CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

CSDS 451: Designing High Performant Systems for AI

Lecture 20

11/6/2025

Sanmukh Kuppannagari

sanmukh.kuppannagari@case.edu

https://sanmukh.research.st/

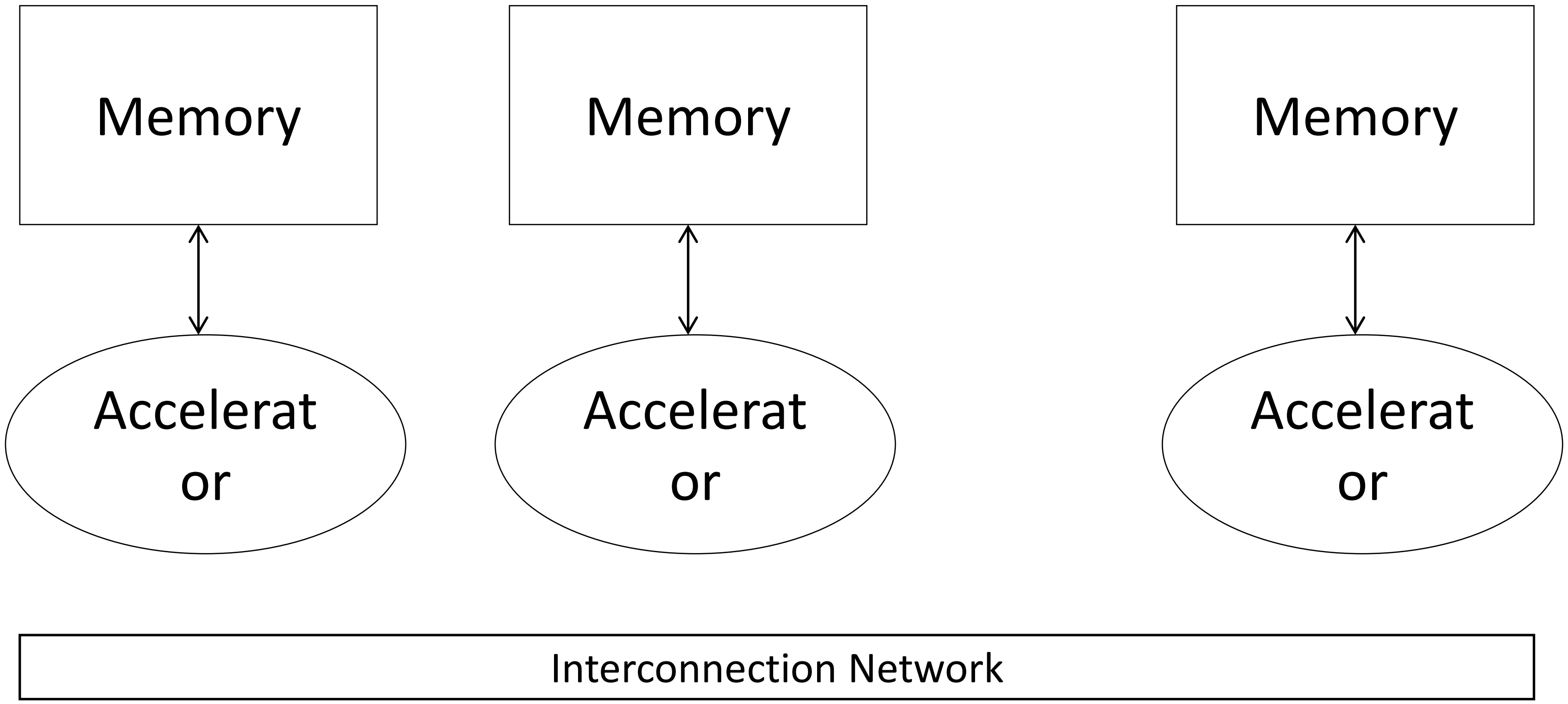
Case Western Reserve University

Outline

- Cluster of Accelerators – Basics

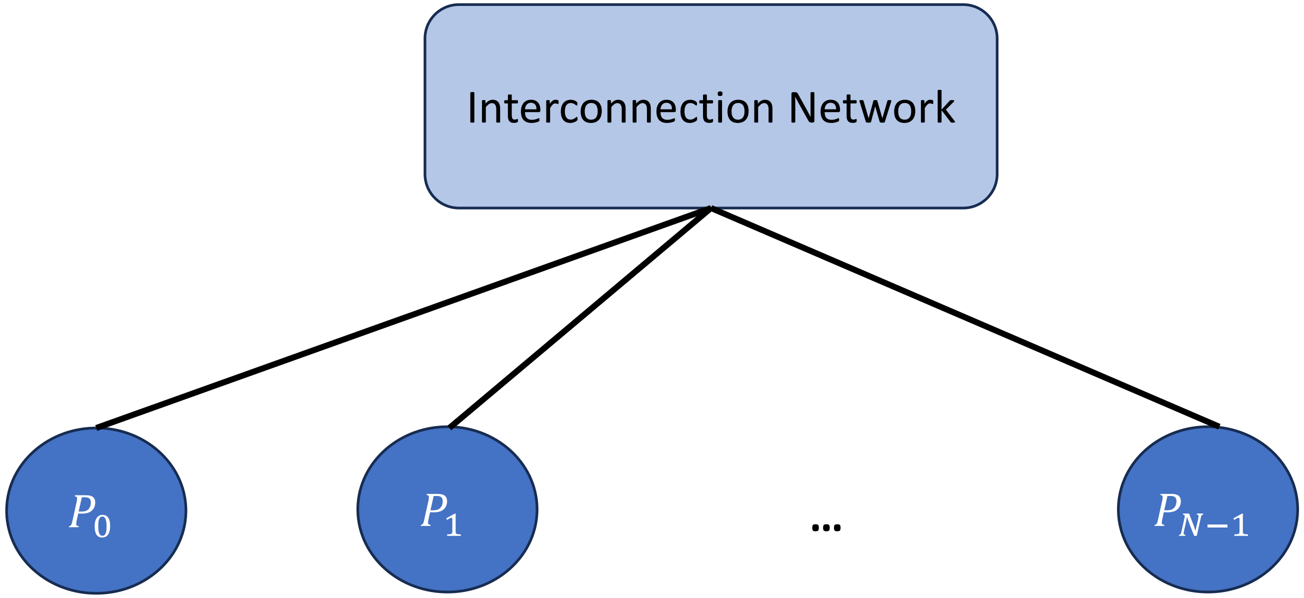
Cluster of Accelerators

- Multiple Accelerator-Memory devices connected through an interconnection network
- Accelerator – CPU,GPU, Systolic Array, …



