Parallel and Distributed Computing

CS3006

Lecture 15

Message Passing and MPI-II

22nd April 2024

By: Dr. Rana Asif Rehman

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?

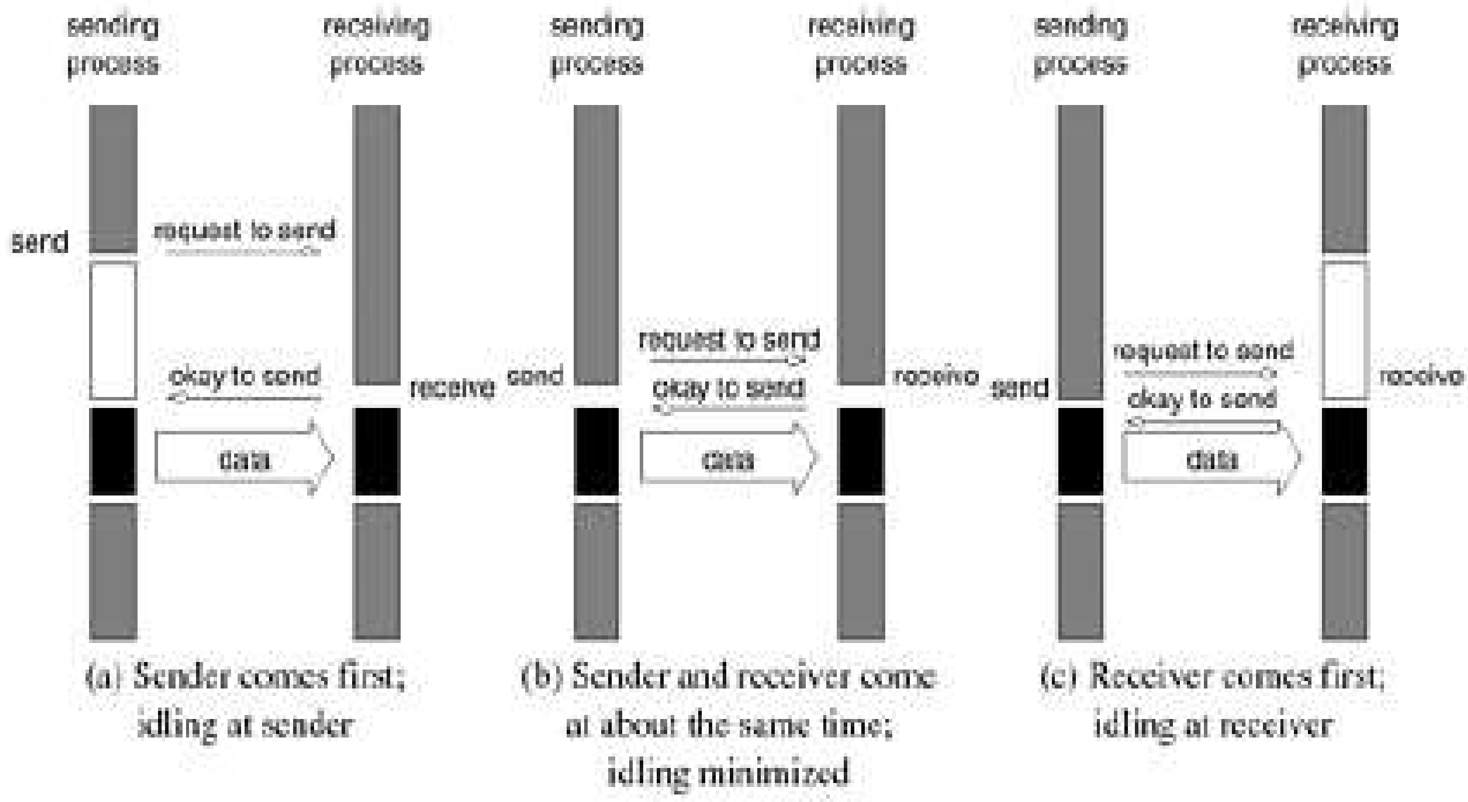
