Week # 9 incuding Lab # 6 slides

- MPI syllabus as per course outline
- MPI Coverage started from Week # 8 (Lab # 5) see Lab slides.

MPI Syllabus (as per Course outline)

Programming Using the Message Passing Paradigm: Principles of MPI, The Building Blocks: Send and Receive Operations, Buffered and non buffered MPI, MPI interface, Starting and Terminating the MPI Library, Communicators, Querying Information, Sending and Receiving Messages, overlapping communication with computation.

Lab3: MPI installation, Communication rank and size in MPI, MPI_send / MPI_Recv, MPI_status, MPI_Tag.

(Book : Book#1)

(9^{th} Oct,23 – 13^{th} Oct,23)

Programming Using the Message Passing Paradigm: collective communication and computation opertaions: barrier, broadcast, reduction, prefix, scatter, gather.

Lab4: MPI Scatter, Gather, Bcast, MPI_wait, MPI_Test, MPI_Allgather.

(Book : Book#1)

(16^{th} Oct,23-20^{th} Oct,23)

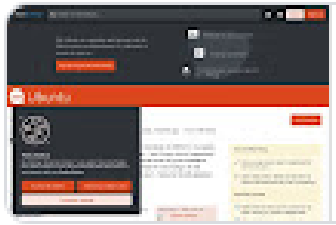
Nadeem Kafi
Aug 27

Show it to your TA at the time of Semester Project DEMO

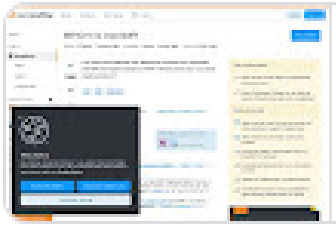
Installing OpenMP and MPI (MPICH) on your PC/Laptop or Lab PC on UBUNTU 20.x OS

- Use only Ubuntu 20.x OS only. Install Ubuntu using Virtualbox on your Windows OS laptop.
- Using OpenMP is already known (as per OS Lab)
- Installing MPI library MPICH https://askubuntu.com/questions/1236553/mpich-installation
- Compile the MPI program given in the above link.
- Optional: See the link below about MPI libraries OpenMP vs MPICH

NK.



mpi - MPICH installation - ...
https://askubuntu.com/question



mpi - MPICH vs OpenMPI ...
https://stackoverflow.com/quest

1 class comment

Nadeem Kafi Sep 15

Please note that any Ubuntu version which compiles OpenMP and MPI code will work for lab submissions.

Programming Models

- A programming model is how you *think* about what the computer is doing when it executes your program.
  - This may not be what the computer *is* doing.
- For example, in a sequential programming model, you write the program *as if* the computer is executing your code one statement at a time, in the order you have written.
  - With modern architectures and compilers, featuring instruction-level parallelism and out-of-order execution, what really happens will be quite different.
- Programming models differ greatly in their *level of abstraction* (how far the the programmer's mental model is from what the computer actually does in response to the program (as partially directed by the compiler).
  - High-level (e.g. Prolog, Lisp, ML): portability, conciseness, "ease of programming"
  - Low-level (e.g. assembly language): performance

What is MPI?

- MPI is a message-passing library interface standard.
  - Specification, not implementation
  - Library, not a language
  - Classical message-passing programming model
- MPI-1 was defined (1994) by a broadly-based group of parallel computer vendors, computer scientists, and applications developers.
  - 2-year intensive process
- Implementations appeared quickly and now MPI is taken for granted as vendor-supported software on any parallel machine.
- Free, portable implementations exist for clusters and other environments (MPICH, Open MPI)

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 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.

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.

How to Compile and run MPI Programs

MPI is a library, called from programs in ordinary programming languages such as C/C++ or Fortran. To compile such a program you use your regular compiler:

```
gcc -c my_mpi_prog.c -I/path/to/mpi.h
gcc -o my_mpi_prog my_mpi_prog.o -L/path/to/mpi -lmpich
```

However, MPI libraries may have different names between different architectures, making it hard to have a portable makefile. Therefore, MPI typically has shell scripts around your compiler call:

```
mpicc -c my_mpi_prog.c
mpicc -o my_mpi_prog my_mpi_prog.o
```

How to Compile and run MPI Programs

mpirun

mpirun is used to run mpi applications. It takes command line arguments that specify the number of processes to spawn, the set of machines on which to run the application processes (or you can specify a hostfile containing the machine names), and the command to run. For example:

```
To run a simple command:
------------------------
  mpirun -np 2 --host robin,loon    ./excecutable
```

Sending and Receiving Messages

- 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)

Input Parameters:
Buf : initial address of send buffer (choice)

Count: number of elements in send buffer (nonnegative integer)

Datatype : datatype of each send buffer element (handle)

Dest : rank of destination (integer)

Tag: message tag (integer)

Comm : communicator (handle)

Sending and Receiving Messages

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

Output Parameters:
Buf : initial address of receive buffer (choice)
Status : status object (Status)

Input Parameters :
Count : maximum number of elements in receive buffer (integer)
Datatype : datatype of each receive buffer element (handle)
Source: rank of source (integer)
Tag: message tag (integer)
Comm: communicator (handle)

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` operation.
- 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)
```

MPI Basic Send

MPI_Send(buf, count, datatype, dest, tag, comm)

buf: address of send buffer

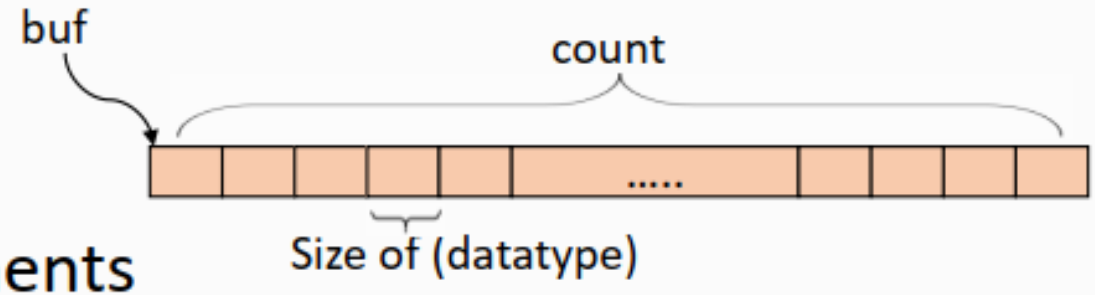
count: number of elements

datatype: data type of send buffer elements

dest: process id of destination process

tag: message tag (ignore for now)

comm: communicator (ignore for now)



MPI Basic Receive

MPI_Recv(buf, count, datatype, source, tag, comm, &status)

buf: address of receive buffer

count: size of receive buffer in elements

datatype: data type of receive buffer elements

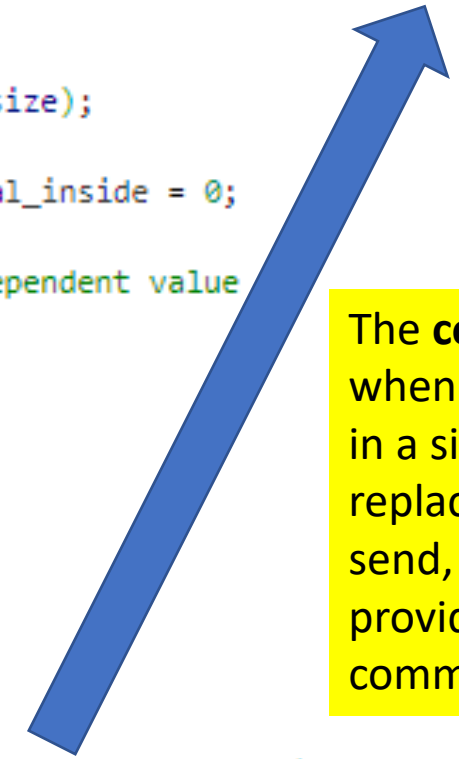
source: source process id or MPI_ANY_SOURCE

tag and comm: ignore for now

status: status object

