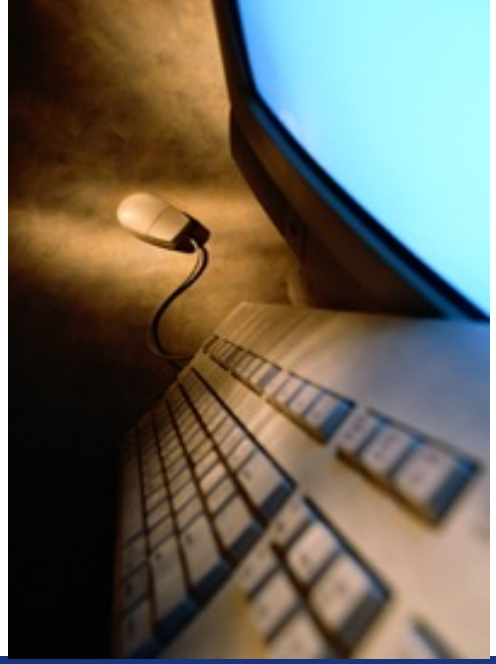
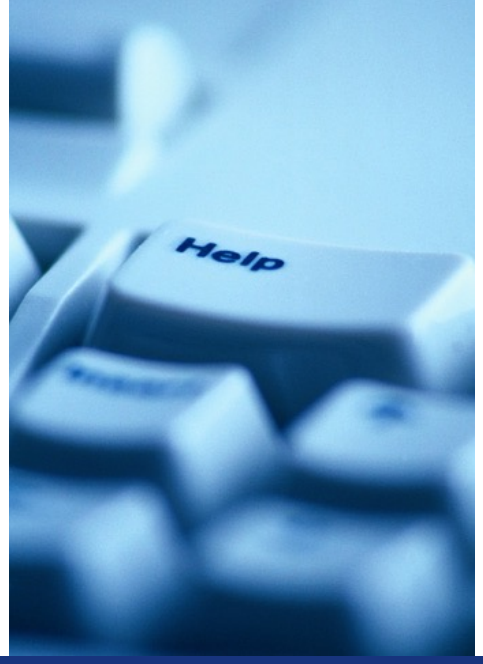
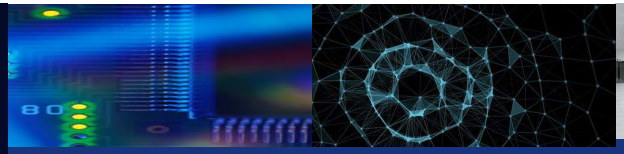



并行与分布式计算

Parallel & Distributed Computing

陈鹏飞
数据科学与计算机学院
2021-10-15

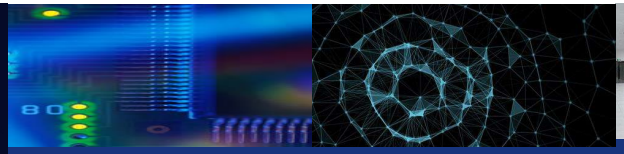


Lecture 4 — Parallel Programming Methodology

Pengfei Chen

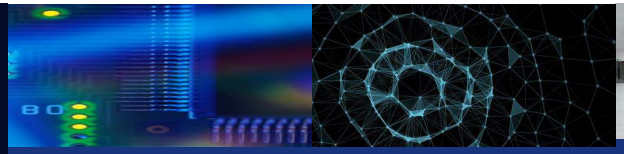
School of Data and Computer Science

Oct 15, 2021



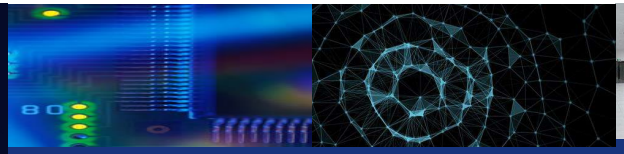
Outline:

1. Review and examples
2. Incremental parallelization
3. Design methodology for parallel algorithm
4. Use of dependency graph



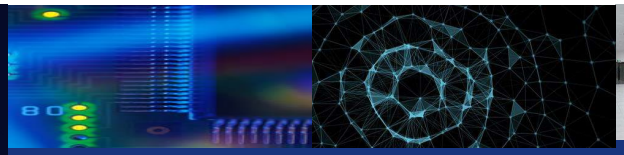
Parallel and Distributed Computing

REVIEW AND EXAMPLE



Review: Parallel Programming models

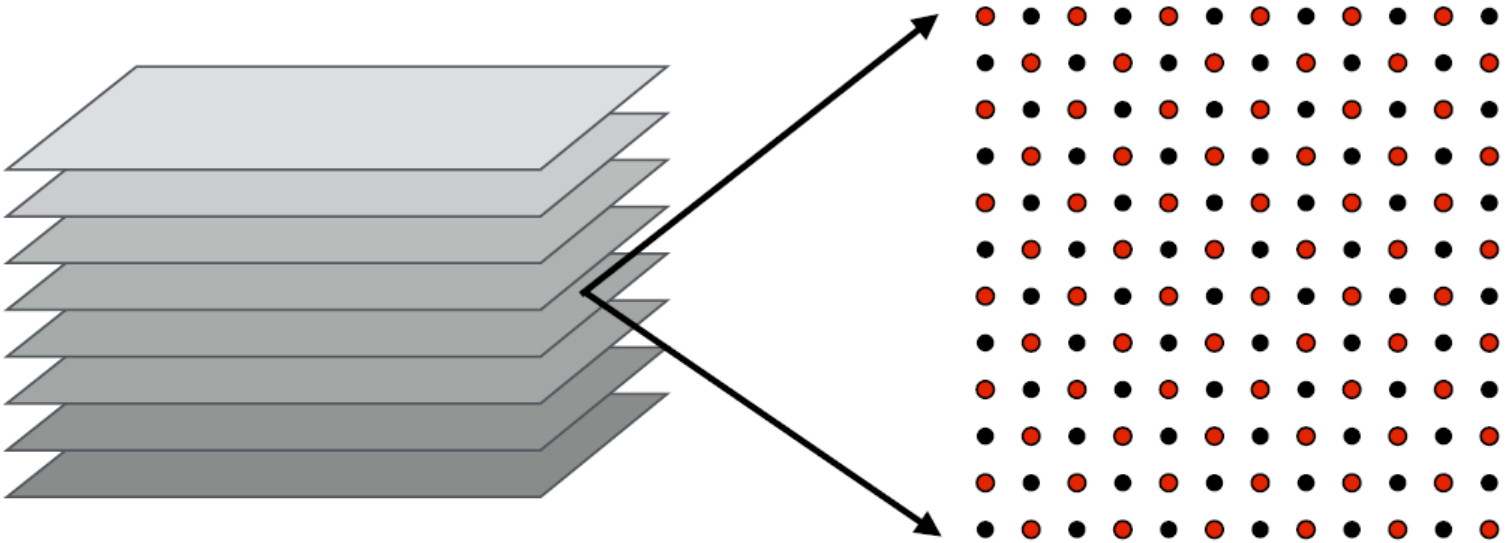
- **Shared address space**
  - Communication is unstructured, implicit in loads and stores
  - Natural way of programming, but can shoot yourself in the foot easily
    - Program might be correct, but not perform well
- **Message passing**
  - Structure all communication as messages
  - Often harder to get first correct program than shared address space
  - Structure often helpful in getting to first correct, scalable program
- **Data parallel**
  - Structure computation as a big “map” over a collection
  - Assumes a shared address space from which to load inputs/store results, but model severely limits communication between iterations of the map (goal: preserve independent processing of iterations)
  - Modern embodiments encourage, but don’t enforce, this structure



Modern Practice: Mixed Programming Models

- **Use shared address space programming within a multi-core node of a cluster, use message passing between nodes**
  - Very, very common in practice
  - Use convenience of shared address space where it can be implemented efficiently (within a node), require explicit communication elsewhere
- **Data-parallel-ish programming models support shared-memory style synchronization primitives in kernels**
  - Permit limited forms of inter-iteration communication (e.g., CUDA, OpenCL)
- **CUDA/OpenCL use data-parallel model to scale to many cores, but adopt shared-address space model allowing threads running on the same core to communicate.**

Example-1: Simulating of ocean currents

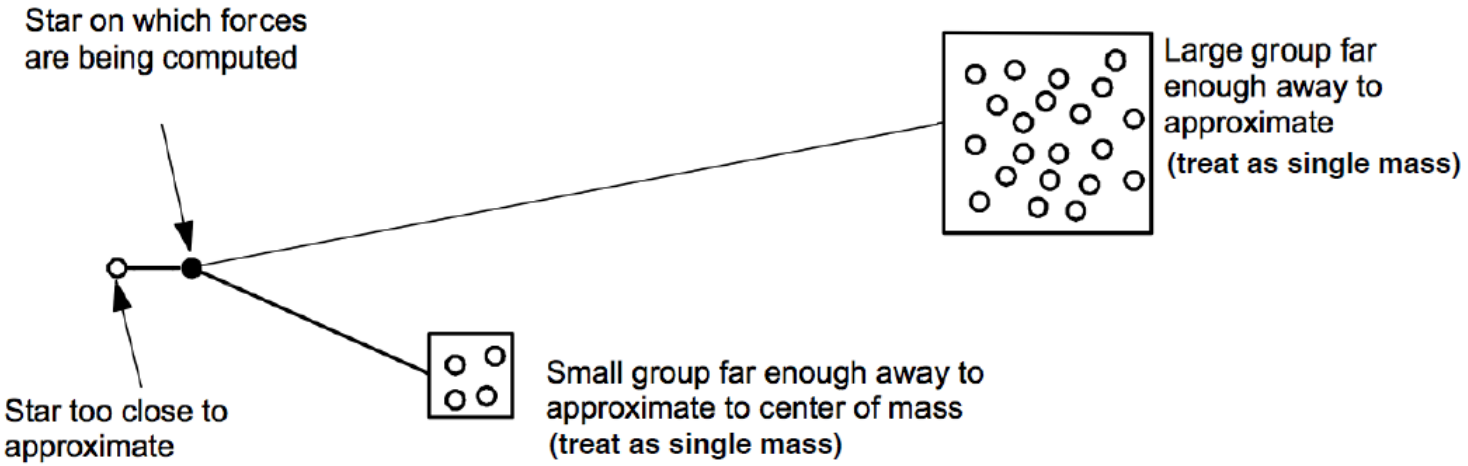


- Discretize 3D ocean volume into slices represented as 2D grids
- Discretize time evolution of ocean: Δt
- High accuracy simulation requires small Δt and high resolution grids

Reference: Parallel Computer Architecture - A Hardware Software Approach, Chapter 2.

Example-2: Galaxy evolution

Barnes-Hut algorithm



- **Represent galaxy as a collection of N particles (think: particle = star)**
- **Compute forces on each particle due to gravity**
  - **Naive algorithm is $O(N^2)$ — all particles interact with all others (gravity has infinite extent)**
  - **Magnitude of gravitational force falls off with distance (so algorithms approximate forces from far away stars to gain performance)**
  - **Result is an O(NlgN) algorithm for computing gravitational forces between all stars**

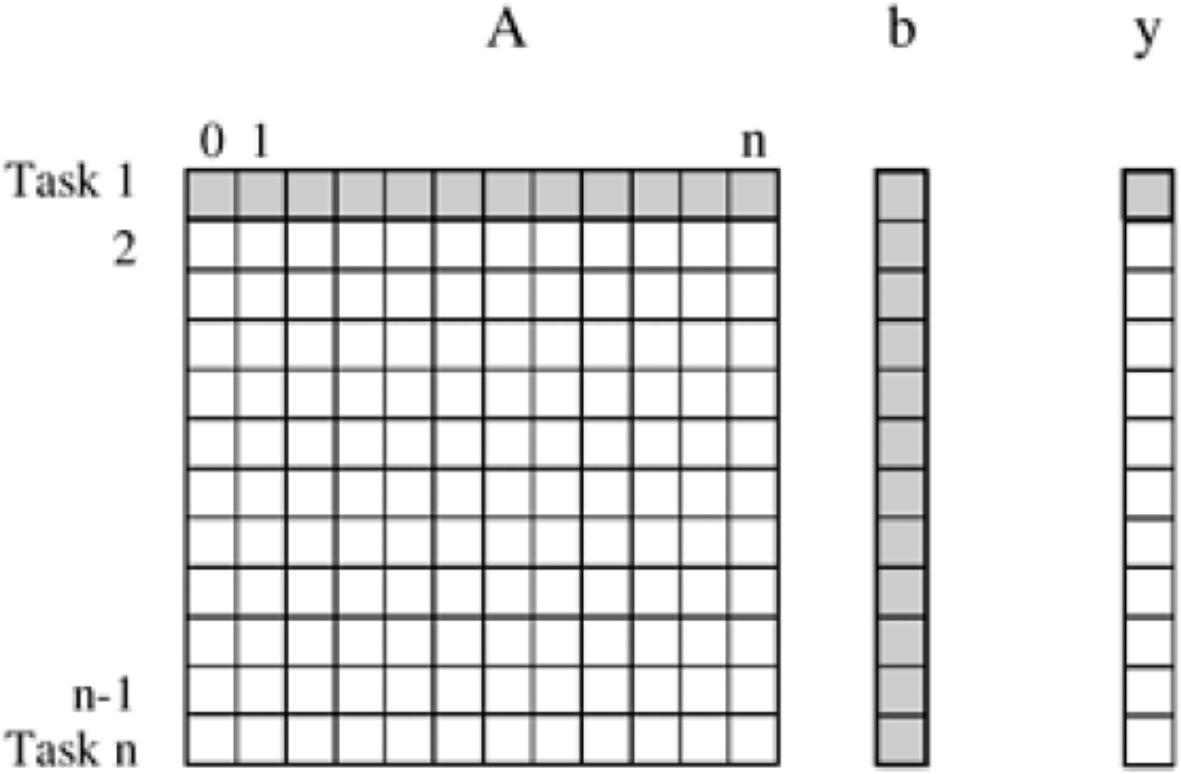
Reference: Parallel Computer Architecture - A Hardware Software Approach, Chapter 2.

Example-3: Black hole（黑洞）

黑洞图片

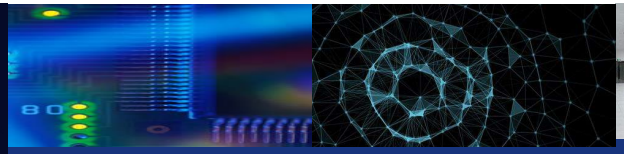
MIT博士生凯蒂与她收到的硬盘

Example-4: Dense matrix-vector multiplication



Decomposition of dense matrix-vector multiplication into *n* tasks, where *n* is the number of rows in the matrix. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.

Reference: Introduction to parallel computing, the second edition, Chapter 3.

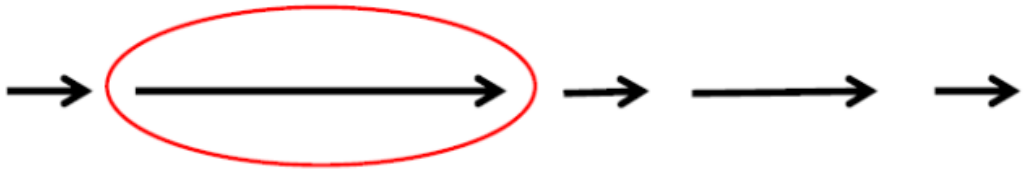


Parallel and Distributed Computing

INCREMENTAL PARALLELIZATION (增量式并行化)

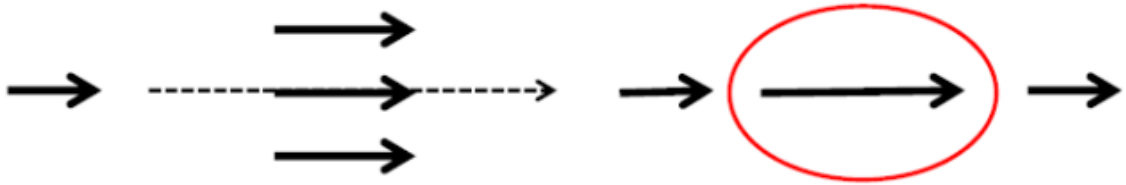
Incremental parallelization

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



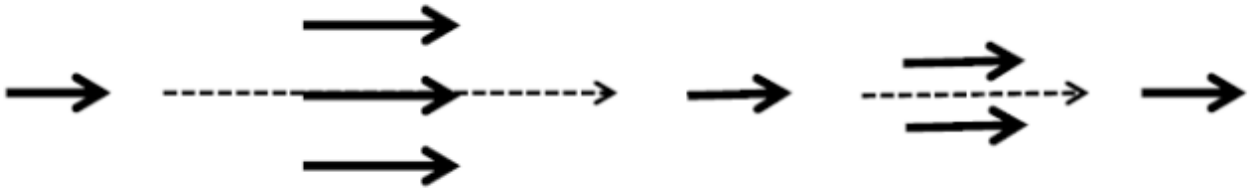
Incremental parallelization

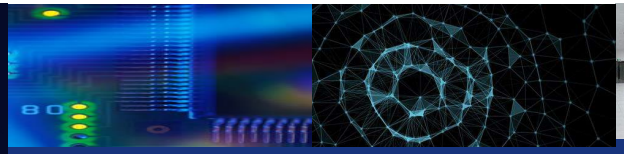
- Study a sequential program (or code segment)
- Look for bottlenecks & opportunities for parallelism
- Try to keep all processors busy doing useful work



Incremental parallelization

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

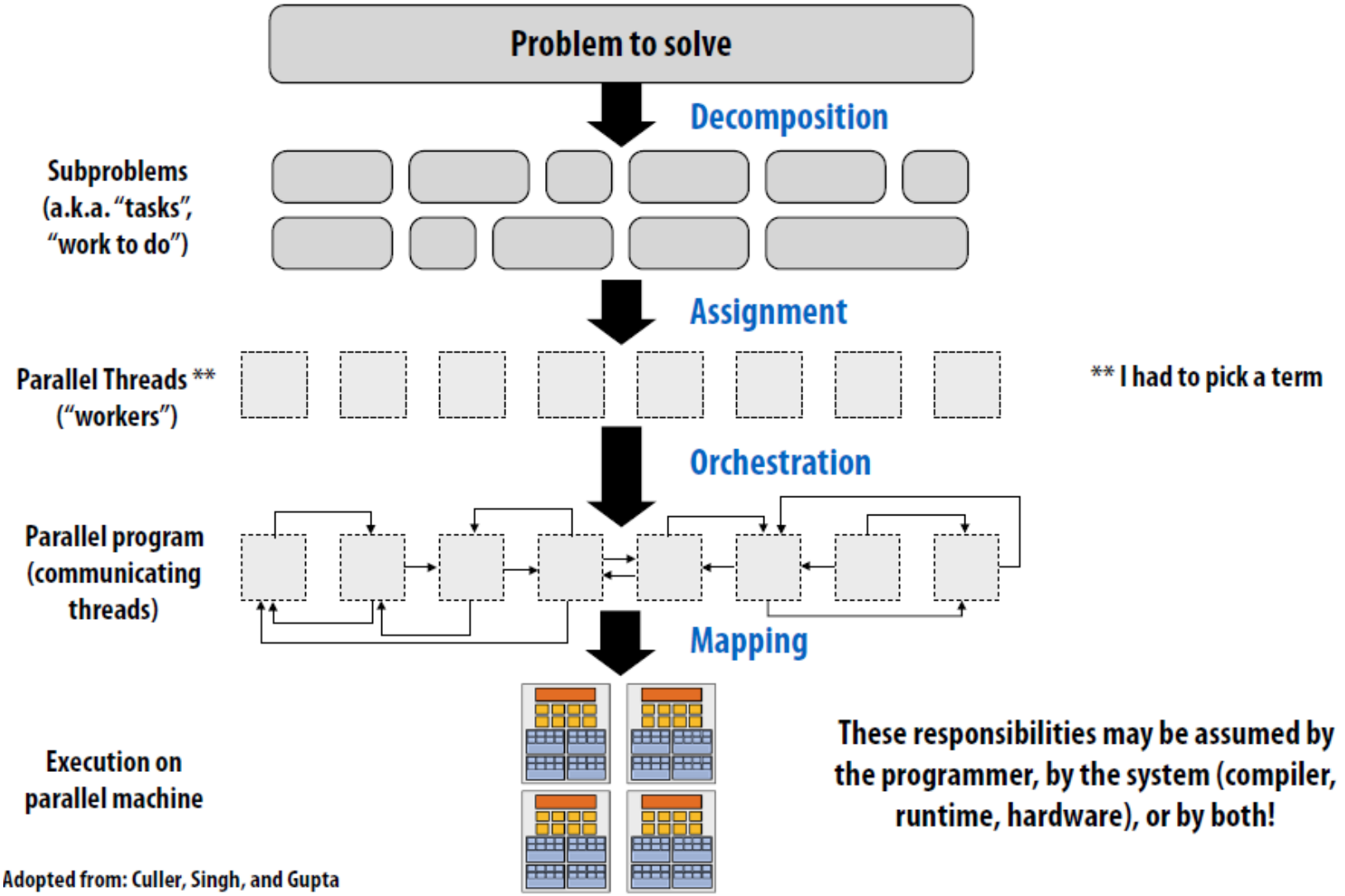




Parallel and Distributed Computing

METHODOLOGY OF PARALLIZATION (并行化方法论)

Culler's Design Methodology



Adopted from: Culler, Singh, and Gupta

Reference: Parallel Computer Architecture - A Hardware Software Approach, Chapter 2.

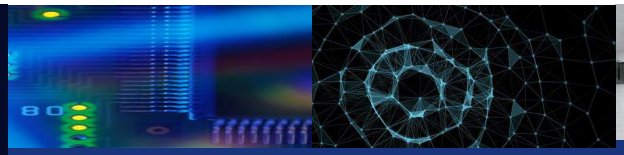
Decomposition

Break up problem into tasks that can be carried out in parallel

- Decomposition need not happen statically
- New tasks can be identified as program executes

Main idea: create *at least* enough tasks to keep all execution units on a machine busy

Key aspect of decomposition: identifying dependencies (or... a lack of dependencies)



Decomposition

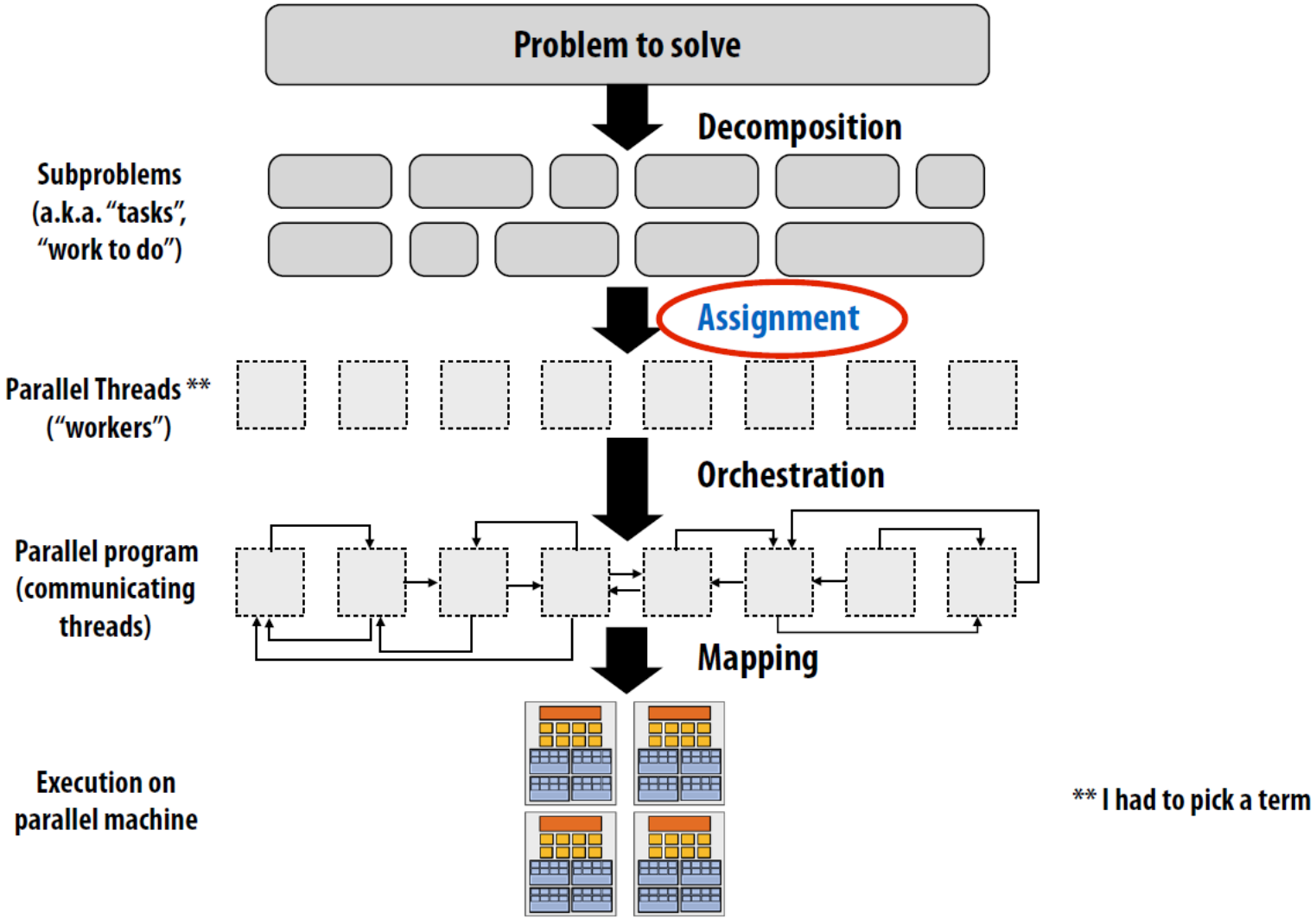
Who is responsible for performing decomposition?

