Parallel and Distributed Computing

CS3006

Lecture 15

Message Passing and MPI-II

16th May 2022

By: Dr. Rana Asif Rehman

Agenda

- Introduction to MPI
  - Basic Routines
  - Hello world Program
  - Basic Send/Receive Primitives
- Blocking and Non-blocking Operations + [Assigned reading]
- Avoiding Deadlocks in Send/Receive
- Sendrecv and Sendrecv_replace

Message Passing and MPI

Massage Passing Paradigm

Programming Using the Message Passing Paradigm

- Oldest and most widely used approach for distributed programming.
- The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
- Most of the communication is done using simple send/receive message passing.

Characteristics

- Provides high scalability
- Complex to program
- High communication costs
- No support for incremental parallelism

Message Passing Interface (MPI)

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran.
- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.
- It is possible to write fully-functional message-passing programs by using only the six routines.

Message Passing Interface (MPI)

The minimal set of MPI routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

Starting and Terminating the MPI Library

- `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- The prototypes of these two functions are:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

- `MPI_Init` also strips off any MPI related command-line arguments.
- All MPI routines, data-types, and constants are prefixed by "`MPI_`". The return code for successful completion is `MPI_SUCCESS`.

Communicators

- A communicator defines a *communication domain*
  - a set of processes that can communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.

Querying Information

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.
- The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

Hello World Program


```
1. #include <mpi.h>
2. main(int argc, char *argv[])
3. {
4.    int np, myrank;
5.    MPI_Init(&argc, &argv);
6.    MPI_Comm_size(MPI_COMM_WORLD, &np);
7.    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
8.    printf("From process %d out of %d,
      HelloWorld!\n",myrank, np);
9.    MPI_Finalize();
10.}
```

Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively.

- The calling sequences of these routines are as follows:

```
int MPI_Send(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype
datatype, int source, int tag,MPI_Comm comm,
MPI_Status *status)
```

- MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons.

- The message-tag can take values ranging from zero up to the MPI defined constant `MPI_TAG_UB`.

MPI Datatypes

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	

Sending and Receiving Messages

- MPI allows specification of **wildcard** arguments for both source and tag.
- If source is set to `MPI_ANY_SOURCE`, then any process of the communication domain can be the source of the message.
- If tag is set to `MPI_ANY_TAG`, then messages with any tag are accepted.

- On the receive side, the message must be of length **equal to or less than** the length field specified.

Sending and Receiving Messages

- On the receiving end, the status variable can be used to get information about the `MPI_Recv` operation.
- The corresponding data structure contains:

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR; };
```

- MPI_Status is usually used to take source and tag information in a 'receive' with wildcard entries on the corresponding positions.

The MPI_Get_count function returns the precise count of data items received.

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype
        datatype, int *count)
```

Sending and Receiving Messages

Example Program

```
if(my_rank==0){
        int sendBuff=10,tag=1,dest=1;
        printf("Process:%d is sending \'%d\' to process:%d \n",my_rank, sendBuff,dest);

        MPI_Send(&sendBuff, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);

}else if(my_rank==1){
        int recvBuff;int source=0,tag=1;

        MPI_Recv(&recvBuff, 1, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
        printf("Process:%d is has received \'%d\' from process:%d\n",my_rank,
        recvBuff,source);

        }else{
        }
```

Ensuring Operation Semantics

- Consider the following code segments:

```
P0                          P1
a = 100;                    receive(&a, 1, 0)
send(&a, 1, 1);             printf("%d\n", a);
a = 0;
```

- The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0.
- There may be an issue if infrastructure has **network interface hardware** for asynchronous send/receive without the involvement of CPU.
- After programming the network hardware, the control may return immediately to the next instruction, causing changes in the buffer before it is communicated to P1.
- Solutions?

