CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

CSDS 451: Designing High Performant Systems for AI

Lecture 21

11/11/2025

Sanmukh Kuppannagari

sanmukh.kuppannagari@case.edu

https://sanmukh.research.st/

Case Western Reserve University

Outline

- Distributed Matrix Multiplication on Cluster
- Collective Communication Primitives

Announcements

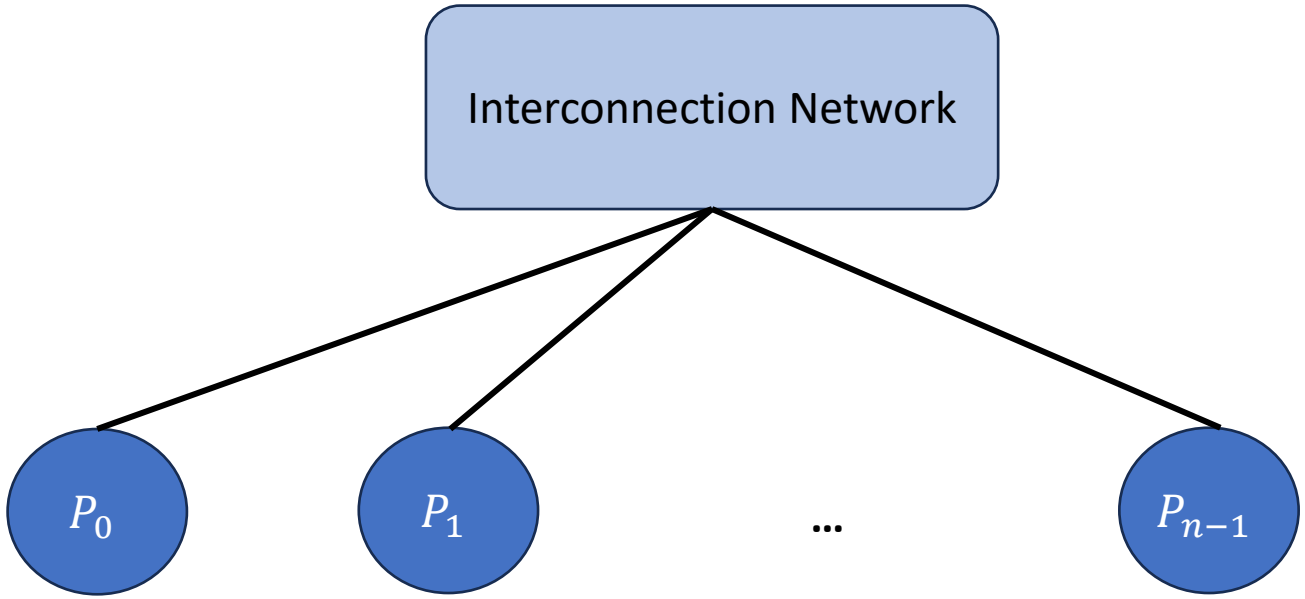
- WA3 Due this Saturday
- WA4 Will be Out
- Project Proposals Graded
  - If you did not receive a score, I may have asked you followup questions on your proposal
  - Please send me weekly updates by responding to the email so that you remain on track

Outline

- Distributed Matrix Multiplication on Cluster
- Collective Communication Primitives

Cluster of Accelerators

- N processors (memory + accelerator)
  - Local compute
  - Local memory
- Connected using an Interconnection Network
- Communication through Message Passing



Anatomy of an MPI Program

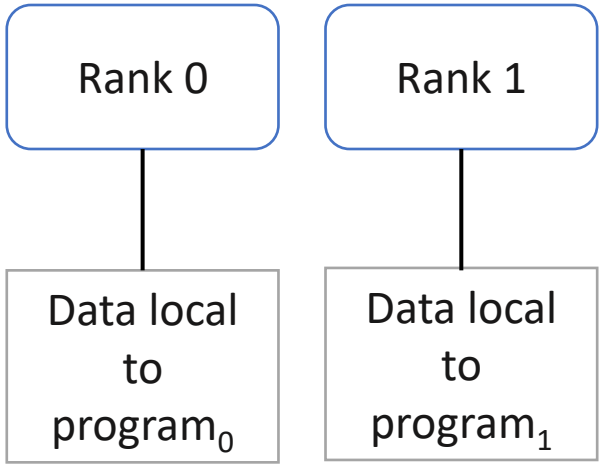
- MPI_Init(…)
- MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)
- MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

- Do rank specific work

Inter-Process Communication

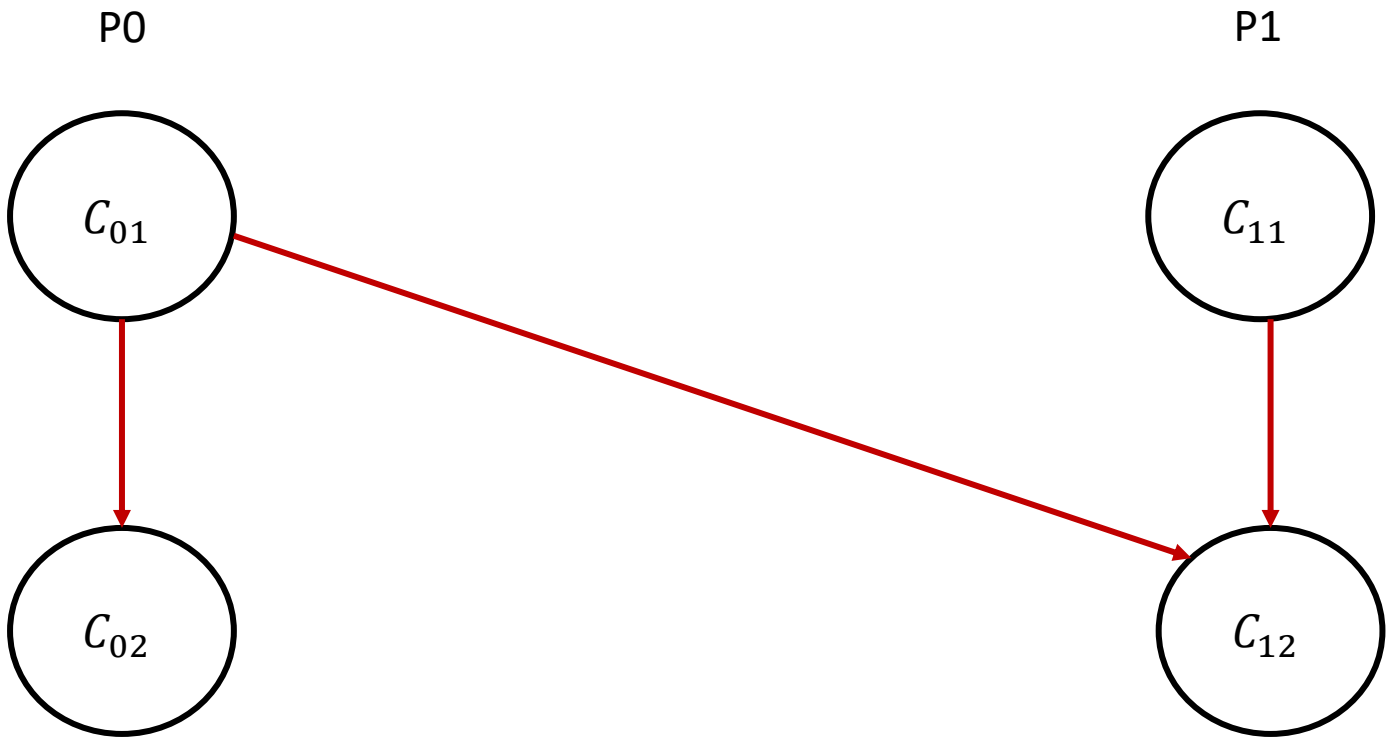
MPI_Init(…)
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

If (rank == 0) {
    $D_0 = C_{00};$
    Send(D_0, size, P1);
    C_{01} ; }
Else {
    $C_{11};$
    Receive(D_1, size, P1);
    $C_{12}(D_1)$; }



Two processor world

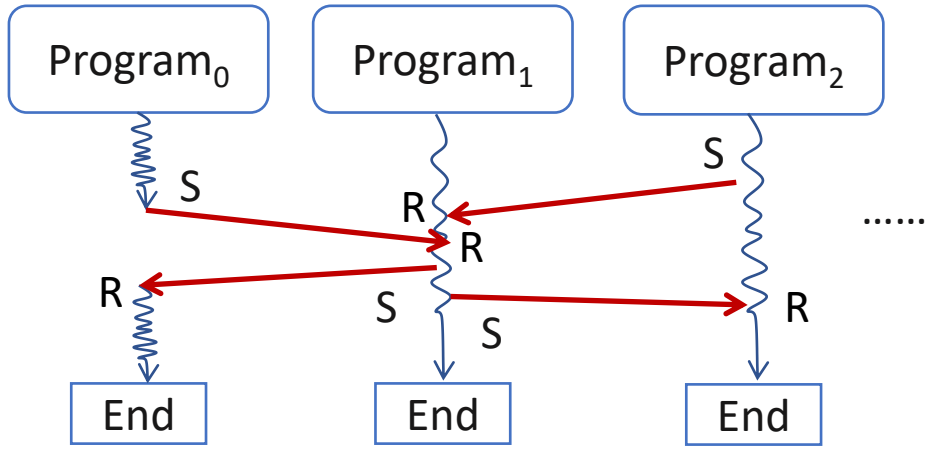
Inter-Process Communication



Dependency across processors due to communication operation

Message Passing Program

Most General Model: Asynchronous

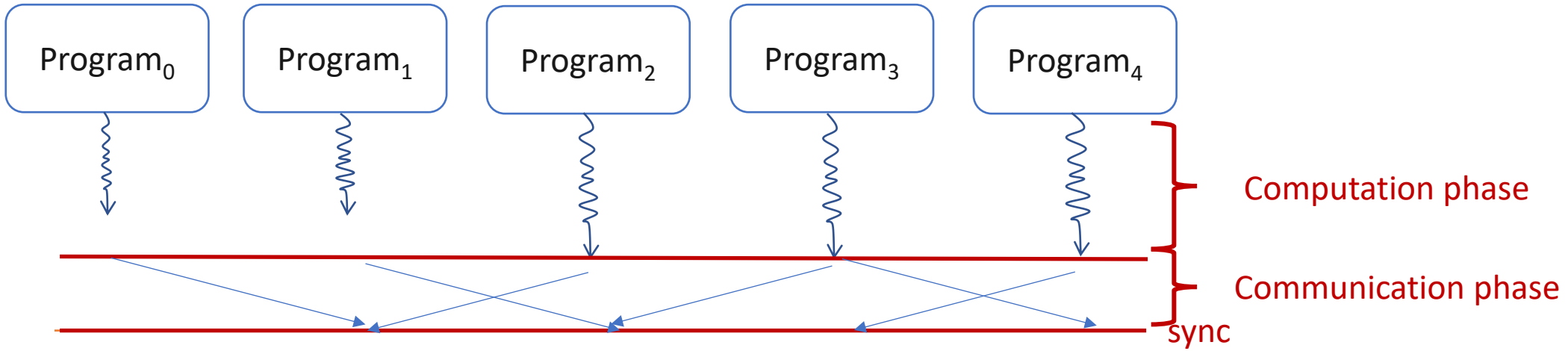


S = send
R = receive

- No structure with respect to instructions, interactions
- No global clock
- Execution is asynchronous
- Programs $0,1,\ldots,p-1$ can be all distinct
- Hard to write/debug

