# EE/CSCI 451: Parallel and Distributed Computation

Lecture #17

10/15/2020


Viktor Prasanna

prasanna@usc.edu

ceng.usc.edu/~prasanna

University of Southern California

# Outline

Last class

- Task Dependency graph
- Critical path
- Max degree of concurrency
- Task Dependency graph (TDG) → Parallel Program

Today

- Decomposition techniques
  - Algorithms
  - Data
- Data distribution (Chapter 3.2)
  - Array data
  - Block distribution
- Graph partitioning
- Mapping (Chapter 3.4)
- Parallel algorithm models (Chapter 3.6)

# Announcement

- PHW5 due on 10/22 (Thursday)
- HW6 solution released
- HW7 due on 10/16 (Friday)
- Project Proposal due: 10/18 (extended!)
    - Submit on blackboard
- HW8 will be out on 10/16 and due on 10/22 (1 day before midterm2)
- Final exam date: 2-4 PM Thursday, November 19

# Announcement: Midterm 2

- Time: Oct. 23 (Friday) 3:30-5:30 PM
- A sample exam is posted on Piazza!
- Covers material from Sept. 22 to Oct. 16
  - Program mapping questions (covered in Week 6) will be on midterm 2!
- Logistics: same as Midterm 1



4

# Decomposition Techniques

Parallel Solution = Tasks + Concurrency + Interactions

Representation:

Tasks

Dependencies

Interactions

Representation:

Task dependency graph

Task interaction graph

# Task interaction graph

Node – task

Edge – interaction between tasks

Type of interactions:

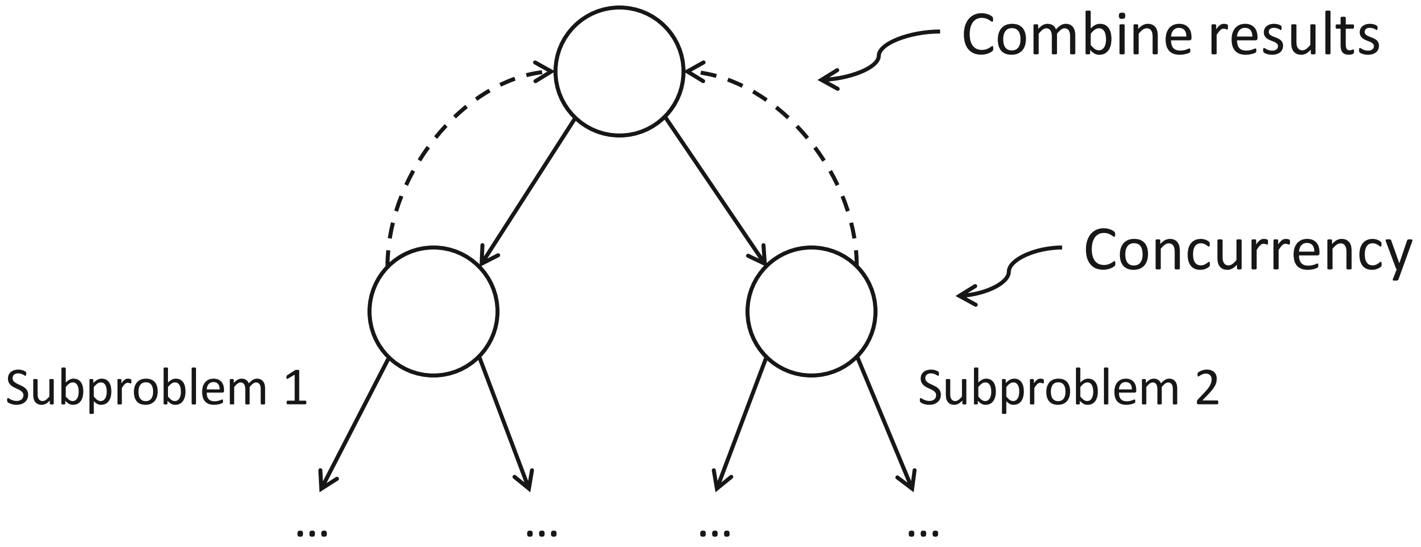
- Static / dynamic

  interactions known at compile time

- Data Access
  - Read data from another task
  - Read/write

# Recursive Decomposition

## Divide and Conquer



# Example (1)

**Quick sort**

$A$: $n$ input array

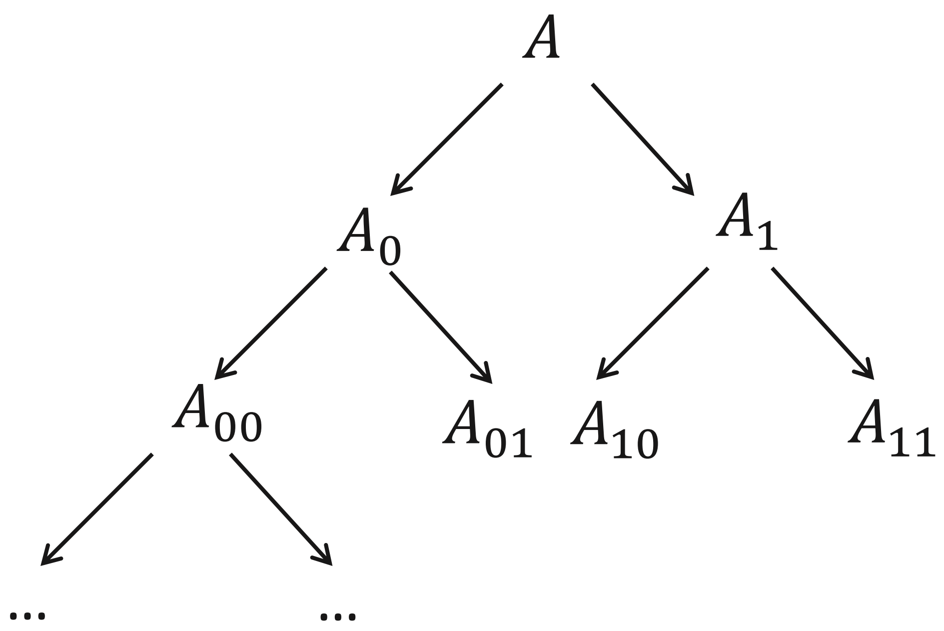
Choose a pivot element $x$

Partition $A$ using $x$

$$\begin{tikzcd} & x \arrow[dl, "\leq"'] \arrow[dr, ">"] & \\ A_0 & & A_1 \end{tikzcd}$$

