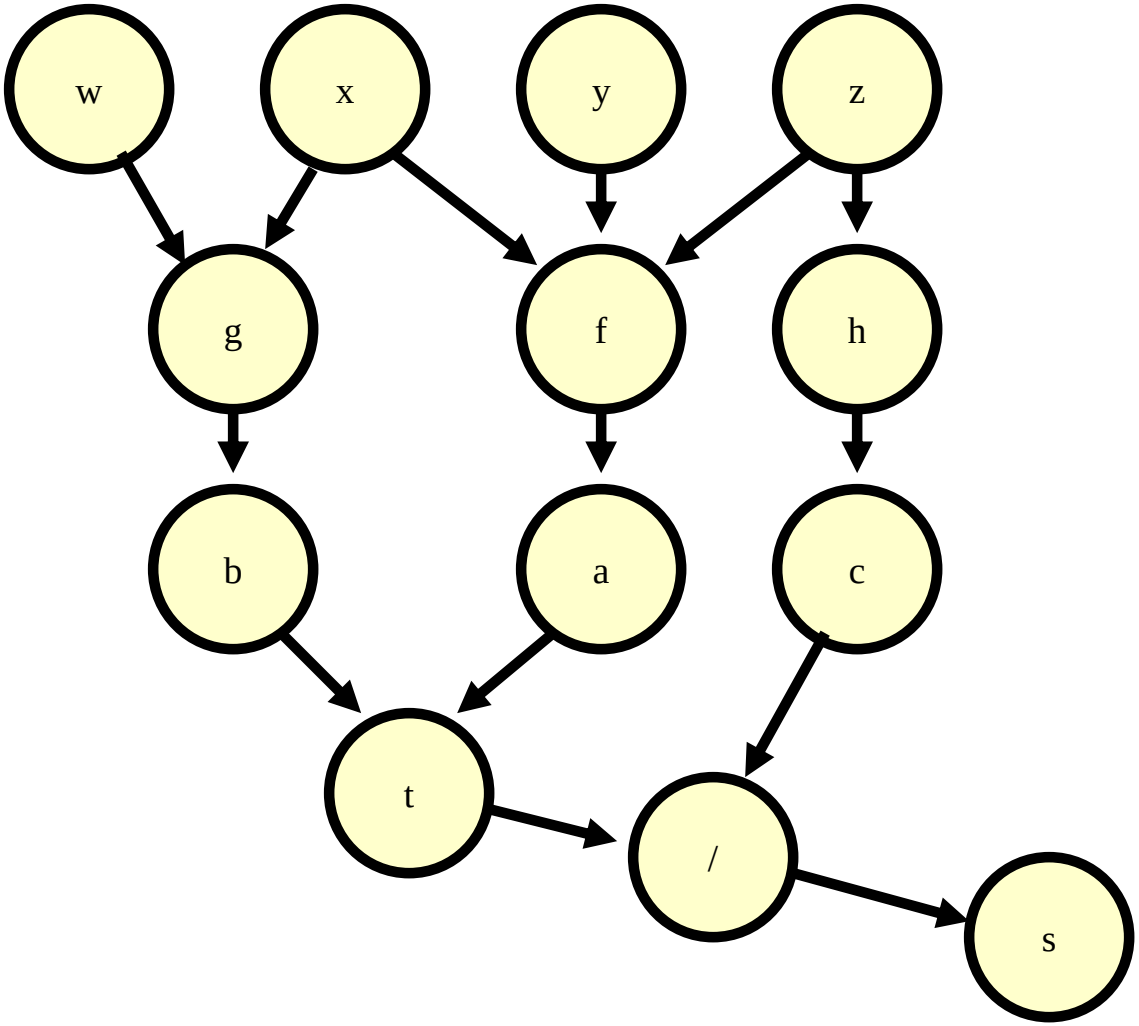
a = f(x, y, z);

b = g(w, x);

t = a + b;

c = h(z);

s = t / c;



Dependence Graph Example #3

a = f(x, y, z);