Message Passing Program

Bulk synchronous



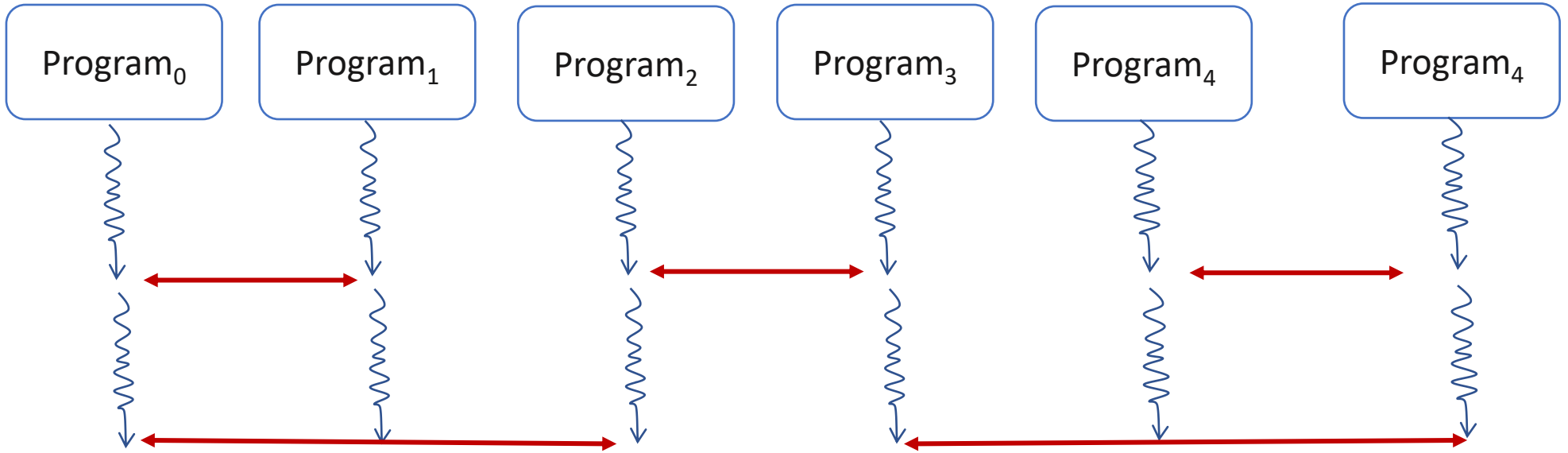
Two phases to the Program

- Computation Phase: Each process executes independently. No communication
- Communication Phase: Processes communicate with each other

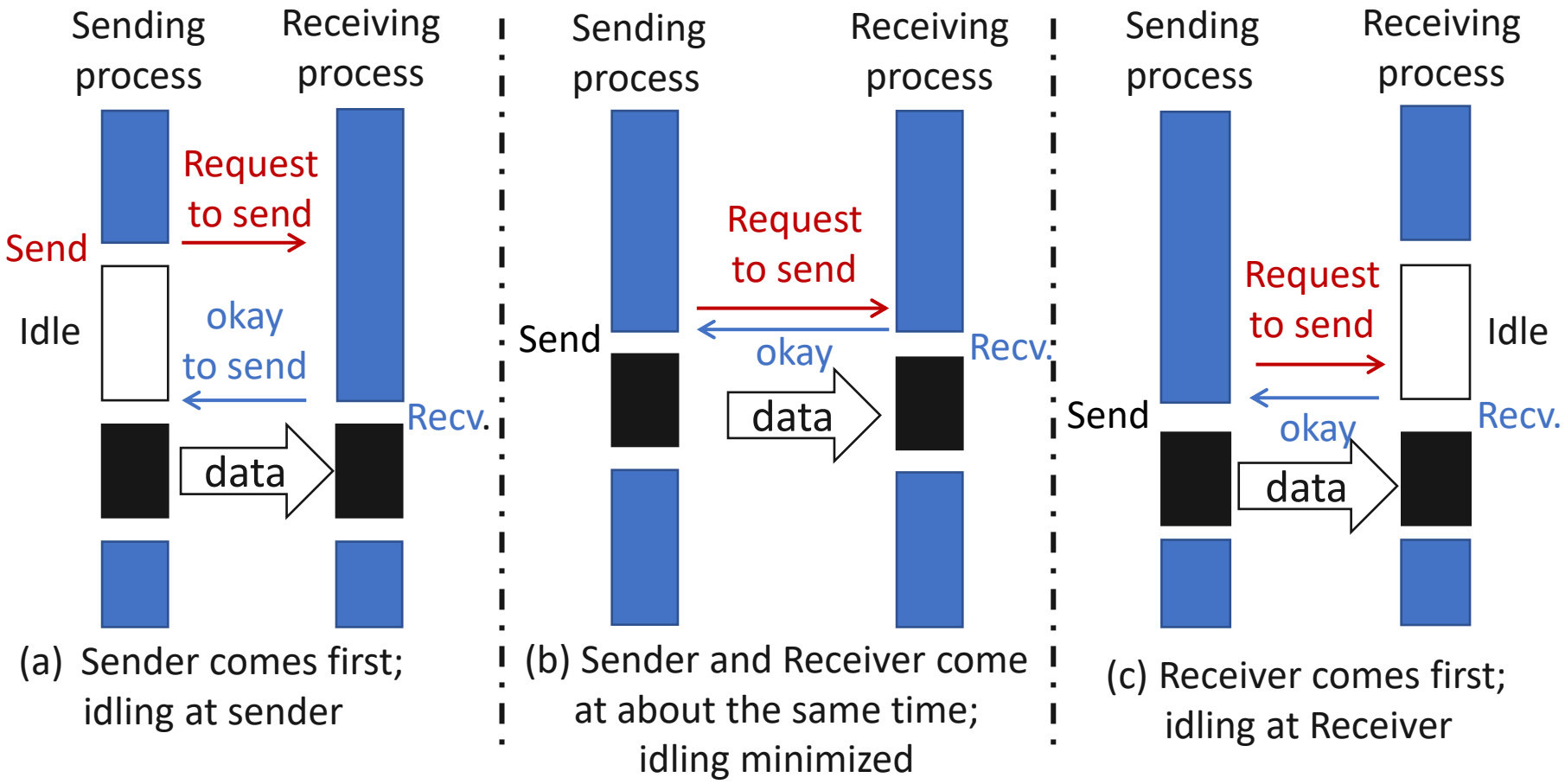
Message Passing Program

SPMD (Single Program Multiple Data)

- Code is same in all the processes except for initialization
- Restrictive model, easy to write and debug
- Easy to do performance analysis
- Widely used in Machine Learning Training

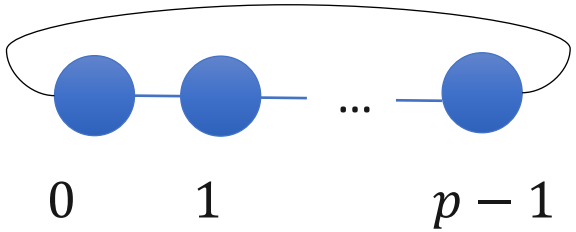


Blocking Non-Buffered Send/Receive

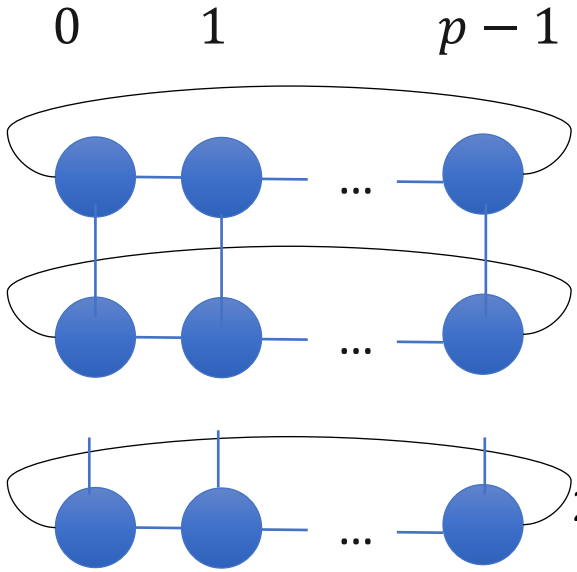


Virtual Topology

- Define the “connectivity pattern” of the processors
- Helps us in developing more intuitive notions of message passing algorithms
- Think of it as 1D versus 2D arrays. It will be hard to visualize matrix multiplication if we write algorithms using 1D arrays