- In most cases: the programmer

Automatic decomposition of sequential programs continues to be a challenging research problem (very difficult in general case)

- Compiler must analyze program, identify dependencies
  - What if dependencies are data dependent (not known at compile time)?
- Researchers have had modest success with simple loop nests
- The "magic parallelizing compiler" for complex, general-purpose code has not yet been achieved

Assignment



** I had to pick a term

Assignment

Assigning tasks to threads **

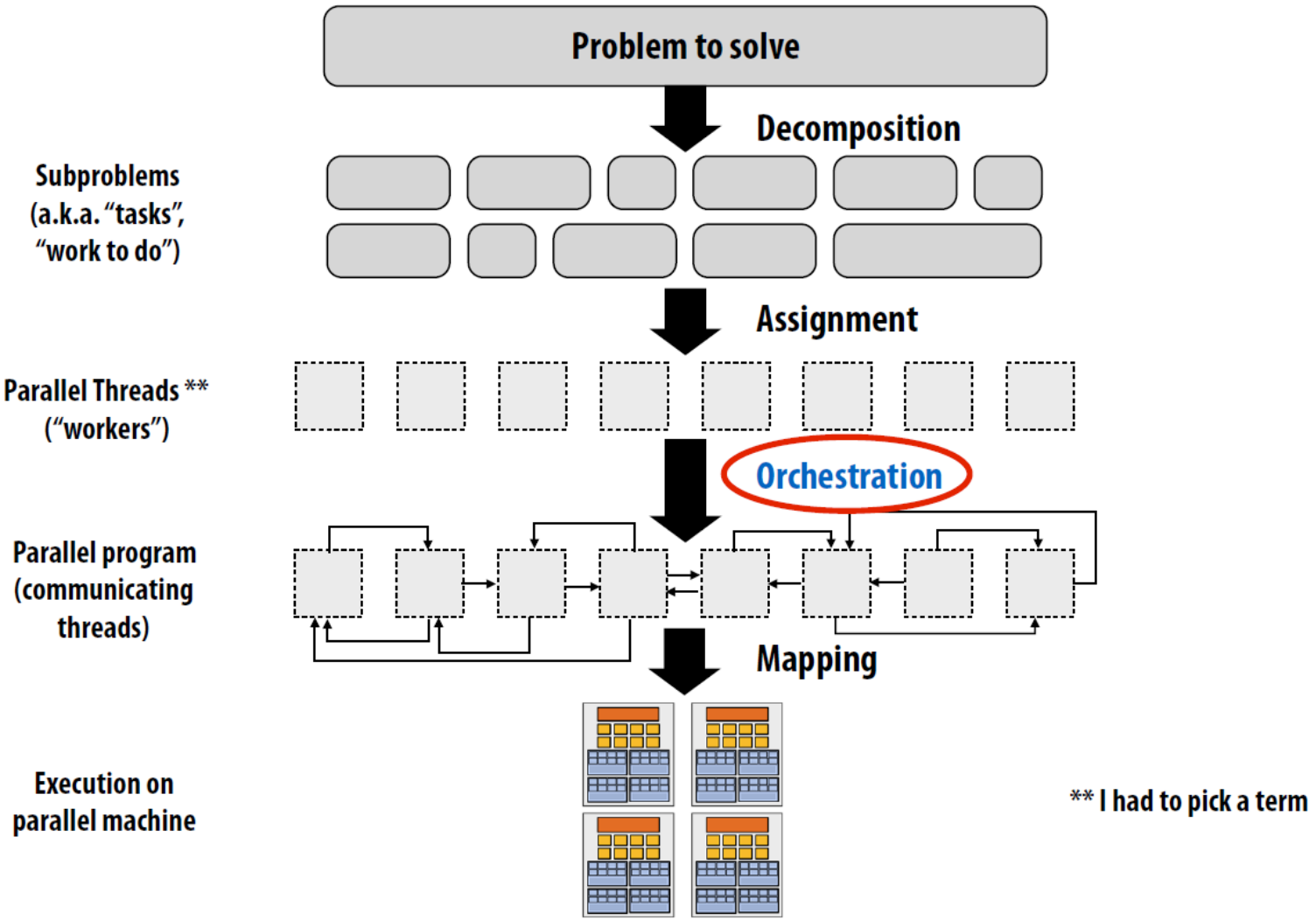
- Think of "tasks" as things to do
- Think of threads as "workers"

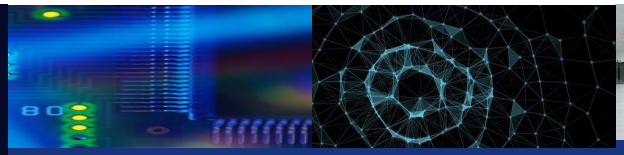
Goals: balance workload, reduce communication costs

Can be performed statically, or dynamically during execution

While programmer often responsible for decomposition, many languages/runtimes take responsibility for assignment.

Orchestration





Orchestration

Involves:

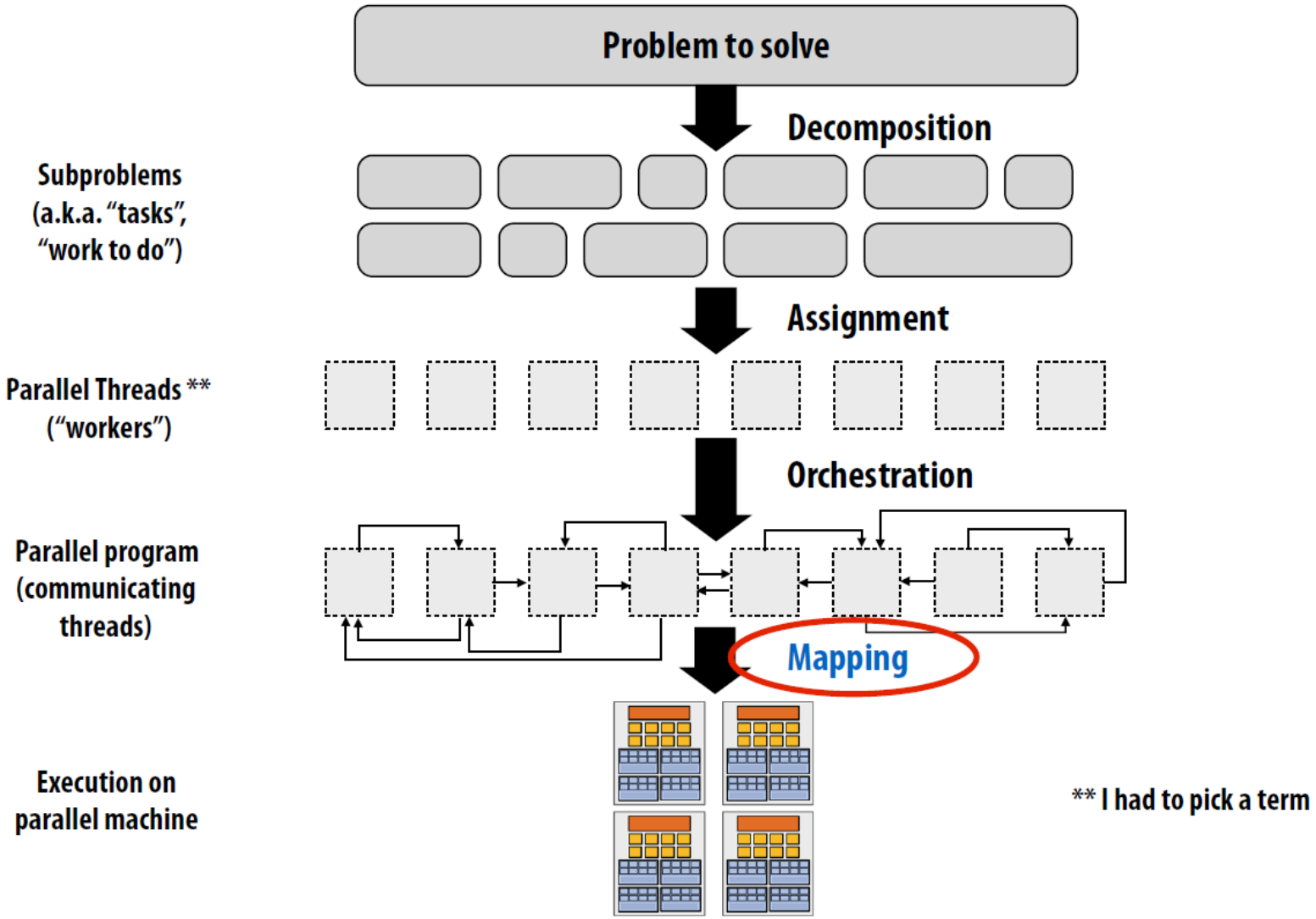
- Structuring communication
- Adding synchronization to preserve dependencies if necessary
- Organizing data structures in memory
- Scheduling tasks

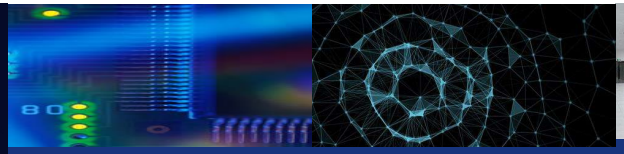
Goals: reduce costs of communication/sync, preserve locality of data reference, reduce overhead, etc.

Machine details impact many of these decisions

- If synchronization is expensive, might use it more sparsely

Orchestration





Mapping

Mapping "threads" ("workers") to hardware execution units

Example 1: mapping by the operating system

- e.g., map pthread to HW execution context on a CPU core

Example 2: mapping by the compiler

- Map ISPC program instances to vector instruction lanes

Example 3: mapping by the hardware

- Map CUDA thread blocks to GPU cores (future lecture)

Some interesting mapping decisions:

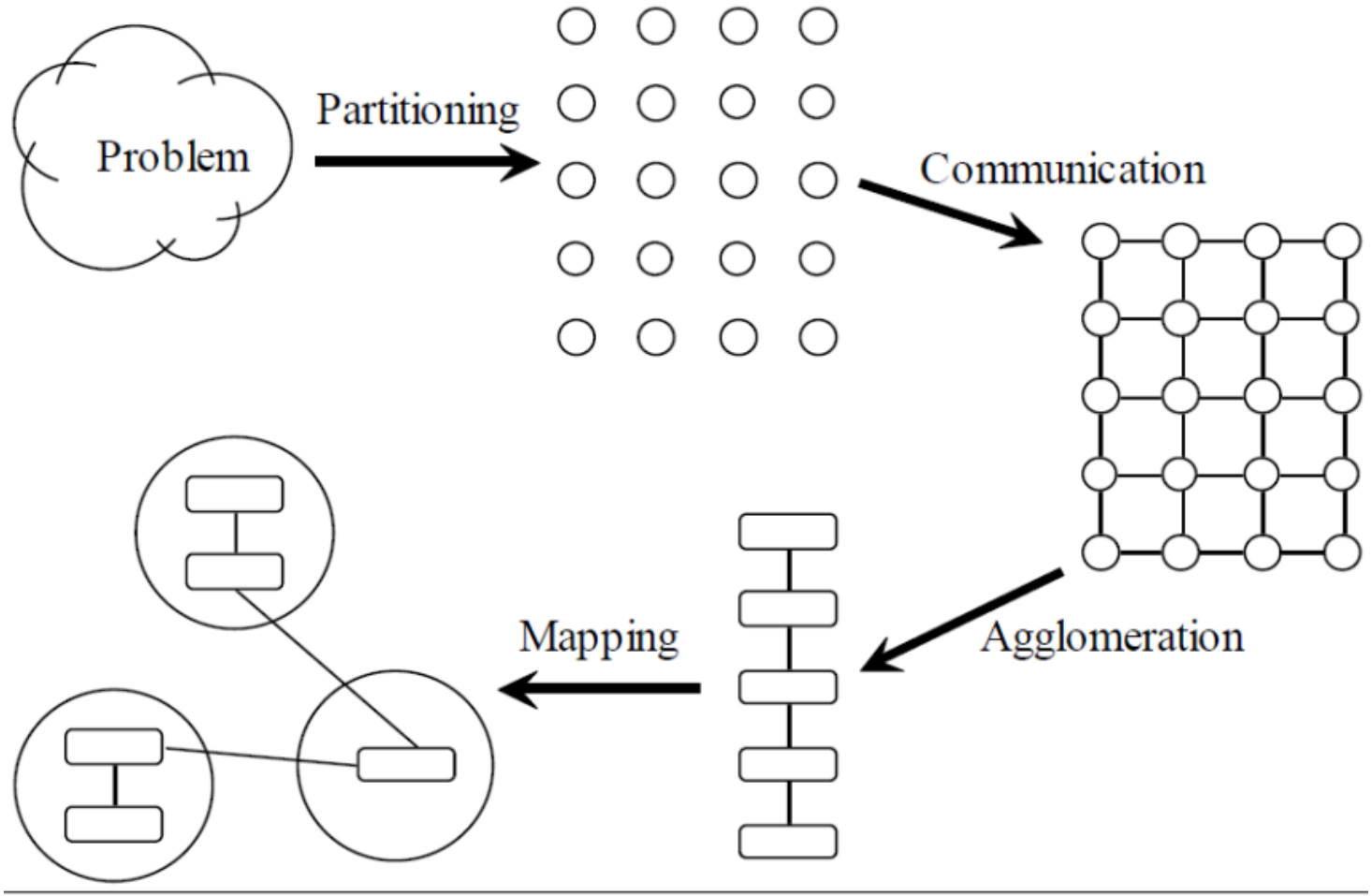
- Place related threads (cooperating threads) on the same processor

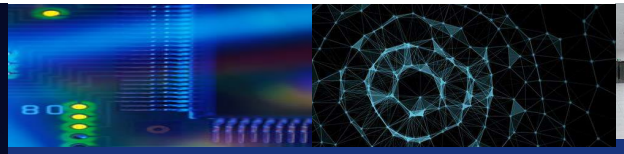
- Place unrelated threads on the same processor (one might be bandwidth limited and another might be compute limited) to use machine more efficiently

Foster's Design Methodology

- **Partitioning**
- **Communication**
- **Agglomeration (归并、组合)**
- **Mapping**

Foster's Design Methodology





Partitioning (分解)

- 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 Partitioning （按域 / 数据分解）

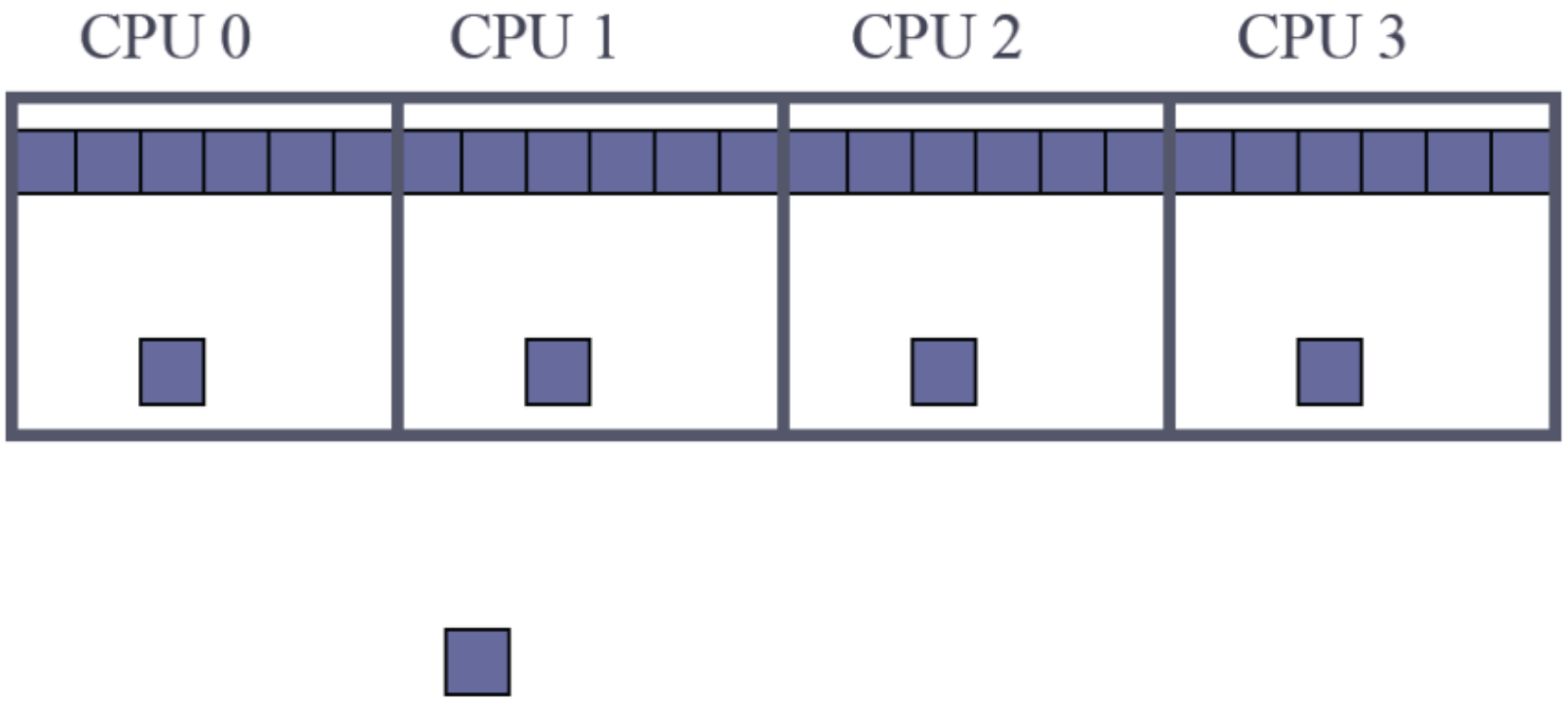
- First, decide how data elements should be divided among processors
- Second, decide which tasks each processor should be doing
- Example: find maximum element in a vector

