Department of Information Technology
National Institute of Technology Karnataka, Surathkal

Distributed Memory Parallelism with MPI

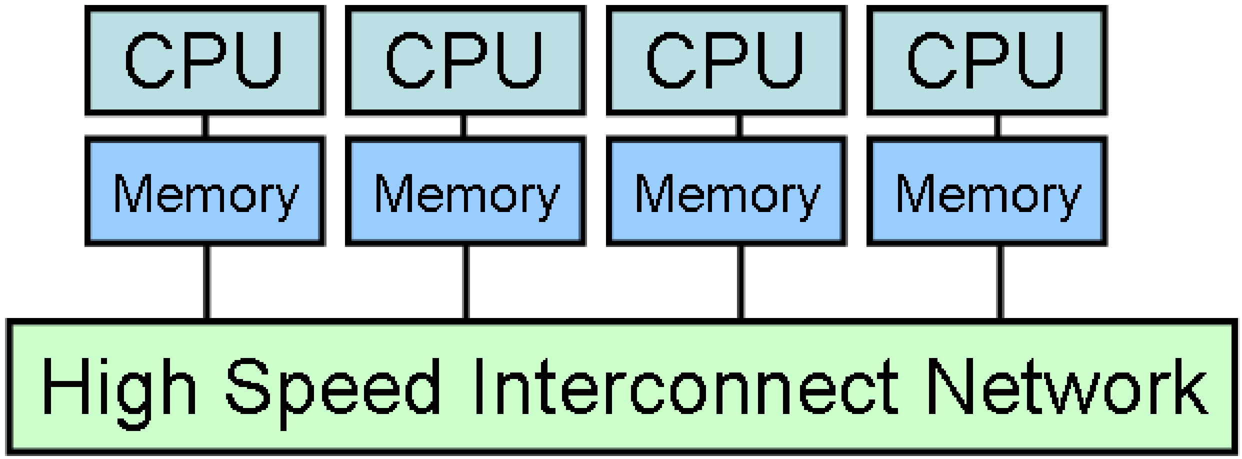
By,
Geetha V
Dept of IT,
NITK Surathkal

Outline

- Distributed Memory Architecture
- Introduction to MPI
- Structure of MPI program
- Types of Message Passing
- Basic Routines in Point to Point Communication
- Example programs on Point to Point Communication
- Basic Routines in Collective Communication
- Sample Programs on Collective Communication

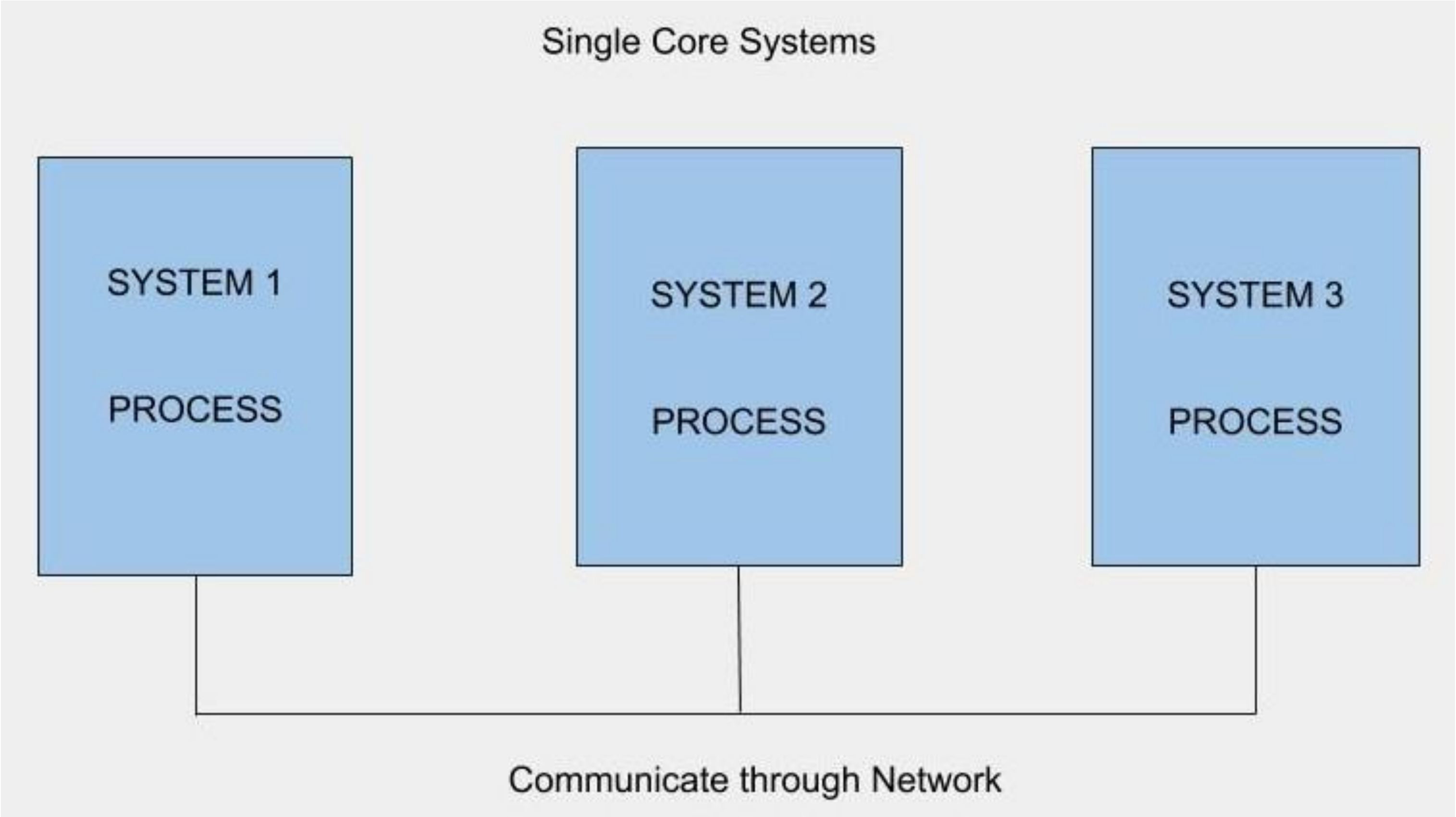
Distributed Memory Architecture

- Each processor has its own memory
- They cannot access the memory of other processors.
- Any data that needs to be shared must be explicitly transmitted from one processor to another using **Message Passing.**