1D Mesh



2D Mesh

Message Passing Programming Paradigm

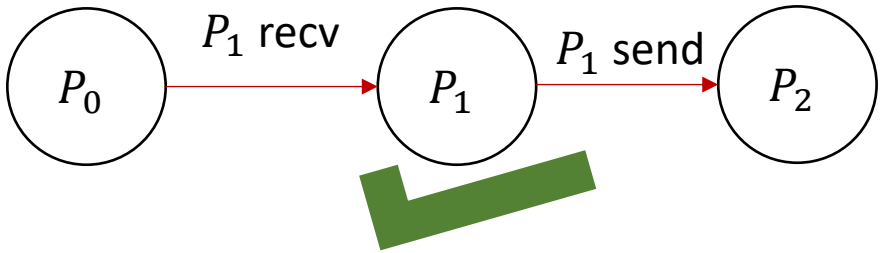
- User Specifies the following
  - Concurrency Model (BSP, SPMD, None/Asynchronous)
    - Processes: The number of processes and the work performed for each process
    - Send, receive that enable data (we will only use blocking non-buffered semantics)
  - A virtual topology of the processes

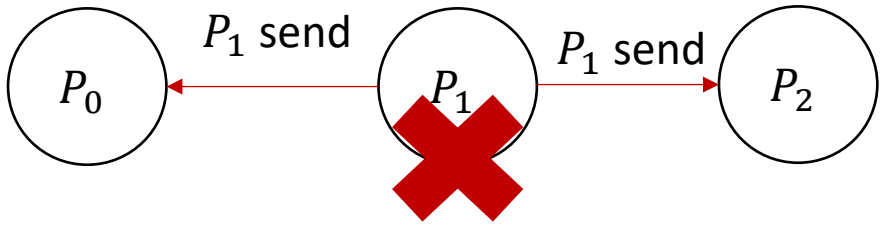
- We will discuss it at algorithmic level.
  - Skip initialization, rank calculation, etc.

Message Passing Programming Paradigm

- Note: We make the following assumption for simplicity

- Each processor can communicate with a maximum of two processor at any given time
  - One communication – send
  - Other communication - receive





Adding Using Message Passing on 1D Mesh (6)

- Message Passing Algorithm (SPMD model)

Program in process $j, 0 \leq j \leq n-1$

1. Do $i = 0$ to ??
2. If $j = k \cdot 2^{i+1} + 2^i$, for some $k \in N$
3. **Send $A(j)$ to process $j - 2^i$**
4. Else if $j = k \cdot 2^{i+1}$, for some $k \in N$
5. **Receive $A(j + 2^i)$ from process $j + 2^i$**
6. $A(j) \leftarrow A(j) + A(j + 2^i)$
7. End
8. **Barrier**
9. End

2^i distance communication

Note:
$A(j)$ is local to process j
N = set of natural numbers = {0, 1, …}

Performance Analysis (1)

- Total Computation time: Number of rounds × computation time per round
  - Note each processor is doing the same computation in each round
- Computation time per round – Calculate using accelerator model – GPU/Systolic array
  - Computation time per round: $O(1)$
- Number of rounds - $\log N$
- Total Computation time - $O(\log N)$

Performance Analysis (2)

- Total Communication time ???
- Depends upon the underlying interconnection topology
- We will assume fully connected in this class
  - A transfer of k data items from any processor to any processor takes $O(k)$ amount of time.
  - Assuming t_w as the per word transfer time, this is equal to $k \times t_w$
  - Note: Actual Interconnection modeling is much more complicated and could be a lecture or 2 in itself

Performance Analysis (3)

- Total Communication time - Number of rounds $\times$ communication time per round
- Number of rounds - $\log N$
- Communication time per round - $1 \times t_w$
- Total Communication time: $\log N \times t_w$

Performance Analysis (4)

- Challenge: Does the system have enough communication bandwidth to support the communication requirement of the algorithm?
- Maximum Communication Requirement – ??
- Note: Iteration 0 has the most $(N/2)$ processors active
- Maximum communication requirement = $\frac{N}{2}.1 < B$
- B: Maximum bandwidth supported by the system.

Distributed Matrix Multiplication

- Objective: Perform matrix multiplication $C = AB$ on a 2D mesh of processors

- Popular example – Tensor Parallelism in AI training (will discuss in next class)

MM on 2D Mesh

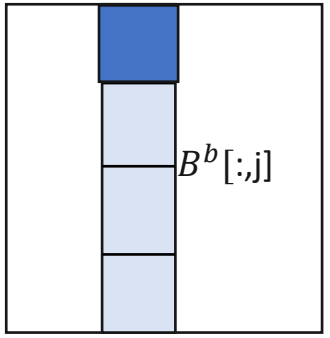
- Assume A, B, C are $n \times n$ matrices
- Concurrency Model: SPMD
- Virtual Topology: 2D mesh of size $\sqrt{p} \times \sqrt{p}$
  - Processors denoted as: P_{ij} $0 \leq i, j < \sqrt{p},\ 1 \leq \sqrt{p} \leq n$
- **Cannon's Algorithm**

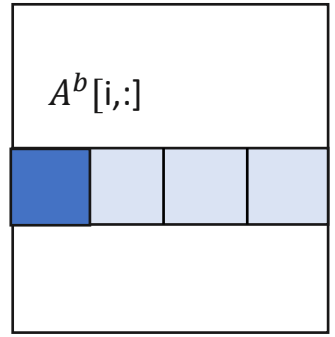
MM on 2D Mesh – Cannon's Algorithm

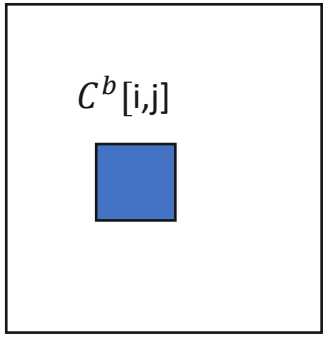
- Similar Idea as Blocked Matrix Multiplication
- Processor P_{ij} produces output block C_{ij}
- What is the block size of C_{ij}?

MM on 2D Mesh – Cannon's Algorithm

- Similar Idea as Blocked Matrix Multiplication
- Processor P_{ij} produces output block C_{ij}
- What is the block size of C_{ij}? $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$
- Do you recall block MM?

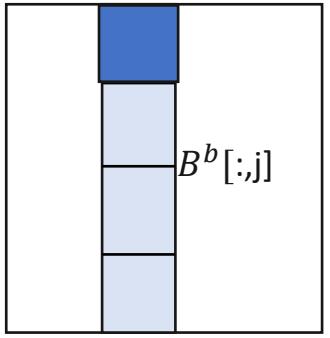


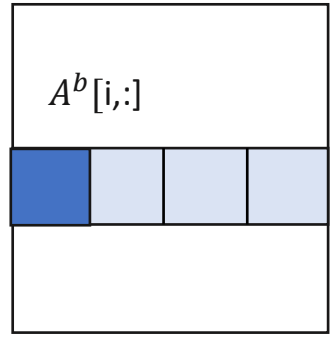


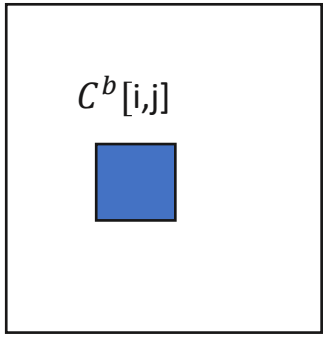


MM on 2D Mesh – Cannon's Algorithm

- Similar Idea as Blocked Matrix Multiplication
- Processor P_{ij} produces output block C_{ij}
- What is the block size of C_{ij}? $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$
- How do we iterate over all blocks?

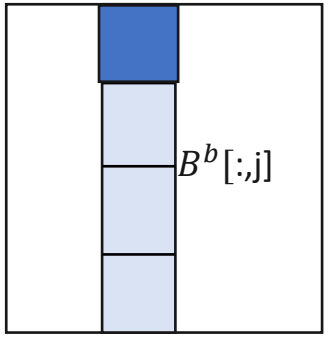


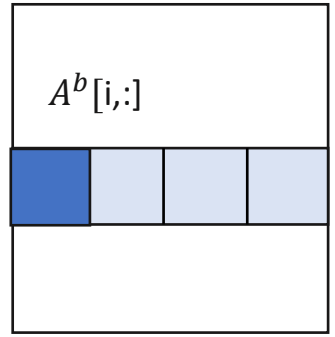


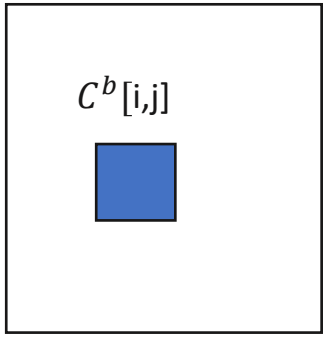


MM on 2D Mesh – Cannon's Algorithm

- Similar Idea as Blocked Matrix Multiplication
- Processor P_{ij} produces output block C_{ij}
- What is the block size of C_{ij}? $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$
- How do we iterate over all blocks? Need to fetch blocks A_{ik} and B_{kj} to processor P_{ij} using Message Passing







MM on 2D Mesh – Cannon's Algorithm

- Key Idea

- Each processor P_{ij} produces block C_{ij}
  - Using Blocks A_i: and $B_{:j}$

- Design an algorithm that can make sure message passing is performed efficiently to achieve the above

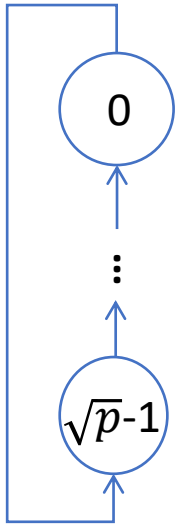
MM on 2D Mesh – Cannon's Algorithm

- For simplicity of discussion, we define two operations
- Circular Left Shift on a row
  - Each processor sends a block of matrix to the processor on the left
- Circular Up Shift on a column
  - Each processor send a block of matrix to the processor above it
- Called by each processor in a row/column

MM on 2D Mesh – Cannon's Algorithm

Circular left shift

Circular up shift



MM on 2D Mesh – Cannon's Algorithm

- Assume initially P_{ij} has data - A_{ij}, B_{ij}

$A_{0,0}$ $B_{0,0}$	$A_{0,1}$ $B_{1,1}$	$A_{0,2}$ $B_{2,2}$	$A_{0,3}$ $B_{3,3}$
$A_{1,1}$ $B_{1,0}$	$A_{1,2}$ $B_{2,1}$	$A_{1,3}$ $B_{3,2}$	$A_{1,0}$ $B_{0,3}$
$A_{2,2}$ $B_{2,0}$	$A_{2,3}$ $B_{3,1}$	$A_{2,0}$ $B_{0,2}$	$A_{2,1}$ $B_{1,3}$
$A_{3,3}$ $B_{3,0}$	$A_{3,0}$ $B_{0,1}$	$A_{3,1}$ $B_{1,2}$	$A_{3,2}$ $B_{2,3}$

MM on 2D Mesh – Cannon's Algorithm

- Step #1: Initial Data Alignment

For $\boldsymbol{A}$:

i^{th} row – circular left shift by i $(0 \leq i < \sqrt{p})$

For $\boldsymbol{B}$:

j^{th} column – circular up shift by j $\left(0 \leq j < \sqrt{p}\right)$

MM on 2D Mesh – Cannon's Algorithm

- Data Distributed after Step 1?