Domain Partitioning Example

Find the largest element of an array

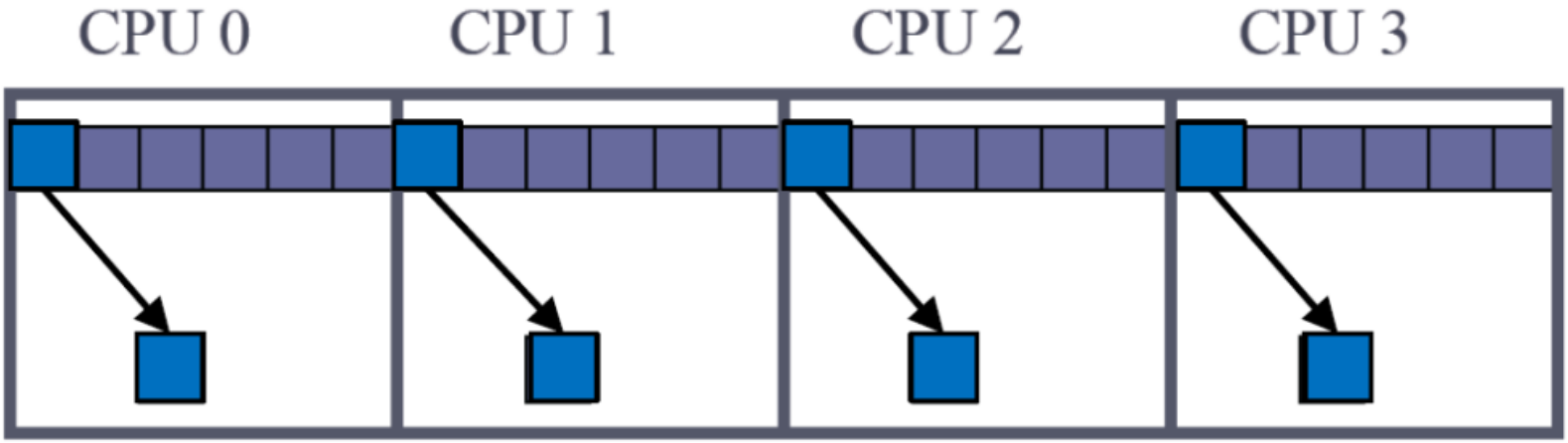
Domain Partitioning Example

Find the largest element of an array



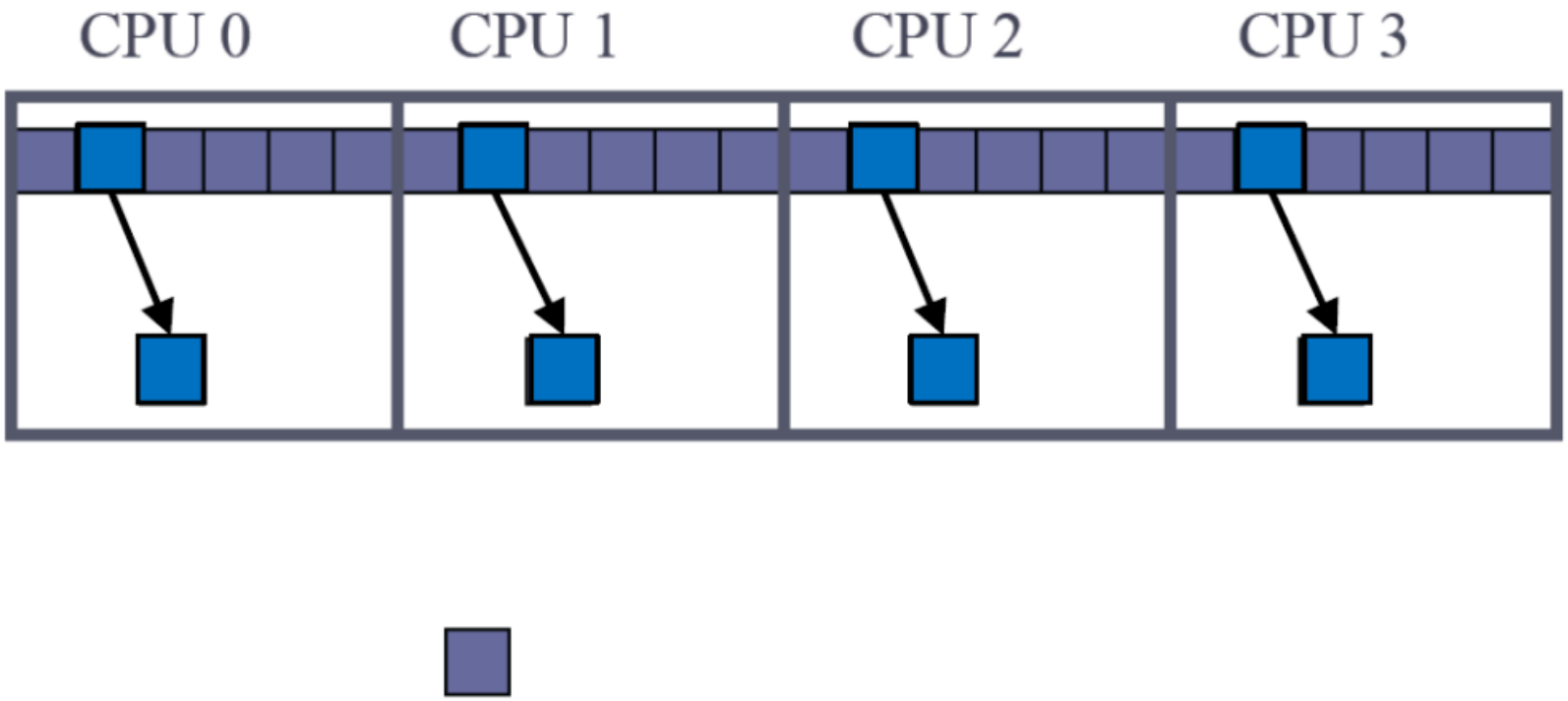
Domain Partitioning Example

Find the largest element of an array



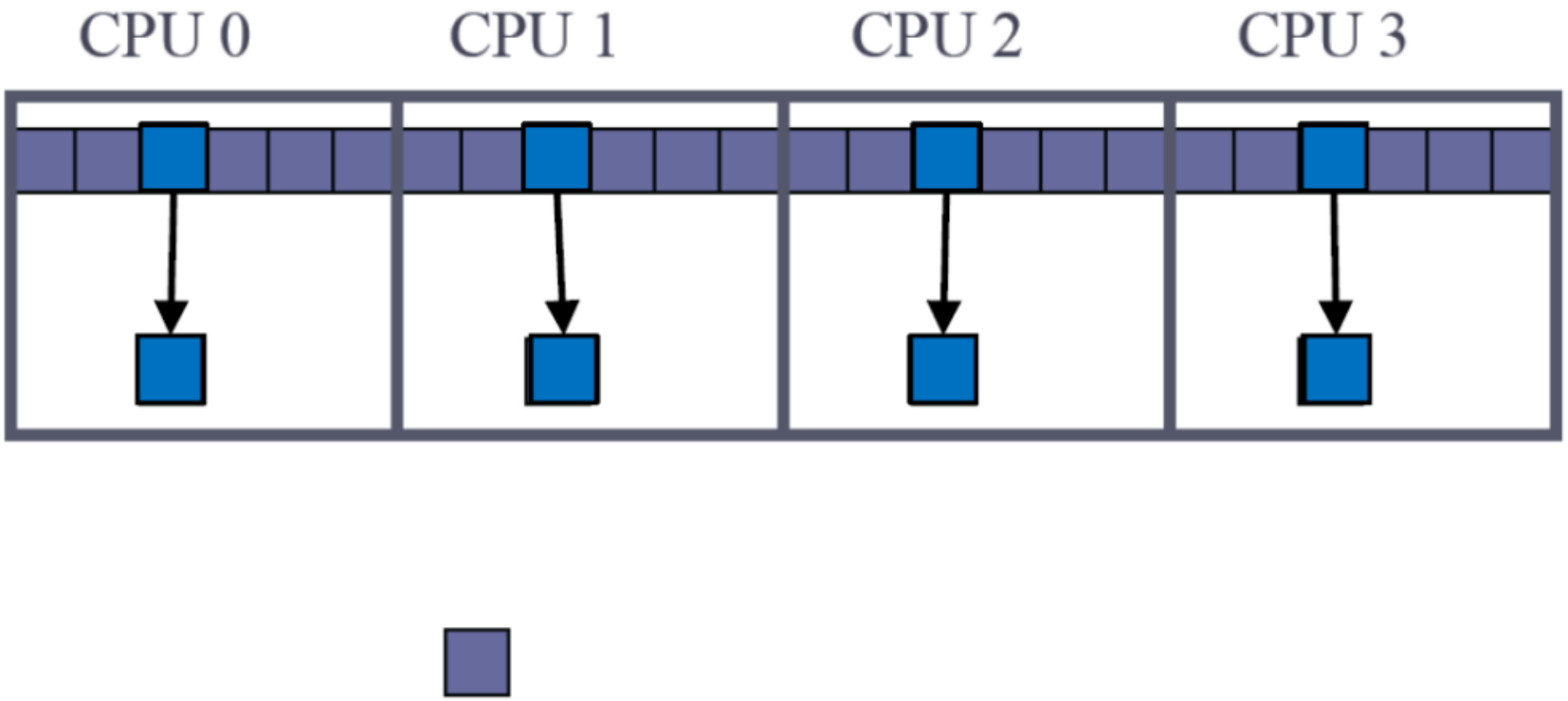
Domain Partitioning Example

Find the largest element of an array



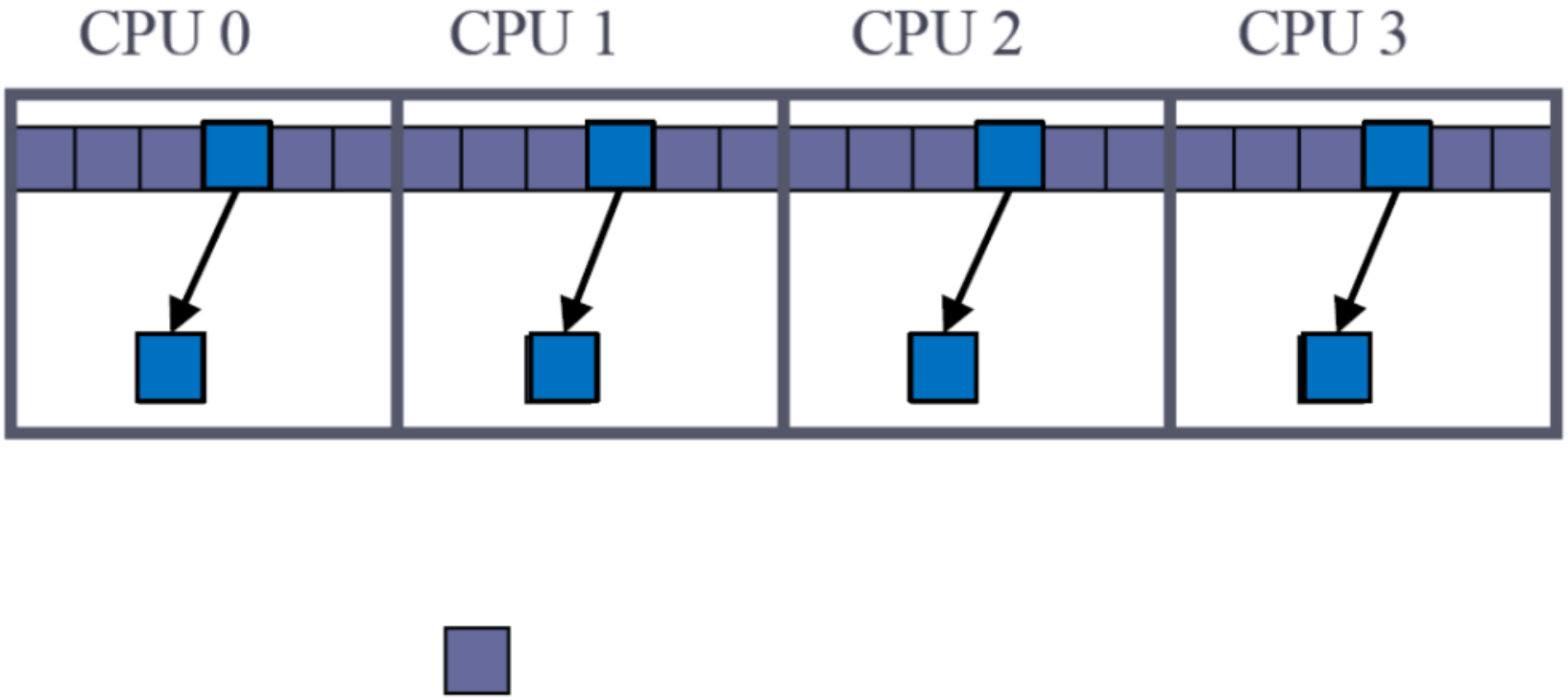
Domain Partitioning Example

Find the largest element of an array



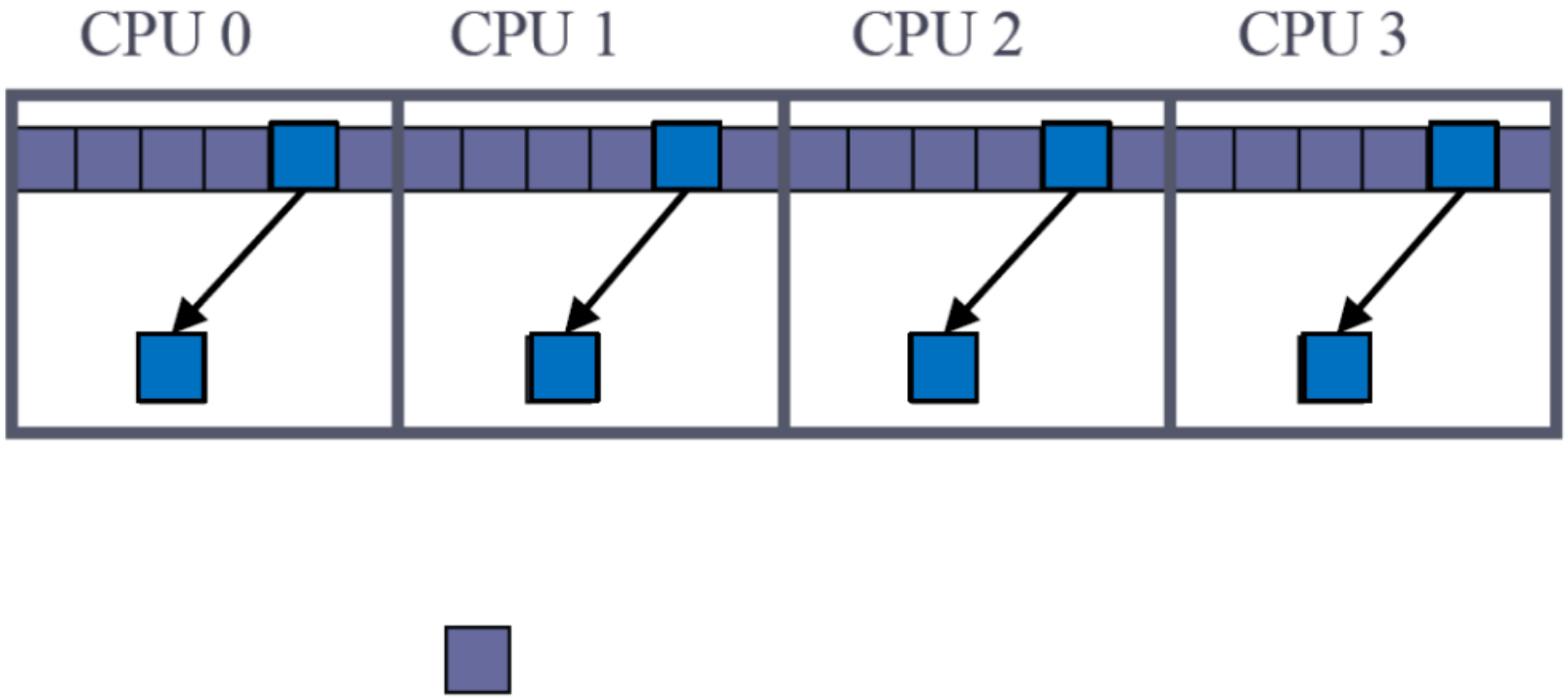
Domain Partitioning Example

Find the largest element of an array



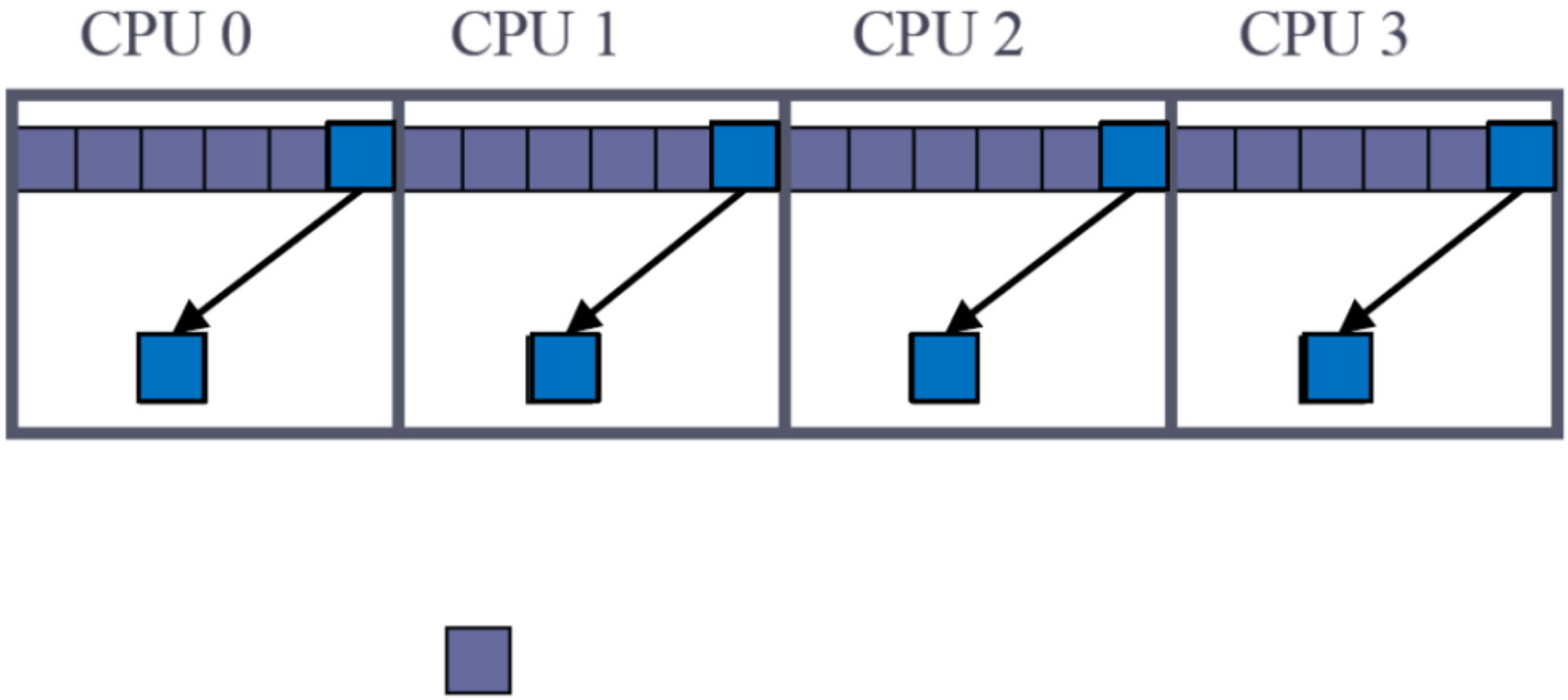
Domain Partitioning Example

Find the largest element of an array



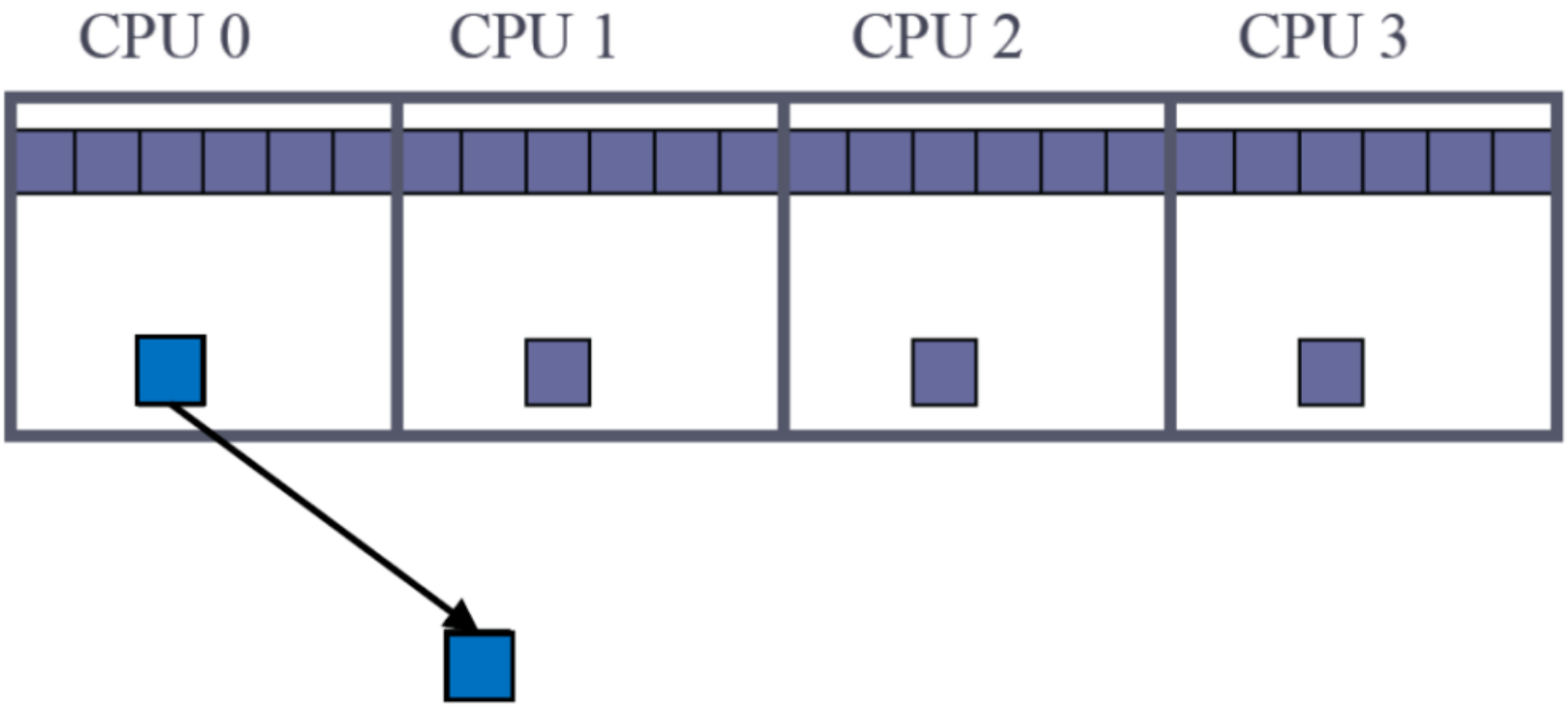
Domain Partitioning Example

Find the largest element of an array



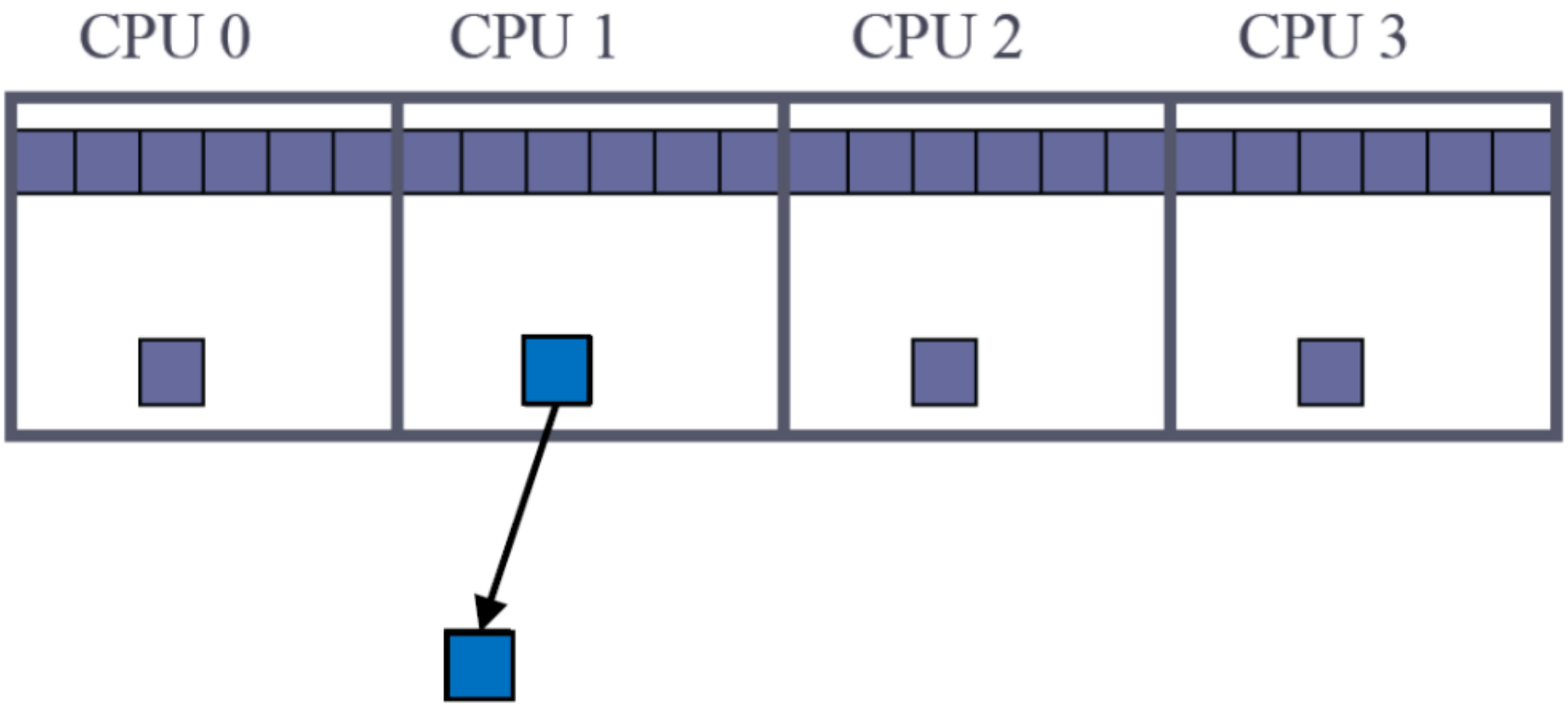
Domain Partitioning Example

Find the largest element of an array



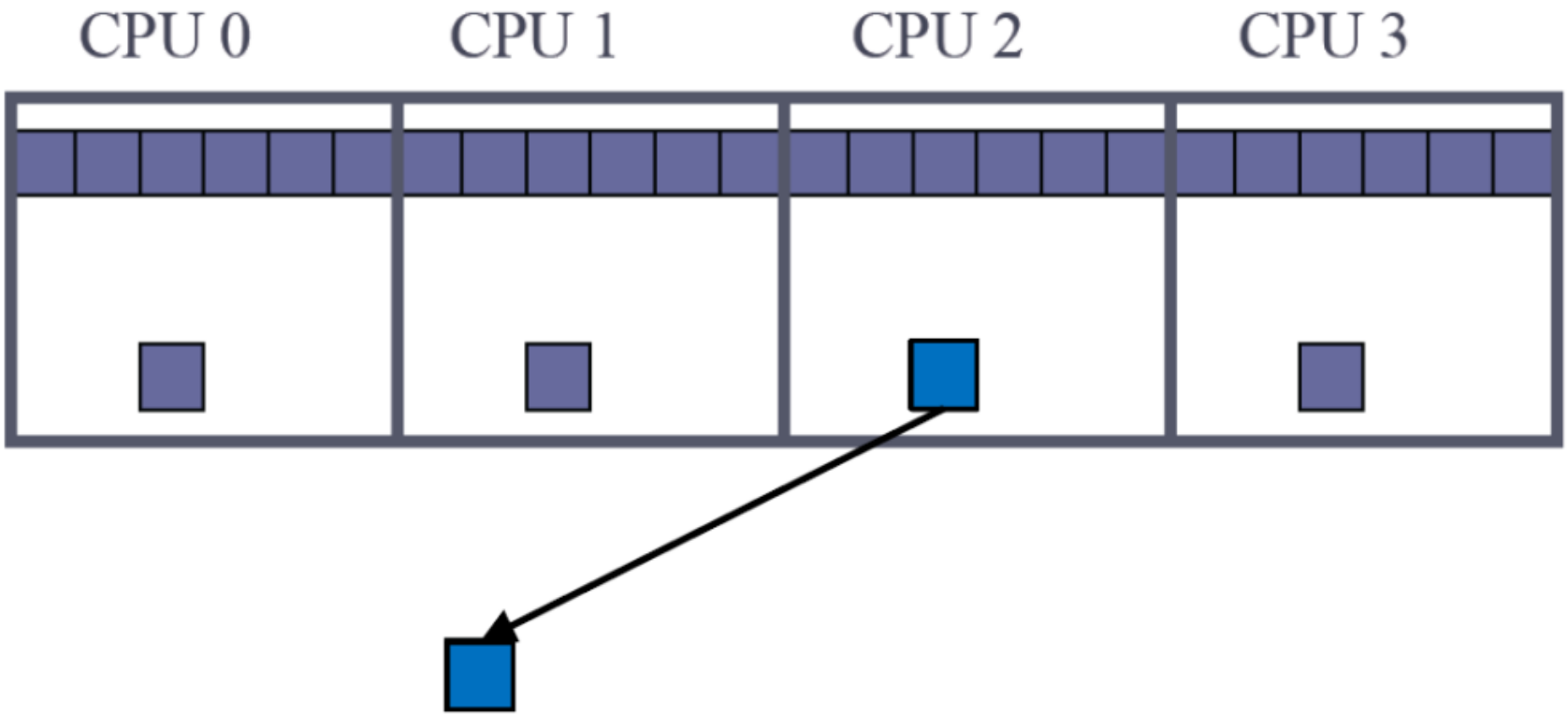
Domain Partitioning Example

Find the largest element of an array



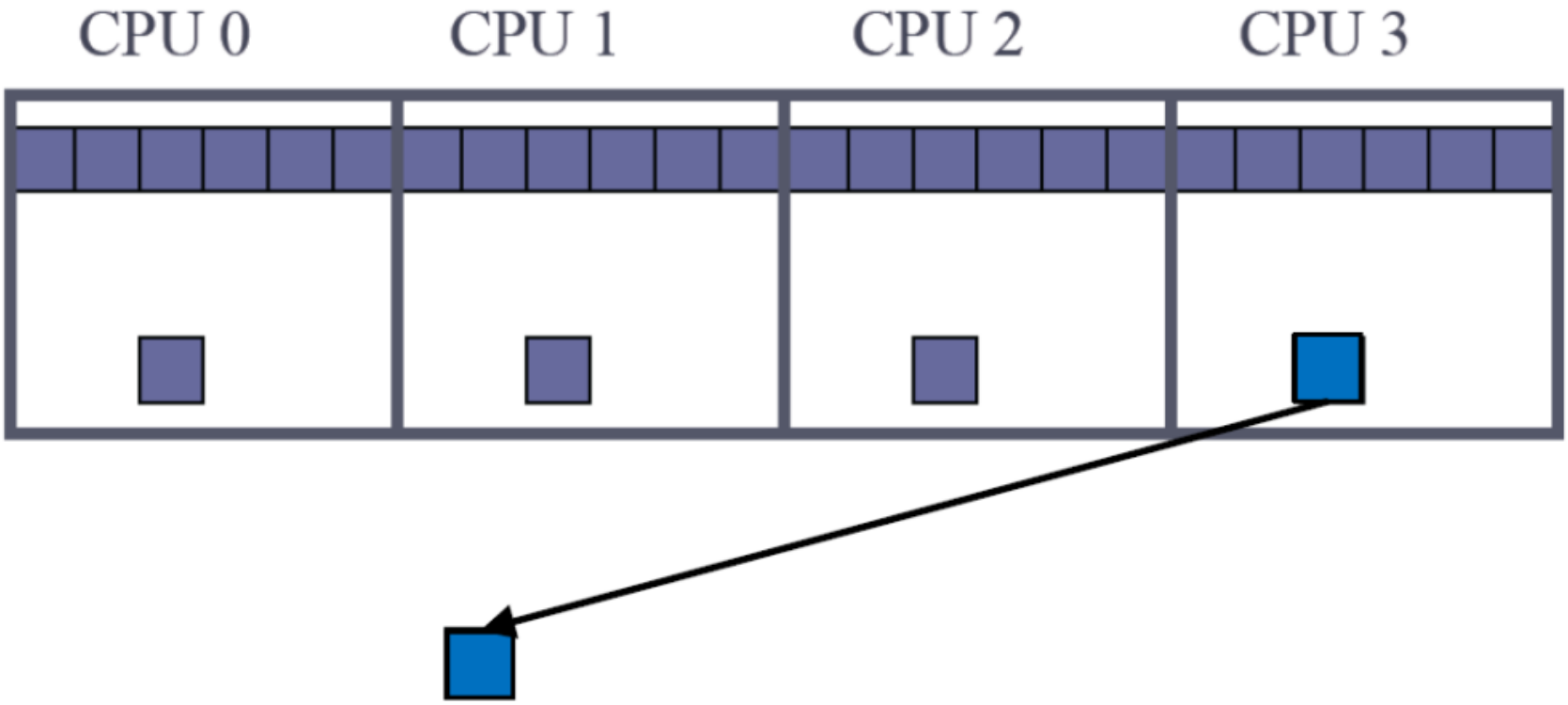
Domain Partitioning Example

Find the largest element of an array



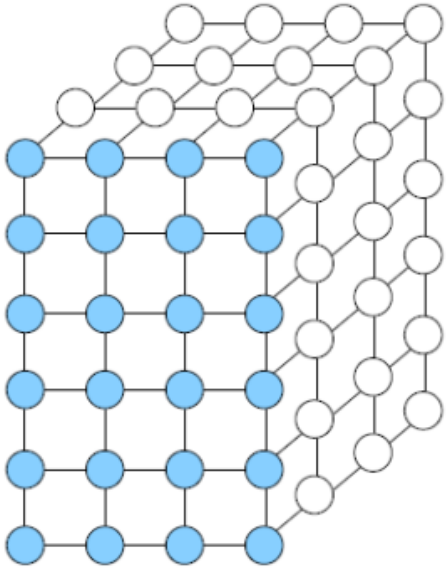
Domain Partitioning Example

Find the largest element of an array

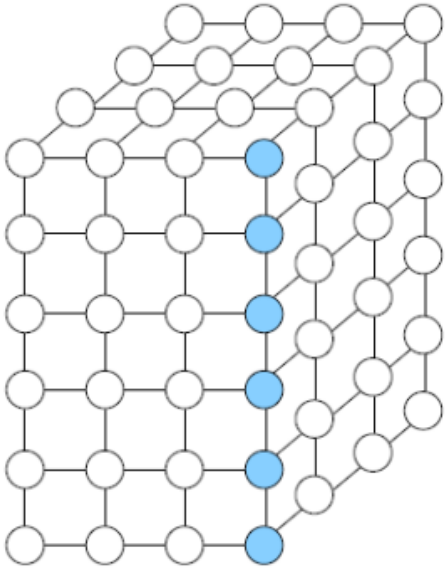


Domain Partitioning: Another Example

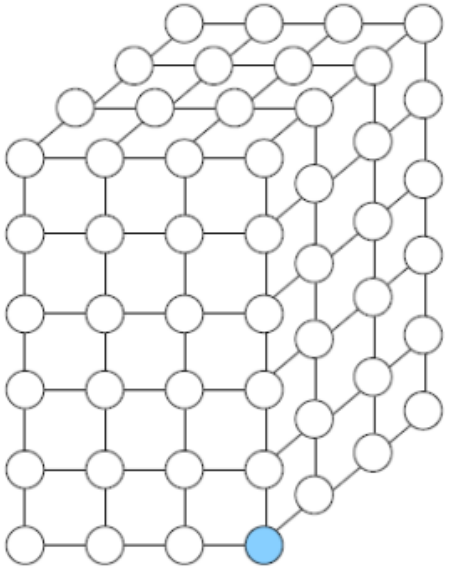
For example, if we have a 3D model of an object, represented as a collection of 3D points on the surface of the object, and we want to rotate that object in space, then in theory we can apply the same operation to each point in parallel, and a domain decomposition would assign a primitive task（原子任务）to each point.



