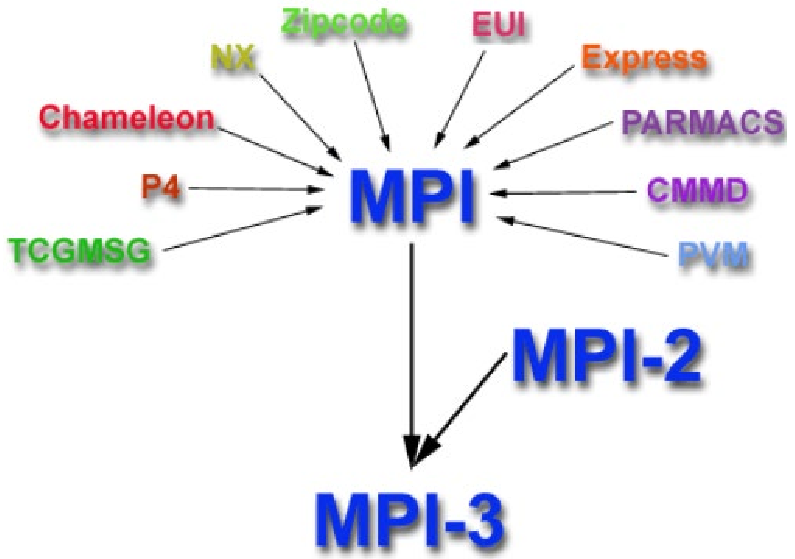
Message Passing Programming

Lecture 10



Outline

- **Message Passing Overview**
- **MPI**
  - Initialization, Finalization and Abort
  - Point-to-point Communication
  - Process Groups and Communicators
  - Collective Communication
- **Summary**

Message Passing: Overview

- Abstraction of a parallel computer with a distributed address space
- Processes running on processors exchange data by **message-passing using communication operations**
- **Explicit parallelism**:
  - Programmer responsible for identifying parallelism

Just message me!

MESSAGE PASSING INTERFACE (MPI)

Message Passing Interface: Overview

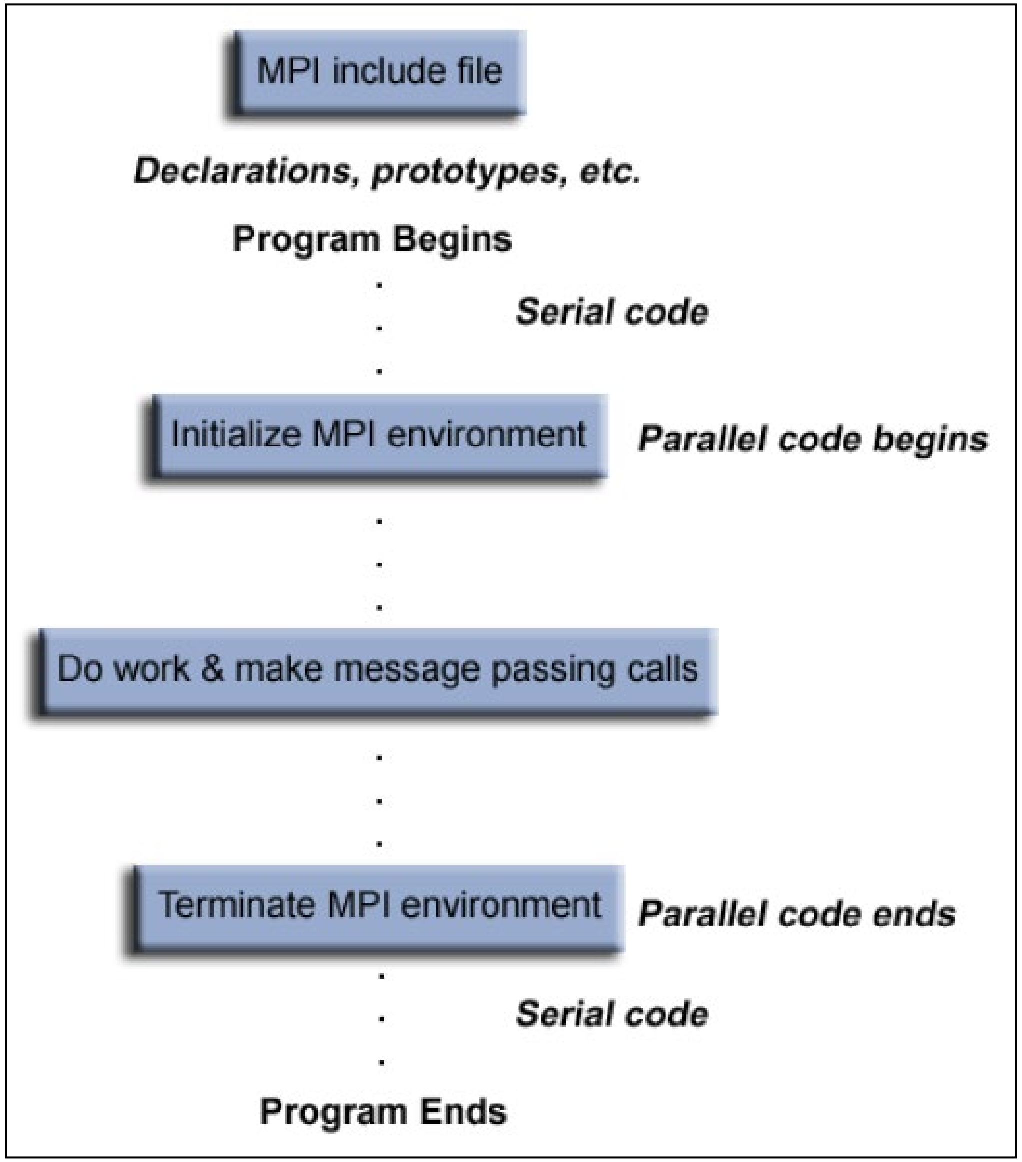
- Message-Passing Interface (MPI) is a standardization of a message-passing library interface specification

- Provided as a set of library calls:
  - Directly callable: C, C++, Fortran-77, and Fortran-95
  - Via interface: Python, Java, C#, ….