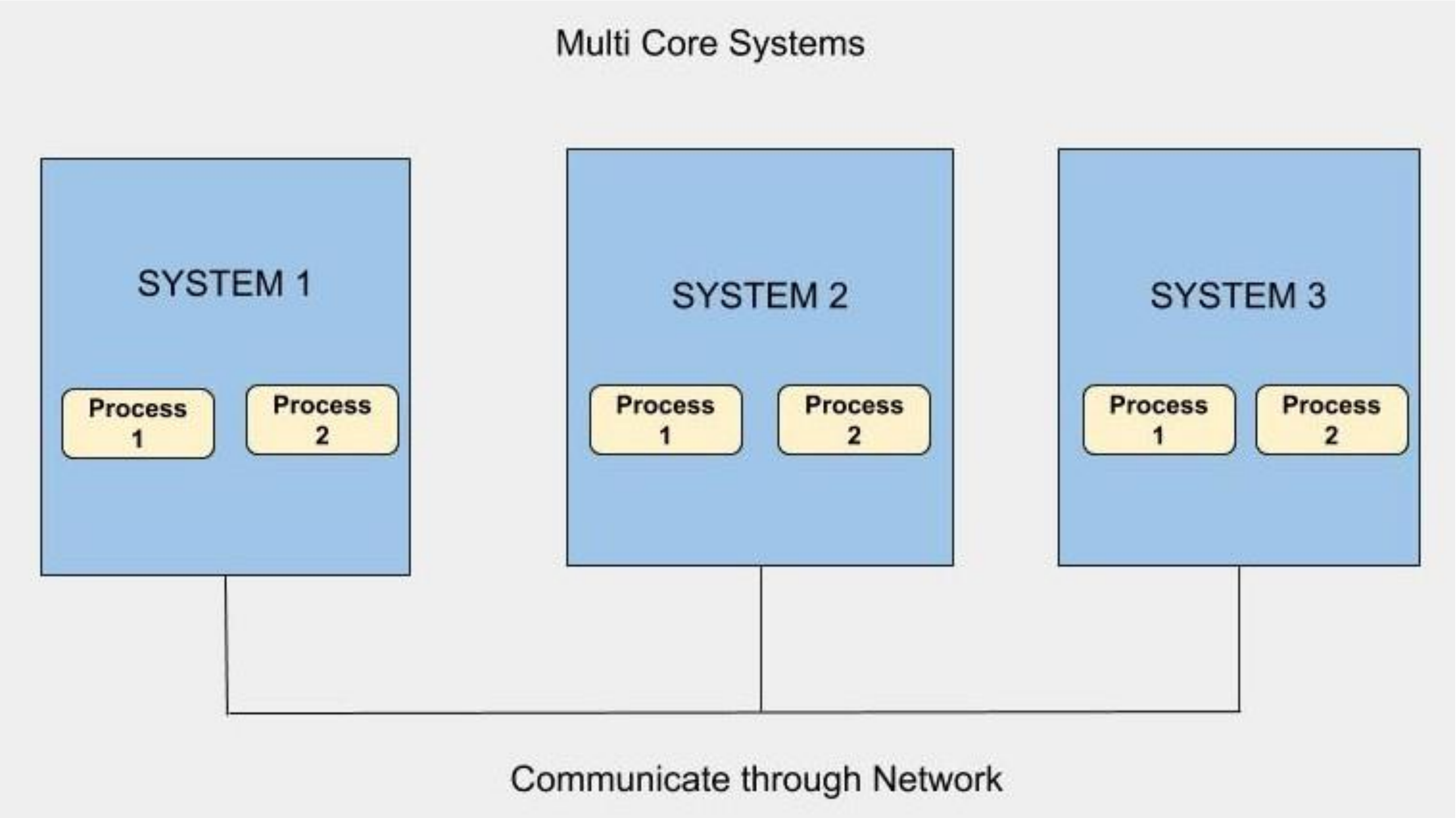
DISTRIBUTED MEMORY ARCHITECTURE

- **Systems with single core communicating through distributed memory.**
- **Heterogeneous systems**