1D Decomposition



2D Decomposition

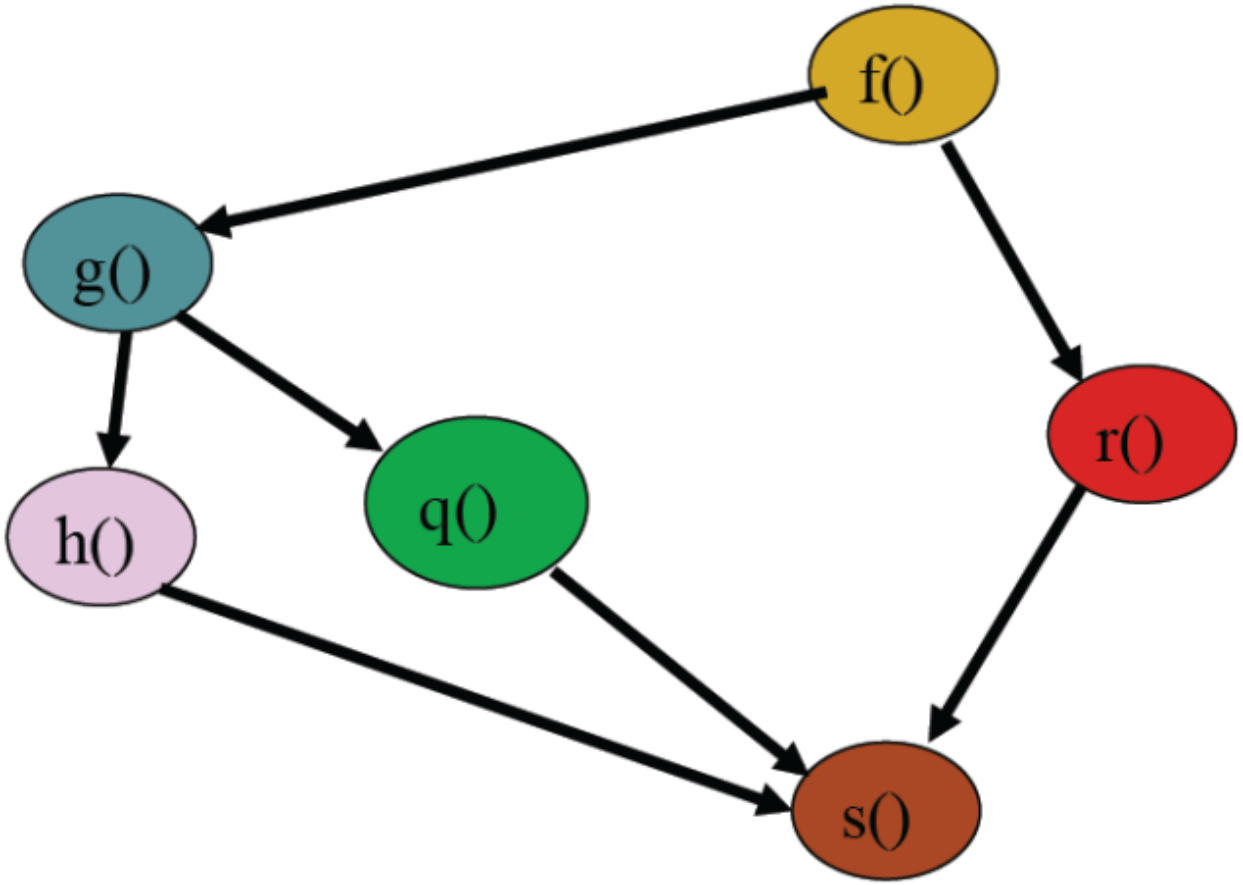


3D Decomposition

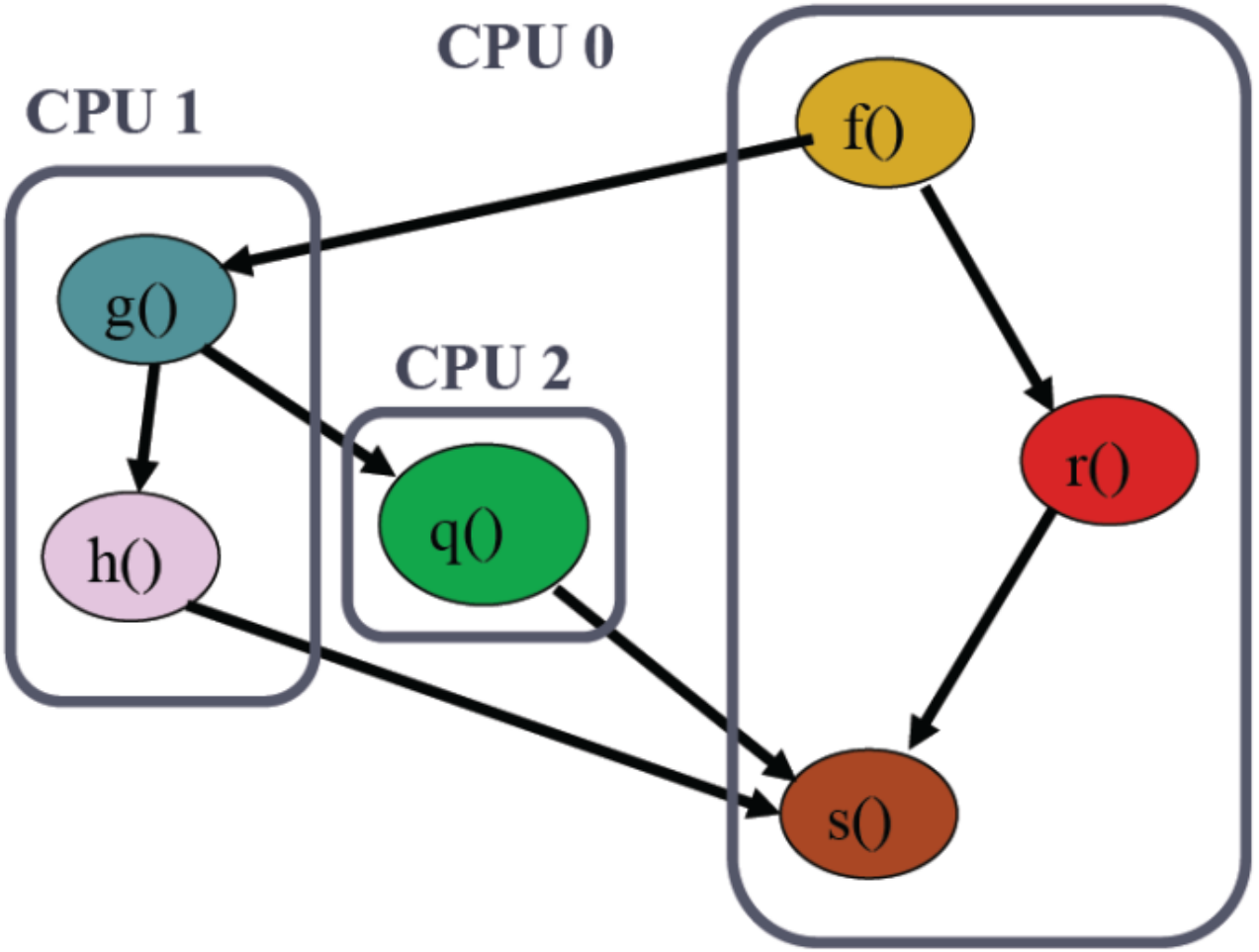
Functional (Task) Decomposition

- **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**

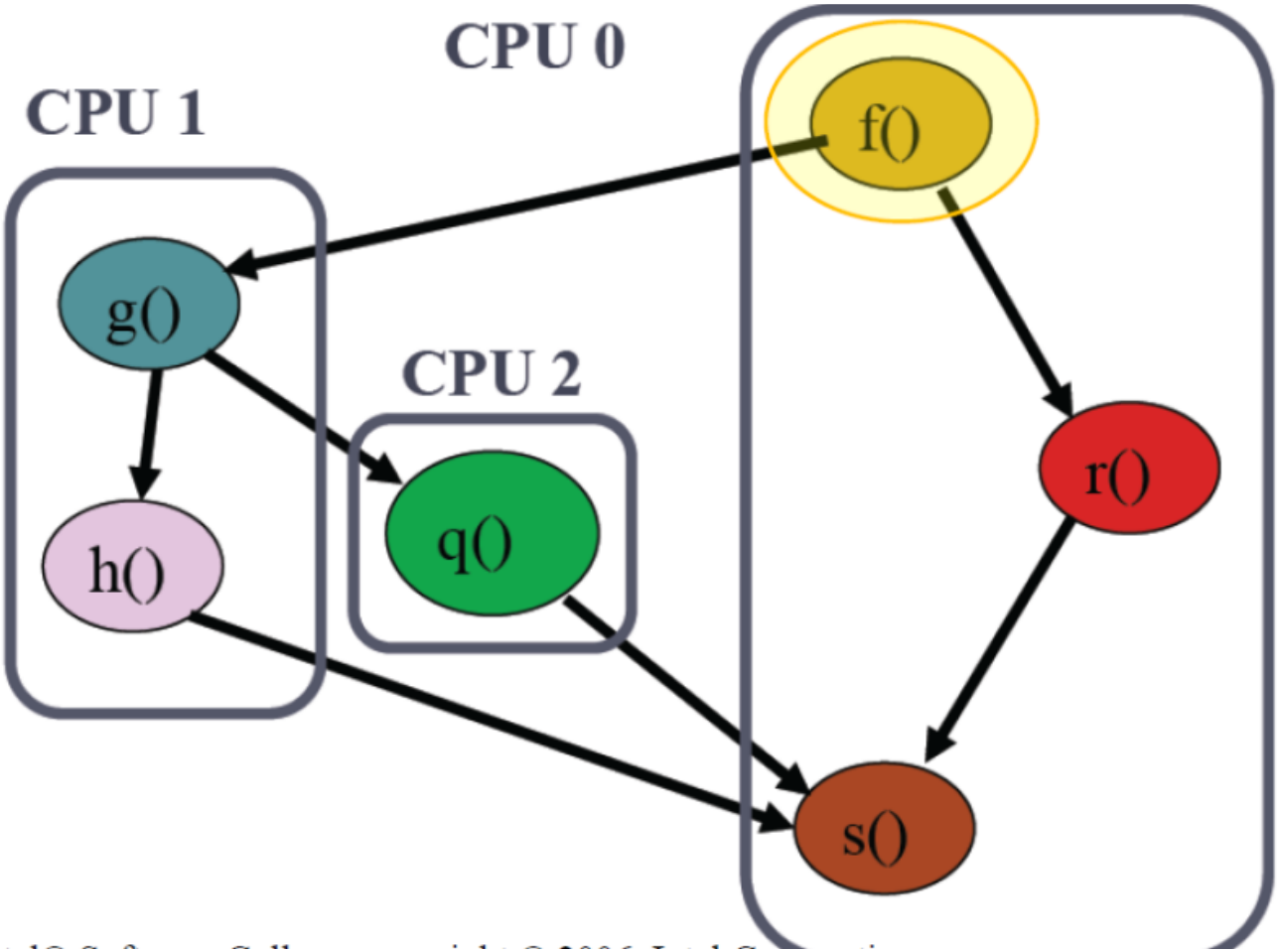
Functional (Task) Decomposition Example



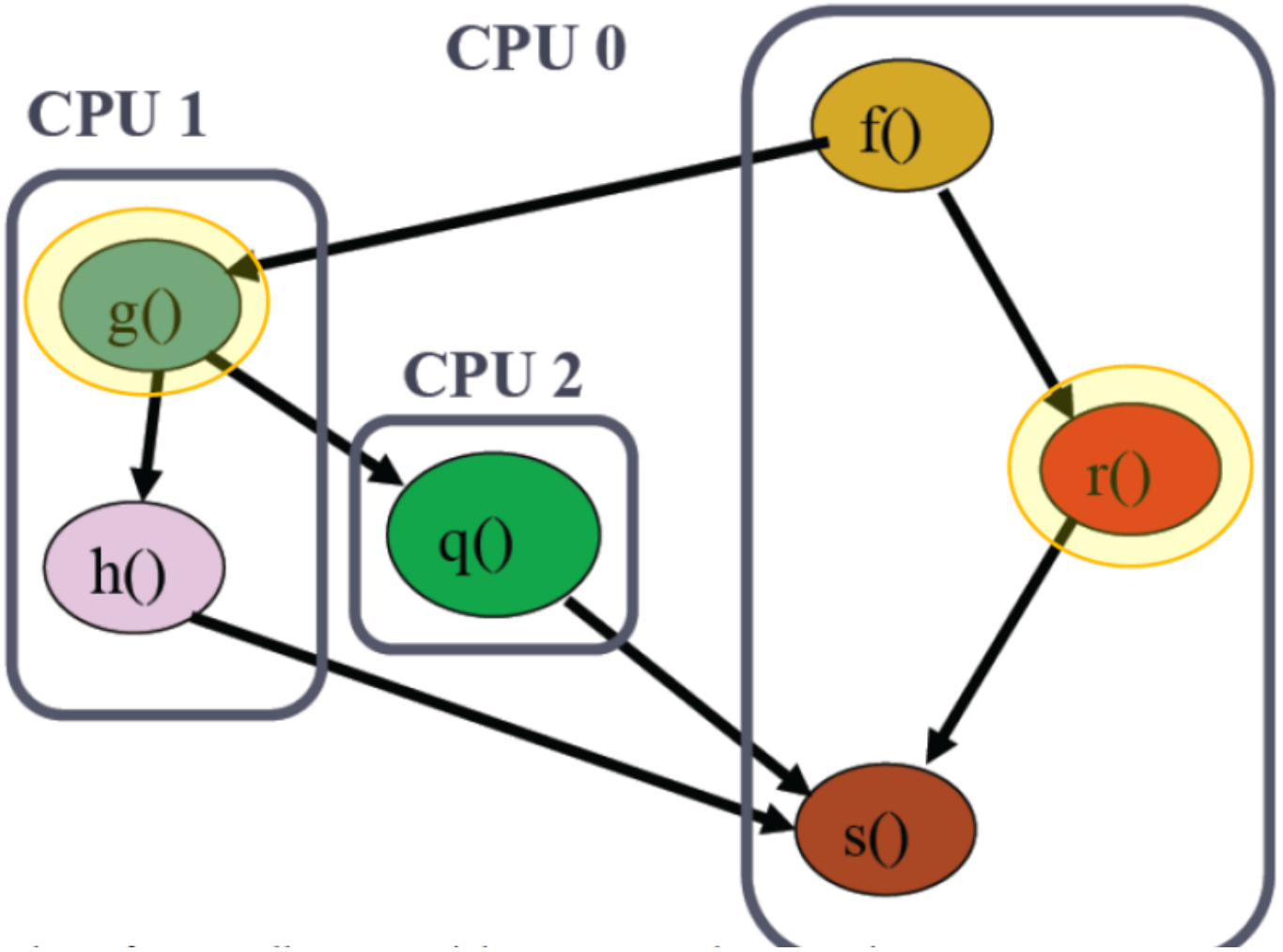
Functional (Task) Decomposition Example



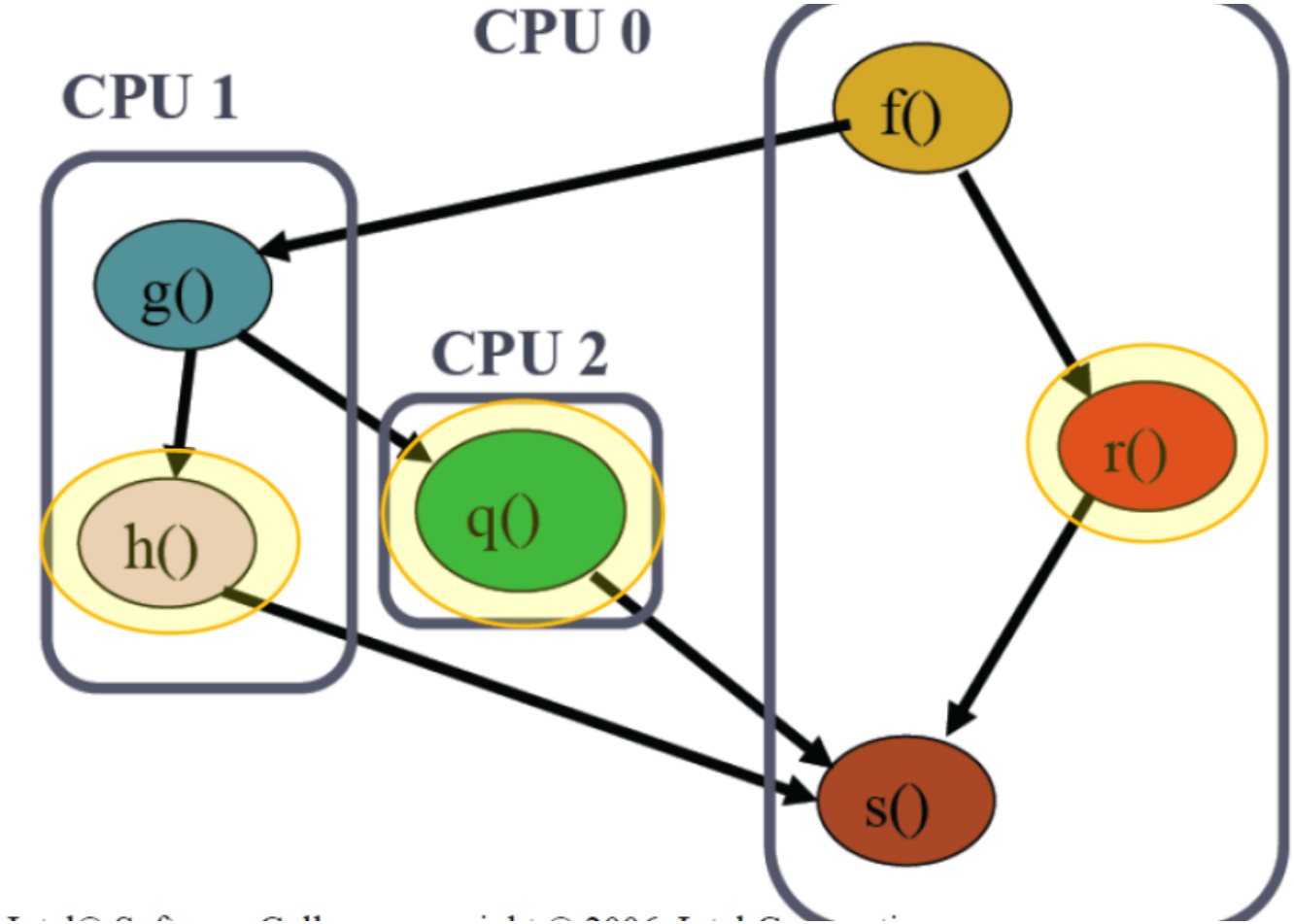
Functional (Task) Decomposition Example



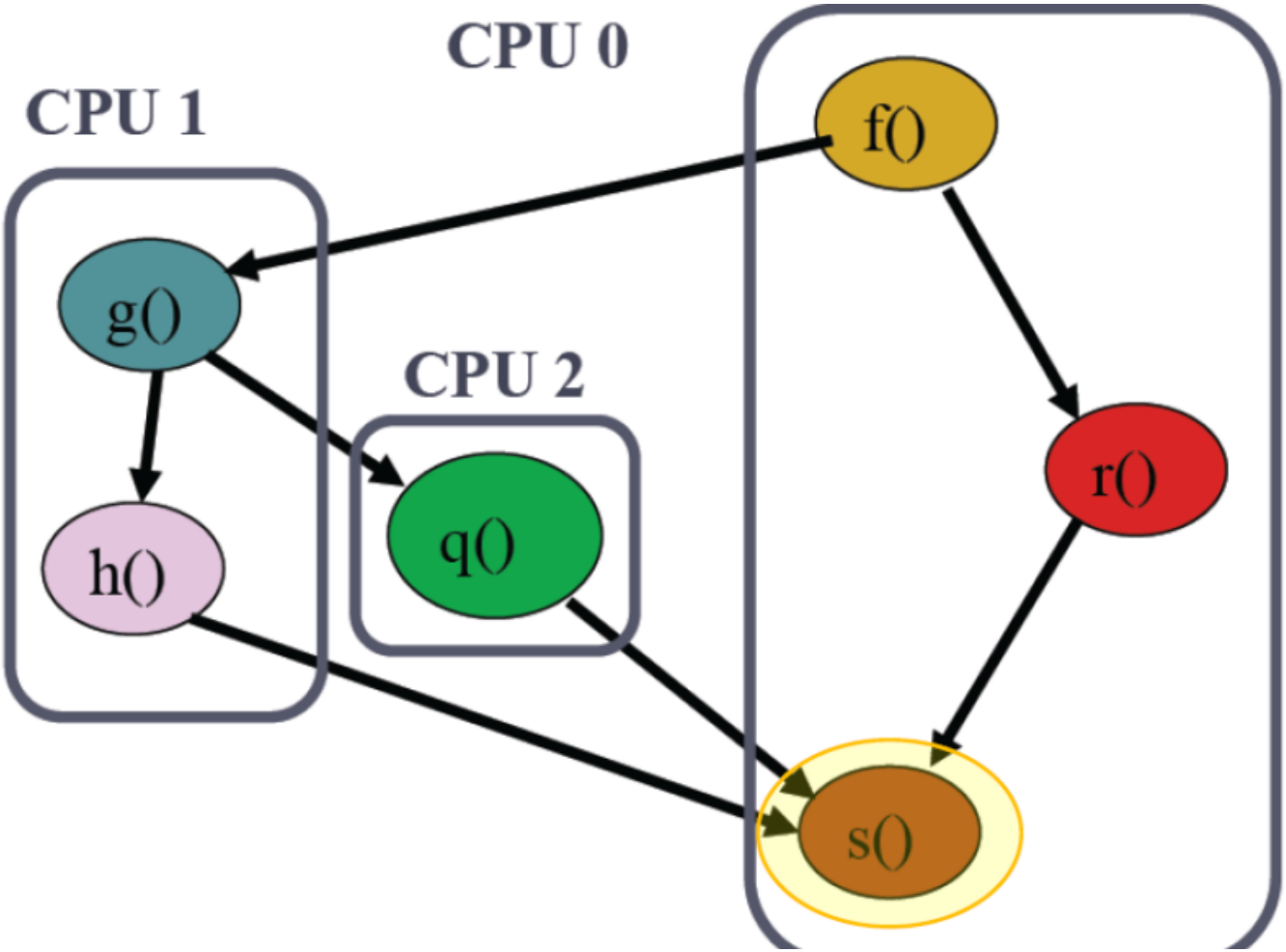
Functional (Task) Decomposition Example



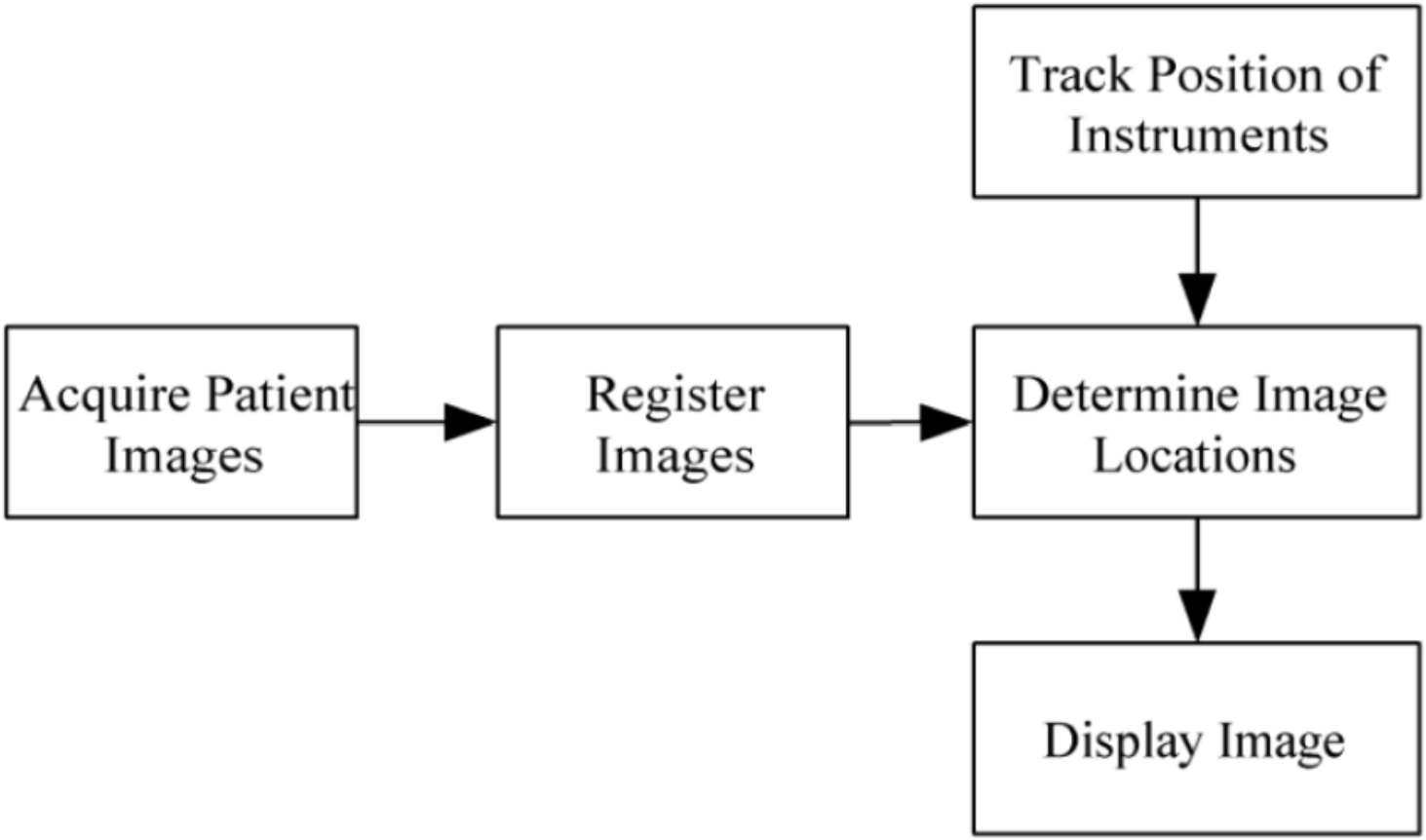
Functional (Task) Decomposition Example



Functional (Task) Decomposition Example



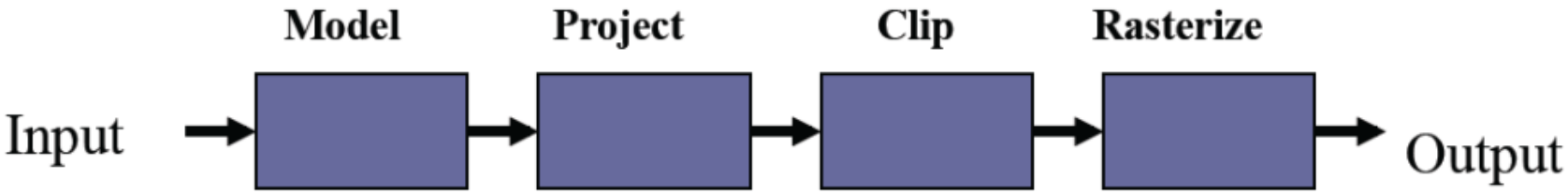
Functional Decomposition: Another Example



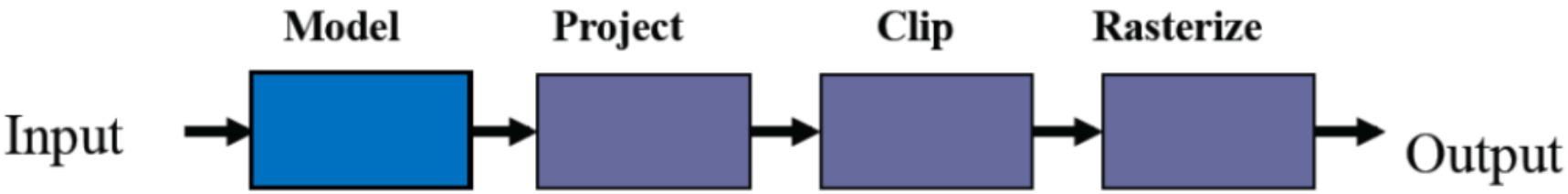
Telemedicine

Pipelining

