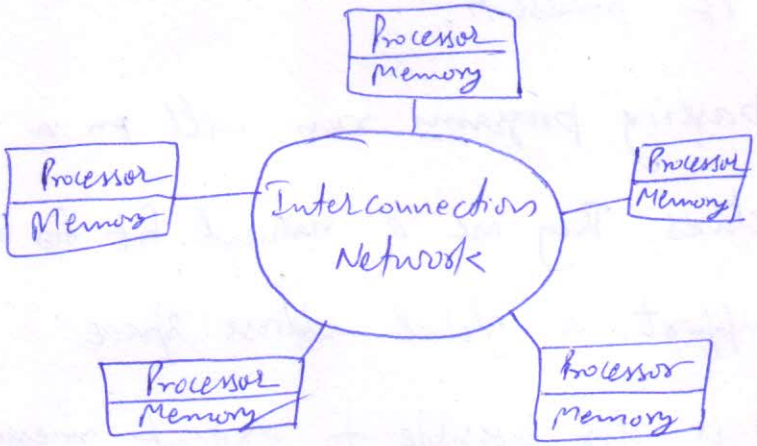
MPI: Message Passing Interface

MPI is a specification for the developers and users of message passing libraries. By itself, it is not a library- but rather the specification of what such a library should be.

MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.



"Message-Passing Model assumes that the underlying hardware is a collection of processors, each with its own local memory, and an interconnection network supporting message-passing between processors.

The user specifies the number of concurrent processes when the program begins, and typically the number of active processes remains constant throughout the execution of the program. Every process executes the same program but because each one has a unique ID number, different processes may perform different operations as the program unfolds. A process alternatively performs computations on its local variables and communicates with other processes or I/O devices.

It is important to realize that in a message-passing model, processes pass messages both to communicate and to synchronize with each other.

When a message containing data passes from one process to another, it serves a communication function.

A message has a synchronization function too. Process B cannot receive a message from Process A until after Process A sends it. Hence receiving a message tells Process B something about the state of process A.

Message passing programs run well on a wide variety of MIMD architectures. They are a natural fit for multicomputers, which do not support a global address space.

However, it is also possible to execute message-passing programs on multiprocessors by using shared variables as message buffers. In fact, the message passing model's distinction between faster, directly accessible local memory and slower, indirectly accessible remote memory encourages designers to develop algorithms that maximize local computations while minimizing communications.

Today, MPI has become the most popular message-passing library standard for parallel programming. It is available on most commercial multicomputers. Free versions of MPI libraries are readily available over the web.

Writing parallel programs using MPI allows you to port them to different parallel computers, though the performance of a particular program may vary widely from one machine to another.

Let's take a look at the C code

(1) Program begins with preprocessor directives to include the header files for MPI

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

Each active MPI process executes its own copy of the main program. That means, each MPI process has its own copy of all of the variables declared in the program, whether they be external variables (declared outside of any function) or automatic variables declared inside a function.

(2) The first MPI function call made by every MPI process is the call to MPI_Init which allows the system to do any setup needed to handle further calls to the MPI library.

- The call to MPI_Init does not have to be the first executable statement of the program
- In fact, it does not even have to be located in function main
- The only requirement is that MPI_Init be called before any other MPI function.

- All MPI Identifiers, including function identifiers, begin with the prefix MPI_, followed by a capital letter and a series of lowercase letters and underscores.
- All MPI constants are strings of capital letters and underscores beginning with MPI_.

```
MPI_Init( &argc, &argv);
```

---

(3) MPI_Comm_rank and MPI_Comm_size

When MPI has been initialized, every active process becomes a member of a communicator called MPI_COMM_WORLD

A communicator is an opaque object that provides the environment for message passing among processes.

MPI_COMM_WORLD is the default communicator that you get "for free". For most of our programs, it is sufficient. However, you can create your own communicator if you need to partition the processes into independent communication groups

```
int pid;
int nump;

MPI_Comm_rank( MPI_COMM_WORLD, &pid);
MPI_Comm_size( MPI_COMM_WORLD, &nump);
```

MPI_Finalize();

Processes within a communicator are ordered. The rank of a process is its position in the overall order. In a communicator with p processes, each process has a unique rank (ID number) between 0 and p-1.

A process may use its rank to determine which portion of a computation and/or a dataset it is responsible for.

A process calls function MPI_Comm_rank to determine its rank within a communicator.

It calls MPI_Comm_size to determine the total number of processes in a communicator.