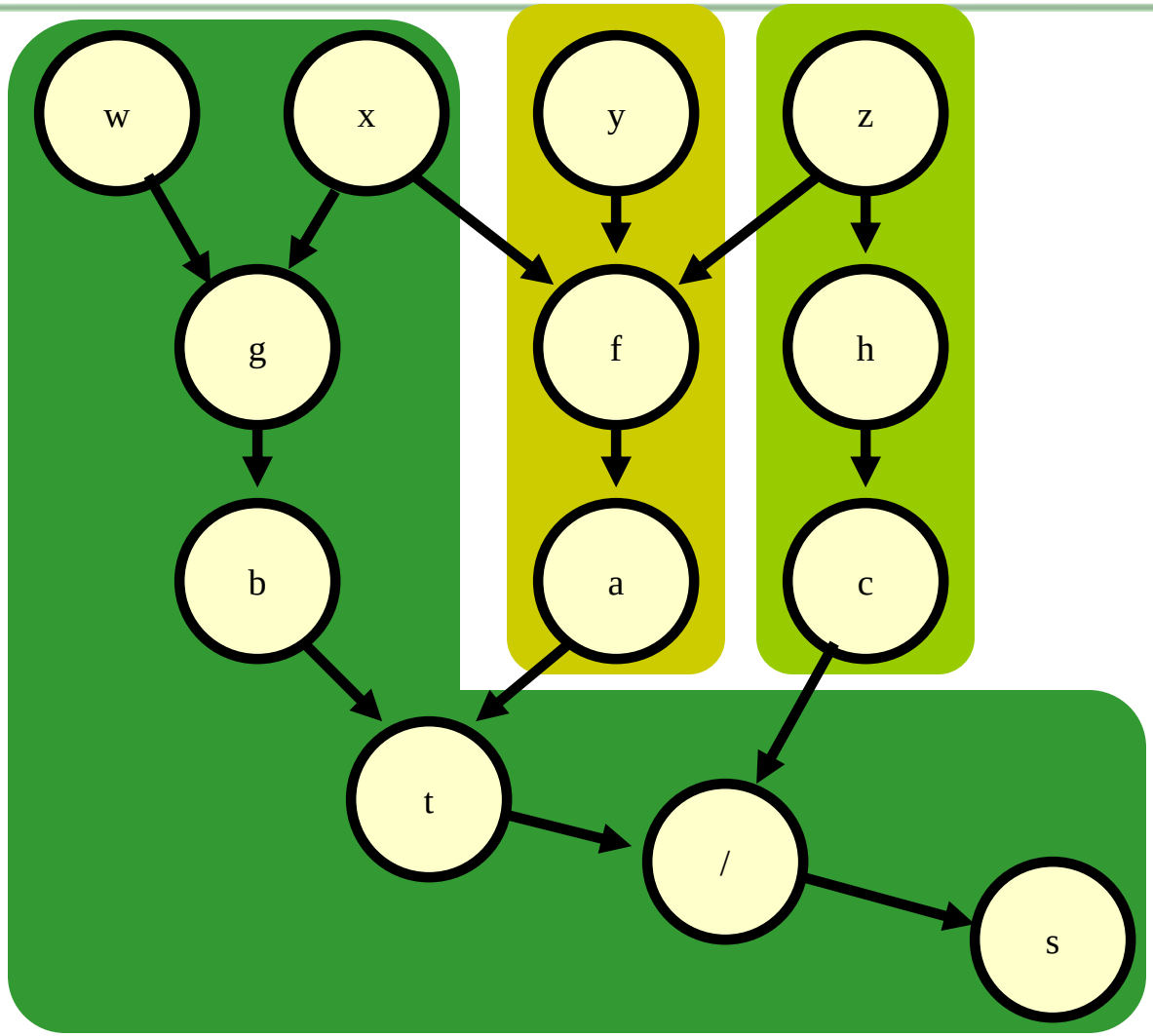
b = g(w, x);

t = a + b;

c = h(z);

