BITS Pilani
Pilani Campus

Message Passing Model

K Hari Babu
Department of Computer Science & Information Systems

innovate achieve lead

BITS Pilani
Pilani Campus

Communication Primitives

The Building Blocks: Send and Receive Operations

- The basic operations in the message-passing programming paradigm are send and receive

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

  - The sendbuf points to a buffer that stores the data to be sent, recvbuf points to a buffer that stores the data to be received, nelems is the number of data units to be sent and received, dest is the identifier of the process that receives the data, and source is the identifier of the process that sends the data

The Building Blocks: Send and Receive Operations

- The basic operations in the message-passing programming paradigm are send and receive

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- Consider a simple example of a process sending a piece of data to another process

```
P0

a = 100;
send(&a, 1, 1);
a=0;
```

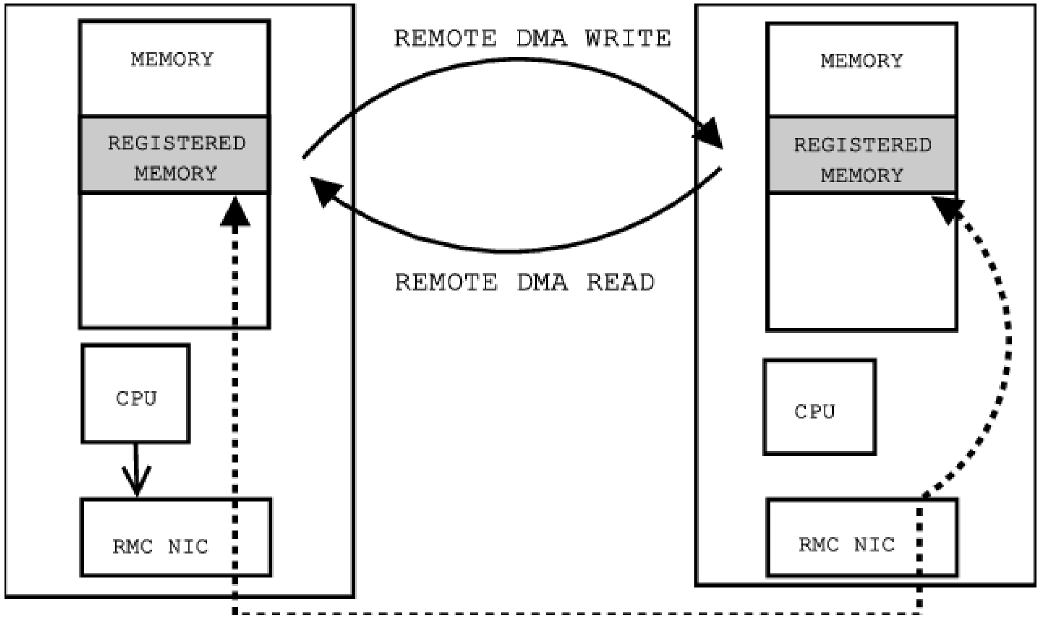
```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

  - What will be the value printed by P1? 100 or 0?
    - Based on how the send and receive operations are implemented, it can be 100 or 0.

BITS Pilani

Hardware Support: Remote Direct Memory Access (RDMA)

- RDMA supports zero-copy networking by enabling the network adapter to transfer data from the wire directly to application memory or from application memory directly to the wire
  - Eliminates the need to copy data between application memory and the data buffers in the operating system
    - Such data transfers require no work to be done by CPUs, caches, or context switches, and transfers continue in parallel with other system operations
    - This reduces latency in message transfer.



The Building Blocks: Send and Receive Operations

BITS Pilani

- Consider a simple example of a process sending a piece of data to another process

```
P0

a = 100;
send(&a, 1, 1);
a=0;
```

```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

  - What will be the value printed by P1?
    - 100 or 0?
  - DMA allows copying of data from one memory location to another or to communication buffers without CPU support (once they have been programmed)
  - As a result, if the send operation programs the communication hardware and returns before the communication operation has been accomplished, process P1 might receive the value 0 in a instead of 100