Sort $A_0$ and $A_1$ in parallel (recursively)

# Example (2)




Each node: Choose pivot

Identify data for left and right subtree

# Example - Task Dependency Graph (3)

QS ($A$)

$|A| = 1$ return

Choose pivot $x$

$A_0 \leftarrow$ those elements in $A \leq x$

$A_1 \leftarrow$ those elements in $A > x$

Do in parallel ← **Parallelism**

QS ($A_0$)

QS ($A_1$)

End

# Analysis

## Parallel Time

PRAM, $n$ processors

$T_p(n)$ = parallel time for QuickSort using $p$ processes on $n$ data items

best case

$$T_n(n) = T_{n/2}(n/2) + O(n)$$

$$T_n(n) = O(n)$$ best case

worst case

$$T_n(n) = \max\{T_{n-1}(n-1), T_1(1)\} + O(n)$$

$$T_n(n) = O(n^2)$$ worst case

# Example

## Merge Sort

MS $(A(0), \ldots, A(n-1))$

Sort $\longrightarrow$ If $|A| = 1$ return

Decompose

Do in parallel

MS $(A(0), \ldots, A(n/_2 - 1))$

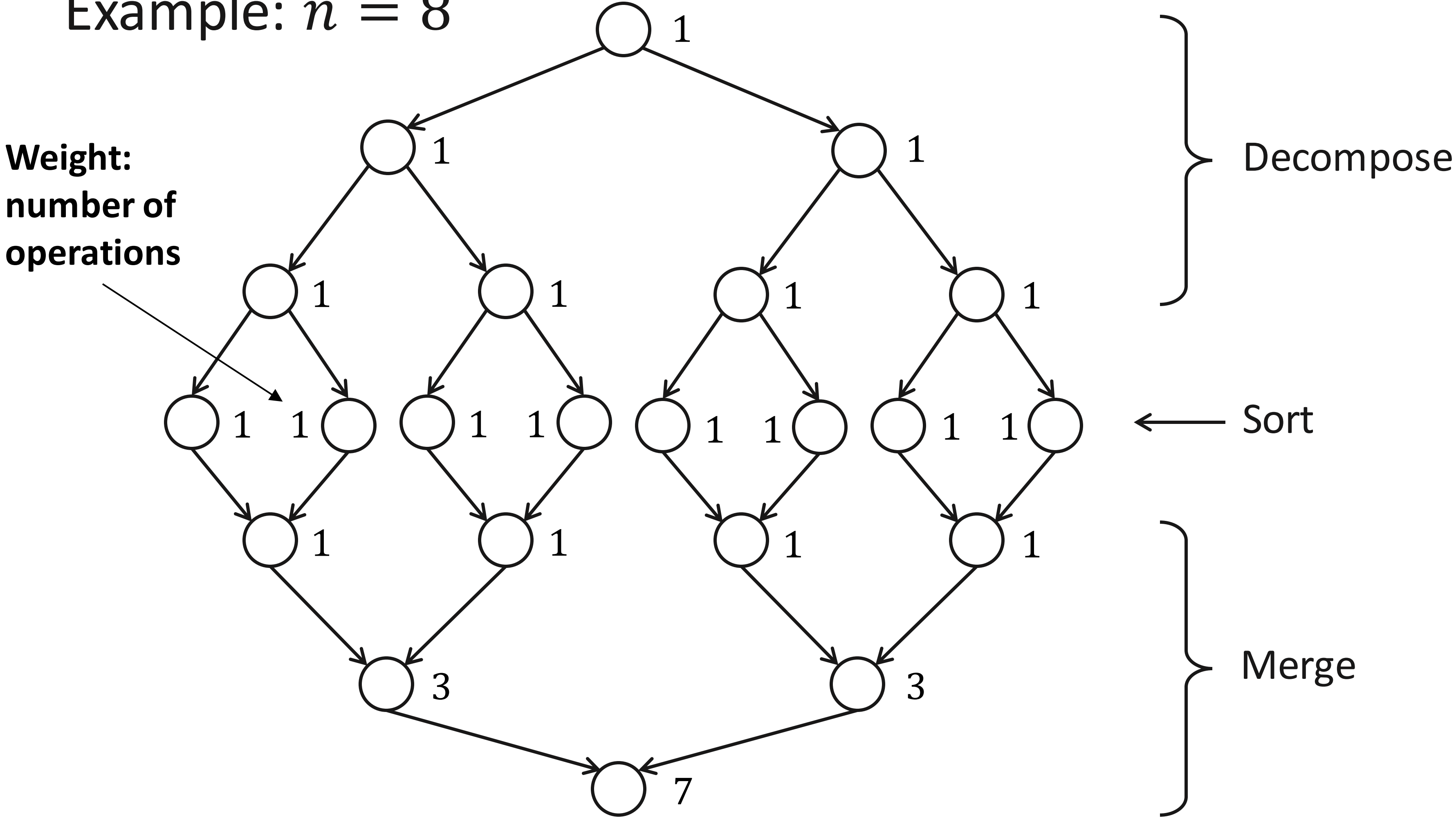
MS $(A(n/_2), \ldots, A(n-1))$

End

Merge $\longrightarrow$ Merge the two sorted sequences of size $n/_2$

# Task Dependency Graph

Example: $n = 8$



# Analysis

**Parallel time on $n$ processor PRAM**

$T_p(n)$ = parallel time for MergeSort using $p$ processes on $n$ data items

$$T_n(n) = T_{n/2}(n/2) + O(n)$$

Serial merge

$$T_1(1) = O(1)$$

$$T_n(n) = O(n)$$

If we use one processor (Serial merge sort)

$$T_1(n) = \mathbf{2}T_1(n/2) + O(n)$$

Note: Decomposition is fast

Merge takes most of the time

# Data Decomposition

Decompose data (partition the data)

Operate on data $\longrightarrow$ tasks

Usually tasks on the partitioned data are similar and can be performed in parallel

Data decomposition $\longrightarrow$ Concurrent tasks $\longrightarrow$ Performance

# Data Decomposition Example

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \to \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Partition of input and output matrices into $2 \times 2$ submatrices