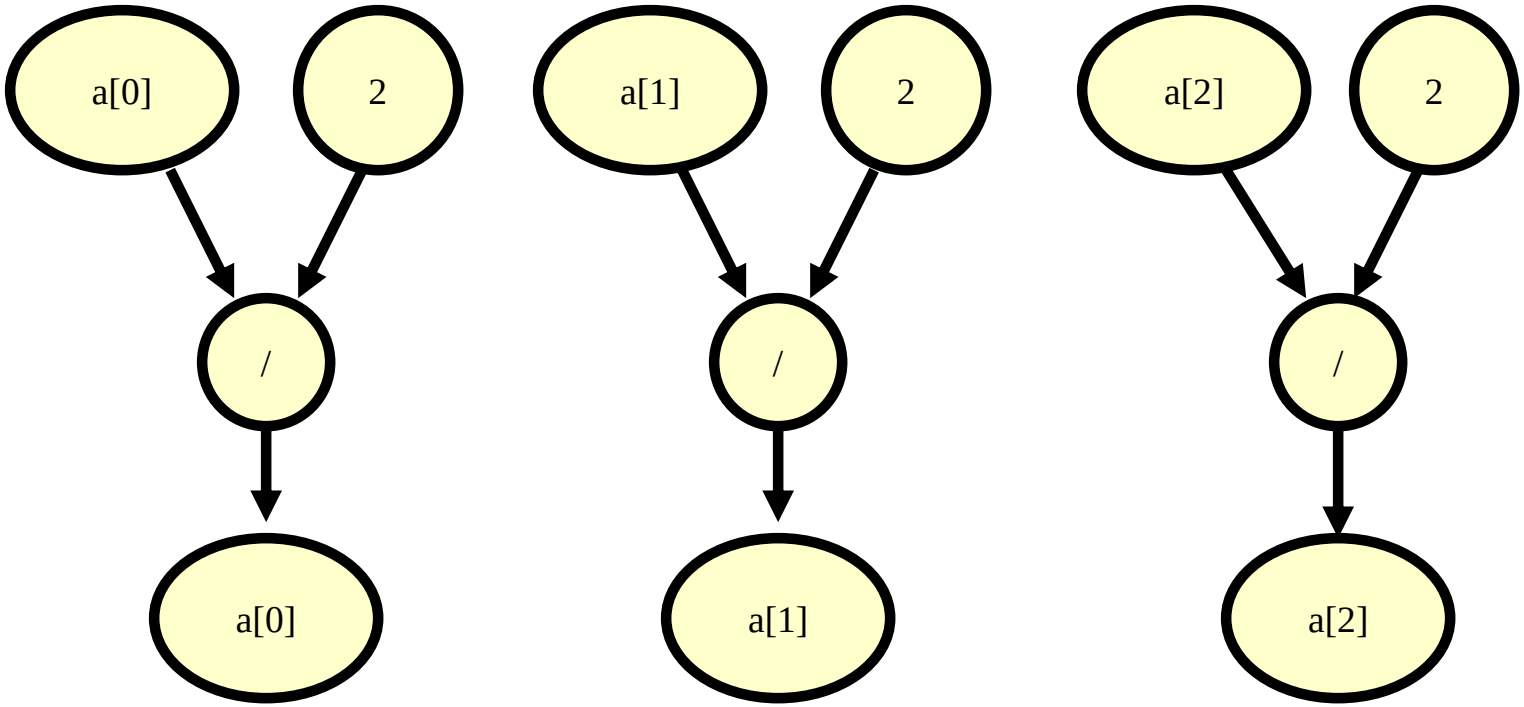
s = t / c;

Task decomposition with 3 CPUs.