Cluster of Accelerators

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



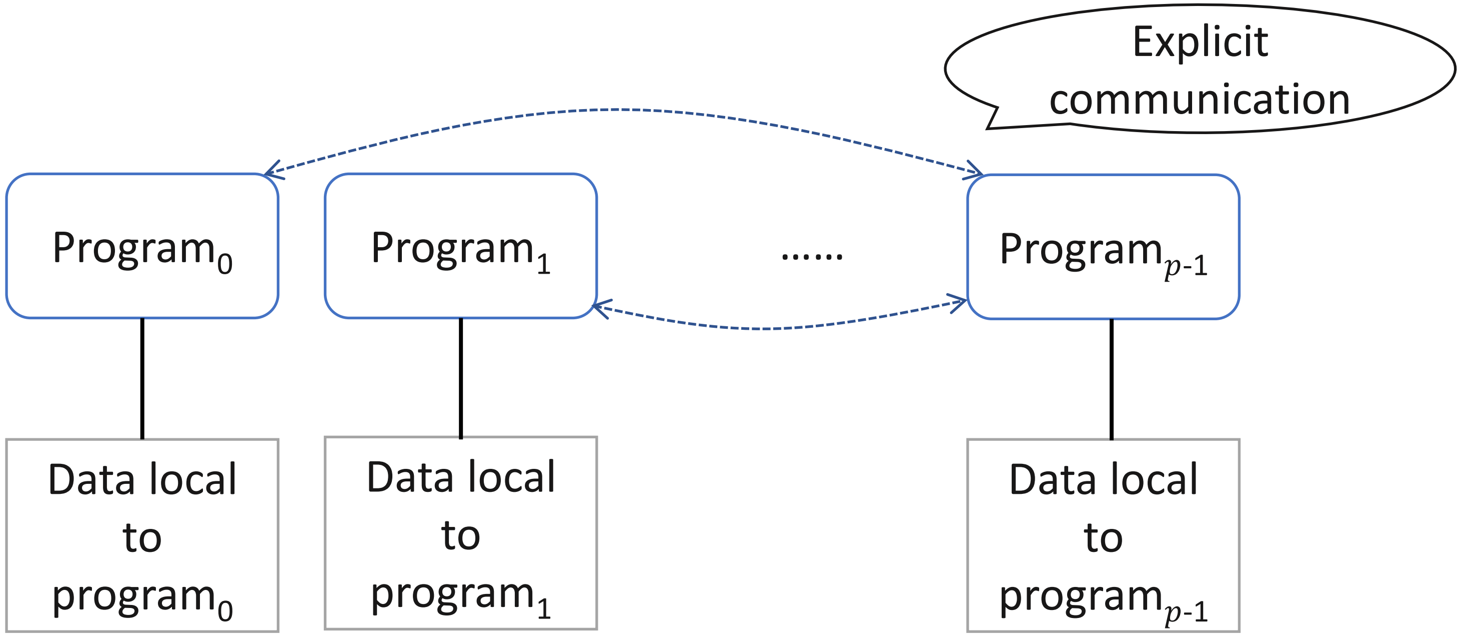
Message Passing Programming Model (1)

- Message passing
  - One of the oldest parallel programming paradigms
  - Widely used
  - Key features
    - Partition address space

      → local data, remote data
    - Explicit parallelization

      → user is responsible to specify and manage concurrency

Message Passing Programming Model (2)



Communication - needs coordination among the communicating processes

Most Popular Framework – Message Passing Interface (MPI)

Message Passing Programming

- How do we program clusters using Message Passing?

- We need to specify the local computations that each processor will execute

- We also need to specify at what time which pairs of processors will communicate and what data they will transfer to each other

Resources

- Slides from University of Stuttgart - https://fs.hlrs.de/projects/par/par_prog_ws/pdf/mpi_3.1_rab.pdf

- Covers all the important details of MPI (744 slides)

- We will cover only enough for its application to AI

Communication Operations (1)

- A pair of Send and Receive is used to implement a communication step

$\boldsymbol{P_0}$

data

Send (sendbuf,

size

size,

Destination process ID

dest)

$\boldsymbol{P_1}$

Receive (
recvbuf,

size,

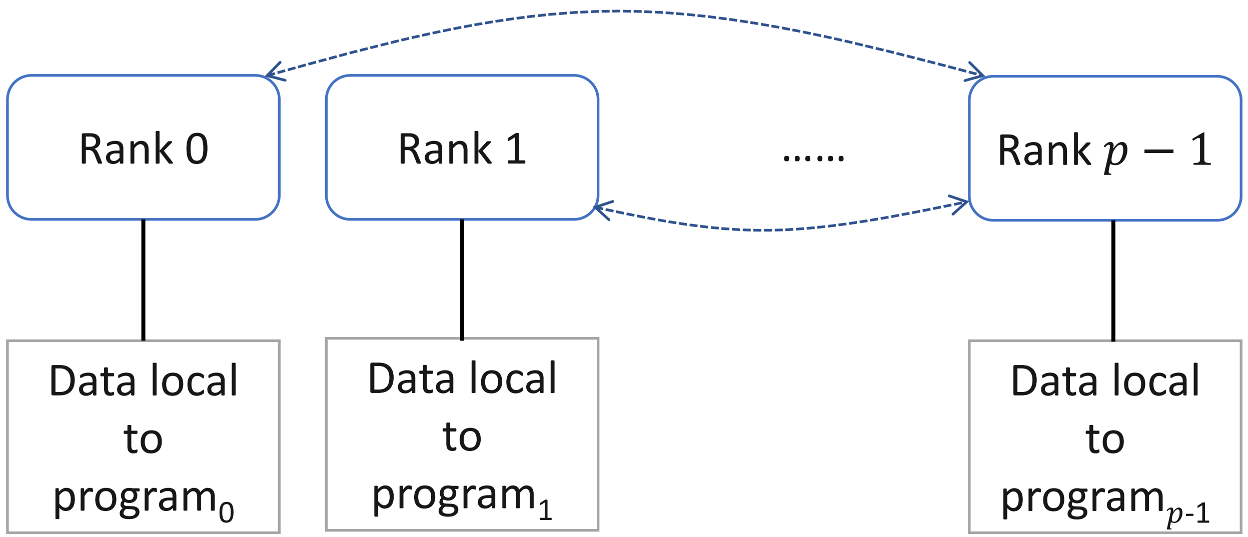
Source process ID

source)

- Processor P_0 *sends* size amount of data to Processor P_1
- Processor P_1 *receives* size amount of data from Processor P_0
- Needs to be symmetric: If not, will lead to hard to debug errors