Blocking Message Passing Operations

- A simple solution to the problem (in prev slide)
  - is for the send operation to return only when it is semantically safe to do so
    - It can mean that the send operation returns only after the receiver has received the data OR
    - It can mean that data is copied locally or directly to another system memory using RDMA before send returns
- Sending operation blocks until it can guarantee that the semantics will not be violated on return irrespective of what happens in the program subsequently
- There are two mechanisms by which this can be achieved
  - Blocking Non-Buffered Send/Receive
  - Blocking Buffered Send/Receive

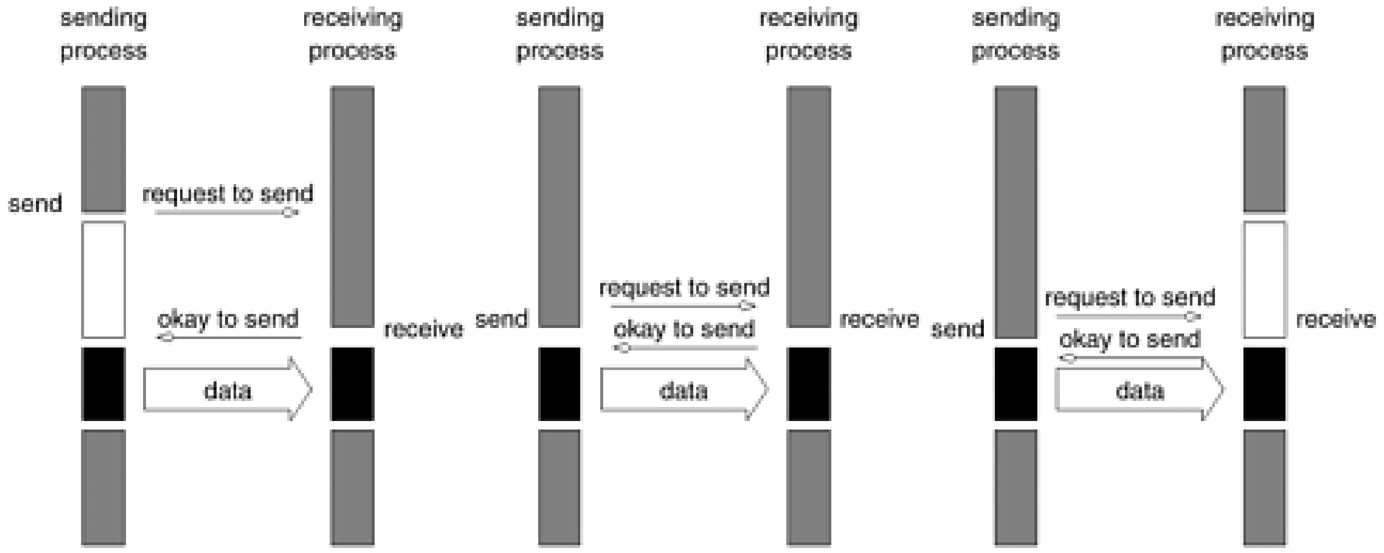
Blocking Non-Buffered Send/Receive

- In the first case, the send operation does not return until the matching receive has been encountered at the receiving process
- When this happens, the message is sent and the send operation returns upon completion of the communication operation
  - This involves a handshake between the sending and receiving processes
  - The sending process sends a request to communicate to the receiving process
  - When the receiving process encounters the target receive, it responds to the request
    - Sender blocks until receiver responds → idling overhead
  - The sending process upon receiving this response initiates a transfer operation

Blocking Non-Buffered Send/Receive

BITS Pilani

- It involves a handshake between the sending and receiving processes
  - The sending process sends a request to communicate to the receiving process
  - When the receiving process encounters the target receive, it responds to the request
  - The sending process upon receiving this response initiates a transfer operation