?? ??	?? ??	?? ??	?? ??
?? ??	?? ??	?? ??	?? ??
?? ??	?? ??	?? ??	?? ??
?? ??	?? ??	?? ??	?? ??

MM on 2D Mesh – Cannon's Algorithm

- Data Distributed after Step 1?

- $A_{ij} : P_{i,(j-i)\%\sqrt{p}}$
- $B_{ij} : P_{(i-j)\%\sqrt{p},j}$

- Can you come up with the reverse mapping? (WA4)
  - Processor to block mapping.

$A_{0,0}$ $B_{0,0}$	$A_{0,1}$ $B_{1,1}$	$A_{0,2}$ $B_{2,2}$	$A_{0,3}$ $B_{3,3}$
$A_{1,1}$ $B_{1,0}$	$A_{1,2}$ $B_{2,1}$	$A_{1,3}$ $B_{3,2}$	$A_{1,0}$ $B_{0,3}$
$A_{2,2}$ $B_{2,0}$	$A_{2,3}$ $B_{3,1}$	$A_{2,0}$ $B_{0,2}$	$A_{2,1}$ $B_{1,3}$
$A_{3,3}$ $B_{3,0}$	$A_{3,0}$ $B_{0,1}$	$A_{3,1}$ $B_{1,2}$	$A_{3,2}$ $B_{2,3}$

MM on 2D Mesh – Cannon's Algorithm

- Step #2: Execute Matrix Multiplication

- In processor P_{ij}
- Repeat $\sqrt{p}$ times
  - Perform $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ matrix multiplication using **local** $\boldsymbol{A}$, $\boldsymbol{B}$ blocks
  - Add the result to C_{ij}
  - Circular_Left_Shift – blocks of A across the row i
  - Circular_Up_Shift – blocks of B across the column j

MM on 2D Mesh – Cannon's Algorithm

Illustration
(4×4 matrix,
4×4 processor array)

$A_{0,0}$ $B_{0,0}$	$A_{0,1}$ $B_{1,1}$	$A_{0,2}$ $B_{2,2}$	$A_{0,3}$ $B_{3,3}$
$A_{1,1}$ $B_{1,0}$	$A_{1,2}$ $B_{2,1}$	$A_{1,3}$ $B_{3,2}$	$A_{1,0}$ $B_{0,3}$
$A_{2,2}$ $B_{2,0}$	$A_{2,3}$ $B_{3,1}$	$A_{2,0}$ $B_{0,2}$	$A_{2,1}$ $B_{1,3}$
$A_{3,3}$ $B_{3,0}$	$A_{3,0}$ $B_{0,1}$	$A_{3,1}$ $B_{1,2}$	$A_{3,2}$ $B_{2,3}$

After Initial Alignment

Step 1

- Compute product of local blocks
- Circular left shift
- Circular up shift

Repeat

MM on 2D Mesh – Cannon's Algorithm

$A_{0,2}$ $B_{2,0}$	$A_{0,3}$ $B_{3,1}$	$A_{0,0}$ $B_{0,2}$	$A_{0,1}$ $B_{1,3}$
$A_{1,3}$ $B_{3,0}$	$A_{1,0}$ $B_{0,1}$	$A_{1,1}$ $B_{1,2}$	$A_{1,2}$ $B_{2,3}$
$A_{2,0}$ $B_{0,0}$	$A_{2,1}$ $B_{1,1}$	$A_{2,2}$ $B_{2,2}$	$A_{2,3}$ $B_{3,3}$
$A_{3,1}$ $B_{1,0}$	$A_{3,2}$ $B_{2,1}$	$A_{3,3}$ $B_{3,2}$	$A_{3,0}$ $B_{0,3}$

After Initial Alignment

Step 2

- Compute product of local blocks
- Circular left shift
- Circular up shift

Repeat

MM on 2D Mesh – Cannon's Algorithm

$A_{0,2}$ $B_{2,0}$	$A_{0,3}$ $B_{3,1}$	$A_{0,0}$ $B_{0,2}$	$A_{0,1}$ $B_{1,3}$
$A_{1,3}$ $B_{3,0}$	$A_{1,0}$ $B_{0,1}$	$A_{1,1}$ $B_{1,2}$	$A_{1,2}$ $B_{2,3}$
$A_{2,0}$ $B_{0,0}$	$A_{2,1}$ $B_{1,1}$	$A_{2,2}$ $B_{2,2}$	$A_{2,3}$ $B_{3,3}$
$A_{3,1}$ $B_{1,0}$	$A_{3,2}$ $B_{2,1}$	$A_{3,3}$ $B_{3,2}$	$A_{3,0}$ $B_{0,3}$

After Initial Alignment

Step 3

- Compute product of local blocks
- Circular left shift
- Circular up shift

Repeat

MM on 2D Mesh – Cannon's Algorithm

$A_{0,3}$ $B_{3,0}$	$A_{0,0}$ $B_{0,1}$	$A_{0,1}$ $B_{1,2}$	$A_{0,2}$ $B_{2,3}$
$A_{1,0}$ $B_{0,0}$	$A_{1,1}$ $B_{1,1}$	$A_{1,2}$ $B_{2,2}$	$A_{1,3}$ $B_{3,3}$
$A_{2,1}$ $B_{1,0}$	$A_{2,2}$ $B_{2,1}$	$A_{2,3}$ $B_{3,2}$	$A_{2,0}$ $B_{0,3}$
$A_{3,2}$ $B_{2,0}$	$A_{3,3}$ $B_{3,1}$	$A_{3,0}$ $B_{0,2}$	$A_{3,1}$ $B_{1,3}$

After Initial Alignment

Step 4

- Compute product of local blocks
- Circular left shift
- Circular up shift

End

Performance Analysis (1)

- Total Computation time = Number of round × time per round
- Number of rounds = ? ?
- Time per round = ? ?
- Total Computation time = ? ?

Performance Analysis (1)

- Total Computation time = Number of round × time per round
- Number of rounds = $\sqrt{p}$
- Time per round = $O(\frac{n^3}{\sqrt{p^3}})$
- Total Computation time = $O(\frac{n^3}{p})$

Performance Analysis (2)

- Total Communication time = Number of round × Communication time per round
- Number of rounds = ??
- Communication time per round on a Fully connected Network - ??
- Total Communication time = ??

Performance Analysis (2)

- Total Communication time = Number of round × Communication time per round
- Number of rounds = $\sqrt{p}$
- Communication time per round on a Fully connected Network - $O\left(\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}\right)$
- Total Communication time = $O\left(\frac{n^2}{\sqrt{p}}\right)$

Performance Analysis (3)

- Communication Bandwidth Needed on a Fully Connected Network

- Total data transferred in each communication step

??

To ensure communication is not a bottleneck, system bandwidth $B >$??

Performance Analysis (3)

- Communication Bandwidth Needed on a Fully Connected Network

- Total data transferred in each communication step

$$2.\frac{n}{\sqrt{p}}.\frac{n}{\sqrt{p}}p = O(n^2)$$

To ensure communication is not a bottleneck, system bandwidth $B > O(n^2)$

Performance Analysis (4)

- In other words, the largest size of matrix that can be solved using p processors on this system, using this algorithm is:

??

Performance Analysis (5)

- In other words, the largest size of matrix that can be solved using p processors on this system, using this algorithm is:

$$\sqrt{B} \times \sqrt{B}$$

Outline

- Distributed Matrix Multiplication on Cluster
- Collective Communication Primitives

Collective Communication (1)

- Writing programs with send/receive can get too cumbersome
  - Difficult to optimize
- If every process needs to be involved in communication, a higher level of abstraction is desired
- Collective Communication API: Communications involving group of processors

Collective Communication (2)

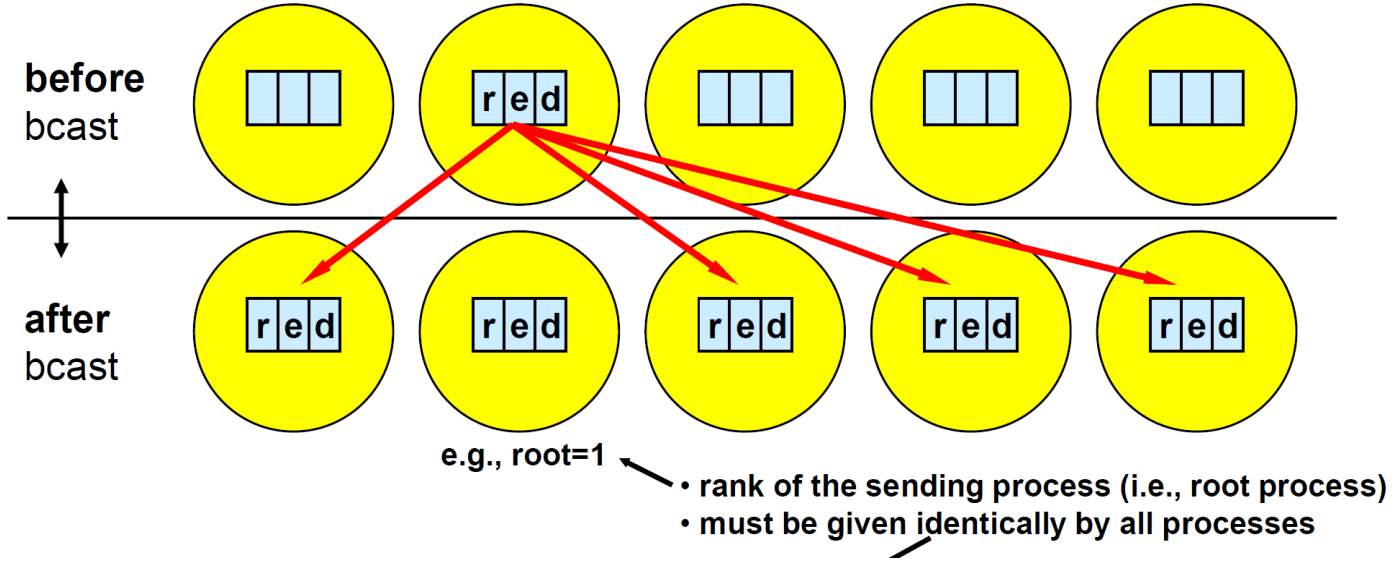
- Called by **ALL** the processors
  - Each processor should call the same API with same size of buffer
  - If a call is missing in one of the processors, it may lead to undefined behavior
- One program in **EACH** processor reaches the call, the communication occurs
- Examples
  - Barrier Synchronization (usually not used because other communication operations lead to synchronization)
  - Broadcast, Scatter, Gather, …
  - Reduction, All_reduce,…