Dependence Graph Example #4

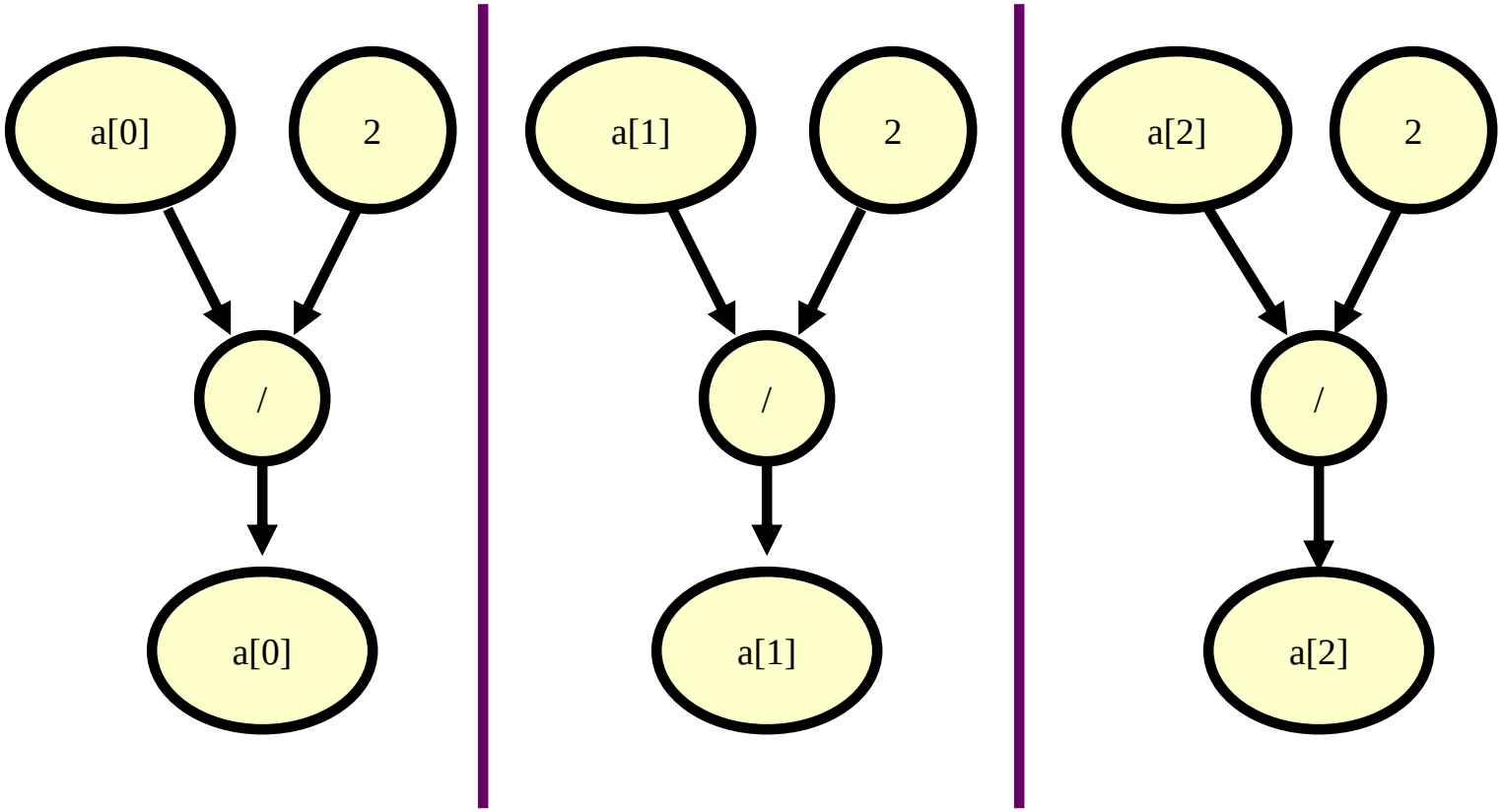
```
for (i = 0; i < 3; i++)
  a[i] = a[i] / 2.0;
```



Dependence Graph Example #4

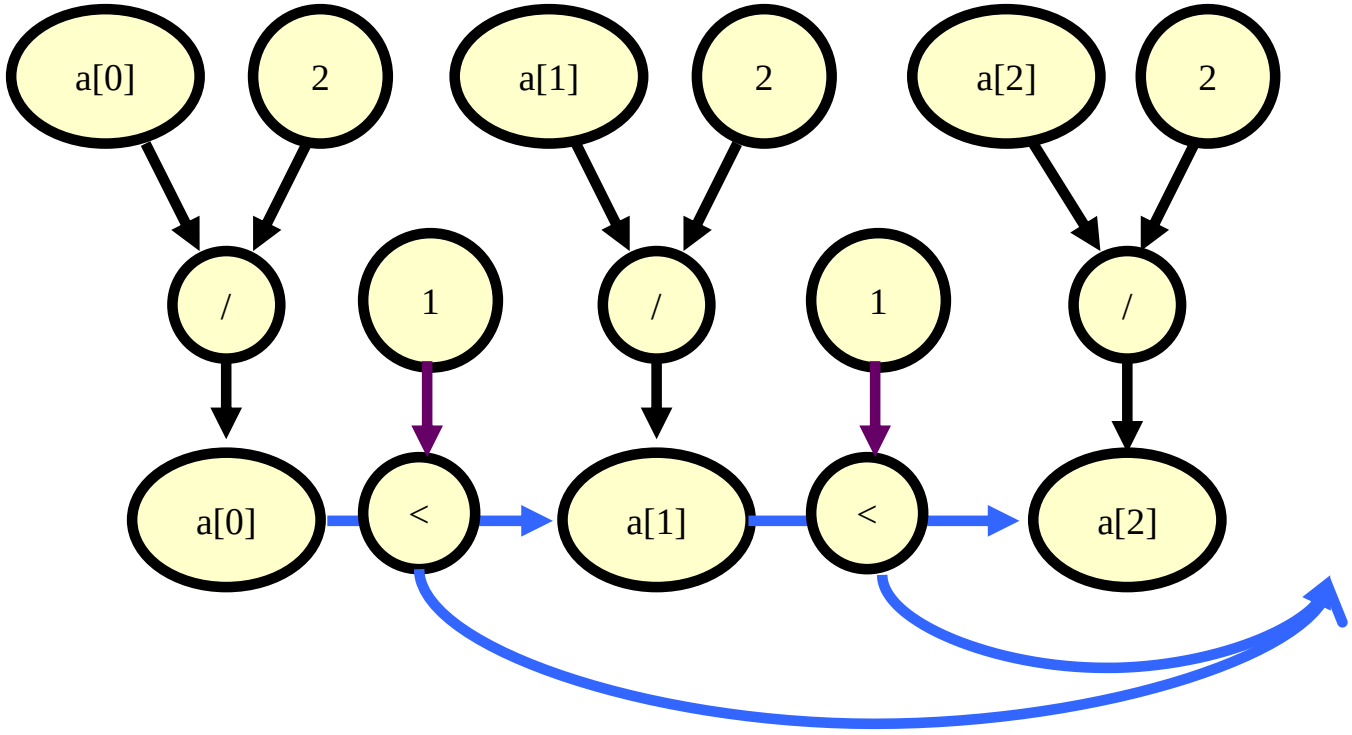
```
for (i = 0; i < 3; i++)
  a[i] = a[i] / 2.0;
```