(a) Sender comes first; idling at sender

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

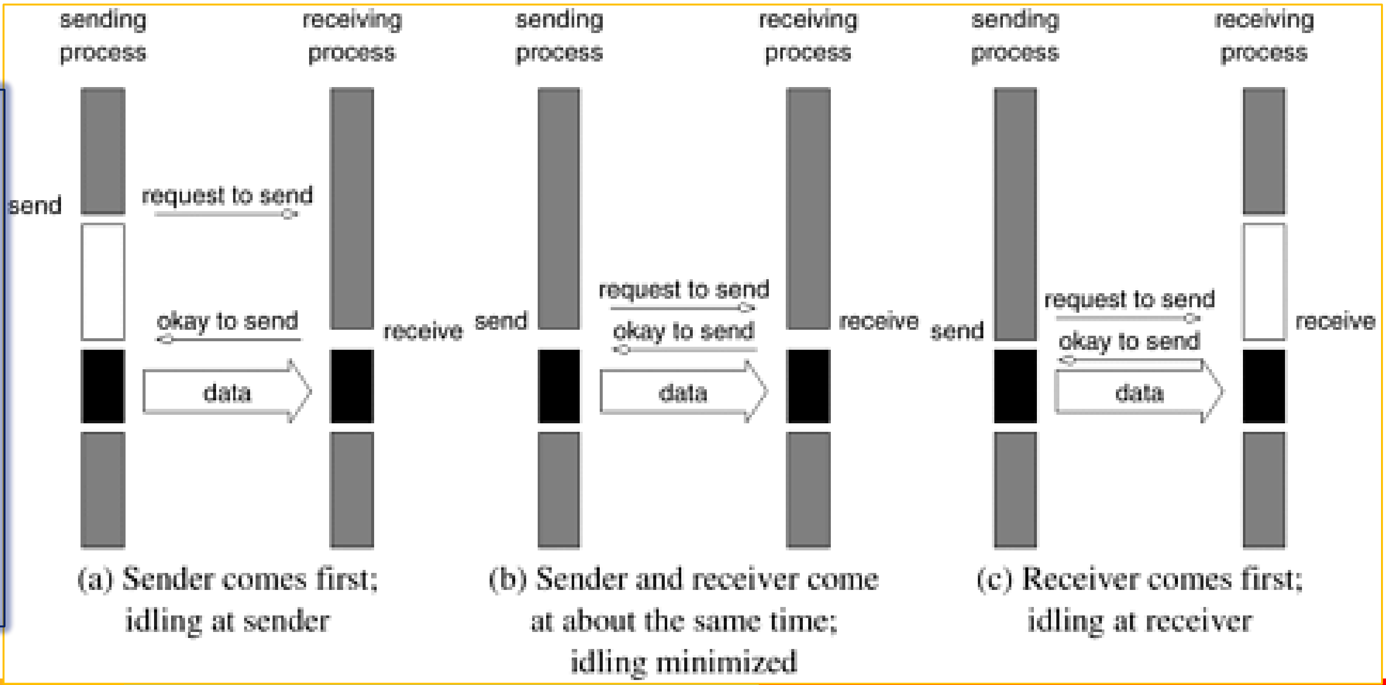
(c) Receiver comes first; idling at receiver

Blocking Non-Buffered Send/Receive

BITS Pilani

- Idling Overheads in Blocking Non-Buffered Operations
  - three scenarios
    - (a) the send is reached before the receive is posted, (b) the send and receive are posted around the same time, and (c) the receive is posted before the send is reached
  - In cases (a) and (c), there is considerable idling at the sending and receiving process

A blocking non-buffered protocol is suitable when the send and receive are posted at roughly the same time. However, in an asynchronous environment, this may be impossible to predict. This idling overhead is one of the major drawbacks of this protocol.