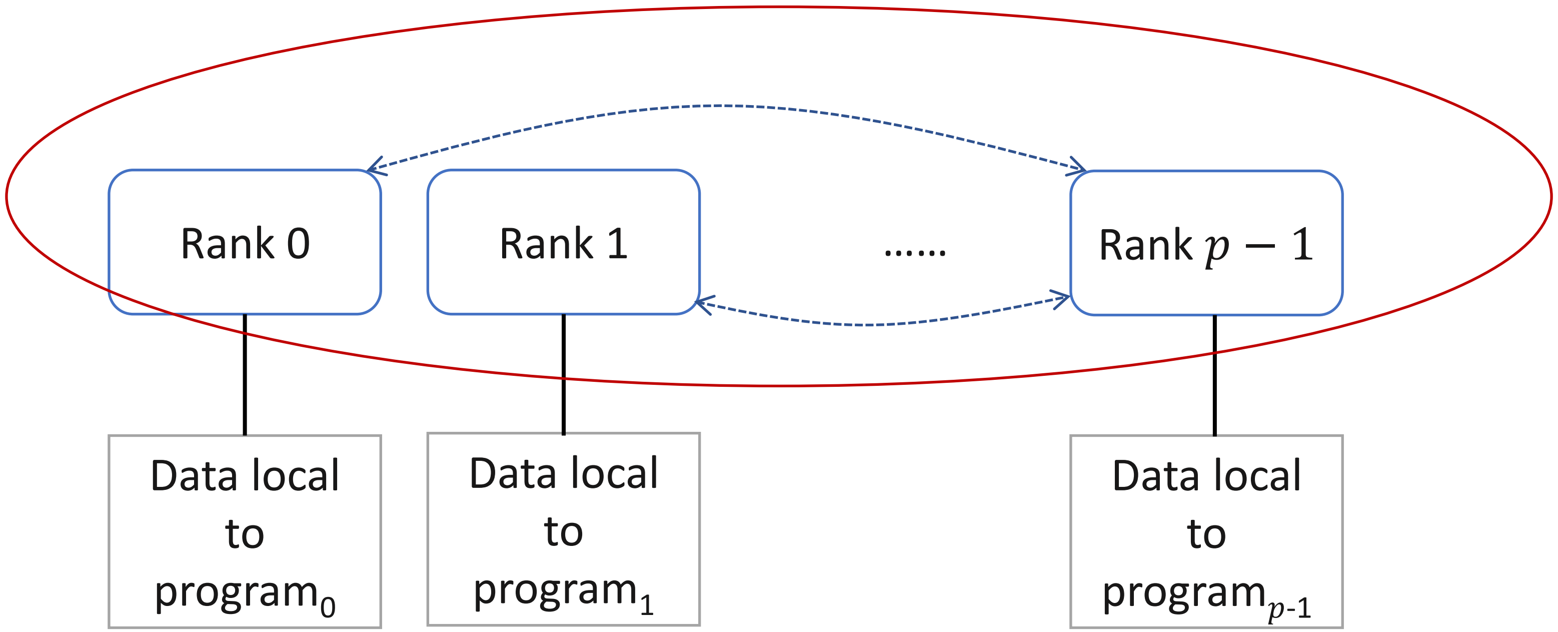
Communication Operations (2)

- How do we identify processor IDs? Ranks



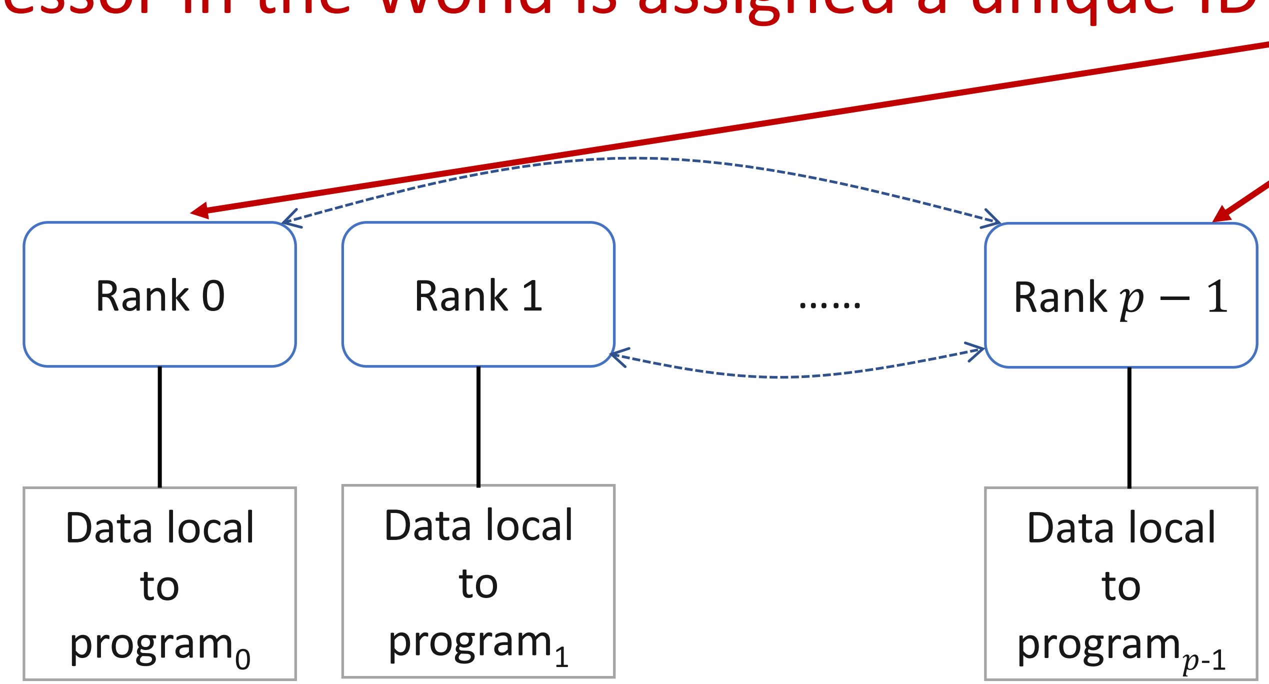
Communication Operations (3)

- How do we identify processor IDs? Ranks
- Collection of all the processors is called **World**



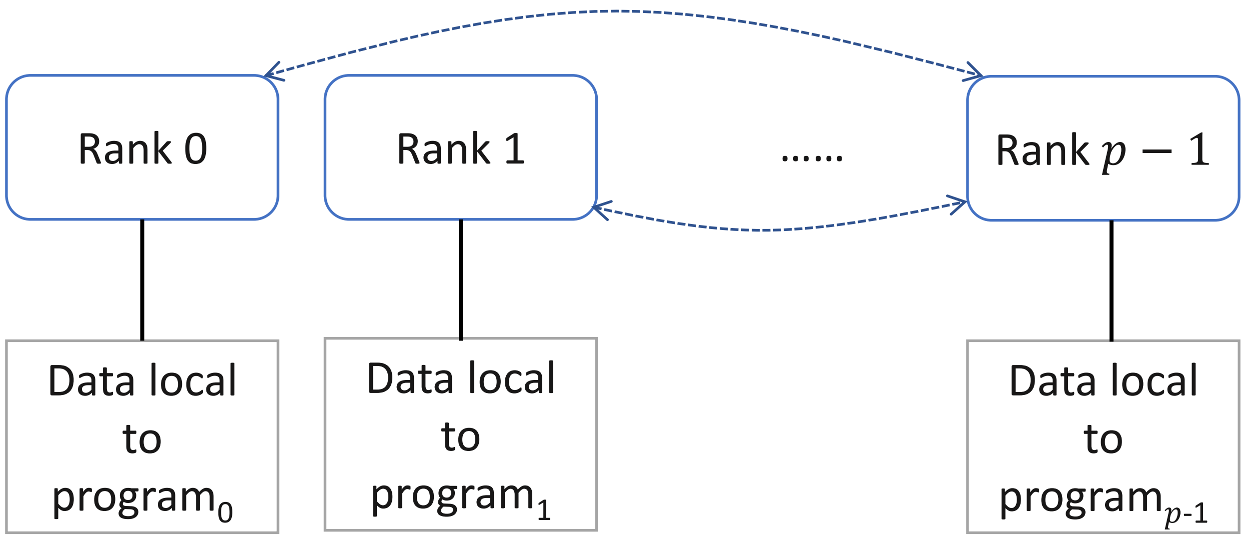
Communication Operations (4)

- How do we identify processor IDs? Ranks
- Each Processor in the World is assigned a unique ID called **Rank**



Communication Operations (5)

- How do we identify processor IDs? Ranks
- It is possible to create partitions of the world. We will not discuss in this class