Domain decomposition



Dependence Graph Example #5

```
for (i = 0; i < 3; i++) {
   a[i] = a[i] / 2.0;
   if (a[i] < 1.0) break;
}
```



Can You Find the Parallelism?

- Resizing a photo
- Searching a document for all instances of a word
- Updating a spreadsheet
- Compiling a program
- Prefetching pages in a Web browser
- Using a word processor to type a report

Good/Bad Opportunities for a Parallel Solution

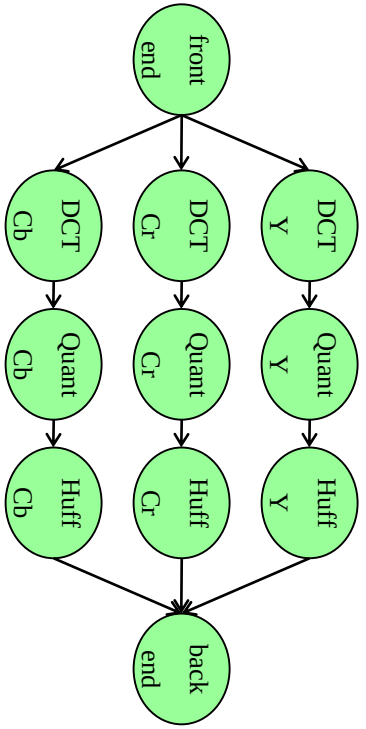
Parallel Solution Easier	Parallel Solution More Difficult or Even Impossible
Larger data sets	Smaller data sets
Dense matrices	Sparse matrices
Dividing space among processors	Dividing time among processors

More on Exploring Parallelism

- **Three basic types of parallelism**
  - **Data parallel**
  - **Task (function) parallel**
  - **Pipelining**
- **Some of them can be combined**
  - **Example: task parallel + pipelining**

Example: Multi-processor Synthesis under Throughput Constraints

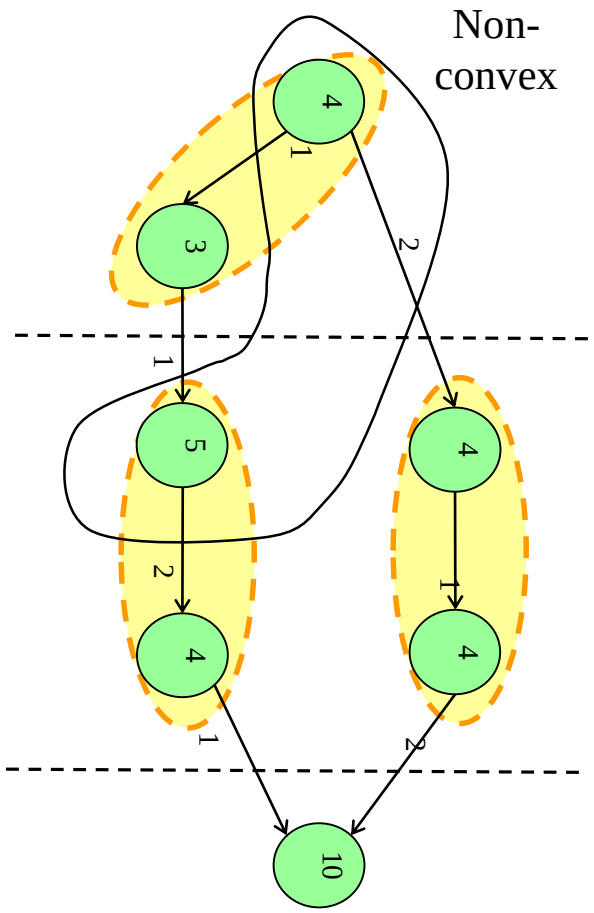
- **Periodic task graph *G(V,E)***
  - Each task is annotated with the estimated execution time
  - Each edge is annotated with the data communication volume between tasks
    - Linear communication delay model
    - Buffer memory to store the incoming data
- **Reference**
  - J. Cong, G. Han, and W. Jiang, "Synthesis of an Application-Specific Soft Multiprocessor System ," *International Symposium on Field Programmable Gate Arrays*, 2007