Blocking and non-blocking Operations

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

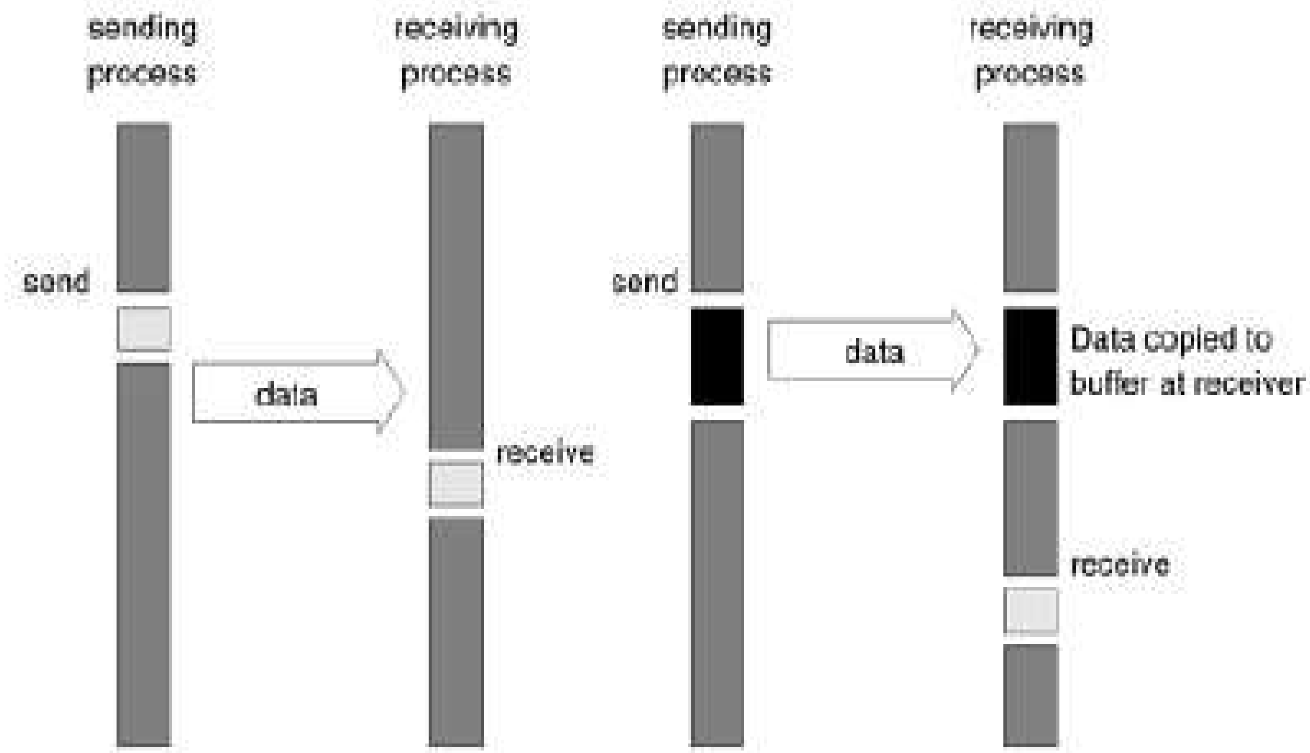
Blocking and non-blocking Operations

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.