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

Mpiprogram.c

Each processor will run the exact same program

Actual instructions that get executed may vary depending upon the rank

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

API to initialize the MPI framework

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

Obtain the rank of the process and the size of the world

Anatomy of an MPI Program

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

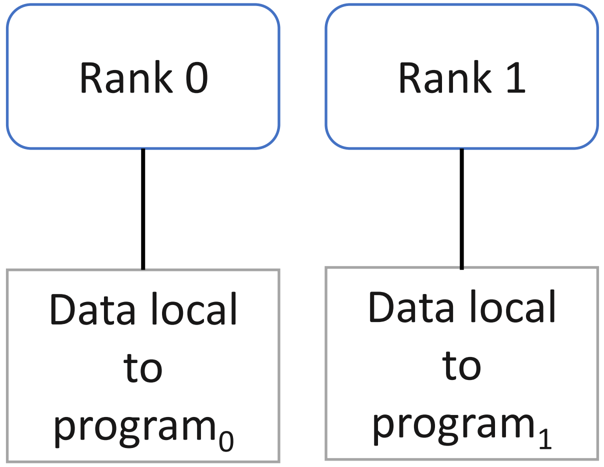
- Do rank specific work

Do work which is specific to the rank

Inter-Process Communication (1)

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);
    $C_{12}(D_1)$; }

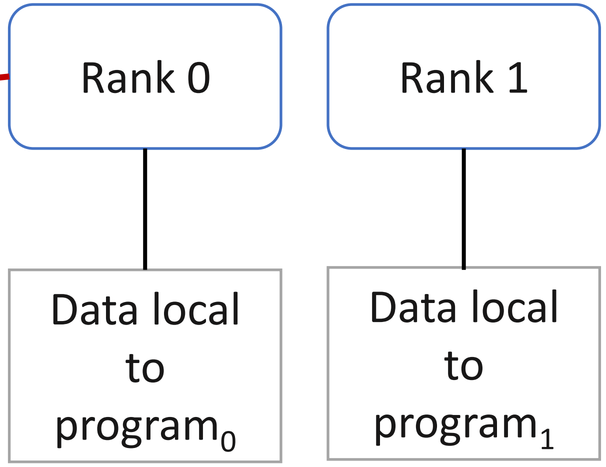


Two processor world

Inter-Process Communication (2)

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);
    $C_{12}(D_1)$; }

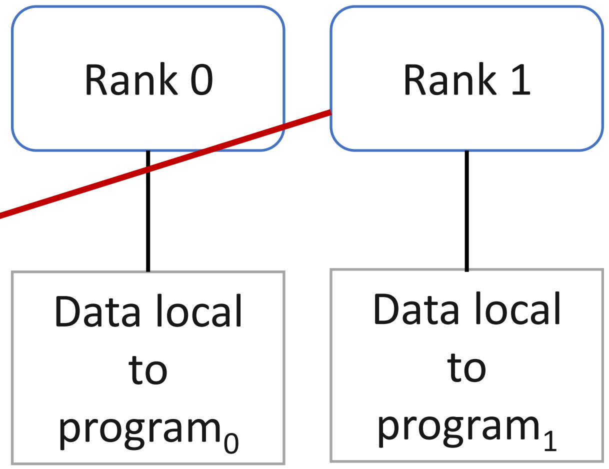


P_0: Executes If portion

Inter-Process Communication (3)

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);
    $C_{12}(D_1)$; }



P_1: Executes Else portion

Inter-Process Communication (4)

P0:

D_0 = C_{00};

Send(D_0, size, P1);

C_{01} ;

P1:

C_{11};

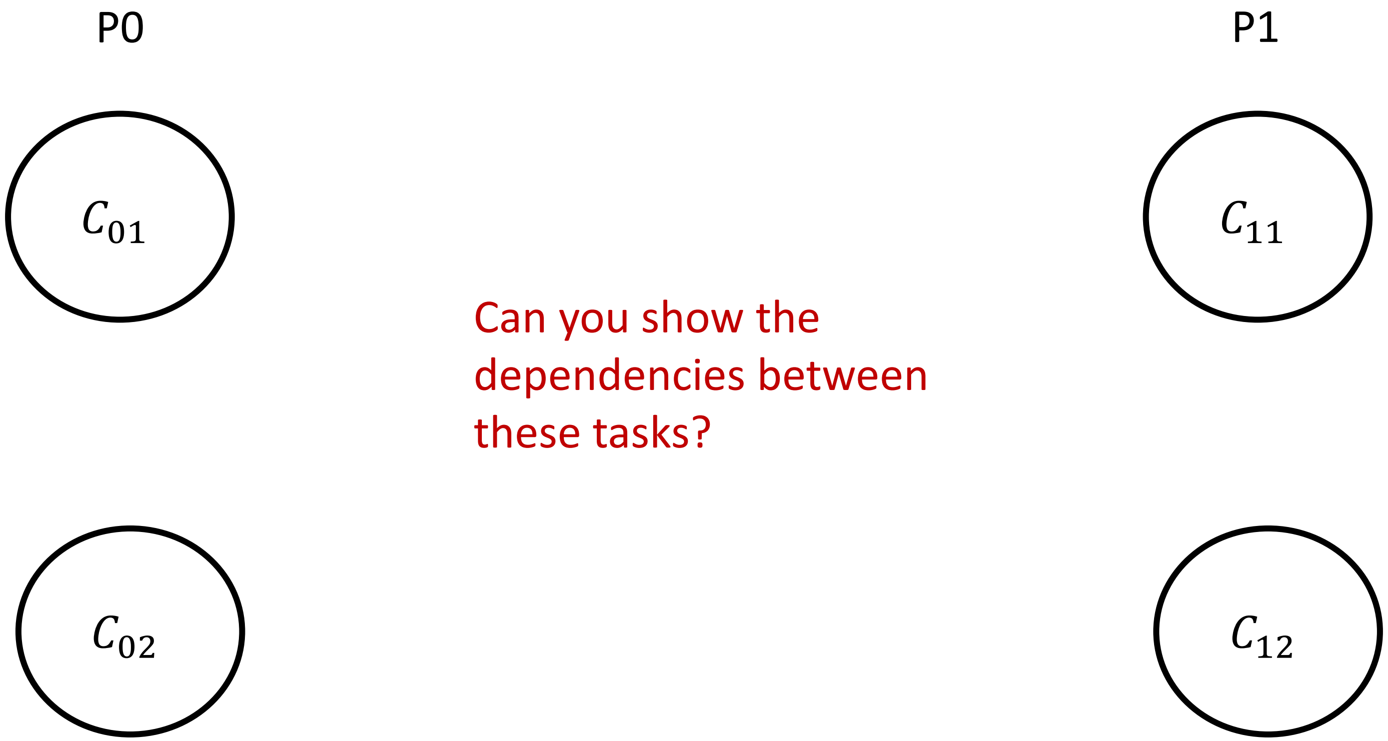
Receive(D_1, size, P1);

$C_{12}(D_1)$

Compute operations

For better visualization, we will represent programs like this. This is equivalent to the previous program

Inter-Process Communication (5)

