Cloud Computing
CS 15-319

Programming Models- Part II

Lecture 5, Jan 30, 2012

Majd F. Sakr and Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

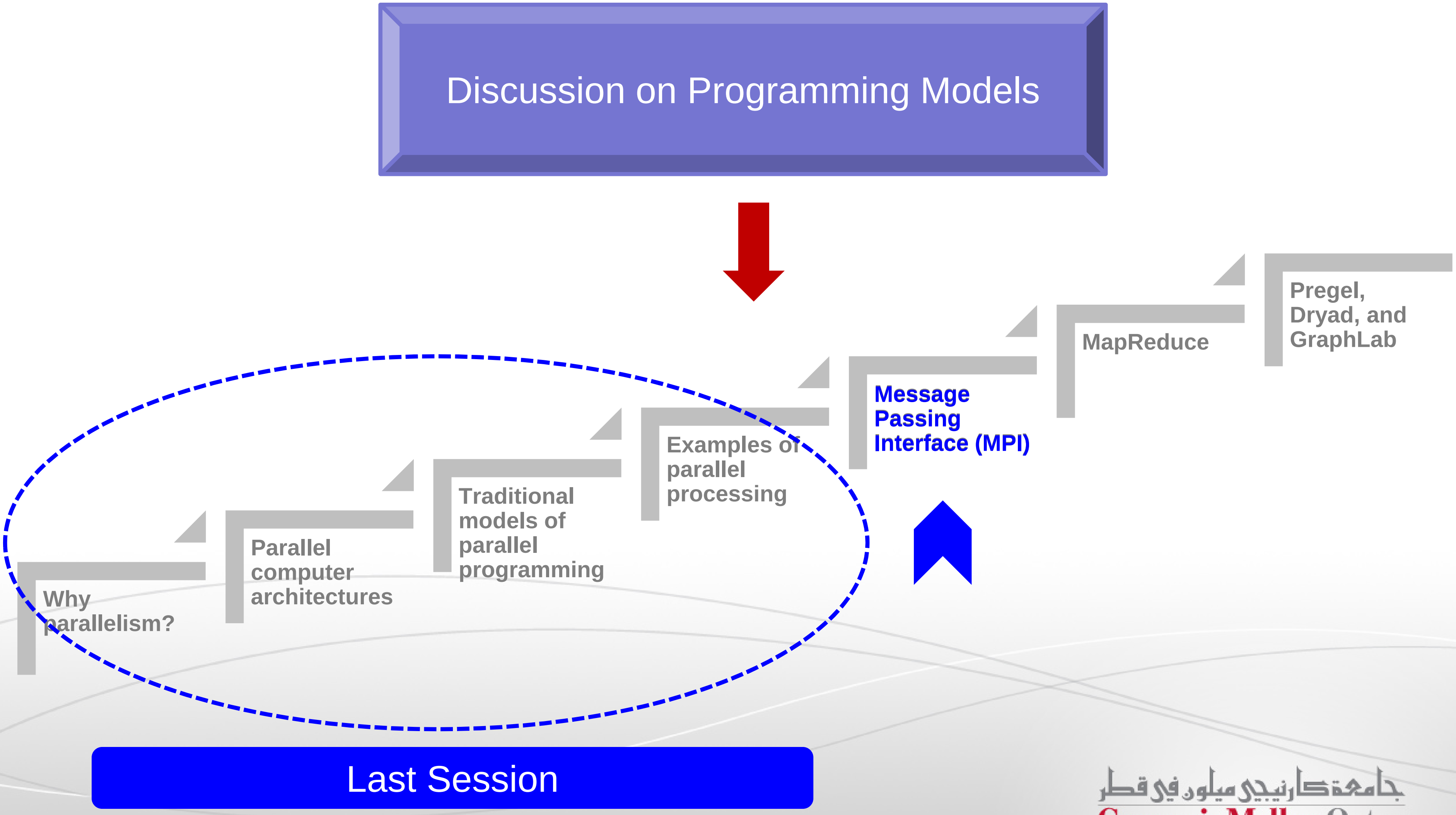
Today…

- Last session
  - Programming Models- *Part I*

- Today's session
  - Programming Models – *Part II*

- Announcement:
  - Project update is due on Wednesday February 1st

Objectives



Discussion on Programming Models

Why parallelism?

Parallel computer architectures

Traditional models of parallel programming

Examples of parallel processing

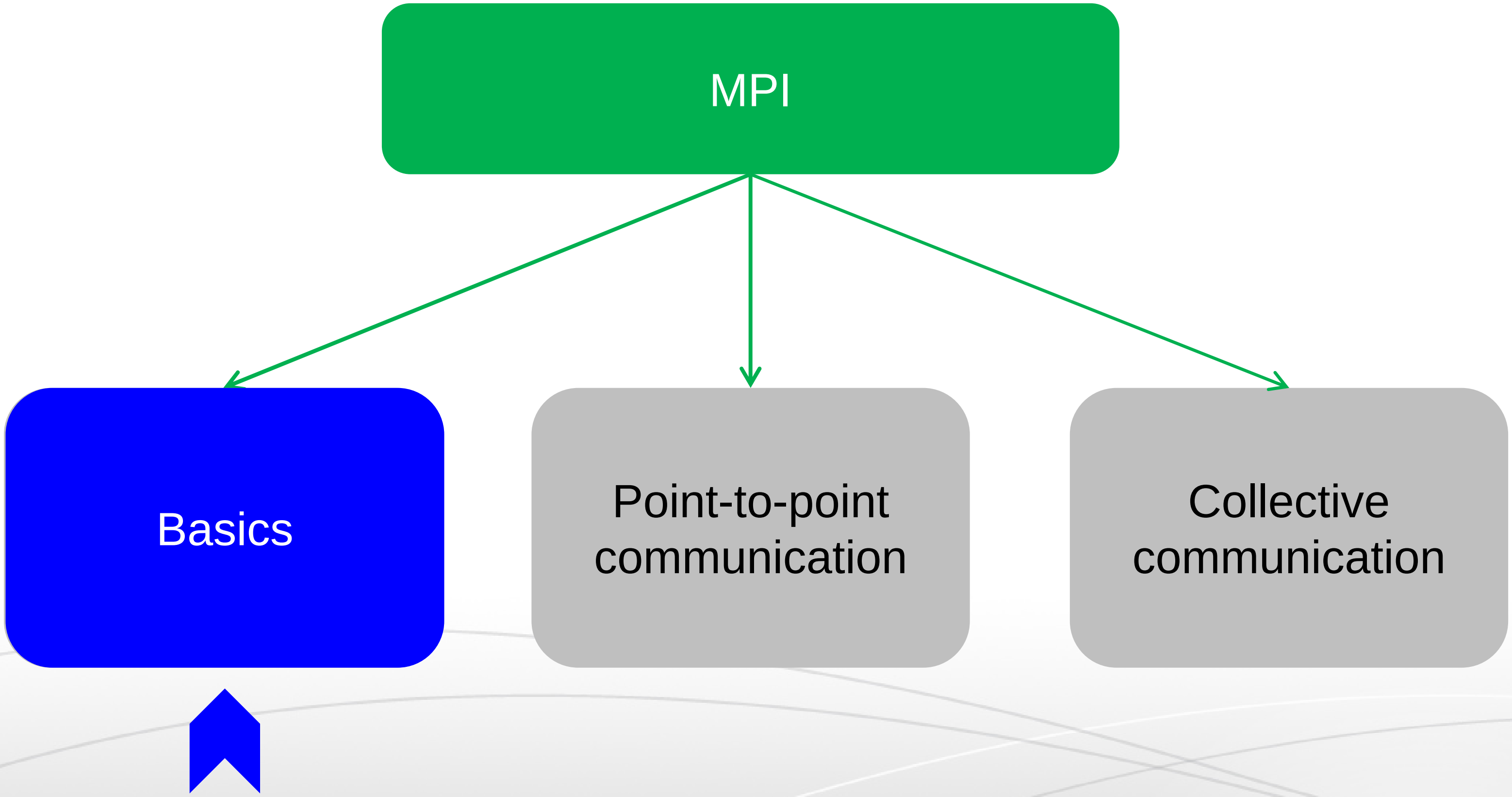
Message Passing Interface (MPI)

MapReduce

Pregel, Dryad, and GraphLab

Last Session

Message Passing Interface



What is MPI?