Task Graph of Motion JPEG

Some Basic Definitions

- **Stage period**
  - the reciprocal of the throughput
- **Latency**
  - the elapsed time from the data input streams to the output streams
- **Convex cluster**



The Synthesis/Mapping Problem

Given: a task graph $G(V, E)$ with profiling information and stage period constraint T

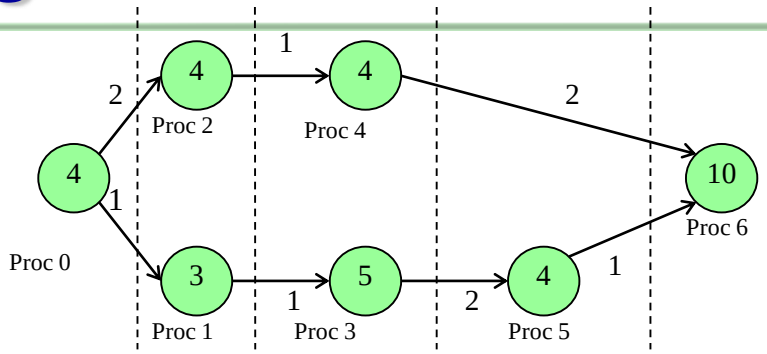
Problem: construct a multiprocessor system by

(i) partitioning the tasks into convex clusters

(ii) mapping the clusters onto the processors and inter-processor communication to FIFOs

Objective: minimize latency and resource usage under the required throughput constraint

Exploration Space: Throughput, Latency & Resources

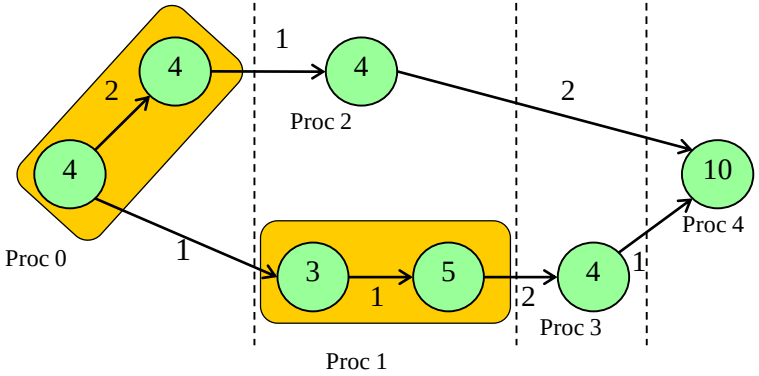


(a)

10*us* stage period

5 stages

7 processors

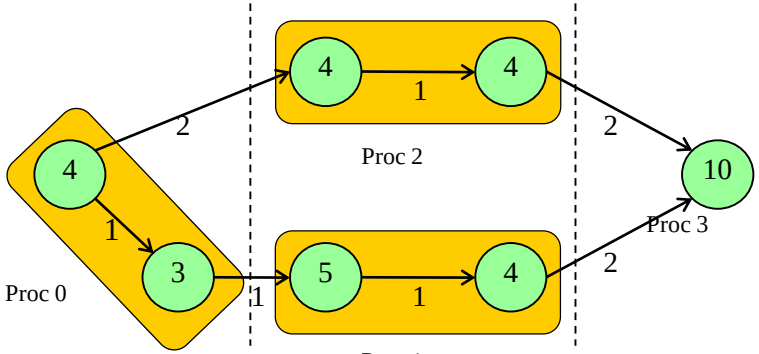


(b)