```c
#include <mpi.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); MPI_Comm_size(MPI_COMM_WORLD, &size);

    long long num_points = 1000000; local_points = num_points / size; local_inside = 0;

    srand(time(NULL) + rank); // Seed random number generator with rank-dependent value

    for (long long i = 0; i < local_points; i++) {
        double x = (double)rand() / RAND_MAX;
        double y = (double)rand() / RAND_MAX;
        double distance = sqrt(x * x + y * y);

        if (distance <= 1.0) local_inside++;
    }

    long long global_inside;
    MPI_Reduce(&local_inside, &global_inside, 1, MPI_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        double pi_estimate = 4.0 * global_inside / num_points;
        printf("Estimated Pi: %f\n", pi_estimate);
    }

    MPI_Finalize();
    return 0;
}
```

```c
long long global_inside;
MPI_Reduce(&local_inside, &global_inside, 1, MPI_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
```



The **collective communications** are when all MPI tasks can be involved in a single function call. This replaces multiple calls to recv and send, easier to understand and provides internal optimizations to communication.

Write compact readable MPI C code to sum up all the numbers from 1 to 1000.

Use 1000 processes, where process 0 will print the sum. Note:

Explain all MPI functions after the code. Ensure that your code computes the correct sum.

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

int main(int argc, char ** argv){
   int mynode, totalnodes;
   int sum,startval,endval,accum;
   MPI_Status status;

   MPI_Init(argc,argv);
   MPI_Comm_size(MPI_COMM_WORLD, &totalprocs); // get totalprocs
   MPI_Comm_rank(MPI_COMM_WORLD, &myid); // get myid

   sum = myid + 1; // rank is an integer ranging from 0 to totalprocs-1

   if(myid != 0)
      MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
   else
      for(int j=1;j<totalprocs;j=j+1) {
         MPI_Recv(&accum,1,MPI_INT,j,1,MPI_COMM_WORLD, &status);
         sum = sum + accum;
      }

   if(myid == 0)
      cout << "The sum from 1 to 1000 is: " << sum << endl;
   MPI_Finalize();
}
```

Distributed System

- The logical view of a machine supporting the message-passing paradigm consists of *p* processes, each with its own exclusive address space.
- Instances of such a view come naturally from clustered workstations and non-shared address space multi-computers.
- Message-passing programming paradigm.

⇒ Two Key attributes

- Partitioned Address Space
  - No directly access to memory attached to each processor
- Explicit Parallelism
  - concurrent computations by means of primitives in the form of special-purpose directives or function calls.

Partitioned Address Space

- Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
  - Programming complexity as data need to be send and received across the network instead of read/write data to memory. Need faster networks and efficient network resources to move large amount of data.
  - Faster access to local data helps achieve high performance on non-UMA architecture.
  - Access to remote computations neeed.
- All interactions (read-only or read/write) require cooperation of two processes – the process that has the data and the process that wants to access the data.
  - Code complexity as data process does not know of events at requesting processes.
  - Programmer has knowledge of all interactions and can think about algorithms (and mappings) that minimize interactions.
  - Scalable to very large number of processors 100K+ and 200K+ cores.

Message-passing programming paradigm

- Parallelism is coded explicitly by the programmer.
  - Programmer identifies ways to decompose computations and extract concurrency.
  - Message-passing paradigm tends to be hard and intellectually demanding.
  - Potential of achieving very high performance by scaling to a very large number of processes on large infrastructures.

- Asynchronous or loosely synchronous programming paradigms.
  - In the asynchronous paradigm, all concurrent tasks execute asynchronously.
  - In Loosely synchronous programs tasks execute completely asynchronously but synchronize to perform interactions.

Message-passing programming paradigm

- Message-passing programs are written using the ***single program multiple data*** (SPMD) approach.
  - SPMD code executed by different processes is identical except for a small number of processes.
  - SPMD programs can be loosely synchronous or completely asynchronous.

- Processes interact by sending and receiving messages. Two primitives are available:
  - send(void *sendbuf, int nelems, int dest)
  - receive(void *recvbuf, int nelems, int source)

- Send and Receive Operations
- Undesirable side-effects of Communication hardware
  - DMA and Asynchronous Communication
- Blocking Message Passing Operations
  - Buffered and Non-Buffered
    - Operations Details
    - Idling and Deadlock
    - Impact of Finite Buffered
- Non-Blocking Message Passing Operations
  - Buffered non-blocking operations
  - Non-buffered
- MPI Offered both Buffered and Non-Buffered Operations

6.2 The Building Blocks: Send and Receive Operations

IMPORTANT: This is theory from chapter # 6 of the textbook. Please use MPI_Send MPI_Receive syntax in your programs

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

Don't use this syntax in your MPI programs // ASL.

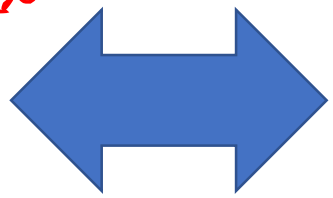
The `sendbuf` points to a buffer that stores the data to be sent, `recvbuf` points to a buffer that stores the data to be received, `nelems` is the number of data units to be sent and received, `dest` is the identifier of the process that receives the data, and `source` is the identifier of the process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

6.2 The Building Blocks: Send and Receive Operations

```
P0

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

# of elements

destination



```
P1

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

# of elements

Source

– when value of a at P1 = 100 ?

– when value of a at P1 = 0 ?

(see next slide)

In this simple example, process P0 sends a message to process P1 which receives and prints the message. The important thing to note is that process P0 changes the value of `a` to 0 immediately following the `send` . The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0. That is, the value of `a` at the time of the send operation must be the value that is received by process `P1` .

6.2 The Building Blocks: Send and Receive Operations

Contd.

```
P0

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

```
P1

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

It may seem that it is quite straightforward to ensure the semantics of the send and receive operations. However, based on how the send and receive operations are implemented this may not be the case. Most message passing platforms have additional hardware support for sending and receiving messages. They may support DMA (direct memory access) and asynchronous message transfer using network interface hardware. Network interfaces allow the transfer of messages from buffer memory to desired location without CPU intervention. Similarly, DMA allows copying of data from one memory location to another (e.g., 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!

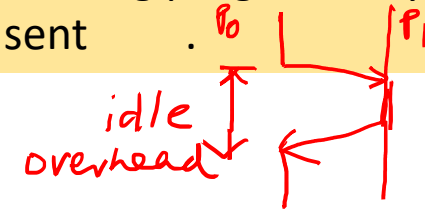
6.2.1 Blocking Message Passing Operations

A simple solution to the dilemma presented in the code fragment above is for the send operation to return only when it is semantically safe to do so. Note that this is not the same as saying that the send operation returns only after the receiver has received the data. It simply means that the 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.

Buffer = Communication Buffer (Hardware communication OR Driver) (Data from program variable copied somewhere else)

Blocking Non-Buffered Send/Receive

Control is returned to the calling program only when receiver has completely received the sent .