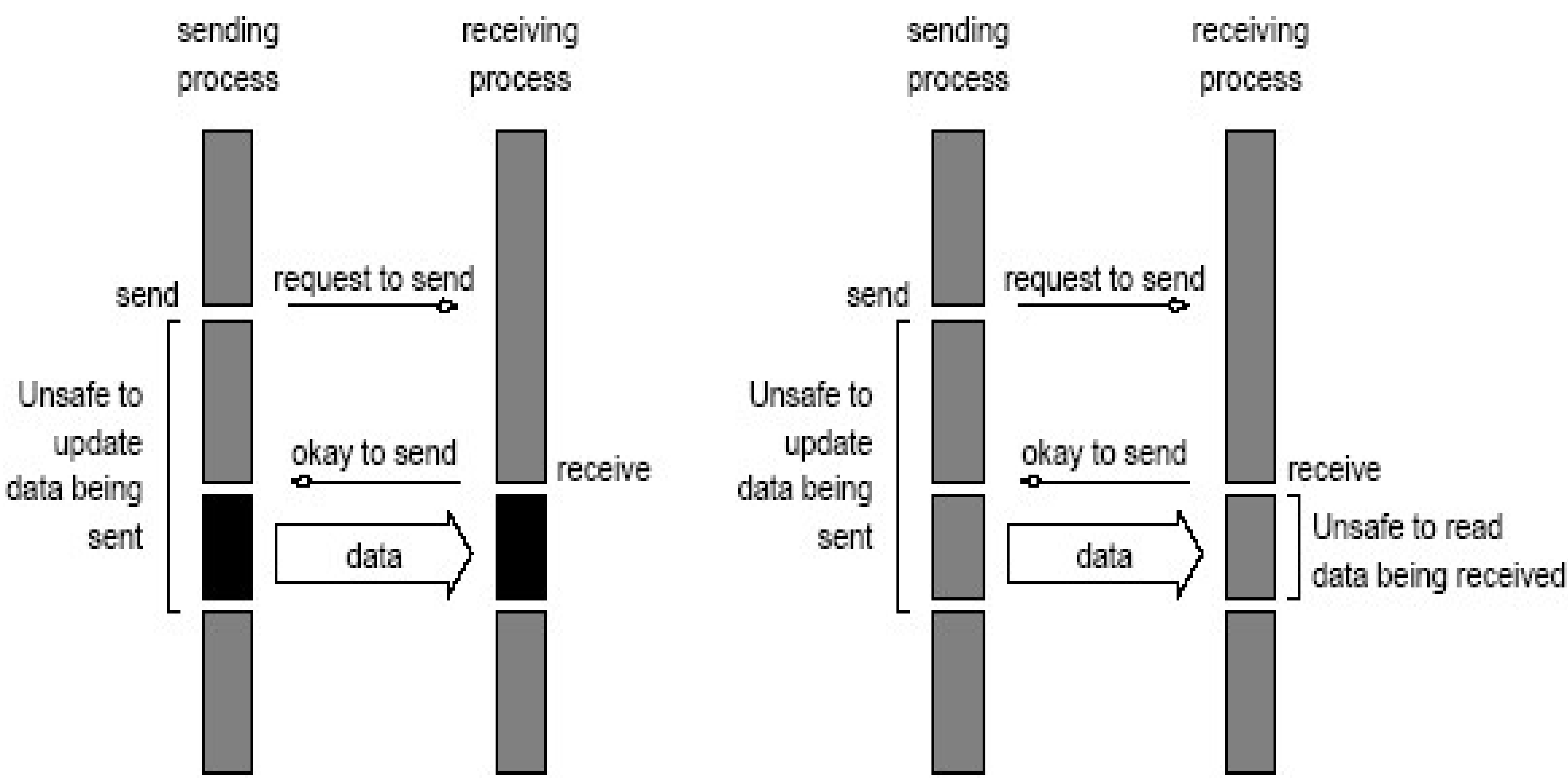
Blocking and non-blocking Operations

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)



(a) Without hardware support

(b) With hardware support

Blocking and non-blocking Operations

- Space of possible protocols for send and receive operations

	Blocking Operations	Non–Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non–Buffered	Sending process blocks until matching receive operation has been encountered	
	Send and Receive semantics assured by corresponding operation	Programmer must explicitly ensure semantics by polling to verify completion

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

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).

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

MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
 MPI_COMM_WORLD);
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:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
    MPI_COMM_WORLD);
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
    MPI_COMM_WORLD);
}
else {
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
    MPI_COMM_WORLD);
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
    MPI_COMM_WORLD);
}
...
```

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)
```

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.