- Four versions
  - MPI-1 (1994): static process model
  - MPI-2 (1996): dynamic process creation/management, one-sided communication, and parallel I/O
  - MPI-3 (2012): support for Fortran 2008
  - MPI-4 (2021): large-count versions of many routines to address the limitations of using an int or INTEGER for the count parameter, partitioned communications

- Implementations: MPICH, LAM/MPI, OpenMPI

MPI Program Structure



Initialization, Finalization and Abort

```
int MPI_Init(int* argc, char** argv[])
```

- Initialize the MPI program
- Must be **called only once** and **before** any other MPI routines

```
int MPI_Finalize(void)
```

- Terminate all MPI processing
- Must be the **last MPI call**

```
int MPI_Abort(MPI_Comm comm, int errorCode)
```

- Force all processes to terminate
- Return the errorcode to mpirun

MPI Program Overview

- 3 general steps:
  1. Initialize communications
  2. Do the communications necessary for coordinating computation
  3. Exit from message-passing system when done

- 6 basic functions to write a program:
  - `MPI_Init`
  - `MPI_Comm_size`
  - `MPI_Comm_rank`
  - `MPI_Send`
  - `MPI_Receive`
  - `MPI_Finalize`

MPI C Program – Hello World 2

```
#include <stdio.h>
#include <mpi.h>
#include <string.h>

int main(int argc, char **argv)
{
  int rank, size, tag, i;
  char message[20];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  tag = 100;

  //continue on next slide
```

MPI C Program – Hello World 2 (cont)

```
  ......
  if(rank == 0) {
    strcpy(message, "Hello World 2");
    for (i=1; i<size; i++)
        MPI_Send(message, 14, MPI_CHAR,
                 i, tag, MPI_COMM_WORLD);
  } else {
    MPI_Status status;
    MPI_Recv(message, 14, MPI_CHAR,
             0, tag, MPI_COMM_WORLD, &status);
  }
  printf( "node %d : %.13s\n", rank, message);
  MPI_Finalize();
  return 0;
}
```

Point-to-Point Communication

- Blocking
  - **`MPI_Send`**
  - **`MPI_Recv`**
  - **`MPI_Sendrecv`**
  - **`MPI_Sendrecv_replace`**

- Non-blocking
  - **`MPI_Isend`**
  - **`MPI_Irecv`**

- Blocking and non-blocking operations can be **mixed**
  - Data sent by **`MPI_Isend`**() can be received by **`MPI_Recv`**()
  - Data sent by **`MPI_Send`**() can be received by **`MPI_Irecv`**()

MPI Messages Format

- Message =
  data (actual data that you want to send/receive) +
  envelope (how to route the data)

- **Data** =
  start-buffer (address where data starts) +
  count (number of elements of data in the message) +
  datatype (type of data to be transmitted) +

- **Envelope** =
  destination or source (using rank in a communicator) +
  tag (an arbitrary number to distinguish among messages) +
  communicator (send/receive must specified
  the same communicator)

Send and Receive

```
int MPI_Send(void* buf, int count,
          MPI_Datatype dt,int dest, int tag,
          MPI_Comm c);

int MPI_Recv(void* buf, int count,
          MPI_Datatype dt, int src, int tag,
          MPI_Comm c, MPI_Status *status);
```

- Received message must be less than or equal to the length of the receive buffer
- For receiving message, use:
  - src = MPI_ANY_SOURCE: from any process
  - tag = MPI_ANY_TAG: message with any tag
- MPI_Status is a structure with:
  - MPI_SOURCE, MPI_TAG, MPI_ERROR

Semantic of MPI Operations

Local view	Global view
Blocking Return from a library call indicates the user is allowed to reuse resources specified in the call	**Synchronous** Communication operation does not complete before both processes have started their communication operation
Non-blocking A procedure may return before the operation completes, and before the user is allowed to reuse resources specified in the call	**Asynchronous** Sender can execute its communication operation without any coordination with the receiver