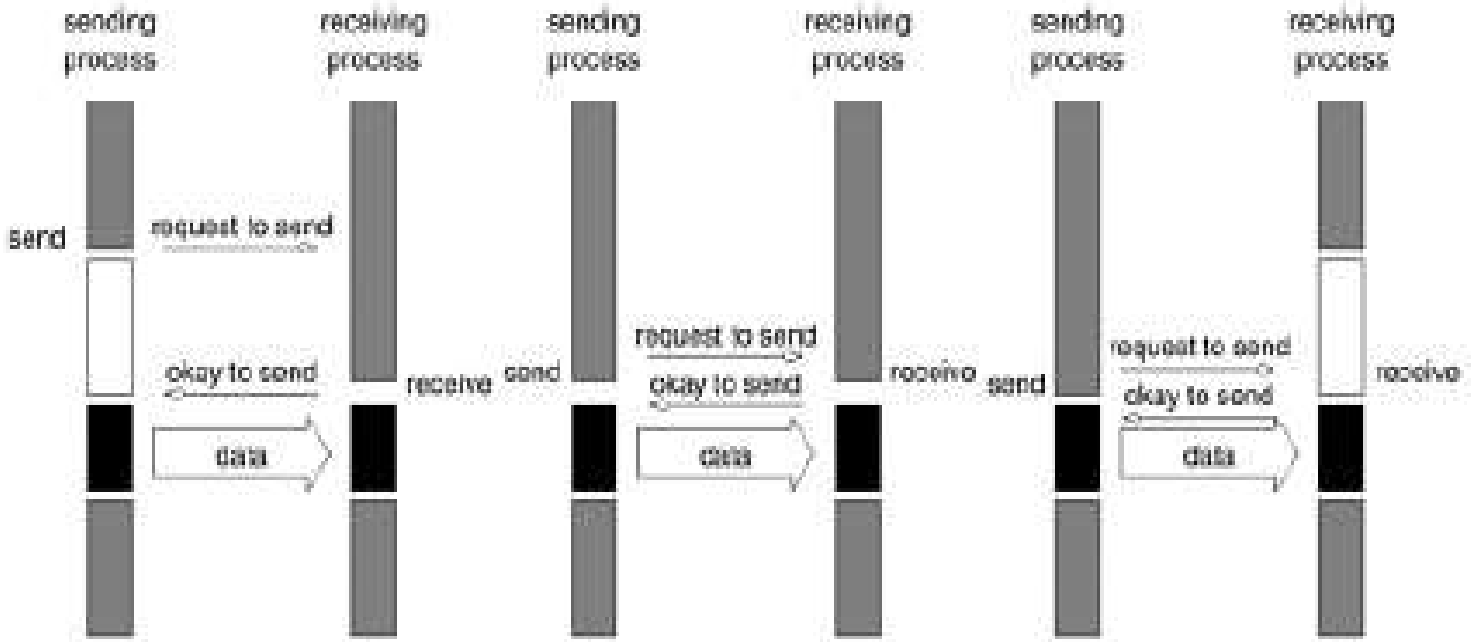
Blocking and non-blocking Operations

Solutions (Assigned Reading 6.2)

1. Blocking without Buffering
   - Simple and easy to enforce
   - Suffers idling and deadlocks
2. Blocking with Buffering
   - Reduces process idling at the cost of buffer management overheads
   - In **presence** of communication hardware, it stores message in a buffer at sender, and communication is done asynchronously when receiver approaches to corresponding receive.
   - In **absence** of communication hardware, sender interrupts the receiver and deposits data in buffer at receiver.
   - **Issues:** (bounded buffer and unexpected delays + blocking receives)
3. Non-blocking with and without buffers
   - Difficult to ensure semantics
   - Almost entirely masks the communication overheads
   - Recommended not to use