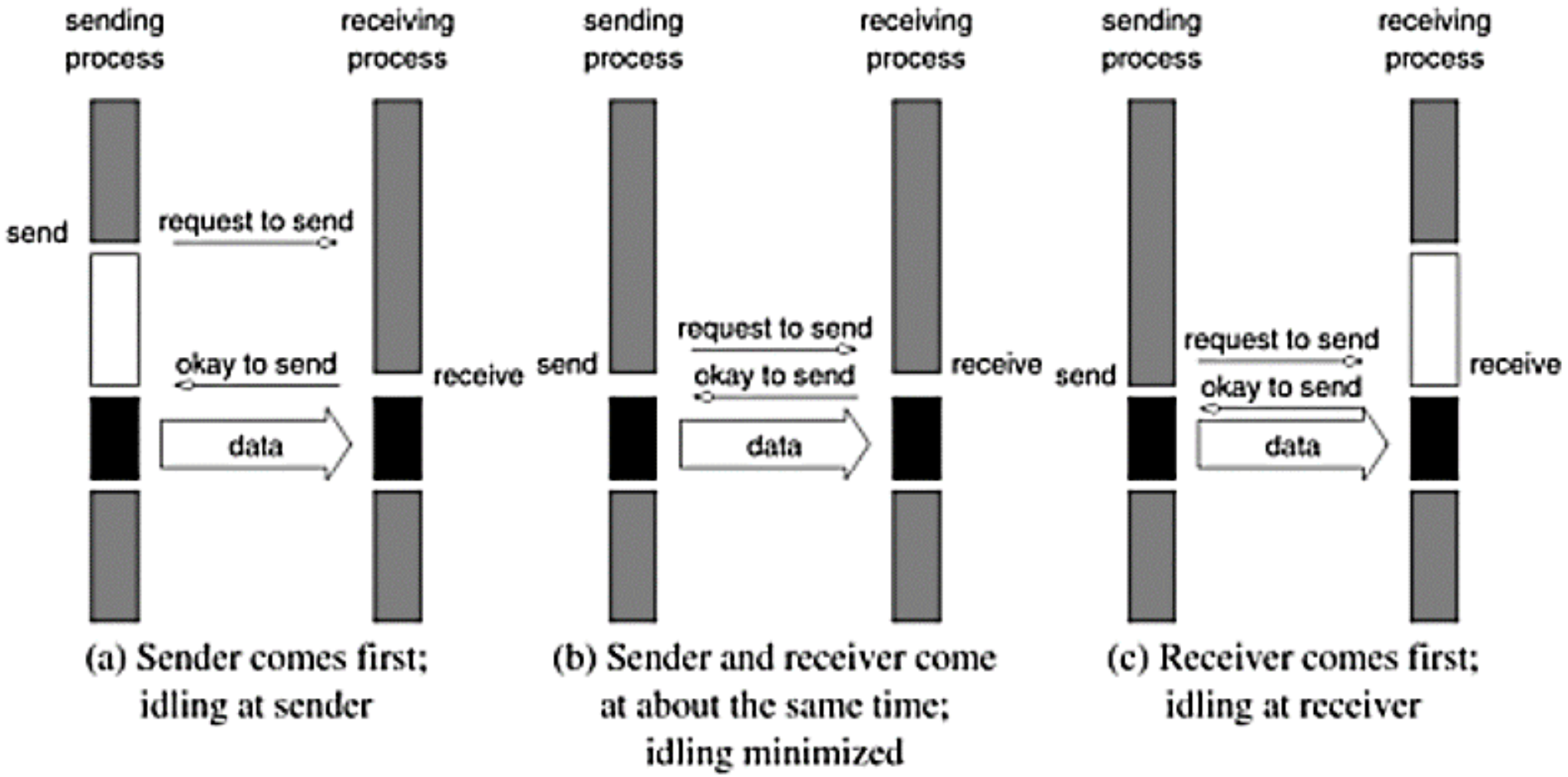
Blocking Buffered Send/Receive

Control is returned to the calling program after copying the data into the buffer. Transfer of data to the sender will be done asynchronously.

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. Typically, this process 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. The operation is illustrated in Figure 6.1 . Since there are no buffers used at either sending or receiving ends, this is also referred to as a ***non-buffered blocking operation*** .

Blocking Non-Buffered Send/Receive



(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

Figure 6.1.

Idling Overheads in Blocking Non-Buffered Operations

also clear from the figures that 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.

Deadlocks in Blocking Non-Buffered Operations

```
P0

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

```
P1

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

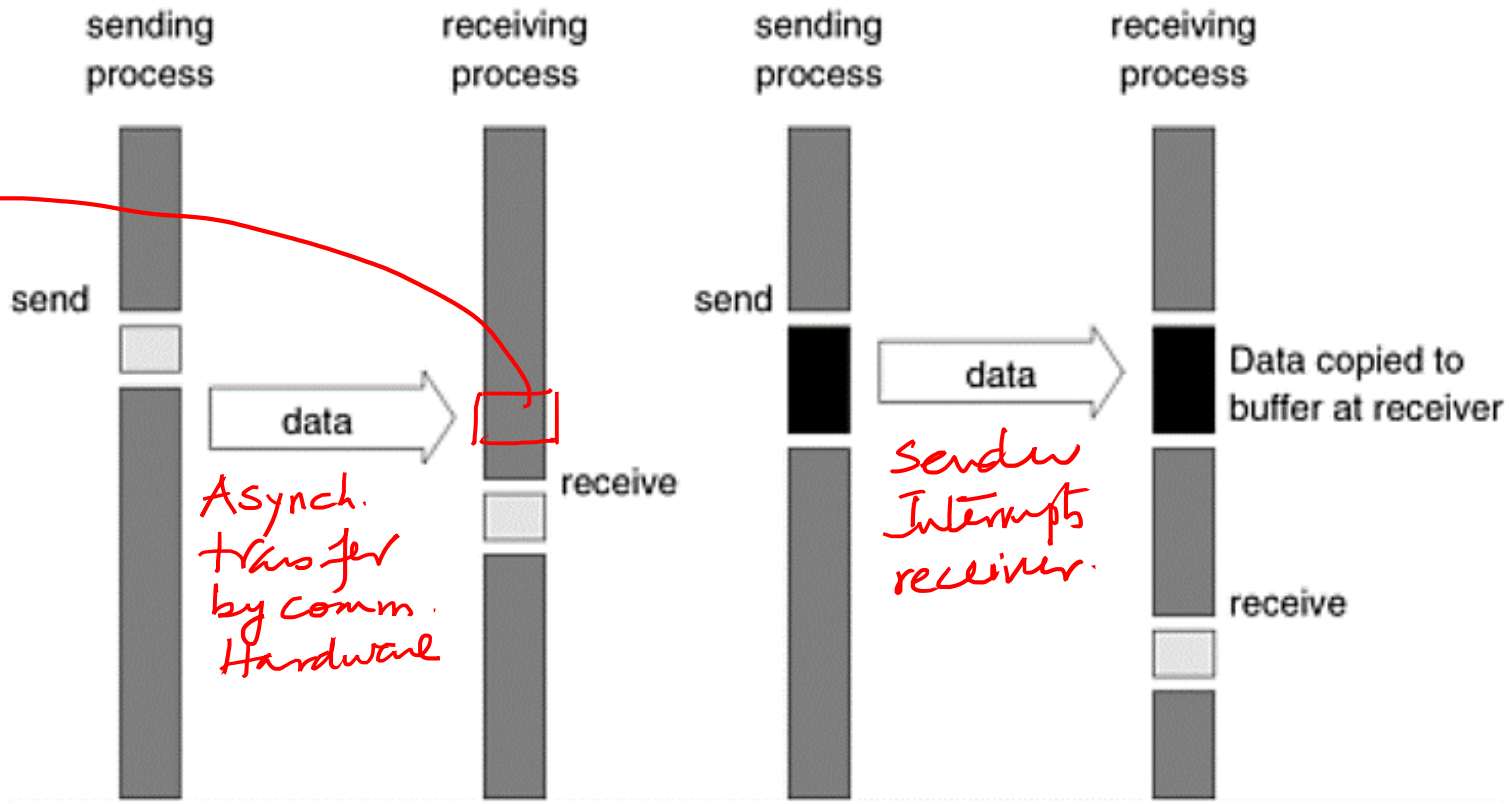
The code fragment makes the values of `a` available to both processes P0 and P1. However, if the send and receive operations are implemented using a blocking non-buffered protocol, the send at P0 waits for the matching receive at P1 whereas the send at process P1 waits for the corresponding receive at P0, resulting in an infinite wait.

As can be inferred, deadlocks are very easy in blocking protocols and care must be taken to break cyclic waits of the nature outlined. In the above example, this can be corrected by replacing the operation sequence of one of the processes by a `receive` and a `send` as opposed to the other way around. This often makes the code more cumbersome and buggy.

Blocking Buffered Send/Receive

A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends. We start with a simple case in which the sender has a buffer pre-allocated for communicating messages. On encountering a send operation, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The sender process can now continue with the program knowing that any changes to the data will not impact program semantics. The actual communication can be accomplished in many ways depending on the available hardware resources. If the hardware supports asynchronous communication (independent of the CPU), then a network transfer can be initiated after the message has been copied into the buffer. Note that 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



saved in communication Buffer at the receiver

Asynch. transfer by comm. Hardware

sender Interrupts receiver.

Figure 6.2.

(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. When the receiver eventually encounters a receive operation, the message is copied from the buffer into the target location.