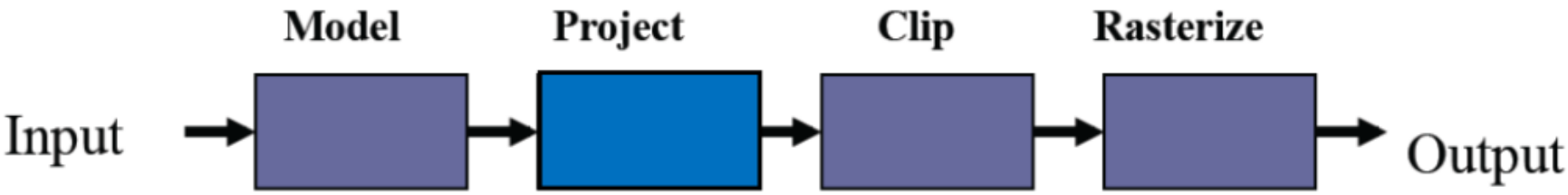
- Special kind of task decomposition
- "Assembly line" parallelism
- Example: 3D rendering in computer graphics



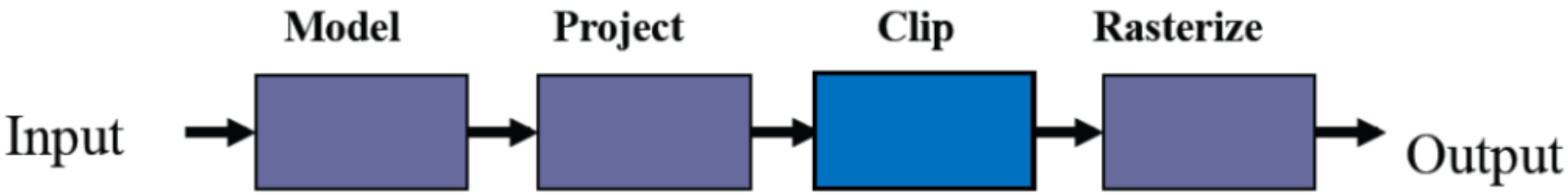
Processing One Data Set (Step 1)



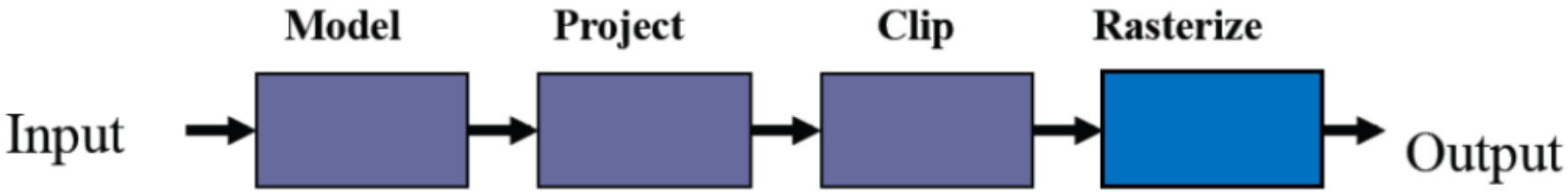
Processing One Data Set (Step 2)



Processing One Data Set (Step 3)

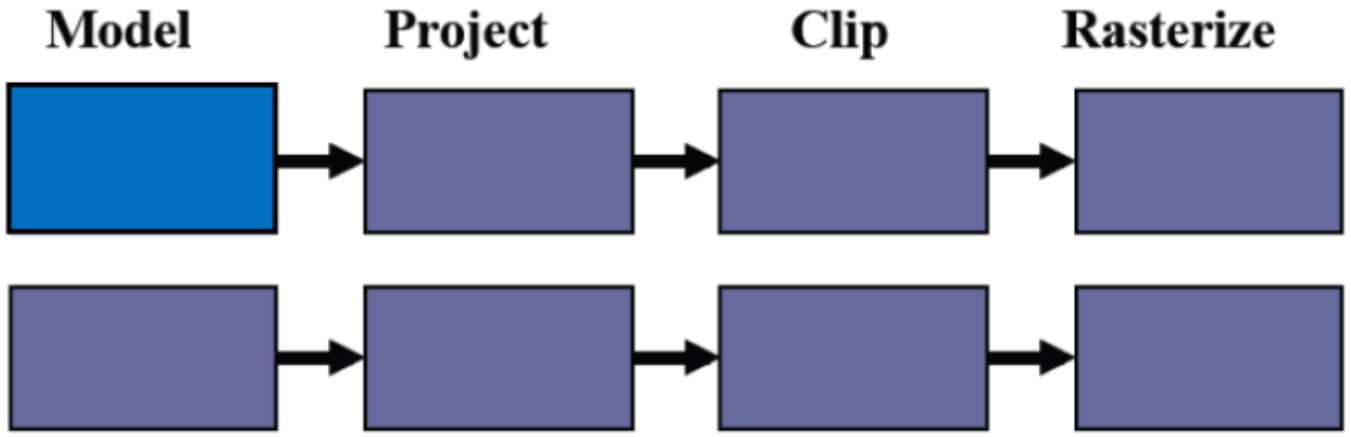


Processing One Data Set (Step 4)

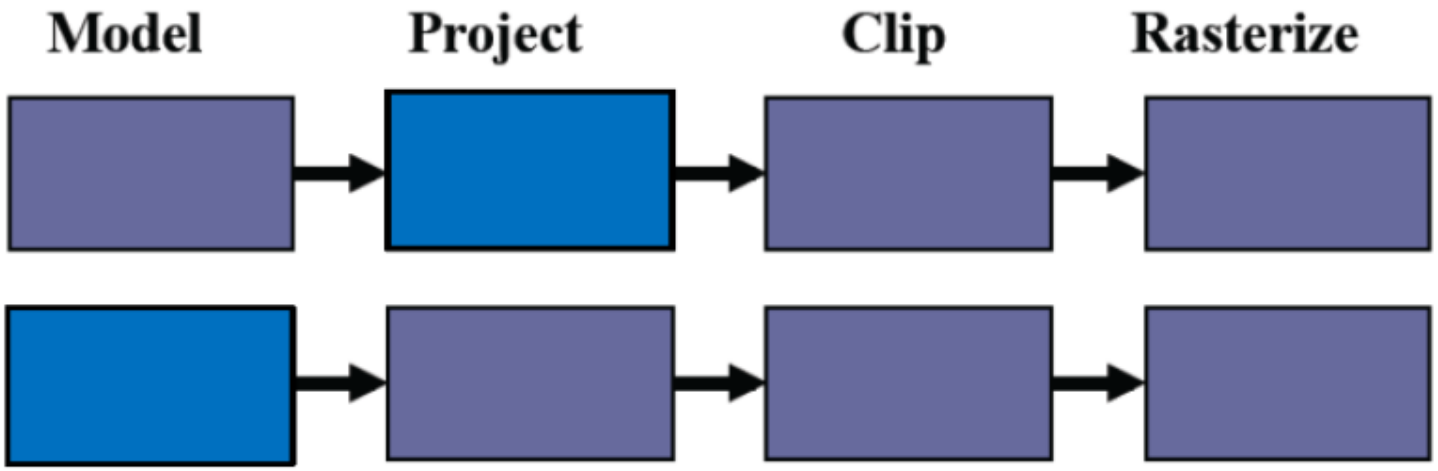


The pipeline processes 1 data set in 4 steps

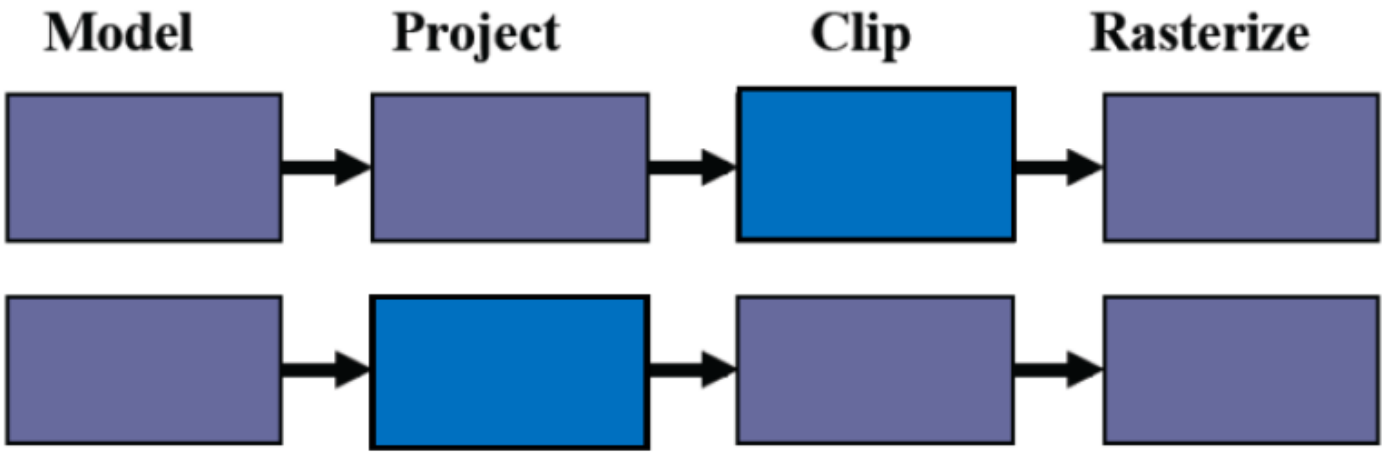
Processing Two Data Set (Step 1)



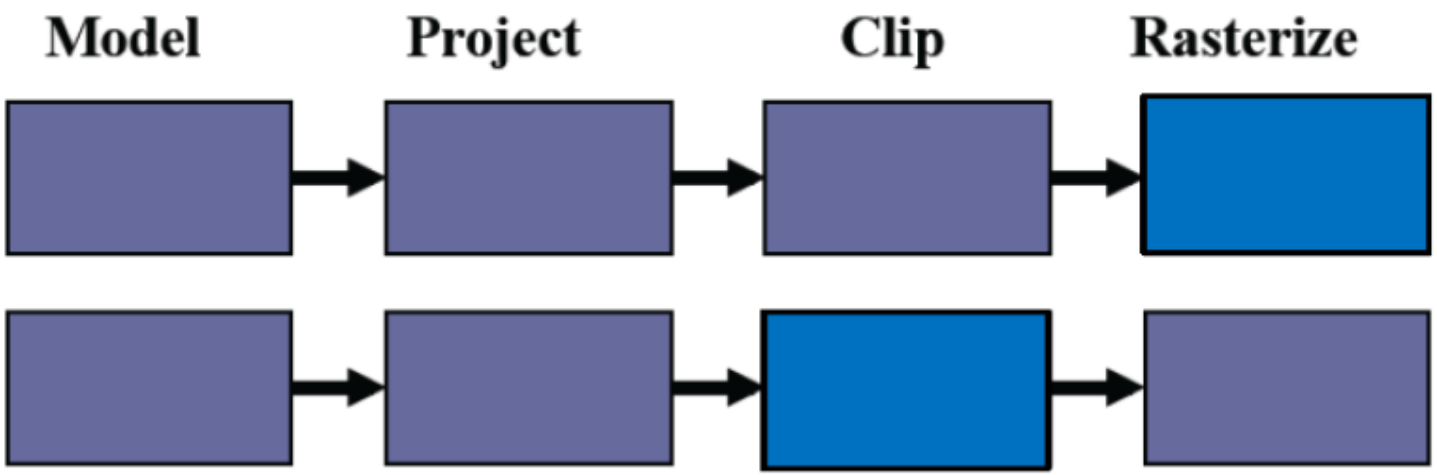
Processing Two Data Set (Step 2)



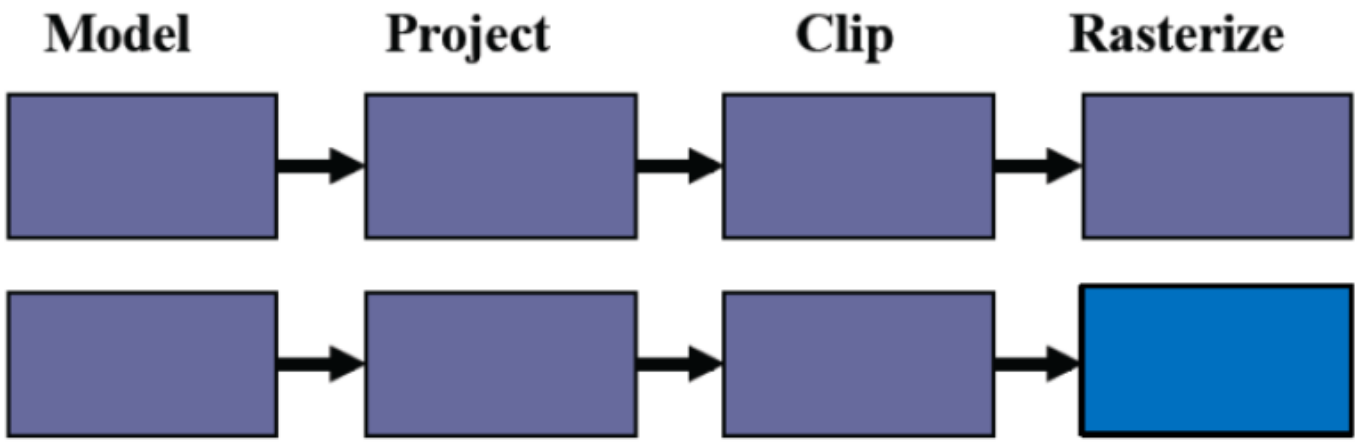
Processing Two Data Set (Step 3)



Processing Two Data Set (Step 4)

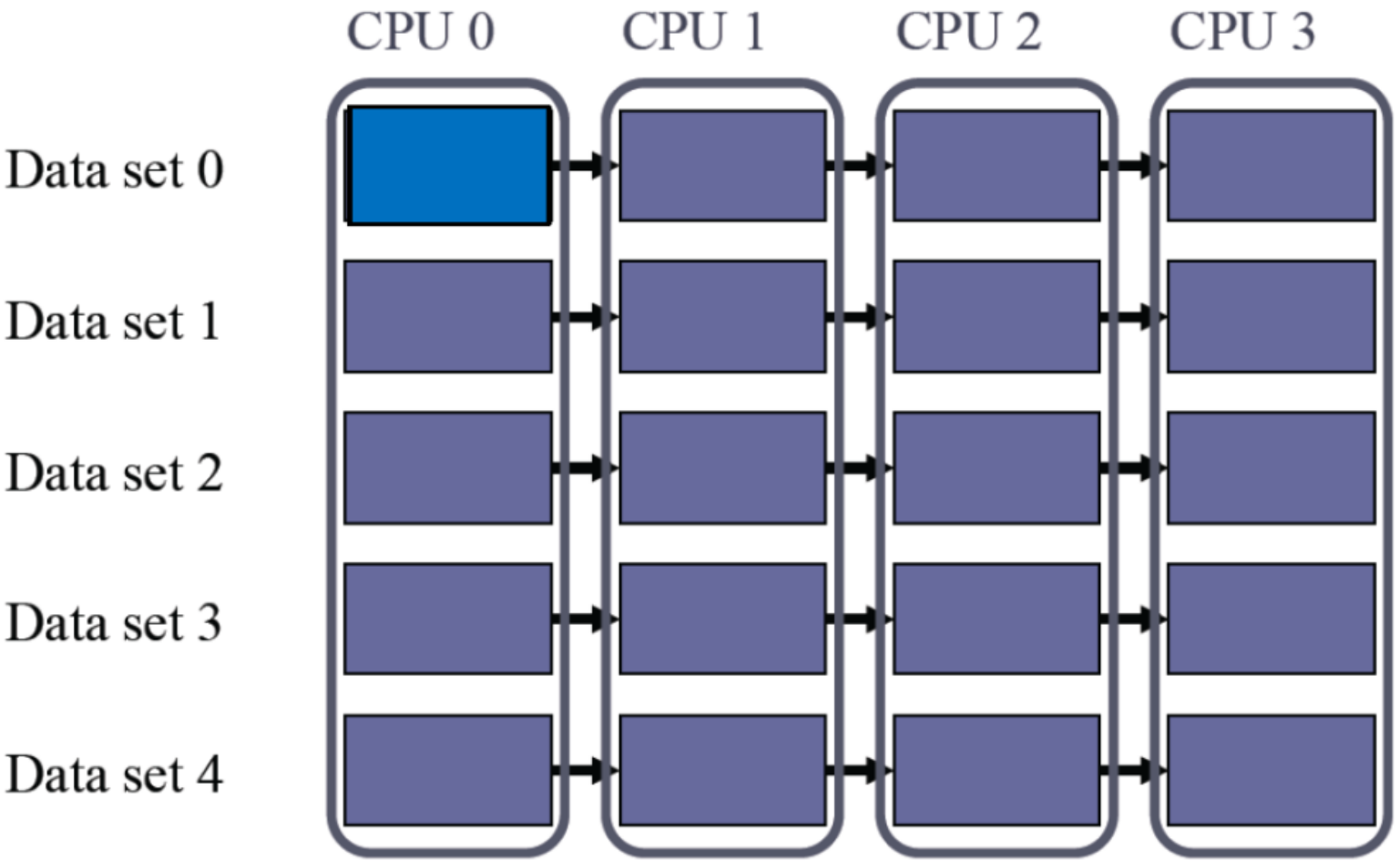


Processing Two Data Set (Step 5)