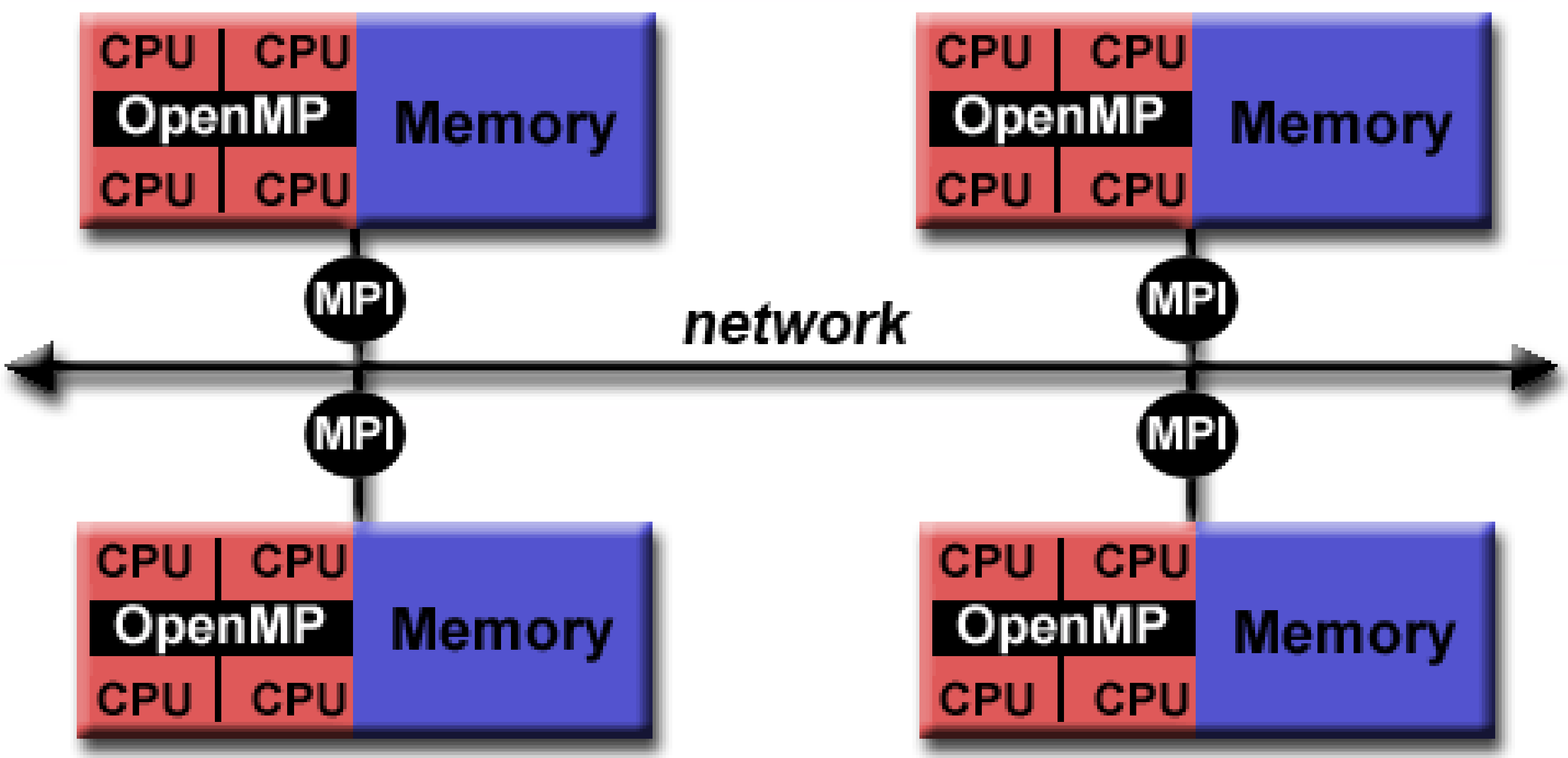


DISTRIBUTED MEMORY ARCHITECTURE

- Systems with multiple core communicating through shared and distributed memory

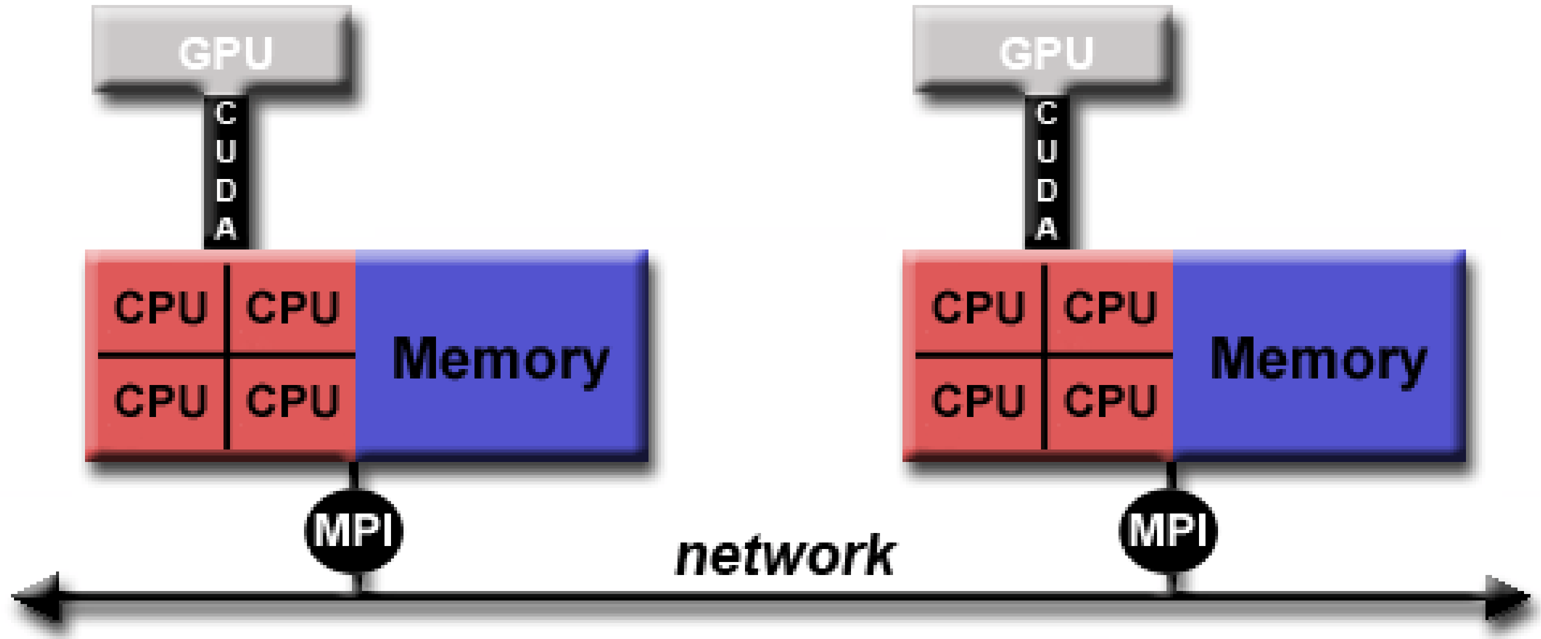


Hybrid Model



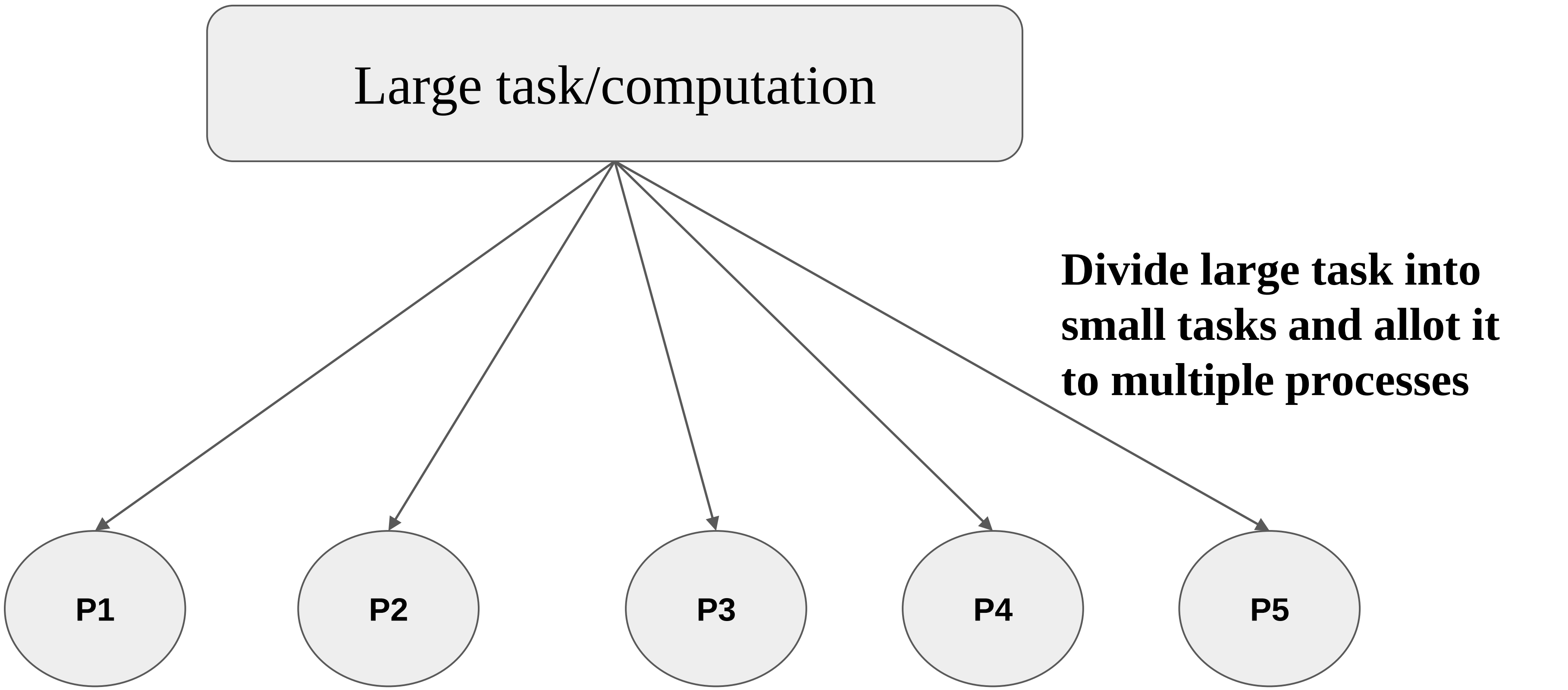
Reference: https://computing.llnl.gov/tutorials/parallel_comp/#ModelsMessage

