CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

CSDS 451: Designing High Performant Systems for AI

Lecture 22

11/13/2025

Sanmukh Kuppannagari

sanmukh.kuppannagari@case.edu

https://sanmukh.research.st/

Case Western Reserve University

Outline

- Collective Communication Primitives – Complexity Analysis

Announcements

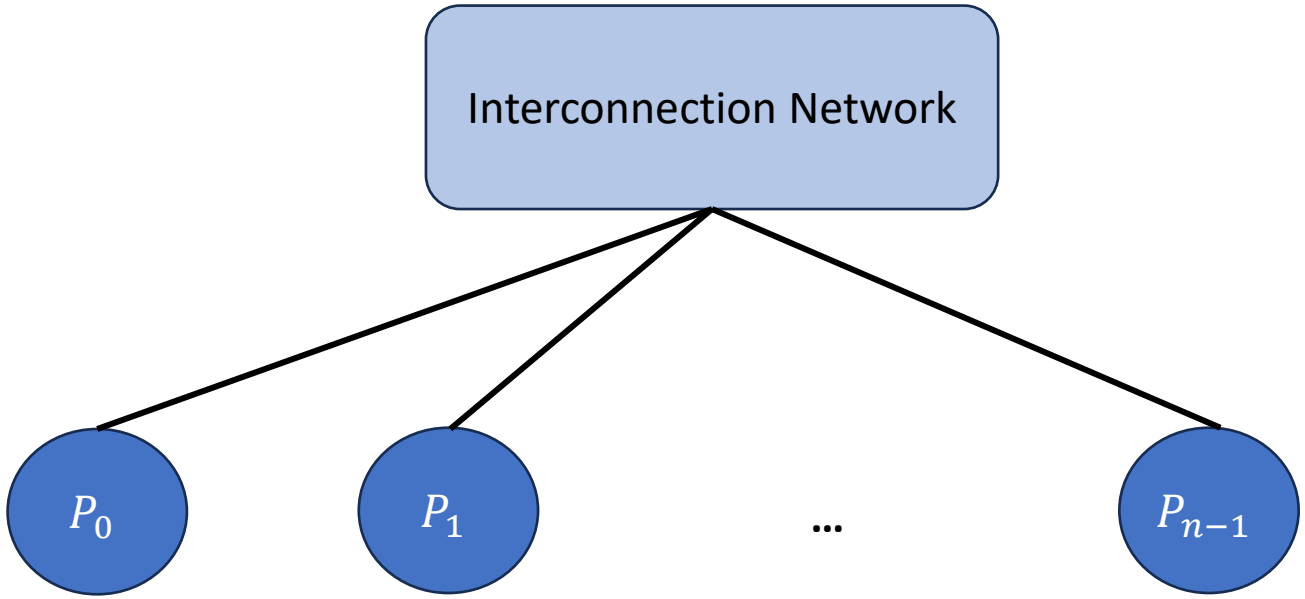
- WA3 Due this Saturday
- WA4 Will be Out
- Project Proposals Graded
  - If you did not receive a score, I may have asked you followup questions on your proposal
  - Please send me weekly updates by responding to the email so that you remain on track

Outline

- Last Class Review

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

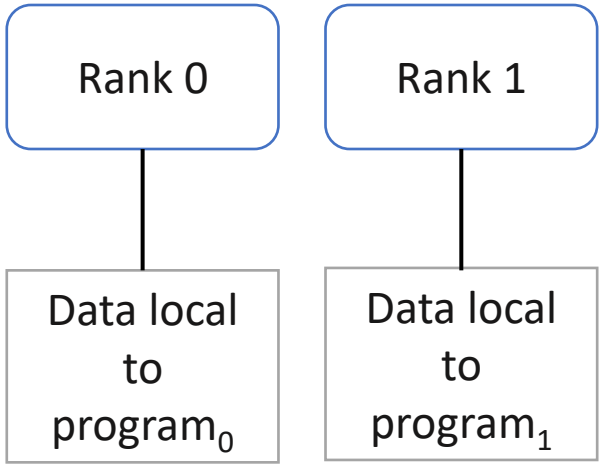
- MPI_Init(…)
- MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)
- MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

- Do rank specific work

Inter-Process Communication

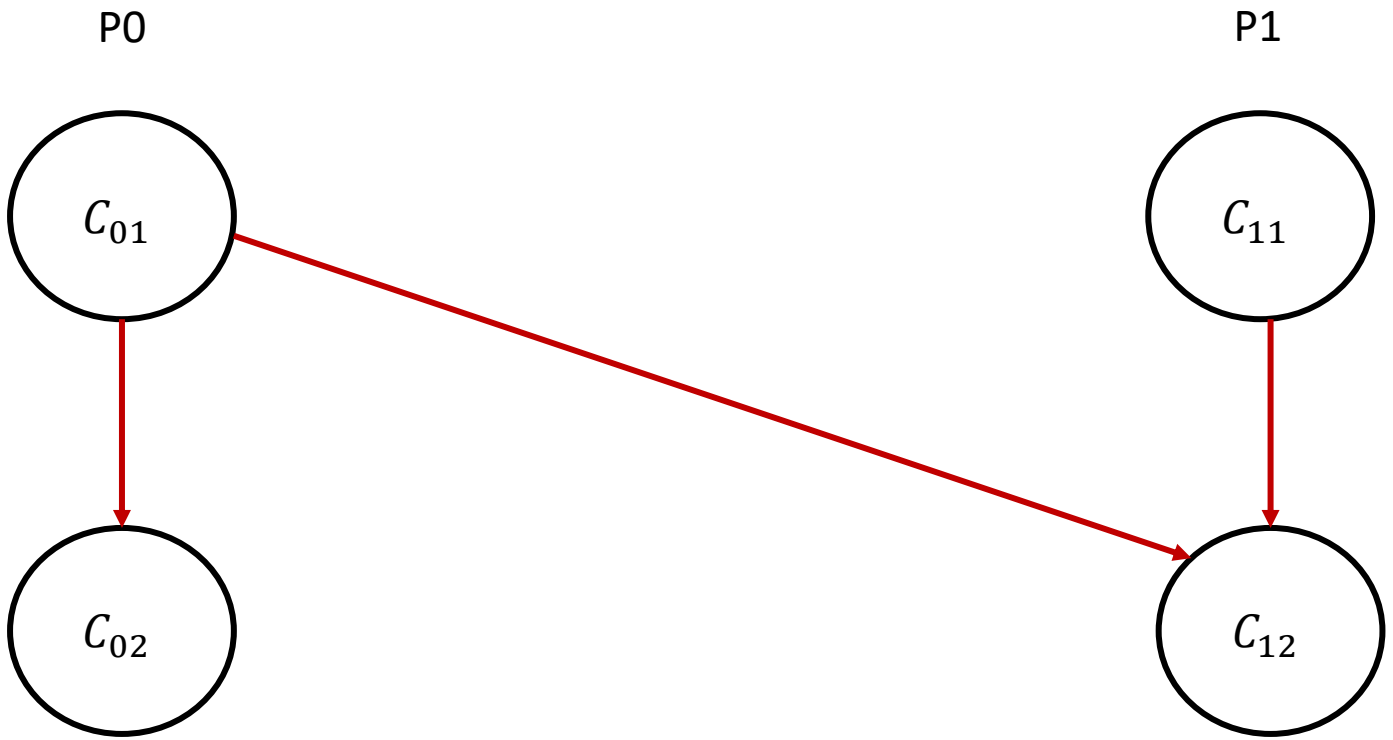
MPI_Init(…)
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