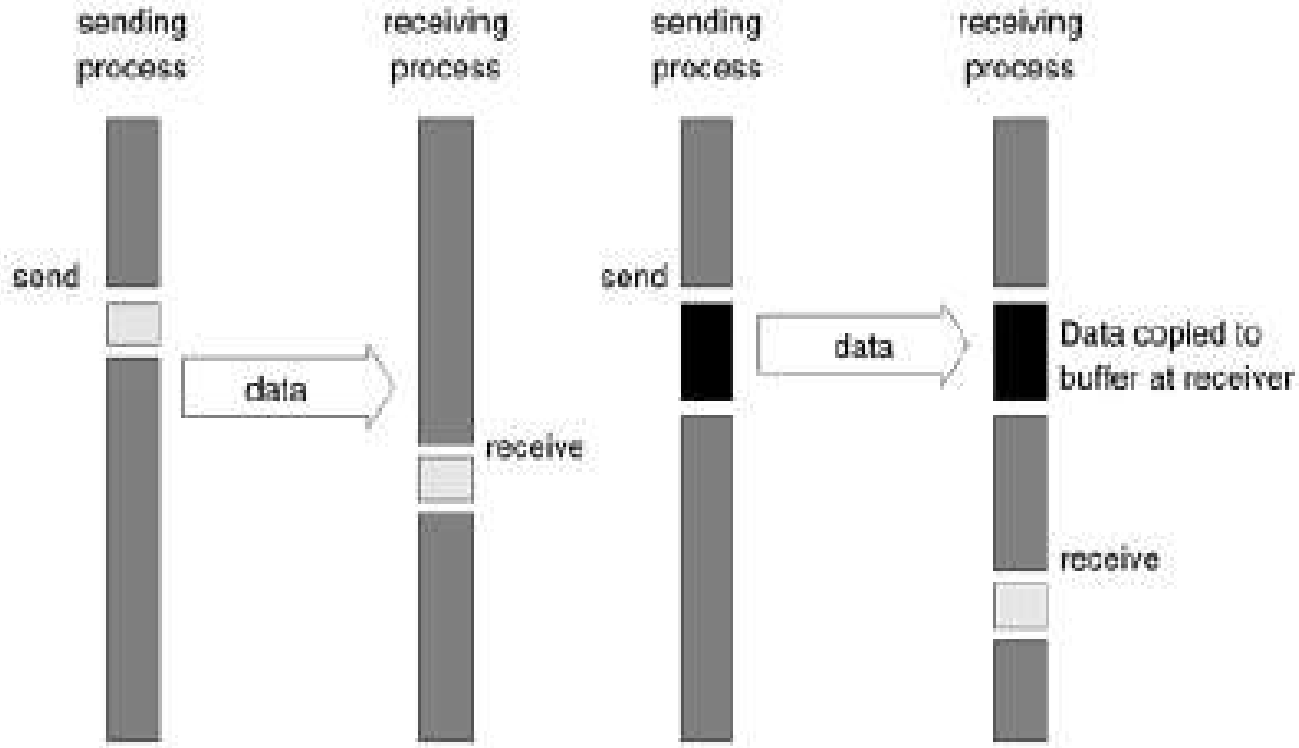
Blocking and non-blocking Operations

Figure 6.1. Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.