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

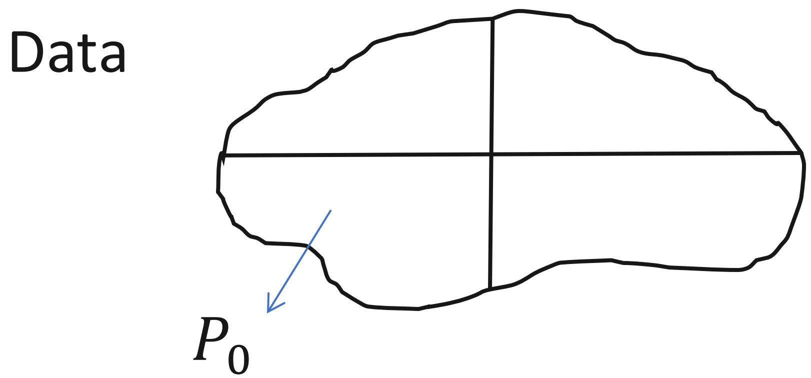
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

A decomposition of matrix multiplication into four tasks based on the partitioning of matrices above

# Owner-Computes Rule




$P_0$ performs all computations on the input and output data it owns

Ex: Matrix multiplication $C \leftarrow A \times B$

$P_{i,j}$ − owns blocks $A(i,j), B(i,j), C(i,j)$

Computes block $C(i,j)$

Block size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ , $p =$ total number of processes

# Array Distribution Schemes

Widely used strategy

- Data decomposition using input data partitioning

- Owner computes rule

⇨ Defines tasks, mapping to processes

Key problems: Distributing 2-D arrays

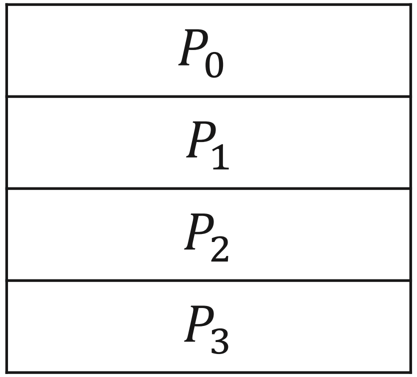
Partitioning graph data

# Block Distribution (1)

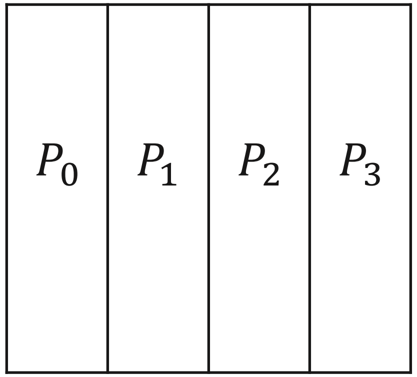
2-D Array $\longrightarrow$ Processes

Block (continuous portion of array) $\longrightarrow$ Process

Example




row-wise distribution




column-wise distribution

# Block Distribution (2)

$$p = \text{number of processes} = p_1 \times p_2$$

Block size $\quad {}^{n}/_{p_1} \times {}^{n}/_{p_2}$

| | | | |
|---|---|---|---|
| $P_0$ | $P_1$ | $P_2$ | $P_3$ |
| $P_4$ | $P_5$ | $P_6$ | $P_7$ |
| $P_8$ | $P_9$ | $P_{10}$ | $P_{11}$ |
| $P_{12}$ | $P_{13}$ | $P_{14}$ | $P_{15}$ |

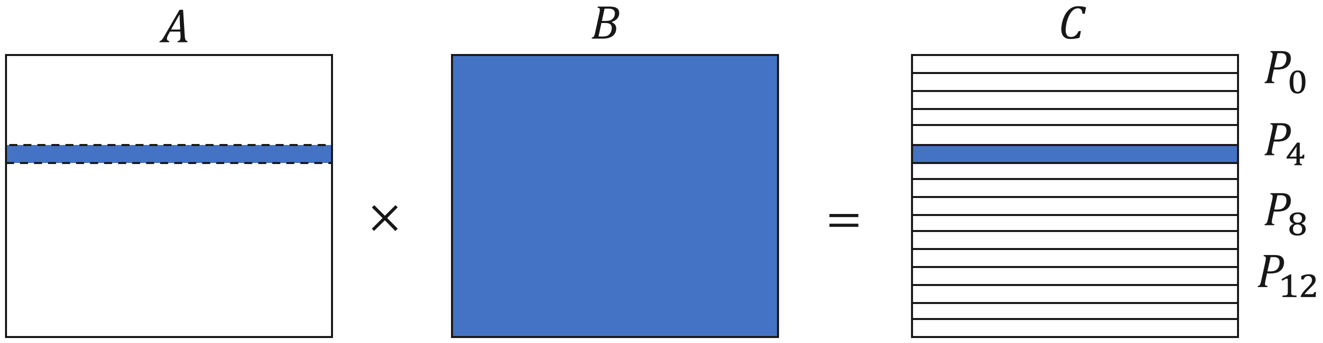
$4 \times 4$ process grid

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ |
| $P_8$ | $P_9$ | $P_{10}$ | $P_{11}$ | $P_{12}$ | $P_{13}$ | $P_{14}$ | $P_{15}$ |

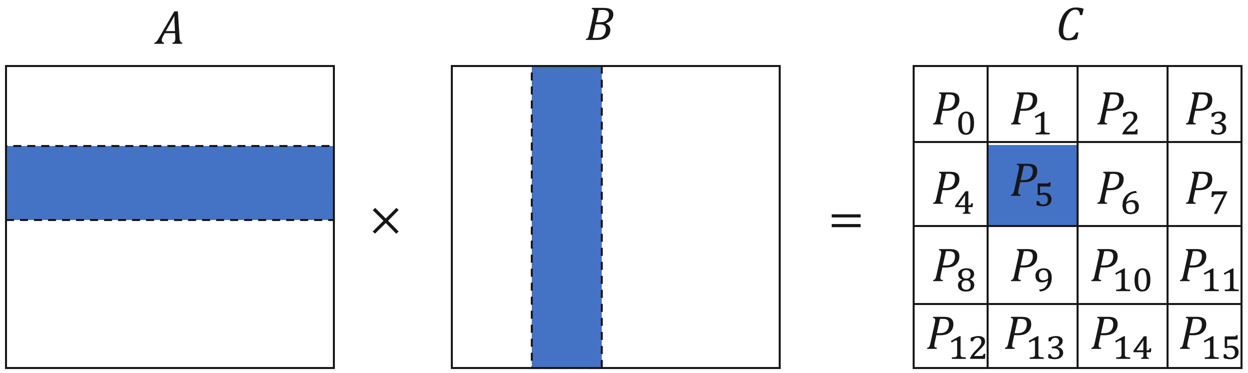
$2 \times 8$ process grid

# Impact of Distribution on Data Sharing (Communication)

Ex.1




Ex.2




Data needed for computing the shaded portion of output matrix (output matrix partitioning)