- The Message Passing Interface (MPI) is a message passing library standard for writing message passing programs

- The goal of MPI is to establish a *portable*, *efficient*, and *flexible* standard for message passing

- By itself, MPI is NOT a library - but rather the specification of what such a library should be

- MPI is not an IEEE or ISO standard, but has in fact, become the *industry standard* for writing message passing programs on HPC platforms

Reasons for using MPI

Reason	Description
Standardization	MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms
Portability	There is no need to modify your source code when you port your application to a different platform that supports the MPI standard
Performance Opportunities	Vendor implementations should be able to exploit native hardware features to optimize performance
Functionality	Over 115 routines are defined
Availability	A variety of implementations are available, both vendor and public domain

What Programming Model?

- MPI is an example of a message passing programming model
- MPI is now used on just about any common parallel architecture including MPP, SMP clusters, workstation clusters and heterogeneous networks
- With MPI, the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

Communicators and Groups

- MPI uses objects called *communicators and groups* to define which collection of processes may communicate with each other to solve a certain problem

- Most MPI routines require you to specify a communicator as an argument

- The communicator **MPI_COMM_WORLD** is often used in calling communication subroutines

- MPI_COMM_WORLD is the predefined communicator that includes *all* of your MPI processes

Ranks

- Within a communicator, every process has its own unique, integer identifier referred to as *rank*, assigned by the system when the process initializes

- A rank is sometimes called a *task ID*. Ranks are contiguous and begin at *zero*

- Ranks are used by the programmer to specify the source and destination of messages

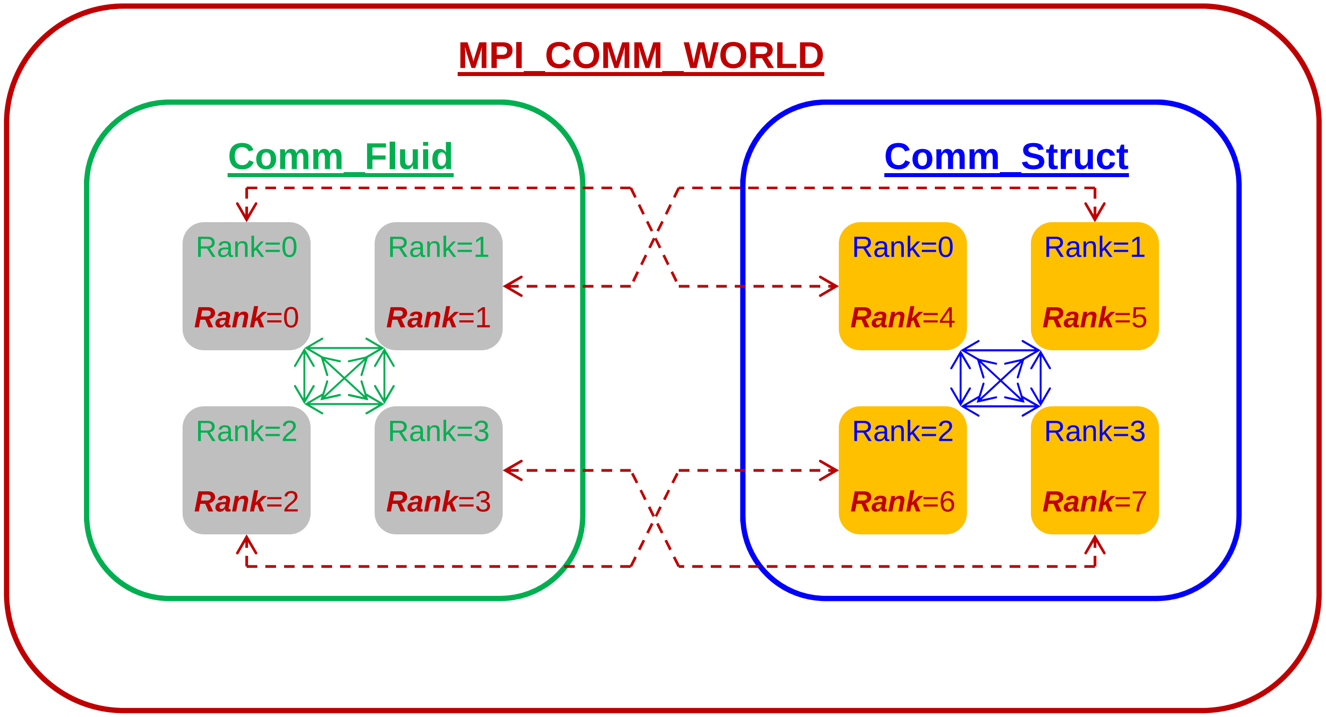
- Ranks are often also used conditionally by the application to control program execution (e.g., *if rank=0 do this / if rank=1 do that*)

Multiple Communicators

- It is possible that a problem consists of several sub-problems where each can be solved independently
- This type of application is typically found in the category of *MPMD* coupled analysis
- We can create a new communicator for each sub-problem as a subset of an existing communicator
- MPI allows you to achieve that by using **MPI_COMM_SPLIT**

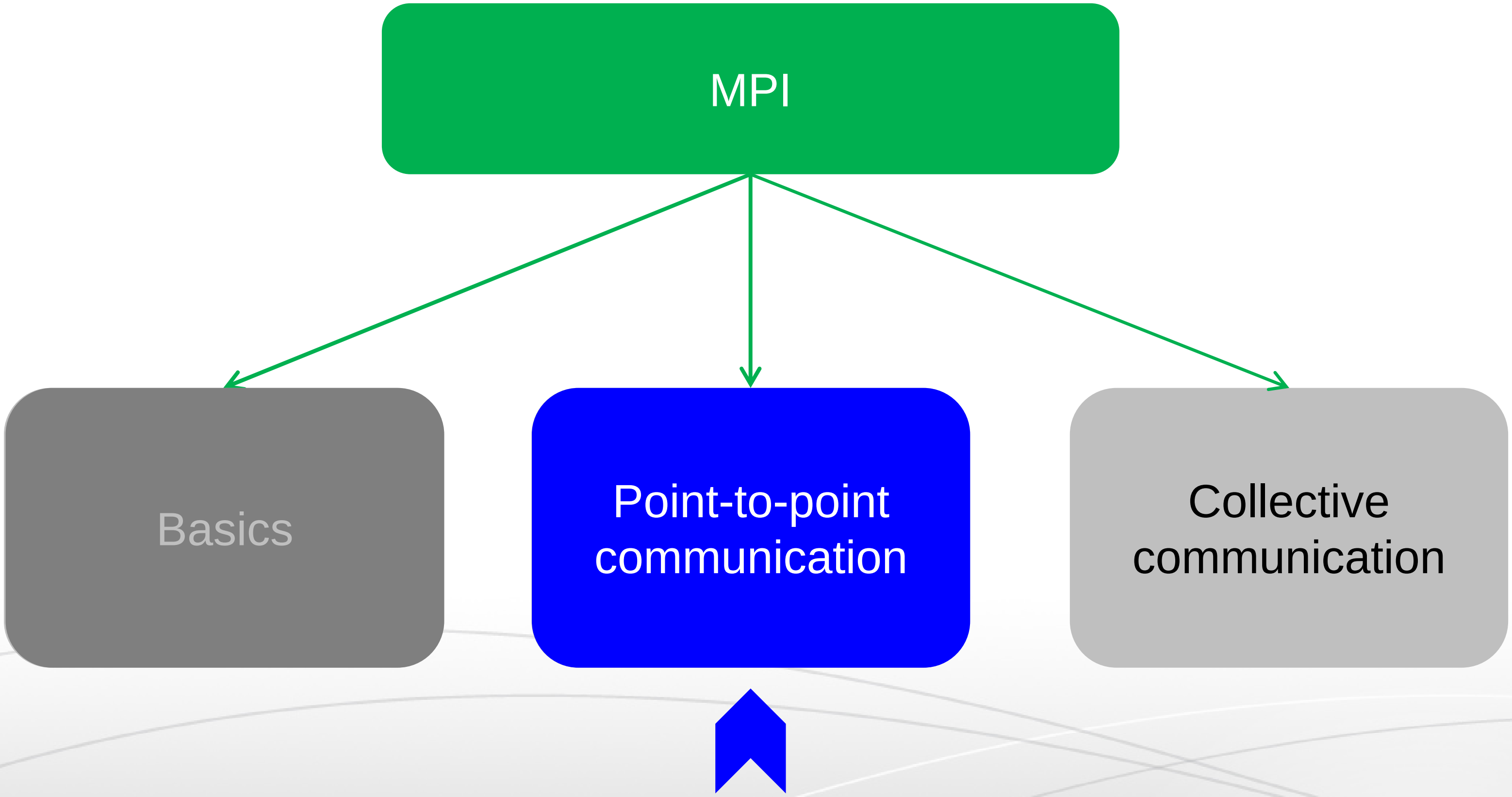
Example of Multiple Communicators

- Consider a problem with a fluid dynamics part and a structural analysis part, where each part can be computed in parallel