(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 Non-Buffered Send/Receive

BITS Pilani

- Deadlocks in Blocking Non-Buffered Operations
  - Consider the following simple exchange of messages

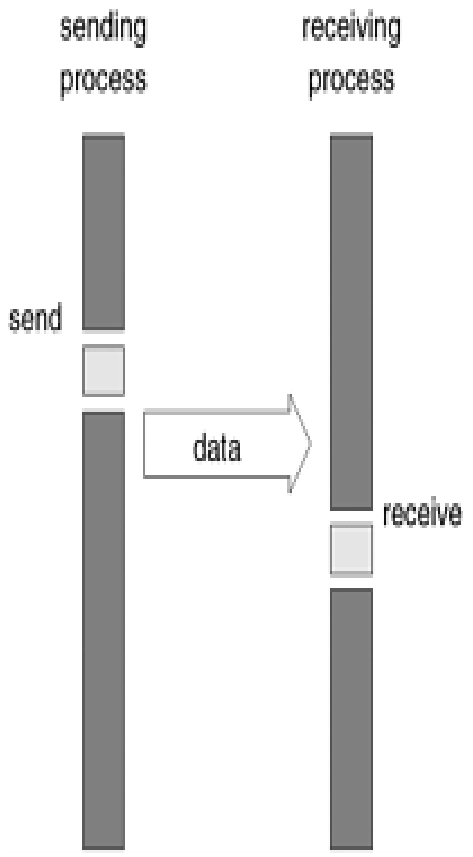
```
1        P0                          P1
2
3        send(&a, 1, 1);                  send(&a, 1, 0);
4        receive(&b, 1, 1);                receive(&b, 1, 0);
```

  - Deadlock!
    - If the send and receive operations are implemented using a blocking non-buffered protocol, the send at P0 waits for the matching receive at P1 whereas the send at process P1 waits for the corresponding receive at P0
  - Deadlocks are very easy in blocking protocols and care must be taken
  - The above example can be corrected by replacing the operation sequence of one of the processes by a receive and a send

Blocking Buffered Send/Receive

BITS Pilani

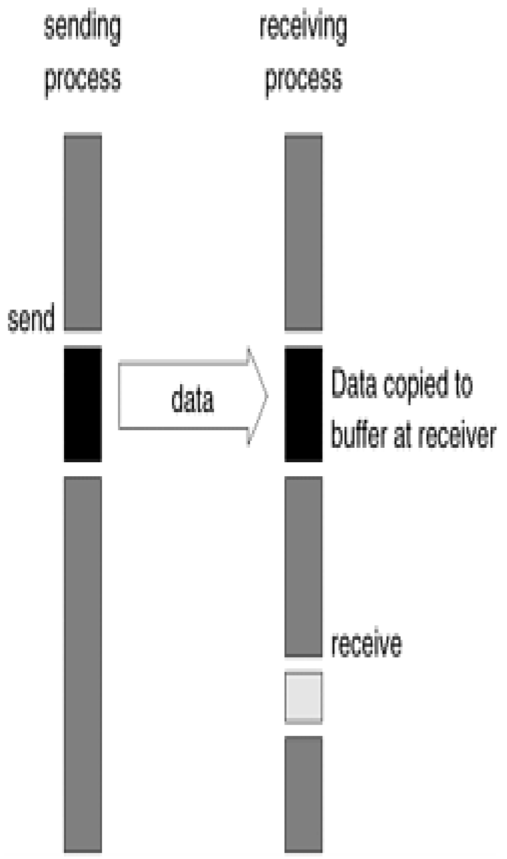
- A simple solution to the idling and deadlocking problem is
  - rely on buffers at the sending and receiving ends
    - A simple case: the sender has a buffer pre-allocated for communicating messages
    - Sender simply copies the data into the designated buffer and returns
    - The sender process can now continue with the program
    - Any changes done to the data will not impact program semantics
    - If the hardware supports remote DMA, a network transfer can be initiated
    - At the receiving end, the data cannot be stored directly at the target location since this would violate program semantics
    - Instead, the data is copied into a buffer at the receiver as well. When the receiving process encounters a receive operation, it checks to see if the message is available in its receive buffer. If so, the data is copied into the target location



