MESSAGE PASSING PARADIGM

Dr. Emmanuel Pilli

Principles of Message Passing

- The logical view of a machine supporting the message-passing paradigm consists of *p* processes, each with its own exclusive address space.
- Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
- All interactions (read-only or read / write) require cooperation of two processes - the process that has the data and process that wants to access the data.
- These two constraints make underlying costs very explicit to the programmer.

Principles of Message Passing

- Message-passing programs are often written using the *asynchronous* paradigm or *loosely synchronous* paradigm.
- In the *asynchronous* paradigm, all concurrent tasks execute asynchronously.
- In the *loosely synchronous* model, tasks or subsets of tasks synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously.
- Most message-passing programs are written using the *single program multiple data* (SPMD) model.

Send and Receive Operations

- The prototypes of these operations are as follows:

```
send(void *sendbuf, int nelems, int dest)

receive(void *recvbuf, int nelems, int source)
```

- Consider the following code segments:

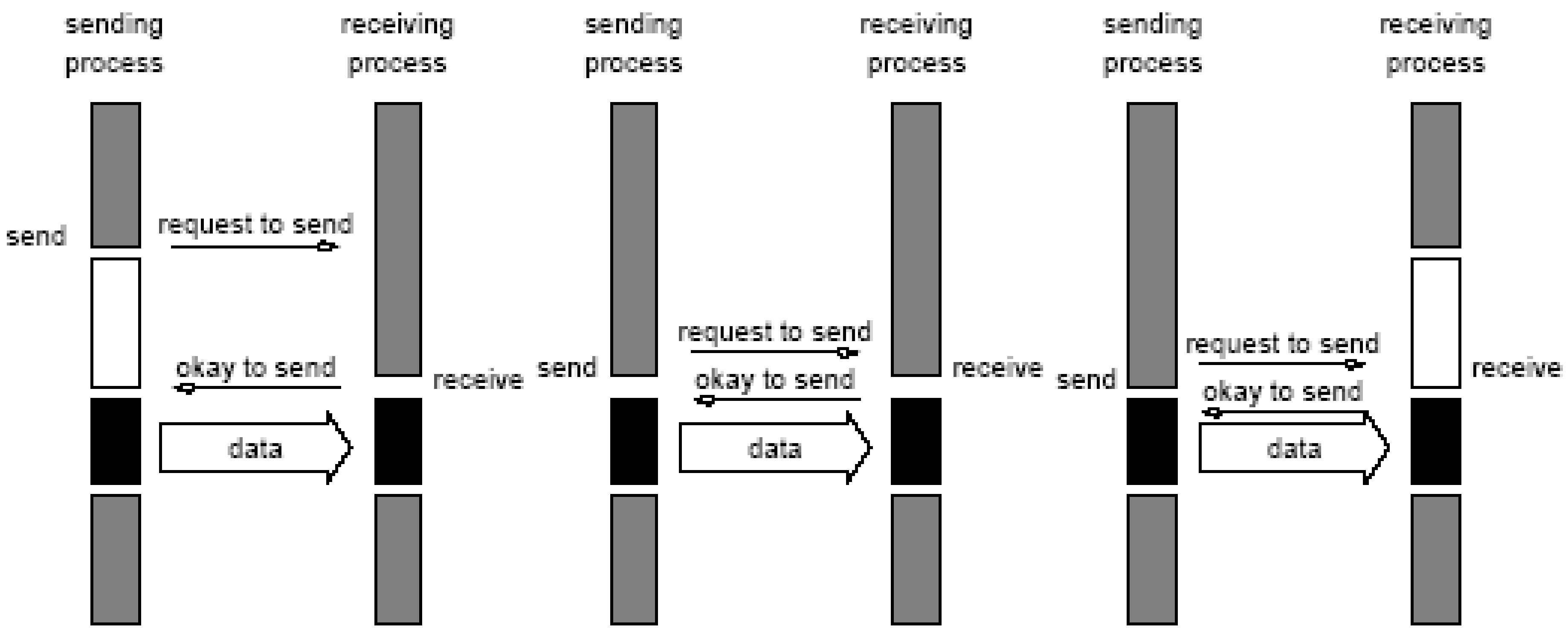
```
P0              P1
a = 100;        receive(&a, 1, 0)
send(&a, 1, 1);    printf("%d\n", a);
a = 0;
```

- The semantics of the send operation require that the value received by process P1 must be 100, but not 0.
- This motivates the design of the send and receive protocols.

Non Buffered Blocking

- A simple method for forcing send / receive semantics is for the send operation to return only when it is safe to do so.
- In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process.
- Idling and deadlocks are major issues with non-buffered blocking sends.

Non Buffered Blocking



(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

Handshake for a blocking non-buffered send/receive operation.