(4) MPI_Finalize( )

After a process has completed all of its MPI library calls, it calls function MPI_Finalize; allowing the system to free up resources (such as memory) that have been allocated to MPI.

```
MPI_Finalize();
```

(5) Compiling the MPI Progeams

```
mpicc  prog.c  -o  progout
```

(6) Running MPI progeams

```
mpirun  -np 10  progout
```

indicates the number of processes to create.

A Collective Communication is a communication operation in which a group of processes works together to distribute or gather together a set of one or more values. Reduction and Broadcast are example of operation that require collective communication in a message-passing environment.

General MPI Program Structure

MPI Include file

Declarations, prototypes etc.

Program Begins

⋮ Serial code

Initialize MPI environment (Parallel code begins)

⋮

Do work & make message passing calls

⋮

Terminate MPI environment (Parallel code ends)

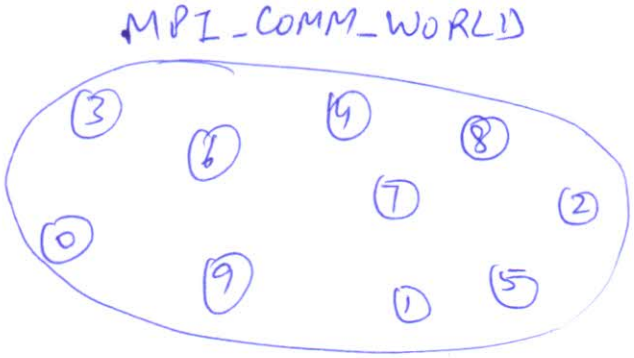
⋮ Serial code

Program Ends

Header file :

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

Communicators and Groups



MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. Most MPI routines require you to

specify a communicator as an argument.

MPI_COMM_WORLD is the predefined communicate that include all of your MPI processes.

Rank: Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero. (between 0 and number of tasks − 1)

These are used by programmers to specify the source and destination of messages. Often used conditionally by the application to control program execution

if (rank==0) do this / if (rank==1) do that

Error handling

Most MPI routines include a return/error code parameter. However, according to MPI standard, the default behavior of an MPI call is to abort if there is an error. This means you will probably not be able to capture a return/error code other than MPI_SUCCESS (zero).

Environment Management Routines

MPI_Init (&argc, &argv) : Initializes the MPI execution environment. This must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

MPI_Comm_size (comm, &size) : returns the total number of MPI processes in the specified communicator. MPI_COMM_WORLD

MPI_Comm_rank (comm, &rank) : returns the rank of calling MPI process within the specified communicator

MPI_Abort (comm, errorcode) : Terminate all MPI processes associated with the communicator

MPI_Initialized (&flag) : Indicates whether MPI_Init has been called

MPI_Wtime : returns an elapsed wall clock time in seconds (double precision) on the calling processor

MPI_Wtick : returns the resolution in seconds (double precision) of MPI_Wtime

MPI_Finalize() : Terminate the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routine may be called after it.

Function MPI_Wtime and MPI_Wtick

One way to measure the performance of a parallel application is to look at the wall clock time, measuring the number of seconds that elapse from time we initiate execution until the program terminates. In production environments, this may be the most useful metric.

However, here, we are going to ignore the time spent initiating MPI processes, establishing communications sockets between them, and performing I/O on sequential devices. Instead, we will measure how well our parallel programs stack up against their sequential counterparts in the "middle area" between reading the dataset and writing the results.

MPI_Wtime :- returns the number of seconds that have elapsed since some point of time in the past

MPI_Wtick :- returns the precision of the result returned by MPI_Wtime

```
double MPI_Wtime (void)
double MPI_Wtick (void)
```

We can compute time of a section of code by putting a pair of calls to function MPI_Wtime before and after the section. The difference between two values returned by the function is the number of seconds elapsed.

[From a logical point of view, every MPI process begins execution at the same time, but this is not true in practice.]

Function MPI_Barrier

The problem of difference in execution initiation time of MPI processes may be addressed by introducing a barrier synchronization before the first call to MPI_Wtime.

No process can proceed beyond a barrier until all processes have reached it. Hence a barrier ensure that all processes are going into the measured section of code at more or less the same time

```
int MPI_Barrier(MPI_Comm comm)

double elapsed_time;

MPI_Init(&argc, &argv);
MPI_Barrier(MPI_COMM_WORLD);
elapsed_time -- MPI_Wtime();

MPI_Reduce(------)
elapsed_time += MPI_Wtime();
```

Point to Point Communication Routines :- Blocking Message Passing Routines

Blocking send operation.