# Load Balancing for MM by Output Partitioning (1)

$n \times n$ matrix multiplication

$p$ processes

Output partitioning : One dimensional: $\frac{n}{p} \times n$

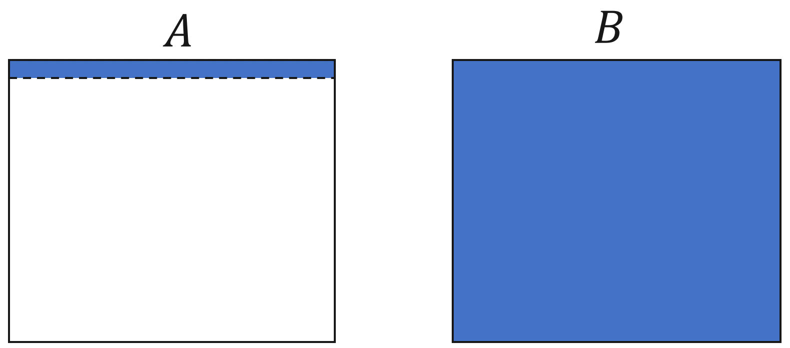
2-dimensional: $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$

Owner Computes Rule:
Each process does the same amount of work ($^{n^3}/_{p}$)
Total amount of communication varies

# Load Balancing for MM by Output Partitioning (2)

**Ex.1:** 1-dimensional $p$ output processes

$$C \leftarrow A \times B$$




$$P_i \rightarrow \frac{n}{p} \text{ Rows of } A$$
$$\frac{n}{p} \text{ Columns of } B$$

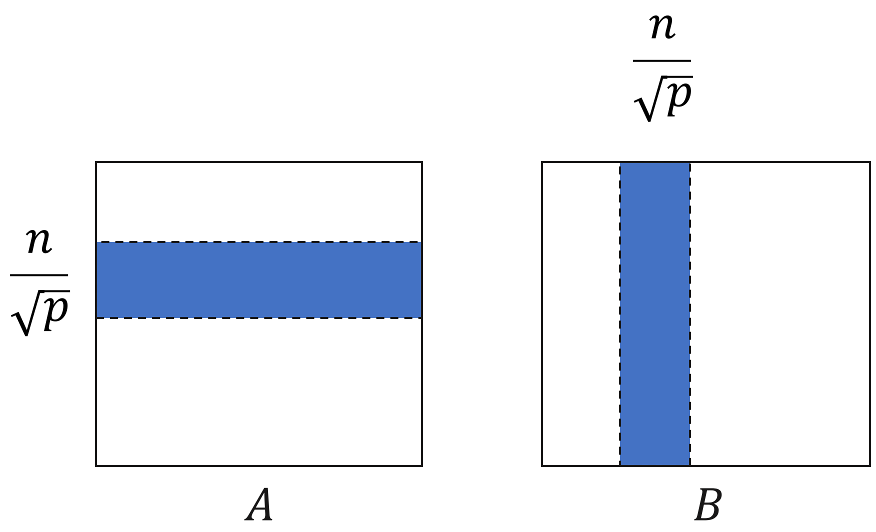
Each process computes $\frac{n}{p}$ output rows

Total Communication = $O(n^2)$ for each output process

Entire $B$ Matrix is needed

# Load Balancing for MM by Output Partitioning (3)

**Ex.2:** 2-dimensional $p$ output processes




$P_i \rightarrow \frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ blocks of $A$ and $B$

Each process computes output block of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$

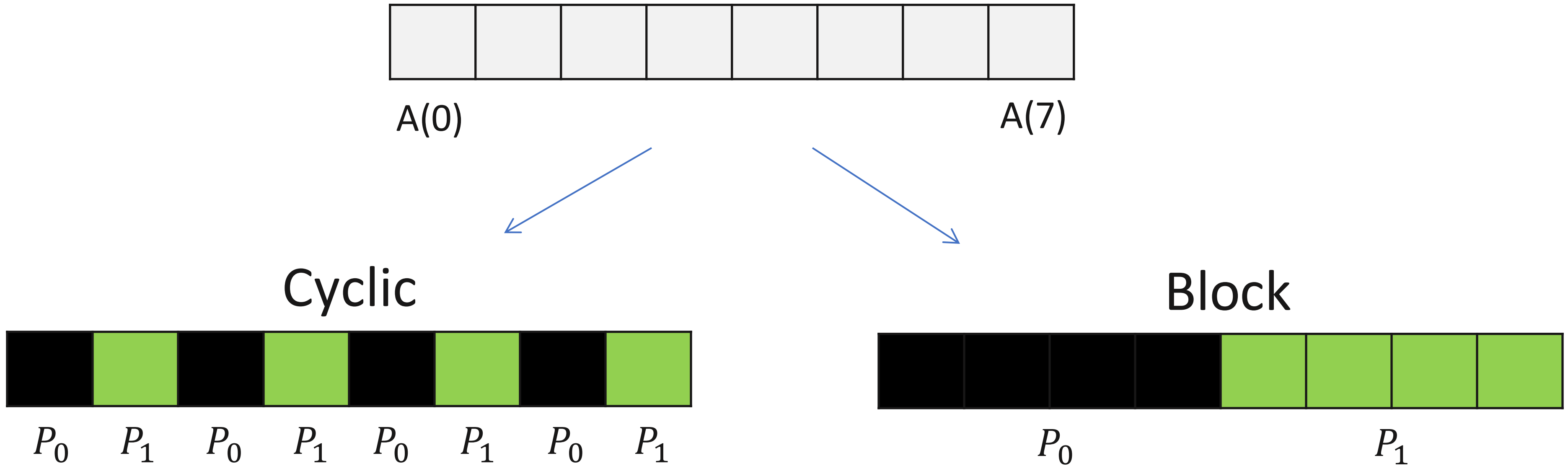
Total Communication = $O\left(\sqrt{p} \cdot \left(\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}\right)\right)$ ← Block MM

= $O\left(\frac{n^2}{\sqrt{p}}\right)$ per each output process

# Cyclic and Block Distribution

Problem: distribute 1-D array $n > p$ elements to $p$ processes

Ex. $n = 8,\ p = 2$




Block size $= \frac{n}{p} = 4$

# Block Cyclic Distribution (1)

- A variation of block distribution
- Partition the work into many more blocks $(\alpha p)$ than the number of processes
  - $n$: problem size (size of input data)
  - $p$: # of processes
  - $L$: block size, $= \frac{n}{\alpha p}$ $\{= 1\ to\ \frac{n}{p}\}$
  - $\alpha$: $1 \leq \alpha \leq \frac{n}{p}$
- Distribute blocks in a wraparound fashion
  - Block $b_i \rightarrow P_{i\%p}$ (% is modulo operator)

# Block Cyclic Distribution (2)

$n = 16, p = 4$ processes, $\alpha = 1$

Block size $L = {}^{n}/_{\alpha p} = 4$


What if we do more work from left to right?