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

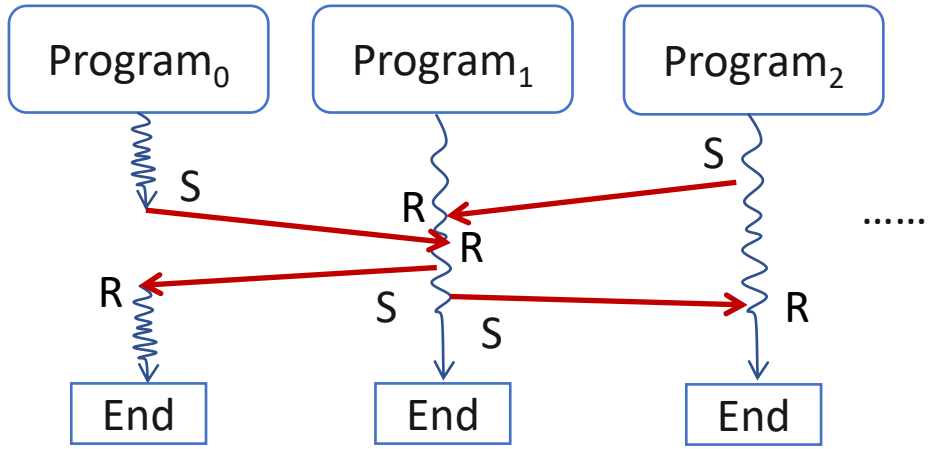
Inter-Process Communication



Dependency across processors due to communication operation

Message Passing Program

Most General Model: Asynchronous

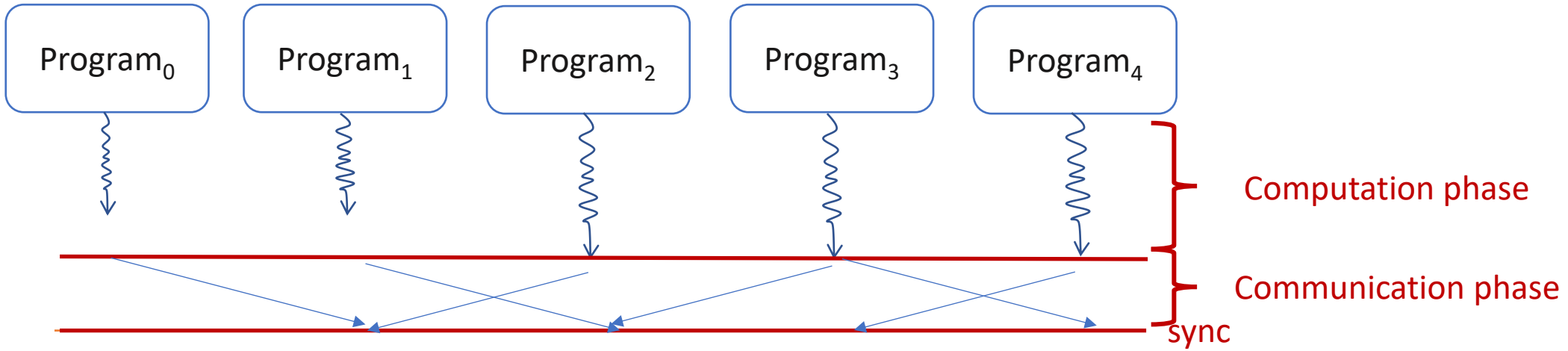


S = send
R = receive

- 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



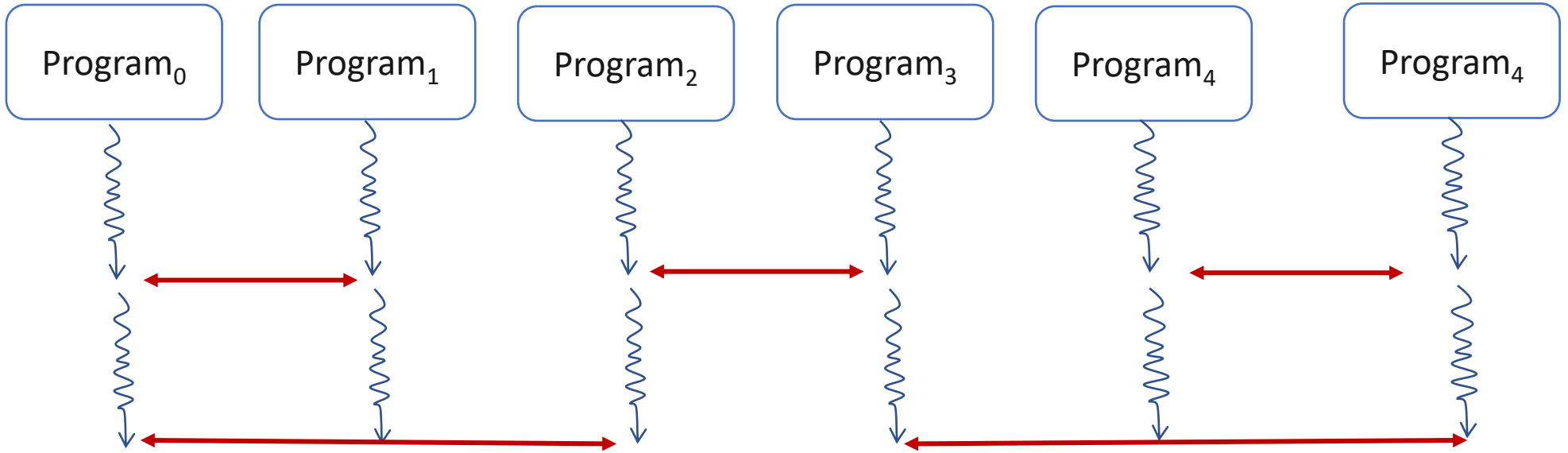
Two phases to the Program

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

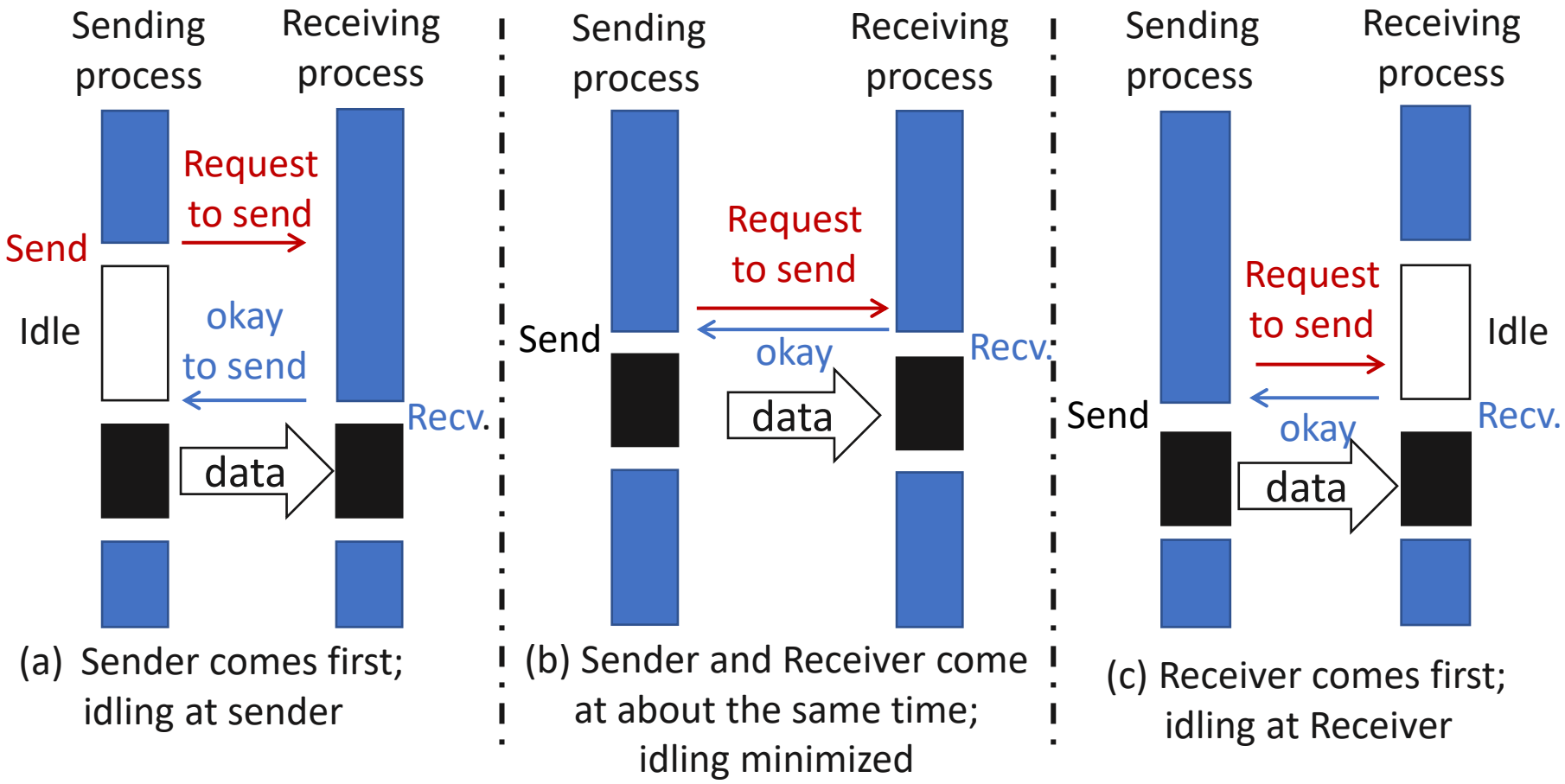
Message Passing Program

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

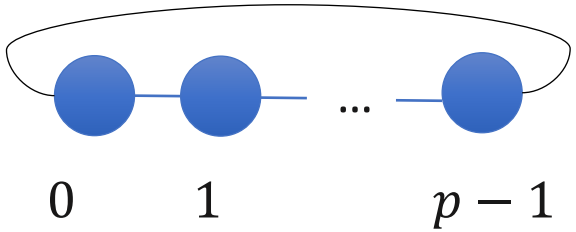


Blocking Non-Buffered Send/Receive

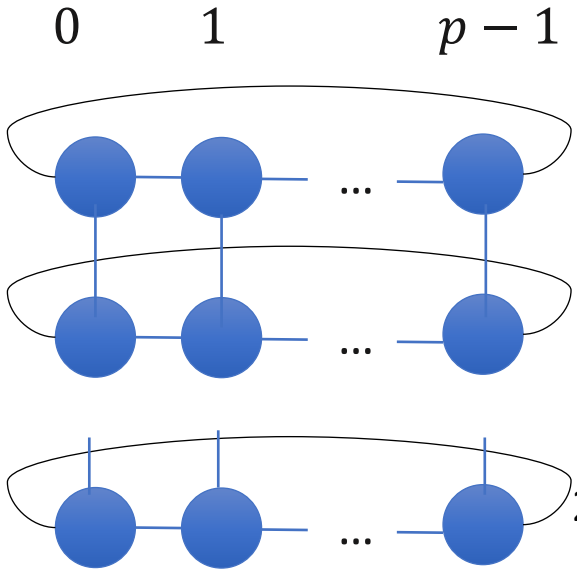


Virtual Topology

- 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.