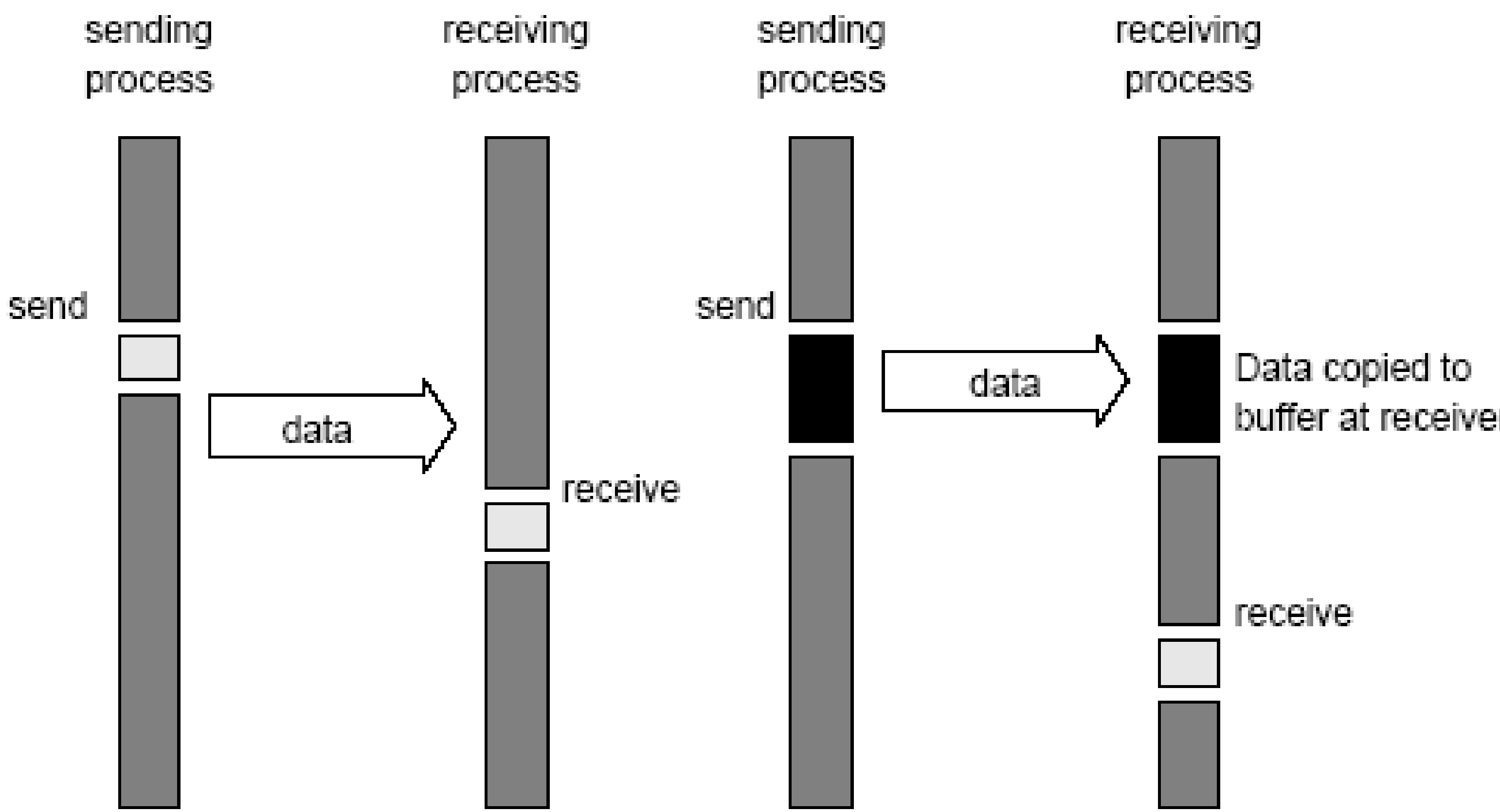
Buffered Blocking

- In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The data is copied at a buffer at the receiving end as well.
- Buffering alleviates idling at the expense of copying overheads.

Buffered Blocking

- A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends.
- The sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
- The data must be buffered at the receiving end as well.
- Buffering trades off idling overhead for buffer copying overhead.

Buffered Blocking



Blocking buffered transfer protocols
(a) in the presence of communication hardware
(b) in the absence of communication hardware

Buffered Blocking

- Bounded buffer sizes can have significant impact on performance.

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

- What if consumer was much slower than producer?