MPI_Broadcast

- MPI_Broadcast(buffer, size, root)

Note: ALL processors are calling this function

- Send the data stored in **buffer** at the **root** processor to all the processors



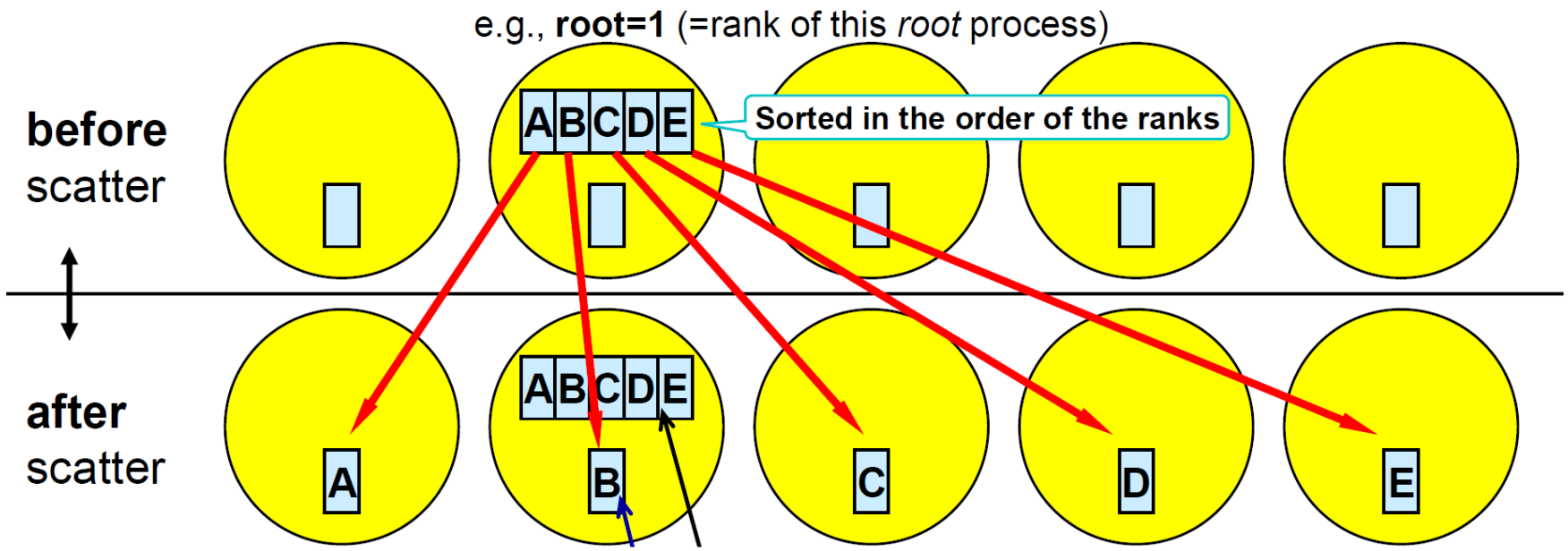
MPI_Scatter

- MPI_Scatter(sendbuffer, sbsize, recvbuffer, rbsize, root)
- Assuming P processors
- Objective: Distributed an array of size $P \times c$ across the P processors
  - Each processor receives c elements

MPI_Scatter

- MPI_Scatter(sendbuffer, sbsize, recvbuffer, rbsize, root)
- Sendbuffer contains sbsize elements
- A processor with rank i will receive elements in the range: $i \times sbsize/P$ to $(i+1) \times \frac{sbsize}{P} - 1$
- Rbsize = sbsize/P

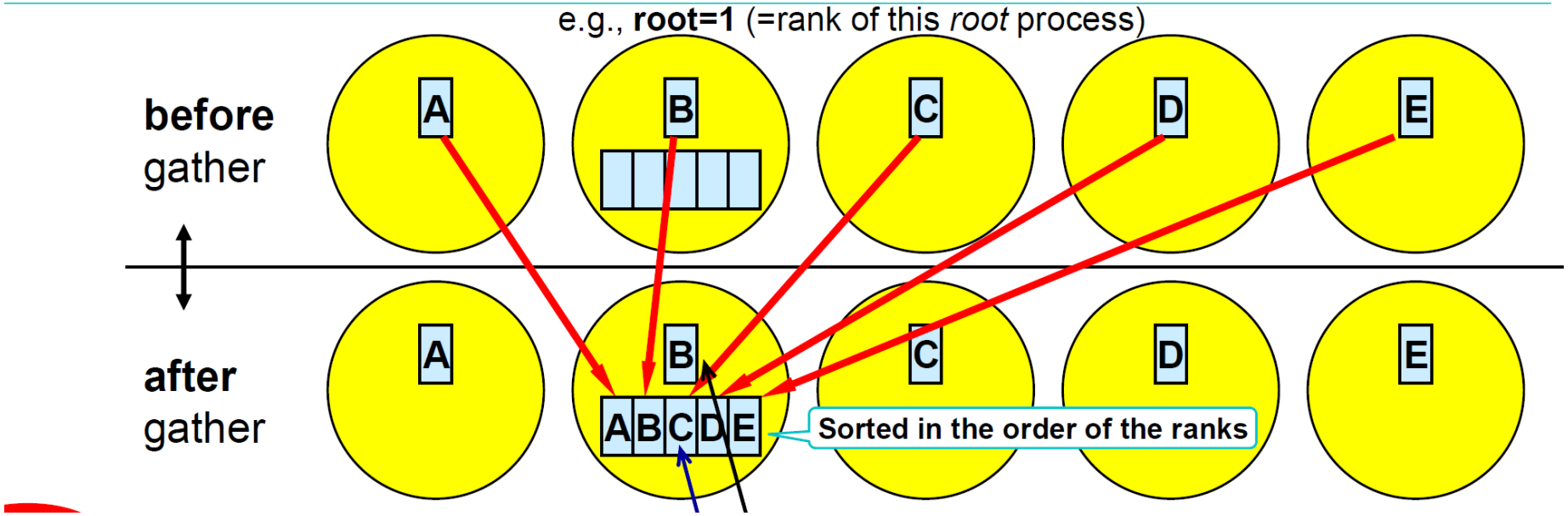
MPI_Scatter


e.g., root=1 (=rank of this root process)
before scatter
after scatter
A B C D E
Sorted in the order of the ranks
A B C D E
A
B
C
D
E

MPI_Gather

- MPI_Gather(sendbuffer, sbsize, recvbuffer, rbsize, root)
- Inverse of MPI_Scatter
- Assuming P processors, each processor has sbsize $= c$ elements in in their sendbuffer
- Objective: collect the $c \times P$ elements (from all processors) into a single recvbuffer of size rbsize = $P \times$sbsize at the root

MPI_Gather


e.g., root=1 (=rank of this root process)
before gather
after gather
A
B
C
D
E
A
B
C
D
E
A B C D E
Sorted in the order of the ranks

MPI_Reduce

- MPI_Reduce(inbuf, ibsize, resultbuf, rbsize, Aggr_op, root)
- Performs a global reduction operation with a specified aggregation function and stores the result at the root
- Inbuf: buffer that stores elements in each processor that will be aggregated
- Resultbuf: buffer in the root processor that stores the aggregated results

MPI_Reduce

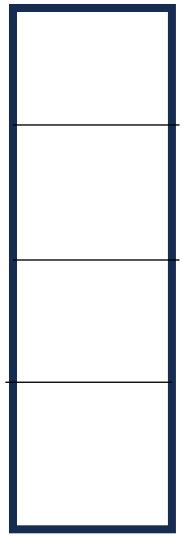
- Examples of Aggregation Operations?

MPI_Reduce

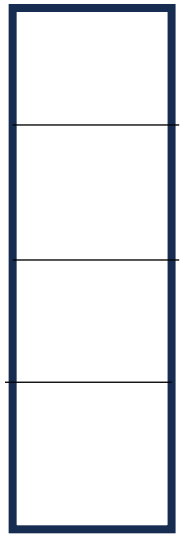
- Examples of Aggregation Operations?