Distributed Matrix Multiplication

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

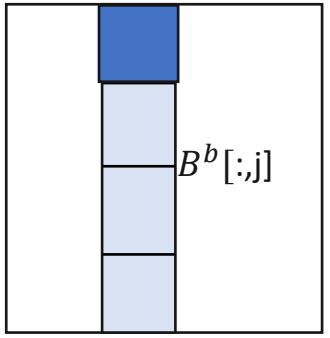
- Popular example – Tensor Parallelism in AI training (will discuss in next class)

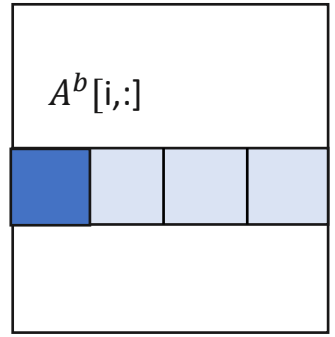
MM on 2D Mesh

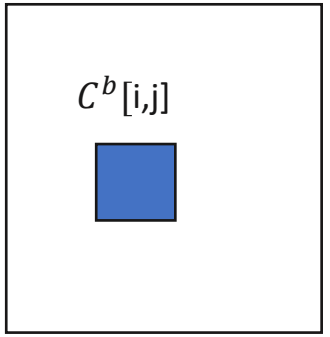
- Assume A, B, C are $n \times n$ matrices
- Concurrency Model: SPMD
- Virtual Topology: 2D mesh of size $\sqrt{p} \times \sqrt{p}$
  - Processors denoted as: P_{ij} $0 \leq i, j < \sqrt{p},\ 1 \leq \sqrt{p} \leq n$
- **Cannon's Algorithm**

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}? $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$
- How do we iterate over all blocks? Need to fetch blocks A_{ik} and B_{kj} to processor P_{ij} using Message Passing



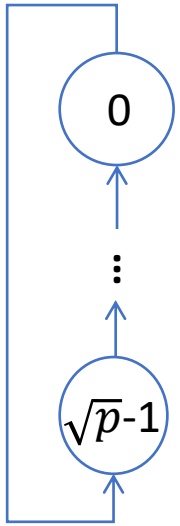




MM on 2D Mesh – Cannon's Algorithm

Circular left shift

Circular up shift



MM on 2D Mesh – Cannon's Algorithm

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

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

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

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

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 = $\sqrt{p}$
- 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 = $\sqrt{p}$
- 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

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

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:

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

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)

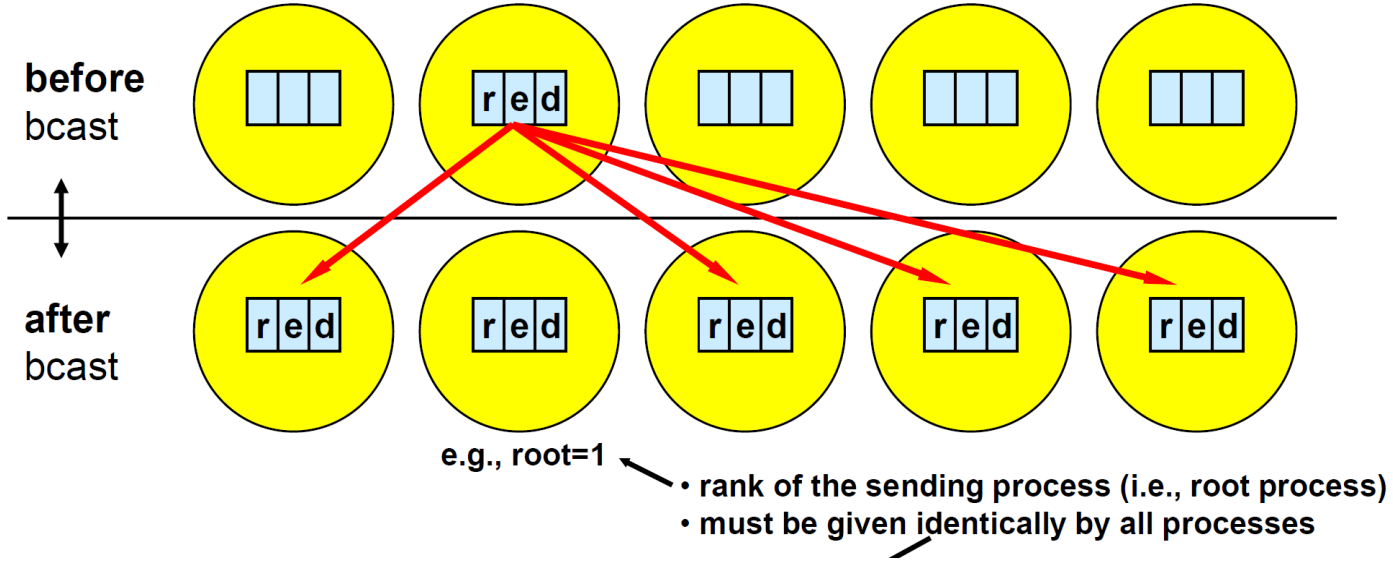
- Called by **ALL** the processors
  - Each processor should call the same API with same size of buffer
  - If a call is missing in one of the processors, it may lead to undefined behavior
- One program in **EACH** processor reaches the call, the communication occurs
- Examples
  - Barrier Synchronization (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

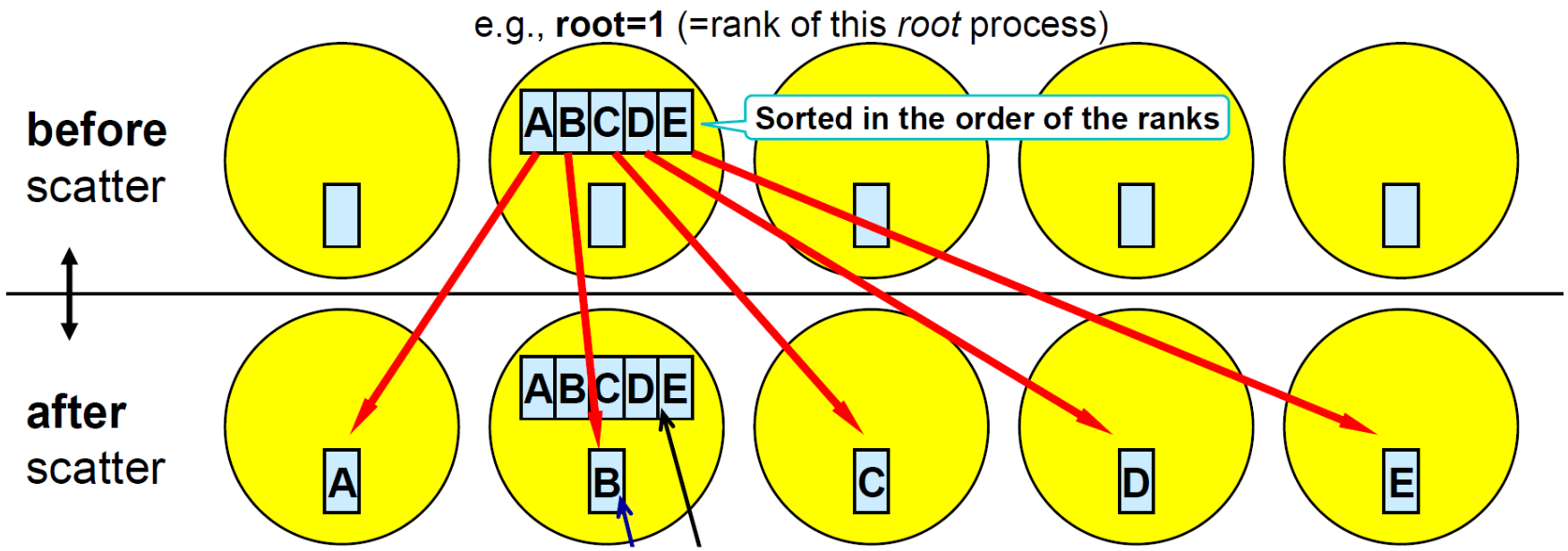
- Send the data stored in **buffer** at the **root** processor to all the processors