MPI_Send and MPI_Recv: (Recv block until the requested data is available in the application buffer in the receiving task.)

(Send Routine returns only after the application buffer in the sending task is free for reuse.)

MPI's send and receive calls operate in the following manner:

First, process A decides a message needs to be sent to process B. Process A then packs up all of its necessary data into a buffer for process B. These buffers are often referred to as envelopes since the data is being packed into a single message before transmission. After the data is packed into a buffer, the communicating device (which is often a network) is responsible for routing the message to the proper location. The location of the message is defined by the process's rank.

Even though the message is routed to B, process B still has to acknowledge that it wants to receive A's data. Once it does this, the data has been transmitted. Process A is acknowledged that the data has been transmitted and may go back to work.

Sometimes there are cases when A might have to send many different types of messages to B. Instead of B having to go through extra measures to differentiate all these messages, MPI allows senders and receivers

to also specify message IDs with the message (known as tags). When process B only requests a message with a certain tag number, messages with different tags will be buffered by the network until B is ready for them.

Prototypes for the MPI sending and receiving functions:

```
MPI_Send(
    void * data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag
    MPI_Comm communicator)
```

Once a program calls MPI_Send, it blocks until the data transfer has taken place and the buffer can be safely reused. As a result, these routines provide a simple synchronization service along with data exchange.

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

count, source and tag values may be specified so as to allow messages of unknown length, from several sources (MPI_ANY_SOURCE) or with various tag values (MPI_ANY_TAG). The amount of information actually received can then be retrieved from status variable. MPI_Recv blocks until the data transfer is complete

1st Argument — data buffer

2nd, 3rd Argument — describe the count and type of elements that reside in the buffer

MPI_Send sends the exact count of elements and MPI_Recv will receive at most the count of elements

4th Arg - specify the rank of the sending process

5th Argument - specify the tag of the message

6th — specifies the communicator

Last Argument in MPI_Recv provides information about the received message.

```
#include <stdio.h>
#include <Stdlib.h>
# include <mpi.h>

int main( int argc, char **argv)
{
   MPI_Init(&argc, &argv)   // we may pass NULL also
   int wrank;
   MPI_Comm_rank(MPI_COMM_WORLD, &wrank);
   int wsize;
  MPI_Comm_size(MPI_COMM_WORLD, &wsize);
  if( wsize <2)
  {
    printf( "World size must be greater than 1\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
  }
  int number;
 if( wrank ==0)
 {   number = -1;
    MPI_Send (&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
 } else if ( wrank==1)
 { MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
    printf("Process 1 received number %d from process 0\n", number);
  }
  MPI_Finalize();
}
```

((For tag, the processes could have also used the predefined constant MPI_ANY_TAG since only one type of message was being transmitted))

((Reference :- https://mpitutorial.com/tutorials/mpi-send-and-receive

Point to Point Communication Routines

Blocking Message Passing Routines

MPI_Send
MPI_Recv
MPI_Ssend
MPI_Sendrecv
MPI_Wait
MPI_Waitany
MPI_Waitall
MPI_Waitsome
MPI_Probe
MPI_Get_count

Non-Blocking Message Passing Routines

MPI_Isend
MPI_Irecv
MPI_Issend
MPI_Test
MPI_Testany
MPI_Testall
MPI_Testsome
MPI_Iprobe

Collective Communication Routines

- Broadcast
- Scatter
- Gather
- Reduction

Types of collective Operations

MPI_Barrier → Synchronization – processes wait until all members of the group have reached the synchronization point

MPI_Bcast → Data Movement → Broadcast, scatter, gather, all to all
MPI_Scatter
MPI_Gather

MPI_Allgather
MPI_Allreduce
MPI_Reduce_scatter
MPI_Alltoall
MPI_Scan

MPI_Reduce → Collective computation (reductions) – one member of the group collects data from the other members and performs an operation (min, max, add, multiply etc.) on that data.