Hybrid Model



Reference: https://computing.llnl.gov/tutorials/parallel_comp/#ModelsMessage

Parallel Computation:



For Example: N = 1,00,000 divided into P1=20,000, P2=20,000 ……

Computation is same. Data is different. Single Program Multiple Data

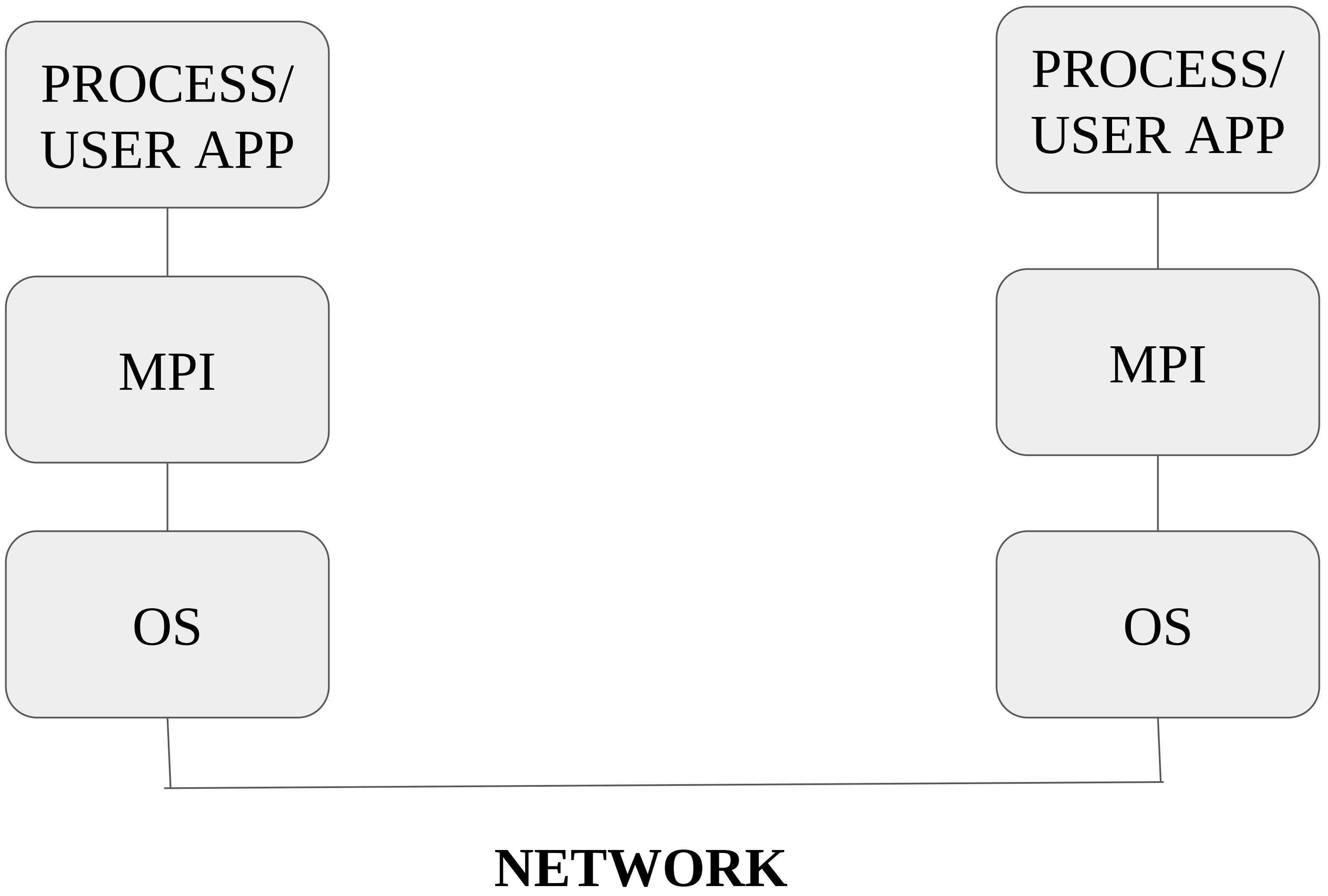
INTRODUCTION TO MPI

- **What is MPI?**
  - Message Passing Interface is a specification.
    - A standard for vendors to implement.
  - It is a library, i.e. a set of subroutines, functions and constants
  - Allows Message Passing between processes.
  - It is based on Single Program, Multiple Data (SPMD)
    - Every process executes the same program
    - Each process performs computations on its local variables, then communicates with other processes, in order to get the final result.