10*us* stage period

4 stages

5 processors



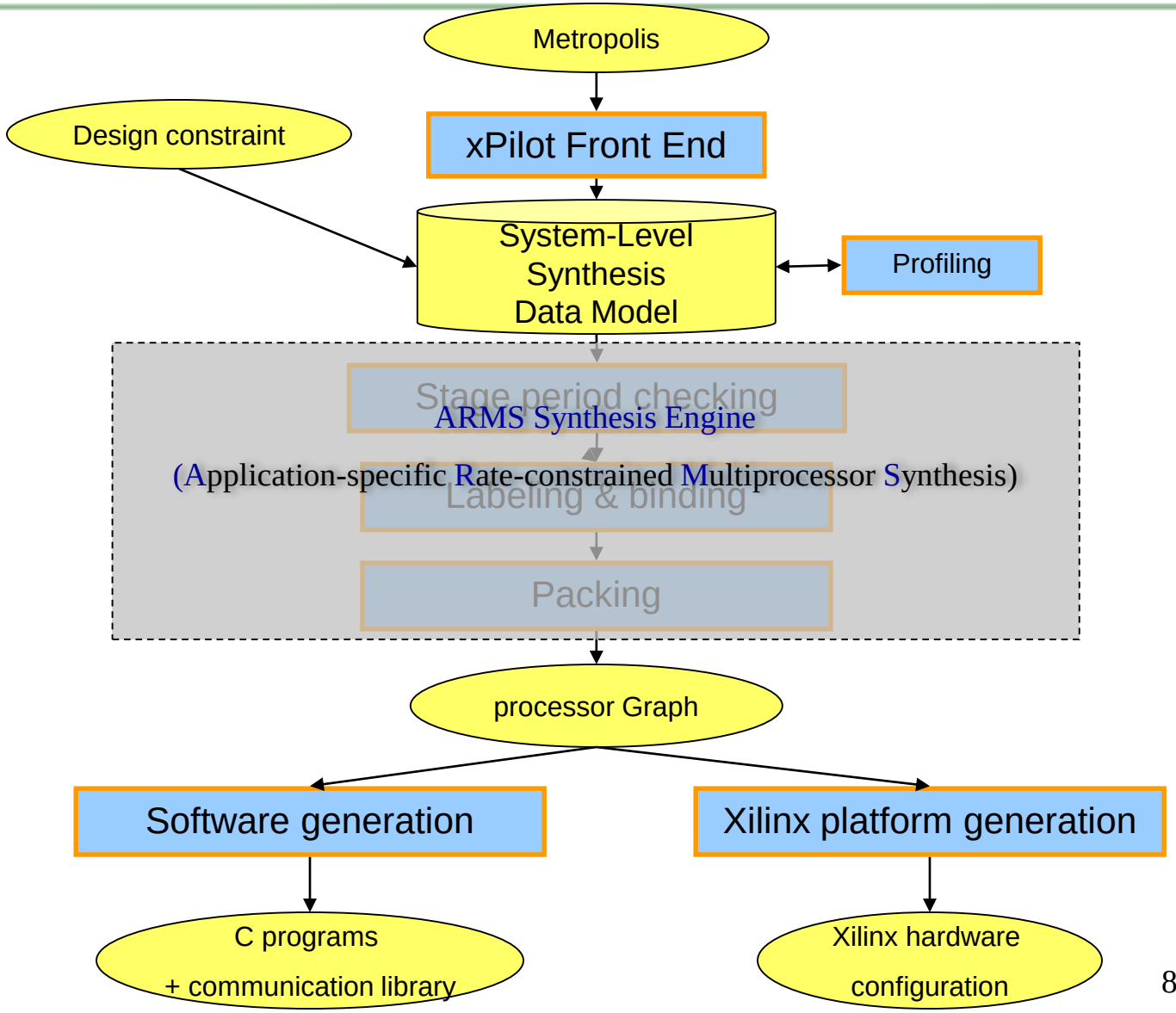
(c)