Scope:
- Collective communication routines must involve all processes within the scope of a communicator
- All processes are by default, members in the communicator MPI_COMM_WORLD.
- Additional communicators can be defined by the programmer.
- Unexpected behavior, including program failure, can occur if even one task in the communicator doesn't participate.
- It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

Function MPI_Reduce

After a process has completed its share of the work, it is ready to participate in the reduction operation. Function MPI_Reduce performs one or more reduction operations on values submitted by all the processes in a communicator.

```
int MPI_Reduce (
    void      *operand,
    void      *result,
    int       count,
    MPI_Datatype  type,
    MPI_Op    operator,
    int       root,
    MPI_Comm  comm)
```

operand is an input parameter. The calling process indicates the location of its element for the first reduction. If count is greater than 1, then the list elements for all of the reductions occupy a contiguous block of memory

count indicates how many reductions are being performed Each process submits count values, and each of these values is a list element for a different reduction.

type is an input parameter designating the type of the elements being reduced.

operator indicates the kind of reduction to perform

root gives the rank of the process that will have the result of all the reductions

result points to the location of the first reduction result. This parameter only has meaning for process root.

comm gives the name of the communicator - that is, the set of processes participating in the reduction

---

MPI constants for C data types

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

MPI Built-in reduction operators

MPI_BAND	Bitwise AND
MPI_BOR	Bitwise OR
MPI_BXOR	Bitwise XOR
MPI_LAND	Logical AND
MPI_LOR	Logical OR
MPI_LXOR	Logical XOR
MPI_MAX	Maximum
MPI_MAXLOC	Maximum and location of maximum
MPI_MIN	Minimum
MPI_MINLOC	Minimum and locations of minimum
MPI_PROD	Product
MPI_SUM	Sum

```
int solutions;
int global_solutions;
```

- `MPI_Reduce(&solutions, &global_solutions, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);`

(an example of call to MPI_Reduce ...)

Every process in the communicator must enter the reduction voluntarily - it cannot be "summoned" by process 0. If you write a program in which not all the processes in a communicator call MPI_Reduce or any other collective communication function, the program will "hang" at the point that function is executed, unable to complete it.

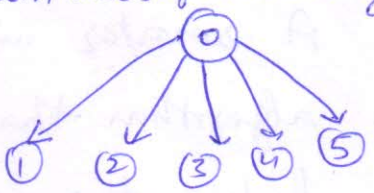
→ Result of reduction is available with only process 0. The value of "global_solutions" (above) variable will be undefined for other processes.

```
if (id == 0) printf("Final Result : %d\n", global_solutions);
```

Function MPI_Bcast

It sends a message from one process to all processes in a communicator. One of the main uses of broadcasting is to send out user input to a parallel program.

```
int MPI_Bcast( void *data_to_be_sent,
               int send_count,
               MPI_Datatype datatype,
               int broadcasting_process_id,
               MPI_Comm comm);
```



In this, process 0 is the root process, and it has the initial copy of data. All of the other processes receive the copy of data.

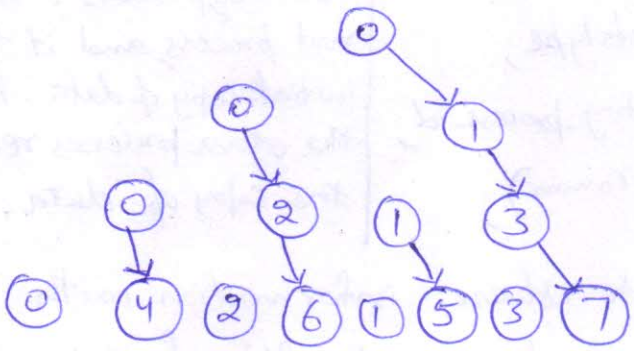
- When processes are ready to share information with other processes as part of a broadcast, ALL of them must execute a call to MPI_Bcast. There is no separate MPI call to receive a broadcast.

MPI_Bcast, MPI_Scatter, and other collective routines build a communication tree among the participating processes to minimize message traffic. If there are N processes involved, there would be normally N-1 transmissions during a broadcast operation, but if a tree is built so that the broadcasting process sends the broadcast to 2 processes, and they each send it on to 2 other processes, the total number of messages transferred ~~at~~ timing instances would be only $O(\log N)$.

⇒ There can be a situation, in which we may implement Broadcast with MPI_Send and MPI_Recv. Root process

sends the data to everyone else while the others receive from the root process. It is very inefficient. Such a function will use only one network link from process zero to send all the data. (Each process may have only one outgoing/ incoming network link)

A smarter implementation is a tree-based communication algorithm that can use more of the available network links at once.



Stage 1 → P0 sends data to P1 in step 1

Stage 2 → P0 sends the data to P2 in second stage. Process 1 forwards the data to P3.