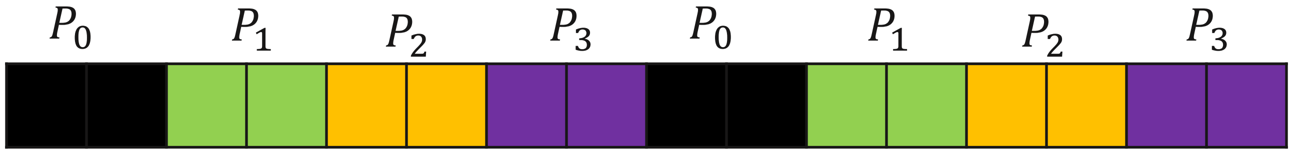
$P_0$ does less work compared with $P_3$

# Block Cyclic Distribution (3)

$n = 16, p = 4$ processes, $\alpha = 2$

Block size $L = {}^{n}/_{\alpha p} = 2$

cyclically distribute

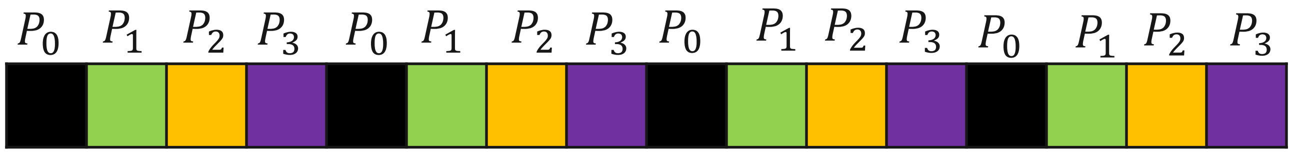



More balanced than $\alpha = 1$

# Block Cyclic Distribution (4)

$n = 16, p = 4$ processes, $\alpha = 4$

Block size $L = {}^{n}/_{\alpha p} = 1$



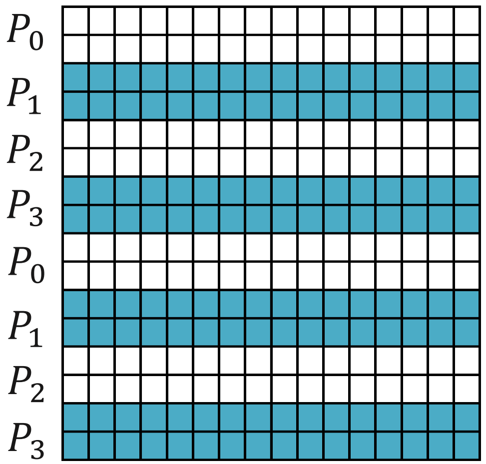

Load balance?

Note: block size $= 1 \longrightarrow$ Cyclic distribution

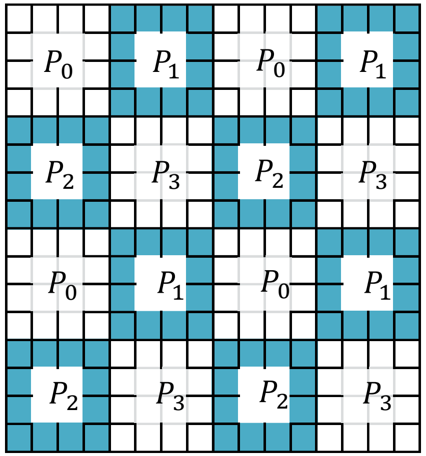
# Example Block Cyclic Distributions

$$p = 4 \text{ processes}$$

$$n \times n = 16 \times 16$$




(a) One dimensional distribution




(b) 2 dimensional distribution

# Block Cyclic Distribution

Array size $n \times n$

Number of processes $= p$

Block size

$1$ — Cyclic

$n/p$ — Block distribution

$\longleftarrow$ Block-cyclic $\longrightarrow$

# Graph Partitioning (1)

Array data distribution (ex. Block cyclic distribution)

Suited for dense matrices

Other class of problems:

- Sparse data structures
- Interactions are (input) data dependent
- Interactions are **irregular**

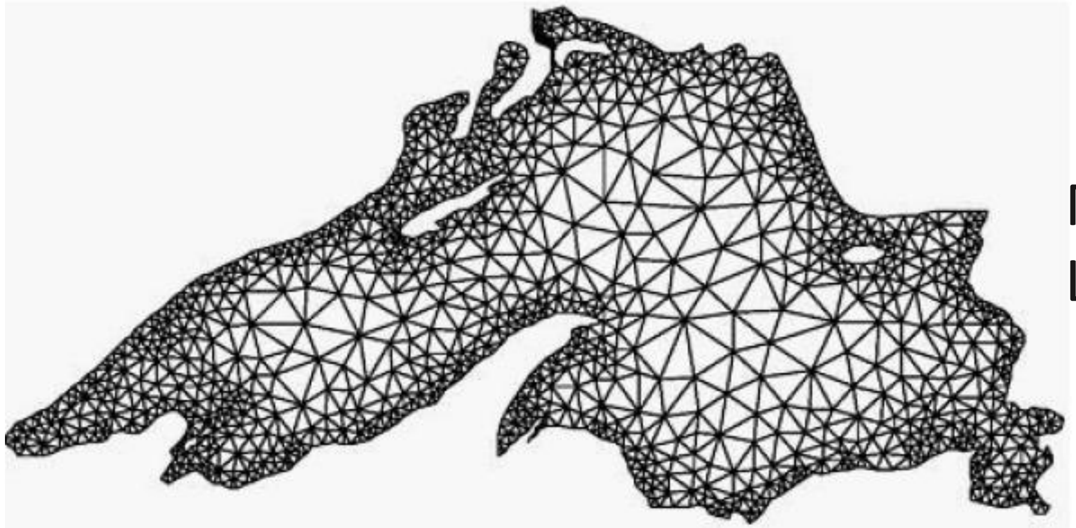
⇨ Represented using a (undirected) graph