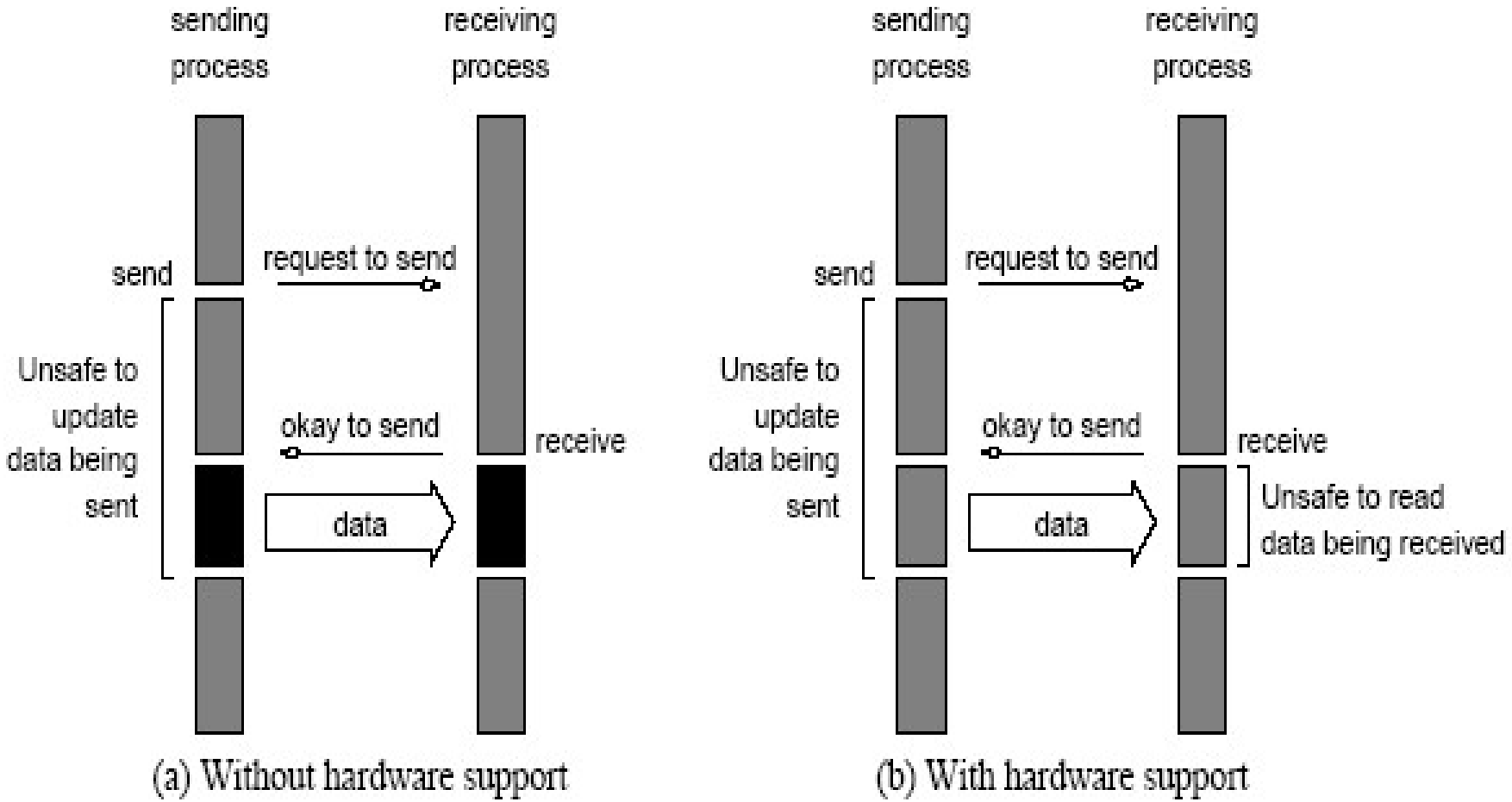
Blocking and non-blocking Operations

Figure 6.2. Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.