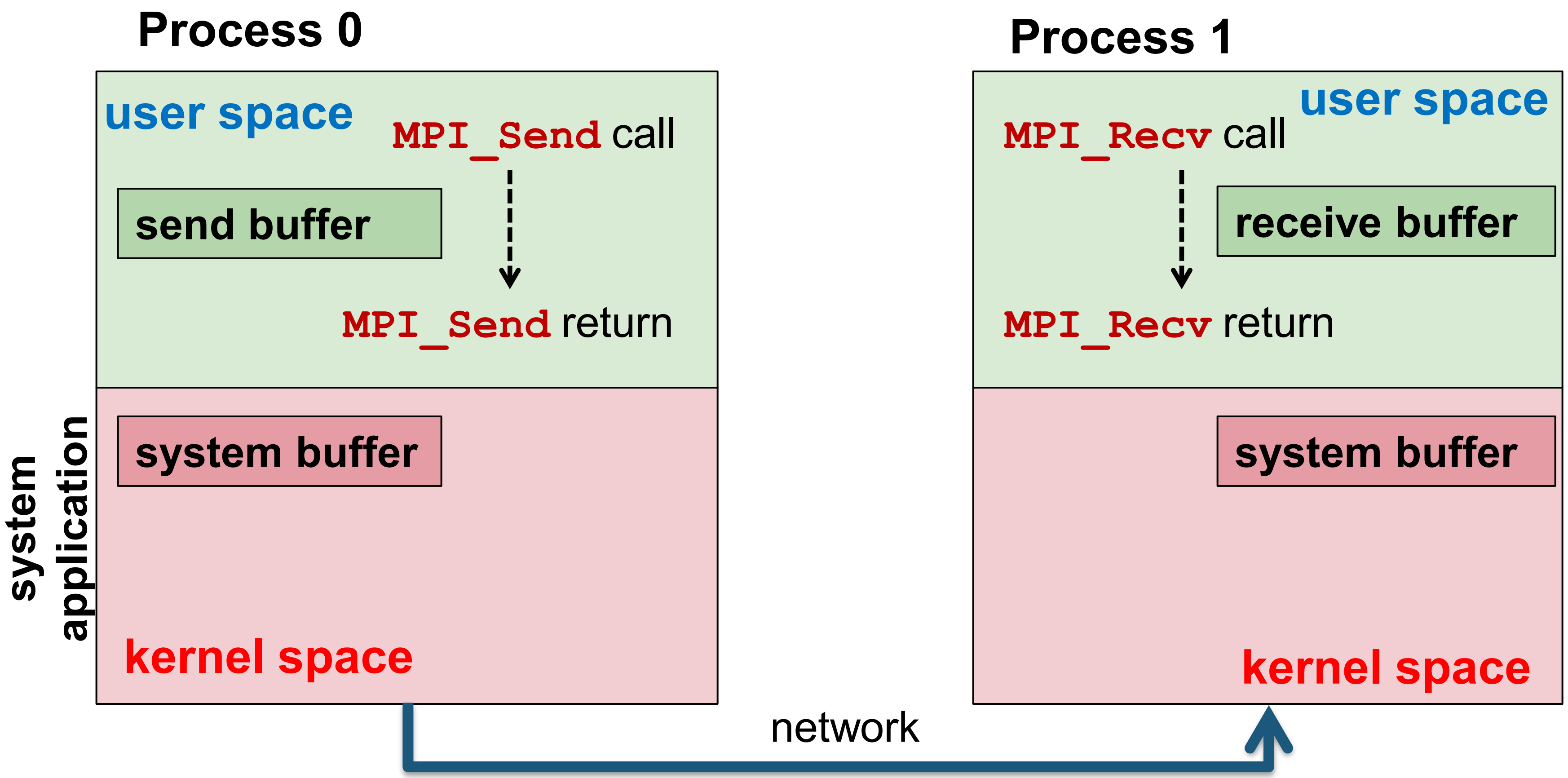
Send and Receive Operations in MPI

	Synchronous	Asynchronous

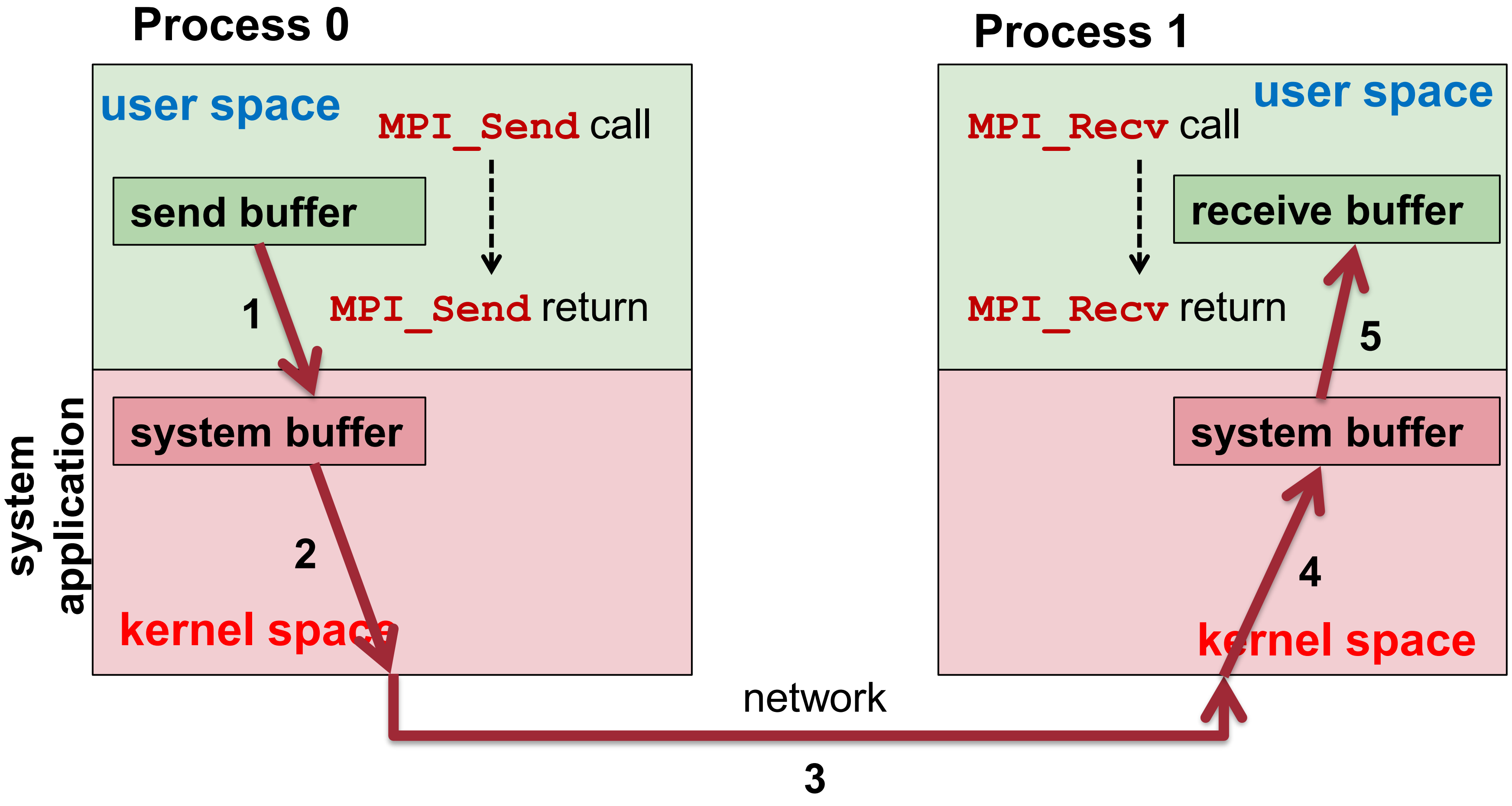
Blocking	`MPI_SSend`
	`(MPI_Mrecv)`
	`MPI_RSend`
		`May be buffered:`
		`MPI_Send`
		`MPI_Recv`
		`Buffered:`
		`MPI_Bsend`

Non-blocking	`MPI_ISSend` `(MPI_ImRecv)`	`MPI_ISend` `MPI_Irecv`

MPI Sending and Receiving



MPI Sending and Receiving



Order of Receive Operations

- Two processes (one sender, one receiver)
  - A sender sends two or more messages to the same receiver, messages delivered in the order in which they have been sent

- If more than two processes
  - Message delivery order not guaranteed!

Example: Three Processes

```
if (my_rank == 0) {
   MPI_Send (sendbuf1, count, MPI_INT, 2, tag, comm);
   MPI_Send (sendbuf2, count, MPI_INT, 1, tag, comm);
}
else if (my_rank == 1) {
   MPI_Recv (recvbuf1, count, MPI_INT, 0, tag, comm, &status);
   MPI_Send (recvbuf1, count, MPI_INT, 2, tag, comm);
}
else if (my_rank == 2) {
   MPI_Recv (recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
      &status);
   MPI_Recv (recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
      &status);
}
```

P_0

P_1

P_2

1. P_0 sends $\mathbf{m_1}$ to P_2 then sends $\mathbf{m_2}$ to P_1
2. P_1 on receiving $\mathbf{m_2}$ from P_0 sends $\mathbf{m_2}$ to P_2

- What is the ordering of messages at P_2?