- ✓ Ranks within MPI_COMM_WORLD are printed in red
- ✓ Ranks within Comm_Fluid are printed with green
- ✓ Ranks within Comm_Struct are printed with blue

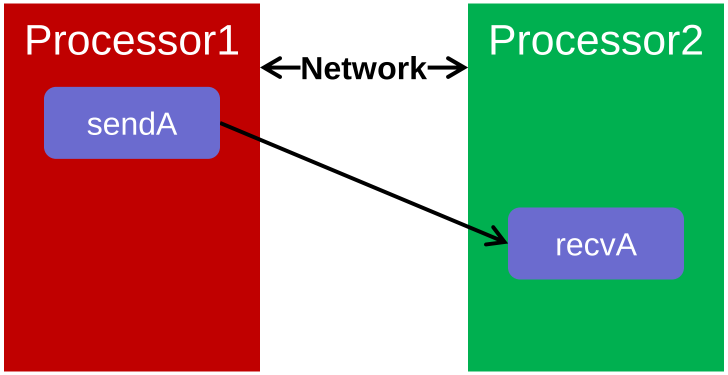
Message Passing Interface



Point-to-Point Communication

- MPI point-to-point operations typically involve message passing between two, ***and only two***, different MPI tasks

  - One task performs a *send* operation and the other performs a ***matching*** *receive* operation



- Ideally, every send operation would be perfectly synchronized with its matching receive

- This is rarely the case. Somehow or other, the MPI implementation must be able to deal with storing data when the two tasks are out of sync

Two Cases

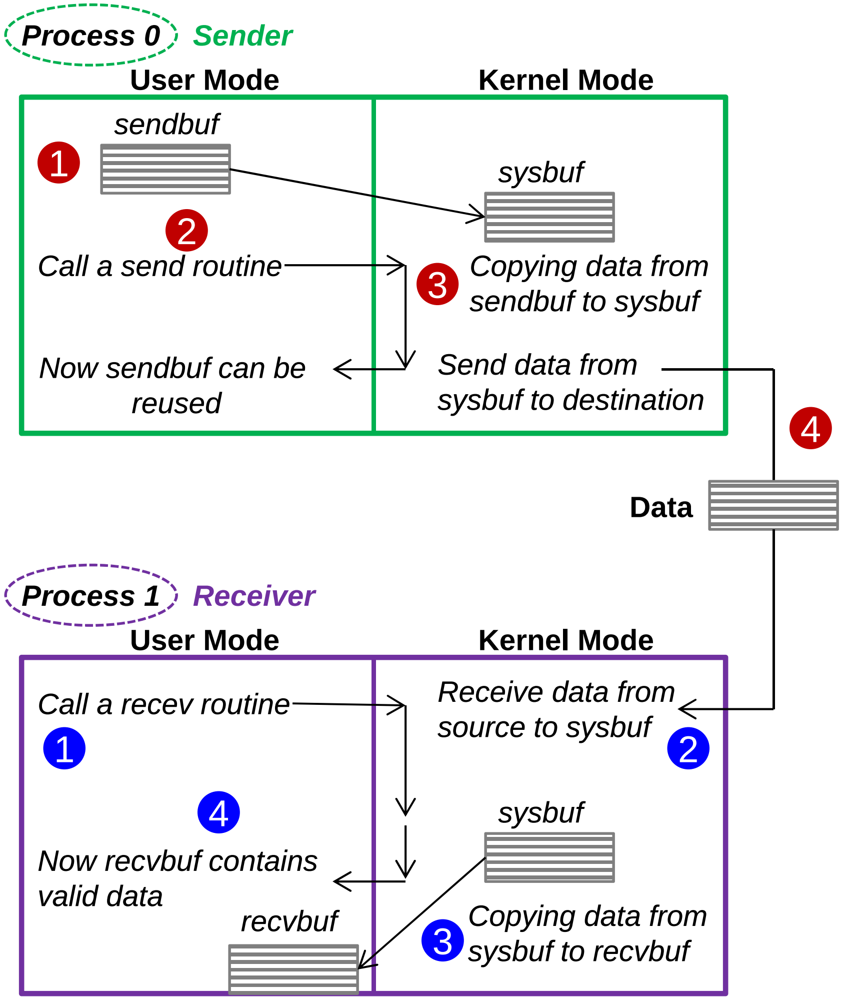
- Consider the following two cases:

1. A send operation occurs 5 seconds before the receive is ready - *where is the message stored while the receive is pending?*

2. Multiple sends arrive at the same receiving task which can only accept one send at a time - *what happens to the messages that are "backing up"?*

Steps Involved in Point-to-Point Communication

1. The data is copied to the user buffer by the user
2. The user calls one of the MPI send routines
3. The system copies the data from the user buffer to the system buffer
4. The system sends the data from the system buffer to the destination process



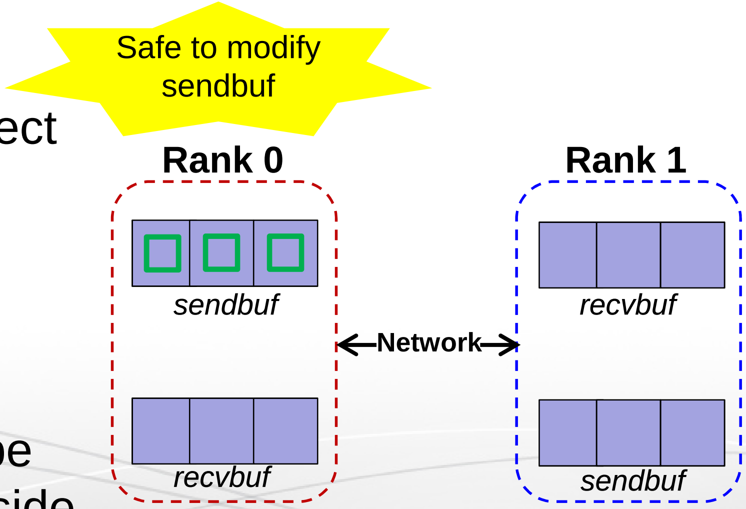
Process 0
Sender
User Mode
Kernel Mode
sendbuf
1
sysbuf
2
Call a send routine
3
Copying data from sendbuf to sysbuf
Now sendbuf can be reused
Send data from sysbuf to destination
4
Data
Process 1
Receiver
User Mode
Kernel Mode
Call a recev routine
1
Receive data from source to sysbuf
2
4
sysbuf
Now recvbuf contains valid data
recvbuf
3
Copying data from sysbuf to recvbuf

1. The user calls one of the MPI receive routines
2. The system receives the data from the source process and copies it to the system buffer
3. The system copies data from the system buffer to the user buffer
4. The user uses data in the user buffer

Blocking Send and Receive

- When we use point-to-point communication routines, we usually distinguish between *blocking and non-blocking communication*

- A blocking send routine will only *return* after it is *safe* to modify the application buffer for reuse
  - Safe means that modifications will not affect the data intended for the receive task
  - This does not imply that the data was actually received by the receiver- it may be sitting in the system buffer at the sender side



Blocking Send and Receive

- A blocking send can be:

  1. *Synchronous:* Means there is a handshaking occurring with the receive task to confirm a safe send

  2. *Asynchronous:* Means the system buffer at the sender side is used to hold the data for eventual delivery to the receiver

- A blocking receive only *returns* after the data has arrived (i.e., stored at the application recvbuf) and is ready for use by the program

Non-Blocking Send and Receive (1)

- Non-blocking send and non-blocking receive routines behave similarly
  - They return almost immediately
  - They do not wait for any communication events to complete such as:
    - Message copying from user buffer to system buffer
    - Or the actual arrival of a message

Non-Blocking Send and Receive (2)

- However, it is unsafe to modify the application buffer until you make sure that the requested non-blocking operation was actually performed by the library

- If you use the application buffer before the copy completes:
  - Incorrect data may be copied to the system buffer (in case of non-blocking send)
  - Or your receive buffer does not contain what you want (in case of non-blocking receive)

- You can make sure of the completion of the copy by using MPI_WAIT() after the send or receive operations

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

Why Non-Blocking Communication?