- Sum
- Product
- Min
- Max
- …

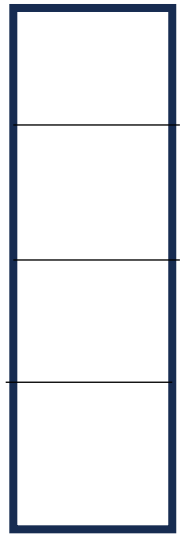
MPI_Reduce



Inbuf P0



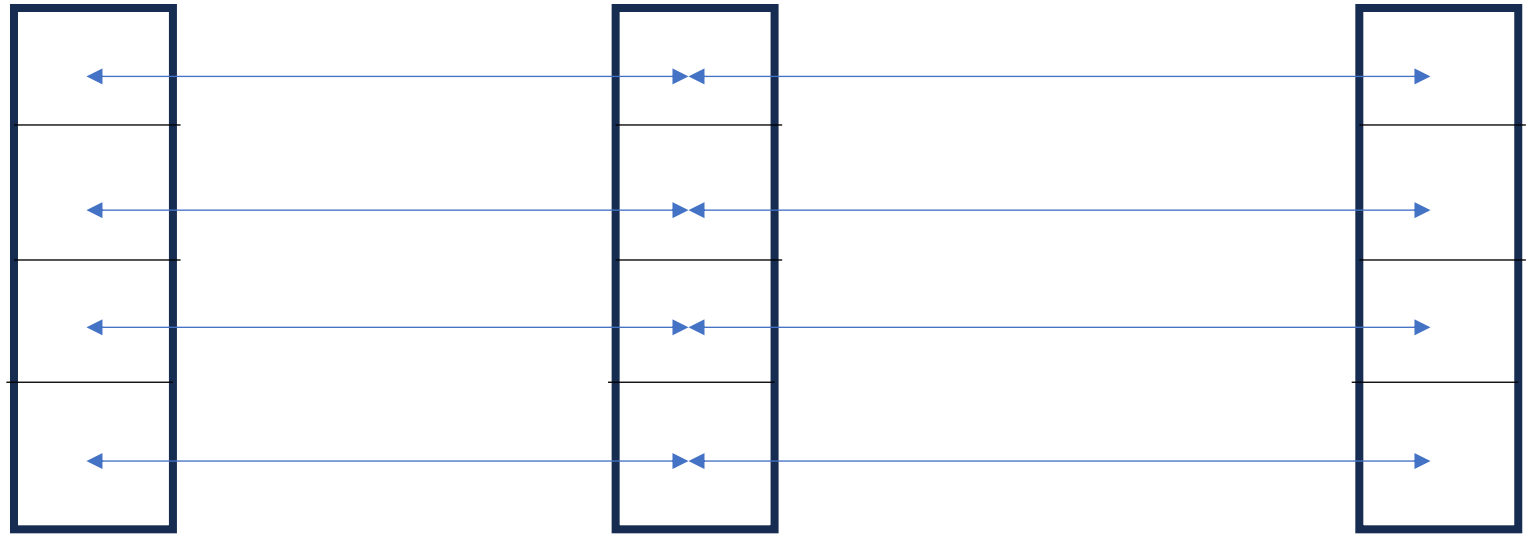
Inbuf P1



Inbuf P_p

Reduction Occurs Elementwise in vectors

MPI_Reduce



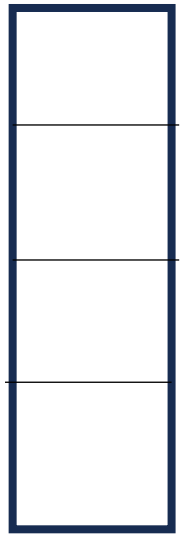
Inbuf P0

Inbuf P1

Inbuf P_p

Reduction Occurs Elementwise in vectors

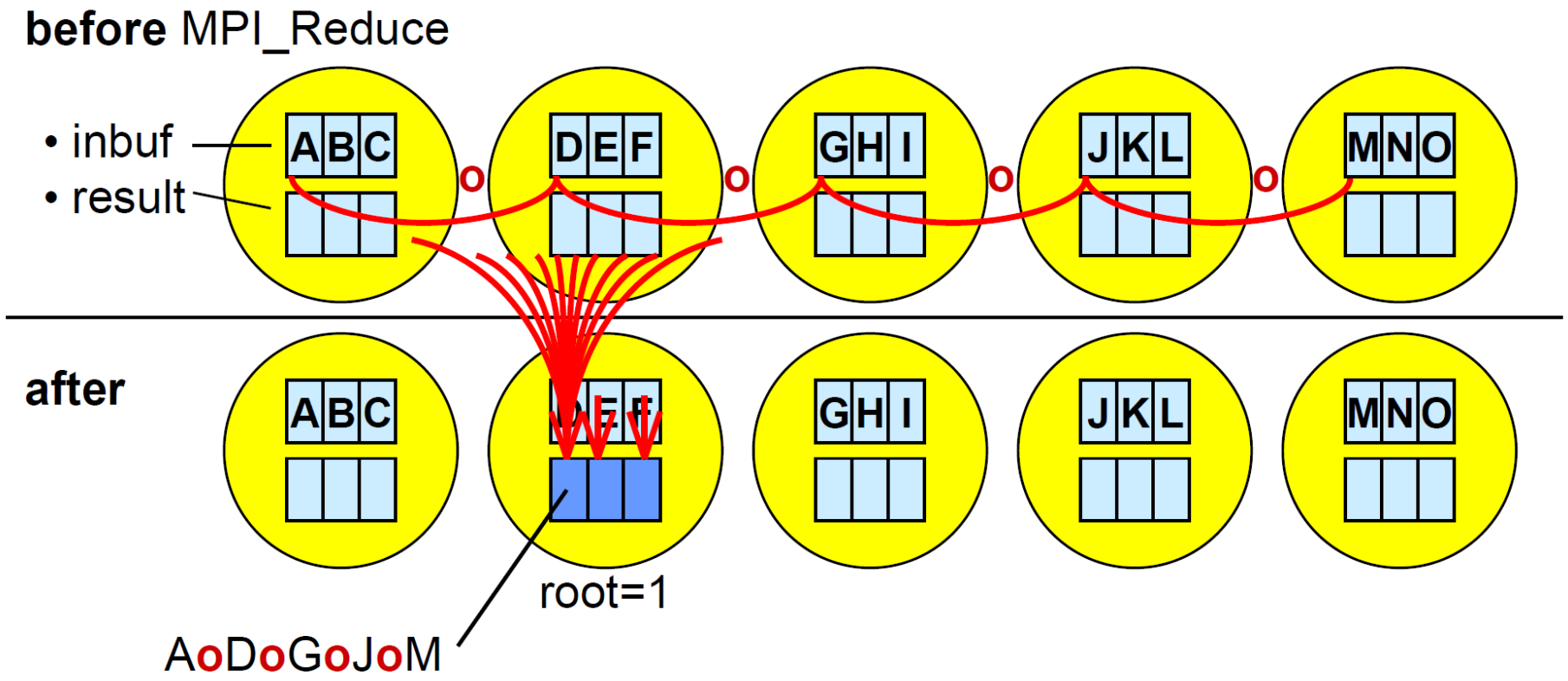
MPI_Reduce



ResultBuf root-processor

Reduction Occurs Elementwise in vectors

MPI_Reduce


before MPI_Reduce
• inbuf
• result
A B C
D E F
G H I
J K L
M N O
o
o
o
o
after
A B C
G H I
J K L
M N O
root=1
AoDoGoJoM

MPI_Reduce

- Does the order of Rank matter in Reduction?
- P0: A, P1: B, P2: C
- P0: B P1: C, P2: A
- MPI_Reduce() – same results or different?

MPI_Reduce

- Does the order of Rank matter in Reduction?
- P0: A, P1: B, P2: C
- P0: B P1: C, P2: A
- MPI_Reduce() – same results.

MPI_AllReduce

- MPI_AllReduce(inbuf, resultbuf, size)
- Same as MPI_Reduce() but stores the results in all the processors
- Equivalent to:
  - MPI_Reduce(inbuf, ibsize, resultbuf, rbsize, Aggr_op, root)
  - MPI_Scatter(resultbuf, rbsize, root)

Collective Operations Communication Complexity (1)

- Operations are implemented using recursive doubling/divide-conquer type of algorithms

- May ask algorithms and computation of the complexity in WA 4 (with sufficient details for you to proceed)

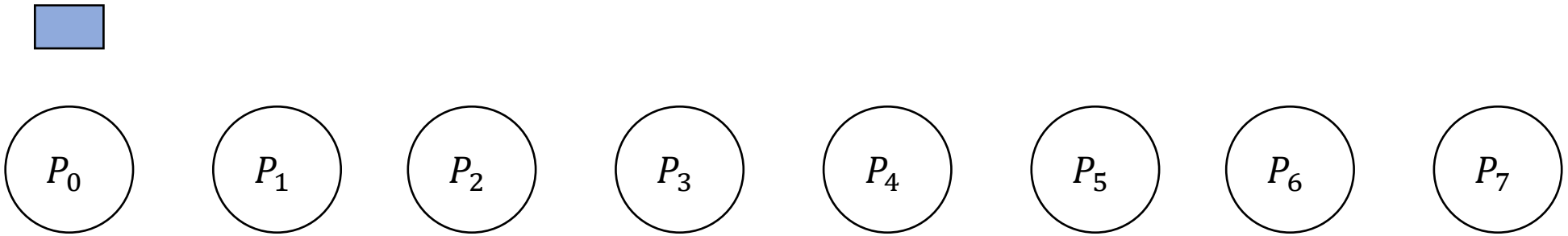
- MPI_Broadcast: $O(\mathrm{t_w} \times \log P)$
  - t_w: size of data to broadcast
  - P: Number of processors
  - Algorithm proceeds in $\log P$ steps, each step communicates t_w amount of data

Collective Operations Communication Complexity (2)

- MPI_Scatter/MPI_Gather: $O(t_w \times P)$
  - t_w: number of elements at each individual processor after scatter (or before gather)
  - In each step ($\log P$ steps), the number of elements that are communicated in doubles (or halved in gather)

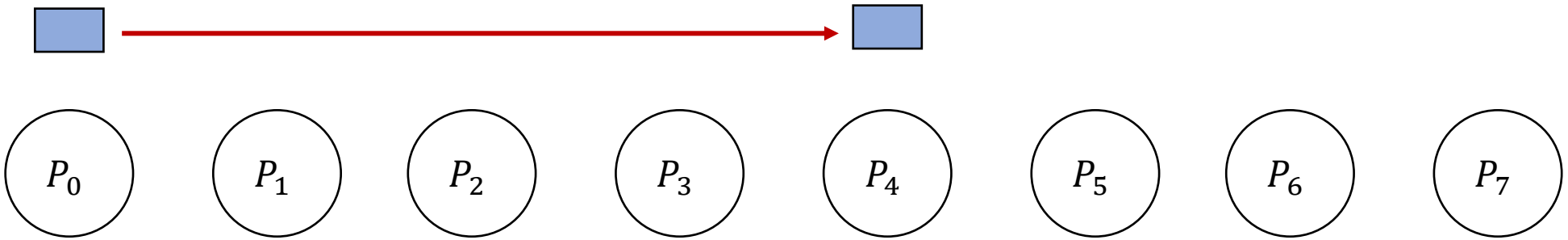
- MPI_Reduce: $O(t_w \times \log P)$
  - t_w: number of elements in the vector used for reduction
  - Algorithm proceeds in $\log P$ steps, each step communicates t_w amount of data. The partial results are aggregated.