Deadlocks in MPI: Message Order

- Deadlock occurs when message passing cannot be completed

```
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
    MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
    MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
}
else if (my_rank == 1) {
    MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
    MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```

- Process 0 waits for process 1 and vice versa
  - ➔ always deadlock!

Deadlocks in MPI: System Buffer

- Deadlock occurs if the runtime system does not use system buffers or if the system buffers used are too small

```
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
   MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
   MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
}
```

- No system buffer / buffer too small ➔ deadlock (because both sends cannot complete)

Deadlock-Free in MPI

- An MPI program is called secure if the correctness of the program does not depend on assumptions about specific properties of the MPI runtime system

```
MPI_Comm_rank (comm, &myrank);
if (my_rank == 0) {
    MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
    MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
    MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
    MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```

No Deadlock – if we specify the execution order of send and receive

Example: Deadlock-Free Logical Ring

- Processes with an **even** rank: **send → receive**
- Processes with an **odd** rank: **receive → send**

Phase	Process 0	Process 1	Process 2	Process 3
1	MPI_Send() to 1	MPI_Recv() from 0	MPI_Send() to 3	MPI_Recv() from 2
2	MPI_Recv() from 3	MPI_Send() to 2	MPI_Recv() from 1	MPI_Send() to 0

Four Logical Processes

Phase	Process 0	Process 1	Process 2
1	MPI_Send() to 1	MPI_Recv() from 0	MPI_Send() to 0
2	MPI_Recv() from 2	MPI_Send() to 2	-wait-
3		-wait-	MPI_Recv() from 1

Three Logical Processes

Process Groups and Communicators

- Process Groups
- Communicators
- Process Virtual Topologies

Process Group

- An ordered set of processes
  - Each process has a unique rank

- A process may be a member of multiple groups

  ➔ may have different ranks in each of these groups

- MPI system handles the representation and management of process groups

Communicator

- Communicator is the communication domain for a group of processes

- Two types:

1. **Intra-communicators**
   - Support the execution of arbitrary collective communication operations on a single group
   - Default: `MPI_COMM_WORLD`

2. **Inter-communicators**
   - Support the point-to-point communication operations between two process groups

Group and Communication: Why?

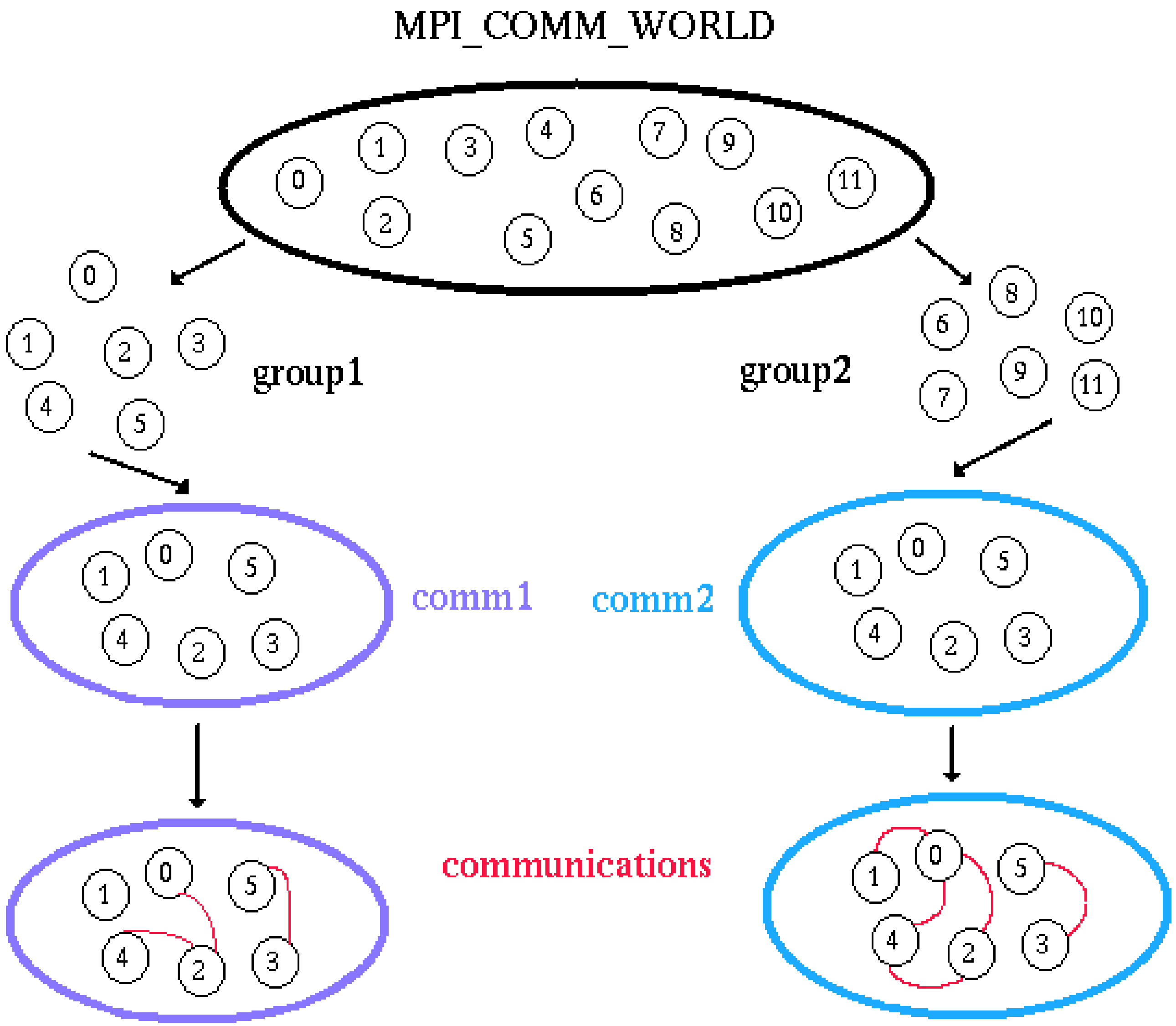
- Allow us to organize tasks, based on functions, into task groups

- Enable collective communication operations across a subset of related tasks

- Provide the basis for user-defined virtual topologies

- Provide for safe communication

Example



MPI_COMM_WORLD
0
1
3
4
7
9
11
2
5
6
8
10
group1
group2
comm1
comm2
communications

Operations on Process Groups

Functionality	MPI Call
Obtain a new group	`MPI_Comm_group`
Get size of a group	`MPI_Group_size`
Rank a process in a group	`MPI_Group_rank`
Group union	`MPI_Group_union`
Group intersection	`MPI_Group_intersection`
Group difference	`MPI_Group_difference`
Group inclusion	`MPI_Group_incl`
Group exclusion	`MPI_Group_excl`
Group compare	`MPI_Group_compare`
Delete group	`MPI_Group_free`

Operations on Communicators

Functionality	MPI Call
Get size of communicator	`MPI_Comm_size`
Get rank of process in communicator	`MPI_Comm_rank`
Create communicator	`MPI_Comm_create`
Compare communicators	`MPI_Comm_comp`
Duplicate communicator	`MPI_Comm_dupl`
Split communicator	`MPI_Comm_split`
Delete communicator	`MPI_Comm_free`

Process Virtual Topologies

- Sometimes it is useful to have an alternative representation and access
  - e.g. processes communicating with neighbor processes only in a mesh pattern
- Create topologies where neighbors are easily addressable

0	1	2	3
4	5	6	7
8	9	10	11

12 Processes

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

row 2, column 3

Virtual 3x4 Grid using 12 Processes

Virtual Topology Operations

- Virtual topology: a communicator with a Cartesian style of addressing the ranks of the processes