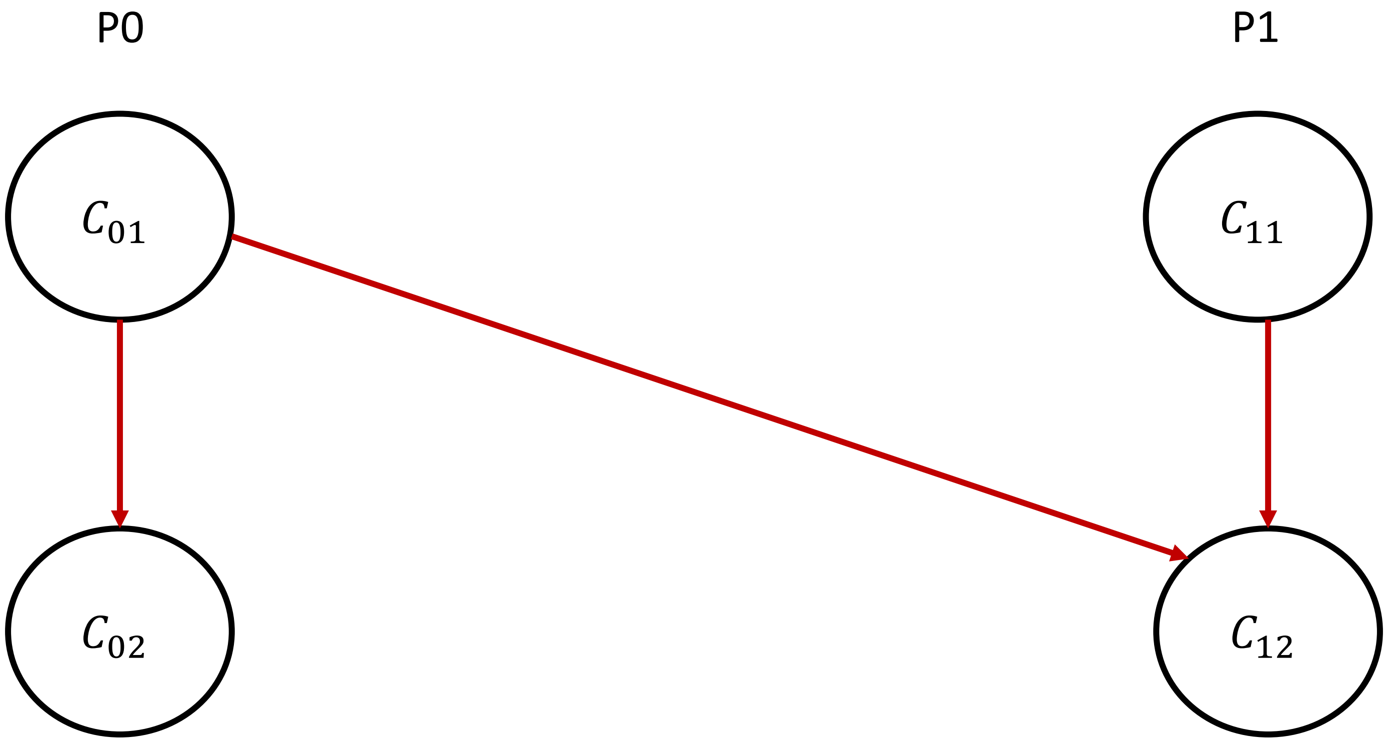


Can you show the dependencies between these tasks?

Inter-Process Communication (6)

Sequential Dependencies due
to same processor

Inter-Process Communication (7)



Dependency across processors due to communication operation

Blocking Semantics

P0:

D_0 = C_{00};

Send(D_0, size, P1);

C_{01} ;

- Does $P0$ wait for the send operation to complete? Blocking/Non-Blocking
- Does MPI directly transfer the data stored on D_0 or does it make another copy? Buffered/Non-Buffered

P1:

C_{11};

Receive(D_1, size, P1);

$C_{12}(D_1)$

Blocking Semantics

P0:

$D_0 = C_{00};$

Send(D_0, size, P1);

C_{01} ;

- Blocking Non-Buffered Send
  - Block sending process
  - Send request to receiving process
  - Wait for receiving process to acknowledge (matched receive operation)
  - Upon receiving acknowledgement, start the transfer
  - No buffers are used for data to be sent

P1:

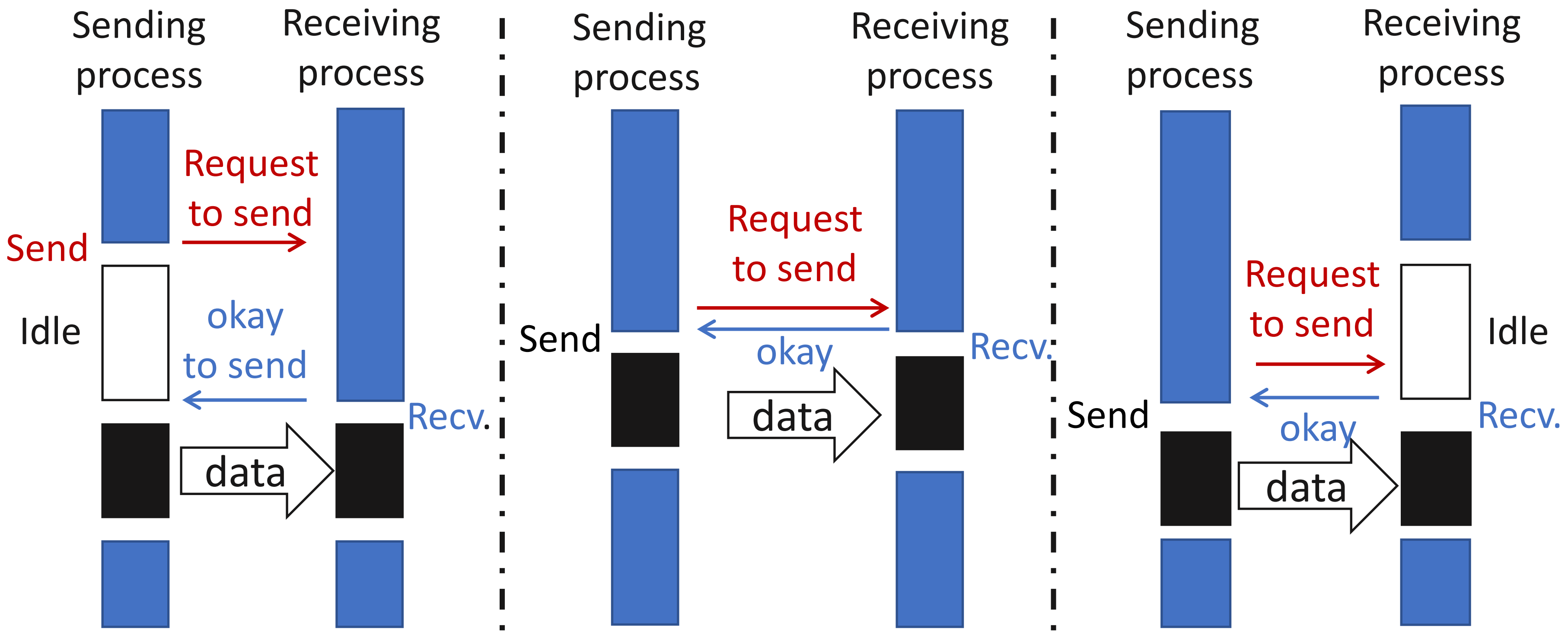
$C_{11};$

Receive(D_1, size, P1);

$C_{12}(D_1)$

Same happens at the receiver end

Blocking Send/Receive (1)



(a) Sender comes first; idling at sender

(b) Sender and Receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at Receiver

Blocking Send/Receive (2)

CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

Issue #2: Deadlocks

blocked

P_0	P_1
1. send($\&a$, 1,1);	1. send($\&a$, 1,0);
2. receive($\&b$, 1,1);	2. receive($\&b$, 1,0);

Deadlocks are very easy in blocking protocols

Blocking Send/Receive (3)

- Non-Block Buffered/Non-buffered addresses these issues, however, it complicates the implementation.