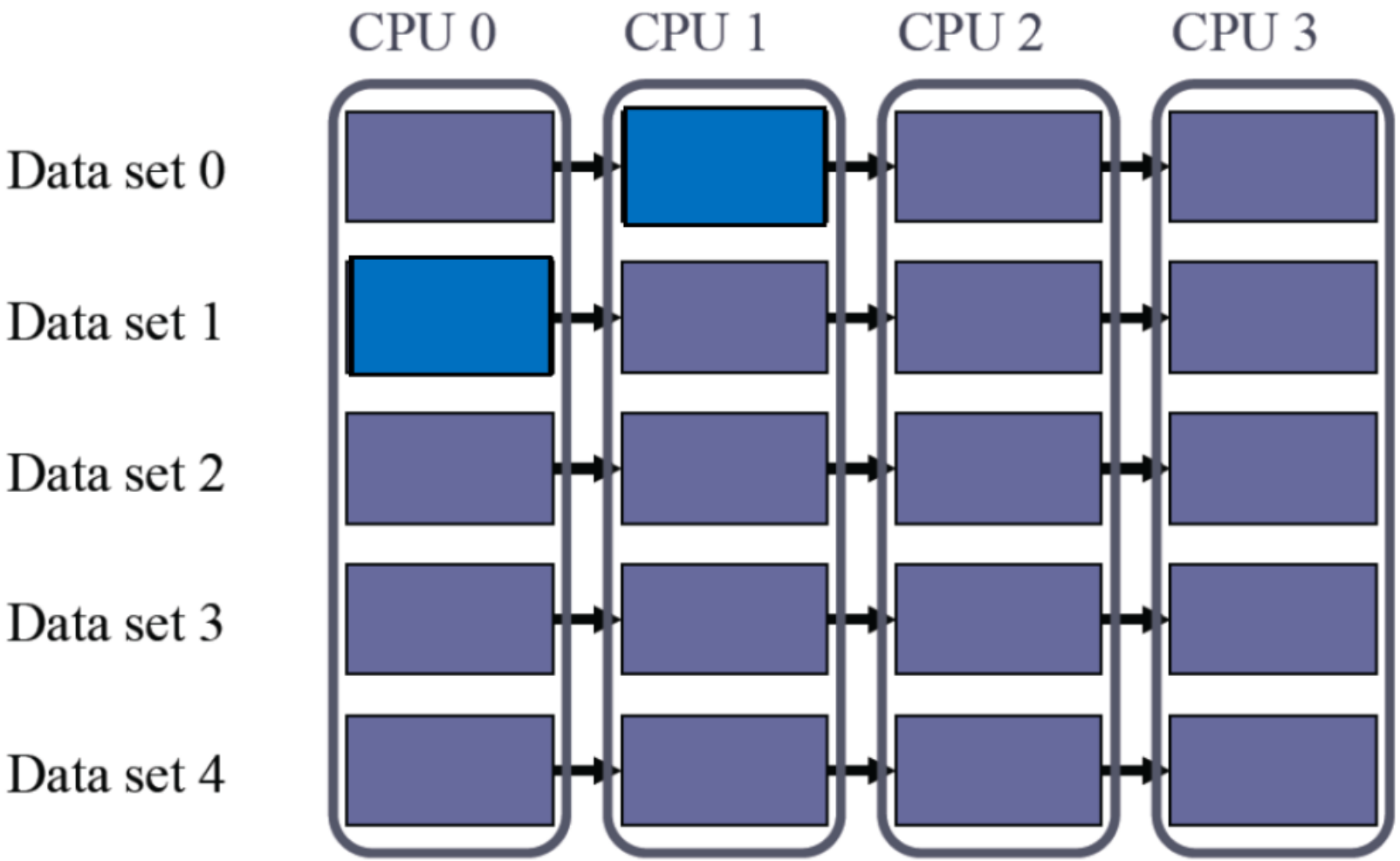


The pipeline processes 2 data sets in 5 steps

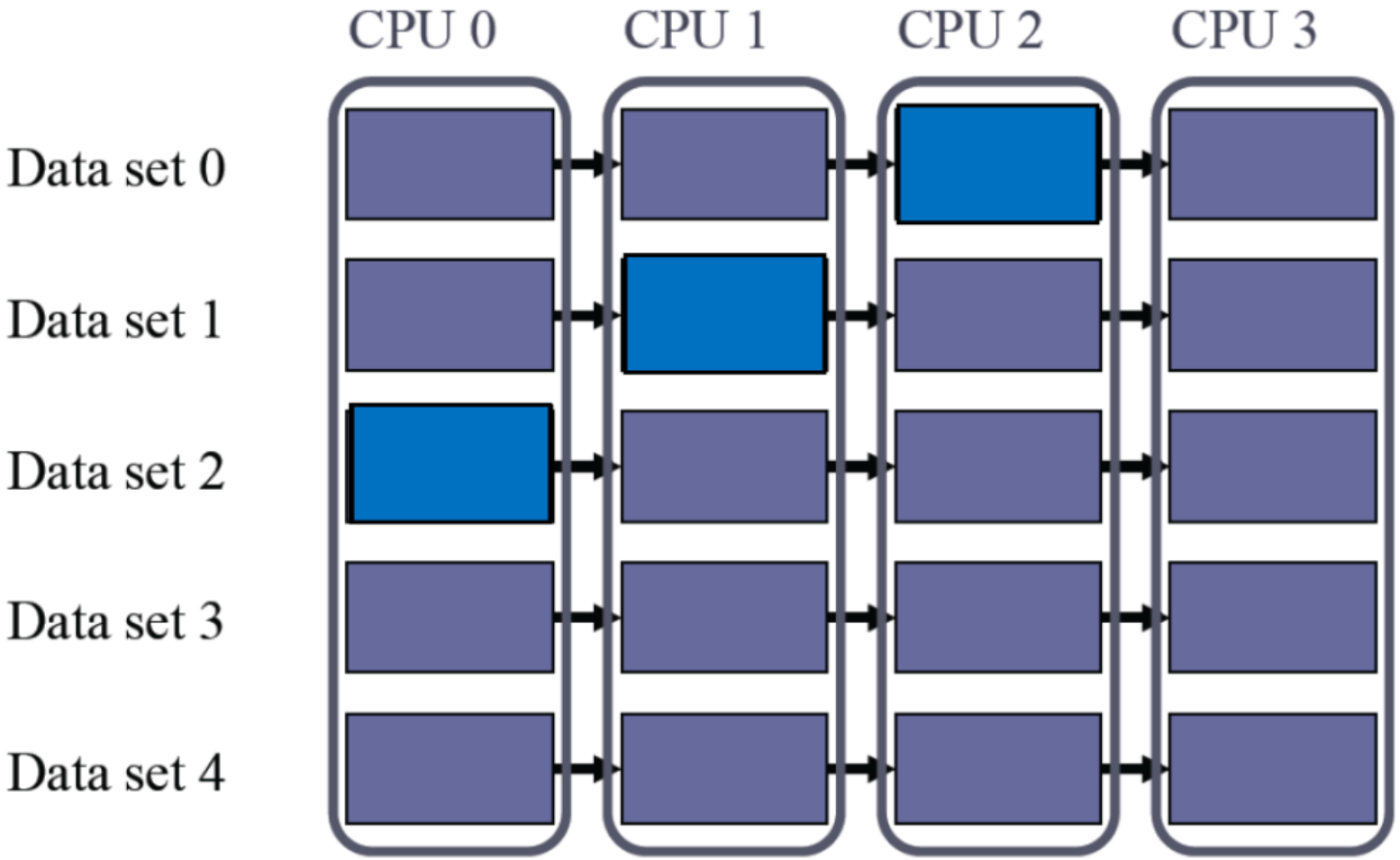
Pipelining Five Data Sets (Step 1)



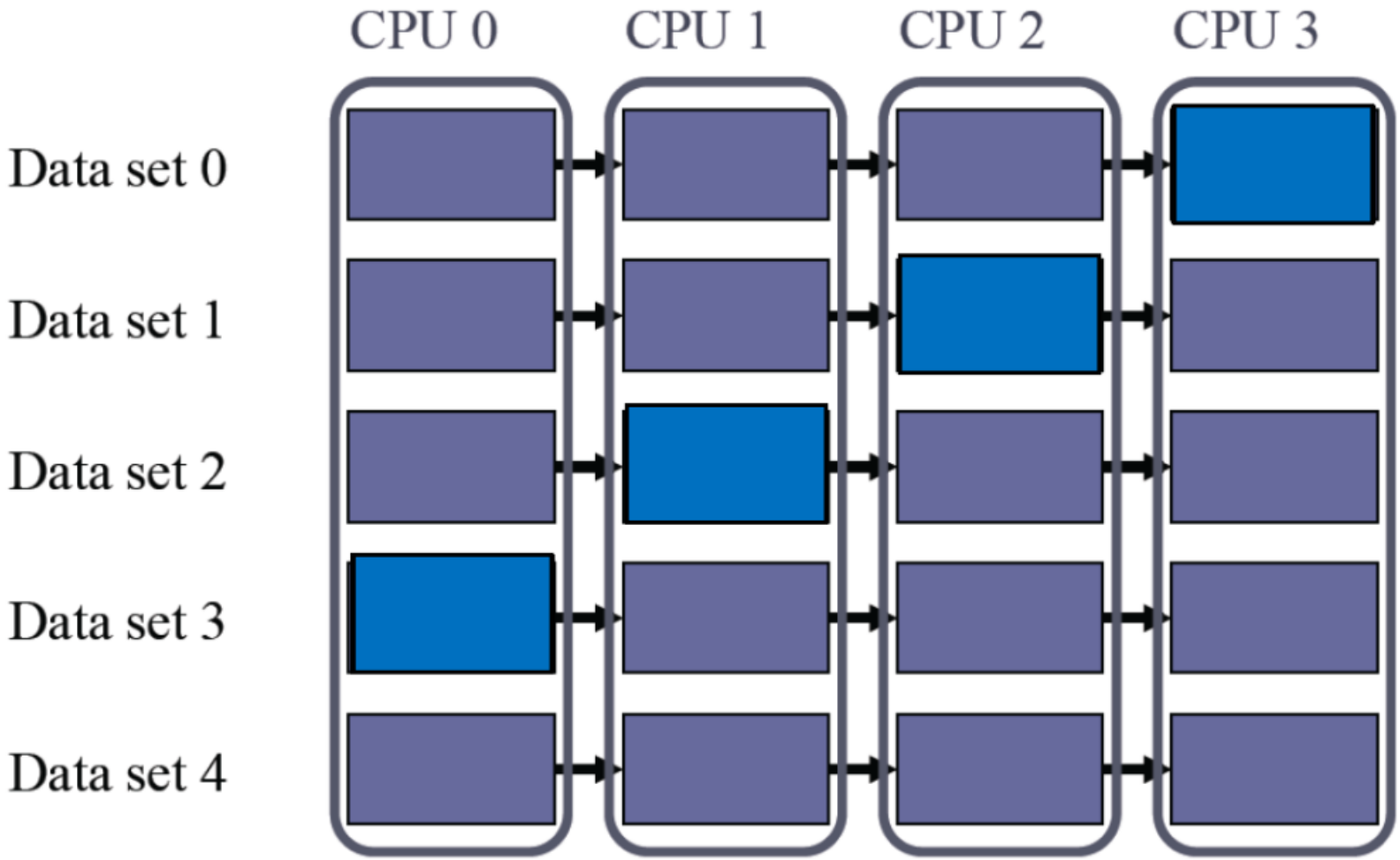
Pipelining Five Data Sets (Step 2)



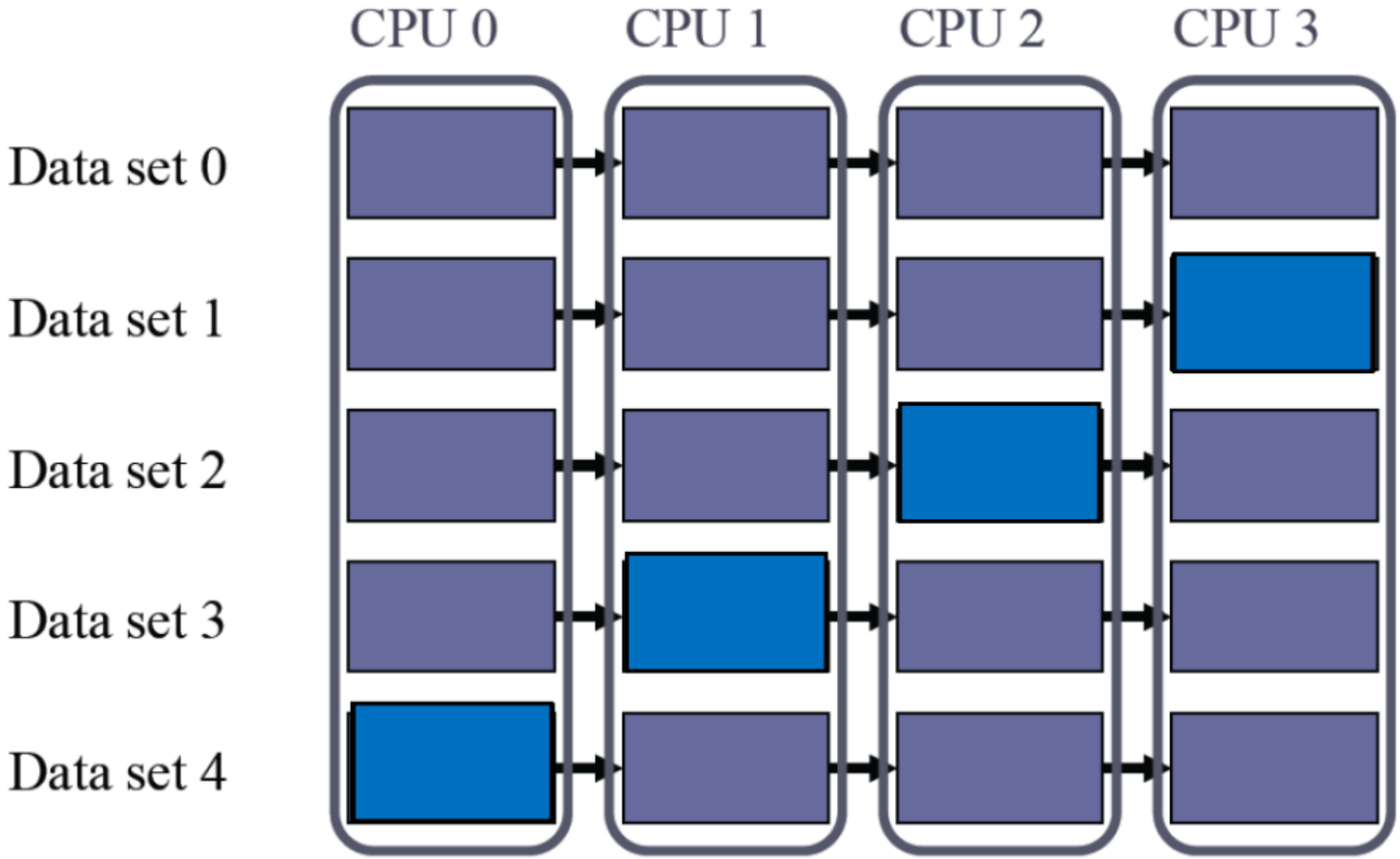
Pipelining Five Data Sets (Step 3)



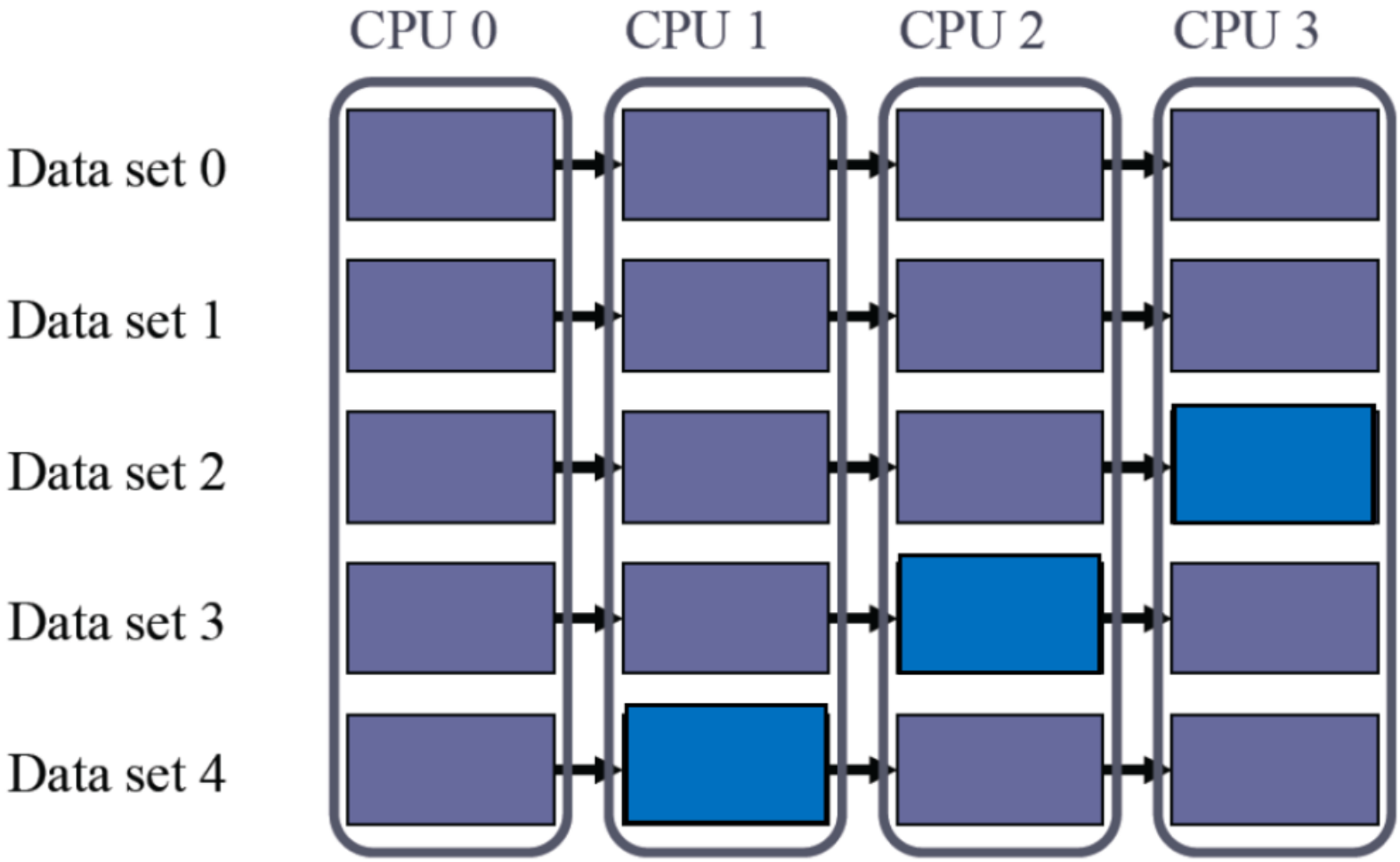
Pipelining Five Data Sets (Step 4)



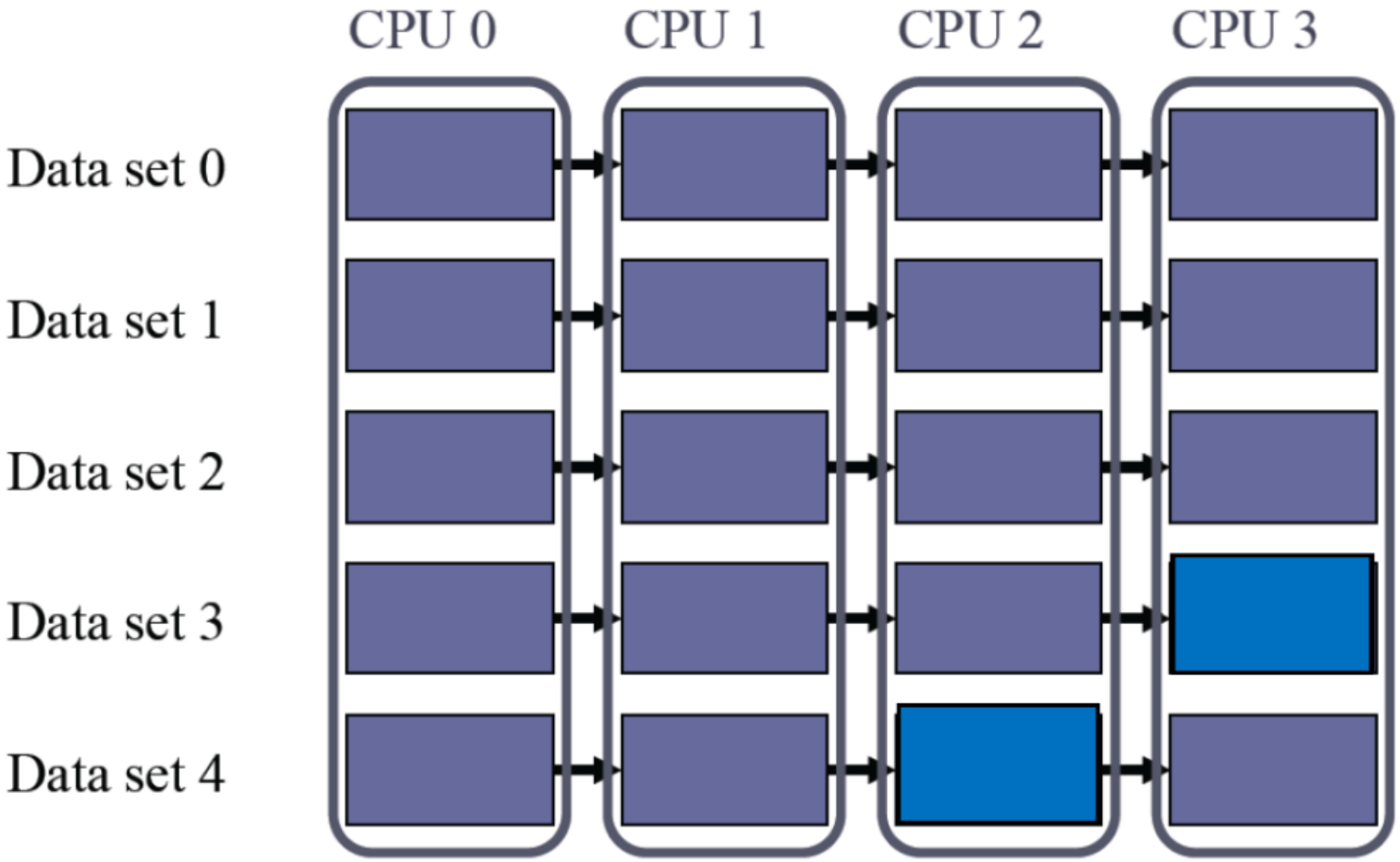
Pipelining Five Data Sets (Step 5)



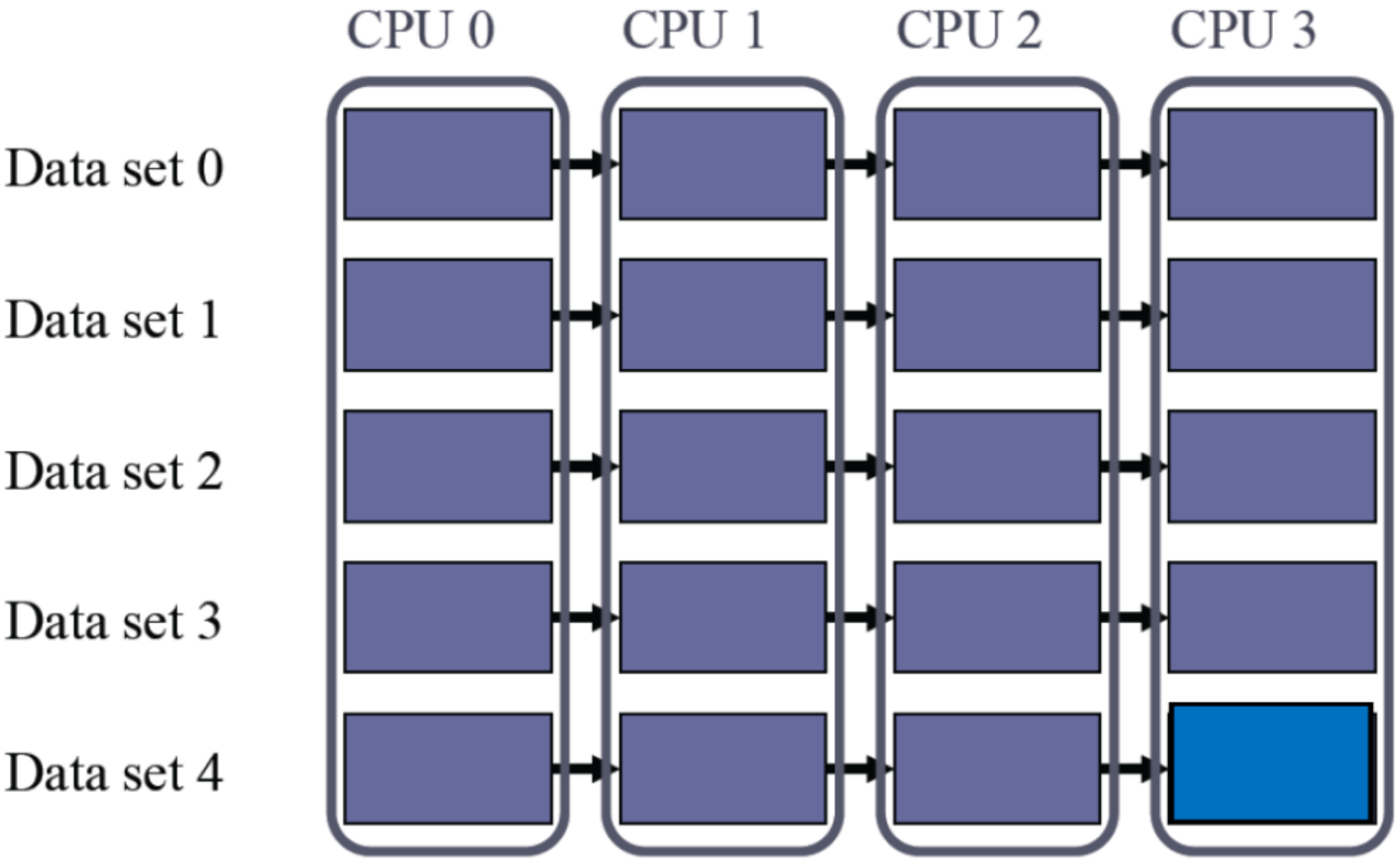
Pipelining Five Data Sets (Step 6)



Pipelining Five Data Sets (Step 7)



Pipelining Five Data Sets (Step 8)



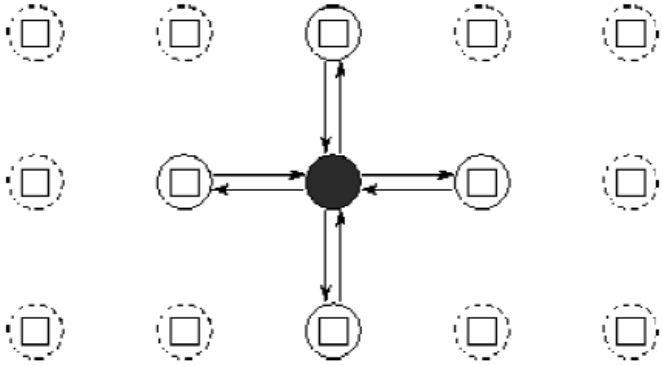
Speedup from Pipelining

- 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

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 are 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 is the overhead of a parallel algorithm, we need to minimize it
- 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 (整合、归并)

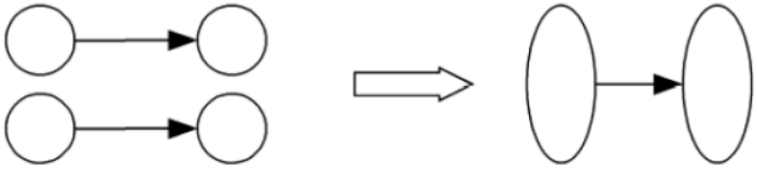
- Grouping tasks into larger tasks
- Goals
  - Improve performance
  - Maintain scalability of program
  - Simplify programming (reduce software engineering costs)
- In message-passing programming, goal often to create one agglomerated task per processor

Agglomeration Can Improve Performance

- Eliminate communication between primitive tasks agglomerated into consolidated task

(a)

- Combine groups of sending and receiving tasks



(b)

Agglomeration Can Maintain the Scalability

- Suppose we want to develop a parallel program that manipulates a 3D matrix of size 8 x 128 x 256.

- If we agglomerate the second and third dimensions, we will not be able to port the program to a parallel computer with more than 8 CPUs.

Agglomeration Can Reduce Engineering Costs

- If we are parallelizing a sequential program, one agglomeration may allow us to make greater use of the existing sequential code, reducing the time and expense of developing the parallel program.