# Graph Partitioning (2)

## Example

(Numerical) Simulation of physical phenomenon

Physical domain ⟶ Discretized
Mesh of elements

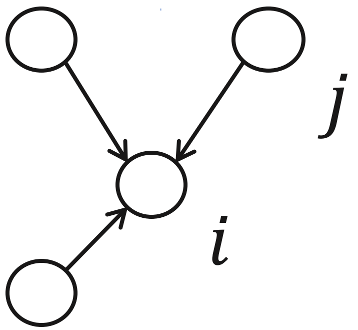


Model of
Lake Superior

Each Point ⇝ Physical attributes
(eg. Temp, chemical composition, water flow value, …)

# Graph Partitioning (3)

## General Computation




Values at node $i$ at time $t+1$

$= f\{\text{Value at time } t \text{ at node } j \mid j \text{ is adjacent to } i\}$

Note: Computation at each node is the same

# Graph Partitioning (4)

Partition

Graph $\rightsquigarrow$ $p$ Processes

Such that

- Load balance
  (approx. same number of nodes in each partition)

- Reduce communication cost
  Eg. # of edges between partitions

In general, computationally expensive (NP-hard, NP-complete)

Classic problem, many heuristics See METIS software

# Mapping based on Task Partitioning (1)

Task-dependency graph (TDG) $\xrightarrow{\text{map}}$ $p$ Processes
(node weights , edge weights)

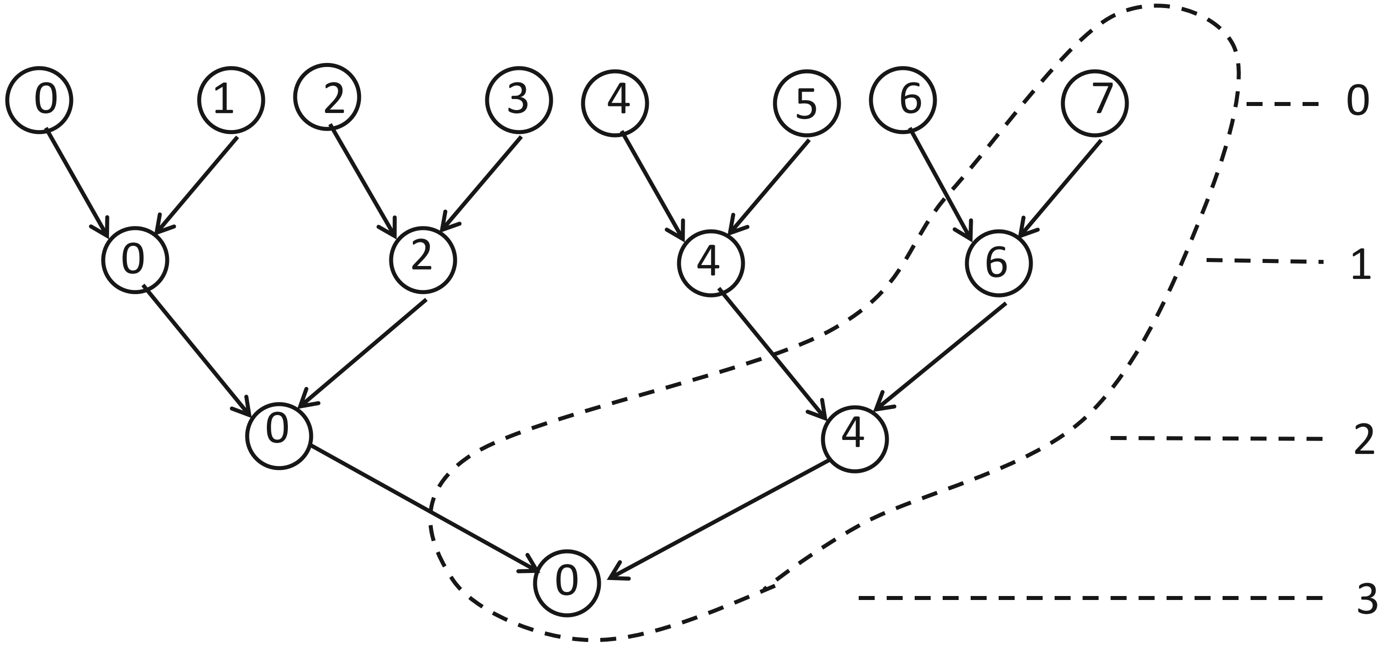
Goals: Load balance

Reduce Idle time

Reduce interaction (communication) time

# Mapping based on Task Partitioning (2)

Process 0




Heuristic: Level by level ordering

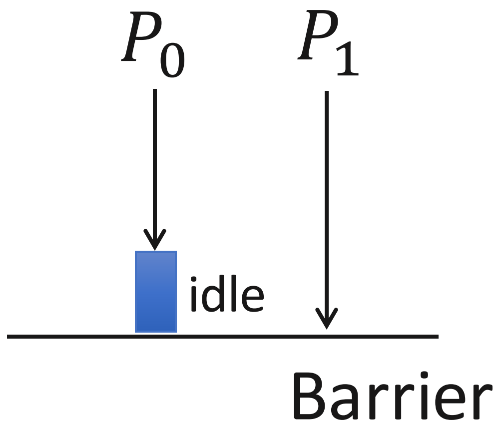
Assign to processes

# Mapping for Load Balancing

Computation $\longrightarrow$ Tasks

$\Downarrow$ map

Processes




## Objectives

- Reduce overhead
- Reduce idle time
- Reduce inter process communication
- Evenly distribute the load among the processes

# Static Mapping (1)

Mapping: Tasks $\longrightarrow$ Processes

performed before the execution

of the algorithm begins

Use Task-dependency graph or

Task-interaction graph

$+$ Meta-data

$\rightarrow$ Task size

Data size

Most problems are computationally expensive (NP-Complete)

# Static Mapping (2)

In most cases,

Static mapping of tasks ⟷ Decomposition based on data partitioning

Many scientific computations (eg. Dense matrix algebra)

- Computations are known (statically) at compile time
- Dependency graph is “static”
- Data decomposition is useful in achieving high performance
- Owner-computes rule