- They are widely used, but we will not discuss here

- You can refer to the Slides from University of Stuttgart - https://fs.hlrs.de/projects/par/par_prog_ws/pdf/mpi_3.1_rab.pdf

Message Passing Programming

- How do we program clusters using Message Passing?

- We need to specify the local computations that each processor will execute

- We also need to specify at what time which pairs of processors will communicate and what data they will transfer to each other
  - P^2 pairs of processors that can communicate at any point
  - Seems too chaotic

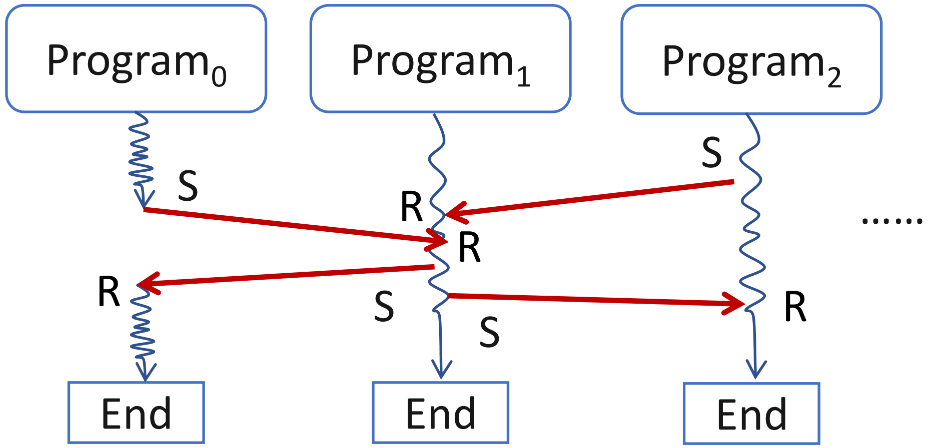
Concurrency Models in Message Passing Programming

- Bring some structure to the programming

- Decide points in the program where processes will not communicate at all or when all processes communicate

Message Passing Program (1)

Most General Model: Asynchronous

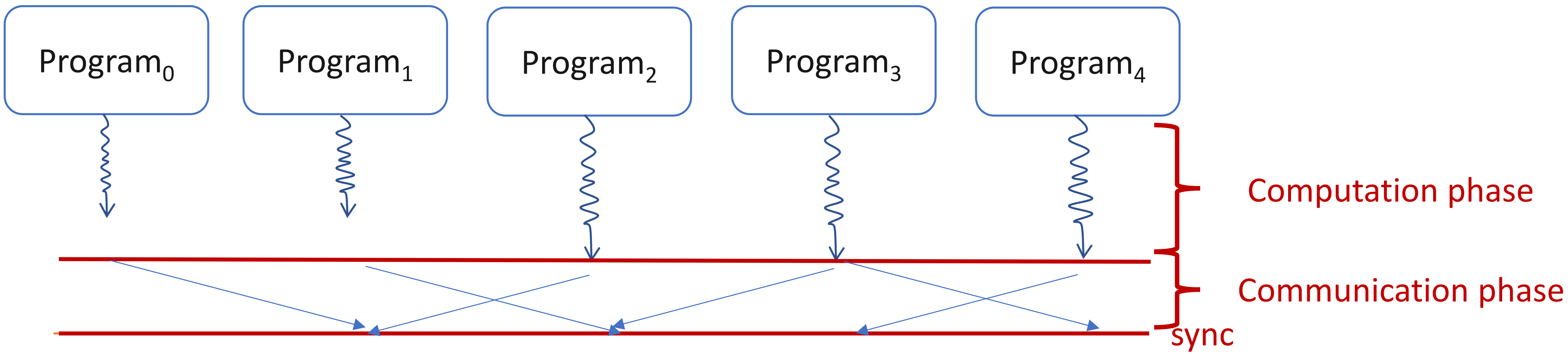


S = send
R = receive

- 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 (2)

Bulk synchronous

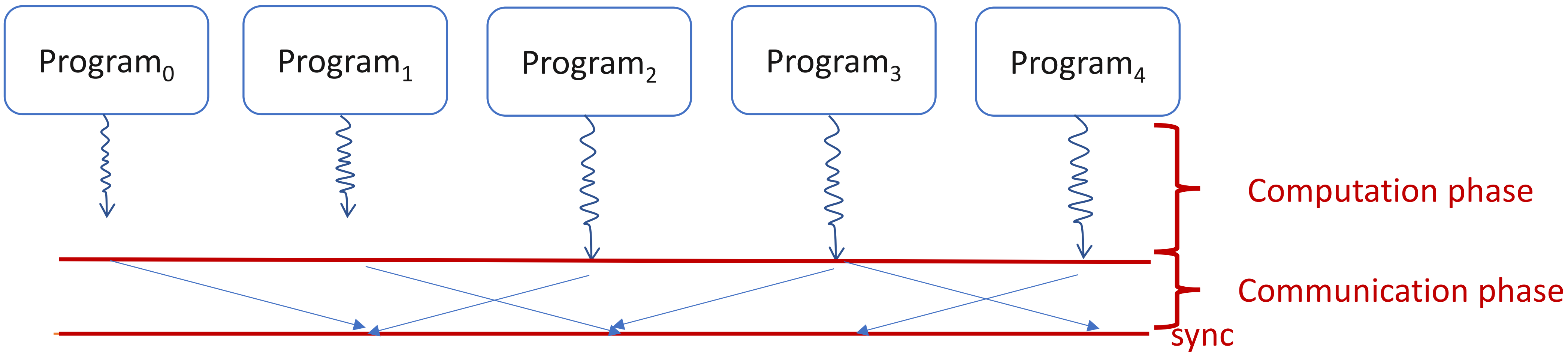


Two phases to the Program

- Computation Phase: Each process executes independently. No communication
- Communication Phase: Processes communicate with each other (usually using group communication primitives – we will discuss in the next class)

Message Passing Program (2)