Functionality	MPI Call
Create a Cartesian topology	`MPI_Cart_create`
Get info on Cartesian topology	`MPI_Cart_get`
Get number of dimension	`MPI_Cartdim_get`
Comm rank → Cartesian coords	`MPI_Cart_coords`
Cartesian coords → comm rank	`MPI_Cart_rank`
Access neighbors in Cartesian coords	`MPI_Cart_shift`

Collective Communication

- Operations that involve all processes in a communicator
  - Otherwise: **deadlock**
  - Blocking operations by default

- Two types of operations:
  - **Scatter**: from 1 process to many processes
  - **Gather**: from many processes to 1 process
    - With accumulation (reduction) using an arithmetic operation OR
    - Without accumulation

MPI Barrier

- The only collective synchronization operation

- No data movement

```
int MPI_Barrier(MPI_Comm comm);
```

- As with any barrier, all processes in communicator must execute the function call

- Processes block until all processes of the communicator reach the barrier

Measuring Program Timings

- Measure the parallel execution time

```
double MPI_Wtime (void)
```

  - Return the absolute time elapsed between the start and the end of a program part

- Get resolution of `MPI_Wtime()`

```
double MPI_Wtick (void)
```

One-to-many, many-to-one, …

COLLECTIVE COMMUNICATION

Communication Operations

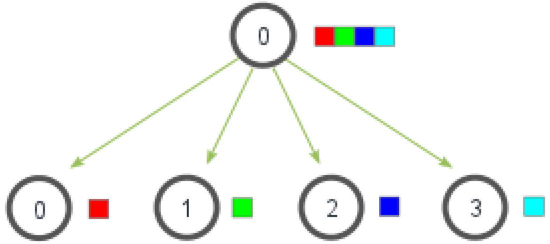
- Examples:
  - Single transfer
  - Gather (scatter)
  - Single-broadcast (multi-broadcast)
  - Single-accumulation (multi-accumulation)
  - Total exchange

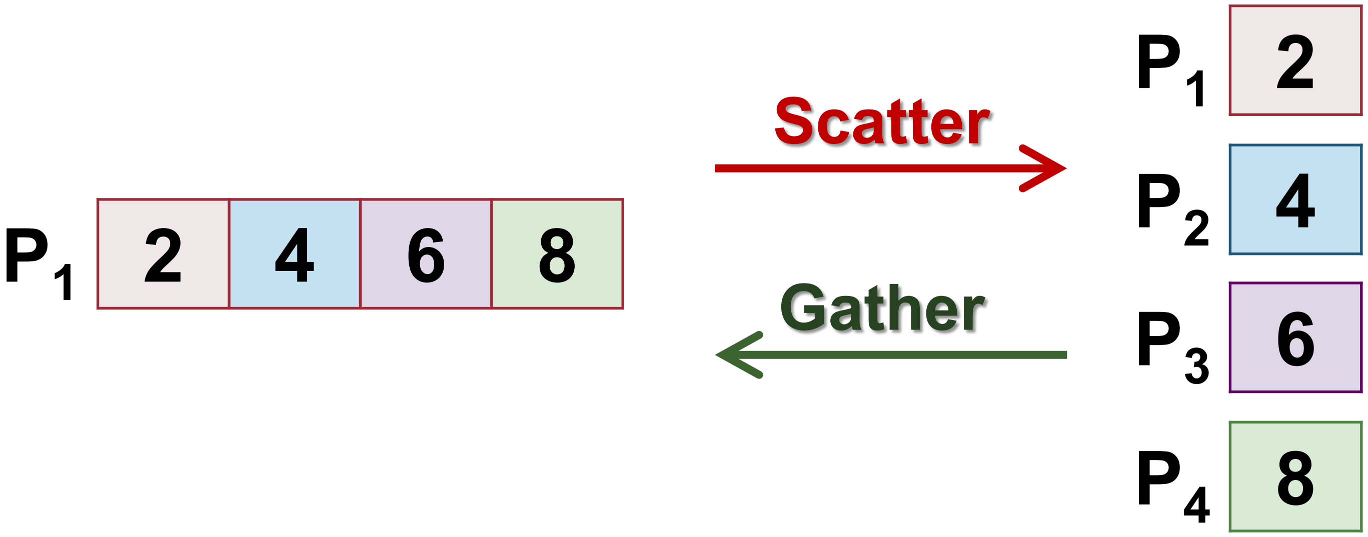
- Consider p identical processors, P_1, P_2, .., P_p, and with processor rank i in {1, 2, .., p}

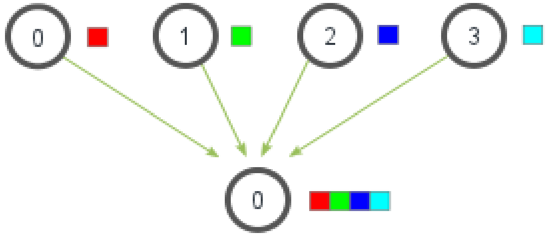
Single Transfer

- Point-to-point communication
  - Send: specifies send buffer, receiving (destination) processor rank
  - Receive: receive buffer, sending (source) processor rank