- Why do we use non-blocking communication despite its complexity?
  - Non-blocking communication is generally faster than its corresponding blocking communication
  - We can overlap computations while the system is copying data back and forth between application and system buffers

MPI Point-To-Point Communication Routines

Routine	Signature
Blocking send	int *MPI_Send*(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
Non-blocking send	int *MPI_Isend*(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
Blocking receive	int *MPI_Recv*(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
Non-blocking receive	int *MPI_Irecv*(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

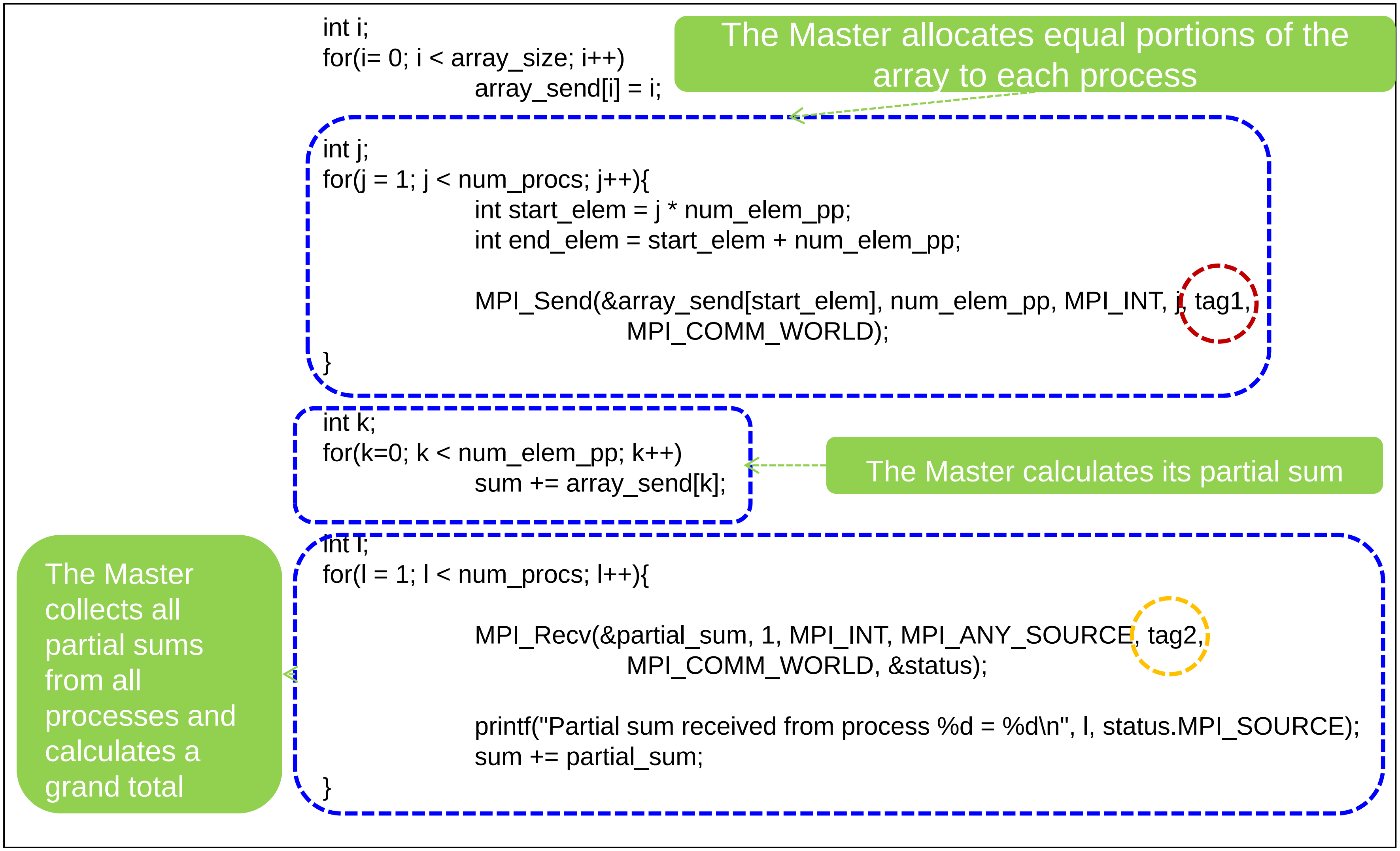
MPI Example: Adding Array Elements

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

#define array_size 20
#define num_elem_pp 10
#define tag1 1
#define tag2 2

int array_send[array_size];
int array_recv[num_elem_pp];

int main(int argc, char **argv){

          int myPID;
          int num_procs;
          int sum = 0;
          int partial_sum = 0;
          double startTime = 0.0;
          double endTime;
          MPI_Status status;

          MPI_Init(&argc, &argv);
          MPI_Comm_rank(MPI_COMM_WORLD, &myPID);
          MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

          if(myPID == 0 )
          {
                    startTime = MPI_Wtime();
```

Initialize MPI environment

The Master

MPI Example: Adding Array Elements



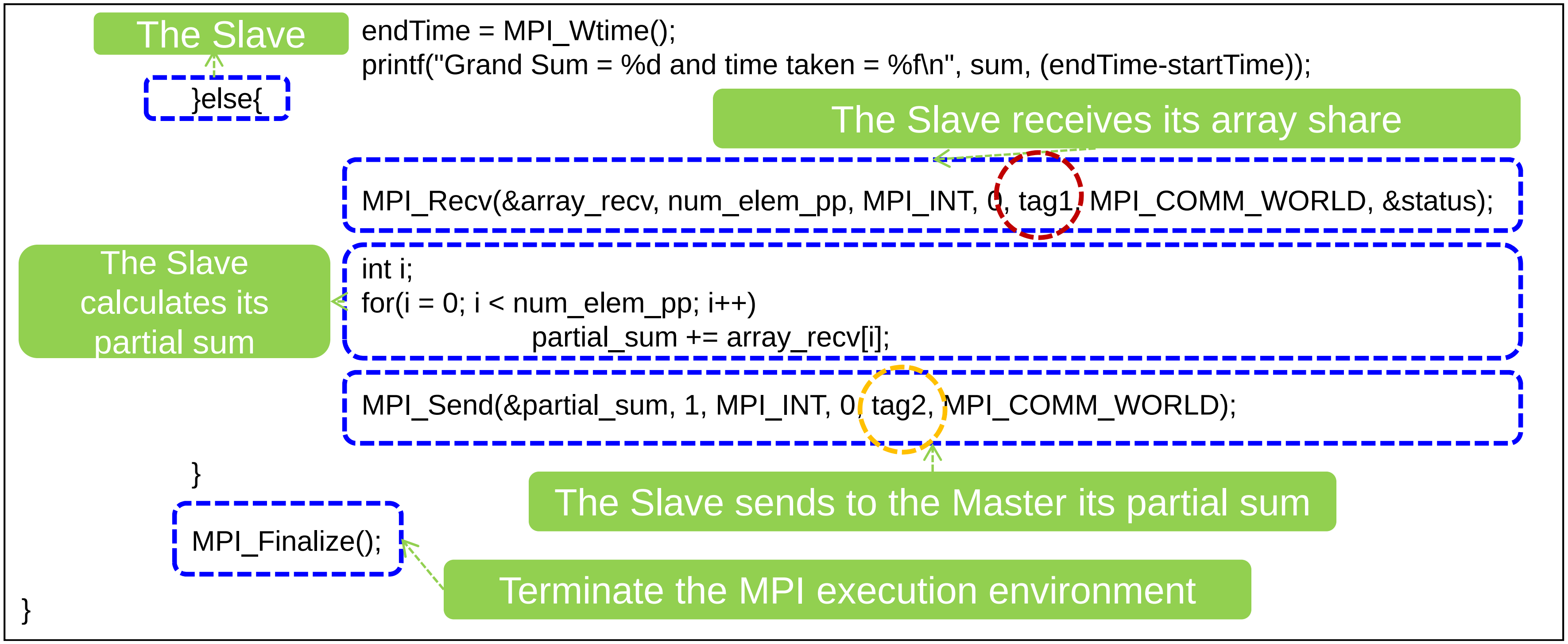
```
int i;
for(i= 0; i < array_size; i++)
          array_send[i] = i;

int j;
for(j = 1; j < num_procs; j++){
          int start_elem = j * num_elem_pp;
          int end_elem = start_elem + num_elem_pp;

          MPI_Send(&array_send[start_elem], num_elem_pp, MPI_INT, j, tag1,
                    MPI_COMM_WORLD);
}

int k;
for(k=0; k < num_elem_pp; k++)
          sum += array_send[k];

int l;
for(l = 1; l < num_procs; l++){

          MPI_Recv(&partial_sum, 1, MPI_INT, MPI_ANY_SOURCE, tag2,
                    MPI_COMM_WORLD, &status);

          printf("Partial sum received from process %d = %d\n", l, status.MPI_SOURCE);
          sum += partial_sum;
}
```

The Master allocates equal portions of the array to each process

The Master calculates its partial sum

The Master collects all partial sums from all processes and calculates a grand total

MPI Example: Adding Array Elements



The Slave