Bulk synchronous

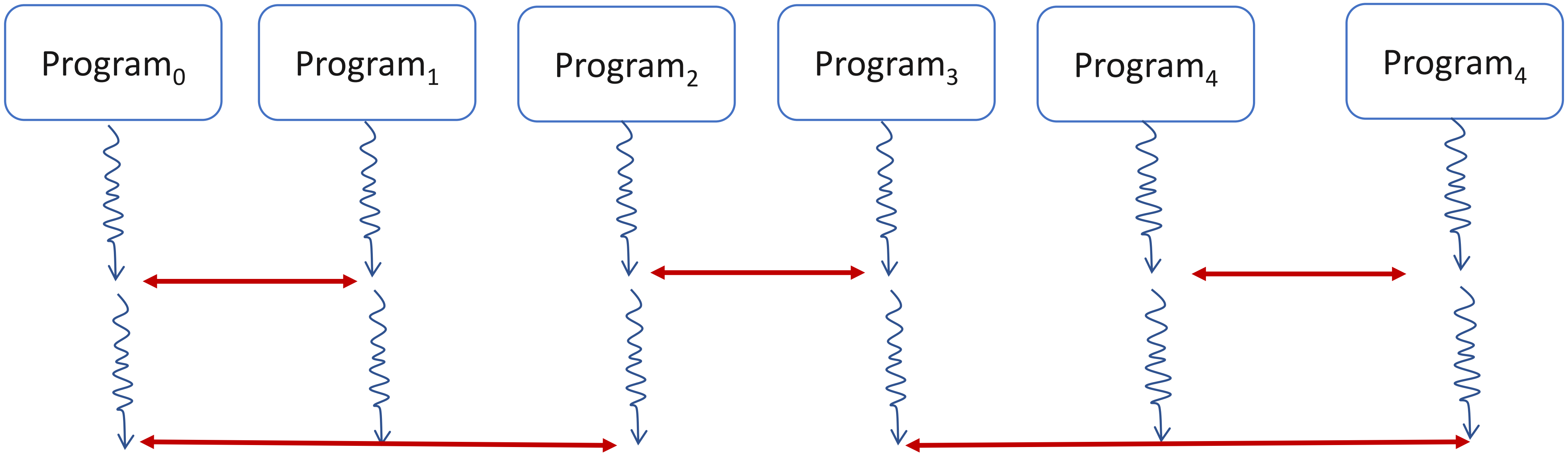


- Simplifies implementation by bringing in structure to the program
- Communication doesn't happen at random times, easier to debug issues
- Using Group communication primitives further simplifies implementations
- Widely use in Machine Learning Training

Message Passing Program (3)

SPMD (Single Program Multiple Data)

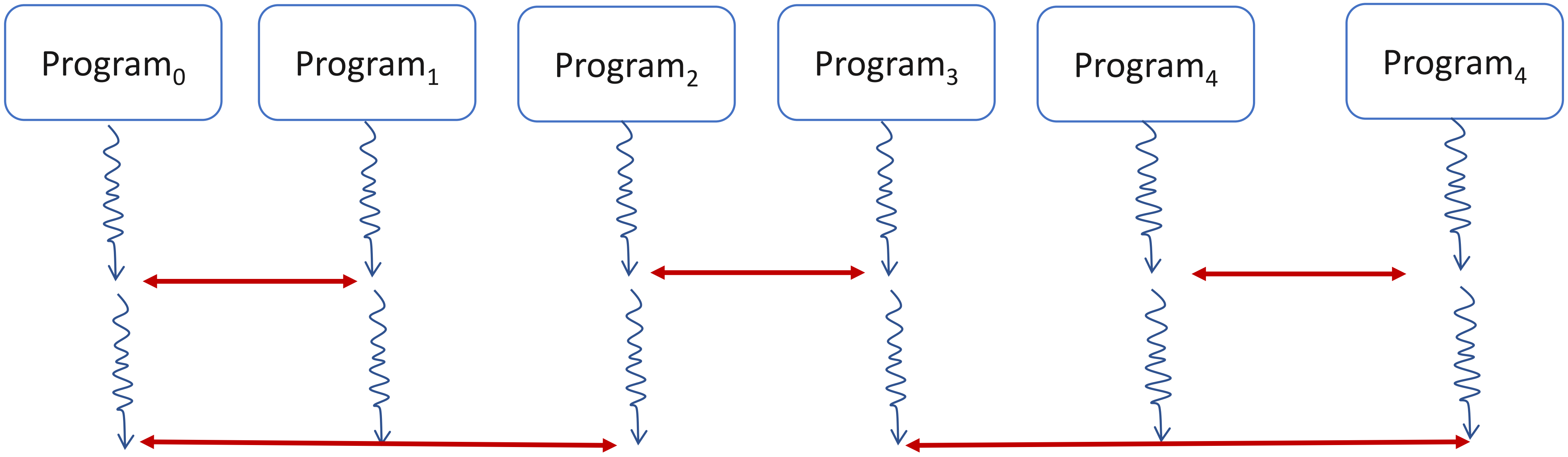
- Code is same in all the processes except for initialization
- Restrictive model, easy to write and debug
- Easy to do performance analysis
- Widely used in Machine Learning Training



Message Passing Program (3)

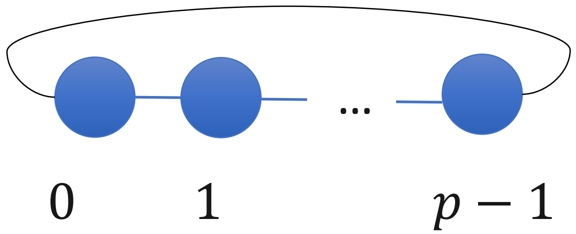
SPMD (Single Program Multiple Data)

- BSP
  - Programs can be different
  - Explicit Barrier (synchronization) between Computation and Communication Phases
- SPMD
  - Programs have to be same – different data
  - No explicit barrier needed (but maybe implied if using blocking send/receive)

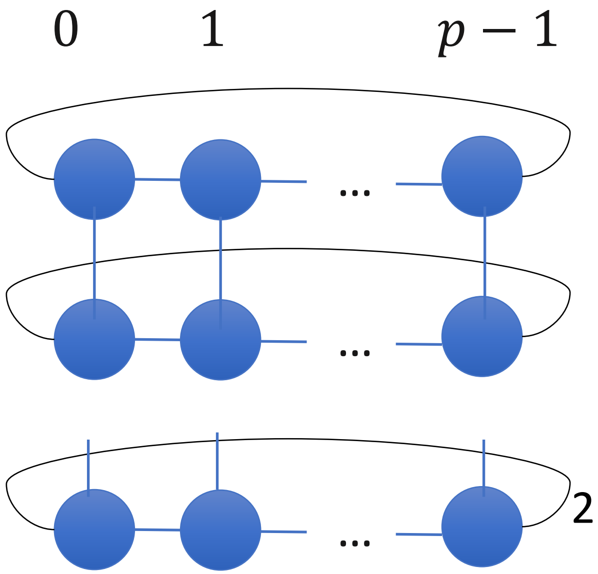


Virtual Topology

- 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



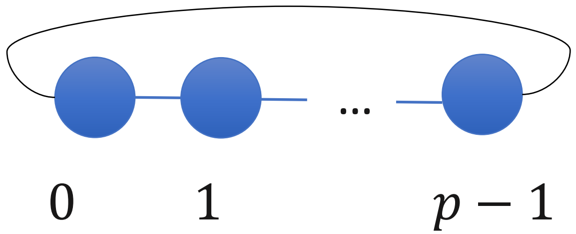
1D Mesh



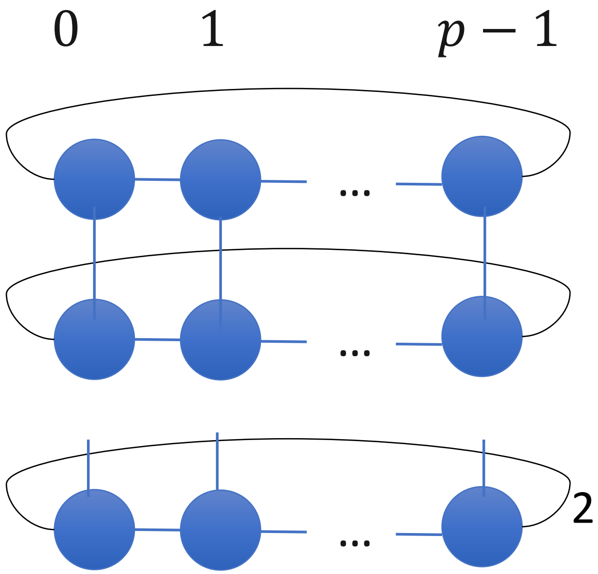
2D Mesh

Virtual Topology

- Define the "connectivity pattern" of the processors
- Helps us in developing more intuitive notions of message passing algorithms
- (also has uses in optimizing the mapping of message passing algorithms onto real interconnection networks)