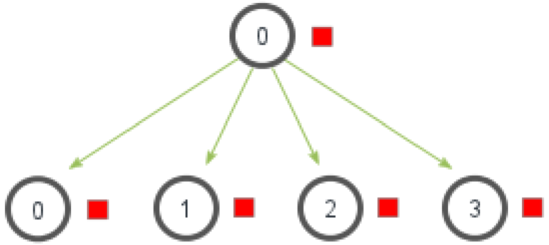
Scatter and Gather

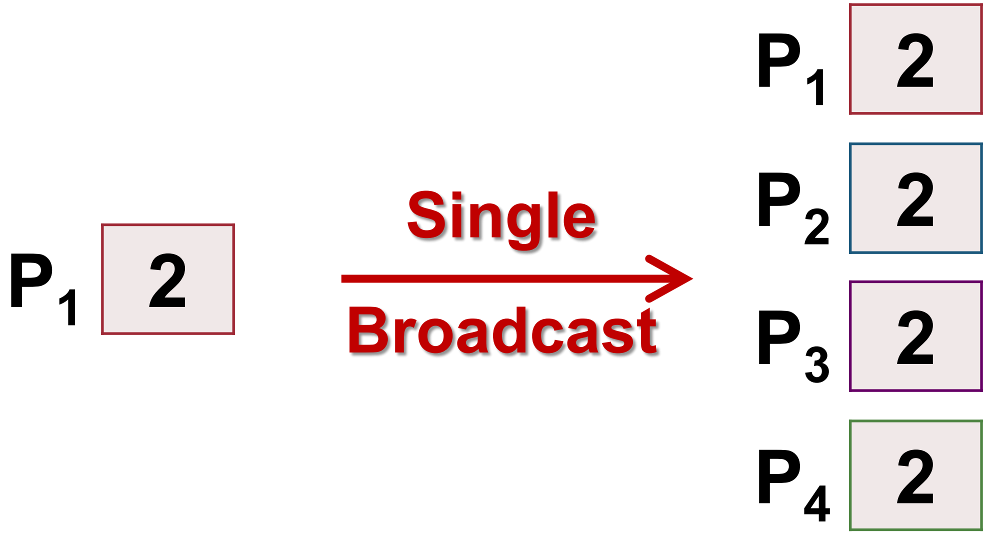






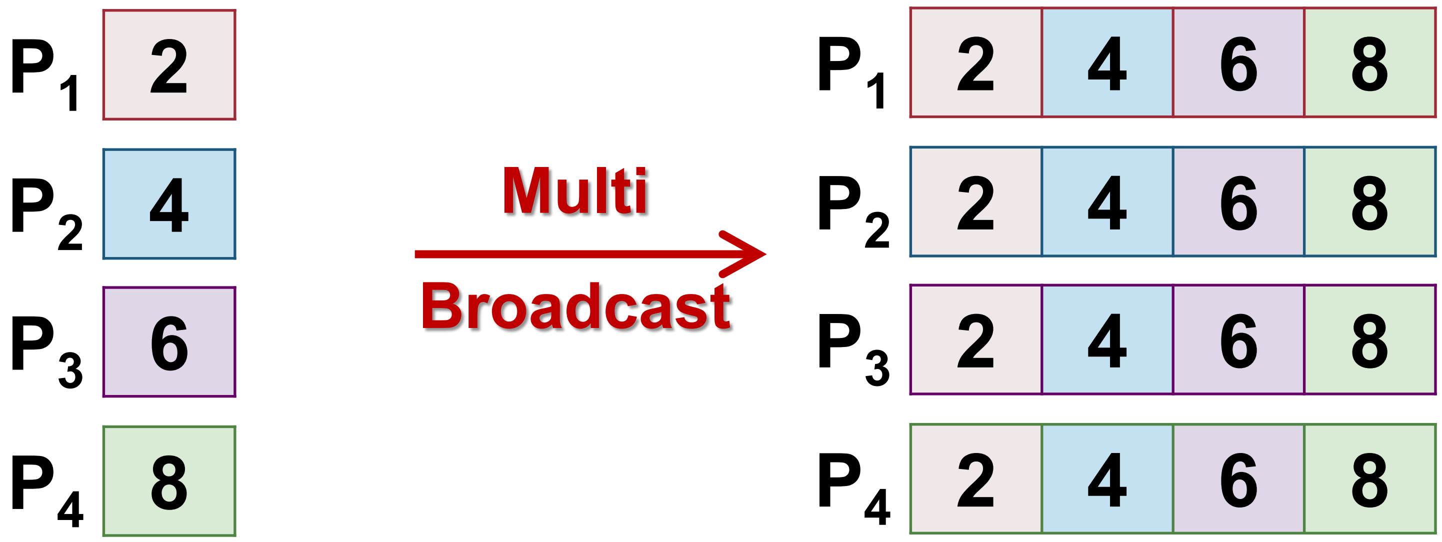
Single Broadcast



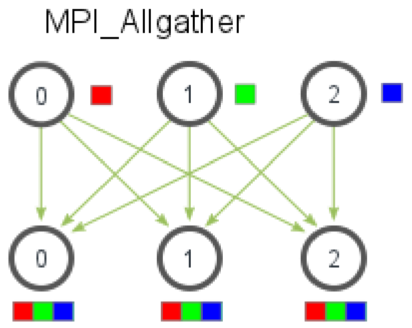


- Sender (the "root" processor) send the same data block to all other processors

Multi-Broadcast

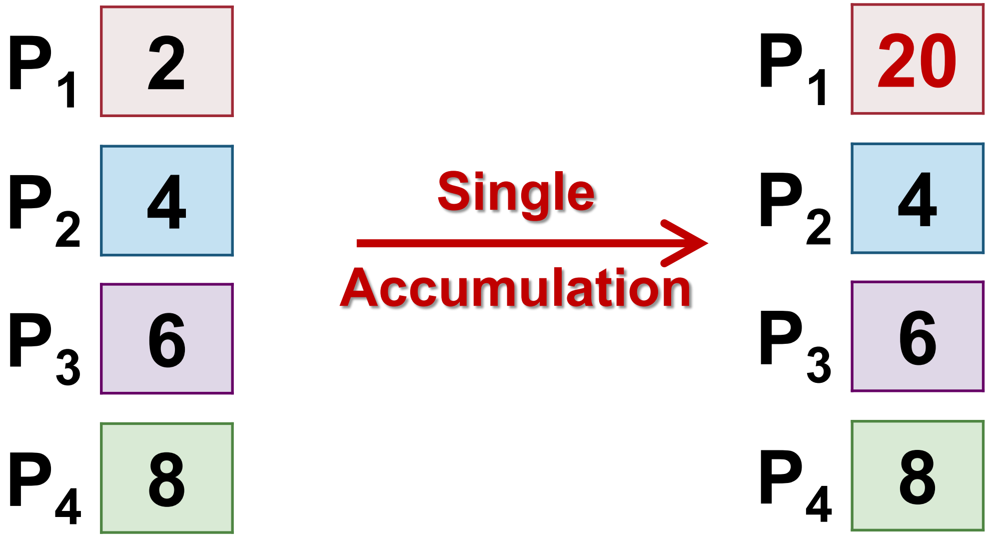


- Each processor sends the same data block to every other processor
  - No root processor