# Dynamic Mapping (1)

Static mapping may not be effective:

- May lead to unbalanced load
- Task interactions are data dependent

# Dynamic Mapping (2)

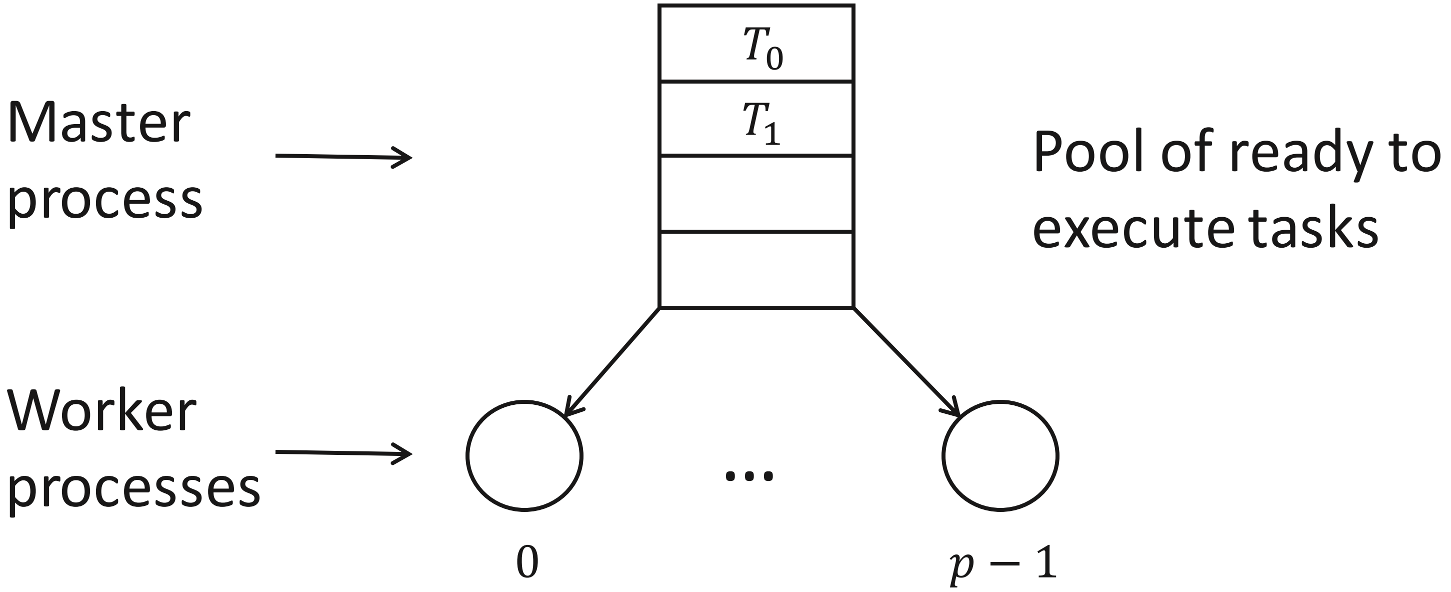
Distribute tasks among the processes at runtime (during execution of the algorithm)