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

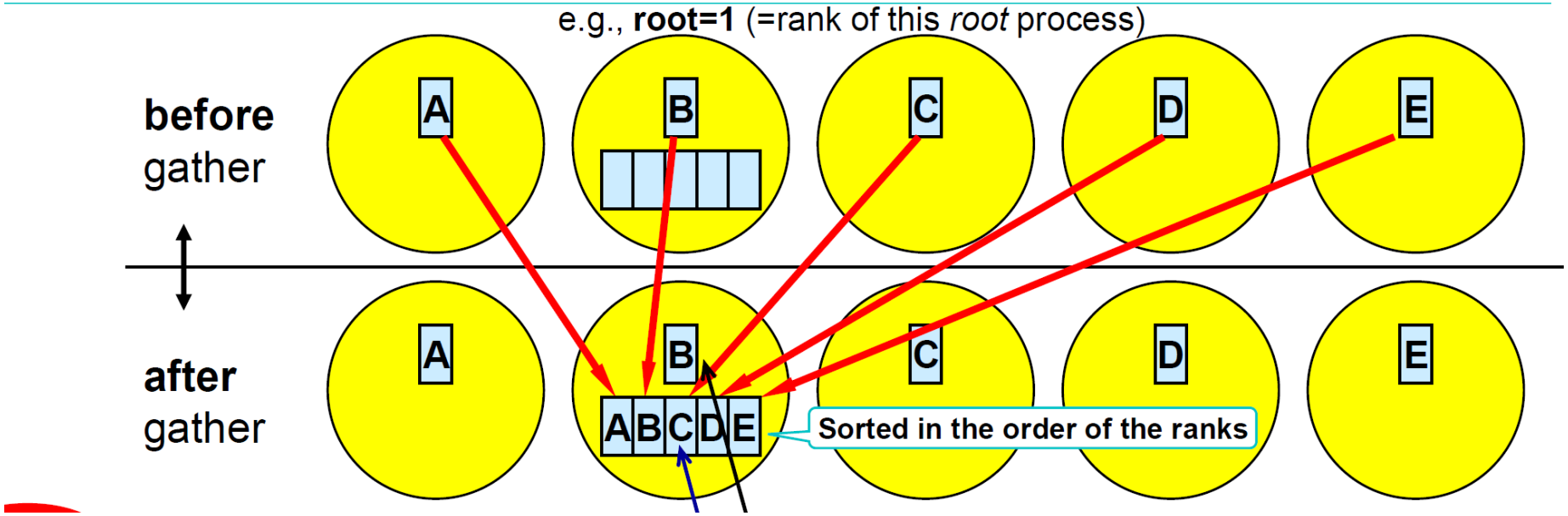
MPI_Scatter


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

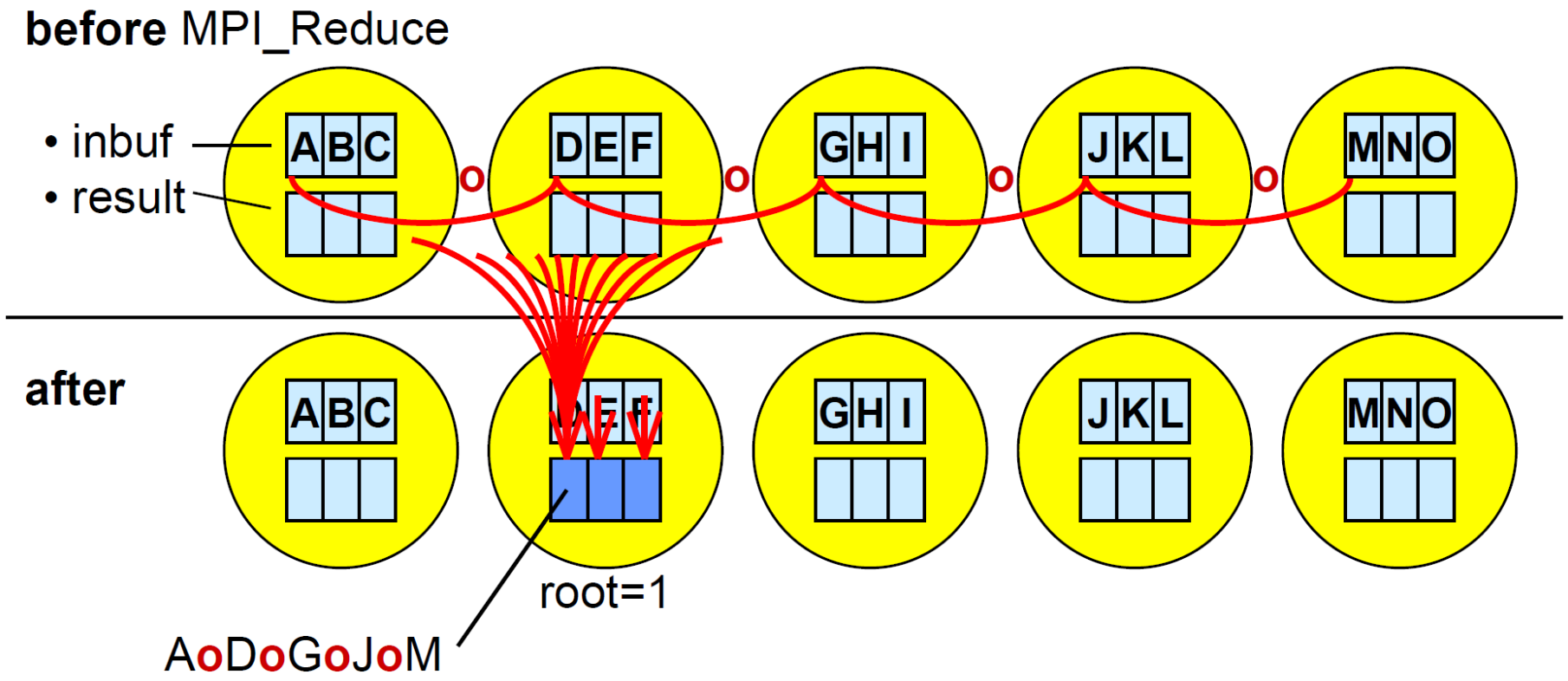
MPI_Gather


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


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_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_Broadcast(resultbuf, rbsize, root)

Outline

- Collective Communication Primitives – Complexity Analysis

Collective Operations Communication Complexity (1)

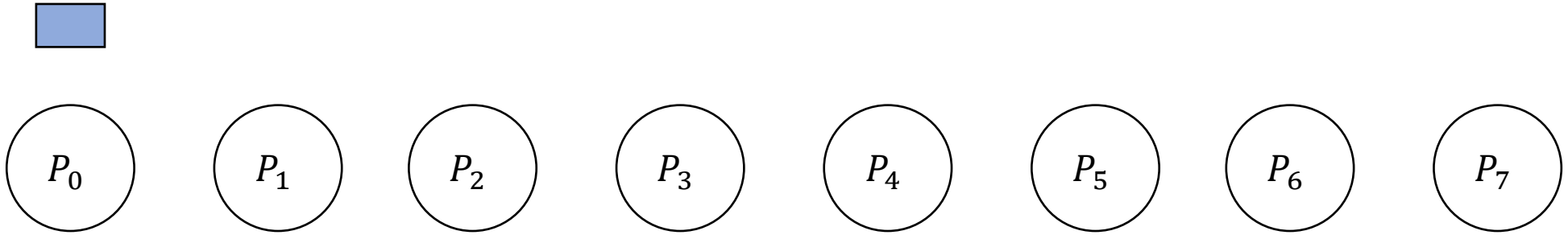
- Objective: Calculate how much time (in order notation) it takes to implement a communication primitive

- Assumptions
  - Computation and other overheads are ignored
  - Fully Connected Interconnect – Communication between any pair of processors takes the same amount of time

- Allows us to compare communication times of different distributed algorithm
  - Example: Cannon's algorithm vs some other matrix multiplication algorithm (SUMMA – another distributed matrix multiplication algorithm:)

MPI_Broadcast

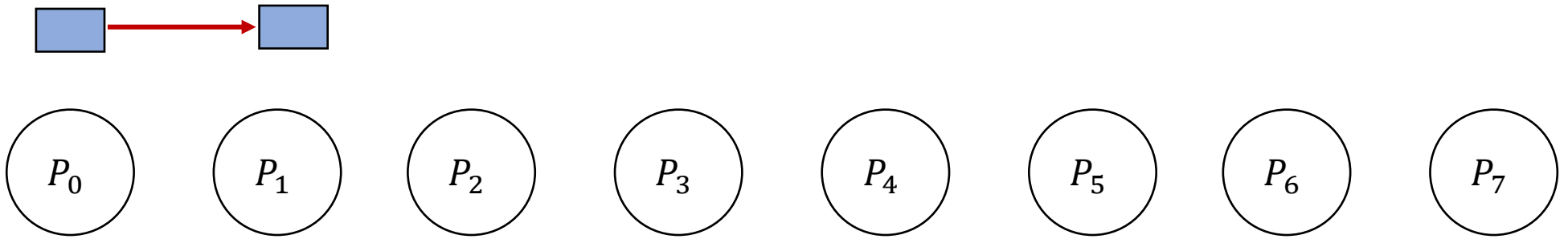
- Can you think about what would be an efficient way?
- Notice:
  - Communication between any pair takes the same amount of time
  - Multiple pairs can communicate at any given time
  - A Processor can do one send and one receive at any given time



Data initially at P_0

MPI_Broadcast

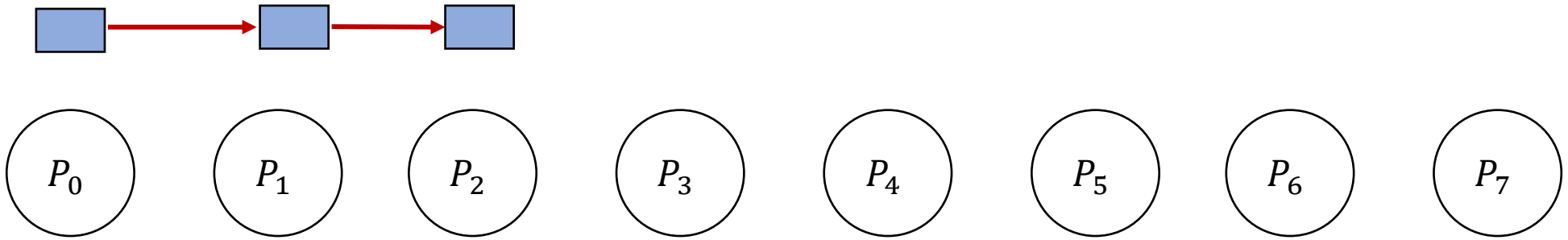
- How about this approach?



Data initially at P_0

MPI_Broadcast

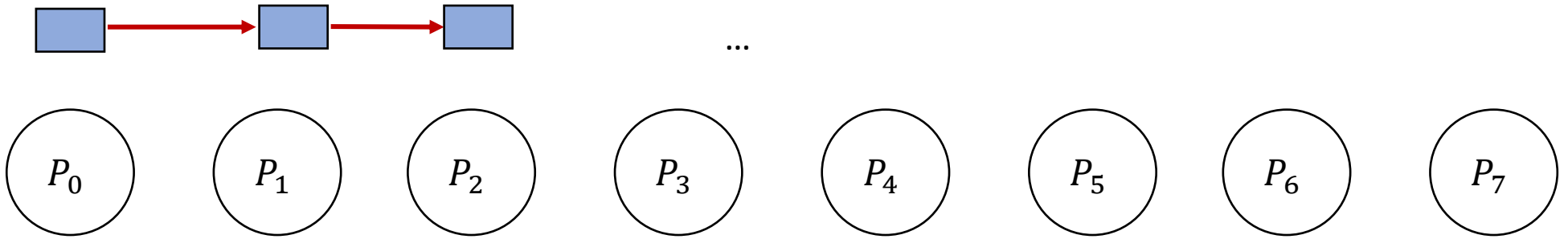
- How about this approach?