10*us* stage period

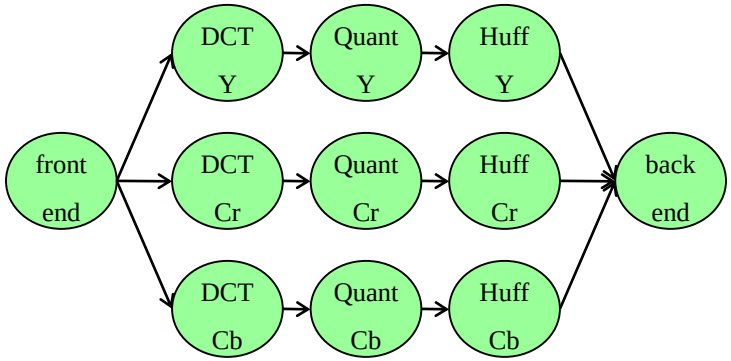
3 stages

4 processors

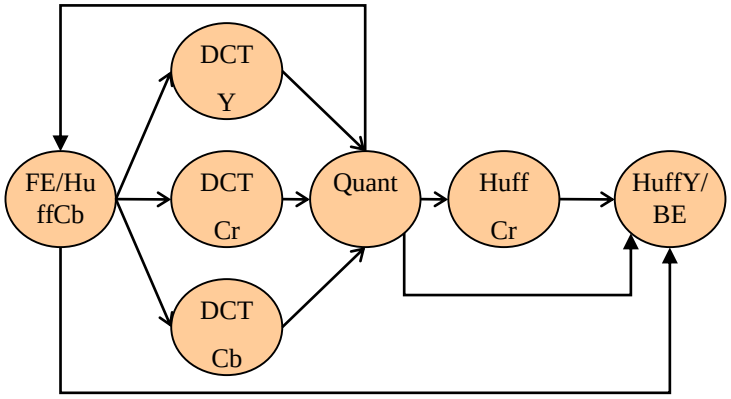
Synthesis Flow [FPGA'2007]

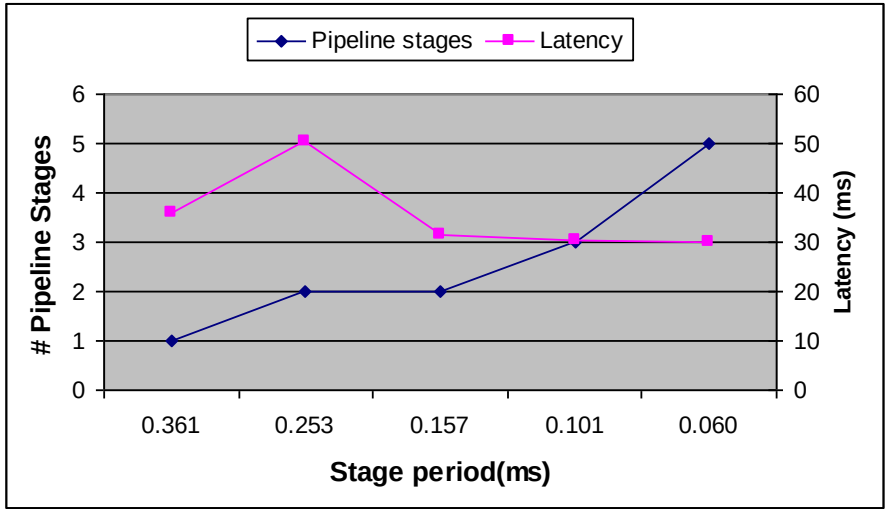


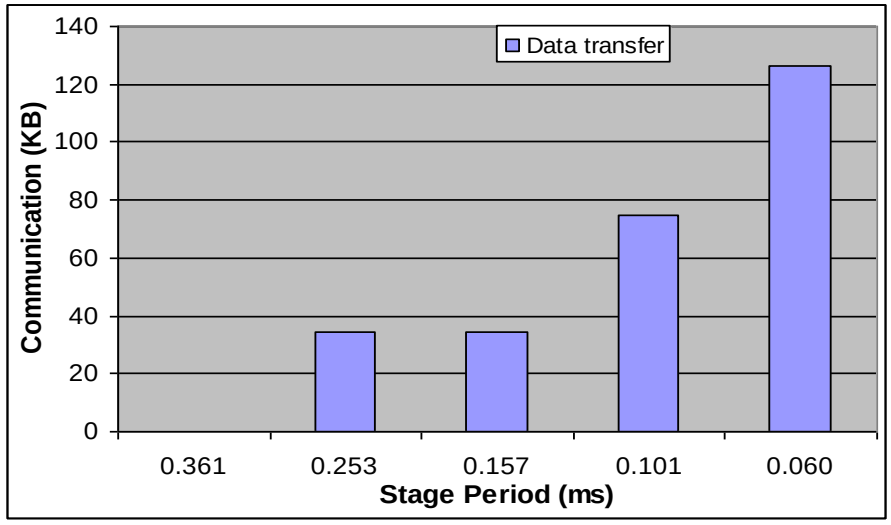
Experimental Results



- **Implemented on Xilinx Virtex II Pro platform FPGAs**
  - **Use Microblazes and FSLs**







Summary: Design Steps

- Partition computation
- Agglomerate tasks
- Map tasks to processors
- Goals
  - Maximize processor utilization
  - Minimize inter-processor communication
- Good designs
  - Maximize local computations
  - Minimize communications
  - Scalable