Blocking Buffered Send/Receive

BITS Pilani

- Buffering introduces buffer management overhead
  - Copying to a local buffer can be done by
    - CPU (no hardware support)
    - DMA (hardware support)
  - Sending message across the network can also be done by
    - CPU (no hardware support)
    - RDMA (hardware support)
  - When CPU is used, process is blocked
- When no hardware support, by buffering only at one end, reduces copying overhead
    - The sender interrupts the receiver, both processes handshake, and the message is deposited in a buffer at the receiver end. When the receiver encounters a receive operation, the message is copied from the buffer into the application buffer



Blocking Buffered Send/Receive

- Buffered protocols alleviate idling overheads at the cost of adding buffer management overheads
- If the parallel program is highly synchronous (i.e., sends and receives are posted around the same time)
  - non-buffered sends may perform better than buffered sends

Blocking Buffered Send/Receive

BITS Pilani

- Impact of finite buffers in message passing
  - Consider the following code fragment where buffering is only at P1:

```
1       P0                              P1
2
3       for (i = 0; i < 1000; i++) {       for (i = 0; i < 1000; i++) {
4         produce_data(&a);                   receive(&a, 1, 0);
5         send(&a, 1, 1);                  consume_data(&a);
6       }                                }
```

  - Process P0 produces 1000 data items and process P1 consumes them
  - If process P1 is slow in execution, process P0 might have sent all of its data what if there is no enough buffer space?
    - then the sender has to be blocked until some of the corresponding receive operations had been posted, thus freeing up buffer space
    - This can often lead to unforeseen overheads and performance degradation

Blocking Buffered Send/Receive

BITS Pilani

- Deadlocks in Buffered Send and Receive Operations
  - the fact is that receive calls are always blocking (to ensure semantic consistency) even in buffered case as in the non-buffered case.

```
P0                      P1

receive(&a, 1, 1);        receive(&a, 1, 0);
send(&b, 1, 1);          send(&b, 1, 0);
```

  - Is there any deadlock in this?

Non-Blocking Message Passing Operations

- In blocking protocols, semantic correctness is at the cost of
  - idling (non-buffered) or buffer management (buffered) overhead
- Another approach is non-blocking:
  - leave the responsibility of ensuring semantic correctness to the programmer
  - This provides a fast send/receive operation that incurs little overhead
  - Send and recv return before it is semantically safe to do so
  - Send and recv are accompanied by a check-status operation
  - After a non-blocking send or receive operation, the process is free to perform any computation that does not depend upon the completion of the operation
    - Later in the program, the process can check whether or not the non-blocking operation has completed, and, if necessary, wait for its completion.