Data initially at P_0

MPI_Broadcast

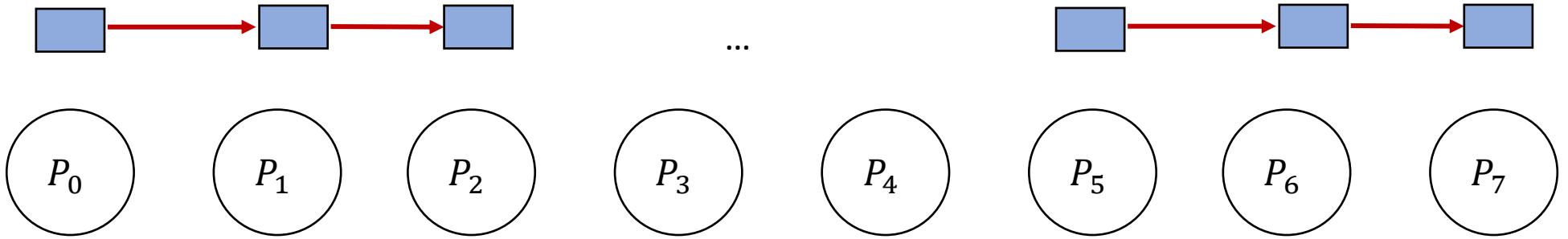
- How about this approach?



Data initially at P_0

MPI_Broadcast

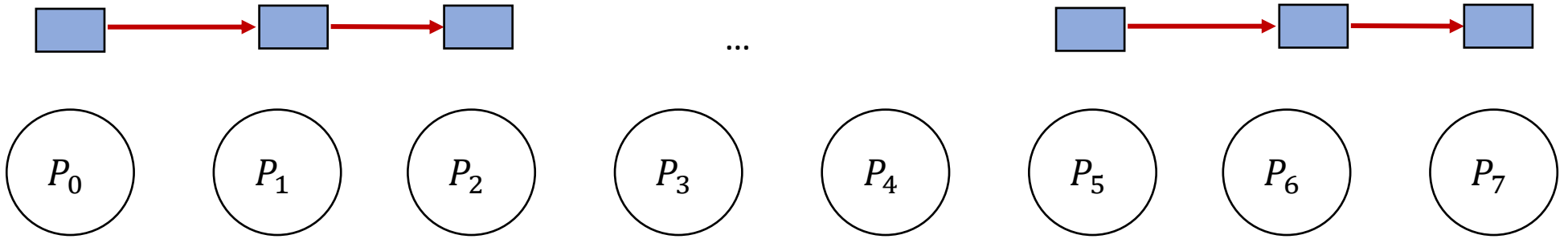
- How about this approach?
- Total communication complexity?



Data initially at P_0

MPI_Broadcast

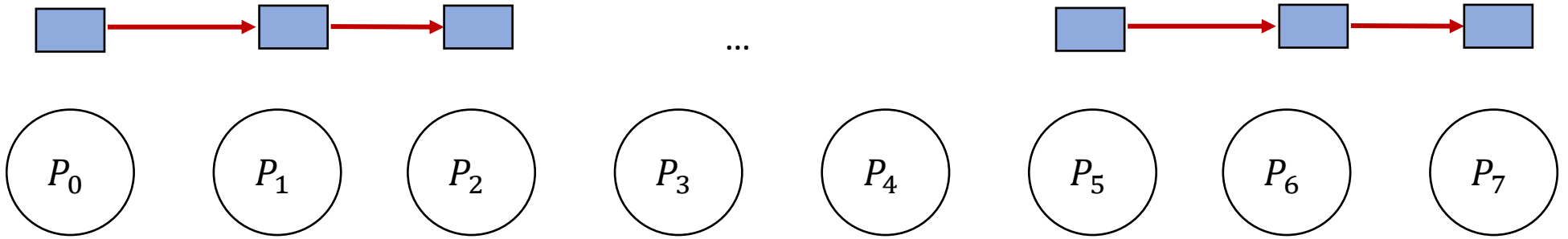
- How about this approach?
- Total communication complexity? $O(P)$



Data initially at P_0

MPI_Broadcast

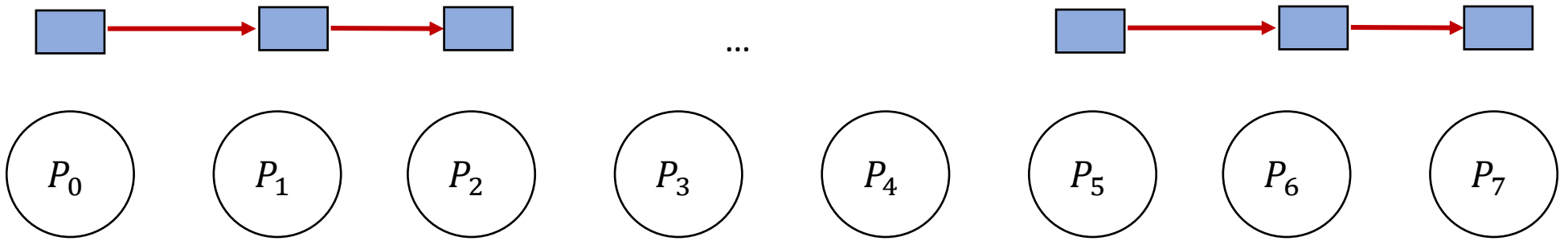
- Can we do better?