```
        endTime = MPI_Wtime();
        printf("Grand Sum = %d and time taken = %f\n", sum, (endTime-startTime));
    }else{
```

The Slave receives its array share

```
        MPI_Recv(&array_recv, num_elem_pp, MPI_INT, 0, tag1, MPI_COMM_WORLD, &status);
```

The Slave calculates its partial sum

```
        int i;
        for(i = 0; i < num_elem_pp; i++)
                partial_sum += array_recv[i];
```

```
        MPI_Send(&partial_sum, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD);
```

The Slave sends to the Master its partial sum

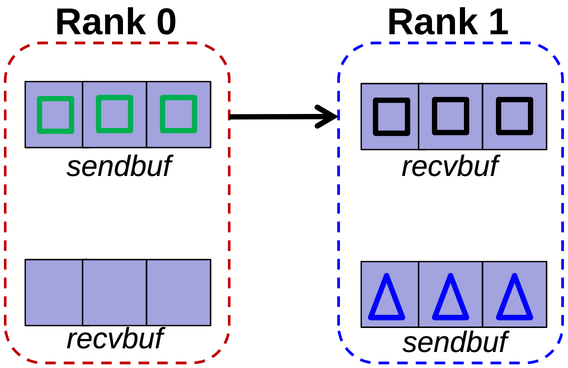
```
    }
    MPI_Finalize();
```

Terminate the MPI execution environment

```
}
```

Unidirectional Communication

- When you send a message from process 0 to process 1, there are four combinations of MPI subroutines to choose from



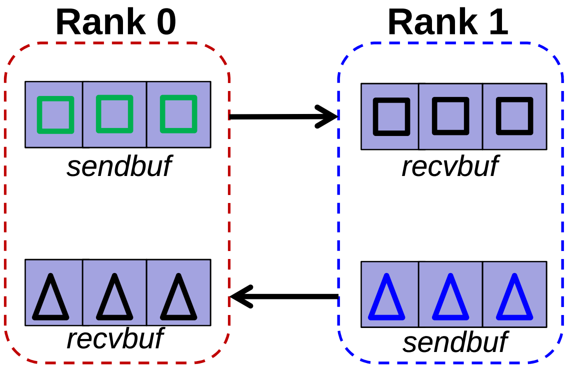
1. Blocking send and blocking receive
2. Non-blocking send and blocking receive
3. Blocking send and non-blocking receive
4. Non-blocking send and non-blocking receive

Bidirectional Communication

- When two processes exchange data with each other, there are essentially 3 cases to consider:

  - Case 1: Both processes call the send routine first, and then receive

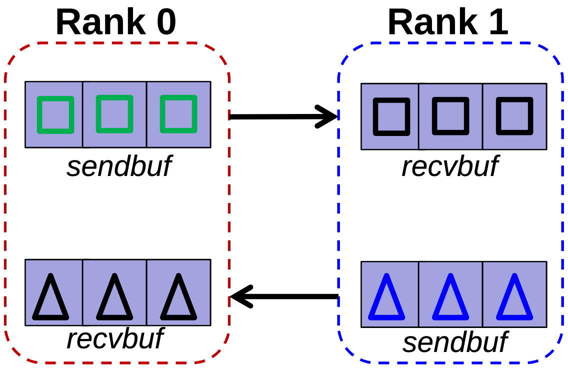
  - Case 2: Both processes call the receive routine first, and then send



  - Case 3: One process calls send and receive routines in this order, and the other calls them in the opposite order

Bidirectional Communication-Deadlocks

- With bidirectional communication, we have to be careful about *deadlocks*



- When a deadlock occurs, processes involved in the deadlock will not proceed any further

- Deadlocks can take place:

    1. Either due to the incorrect order of send and receive
    2. Or due to the limited size of the system buffer

Case 1. Send First and Then Receive

- Consider the following two snippets of pseudo-code:

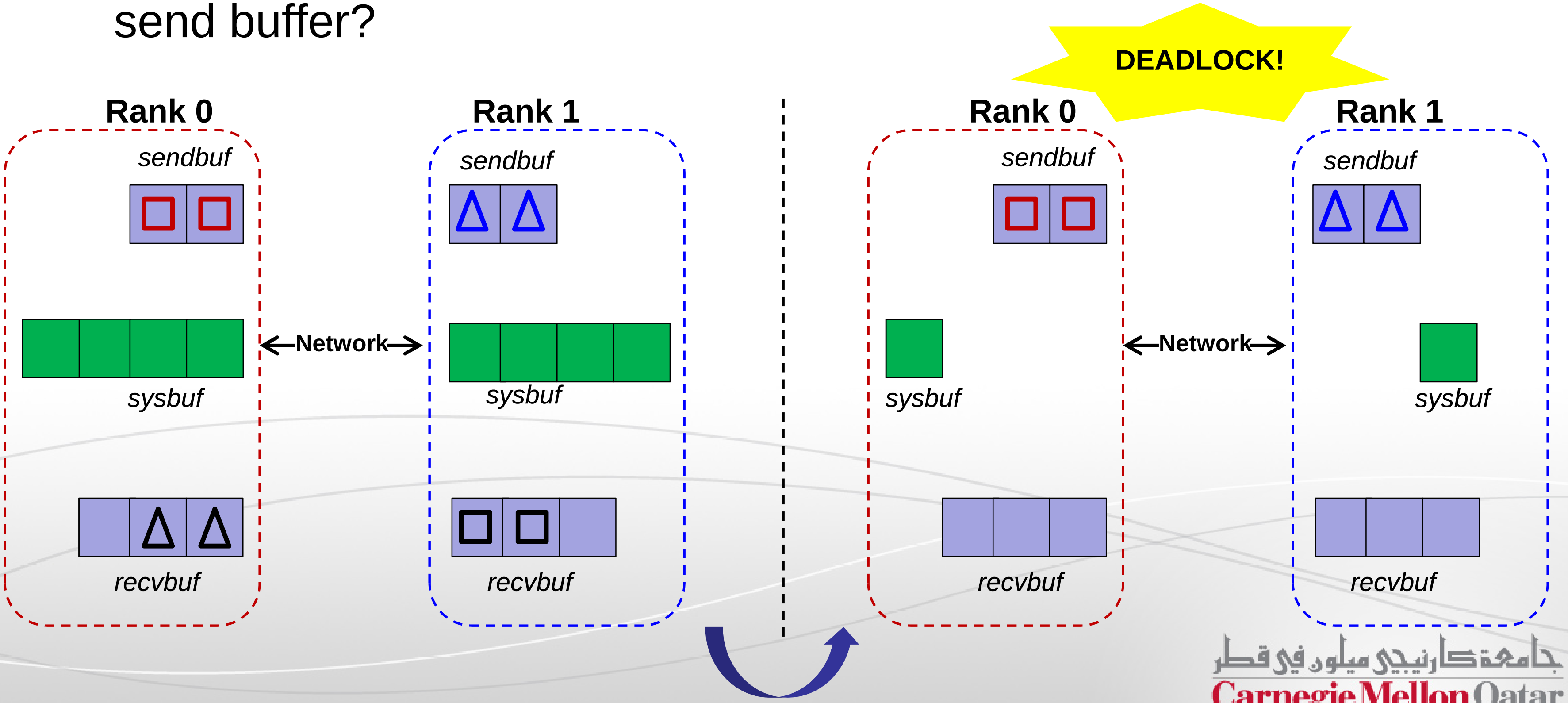
```
IF (myrank==0) THEN
  CALL MPI_SEND(sendbuf, …)
  CALL MPI_RECV(recvbuf, …)
ELSEIF (myrank==1) THEN
  CALL MPI_SEND(sendbuf, …)
  CALL MPI_RECV(recvbuf, …)
ENDIF
```

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, …, ireq, …)
  CALL MPI_WAIT(ireq, …)
  CALL MPI_RECV(recvbuf, …)
ELSEIF (myrank==1) THEN
  CALL MPI_ISEND(sendbuf, …, ireq, …)
  CALL MPI_WAIT(ireq, …)
  CALL MPI_RECV(recvbuf, …)
ENDIF
```

- MPI_ISEND immediately followed by MPI_WAIT is logically equivalent to MPI_SEND

Case 1. Send First and Then Receive

- What happens if the system buffer is larger than the send buffer?
- What happens if the system buffer is smaller than the send buffer?