MPI_Broadcast



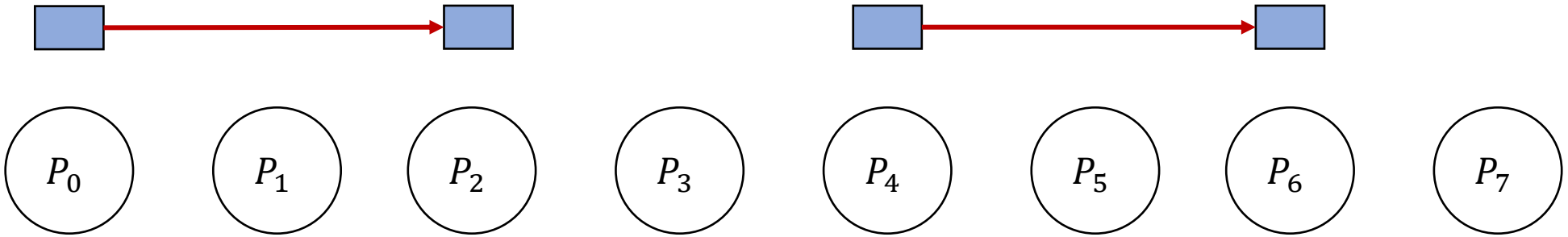
Data initially at P_0

MPI_Broadcast



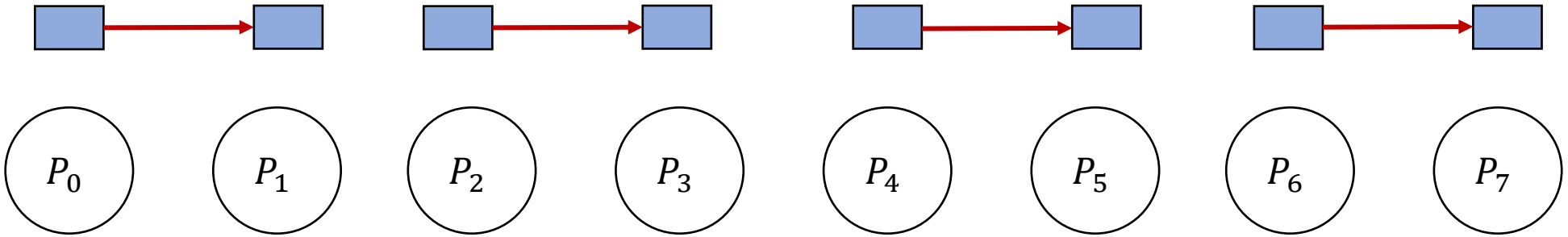
Step 1

MPI_Broadcast



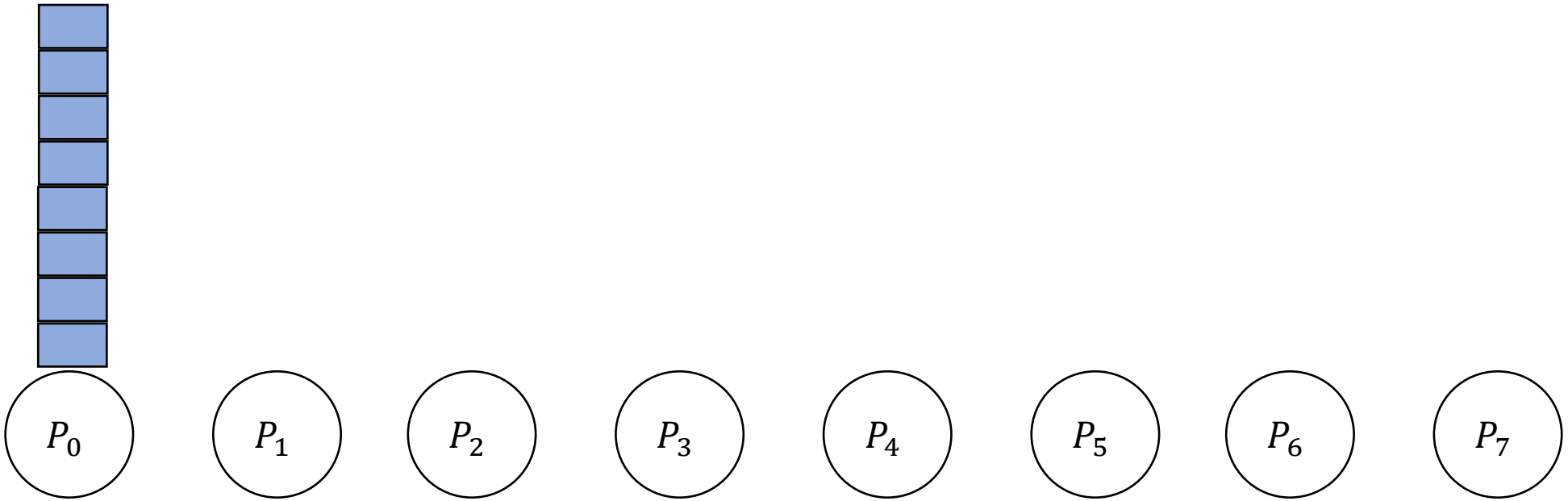
Step 2

MPI_Broadcast



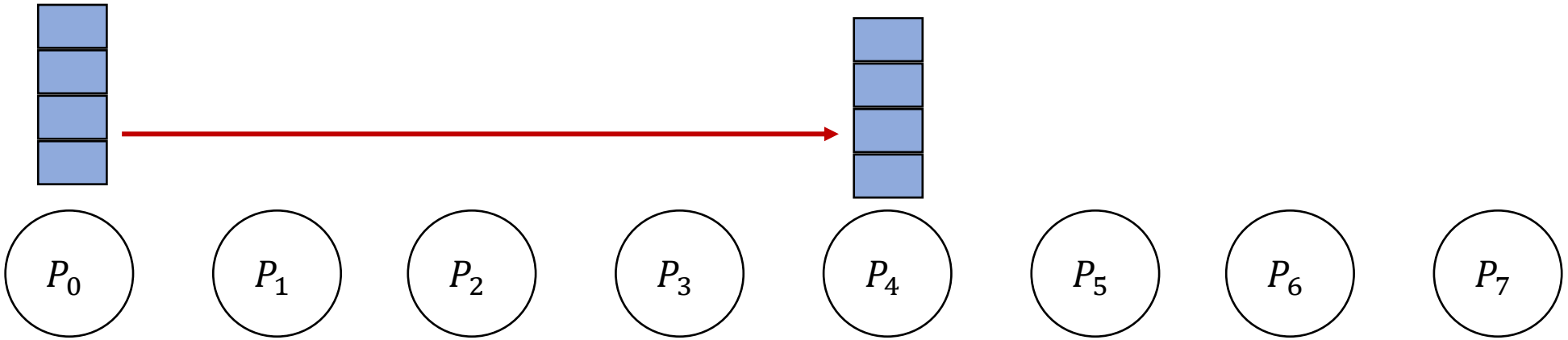
Step 3

MPI_Scatter



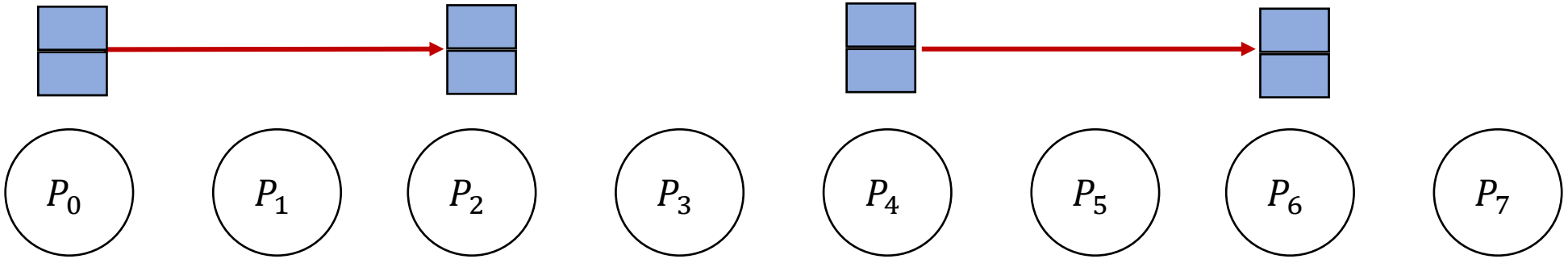
Data initially at P_0
Notice the size

MPI_Scatter



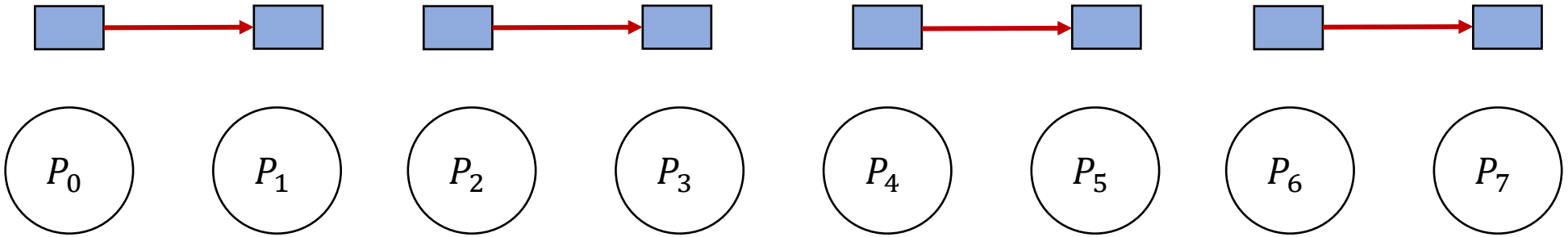
Step 1

MPI_Scatter



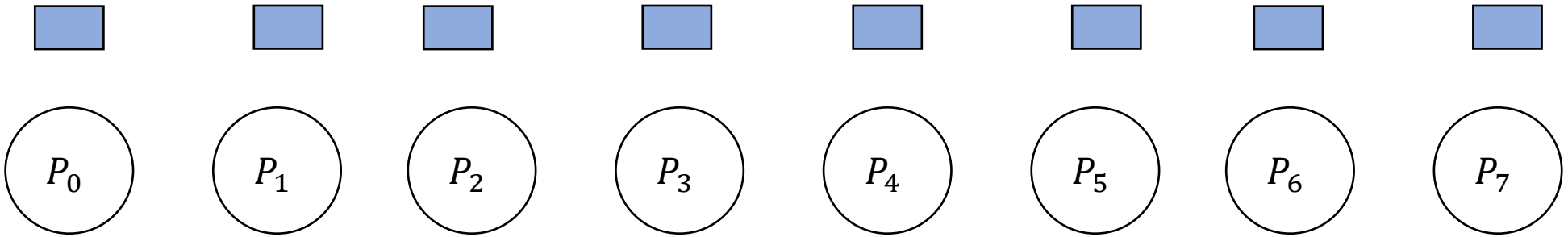
Step 2

MPI_Scatter



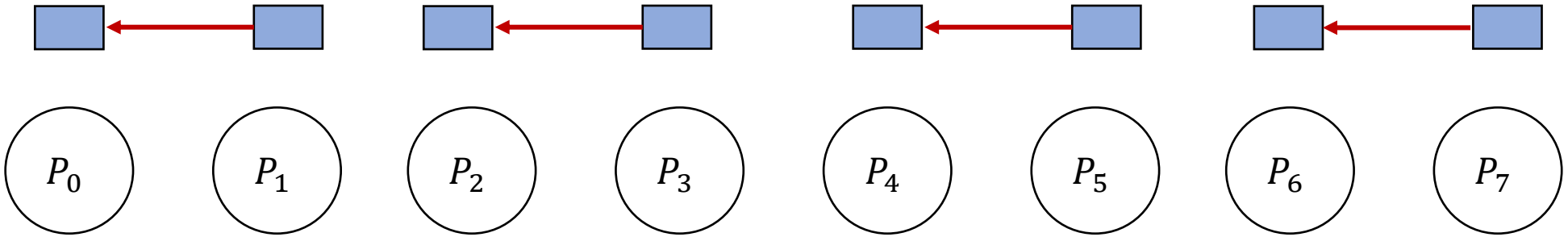
Step 3

MPI_Reduce



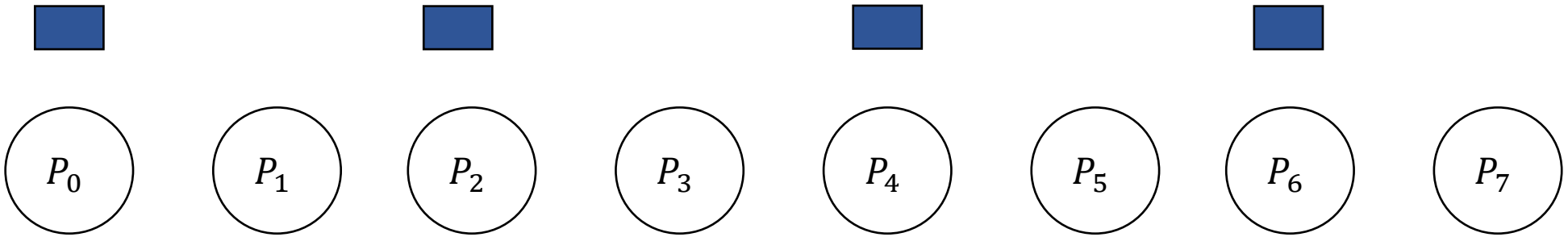
Initial data

MPI_Reduce



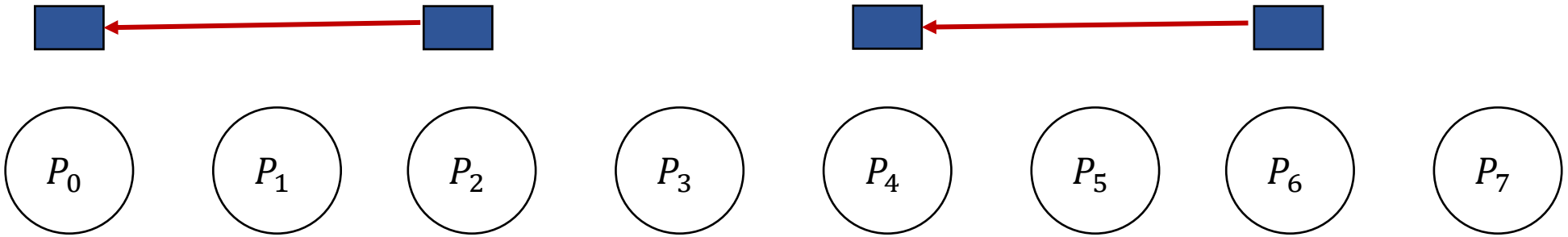
Step 1

MPI_Reduce



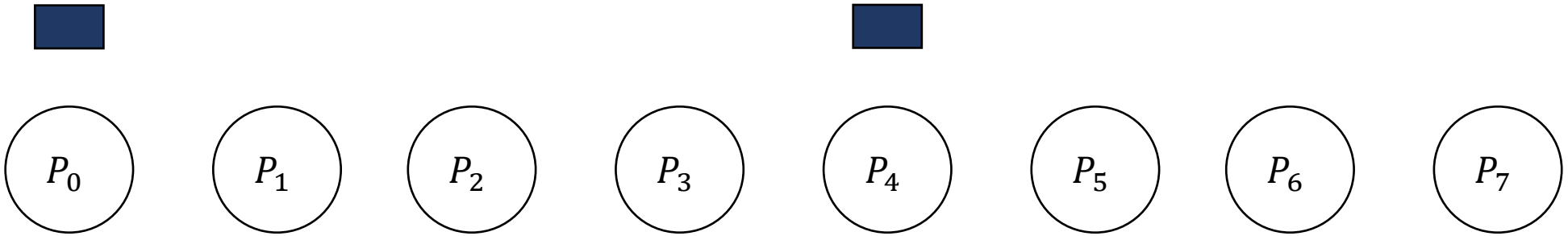
Step 1

MPI_Reduce



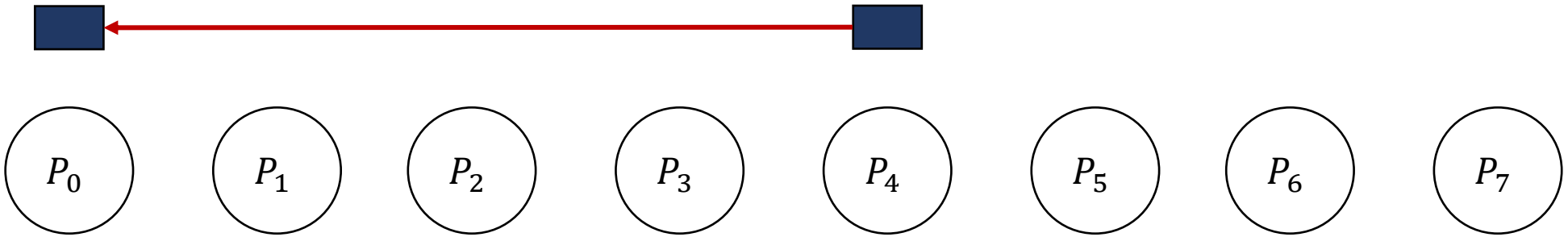
Step 2

MPI_Reduce



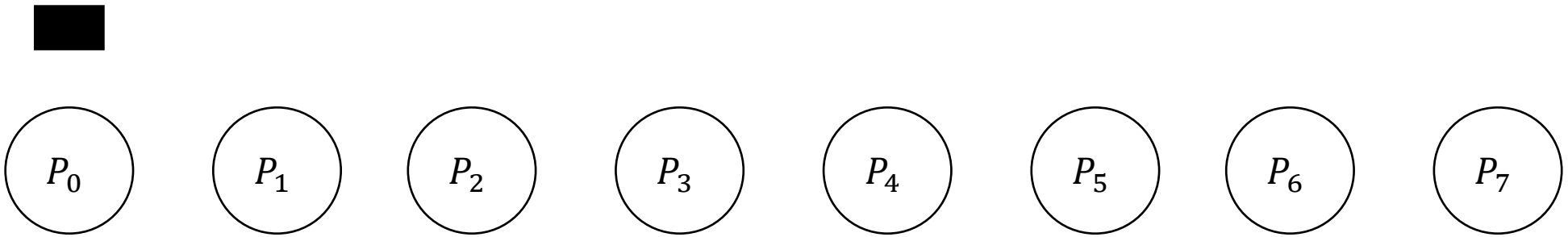
Step 2

MPI_Reduce



Step 3

MPI_Reduce



Step 3

Collective Communication Operation Example

- Ungraded HW Assignment (may ask in WA 4)

- Calculate the average of an array of size P using P processors
  - Initial data mapping: Each processor owns one element of the array

- Approach #1:
  - P0 performs MPI_Gather
  - P0 calculates the average of the gathered array
  - P0 performs MPI_Scatter of the average

Collective Communication Operation Example

- Approach #2
  - P0 performs MPI_Reduce with MPI_SUM as aggregation operation
  - P0 divides the aggregated value by P (number of processors) to obtain
  - P0 performs MPI_Broadcast of the average
- What is the communication time complexity of these two approaches?
- Which approach is better?

Next Class

- 11/13 Lecture 22
  - Communication Complexity of Communication Collectives
  - Distribute Training Parallelism Paradigms – Implementation Using Communication Primitives

CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

Thank You

- Questions?

- Email: sanmukh.kuppannagari@case.edu