During the second stage, two network communications are being utilized at a time. The network utilization doubles at every subsequent stage of the tree communication until all processes have received the data.

Reference

https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

Collective Communications

When all processes in a group (communicator) participate in a global communication operation, the resulting communication is called a collective communication. MPI provides a number of collective communication routines. Below are some of them assuming a communicator Comm contains n processes:

1. Broadcast: Using the routine

MPI_Bcast (Address, Count, Datatype, Root, Comm)

the process ranked Root sends the same message whose content is identified by the triple (Address, Count, Datatype) to all processes (including itself) in the communicator Comm. This triple specified both the send and the receive buffers for the Root process whereas only the receive buffer for the other processes.

2. Gather: The routine

MPI_Gather (Send Address, Send Count, Send Datatype, Recv Address, Recv Count, Recv Datatype, Root, Comm) facilitates the Root process receiving a personalized message from all the n processes (including itself). These n received messages are concatenated in rank order and stored in the receive buffer of the Root process. Here the send buffer of each process is identified by (Send Address, Send Count, Send Datatype) while the receive buffer is ignored for all processes except

the root process where it is identified by RecvAddress, RecvCount, RecvDatatype).

3) Scatter: This is the opposite of gather operation. The routine MPI_Scatter() ensure that the Root process sends out personalized messages, which are stored in rank order in its send buffer, to all the n processes (including itself)

4) Total Exchange: This is also known as an all-to-all In the routine MPI_Alltoall() each process sends a personalized message to every other process including itself. Note that this operation is equivalent to n gathers, each by a different process and in all n^2 messages are exchanged.

5) Aggregation: MPI provides two forms of aggregation - Reduction and scan.

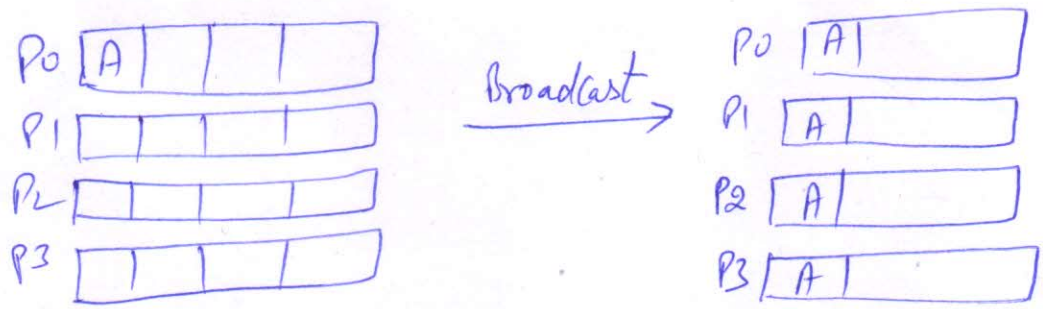
The MPI_Reduce (SendAddress, RecvAddress, Count, DataType, Op, Root, Comm) routine reduces the partial values stored in SendAddress of each process into a final result and stores it in RecvAddress of the Root process. The reduction operator is specified by the Op field.

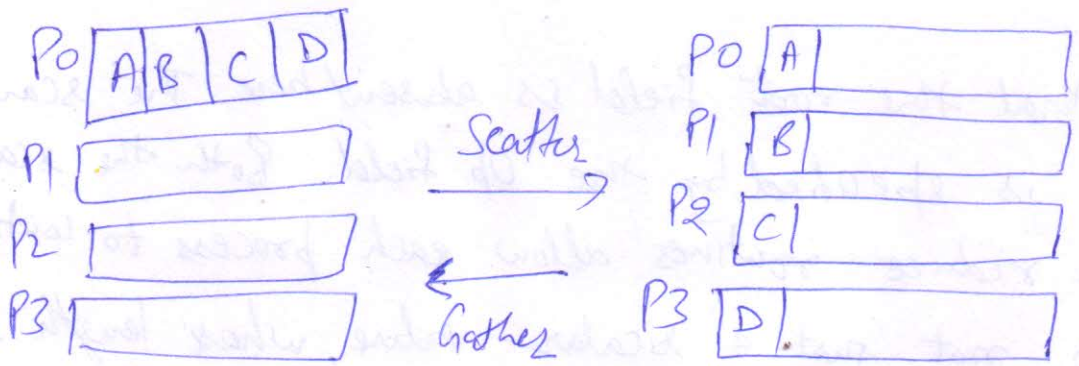
The MPI_Scan (SendAddress, RecvAddress, Count, Datatype, Op, Comm) routine combines the partial values into n final results which it stores in the RecvAddress of the n processes.