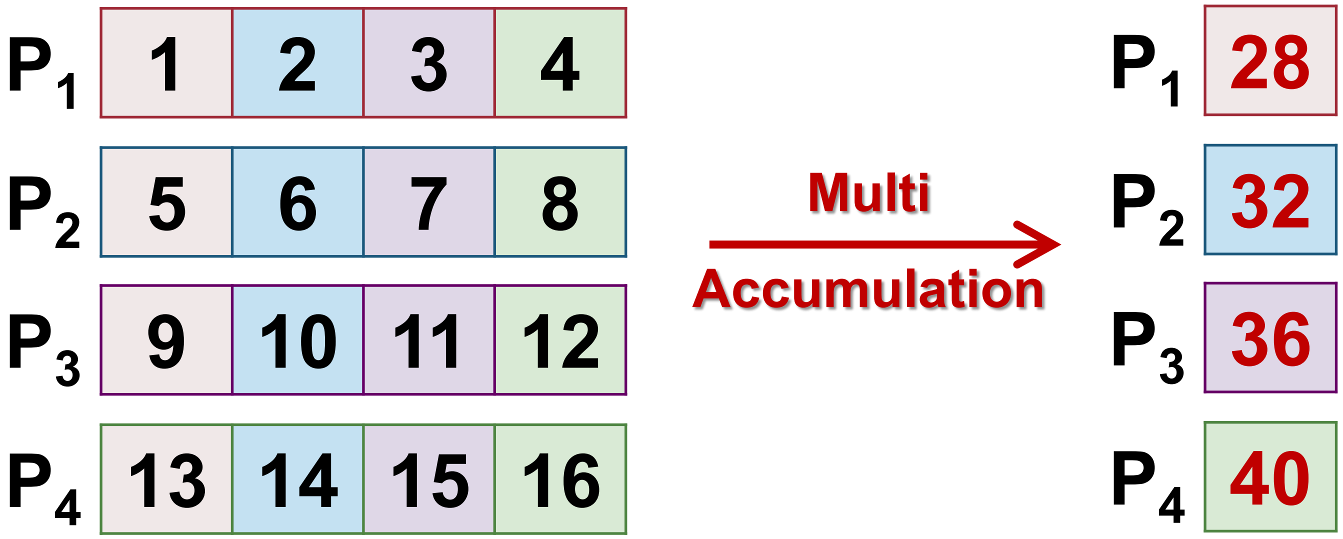
- Data blocks are collected in rank order

Single-accumulation (Gather with Reduction)



- Each processor provides a block of data with the same type and size
  - A reduction (binary, associative and commutative) operation is applied element by element to the data blocks
  - → results in root processor

Multi-accumulation



- Each processor provides for every other processor a potentially different data block
  - Data blocks for the same receiver are combined with a given reduction operation
  - No root processor

Total Exchange

P_1	1	2	3	4
P_2	5	6	7	8
P_3	9	10	11	12
P_4	13	14	15	16

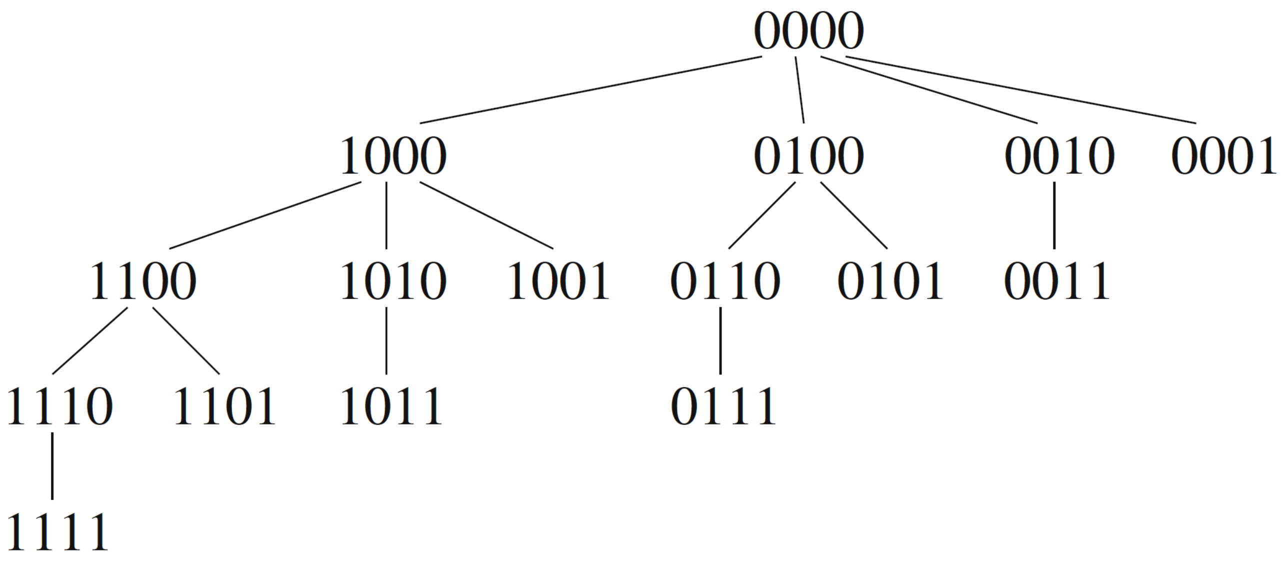
$\xrightarrow{\text{Total Exchange}}$

P_1	1	5	9	13
P_2	2	6	10	14
P_3	3	7	11	15
P_4	4	8	12	16

- Each processor provides for each other processor a potentially different data block
  - Effectively each processor executes a scatter operation
  - No root processor

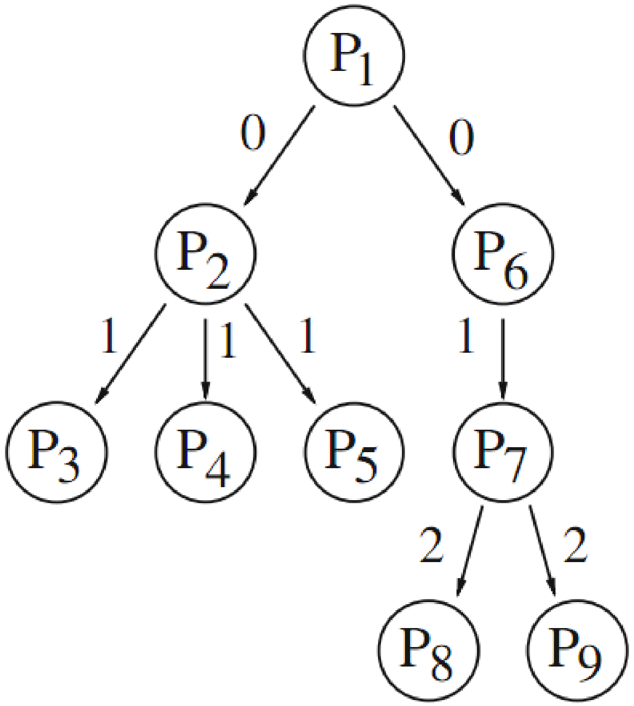
Duality of Communication Operations

- A communication operation can be presented by a graph
  - Spanning tree: a acyclic subgraph which contains all nodes and a subset of the edges, i.e. a tree
- Two communication operations are a **duality** if:
  - The same spanning tree can be used for both operations