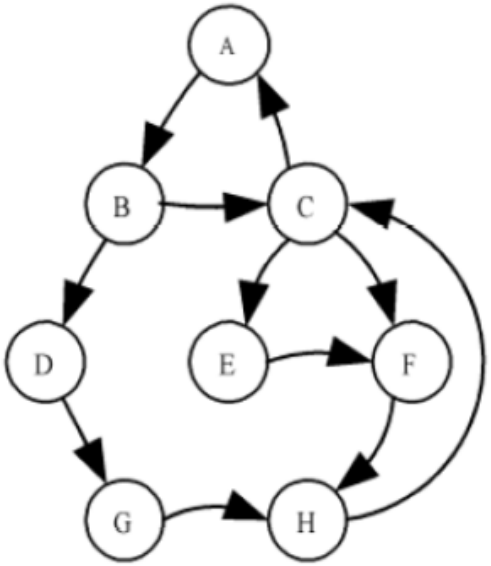
Agglomeration Checklist

- 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

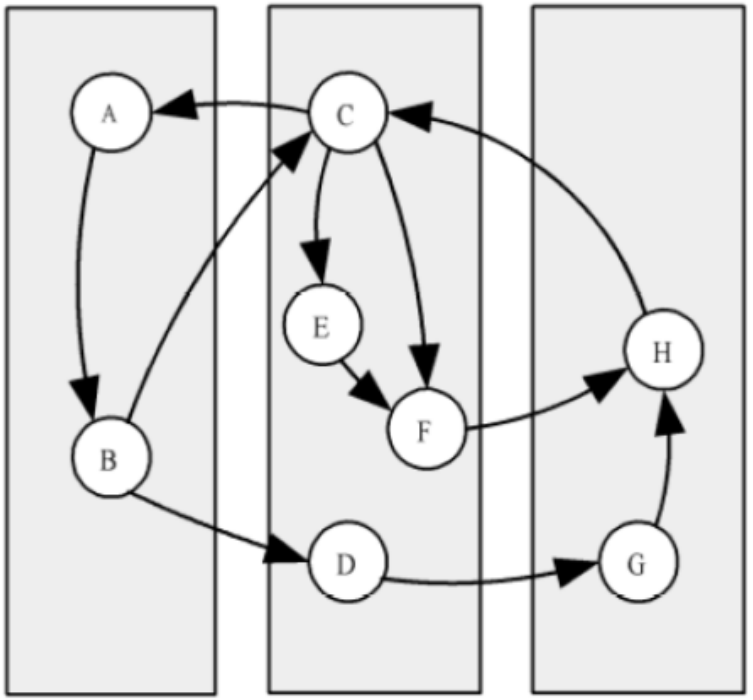
Mapping (映射)

- 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)

A research problem?

Mapping Decision Tree

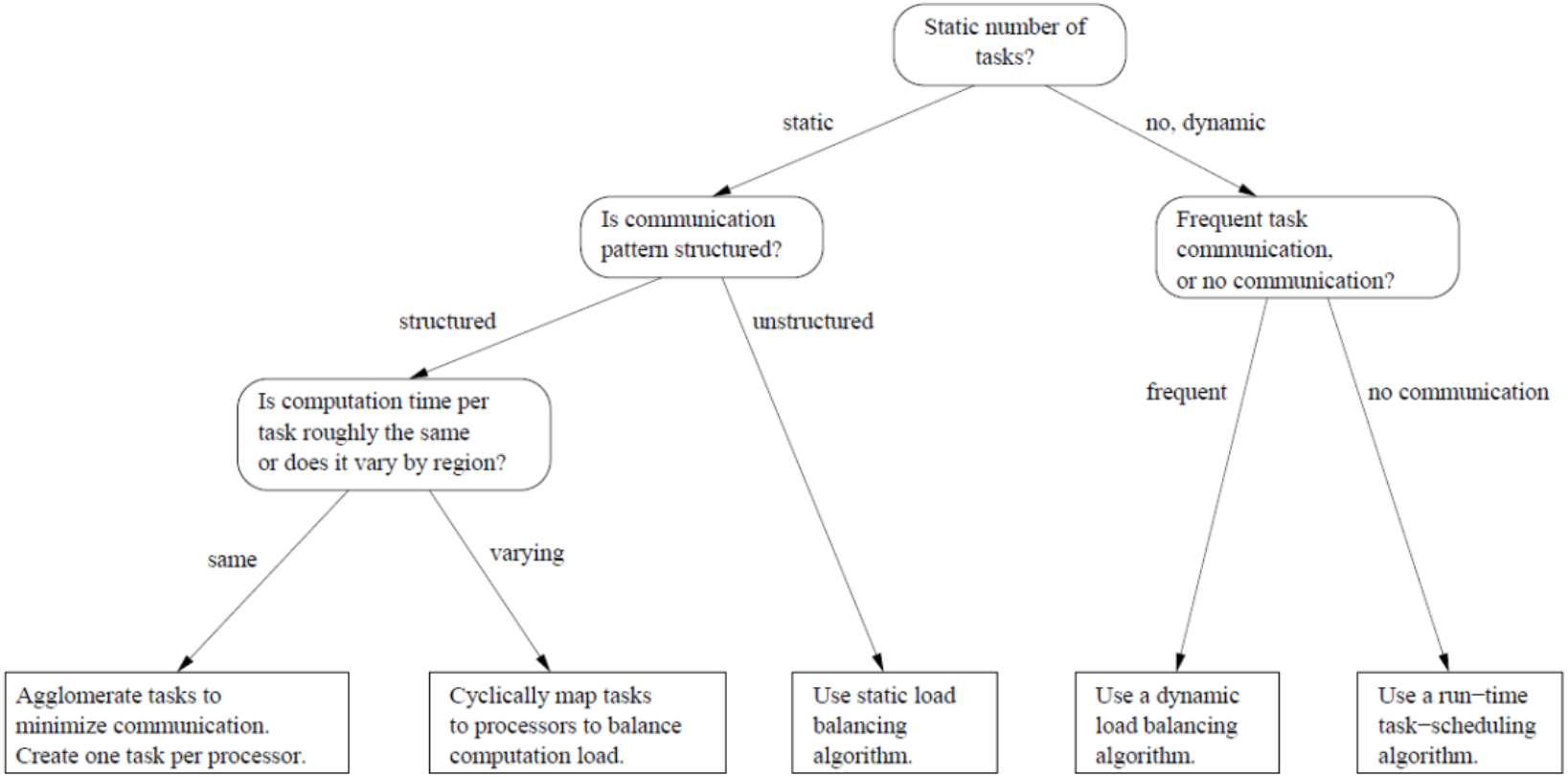
- **Static number of tasks**
  - **Structured communication**
    - **Constant computation time per task**
      - (a) Agglomerate tasks to minimize comm
      - (b) Create one task per processor
    - **Variable computation time per task**
      - (a) Cyclically map tasks to processors
  - **Unstructured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

…

Mapping Decision Tree

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

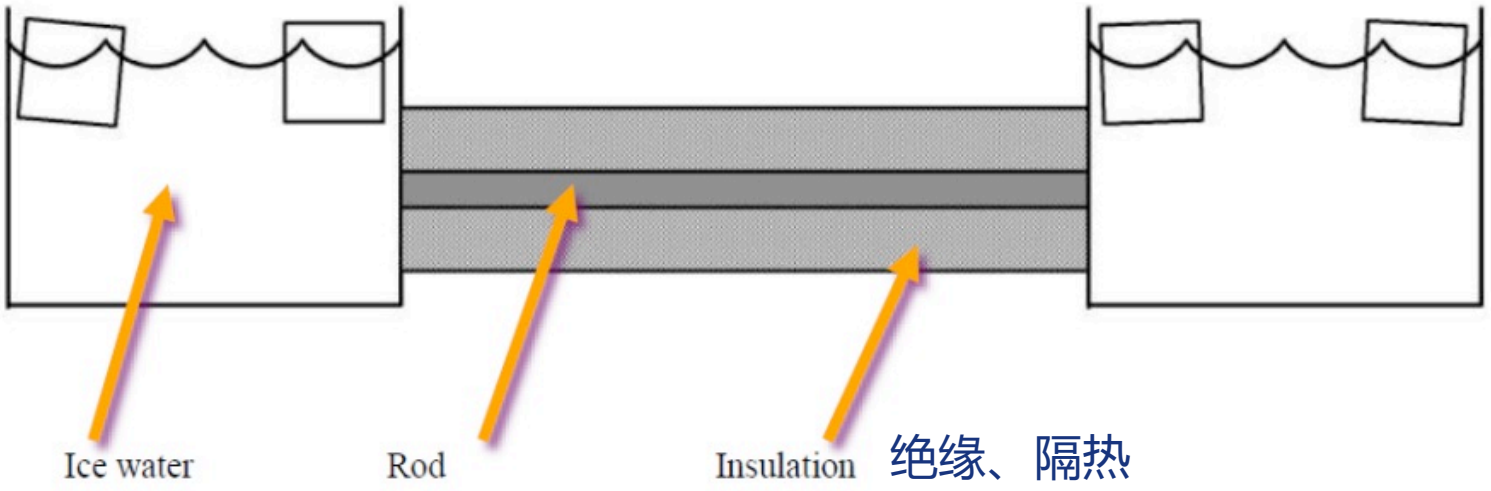
Mapping Decision Tree



Mapping Checklist

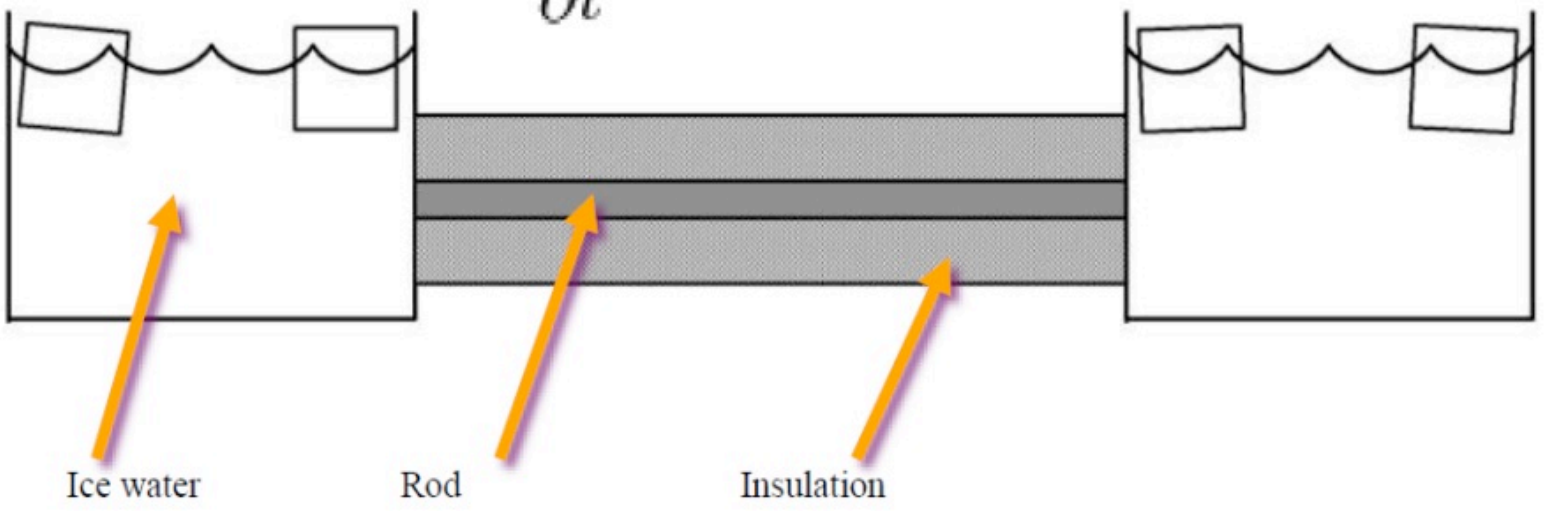
- Considered designs based on one task per processor and multiple tasks per processor
- Evaluated static and dynamic task allocation
- If dynamic task allocation chosen, task allocator is not a bottleneck to performance
- If static task allocation chosen, ratio of tasks to processors is at least 10:1

Case Study Boundary Value Problem

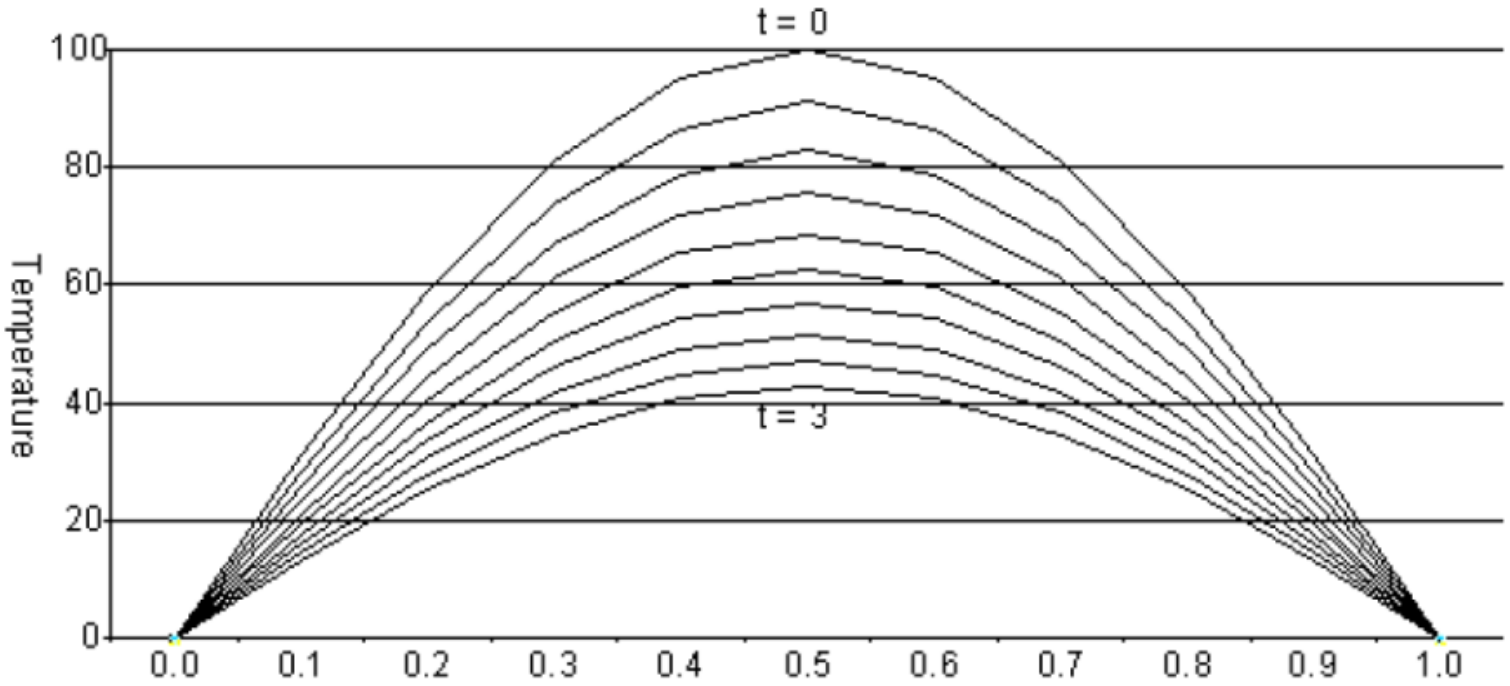


Problem Modeling

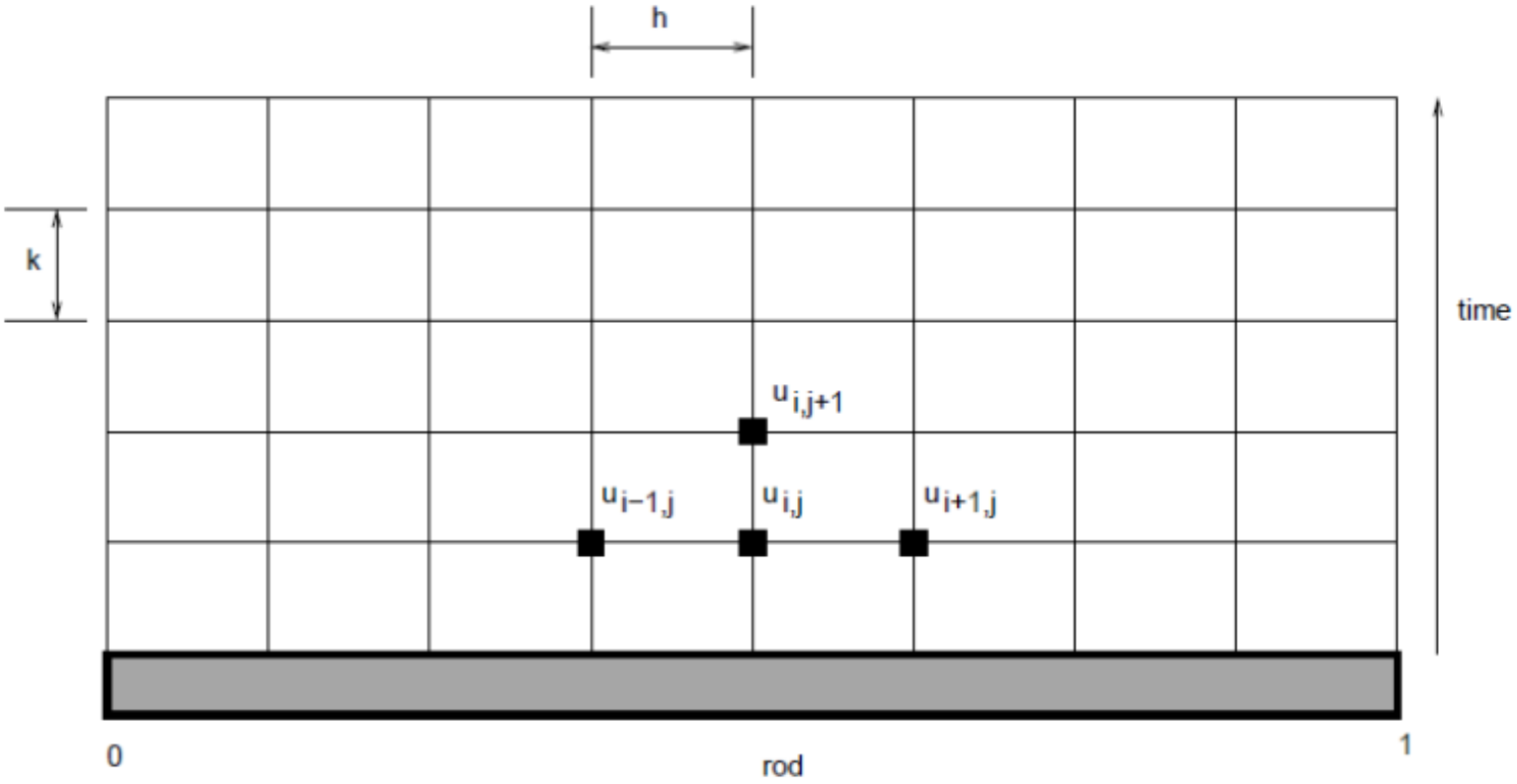
$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$$



Rod Cools as Time Progresses

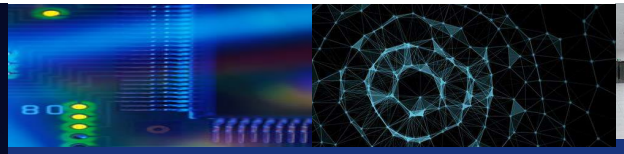


Finite Difference Approximation (有限差分)



Two dimensional grid representing points at which solution is approximated.

$$u_{i,j+1} = u_{i,j} + \frac{k \cdot u_{i-1,j} - 2k \cdot u_{i,j} + k \cdot u_{i+1,j}}{h^2} = u_{i,j} + \frac{k}{h^2} \cdot (u_{i-1,j} - 2u_{i,j} + u_{i+1,j})$$



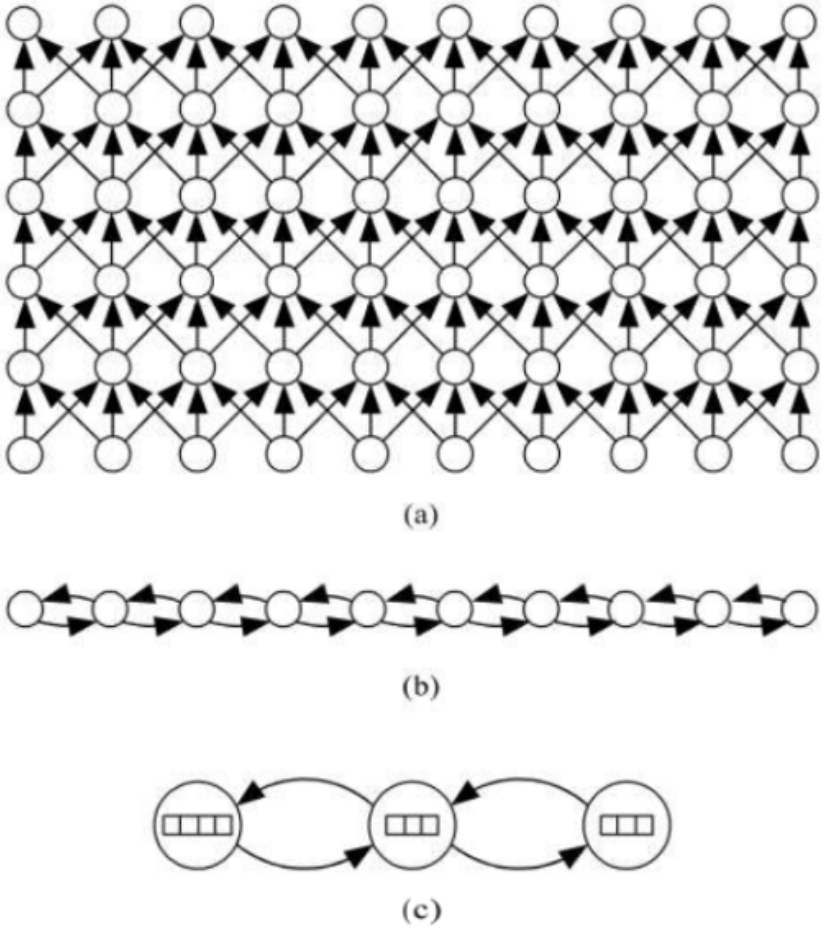
Partitioning

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

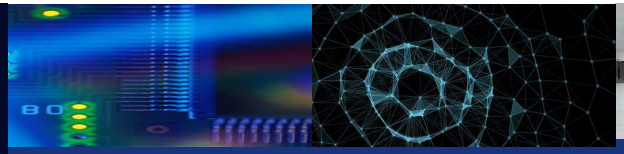
Communication

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

Agglomeration and Mapping



Agglomeration



Parallel and Distributed Computing

DEPENDENCY GRAPH

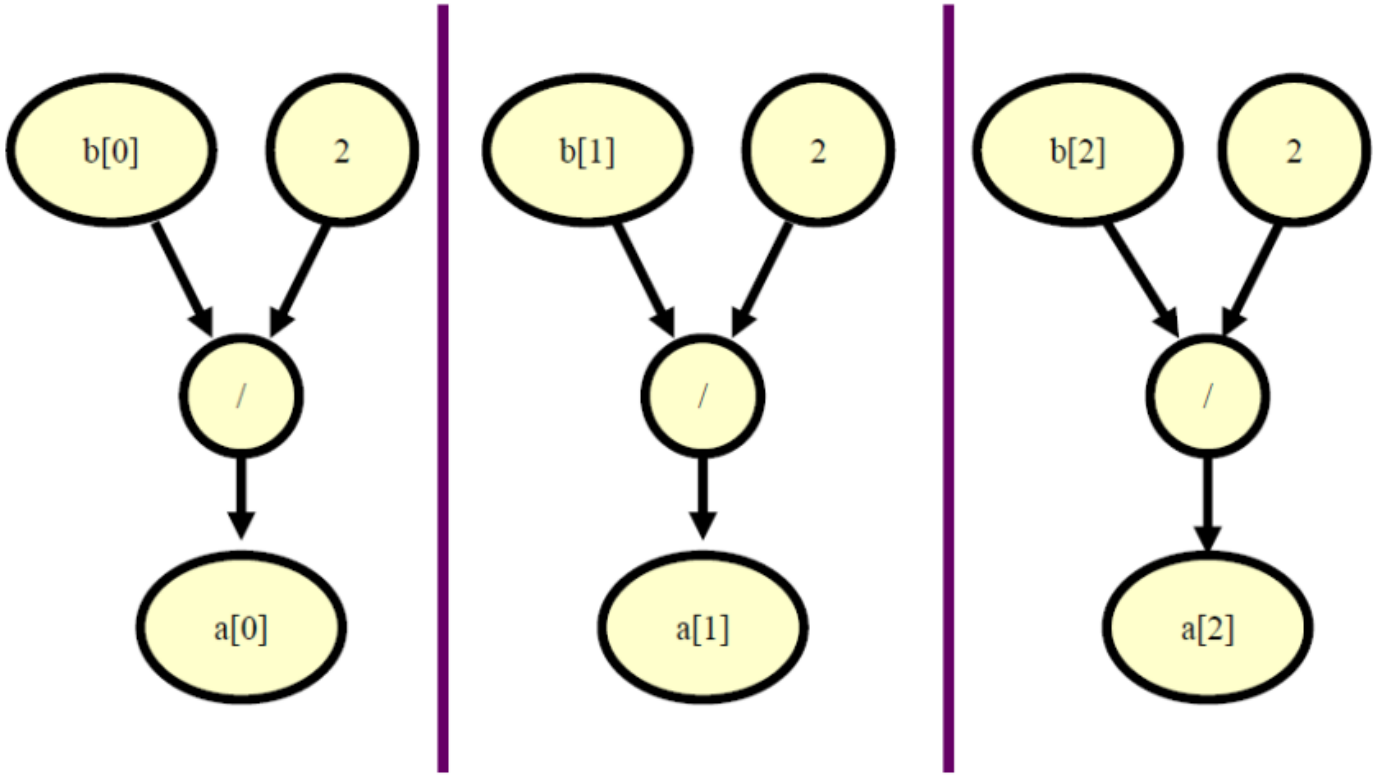
Dependence Graph

- 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

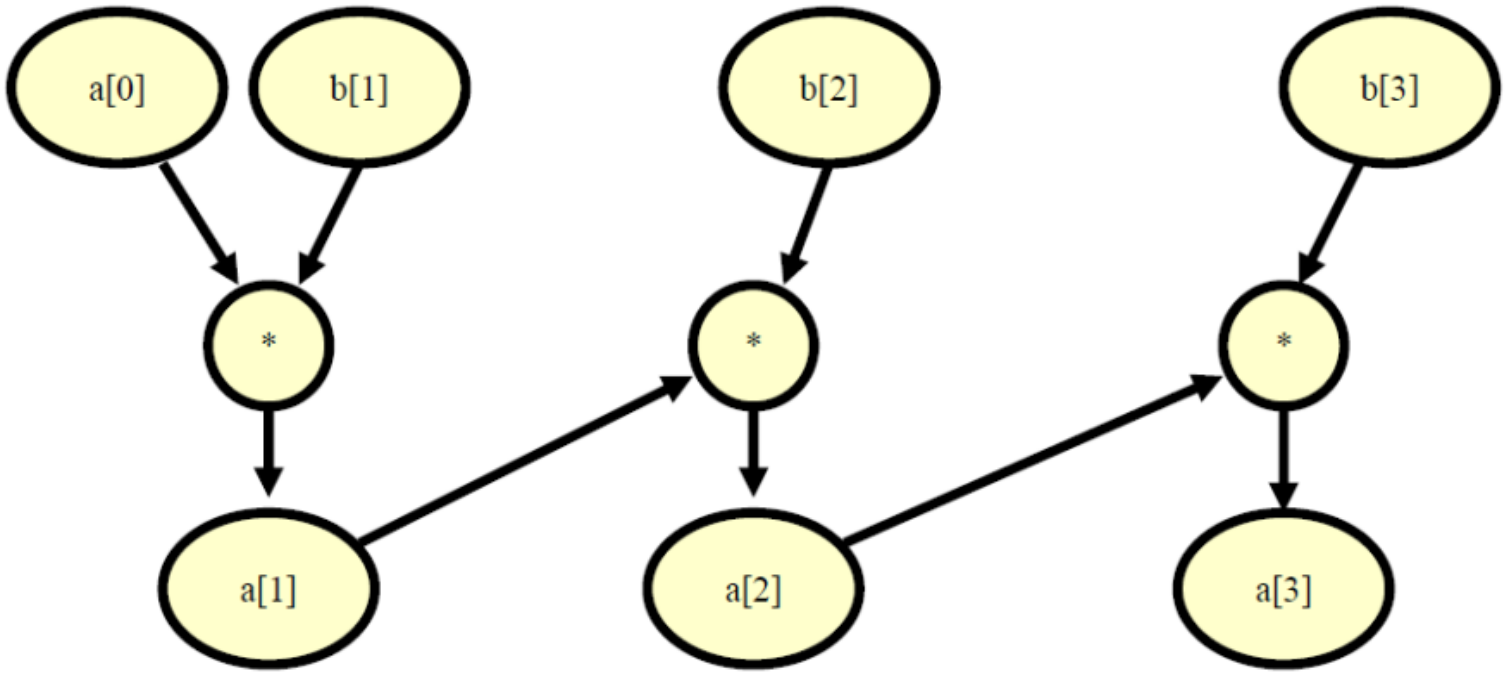
```
for (i = 0; i < 3; i++)
  a[i] = b[i] / 2.0;
```