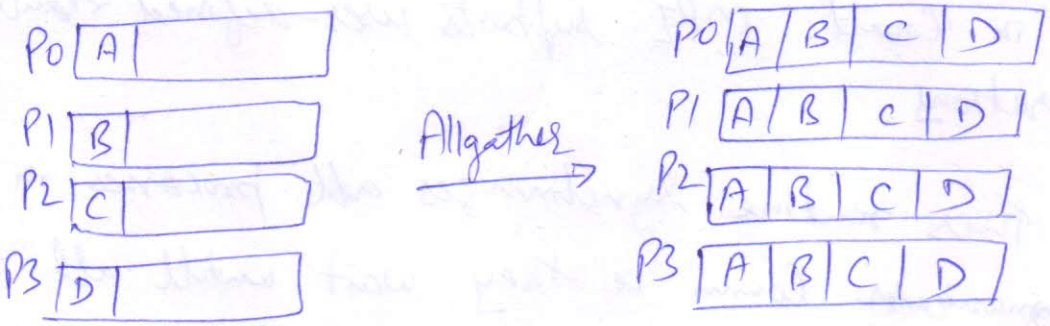
Note that the root field is absent here. The scan operator is specified by the Op field. Both the scan and the reduce routines allow each process to contribute a vector, not just a scalar value, whose length is specified in Count. MPI supports user-defined reduction/scan operations.

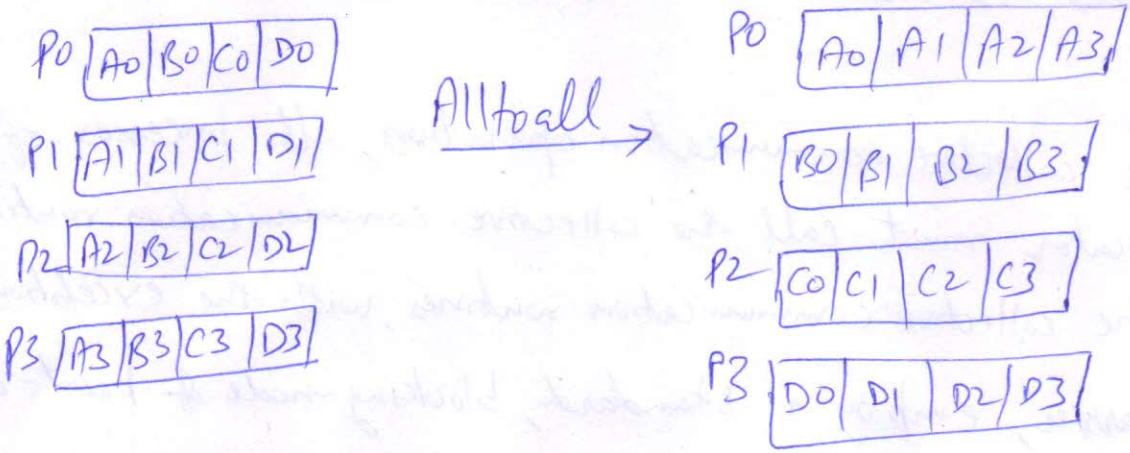
6) Barrier: This routine synchronizes all processes in the communicator Comm i.e. they wait until all n processes execute their respective MPI_Barrier(Comm) routine.

- In a collective communication operation, all processes of the communicator must call the collective communication routine.
- All the collective communication routines, with the exception of MPI_Barrier, employ a standard, blocking mode of point-to-point communication.
- The count and the Datatype should match on all the processes involved in the collective communication.
- There is no tag argument in a collective communication routine.




P0
A
B
C
D
P1
P2
P3
Scatter
Gather
P0
A
P1
B
P2
C
P3
D


P0
A
P1
B
P2
C
P3
D
Allgather
P0
A
B
C
D
P1
A
B
C
D
P2
A
B
C
D
P3
A
B
C
D


P0
A0
B0
C0
D0
P1
A1
B1
C1
D1
P2
A2
B2
C2
D2
P3
A3
B3
C3
D3
Alltoall
P0
A0
A1
A2
A3
P1
B0
B1
B2
B3
P2
C0
C1
C2
C3
P3
D0
D1
D2
D3