Example 6.1 Impact of finite buffers in message passing

Consider the following code fragment:

- if P1 start late the buffer should be large enough to accomodate producer's data
- safe solution is to have a bounded buffer implementation //

```
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          }                                     }
```

In this code fragment, process P0 produces 1000 data items and process P1 consumes them. However, if process P1 was slow getting to this loop, process P0 might have sent all of its data. If there is enough buffer space, then both processes can proceed; however, if the buffer is not sufficient (i.e., buffer overflow), the sender would have 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. In general, it is a good idea to write programs that have bounded buffer requirements. ■

Deadlocks in Buffered Send and Receive Operations While buffering alleviates many of the deadlock situations, it is still possible to write code that deadlocks. This is due to the fact that as in the non-buffered case, receive calls are always blocking (to ensure semantic consistency). Thus, a simple code fragment such as the following deadlocks since both processes wait to receive data but nobody sends it.

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

Source

Source

Once again, such circular waits have to be broken. However, deadlocks are caused only by waits on receive operations in this case.

check status and wait if background operation not completed.

6.2.2 Non-Blocking Message Passing Operations

- Non-blocking protocols returns from the send or receive operation before it is semantically safe to do so. Therefore, the user must be careful not to alter data that may be potentially participating in a communication operation.

  - Non-blocking operations are generally accompanied by a check-status operation, which indicates whether the semantics of a previously initiated transfer may be violated or not.
  - Upon return from 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.

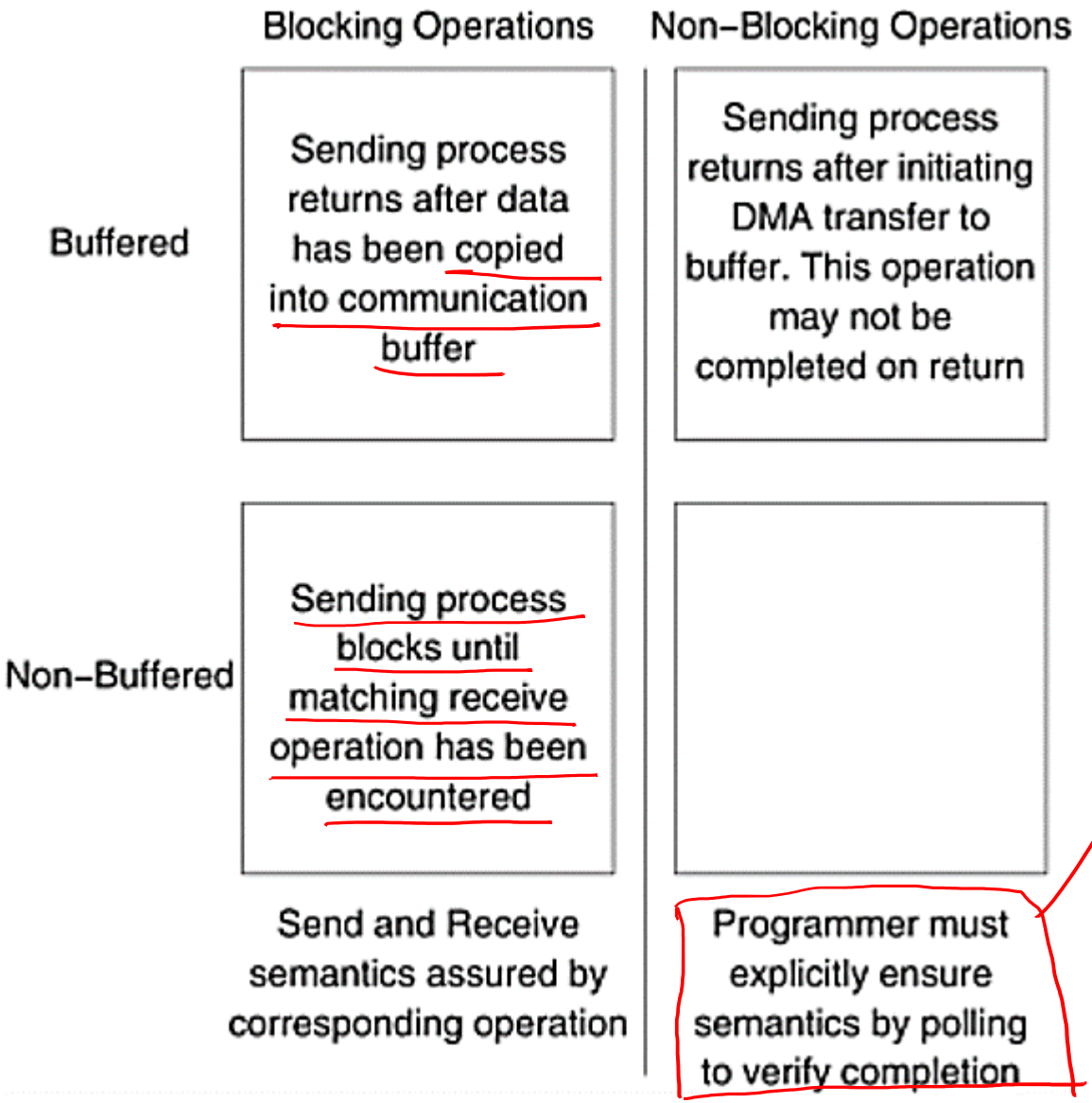

Blocking Operations
Non–Blocking Operations
Buffered
Sending process returns after data has been copied into communication buffer
Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non–Buffered
Sending process blocks until matching receive operation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

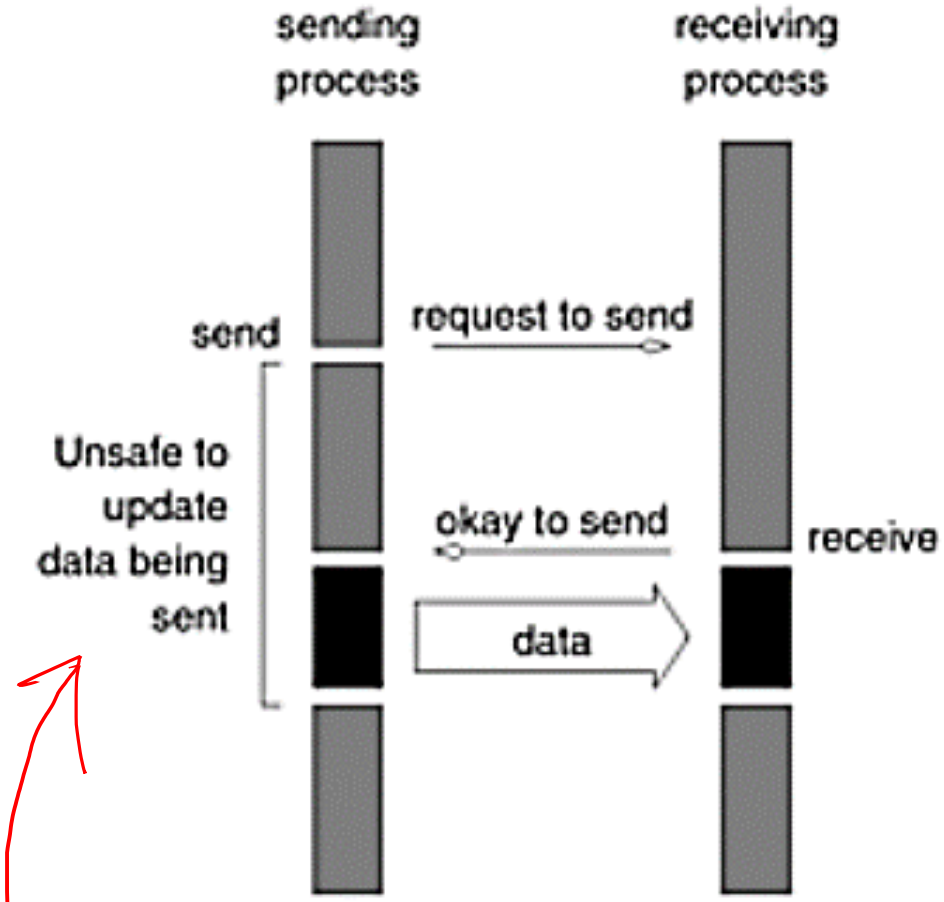
Figure 6.3.