Data initially at P_0

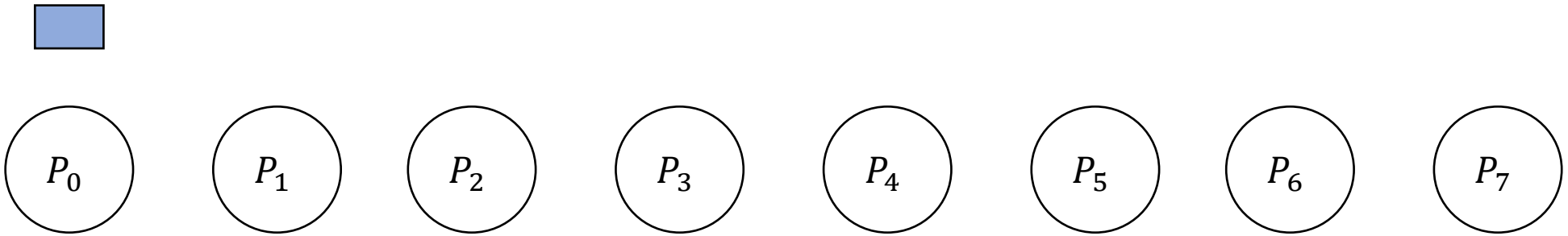
MPI_Broadcast

- Can we do better? Recursive Doubling



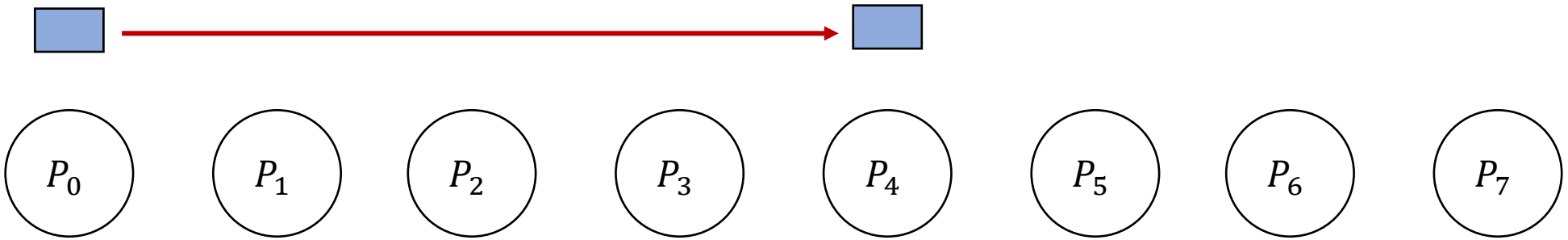
Data initially at P_0

MPI_Broadcast



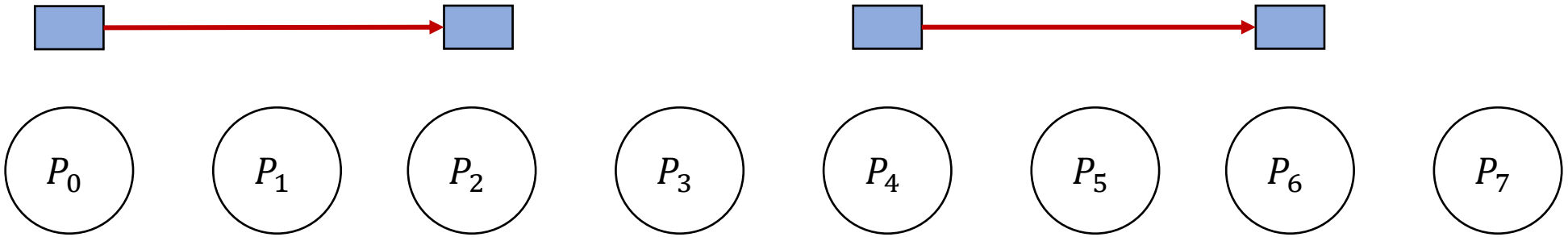
Data initially at P_0

MPI_Broadcast



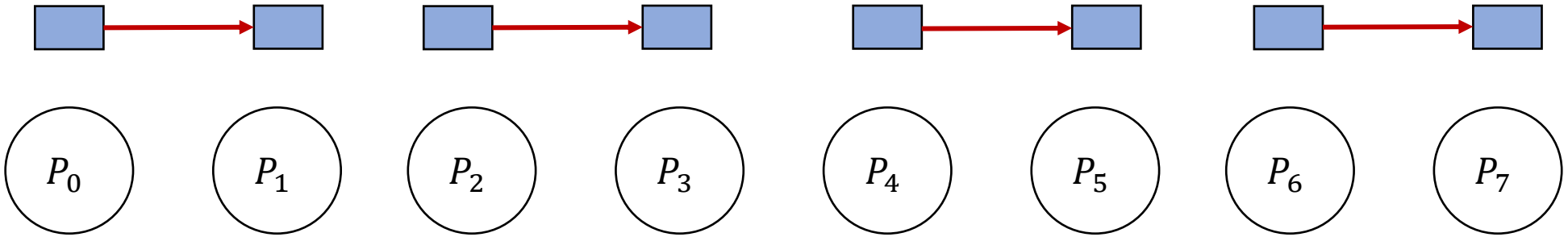
Step 1

MPI_Broadcast



Step 2

MPI_Broadcast



Step 3

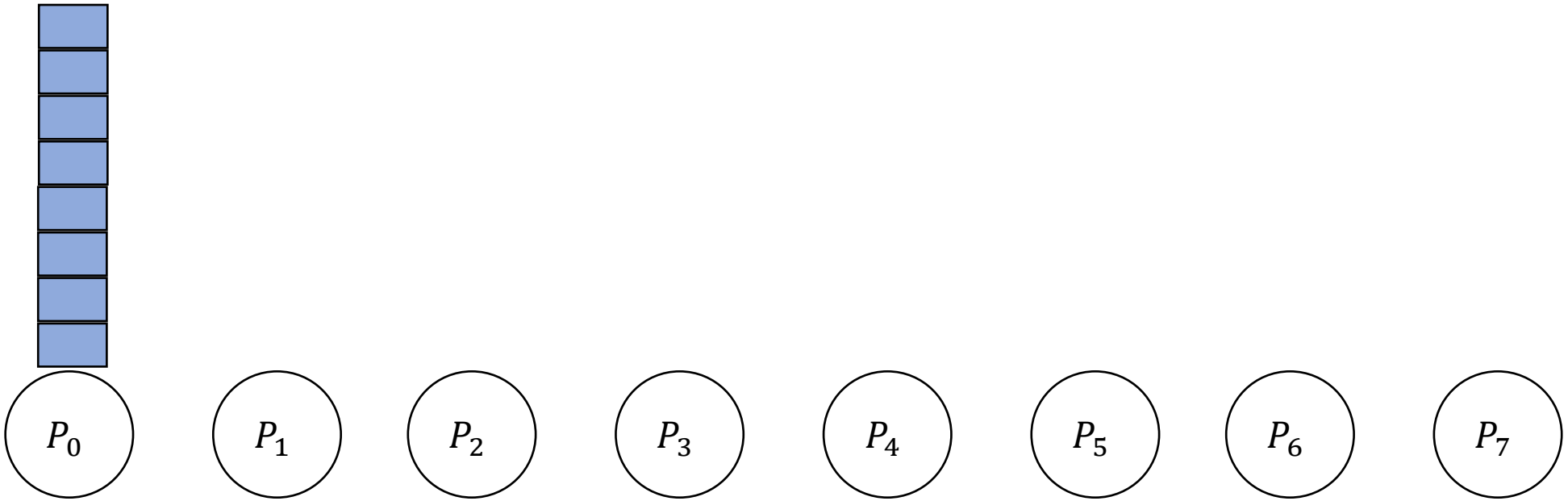
MPI_Broadcast

- MPI_Broadcast Communication Complexity??
  - 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

MPI_Broadcast

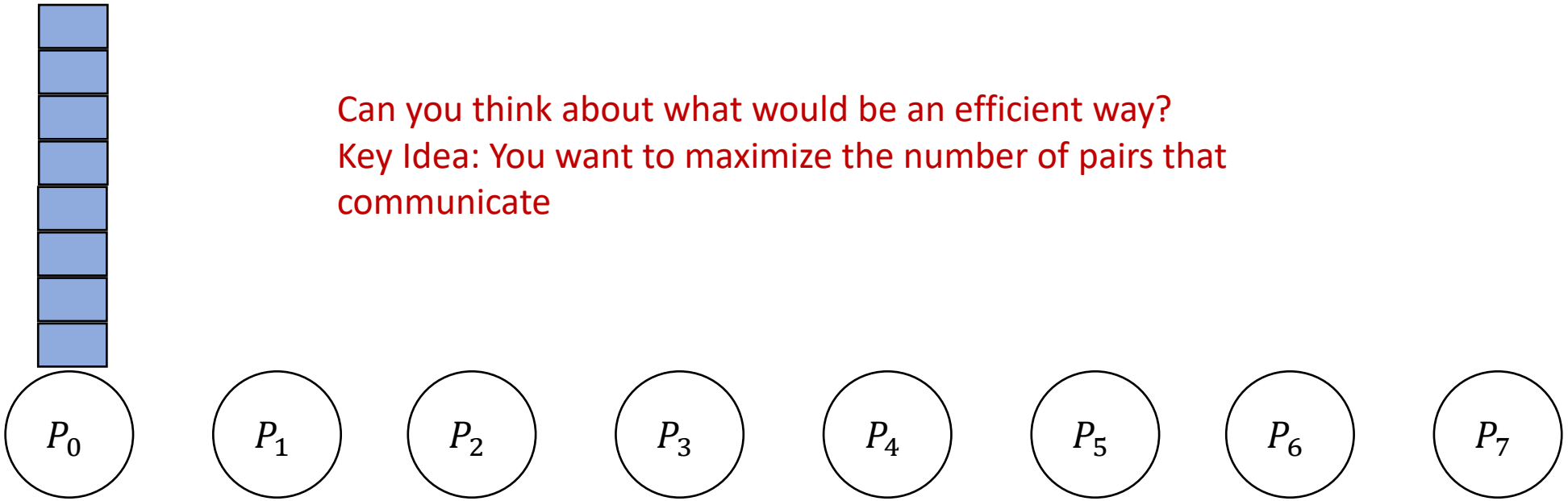
- MPI_Broadcast: $O(\mathrm{t_w} \times \log P)$
  - t_w: size of data to broadcast
  - P: Number of processors
  - Algorithm proceeds in $\log P$ steps, each step communicates t_w amount of data

MPI_Scatter



Data initially at P_0
Notice the size

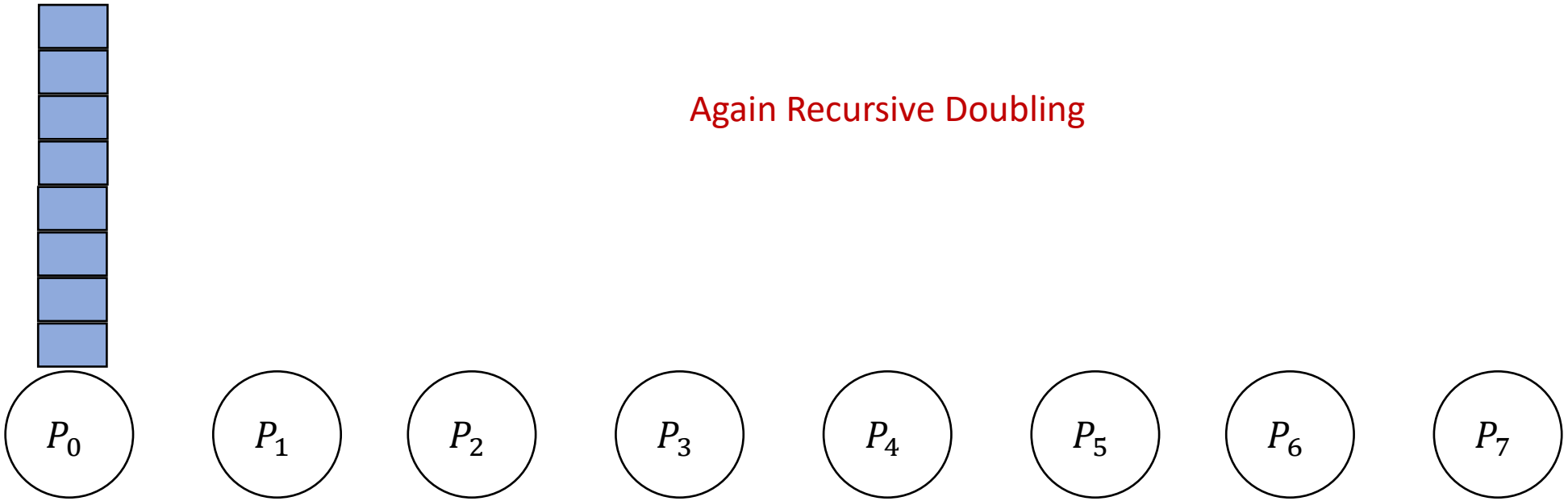
MPI_Scatter



Can you think about what would be an efficient way?
Key Idea: You want to maximize the number of pairs that communicate