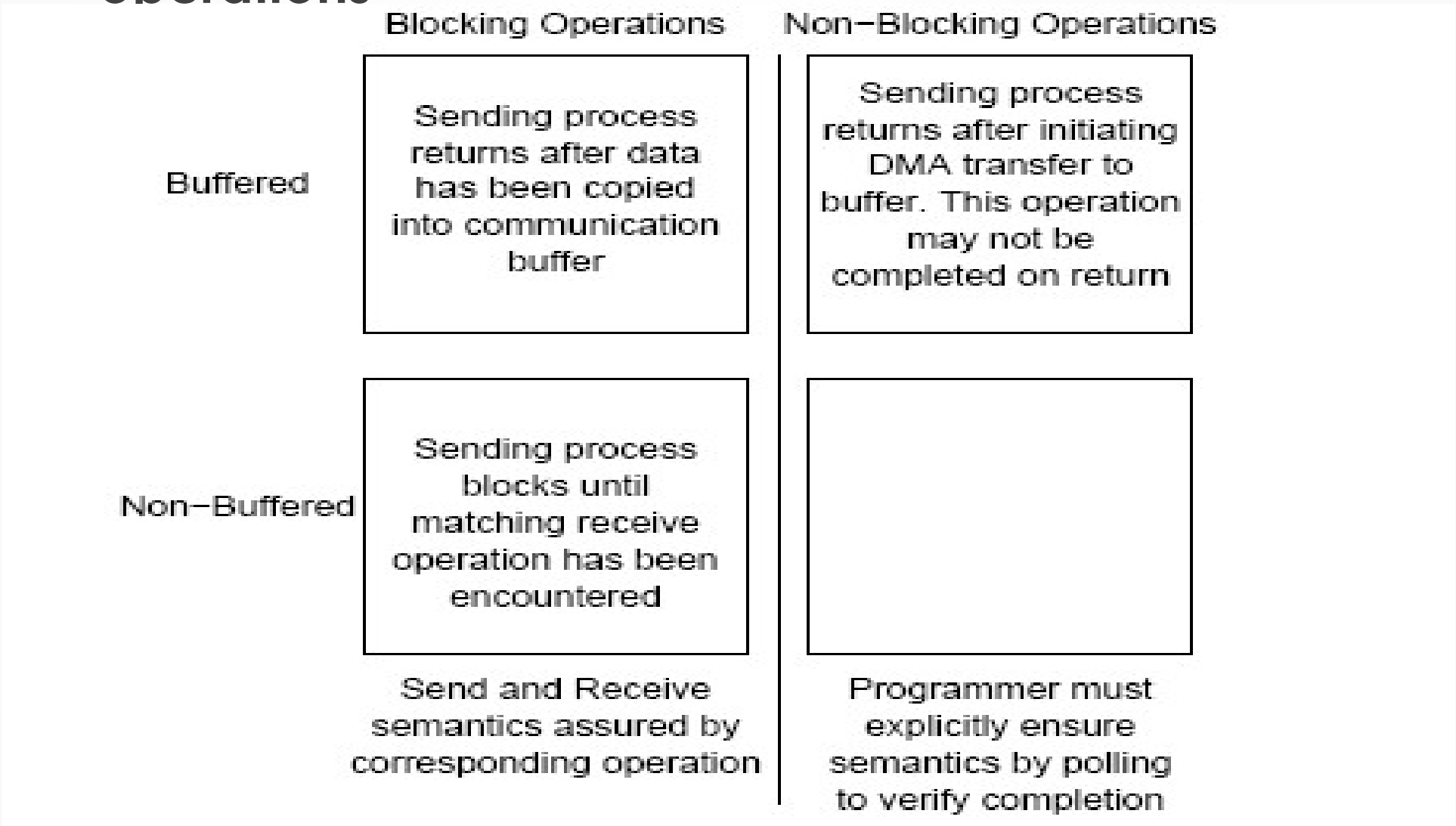
Blocking and non-blocking Operations

- Non-Blocking (without a buffer)



Blocking and non-blocking Operations

- Space of possible protocols for send and receive operations



MPI Rules for Send/Receive

- MPI usually uses blocking buffered Send only if there is enough buffer space to store whole message
- Otherwise, it uses blocking send
- Receive is always blocking

Deadlocks and Avoidance

- Let's see an example: **deadlocks.c**

Deadlocks (Circular)

Consider the following piece of code, in which process i sends a message to process i + 1 (modulo the number of processes) and receives a message from process i - 1 (module the number of processes).

```
1. int a[10], b[10], npes, myrank;
2. MPI_Status status;
3. ...
4. MPI_Comm_size(MPI_COMM_WORLD, &npes);
5. MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

6. MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
    MPI_COMM_WORLD);
7. MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
    MPI_COMM_WORLD);
   ...
```

- Once again, we have a deadlock if MPI_Send is blocking

Deadlocks→Solution

We can break the circular wait to avoid deadlocks as follows:

```
1.  int a[10], b[10], npes, myrank;
2.  MPI_Status status;
3.  ...
4.  MPI_Comm_size(MPI_COMM_WORLD, &npes);
5.  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6.  if (myrank%2 == 1) {
7.      MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
        MPI_COMM_WORLD);
8.      MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
        MPI_COMM_WORLD);
9.  }
10. else {
11.     MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
        MPI_COMM_WORLD);
12.     MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
        MPI_COMM_WORLD);
13. }
14. ...
```

Avoiding deadlocks using Simultaneous sendReceive operation

- To avoid earlier deadlocks, MPI provides **MPI_Sendrecv** function
  - It can both send and receive message
  - Does not suffers from the circular deadlock problems
  - One can think MPI_Sendrecv as allowing data to travel for both send and receive simultaneously.

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
                 MPI_Datatype senddatatype, int dest,int sendtag,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvdatatype, int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status)
```

- **Programming example:** sendReceive_simult.c

Avoiding deadlocks using Simultaneous sendReceive operation

- **MPI_Sendrecv_replace** function
  - If we wish to use the same buffer for both send and receive
  - First sends value[s] of current buffer and then overwrites them with received ones
- **Syntax**

```
int MPI_Sendrecv_replace(void *buf, int count,
      MPI_Datatype datatype, int dest, int
      sendtag,int source, int recvtag,
      MPI_Comm comm, MPI_Status *status)
```

Questions

References

1. Kumar, V., Grama, A., Gupta, A., & Karypis, G. (2017). *Introduction to parallel computing*. Redwood City, CA: Benjamin/Cummings.