Buffered or Non-buffered non-blocking operations

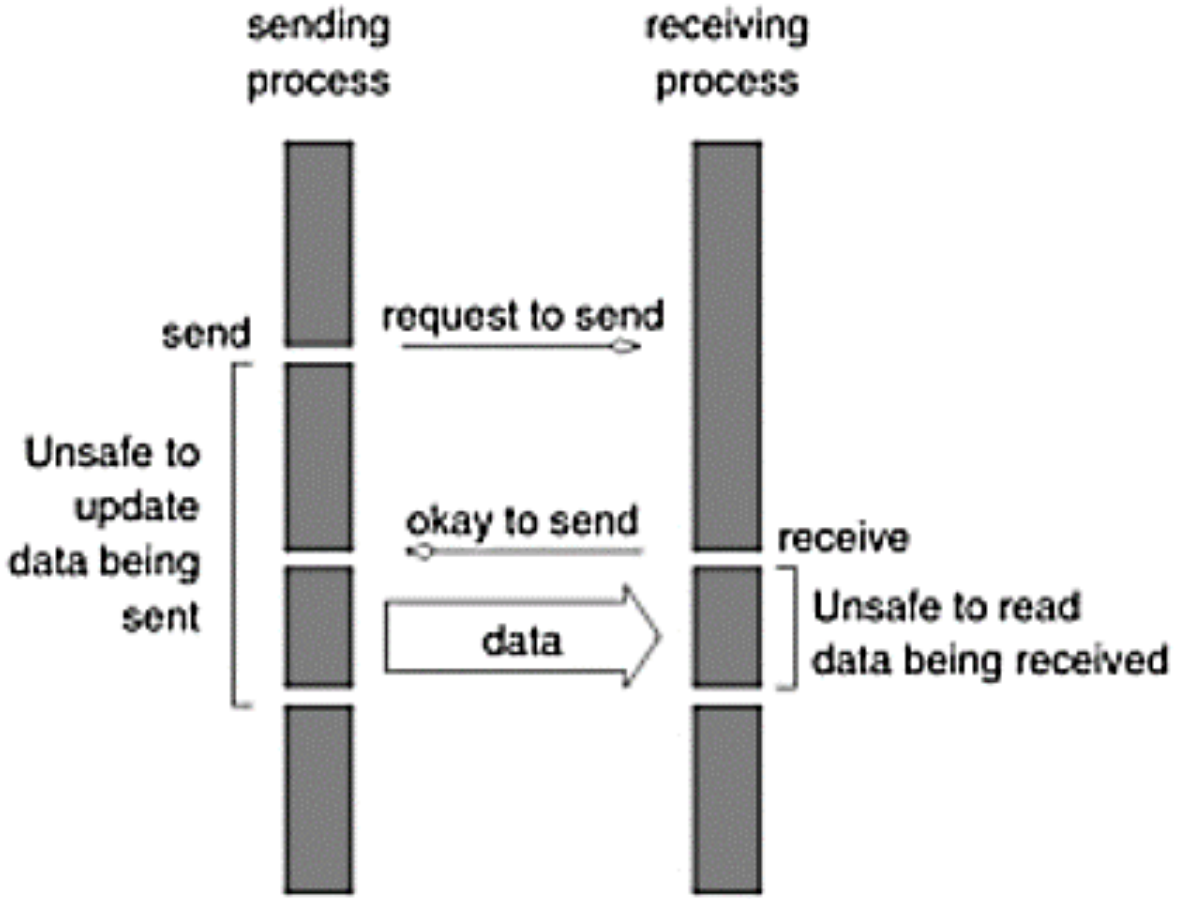
- In the **buffered case**, the sender *initiates a DMA operation and returns immediately.*
- The data becomes safe the moment the DMA operation has been completed.
- At the receiving end, the receive operation initiates a transfer from the sender's buffer to the receiver's target location.
- Using buffers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

- In the **non-buffered case**, a process wishing to send data to another *simply posts a pending message and returns to the user program.*
- 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.

Non-blocking non-buffered send and receive operations



(a) Without hardware support

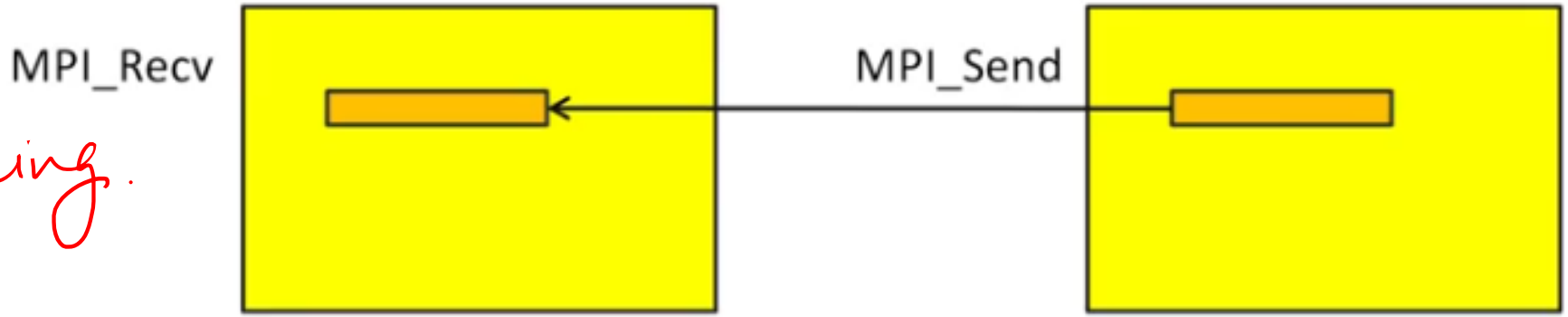


(b) With hardware support (without Buffers case only.

Using Blocking and Non-Blocking Operations

- Typical message-passing libraries such as Message Passing Interface (MPI) 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.
- One must, however, be careful using non-blocking protocols since errors can result from unsafe access to data that is in the process of being communicated

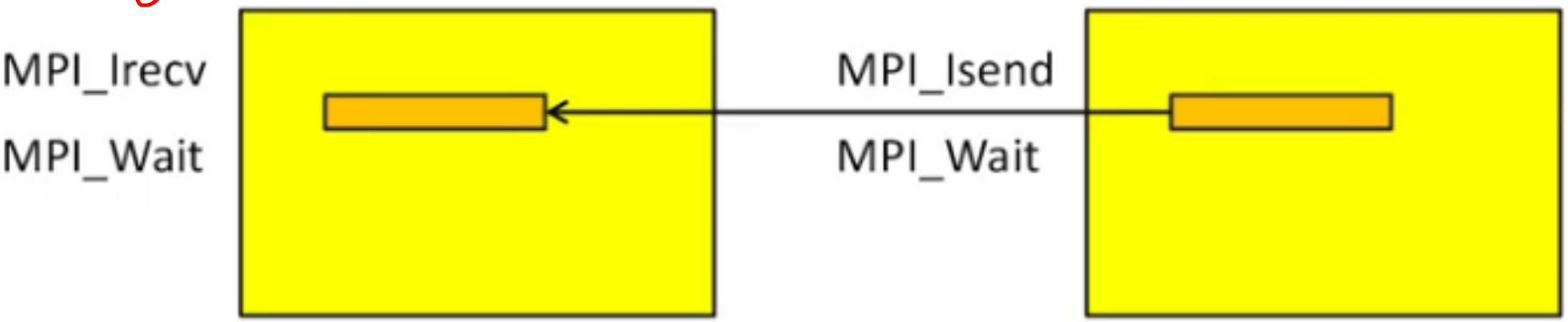
Basic MPI Communication



Blocking.

OR

Non-blocking.



MPI_Test

6.3.4 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)
```

Avoiding Deadlocks The semantics of `MPI_Send` and `MPI_Recv` place some restrictions on how we can mix and match send and receive operations. For example, consider the following piece of code in which process 0 sends two messages with different tags to process 1, and process 1 receives them in the reverse order.

```
1    int a[10], b[10], myrank;
2    MPI_Status status;
3    ...
4     MPI_Comm_rank(MPI_COMM_WORLD,  &myrank);
5    if (myrank == 0) {
6      MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
7      MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
8    }
9    else if (myrank == 1) {
10     MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
11     MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
12   }
13   ...
```

The above example can be made safe, by rewriting it 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   ...
```

This new implementation partitions the processes into two groups. One consists of the odd-numbered processes and the other of the even-numbered processes. The odd-numbered processes perform a send followed by a receive, and the even-numbered processes perform a receive followed by a send. Thus, when an odd-numbered process calls `MPI_Send` ,the target process (which has an even number) will call `MPI_Recv` to receive that message, before attempting to send its own message.