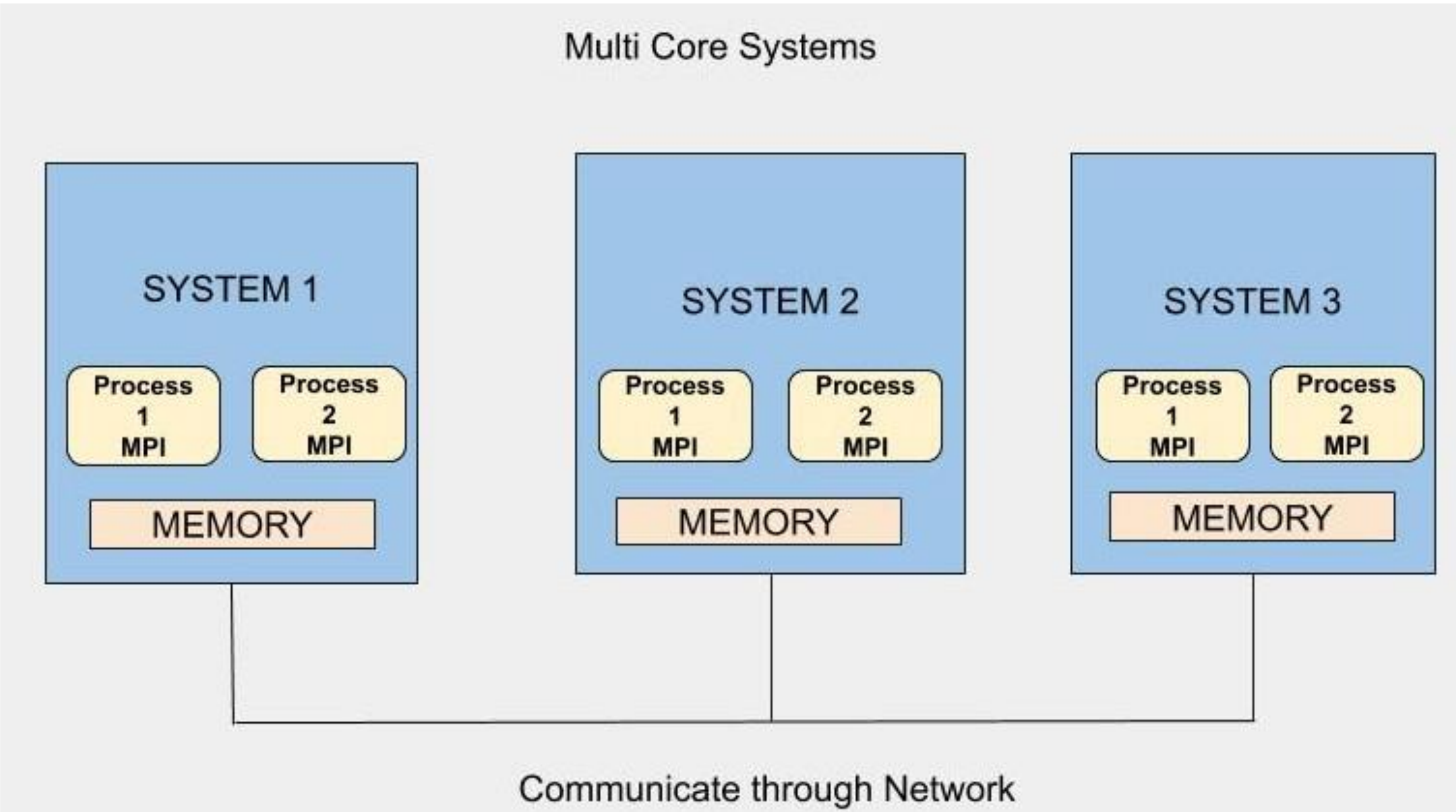
MPI : Major Goals

- Portability :
  - An MPI library exists on ALL parallel computing platforms so it is highly portable.
- Support heterogeneity
- High performance through efficient implementations
- Encourage overlap of communication and computations.
- Reliability

MPI is a Middleware



MPI is a Middleware



MPI Implementations

- OpenMPI (www.open-mpi.org)
- MPICH (www.mpich.org)
- HP MPI
- Intel MPI
- Scali MPI
- IBM MPI

Outline

- Distributed Memory Architecture
- Introduction to MPI
- Structure of MPI program
- Types of Message Passing
- Basic Routines in Point to Point Communication
- Example programs on Point to Point Communication
- Basic Routines in Collective Communication
- Sample Programs on Collective Communication

STRUCTURE OF MPI PROGRAM

MPI Include File

Initialize MPI Environment

Computations and Message Passing

Terminate MPI Environment

MPI Routines

- **Start and terminate :**
  - To initialize and terminate the MPI environment
- **Communicators :**
  - To identify the communication world (cluster of processes)
- **Getting Information :**
  - To get the number of processes and process ids
- **Sending and Receiving messages :**
  - Actual computation and communication

STRUCTURE OF MPI PROGRAM

MPI Include File

#include<mpi.h>

Initialize MPI Environment

MPI_Init(&argc,&argv);

Computations and Message Passing

Terminate MPI Environment

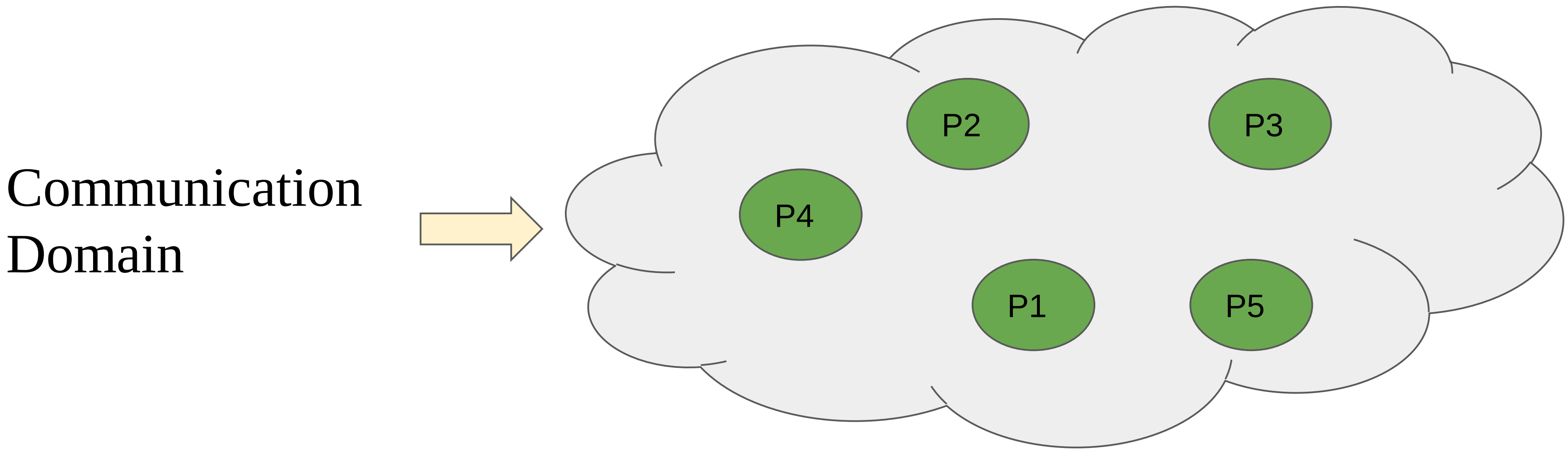
MPI_Finalize();