1D Mesh



2D Mesh

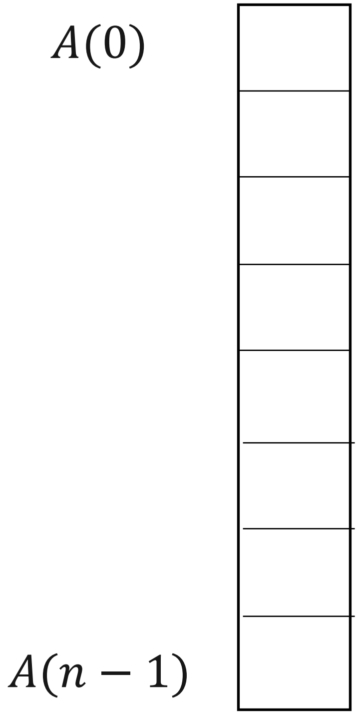
Message Passing Programming Paradigm

- 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.

Adding Using Message Passing on 1D Mesh (1)

Output $= \sum_{i=0}^{n-1} A(i)$ in $A(0)$



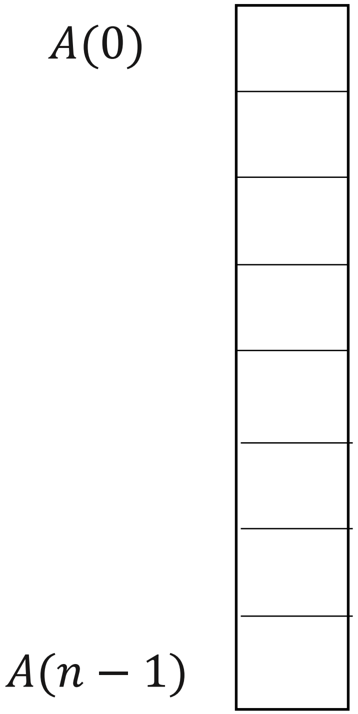
Adding Using Message Passing on 1D Mesh (1)

- Concurrency Model? SPMD
- How many Processors? n
- What does each processor do?
- Send/receive commands?
- Virtual topology – 1D mesh

Adding Using Message Passing on 1D Mesh (2)

- Key Idea: Recursive doubling

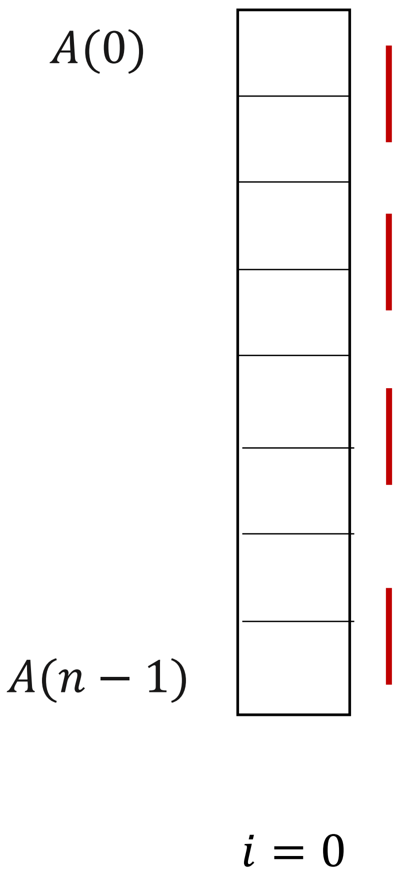
- In iteration i:
  - Processors j and $j + 2^i$ communicate, where $j = k.2^{i+1}$
  - Processor j computes
  - j – Active Processors



Adding Using Message Passing on 1D Mesh (3)

- Key Idea: Recursive doubling

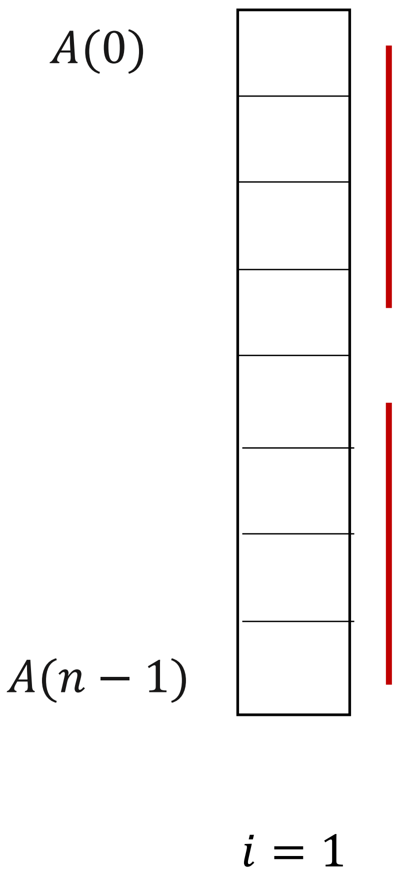
- In iteration i:
  - Processors j and $j + 2^i$ communicate, where $j = k.2^{i+1}$
  - Processor j computes
  - 0, 2, 4, 6 – Active Processors



Adding Using Message Passing on 1D Mesh (4)

- Key Idea: Recursive doubling

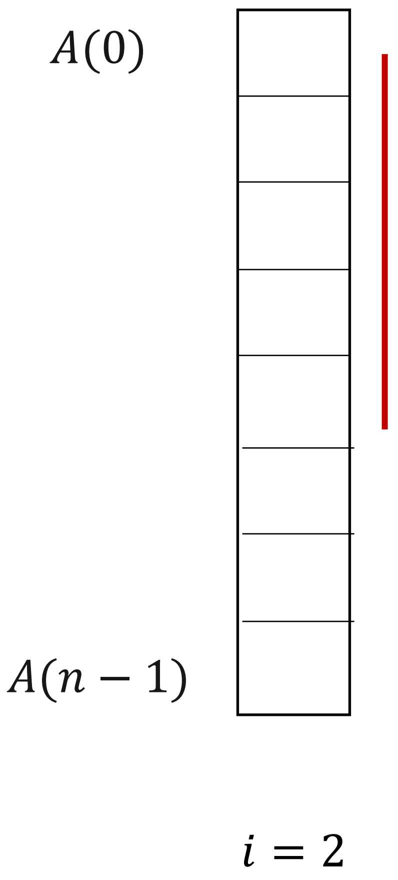
- In iteration i:
  - Processors j and $j + 2^i$ communicate, where $j = k.2^{i+1}$
  - Processor j computes
  - $0,\ 4\ -$ Active Processors



Adding Using Message Passing on 1D Mesh (5)

- Key Idea: Recursive doubling

- In iteration i:
  - Processors j and $j + 2^i$ communicate, where $j = k.2^{i+1}$
  - Processor j computes
  - $0\ -$ Active Processors



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, …}

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 $\log_2 n - 1$
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 $\times$ 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.

Next Class

- 11/13 Lecture 21
  - Distributed Matrix Multiplication on a Cluster of Accelerators
  - Communication Primitives

Thank You

- Questions?

- Email: sanmukh.kuppannagari@case.edu