Sending and Receiving Messages Simultaneously The above communication pattern appears frequently in many message-passing programs, and for this reason MPI provides the `MPI_Sendrecv` function that both sends and receives a message.

`MPI_Sendrecv` does not suffer from the circular deadlock problems of `MPI_Send` and `MPI_Recv`. You can think of `MPI_Sendrecv` as allowing data to travel for both send and receive simultaneously. The calling sequence of `MPI_Sendrecv` is the following:

```
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)
```

The arguments of `MPI_Sendrecv` are essentially the combination of the arguments of `MPI_Send` and `MPI_Recv`. The send and receive buffers must be disjoint, and the source and destination of the messages can be the same or different. The safe version of our earlier example using `MPI_Sendrecv` is 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    MPI_SendRecv(a, 10, MPI_INT, (myrank+1)%npes, 1,
7                 b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
8                 MPI_COMM_WORLD, &status);
9    ...
```

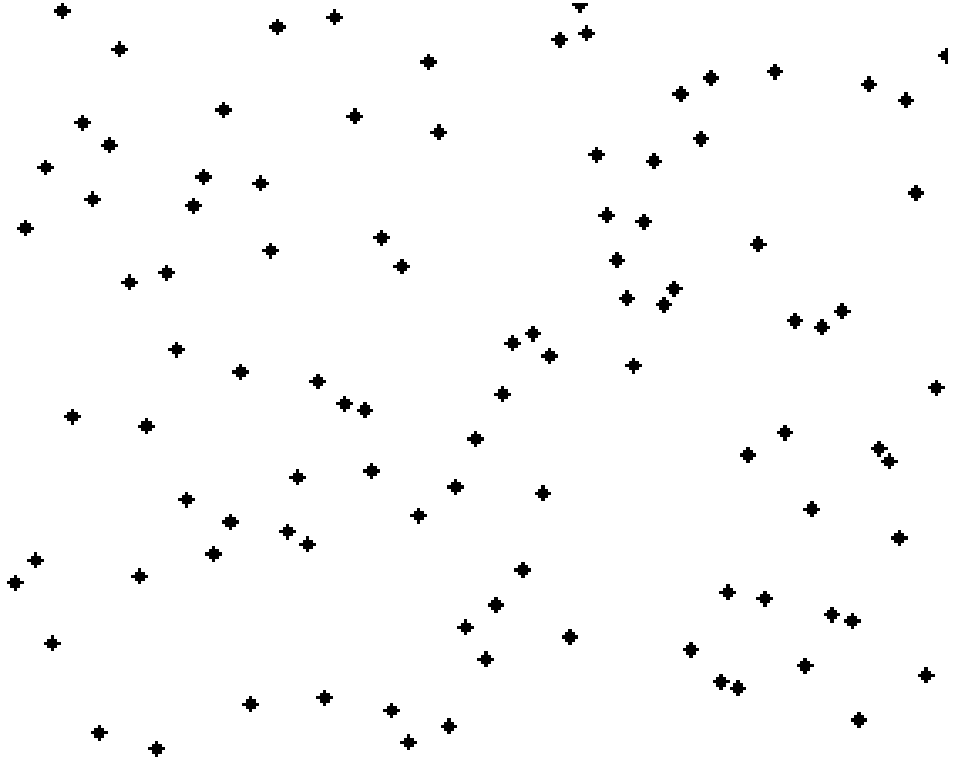
6.3.5 Example: Odd-Even Sort

https://en.wikipedia.org/wiki/Odd–even_sort

- Odd-Even sorting algorithm sorts a sequence of *n* elements using *p* processes in a total of *p* phases.
- In each phase, each process performs a compare-split step with its right neighbor.
- The MPI program for performing the odd-even sort in parallel is shown below.
- To simplify the presentation, this program assumes that *n* is divisible by *p* .

In computing, an **odd–even sort** or **odd–even transposition sort** (also known as **brick sort** or **parity sort**) is a relatively simple sorting algorithm, developed originally for use on parallel processors with local interconnections.

It functions by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. The next step repeats this for even/odd indexed pairs (of adjacent elements). Then it alternates between odd/even and even/odd steps until the list is sorted.

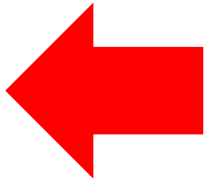


```c
#include <stdlib.h>
#include <mpi.h> /* Include MPI's header file */

int main(int argc, char *argv[]) {
   int n;        /* The total number of elements to be sorted */
   int npes;     /* The total number of processes */
   int myrank;   /* The rank of the calling process */
   int nlocal;   /* The local number of elements, and the array that stores them */
   int *elmnts;  /* The array that stores the local elements */
   int *relmnts; /* The array that stores the received elements */
   int oddrank;  /* The rank of the process during odd-phase communication */
   int evenrank; /* The rank of the process during even-phase communication */
   int *wspace;  /* Working space during the compare-split operation */
   int i;
   MPI_Status status;

   /* Initialize MPI and get system information */
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &npes);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

   n = atoi(argv[1]);
   nlocal = n / npes; /* Compute the number of elements to be stored locally. */
```

```
25    /* Allocate memory for the various arrays */
26    elmnts = (int *)malloc(nlocal * sizeof(int));
27    relmnts = (int *)malloc(nlocal * sizeof(int));
28    wspace = (int *)malloc(nlocal * sizeof(int));
29
30    /* Fill-in the elmnts array with random elements */
31    srandom(myrank);
32    for (i = 0; i < nlocal; i++)
33       elmnts[i] = random();
34
35    /* Sort the local elements using the built-in quicksort routine */
36    qsort(elmnts, nlocal, sizeof(int), IncOrder);
```

```
 99   /* The IncOrder function that is called by qsort is defined as follows */
100   int IncOrder(const void *e1, const void *e2) {
101      return (*((int *)e1) - *((int *)e2));
102   }
```

```
38      /* Determine the rank of the processors that myrank needs to communicate during */
39      /* the odd and even phases of the algorithm */
40      if (myrank % 2 == 0)
41      {
42          oddrank = myrank - 1;
43          evenrank = myrank + 1;
44      }
45      else
46      {
47          oddrank = myrank + 1;
48          evenrank = myrank - 1;
49      }
50
51      /* Set the ranks of the processors at the end of the linear */
52      if (oddrank == -1 || oddrank == npes)
53          oddrank = MPI_PROC_NULL;
54      if (evenrank == -1 || evenrank == npes)
55          evenrank = MPI_PROC_NULL;
```



```
57      /* Get into the main loop of the odd-even sorting algorithm */
58      for (i = 0; i < npes - 1; i++) {
59        if (i % 2 == 1) /* Odd phase */
60          MPI_Sendrecv( elmnts, nlocal, MPI_INT, oddrank, 1,
61                        relmnts, nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
62        else /* Even phase */
63          MPI_Sendrecv( elmnts, nlocal, MPI_INT, evenrank, 1,
64                        relmnts, nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);
65
66        CompareSplit(nlocal, elmnts, relmnts, wspace, myrank < status.MPI_SOURCE);
67      }
68
69      free(elmnts); free(relmnts); free(wspace);
70      MPI_Finalize();
71  }
```