MPI Start and Terminate Routines

```
#include<stdio.h>
int main(int argc,char **argv)
            {
                  -----------
                  -----------
             MPI_Init(&argc,&argv);
              -----------
                  -----------
             MPI_Finalize();
             -----------
             return 0;
             }
```

Communicators

- MPI defines **communication domain** – set of processes that can communicate with each other.
- MPI_comm : data type – stores information about communication domains.
- Default communicator - MPI_COMM_WORLD



Getting Information

- MPI_Comm_size
- MPI_Comm_rank

- **Syntax :**
- **int MPI_Comm_size(MPI_Comm comm, int *size)**
- **Int MPI_Comm_rank(MPI_Comm comm, int *rank)**

General MPI Program

```
#include<mpi.h>
        int main(int argc,char **argv)
        {
         -----------
         -----------
         MPI_Init(&argc,&argv);
         -----------
         MPI_Comm_size(MPI_COMM_WORLD,&size);
         MPI_Comm_rank(MPI_COMM_WORLD,&rank);
         -----------
         MPI_Finalize();
         -----------
         return 0;
         }
```

Example: Hello World

```
#include<mpi.h>
int main(int argc,char *argv[ ])
{
 int size,myrank;
 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&size);
 MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
 printf(“Process %d of %d, Hello World”,myrank,size);
 MPI_Finalize();
 return 0;
 }
```

MPI Hello World :

```
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 5 ./a.out
Process 0 of 5, Hello World
Process 1 of 5, Hello World
Process 4 of 5, Hello World
Process 2 of 5, Hello World
Process 3 of 5, Hello World
```

MPI Include File

Initialize MPI Environment

Computations and Message Passing

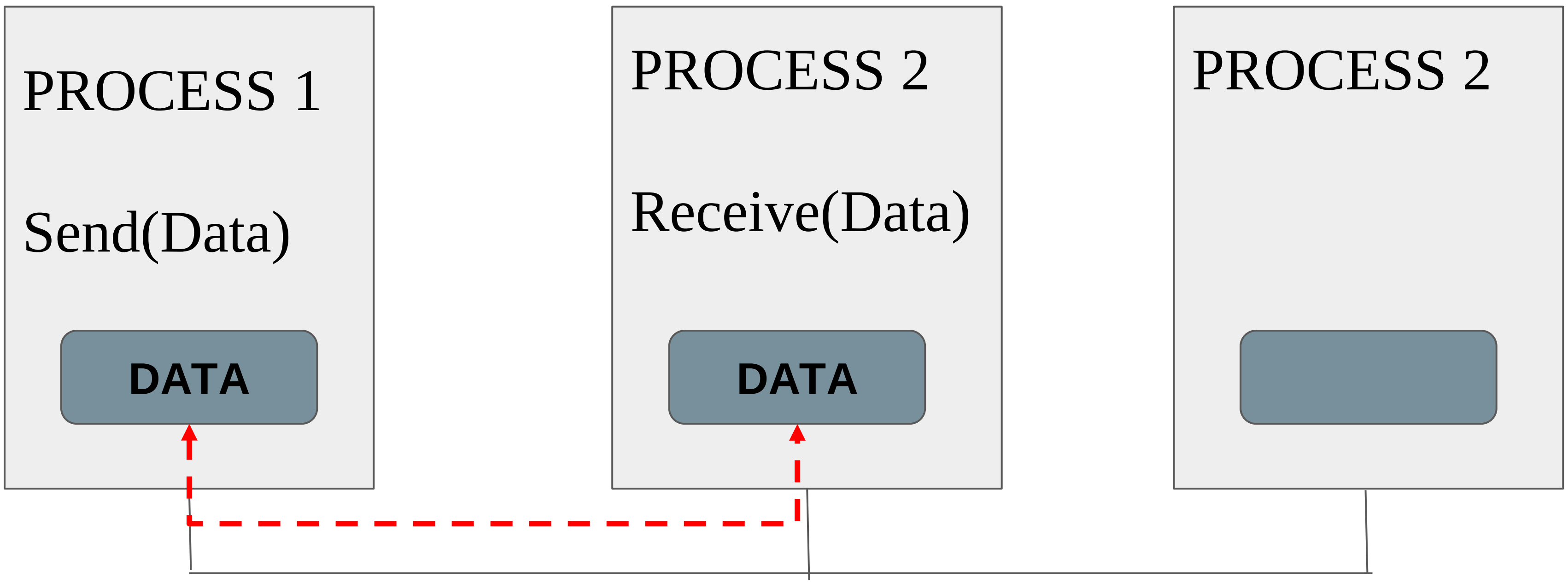
Terminate MPI Environment

Types of Message Passing:

- **Point to Point**
  - Two processes
  - Send and Receive are the basic functions
- **Collective messages**
  - Group of processes involved in communication
  - Functions like Broadcast, Scatter, Gather, Parallel Reduction