Buffered Blocking

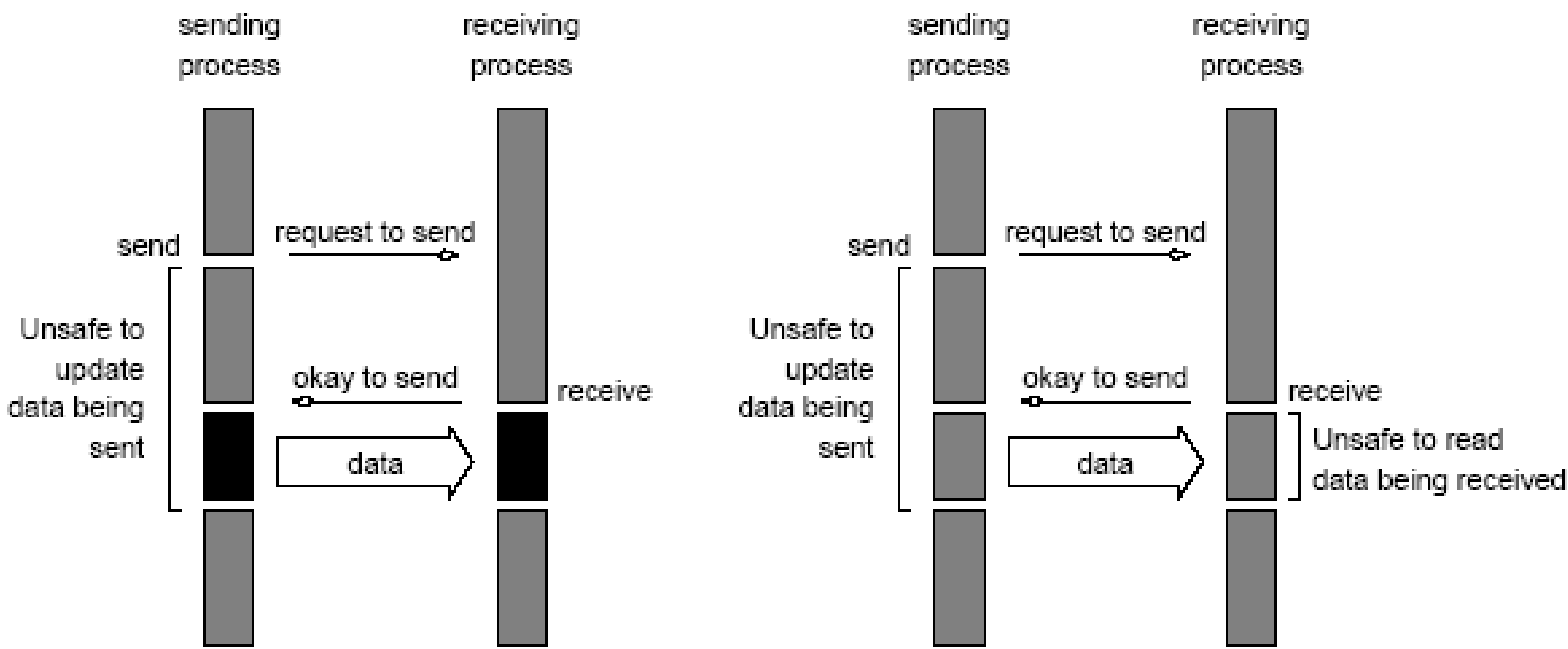
- Deadlocks are still possible with buffering since receive operations block.

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

Non Blocking

- The programmer must ensure semantics of the send and receive.
- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.
- Non-blocking operations are generally accompanied by a check-status operation.
- When used correctly, these primitives are capable of overlapping communication overheads with useful computations.
- Message passing libraries typically provide both blocking and non-blocking primitives.

Non Blocking



(a) Without hardware support

(b) With hardware support

Non-blocking non-buffered send and receive operations
(a) In absence of communication hardware;
(b) in presence of communication hardware.

Send and Receive Protocols

	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: Message Passing Interface

- 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.
- Vendor implementations of MPI are available on almost all commercial parallel computers.
- It is possible to write fully-functional message-passing programs by using only the six routines.

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

- 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()
```

Starting and Terminating

- 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.

Communicator

- A communicator defines a *communication domain* - a set of processes that are allowed to 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.

Number and Rank of Process

- The MPI_Comm_size and MPI_Comm_rank functions are used to determine the number of processes and the label of the calling process.
- 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.

First MPI Program

```
#include <mpi.h>
main(int argc, char *argv[])
{
   int npes, myrank;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &npes);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

   printf("From process %d out of %d,
          Hello World!\n", myrank, npes);
   MPI_Finalize();
}
```

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)

Sending and Receiving Messages

- MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons.
- The datatype MPI_BYTE corresponds to a byte (8 bits) and MPI_PACKED corresponds to a collection of data items that has been created by packing non-contiguous data.
- 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	
MPI_PACKED	

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.
- The corresponding data structure contains:

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

- 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)
```