```
74  CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace, int keepsmall) {
75     int i, j, k;
76
77     for (i = 0; i < nlocal; i++)
78        wspace[i] = elmnts[i]; /* Copy the elmnts array into the wspace array */
79
80     if (keepsmall) { /* Keep the nlocal smaller elements */
81        for (i = j = k = 0; k < nlocal; k++) {
82           if (j == nlocal || (i < nlocal && wspace[i] < relmnts[j]))
83              elmnts[k] = wspace[i++];
84           else
85              elmnts[k] = relmnts[j++];
86        }
87     }
88     else { /* Keep the nlocal larger elements */
89        for (i = k = nlocal - 1, j = nlocal - 1; k >= 0; k--) {
90           if (j == 0 || (i >= 0 && wspace[i] >= relmnts[j]))
91              elmnts[k] = wspace[i--];
92           else
93              elmnts[k] = relmnts[j--];
94        }
95     }
96  }
```

6.5 Overlapping Communication with Computation

The MPI programs we developed so far used blocking send and receive operations whenever they needed to perform point-to-point communication. Recall that a blocking send operation remains blocked until the message has been copied out of the send buffer (either into a system buffer at the source process or sent to the destination process). Similarly, a blocking receive operation returns only after the message has been received and copied into the receive buffer. For example, consider Cannon's matrix-matrix multiplication program described in Program 6.2. During each iteration of its main computational loop (lines 47– 57), it first computes the matrix multiplication of the sub-matrices stored in `a` and `b`, and then shifts the blocks of `a` and `b`, using `MPI_Sendrecv_replace` which blocks until the specified matrix block has been sent and received by the corresponding processes. In each iteration, each process spends $O(n^3/p^{1.5})$ time for performing the matrix-matrix multiplication and $O(n^2/p)$ time for shifting the blocks of matrices A and B. Now, since the blocks of matrices A and B do not change as they are shifted among the processors, it will be preferable if we can overlap the transmission of these blocks with the computation for the matrix-matrix multiplication, as many recent distributed-memory parallel computers have dedicated communication controllers that can perform the transmission of messages without interrupting the CPUs.

6.5.1 Non-Blocking Communication Operations

In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations. These functions are `MPI_Isend` and `MPI_Irecv`. `MPI_Isend` starts a send operation but does not complete, that is, it returns before the data is copied out of the buffer. Similarly, `MPI_Irecv` starts a receive operation but returns before the data has been received and copied into the buffer. With the support of appropriate hardware, the transmission and reception of messages can proceed concurrently with the computations performed by the program upon the return of the above functions.

However, at a later point in the program, a process that has started a non-blocking send or receive operation must make sure that this operation has completed before it proceeds with its computations. This is because a process that has started a non-blocking send operation may want to overwrite the buffer that stores the data that are being sent, or a process that has started a non-blocking receive operation may want to use the data it requested. To check the completion of non-blocking send and receive operations, MPI provides a pair of functions `MPI_Test` and `MPI_Wait`. The first tests whether or not a non-blocking operation has finished and the second waits (i.e., gets blocked) until a non-blocking operation actually finishes.

The calling sequences of `MPI_Isend` and `MPI_Irecv` are the following:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm, MPI_Request *request)
```

Note that these functions have similar arguments as the corresponding blocking send and receive functions. The main difference is that they take an additional argument `request`. `MPI_Isend` and `MPI_Irecv` functions allocate a *request object* and return a pointer to it in the `request` variable. This request object is used as an argument in the `MPI_Test` and `MPI_Wait` functions to identify the operation whose status we want to query or to wait for its completion.

Note that the `MPI_Irecv` function does not take a `status` argument similar to the blocking receive function, but the status information associated with the receive operation is returned by the `MPI_Test` and `MPI_Wait` functions.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

`MPI_Test` tests whether or not the non-blocking send or receive operation identified by its `request` has finished. It returns `flag = {true}` (non-zero value in C) if it completed, otherwise it returns `{false}` (a zero value in C). In the case that the non-blocking operation has finished, the request object pointed to by `request` is deallocated and `request` is set to `MPI_REQUEST_NULL`. Also the `status` object is set to contain information about the operation. If the operation has not finished, `request` is not modified and the value of the `status` object is undefined. The `MPI_Wait` function blocks until the non-blocking operation identified by `request` completes. In that case it deal-locates the `request` object, sets it to `MPI_REQUEST_NULL`, and returns information about the completed operation in the `status` object.

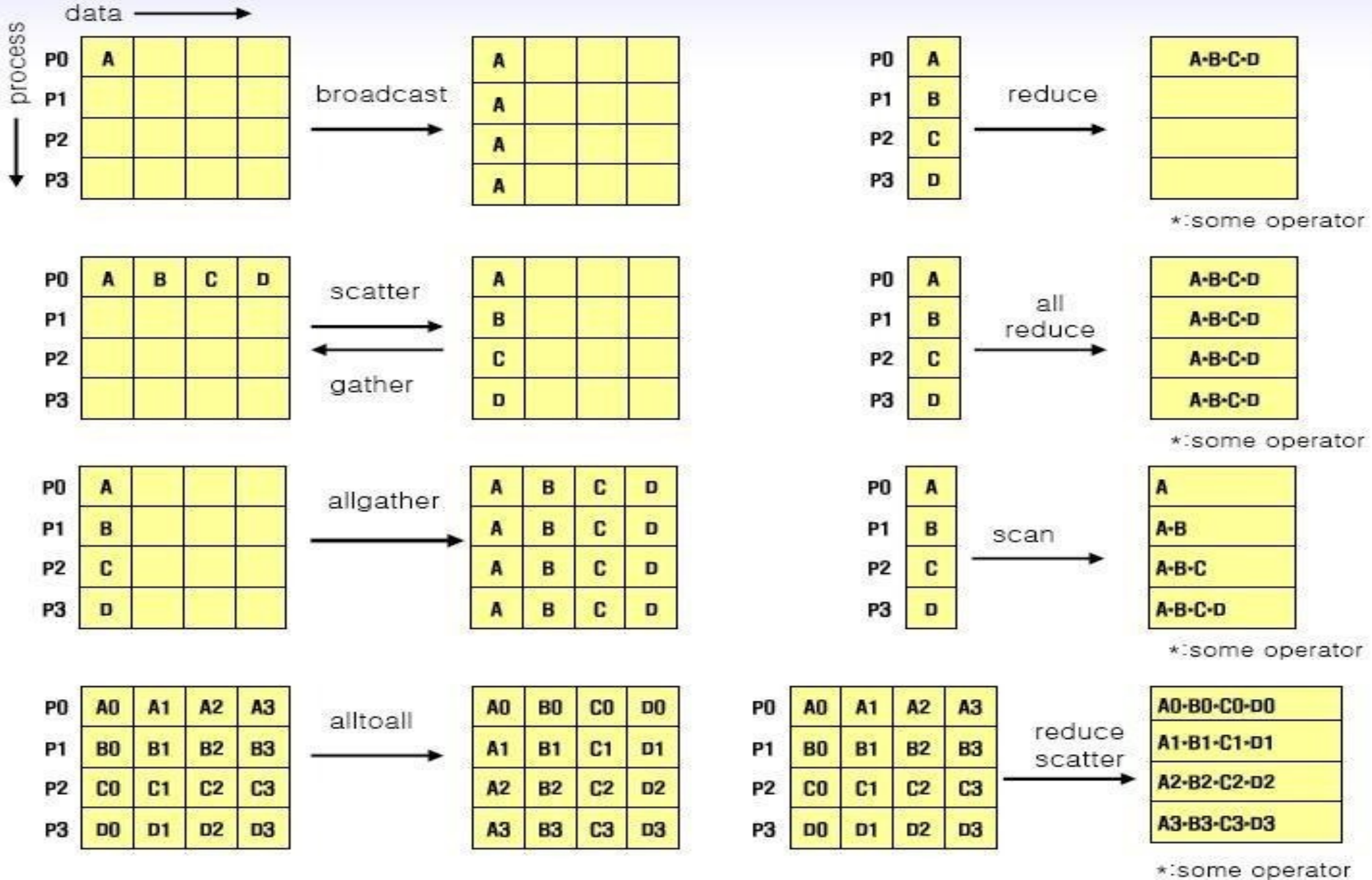
Avoiding Deadlocks By using non-blocking communication operations we can remove most of the deadlocks associated with their blocking counterparts. For example, as we discussed in Section 6.3 the following piece of code is not safe.

```
 1    int a[10], b[10], myrank;
 2    MPI_Status status;
 3    MPI_Request requests[2];
 4    ...
 5     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 6    if (myrank == 0) {
 7      MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
 8      MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
 9    }
10    else if (myrank == 1) {
11      MPI_Irecv(b, 10, MPI_INT, 0, 2, &requests[0], MPI_COMM_WORLD);
12      MPI_Irecv(a, 10, MPI_INT, 0, 1, &requests[1], MPI_COMM_WORLD);
13    }
14    ...
```

This example also illustrates that the non-blocking operations started by any process can finish in any order depending on the transmission or reception of the corresponding messages. For example, the second receive operation will finish before the first does.