Point to Point Communication

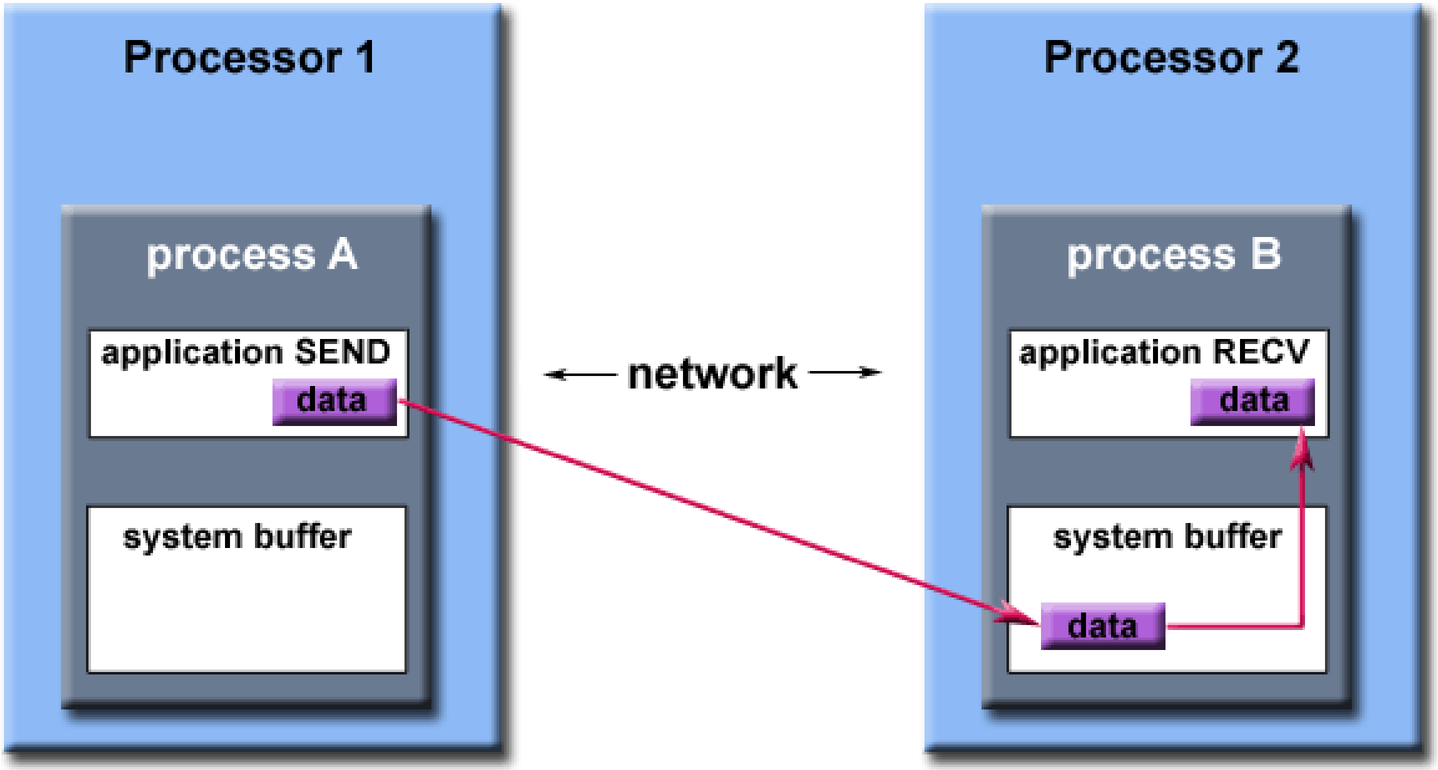
- Two processes involved in sending and receiving data.



- ID of sender and receiver is required.
- Specify what has to be sent and received.
- Communication needs to be synchronized.
- Communication makes use of buffers.

Point to Point Communication

- **Data transfer from Sender Process to Receiver Process.**



Path of a message buffered at the receiving process

Send and Receive Variants

- **Blocking Send and Receive**
- **Non Blocking Send and Receive**
- **Based on modes of Communication:**
  - Standard
  - Synchronous
  - Buffered
  - Ready

Blocking Send and Receive

- **Basic Send and Receive routine for point to point communication.**
- **MPI Routines:**
    - MPI_Send()
    - MPI_Recv()

Outline

- Distributed Memory Architecture
- Introduction to MPI
- Structure of MPI program
- Types of Message Passing
- Basic Routines in Point to Point Communication
- Example programs on Point to Point Communication
- Basic Routines in Collective Communication
- Sample Programs on Collective Communication

Blocking Send and Receive

- **MPI_Send()**

MPI_Send (void *buf, int count, MPI_Datatype type,int dest, int tag, MPI_Comm comm)

Parameters:

buf	**:**	initial address of send buffer
count :		number of elements in send buffer (nonnegative integer)
datatype :		datatype of each send buffer element. Ex : MPI_INT, MPI_CHAR
dest :		rank of destination (integer)
tag :		message tag (integer). For tagging send and receive.
comm :		Communication domain of the communicating processes.

Blocking Send and Receive

- **MPI_Recv():**

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

Parameters:

buf	**:**	initial address of receive buffer
count :		max number of elements in receive buffer (nonnegative integer)
datatype :		datatype of each receive buffer element. Ex : MPI_INT, MPI_CHAR
source :		rank of source (integer)
tag :		message tag (integer). For tagging send and receive.
comm :		Communication domain of the communicating processes.
status:		status object (Status). It is a structure containing information about source, tag and error code.

- **MPI DATATYPES:**

Table 1: Basic C datatypes in MPI

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	

General MPI Program

```
#include<mpi.h>
    int main(int argc,char **argv)
    {
        ...
        MPI_Init(&argc,&argv);
        ...
        MPI_Comm_size(MPI_COMM_WORLD,&size);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    COMPUTATIONS AND MESSAGE PASSING

        MPI_Finalize();
        ...
        return 0;
    }
```

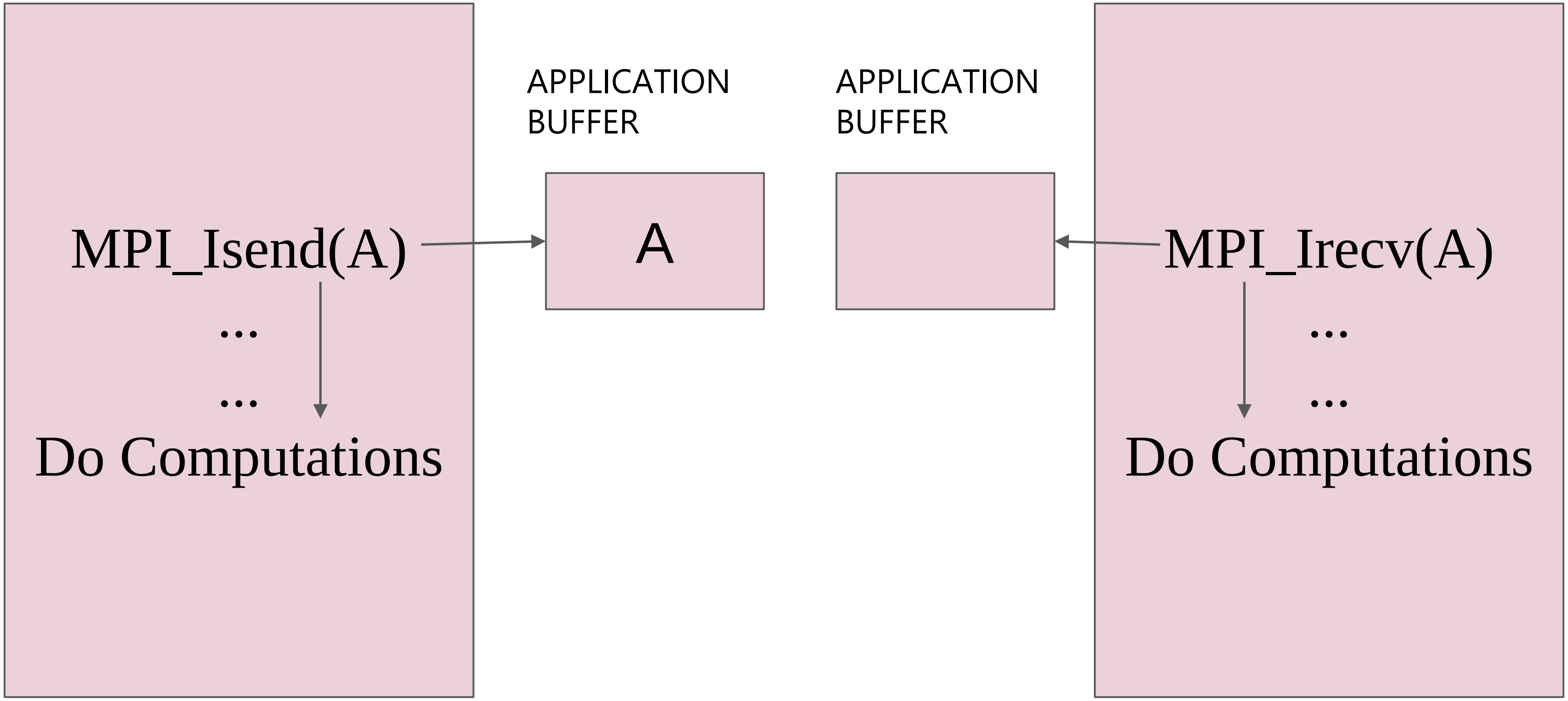
MPI Example - 1

```
        for(i=0;i<50;i++) //Process 0 initializes array x
        x[i]=i+1;
if(myrank==0)
MPI_Send(x,10,MPI_INT,1,1,MPI_COMM_WORLD);
else if(myrank==1)
{
MPI_Recv(y,10,MPI_INT,0,1,MPI_COMM_WORLD,&status);
printf("Process %d Received Data from Process %d\n",
myrank,status.MPI_SOURCE);
        for(i=0;i<10;i++)
        printf("%d\t",y[i]);
        }
```

```
Process 1 Recieved data from Process 0
1       2       3       4       5       6       7       8       9       10
```