Case 1. Send First and Then Receive

- Consider the following pseudo-code:

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, …, ireq, …)
  CALL MPI_RECV(recvbuf, …)
  CALL MPI_WAIT(ireq, …)
ELSEIF (myrank==1) THEN
  CALL MPI_ISEND(sendbuf, …, ireq, …)
  CALL MPI_RECV(recvbuf, …)
  CALL MPI_WAIT(ireq, …)
ENDIF
```

- The code is free from deadlock because:
  - The program immediately returns from MPI_ISEND and starts receiving data from the other process
  - In the meantime, data transmission is completed and the calls of MPI_WAIT for the completion of send at both processes do not lead to a deadlock

Case 2. Receive First and Then Send

- Would the following pseudo-code lead to a deadlock?
  - A deadlock will occur regardless of how much system buffer we have

```
IF (myrank==0) THEN
  CALL MPI_RECV(recvbuf, …)
  CALL MPI_SEND(sendbuf, …)
ELSEIF (myrank==1) THEN
  CALL MPI_RECV(recvbuf, …)
  CALL MPI_ISEND(sendbuf, …)
ENDIF
```

- What if we use MPI_ISEND instead of MPI_SEND?
  - Deadlock still occurs

Case 2. Receive First and Then Send

- What about the following pseudo-code?
  - It can be safely executed

```
IF (myrank==0) THEN
  CALL MPI_IRECV(recvbuf, …, ireq, …)
  CALL MPI_SEND(sendbuf, …)
  CALL MPI_WAIT(ireq, …)
ELSEIF (myrank==1) THEN
  CALL MPI_IRECV(recvbuf, …, ireq, …)
  CALL MPI_SEND(sendbuf, …)
  CALL MPI_WAIT(ireq, …)
ENDIF
```

Case 3. One Process Sends and Receives; the other Receives and Sends

- What about the following code?

```
IF (myrank==0) THEN
  CALL MPI_SEND(sendbuf, …)
  CALL MPI_RECV(recvbuf, …)
ELSEIF (myrank==1) THEN
  CALL MPI_RECV(recvbuf, …)
  CALL MPI_SEND(sendbuf, …)
ENDIF
```

- It is *always safe* to order the calls of MPI_(I)SEND and MPI_(I)RECV at the two processes in an opposite order

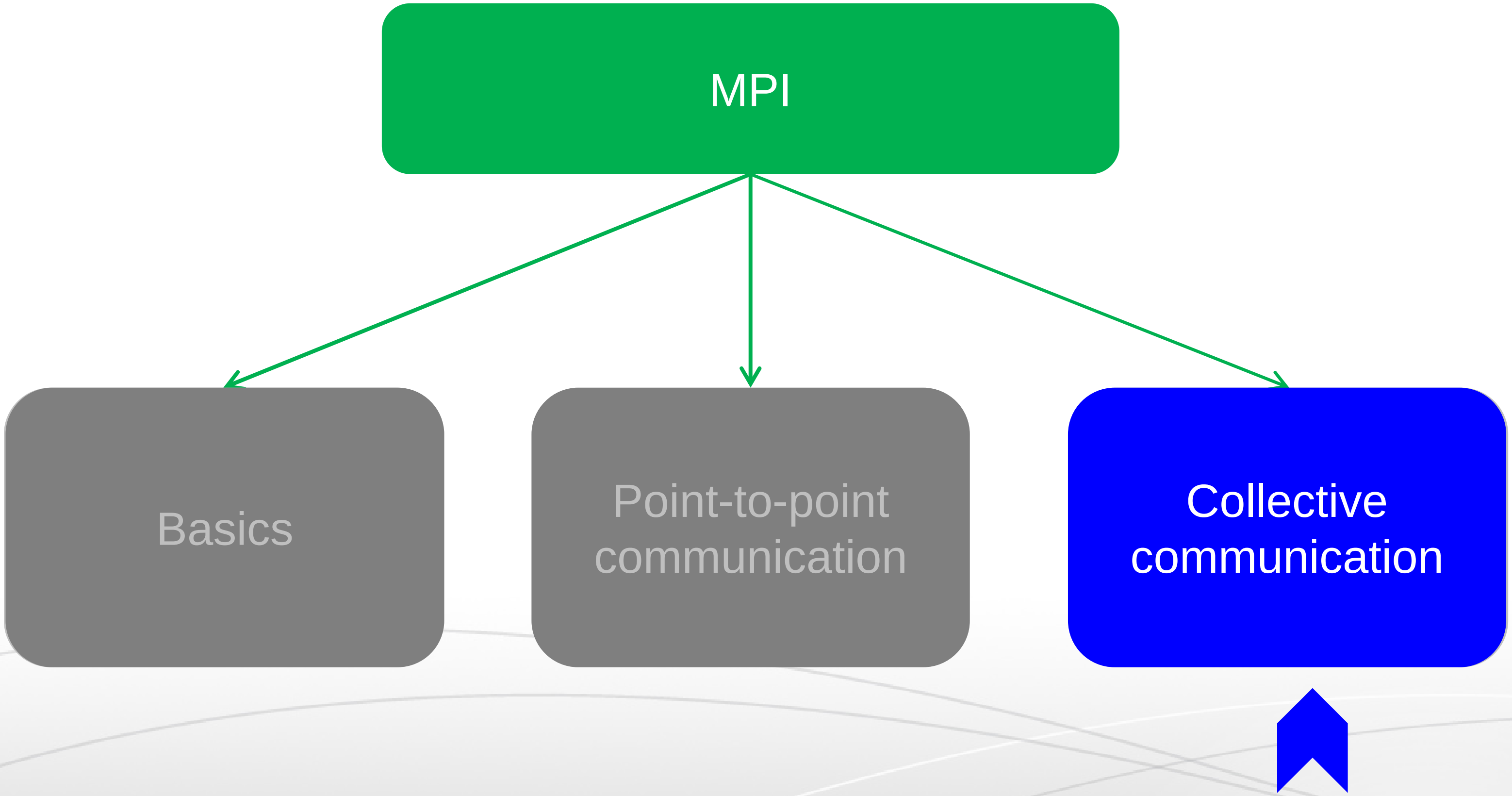
- In this case, we can use either blocking or non-blocking subroutines

A Recommendation

- Considering the previous options, performance, and the avoidance of deadlocks, it is recommended to use the following code:

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, …, ireq1, …)
  CALL MPI_IRECV(recvbuf, …, ireq2, …)
ELSEIF (myrank==1) THEN
  CALL MPI_ISEND(sendbuf, …, ireq1, …)
  CALL MPI_IRECV(recvbuf, …, ireq2, …)
ENDIF
  CALL MPI_WAIT(ireq1, …)
  CALL MPI_WAIT(ireq2, …)
```

Message Passing Interface



Collective Communication

- Collective communication allows you to exchange data among a group of processes

- It must involve ***all*** processes in the scope of a communicator

- The communicator argument in a collective communication routine should specify which processes are involved in the communication

- Hence, it is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operation

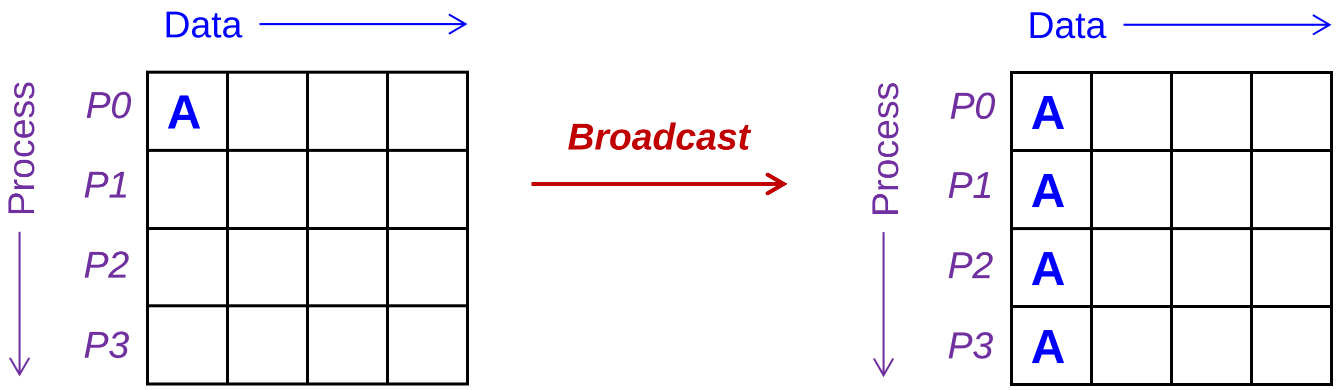
Patterns of Collective Communication

- There are several patterns of collective communication:

1. *Broadcast*
2. *Scatter*
3. *Gather*
4. *Allgather*
5. *Alltoall*
6. *Reduce*
7. *Allreduce*
8. *Scan*
9. *Reducescatter*

1. Broadcast

- *Broadcast* sends a message from the process with rank *root* to all other processes in the group



Data →

Process ↓

P0	A			
P1				
P2				
P3				

Broadcast →

Data →

Process ↓

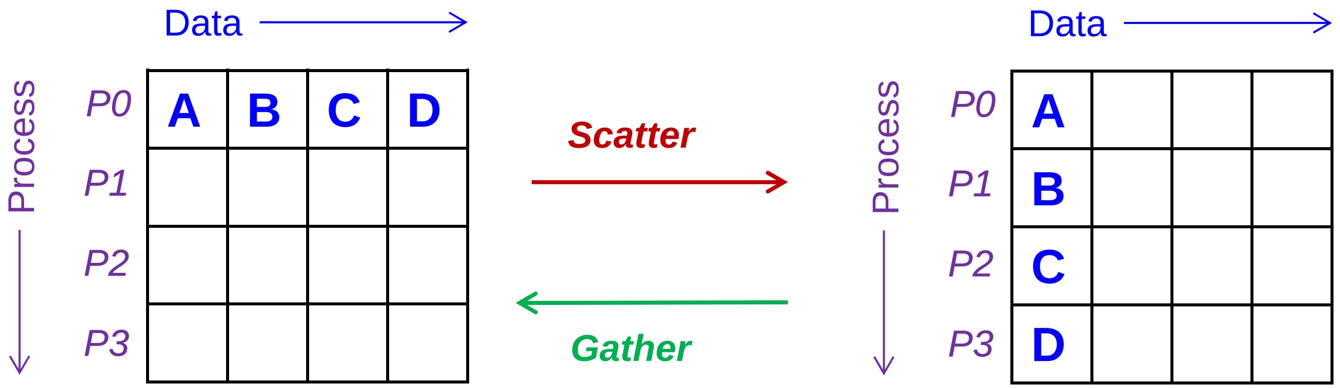
P0	A			
P1	A			
P2	A			
P3	A			

int **MPI_Bcast** (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

2-3. Scatter and Gather

- *Scatter* distributes distinct messages from a single source task to each task in the group

- *Gather* gathers distinct messages from each task in the group to a single destination task



P0	A	B	C	D
P1				
P2				
P3				

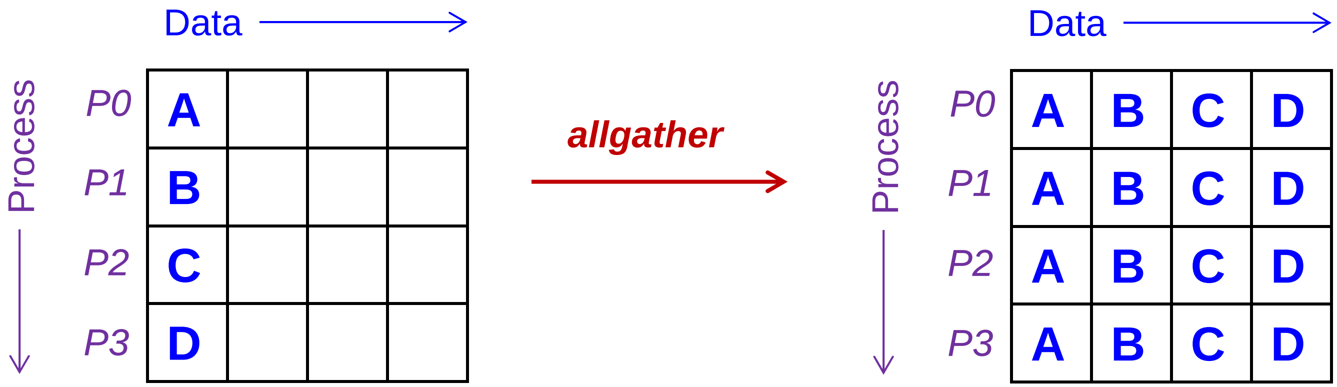
P0	A			
P1	B			
P2	C			
P3	D			

```
int MPI_Scatter ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt,
                  MPI_Datatype recvtype, int root, MPI_Comm comm )
```

```
int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm )
```

4. All Gather

- *Allgather* gathers data from all tasks and distribute them to all tasks. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group



Data →

Process ↓

P0	A			
P1	B			
P2	C			
P3	D			

allgather →

Data →

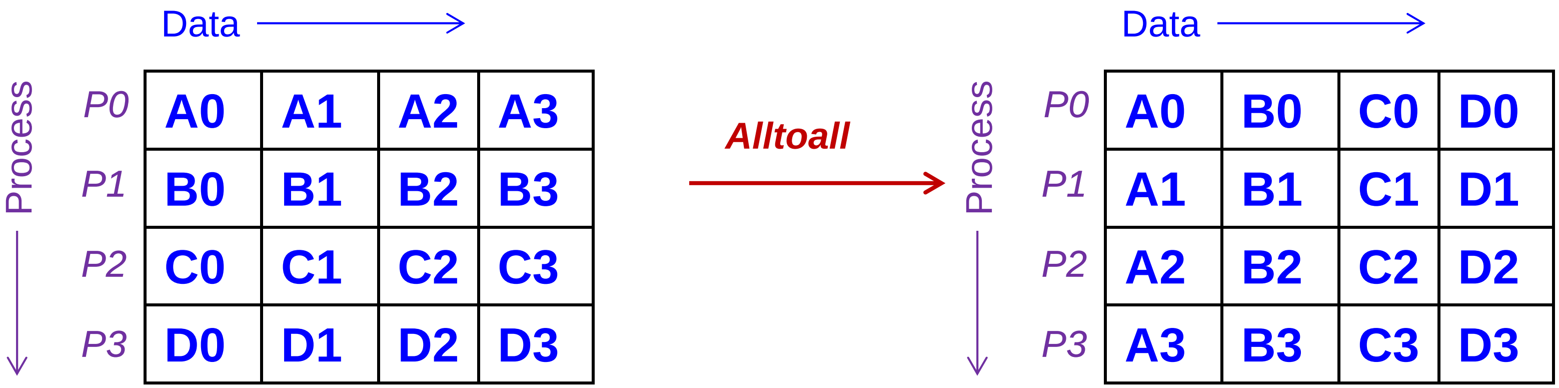
Process ↓

P0	A	B	C	D
P1	A	B	C	D
P2	A	B	C	D
P3	A	B	C	D

int MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

5. All To All

- With *Alltoall*, each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index



P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

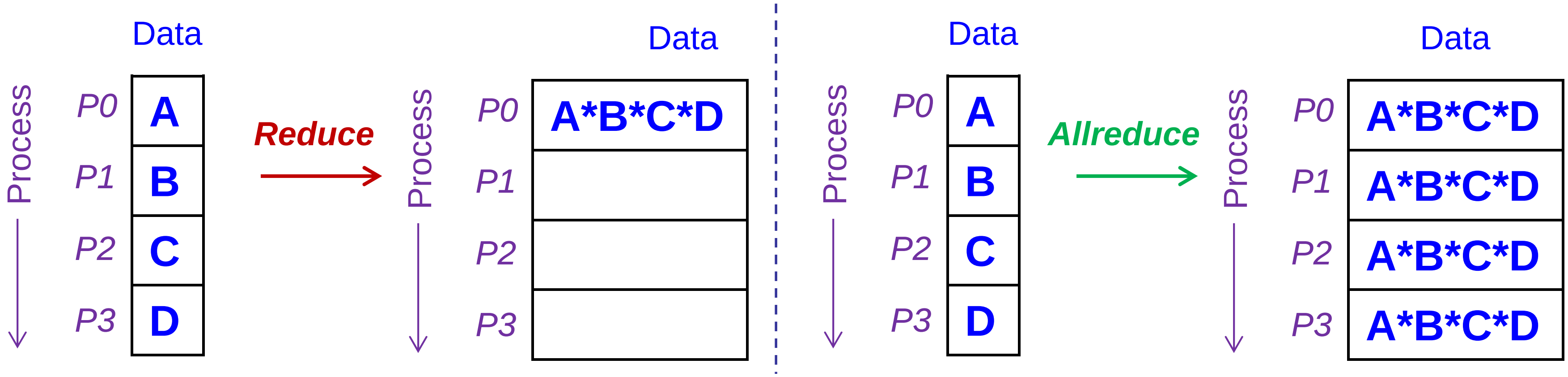
P0	A0	B0	C0	D0
P1	A1	B1	C1	D1
P2	A2	B2	C2	D2
P3	A3	B3	C3	D3

```
int MPI_Alltoall( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcnt,
                  MPI_Datatype recvtype, MPI_Comm comm )
```