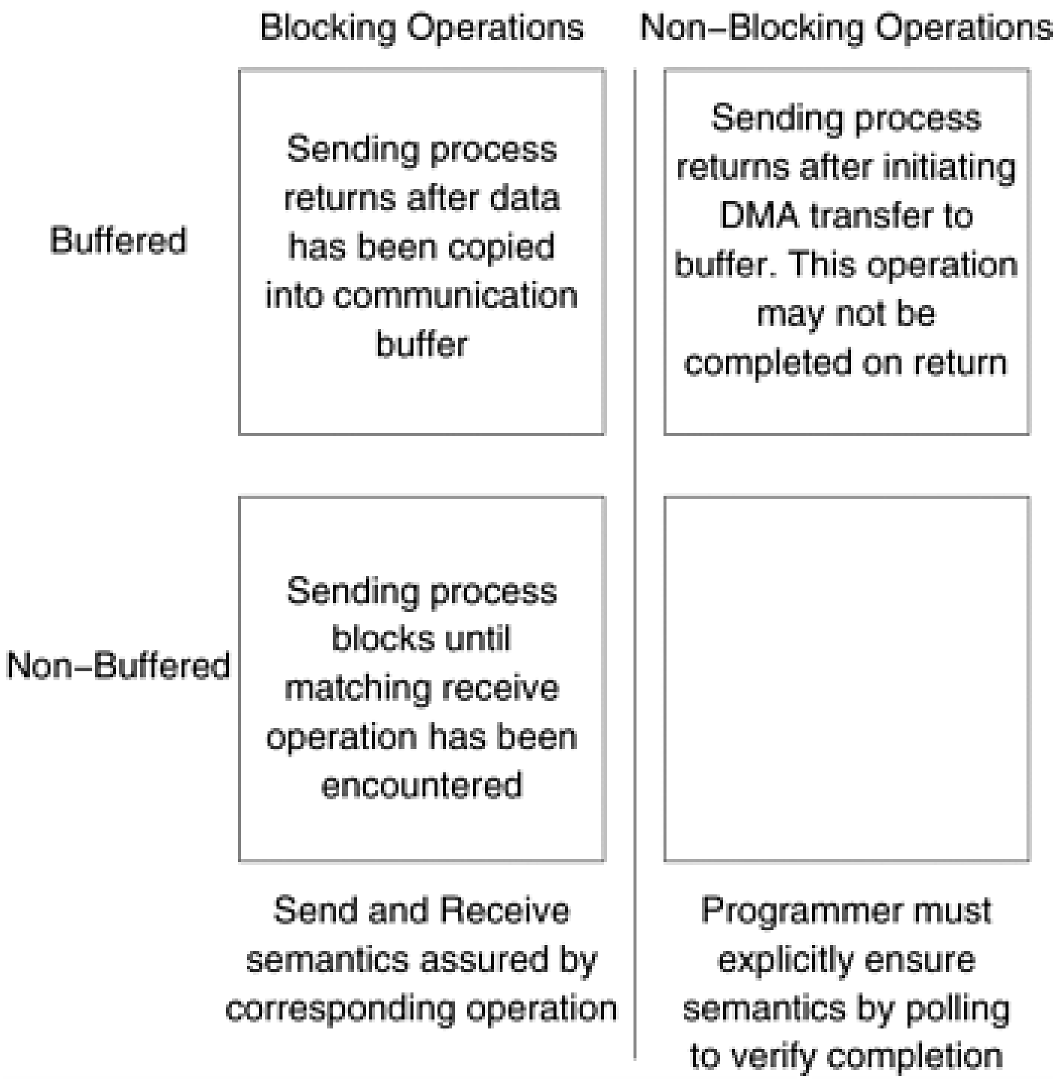
Non-Blocking Message Passing Operations

- Non-blocking operations can themselves be buffered or non-buffered
- In the non-buffered case
  - a process posts a pending message and returns. The program can then do other useful work
  - At some point in the future, when the corresponding receive is posted, the communication operation is initiated
  - When this operation is completed, the check-status operation indicates that it is safe for the programmer to touch this data
  - Data transfer may be done by CPU or a dedicated hardware (rDMA).
    - When it is done remote DMA, the process can completely overlap the communication overhead with computation time

Non-Blocking Message Passing Operations

- Non-blocking operations with a buffered protocol
  - In this case, the sender initiates a DMA operation and returns immediately
  - The data becomes safe the moment the DMA operation has been completed
  - At the receiving end, the receive operation initiates a transfer from the sender's buffer to the receiver's target location
  - Using buffers with non-blocking operation has the effect of reducing the time during which the data is unsafe.

Summary



	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

Summary

- Message-passing libraries Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) implement both blocking and non-blocking operations
- Blocking operations facilitate safe and easier programming and non-blocking operations are useful for performance optimization by masking communication overhead

References

- Ananth Grama, Anshul Gupta, George Karypis & Vipin Kumar Introduction to Parallel Computing, Second Edition, Pearson Education, First Indian Reprint 2004. (Section 6.2)

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You

Q&A

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You