Domain decomposition possible



Dependence Graph Example #2

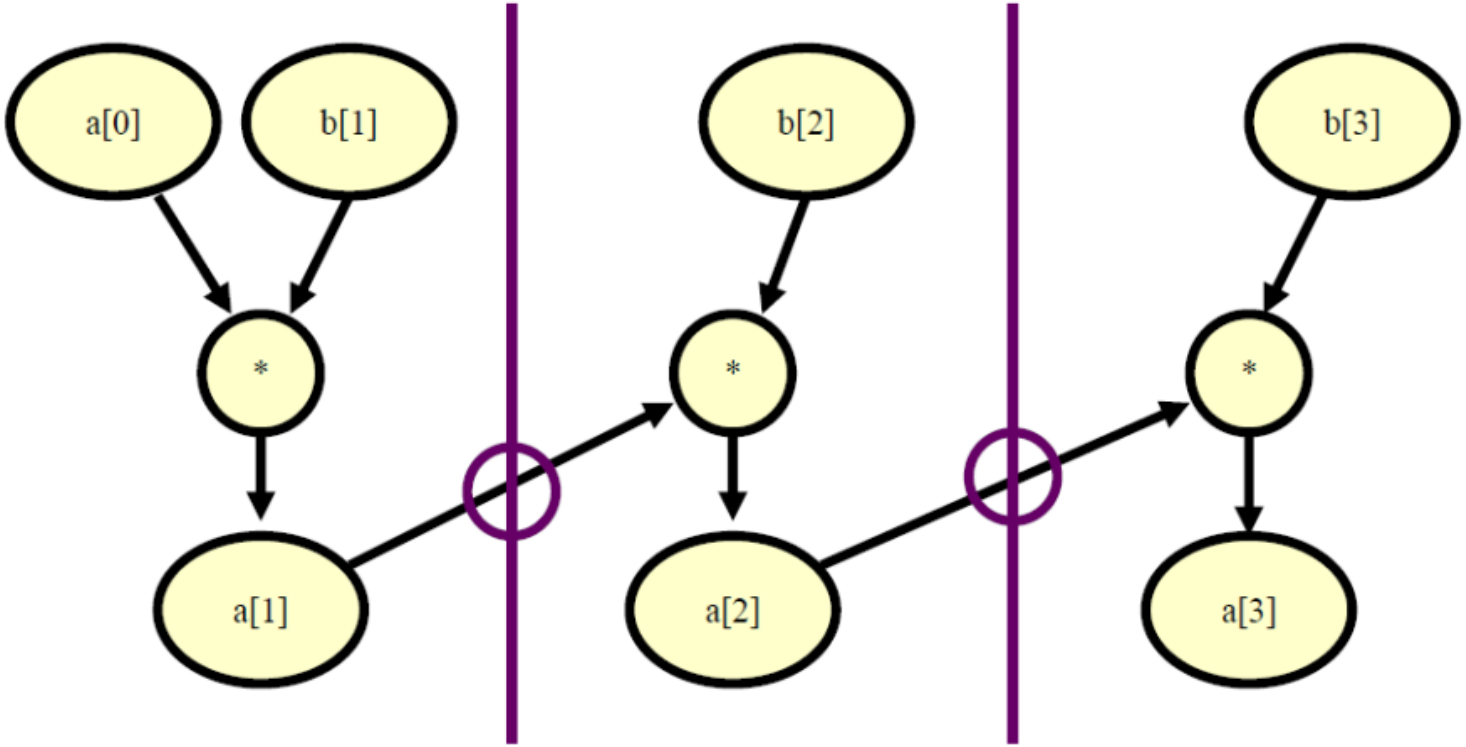
```
for (i = 1; i < 4; i++)
  a[i] = a[i-1] * b[i];
```



Dependence Graph Example #2

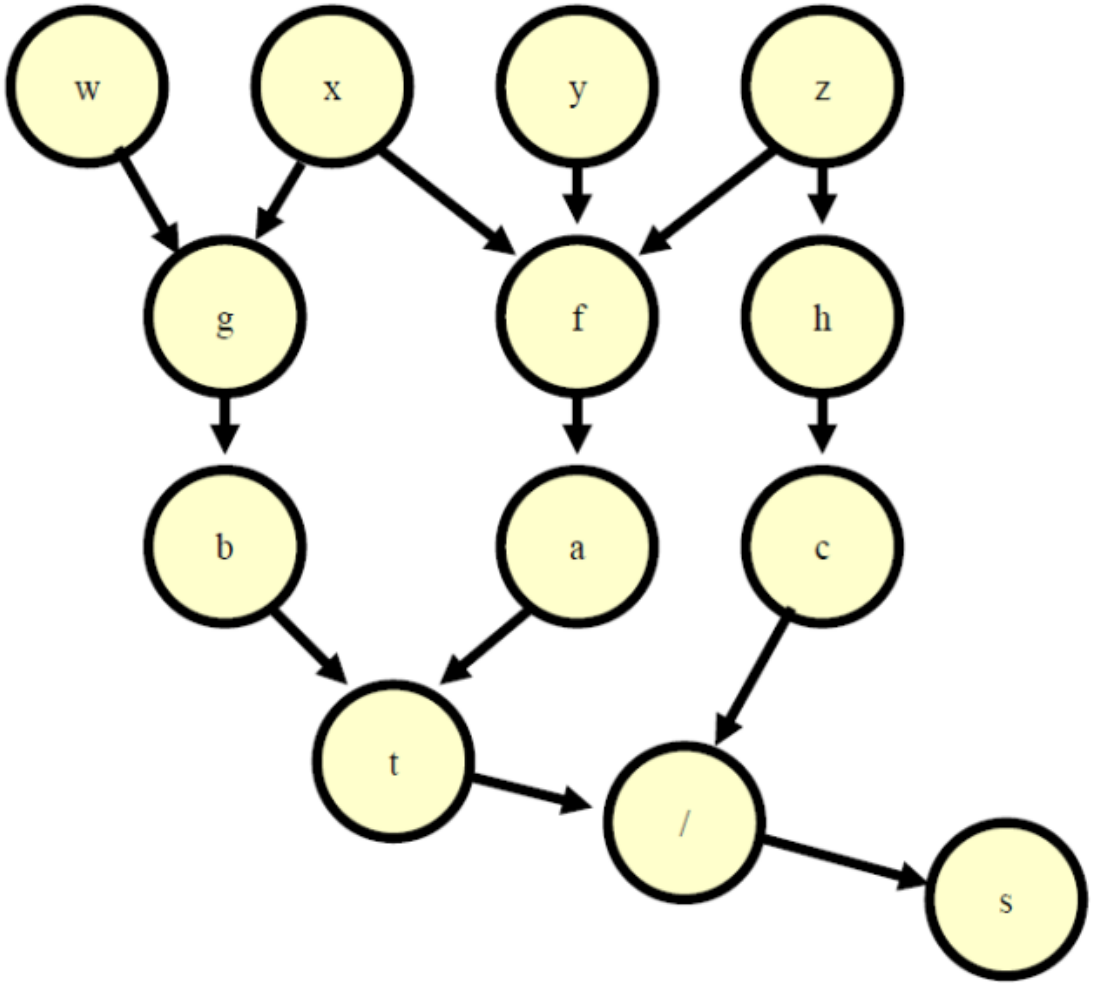
```
for (i = 1; i < 4; i++)
  a[i] = a[i-1] * b[i];
```

No domain decomposition



Dependence Graph Example #3

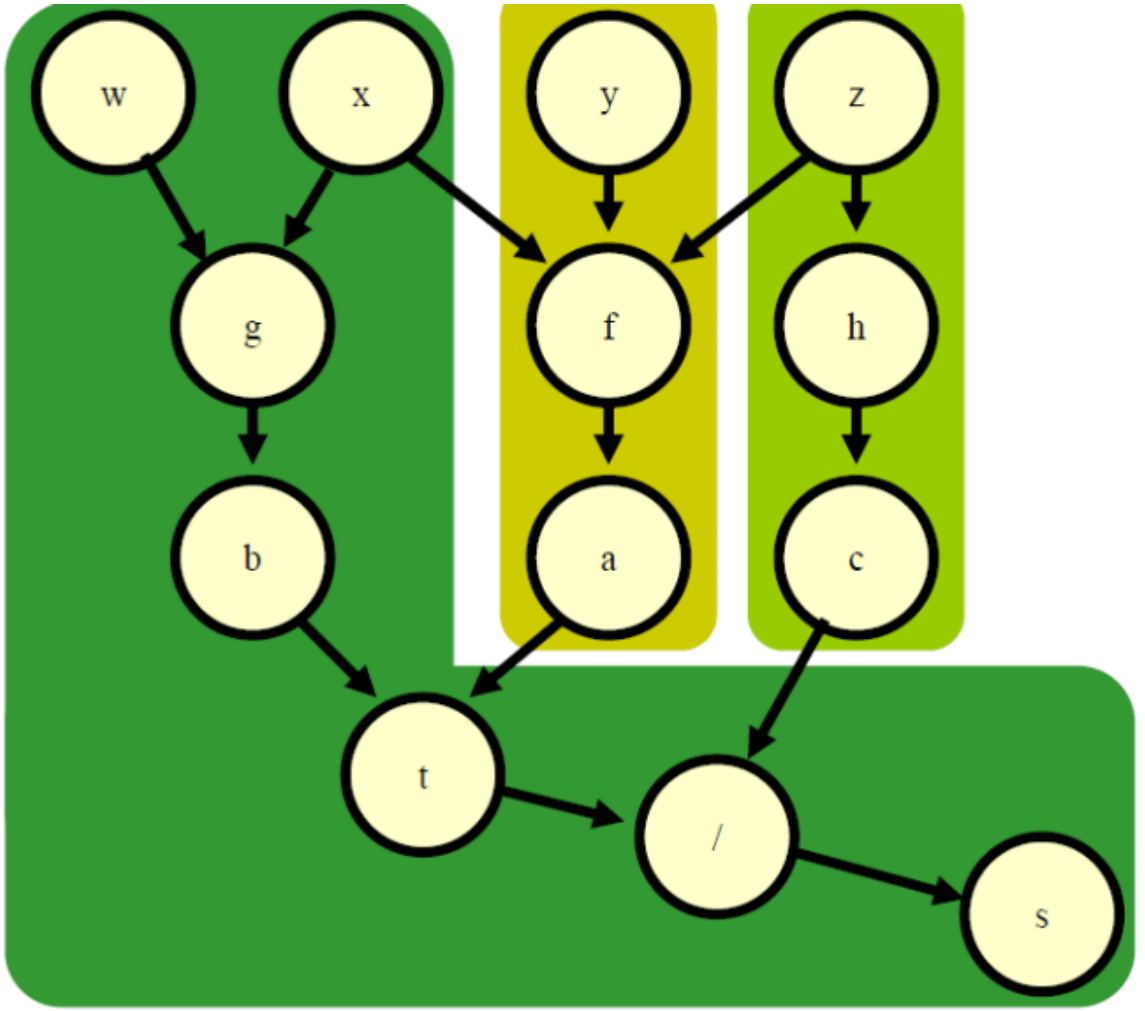
```
a = f(x, y, z);
b = g(w, x);
t = a + b;
c = h(z);
s = t / c;
```



Dependence Graph Example #3

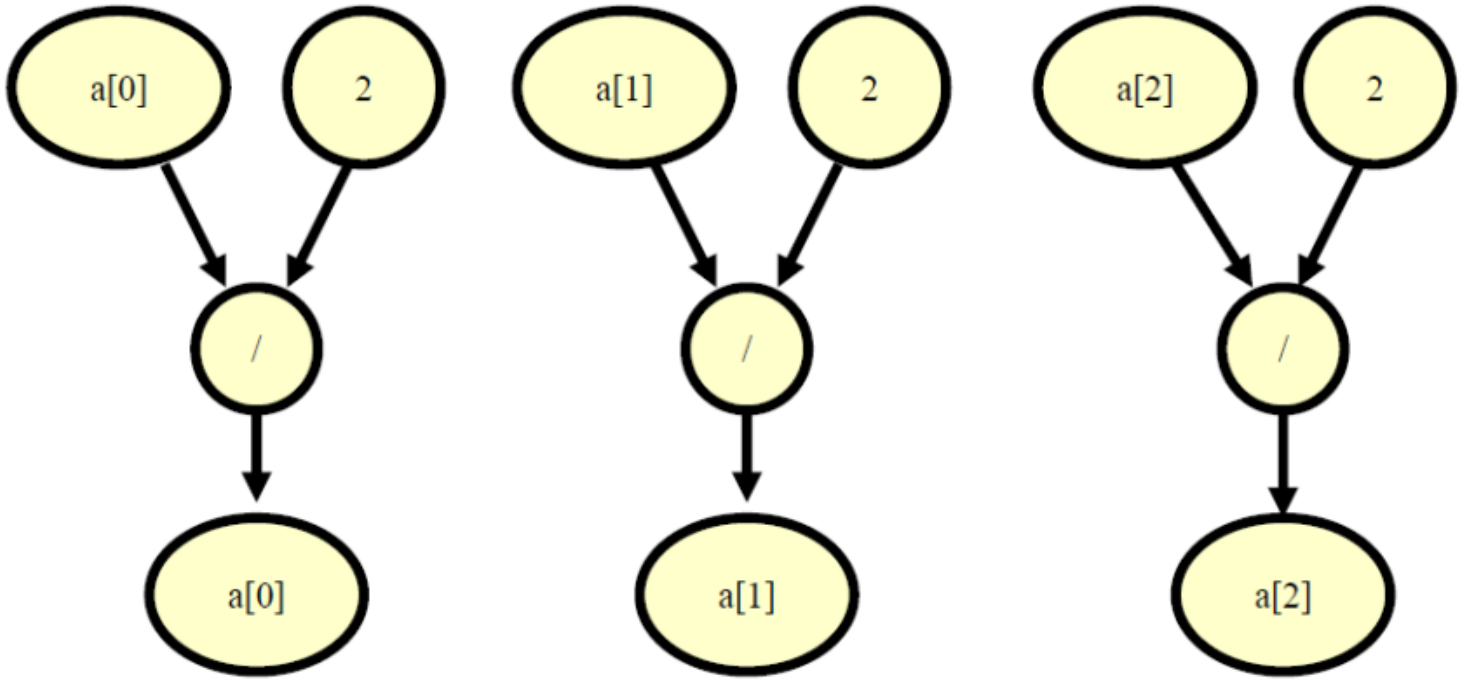
```
a = f(x, y, z);
b = g(w, x);
t = a + b;
c = h(z);
s = t / c;
```

Task decomposition with 3 CPUs.



Dependence Graph Example #4

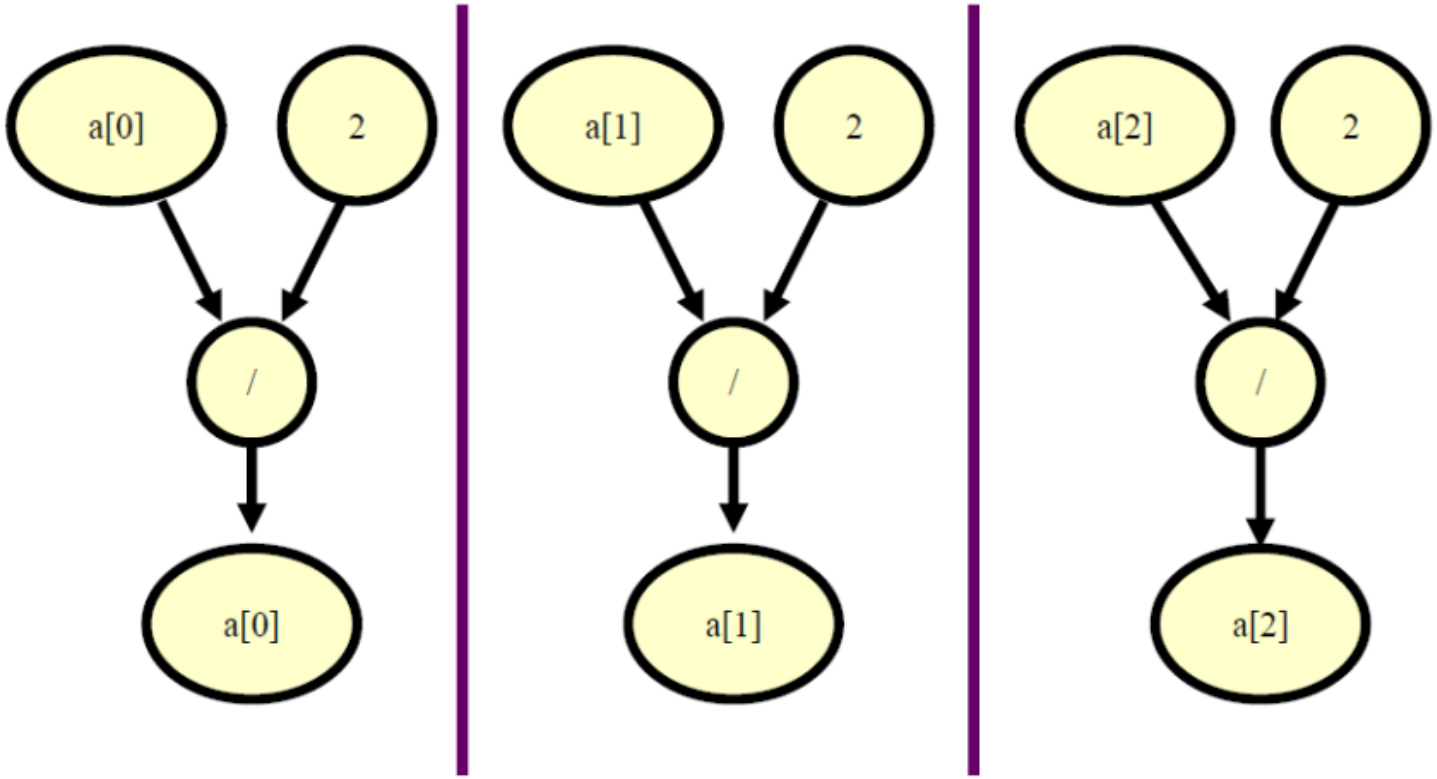
```
for (i = 0; i < 3; i++)
  a[i] = a[i] / 2.0;
```



Dependence Graph Example #4

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

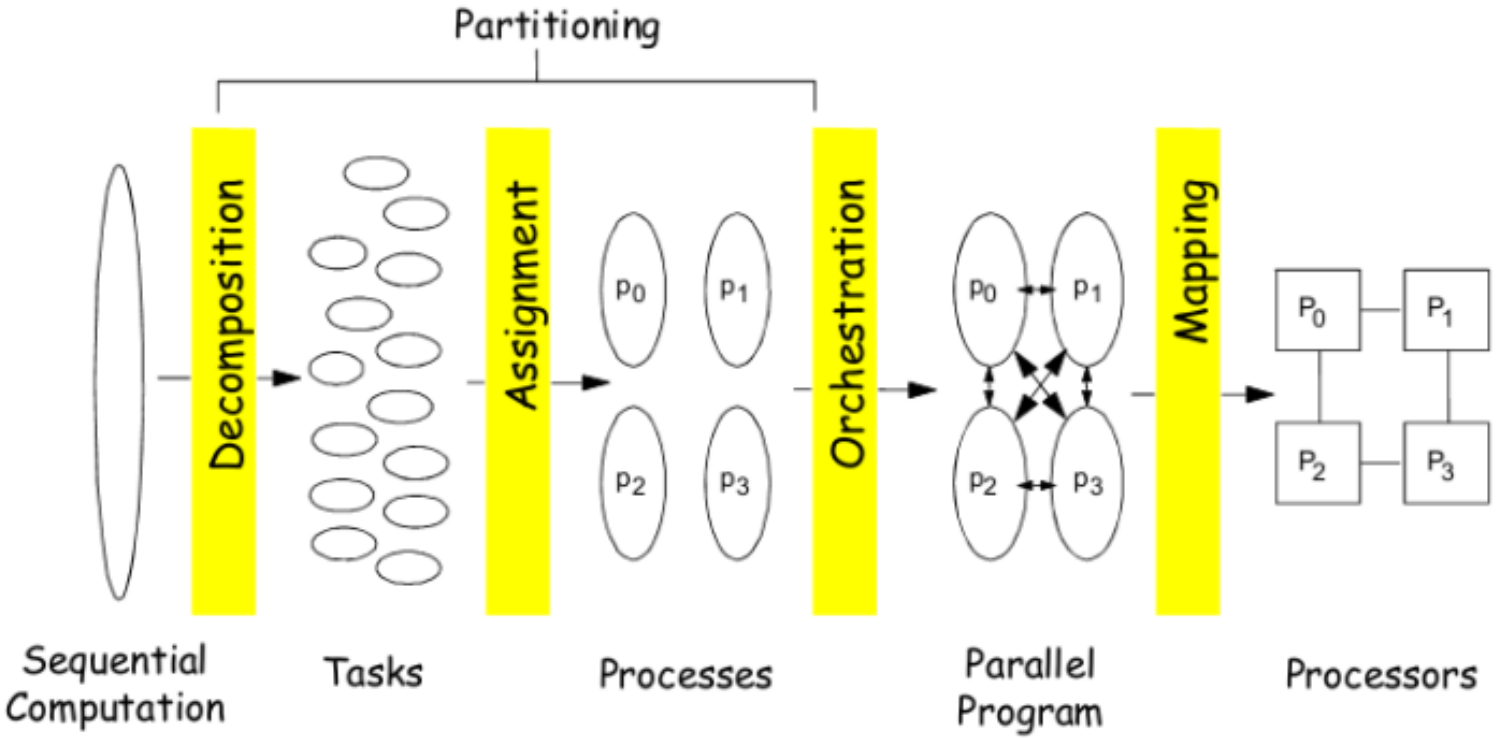
Domain decomposition



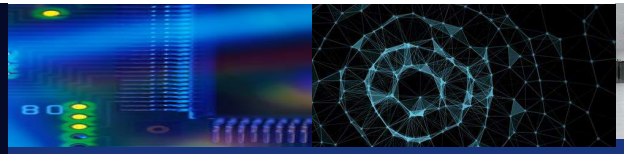
More on Exploring Parallelism

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

Steps in Creating a Parallel Program



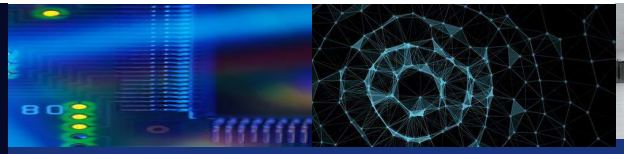
Carnegie Mellon University's public course, Parallel Computer Architecture and Programming, (CS 418) (http://www.cs.cmu.edu/afs/cs/academic/class/15418-s11/public/lectures/)



References:

- Textbook:
  - **Introduction to Parallel Computing, Second Edition, Chapter 3.**
  - **Parallel Computing, Chapter 3. “Parallel Algorithm Design”**
- Website:
  - **CMU Parallel Computing Course:**

    http://www.cs.cmu.edu/afs/cs/academic/class/15418-s11/public/lectures/



Thank You !