Data initially at P_0
Notice the size

MPI_Scatter

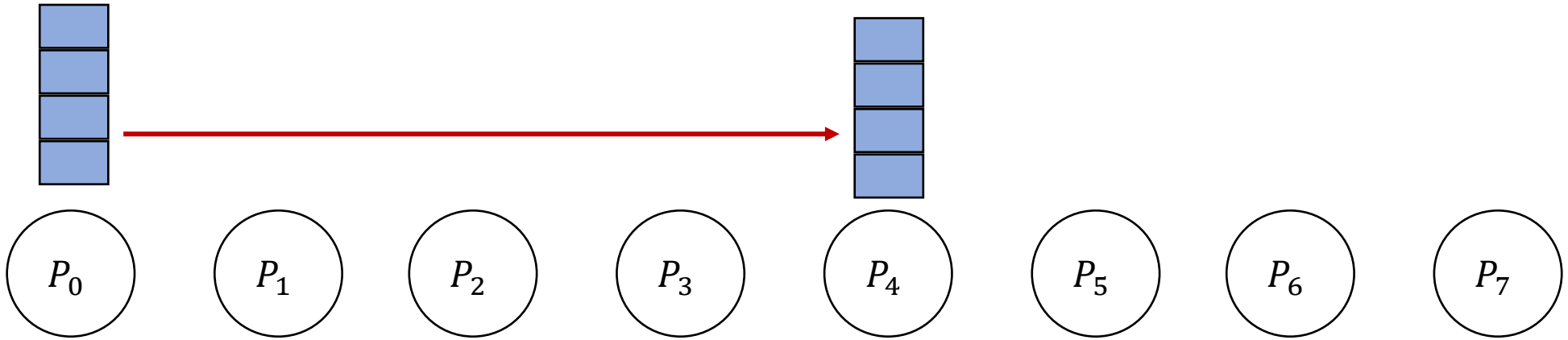


Again Recursive Doubling

Data initially at P_0
Notice the size

MPI_Scatter

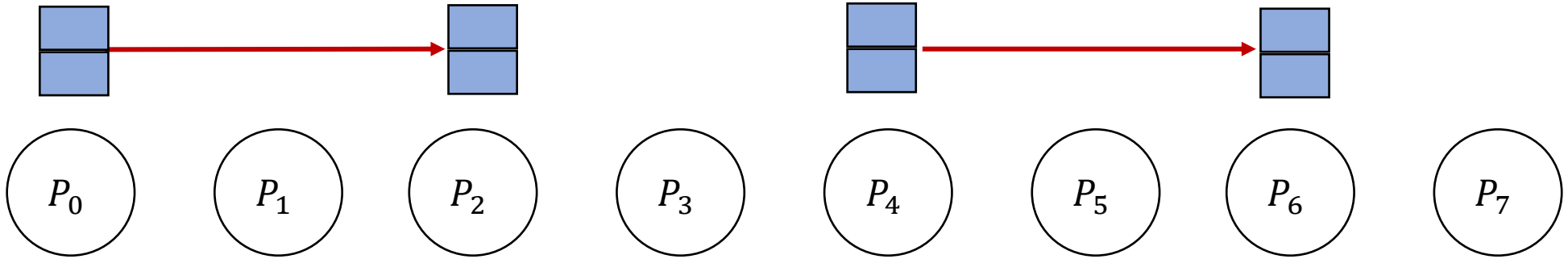
Again Recursive Doubling



Step 1

MPI_Scatter

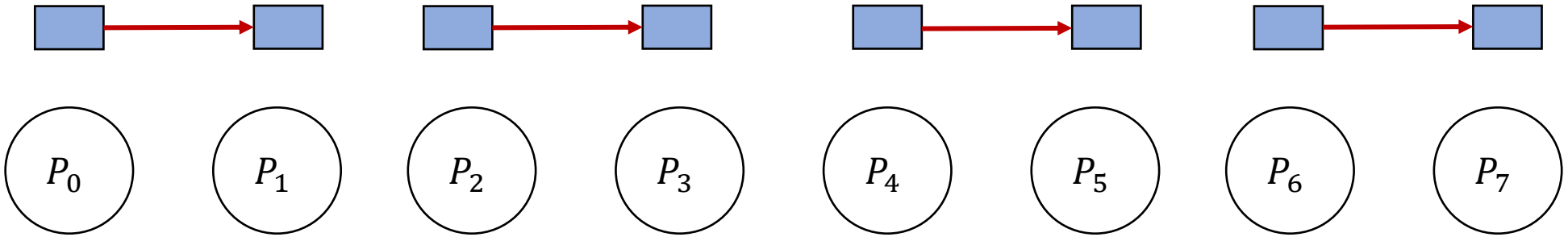
Again Recursive Doubling



Step 2

MPI_Scatter

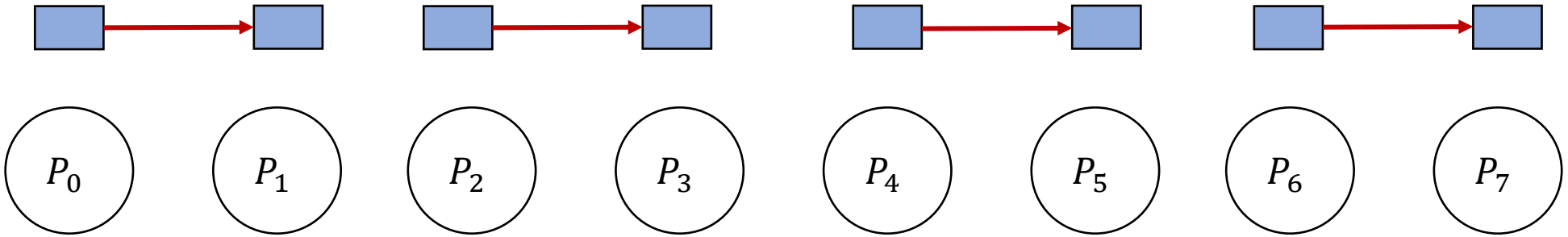
Again Recursive Doubling



Step 3

MPI_Scatter

Can you calculate the communication complexity?

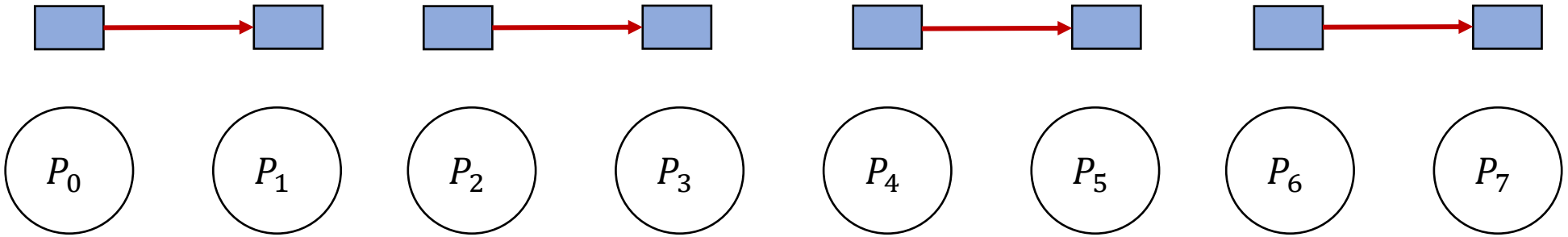


Step 3

MPI_Scatter

Can you calculate the communication complexity?

$$\frac{P}{2} + \frac{P}{4} + \frac{P}{8} \ldots . 1 =$$

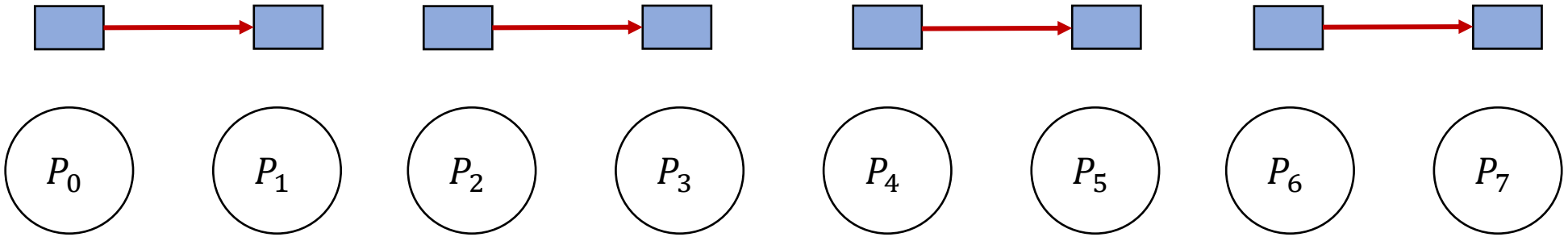


Step 3

MPI_Scatter

Can you calculate the communication complexity (Assuming $t_w = 1$)?

$$\frac{P}{2} + \frac{P}{4} + \frac{P}{8} \ldots . 1 = O(P)$$



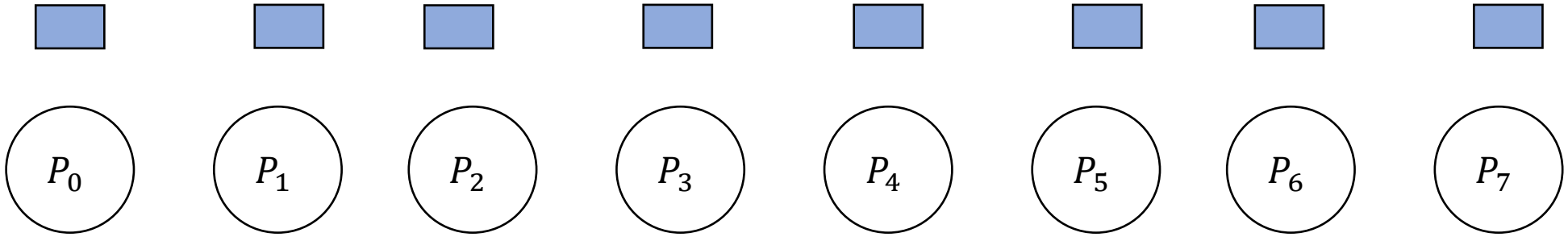
Step 3

MPI_Gather

- Inverse of MPI_Scatter
- Complexity: $O(P)$

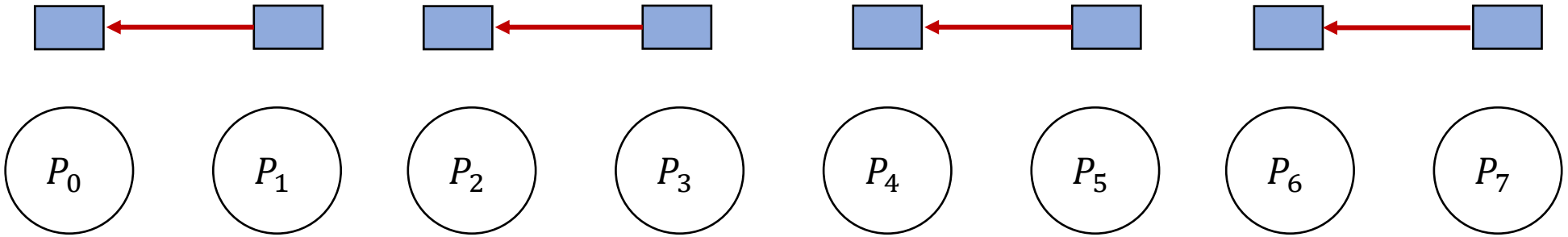
MPI_Reduce

Can you think about what would be an efficient way?
Key Idea: You want to maximize the number of pairs that communicate