6.6 Collective Communication and Computation Operations

MPI provides an extensive set of functions for performing many commonly used collective communication operations. In particular, the majority of the basic communication operations described in Chapter 4 are supported by MPI. All of the collective communication functions provided by MPI take as an argument a communicator that defines the group of processes that participate in the collective operation. All the processes that belong to this communicator participate in the operation, and all of them must call the collective communication function. Even though collective communication operations do not act like barriers (i.e., it is possible for a processor to go past its call for the collective communication operation even before other processes have reached it), it acts like a *virtual* synchronization step in the following sense: the parallel program should be written such that it behaves correctly even if a global synchronization is performed before and after the collective call. Since the operations are virtually synchronous, they do not require tags. In some of the collective functions data is required to be sent from a single process (source-process) or to be received by a single process (target-process). In these functions, the source- or target-process is one of the arguments supplied to the routines. All the processes in the group (i.e., communicator) must specify the same source- or target-process. For most collective communication operations, MPI provides two different variants. The first transfers equal-size data to or from each process, and the second transfers data that can be of different sizes.


data
process
P0
P1
P2
P3
A
broadcast
A
A
A
A
reduce
A·B·C·D
*:some operator
scatter
gather
B
C
D
all reduce
allgather
scan
A·B
A·B·C
alltoall
A0
A1
A2
A3
B0
B1
B2
B3
C0
C1
C2
C3
D0
D1
D2
D3
reduce scatter
A0·B0·C0·D0
A1·B1·C1·D1
A2·B2·C2·D2
A3·B3·C3·D3

6.6.1 Barrier

The barrier synchronization operation is performed in MPI using the `MPI_Barrier` function.

```
int MPI_Barrier(MPI_Comm comm)
```

The only argument of `MPI_Barrier` is the communicator that defines the group of processes that are synchronized. The call to `MPI_Barrier` returns only after all the processes in the group have called this function.

6.6.2 Broadcast

The one-to-all broadcast operation described in Section 4.1 is performed in MPI using the `MPI_Bcast` function.

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
        int source, MPI_Comm comm)
```

`MPI_Bcast` sends the data stored in the buffer `buf` of process `source` to all the other processes in the group. The data received by each process is stored in the buffer `buf` . The data that is broadcast consist of `count` entries of type `datatype` . The amount of data sent by the `source` process must be equal to the amount of data that is being received by each process; i.e., the `count` and `datatype` fields must match on all processes.

6.6.3 Reduction

The all-to-one reduction operation described in Section 4.1 is performed in MPI using the `MPI_Reduce` function.

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

`MPI_Reduce` combines the elements stored in the buffer `sendbuf` of each process in the group, using the operation specified in `op` , and returns the combined values in the buffer `recvbuf` of the process with rank `target` . Both the `sendbuf` and `recvbuf` must have the same number of `count` items of type `datatype` . Note that all processes must provide a `recvbuf` array, even if they are not the *target* of the reduction operation. When `count` is more than one, then the combine operation is applied element-wise on each entry of the sequence. All the processes must call `MPI_Reduce` with the same value for `count` , `datatype` , `op` , `target` , and `comm` .

MPI provides a list of predefined operations that can be used to combine the elements stored in `sendbuf` . MPI also allows programmers to define their own operations, which is not covered in this book. The predefined operations are shown in Table 6.3 . For example, in order to compute the maximum of the elements stored in `sendbuf` , the `MPI_MAX` value must be used for the `op` argument. Not all of these operations can be applied to all possible data-types supported by MPI. For example, a bit-wise OR operation (i.e., *op* = `MPI_BOR`) is not defined for real-valued data-types such as `MPI_FLOAT` and `MPI_REAL` . The last column of Table 6.3 shows the various data-types that can be used with each operation.

`MPI_MAX`	Maximum	C integers and floating point	
`MPI_MIN`	Minimum	C integers and floating point	
`MPI_SUM`	Sum	C integers and floating point	
`MPI_PROD`	Product	C integers and floating point	
`MPI_LAND`	Logical AND	C integers	
`MPI_BAND`	Bit-wise AND	C integers and byte	
`MPI_LOR`	Logical OR	C integers	
`MPI_BOR`	Bit-wise OR	C integers and byte	
`MPI_LXOR`	Logical XOR	C integers	
`MPI_BXOR`	Bit-wise XOR	C integers and byte	
`MPI_MAXLOC`	max-min value-location	Data-pairs	
`MPI_MINLOC`	min-min value-location	Data-pairs	

6.6.4 Prefix

The prefix-sum operation described in Section 4.3 is performed in MPI using the `MPI_Scan` function.

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

`MPI_Scan` performs a prefix reduction of the data stored in the buffer `sendbuf` at each process and returns the result in the buffer `recvbuf` . The receive buffer of the process with rank *i* will store, at the end of the operation, the reduction of the send buffers of the processes whose ranks range from 0 up to and including *i* . The type of supported operations (i.e., `op`) as well as the restrictions on the various arguments of `MPI_Scan` are the same as those for the reduction operation `MPI_Reduce` .

6.6.5 Gather

The gather operation described in Section 4.4 is performed in MPI using the `MPI_Gather` function.

```
int MPI_Gather(void *sendbuf, int sendcount,
          MPI_Datatype senddatatype, void *recvbuf, int recvcount,
          MPI_Datatype recvdatatype, int target, MPI_Comm comm)
```

Each process, including the `target` process, sends the data stored in the array `sendbuf` to the `target` process. As a result, if *p* is the number of processors in the communication `comm` , the target process receives a total of *p* buffers. The data is stored in the array `recvbuf` of the target process, in a rank order. That is, the data from process with rank *i* are stored in the `recvbuf` starting at location *i* * `sendcount` (assuming that the array `recvbuf` is of the same type as `recvdatatype`).

The data sent by each process must be of the same size and type. That is, `MPI_Gather` must be called with the `sendcount` and `senddatatype` arguments having the same values at each process. The information about the receive buffer, its length and type applies only for the target process and is ignored for all the other processes. The argument `recvcount` specifies the number of elements received by each process and not the total number of elements it receives. So, `recvcount` must be the same as `sendcount` and their datatypes must be matching.

6.6.5 Gather

MPI also provides the `MPI_Allgather` function in which the data are gathered to all the processes and not only at the target process.

```
int MPI_Allgather(void *sendbuf, int sendcount,
          MPI_Datatype senddatatype, void *recvbuf, int recvcount,
          MPI_Datatype recvdatatype, MPI_Comm comm)
```

The meanings of the various parameters are similar to those for `MPI_Gather` ; however, each process must now supply a `recvbuf` array that will store the gathered data.

6.6.6 Scatter

The scatter operation described in Section 4.4 is performed in MPI using the `MPI_Scatter` function.

```
int MPI_Scatter(void *sendbuf, int sendcount,
          MPI_Datatype senddatatype, void *recvbuf, int recvcount,
          MPI_Datatype recvdatatype, int source, MPI_Comm comm)
```

The `source` process sends a different part of the send buffer `sendbuf` to each processes, including itself. The data that are received are stored in `recvbuf` . Process *i* receives `sendcount` contiguous elements of type `senddatatype` starting from the *i* * `sendcount` location of the `sendbuf` of the source process (assuming that `sendbuf` is of the same type as `senddatatype`). `MPI_Scatter` must be called by all the processes with the same values for the `sendcount` , `senddatatype` , `recvcount` , `recvdatatype` , `source` , and `comm` arguments. Note again that `sendcount` is the number of elements sent to each individual process.

6.6.7 All-to-All

The all-to-all personalized communication operation described in Section 4.5 is performed in MPI by using the `MPI_Alltoall` function.

```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

Each process sends a different portion of the `sendbuf` array to each other process, including itself. Each process sends to process *i* `sendcount` contiguous elements of type `senddatatype` starting from the *i* * `sendcount` location of its `sendbuf` array. The data that are received are stored in the `recvbuf` array. Each process receives from process *i* `recvcount` elements of type `recvdatatype` and stores them in its `recvbuf` array starting at location *i* * `recvcount` . `MPI_Alltoall` must be called by all the processes with the same values for the `sendcount` , `senddatatype` , `recvcount` , `recvdatatype` , and `comm` arguments. Note that `sendcount` and `recvcount` are the number of elements sent to, and received from, each individual process.