May also move data

Run-time system monitors and performs dynamic task assignment

# Dynamic Mapping (3)

## **Centralized scheme**

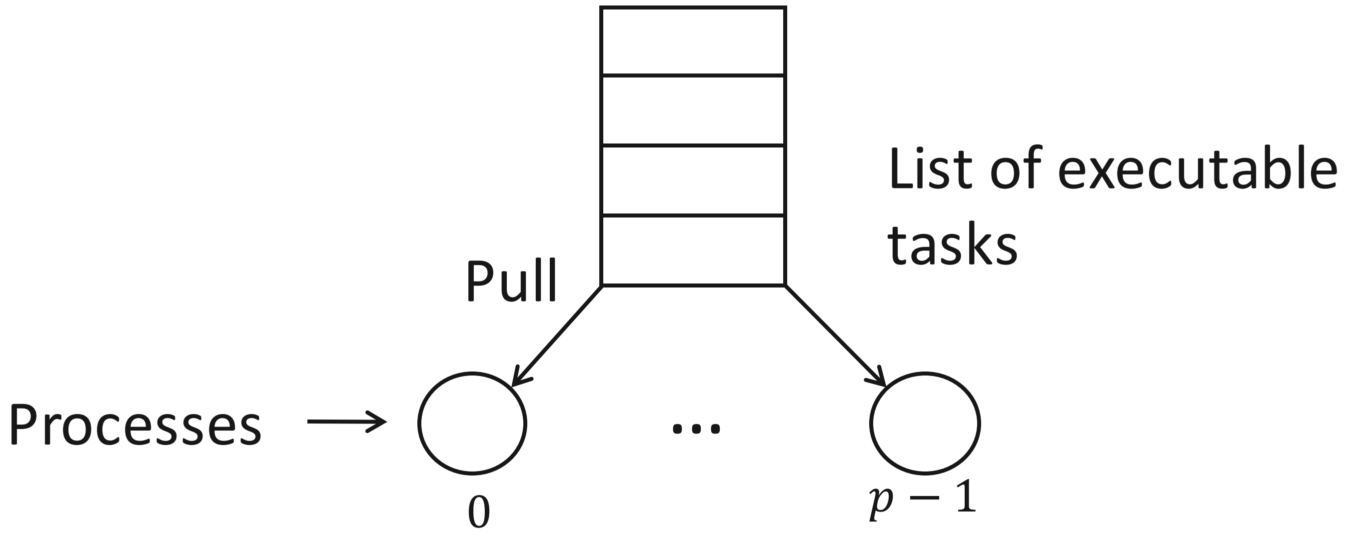



Master process is responsible to map tasks to processes

Can become the bottleneck

# Dynamic Mapping (4)

## Self scheduling

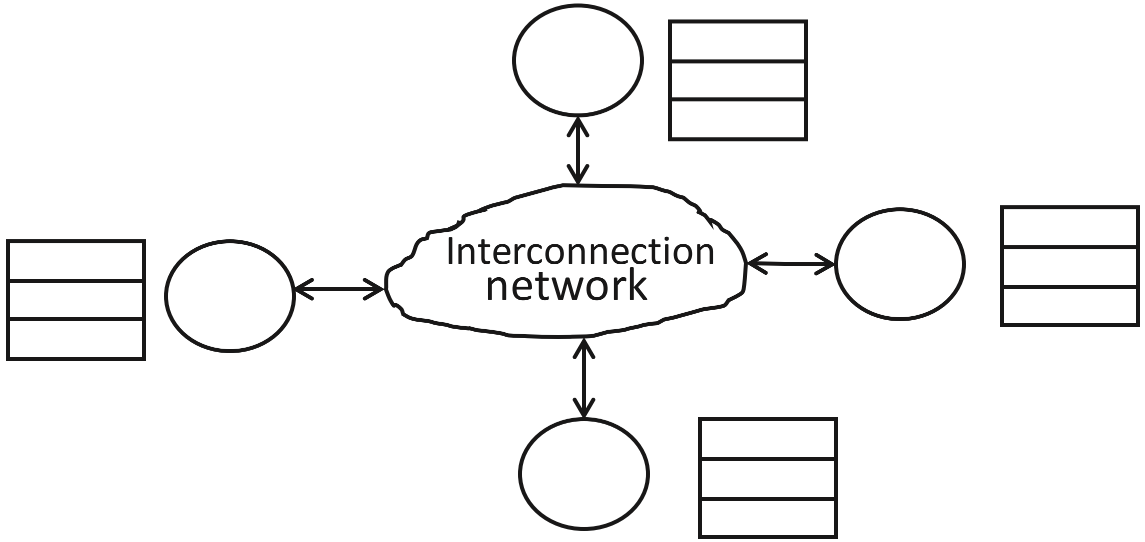



When a process becomes idle (completes execution), access the list and fetch a task to execute

Used for scheduling independent iterations of a loop

# Dynamic Mapping (5)

## Distributed scheme




Processes maintain list of executable tasks

Interact to send/receive work to balance work load

Peer-to-peer system

# Parallel Algorithm Models (1)

A parallel algorithm model is a way of structuring a parallel algorithm by selecting a decomposition and mapping technique and applying appropriate strategy to minimize interactions

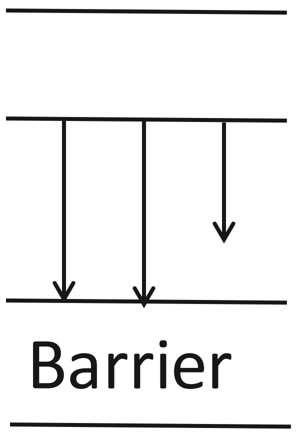
Parallel algorithm $\longrightarrow$ Tasks + Interactions + Data decomposition

$\big\downarrow$ Map

Processes + Interactions

# Parallel Algorithm Models (2)

## Data-parallel model (1)




Tasks $\longrightarrow$ processes

Similar tasks operate (in parallel) on different sets of data

- Data partitioning is important to achieve good performance
- Static mapping

# Parallel Algorithm Models (3)

## Data-parallel model (2)

Examples:

- $$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \leftarrow \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

  Block MM
  4 processes

  $$C_{11} \downarrow \quad \downarrow \quad \downarrow \quad \downarrow C_{22}$$

- SIMD execution model

  Single Instruction Multiple Data

  Synchronous model

# Parallel Algorithm Models (4)

**Task Parallel Model**

**(Task graph model)**

Problem

↓

Decomposition

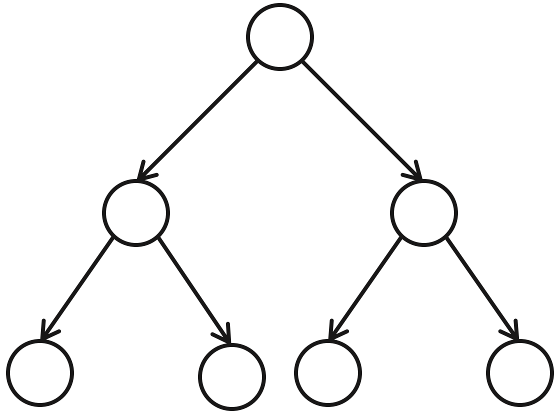
↓

Task graph
Interaction graph

↓

Exploit parallelism in the graph representation
(mapping + scheduling)

Ex. Sorting



# Parallel Algorithm Models (5)

## Work Pool Model

Collection of tasks

Any task can be performed by any process

Dynamic mapping of tasks to processes for load balancing

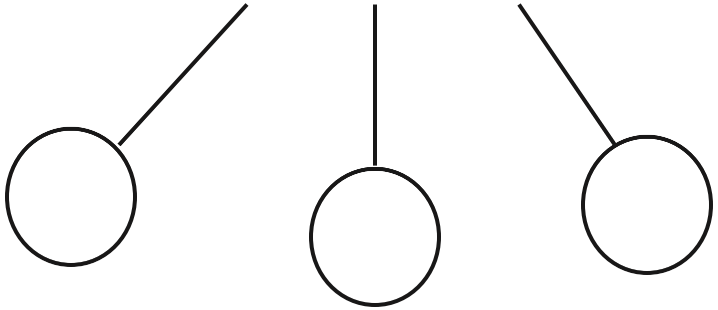
Note: Work pool can be centralized or distributed

Typically: (Small data, large amount of work) per task

# Parallel Algorithm Models (6)

## Master-Worker Model

Master process ← Generate tasks + allocate



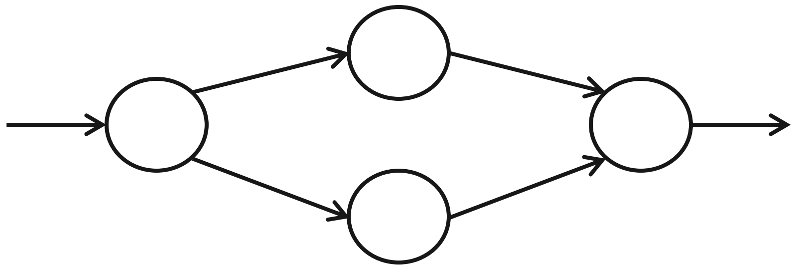
Worker processes

# Parallel Algorithm Models (7)

Producer-Consumer Model

(Pipeline model)

Data (intermediate results) is passed through a series of processes



**Stream parallelism**

Arrival of data triggers the process

Throughput oriented implementation

# Summary

- Parallel Algorithm design
- Data decomposition
- Block Cyclic distribution
- Graph partitioning
- Mapping
  - Static mapping
  - Dynamic mapping
- Parallel algorithm models
  - Task parallel model
  - Work pool model
  - Master-Worker model
  - Producer-Consumer model