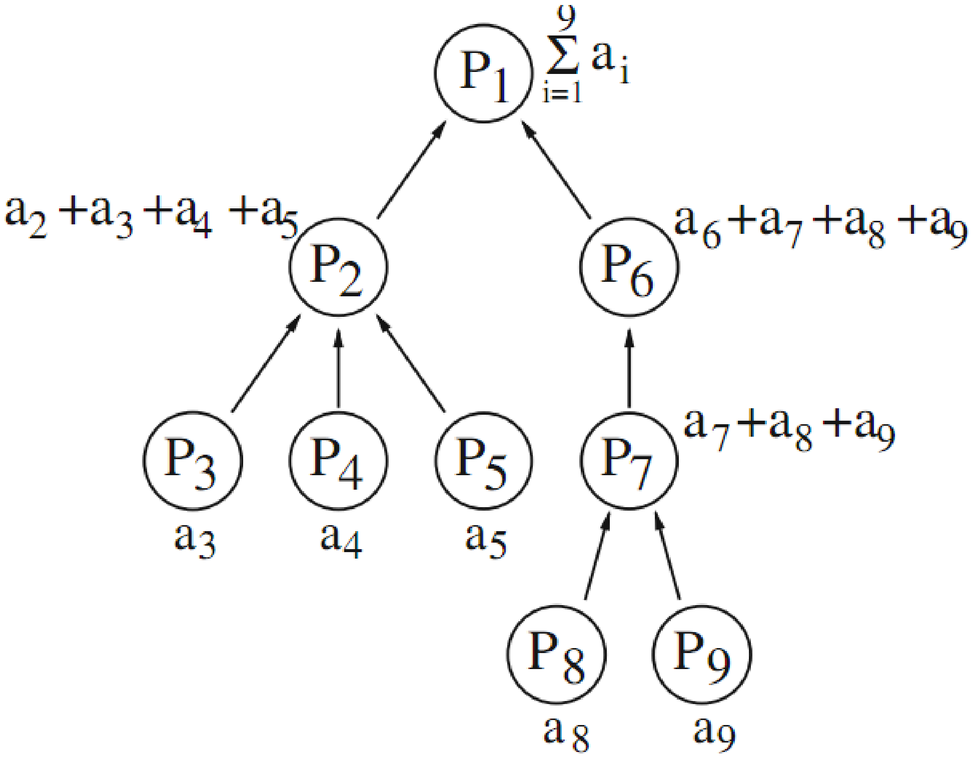


Spanning tree for a single-broadcast operation on a hypercube for $d = 4$.

Duality: Single-Broadcast & Single-Accumulation



Single-broadcast operation
(top-down traversal)



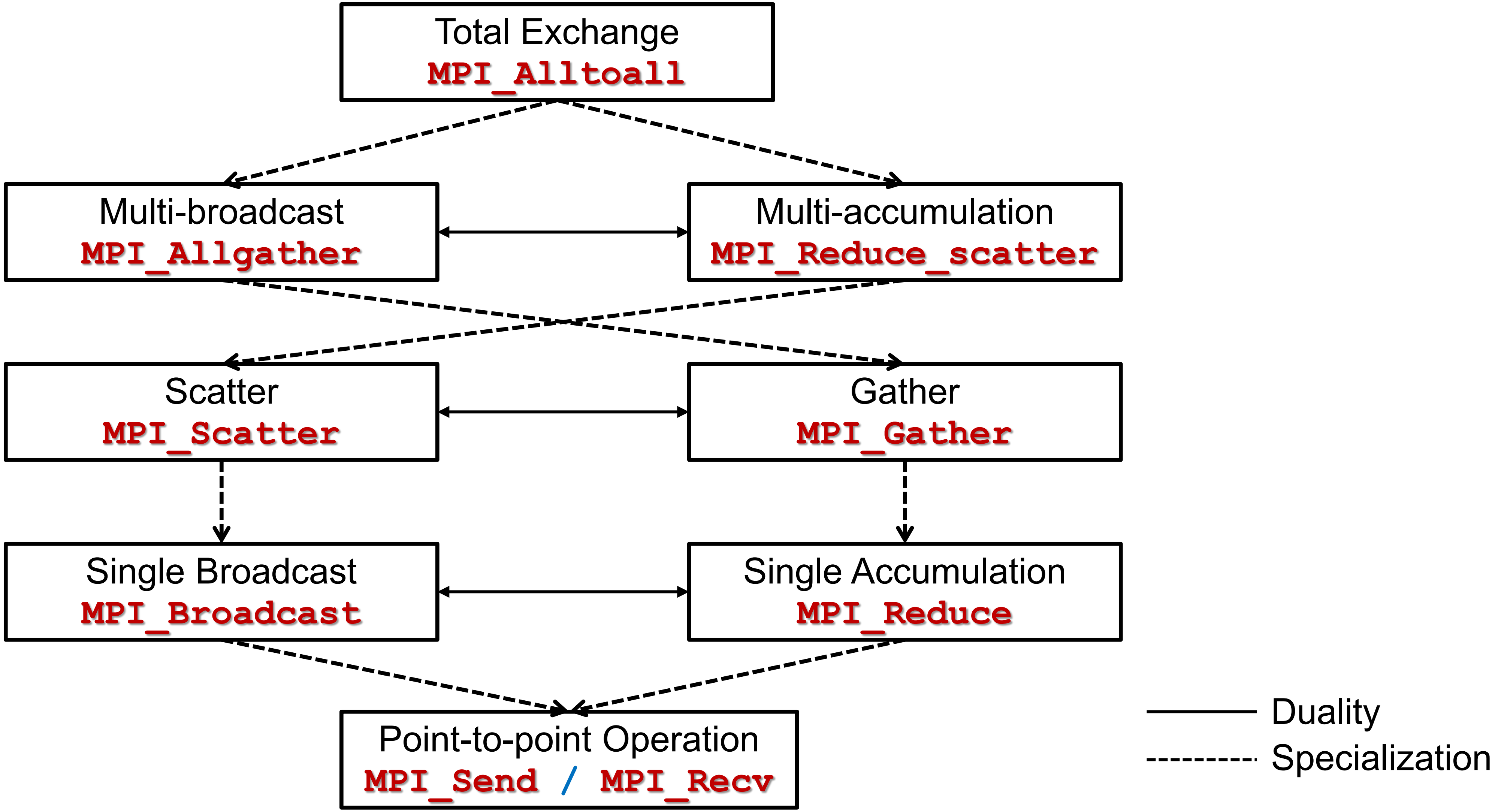
Single-accumulation operation
(bottom-up traversal)

Stepwise Specialization

- Communication operations can be ordered into a **hierarchy**:
  - From the most general to the most specific
  - Operations that are resulted from stepwise specialization are placed near to each other

- Example:
  - A stepwise specialization from total exchange to multi-broadcast
    - total exchange: each processor sends a different message to each other processor
    - multi-broadcast: each processor sends the **same** message to each other

MPI Collective Communication



Summary

- MPI (Message Passing Interface) for programming distributed-memory systems
- Point-to-point communication
  - Deadlocks may appear if send/recv are not paired securely
- Process Groups and Communicators
- Collective communication

- Reading
  - MPI Tutorial: https://computing.llnl.gov/tutorials/mpi/