6-7. Reduce and All Reduce

- *Reduce* applies a reduction operation on all tasks in the group and places the result in one task

- *Allreduce* applies a reduction operation and places the result in all tasks in the group. This is equivalent to an MPI_Reduce followed by an MPI_Bcast

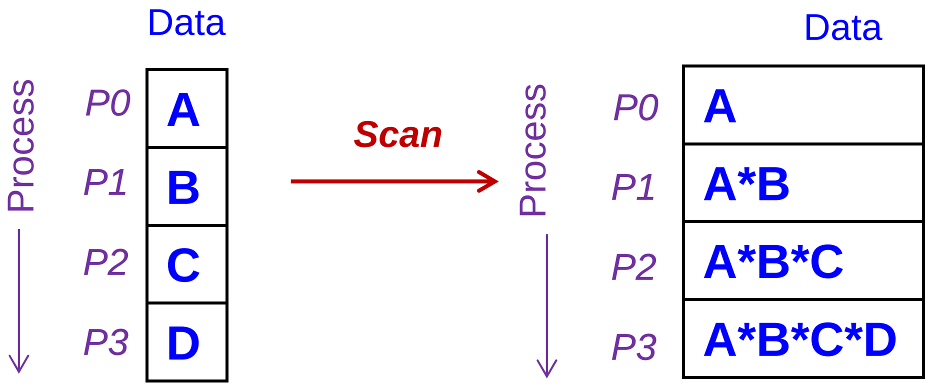


```
int MPI_Reduce ( void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int
                    root, MPI_Comm comm )
```

```
int MPI_Allreduce ( void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op,
                    MPI_Comm comm )
```

8. Scan

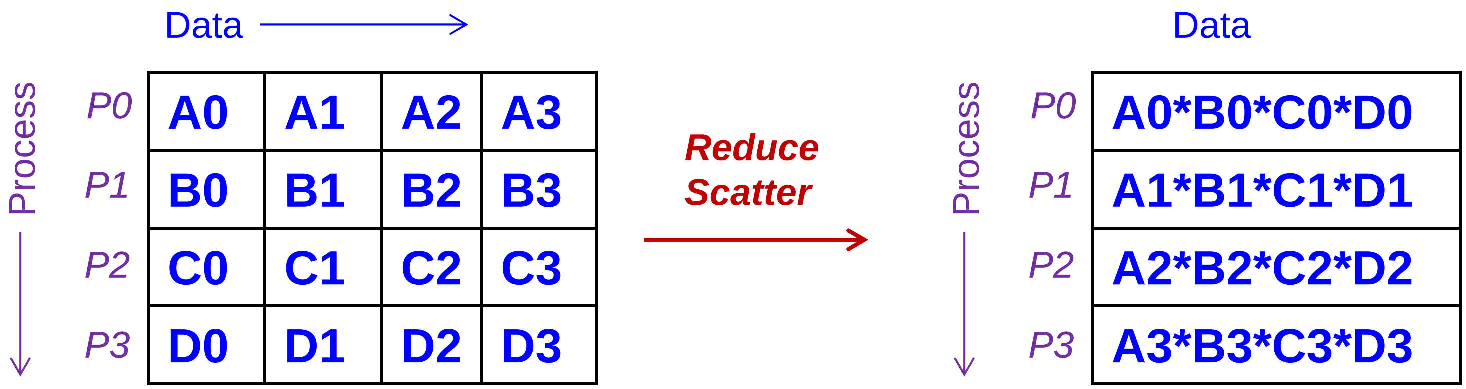
- *Scan* computes the scan (partial reductions) of data on a collection of processes



int MPI_Scan (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op,
MPI_Comm comm)

9. Reduce Scatter

- *Reduce Scatter* combines values and scatters the results. It is equivalent to an MPI_Reduce followed by an MPI_Scatter operation.



P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

P0	A0*B0*C0*D0
P1	A1*B1*C1*D1
P2	A2*B2*C2*D2
P3	A3*B3*C3*D3

```
int MPI_Reduce_scatter ( void *sendbuf, void *recvbuf, int *recvcnts, MPI_Datatype datatype, MPI_Op
                         op, MPI_Comm comm )
```

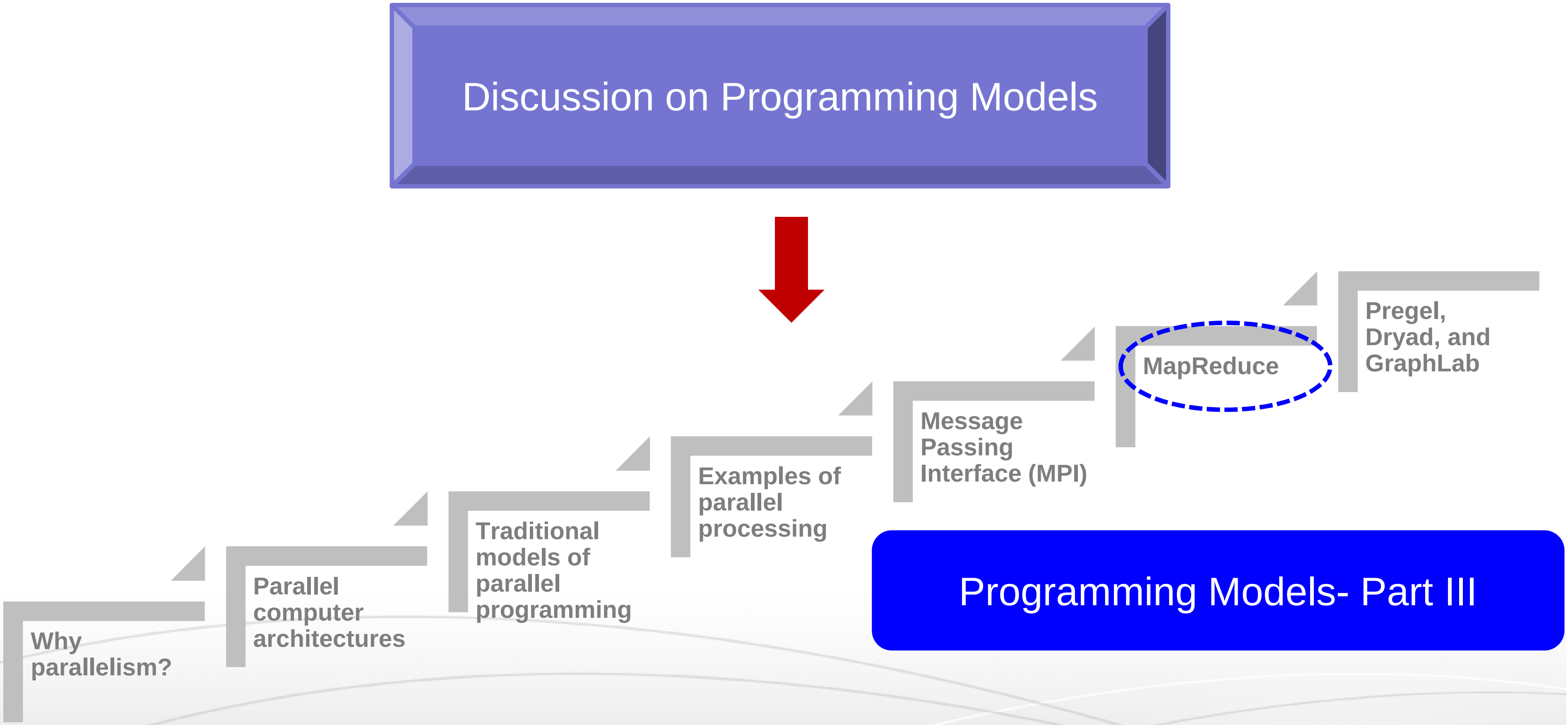
Considerations and Restrictions

- Collective operations are blocking

- Collective communication routines do not take message tag arguments

- Collective operations within subsets of processes are accomplished by first partitioning the subsets into new groups and then attaching the new groups to new communicators

Next Class



Discussion on Programming Models

Why parallelism?

Parallel computer architectures

Traditional models of parallel programming

Examples of parallel processing

Message Passing Interface (MPI)

MapReduce

Pregel, Dryad, and GraphLab

Programming Models- Part III