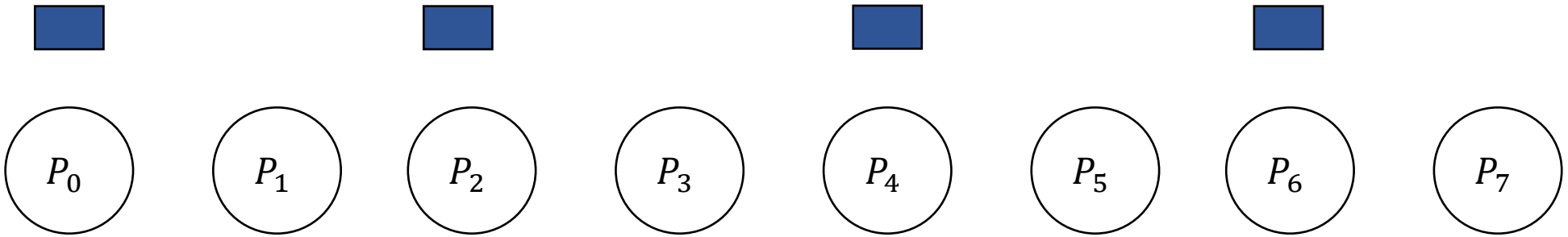
Initial data. Need to store the Aggregate at P_0

MPI_Reduce



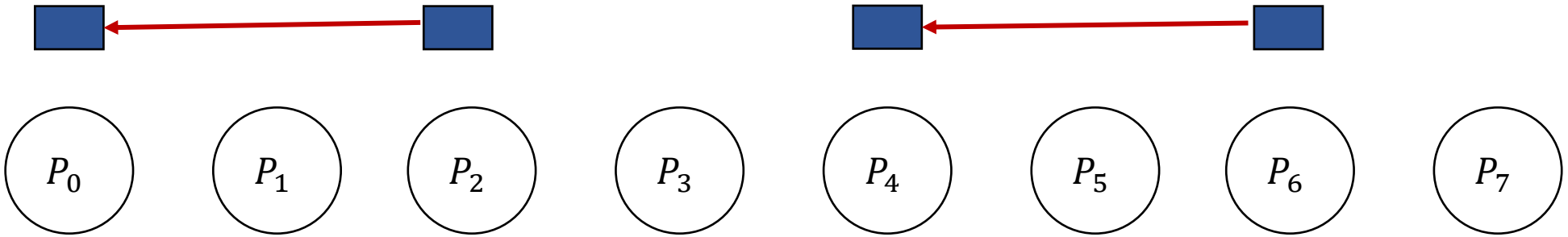
Step 1

MPI_Reduce



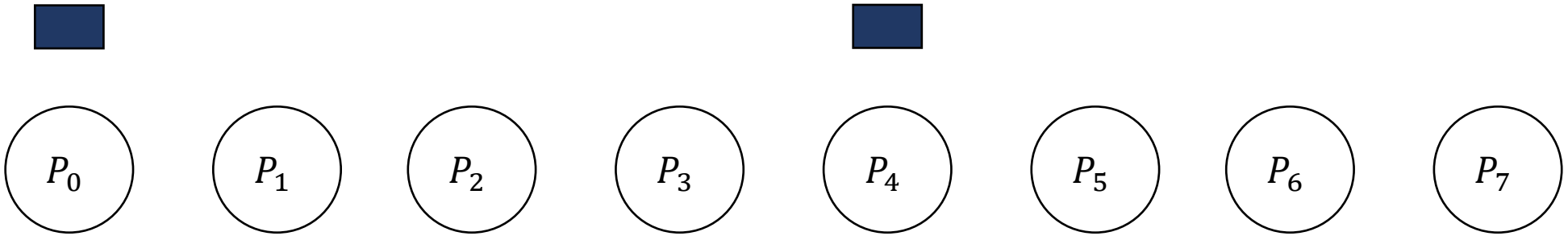
Step 1

MPI_Reduce



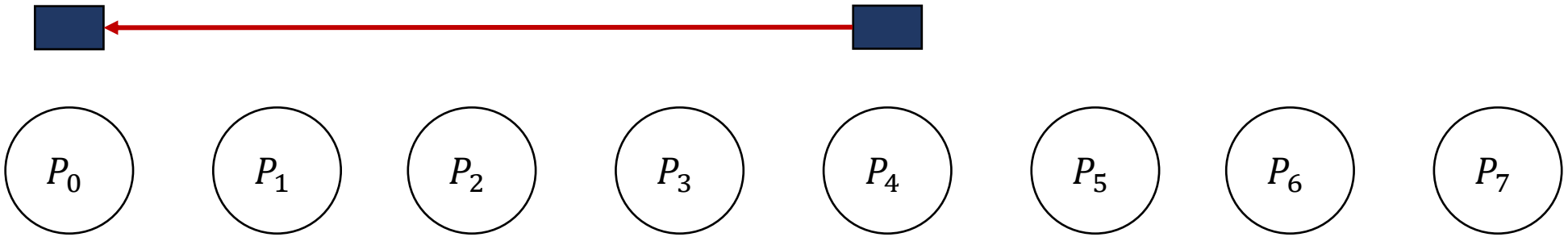
Step 2

MPI_Reduce



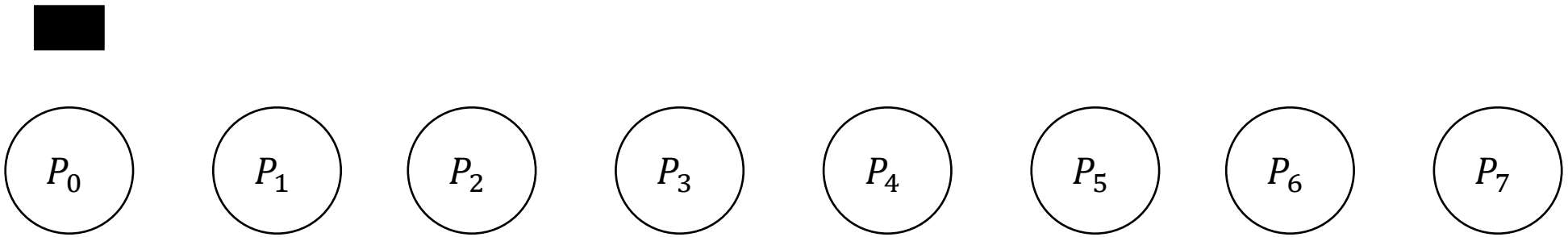
Step 2

MPI_Reduce



Step 3

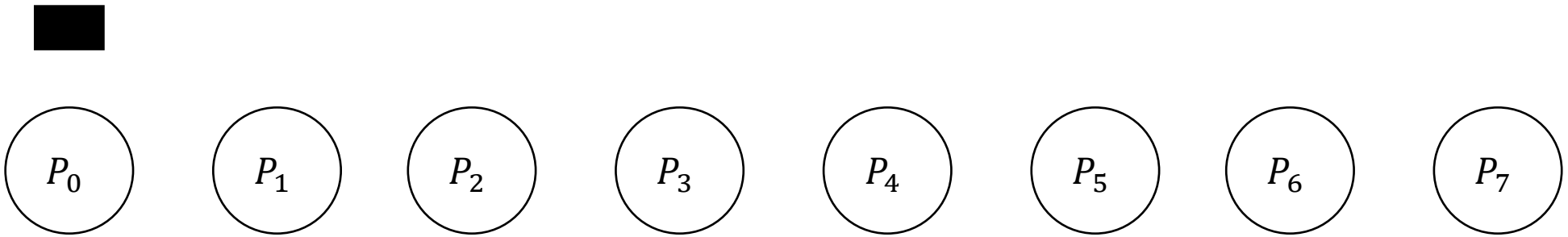
MPI_Reduce



Step 3

MPI_Reduce

Can you calculate the communication complexity?

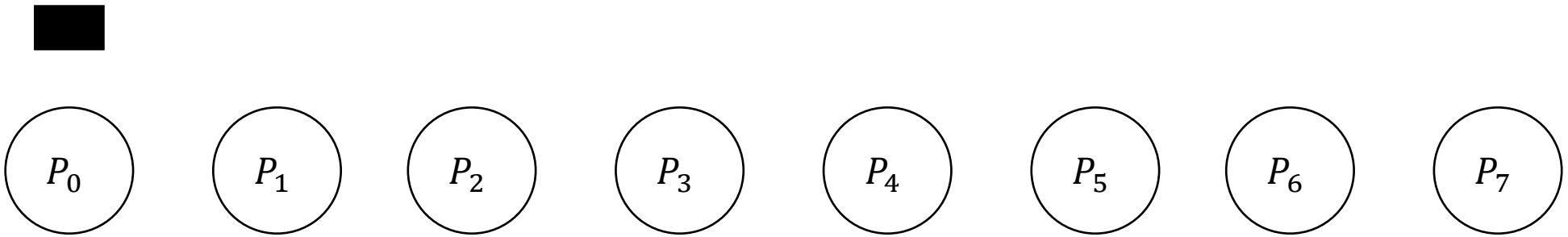


Step 3

MPI_Reduce

Can you calculate the communication complexity?

1 + 1 + … 1 ($\log P$)times = $O(\log P)$



Step 3

Implementing Algorithms using Communication Primitives

- Sum of Algorithms on 1D Array: ??

Implementing Algorithms using Communication Primitives

- Sum of Algorithms on 1D Array: MPI_Reduce(inbuf, 1,resultbuf,1,SUM, 0)

- Communication Complexity - $O(\log P)$

Implementing Algorithms using Communication Primitives

- 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

- What is the communication complexity?
  - Ungraded HW Assignment (may ask in WA 4)

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?
  - Ungraded HW Assignment (may ask in WA 4)
- Which approach is better?

Next Class

- 11/20 Lecture 23
  - Distribute Training Parallelism Paradigms – Implementation Using Communication Primitives

CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

Thank You

- Questions?

- Email: sanmukh.kuppannagari@case.edu