Non Blocking Send and Receive

- **Allows overlapping of computation and communication**
- **Advantage is Performance Gain**



Non Blocking Send and Receive

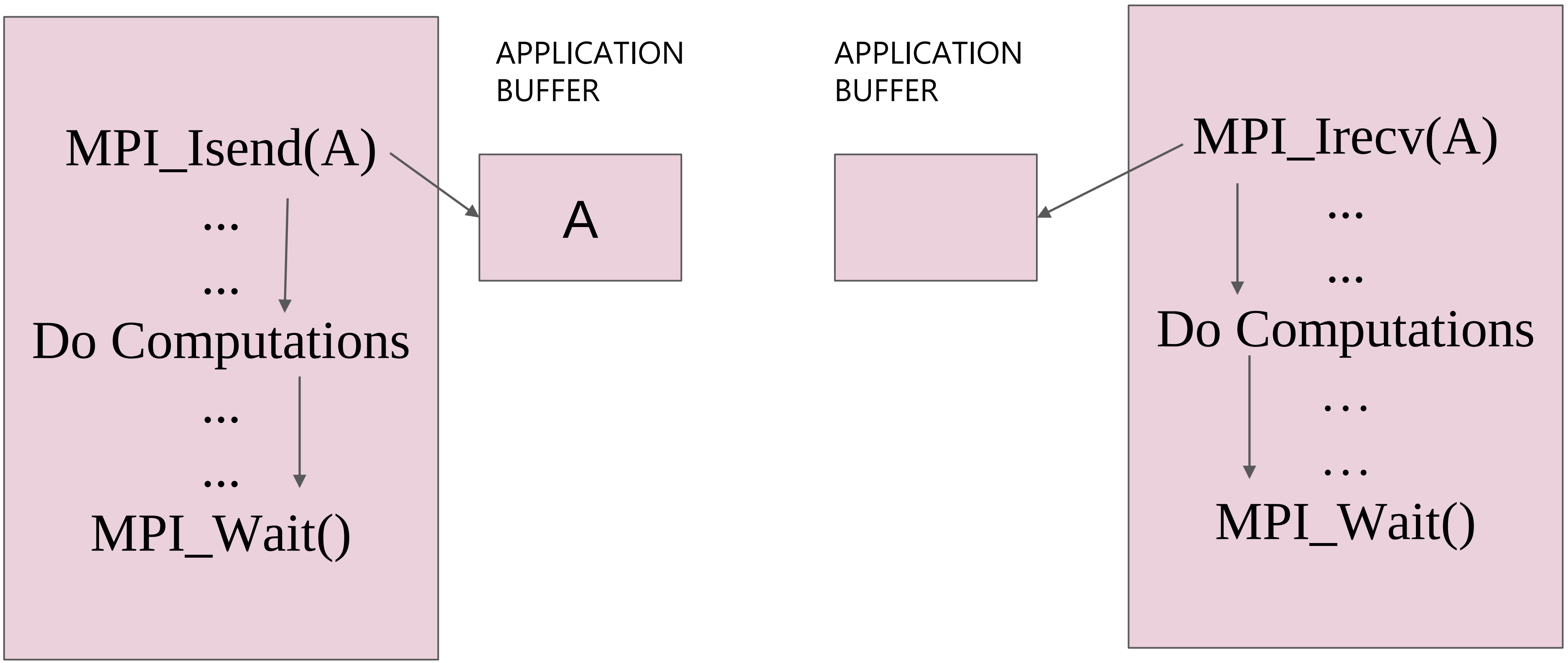
MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)

MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)

Parameters:

- **Same as Send() and Recv() except for request**
- **request : handle. This helps to get information about MPI_Isend and MPI_Irecv status.**
- **Used in routines : MPI_Wait() and MPI_Test()**

Non Blocking Send and Receive



MPI_Wait() and MPI_Test()

Syntax :

int MPI_Wait(MPI_Request *request, MPI_Status *status);

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);

- **If request is set to MPI_REQUEST_NULL (set if operation is completed) then:**
  - **MPI_Wait returns immediately with an empty status.**
  - **MPI_Test sets flag to true and returns an empty status.**

MPI Example - 2

```
if(myrank==0)
{
x=10;
MPI_Isend(&x,1,MPI_INT,1,20,MPI_COMM_WORLD,&request);
printf("Send returned immediately\n");
}
else if(myrank==1)
{
MPI_Irecv(&x,1,MPI_INT,0,25,MPI_COMM_WORLD,&request);
printf("Receive returned immediately\n");
printf("Process %d of %d, Value of x is %d\n",myrank,size,x);
}
```

```
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 2 ./a.out
Send returned immediately
Receive returned immediately
Process 1 of 2, Value of x is 0
```

What is the risk here?

```
if(myrank==0)

{

x=10;

MPI_Isend(&x,1,MPI_INT,1,20,MPI_COMM_WORLD,&request);

printf("Send returned immediately\n");

x=x+10;

}
```

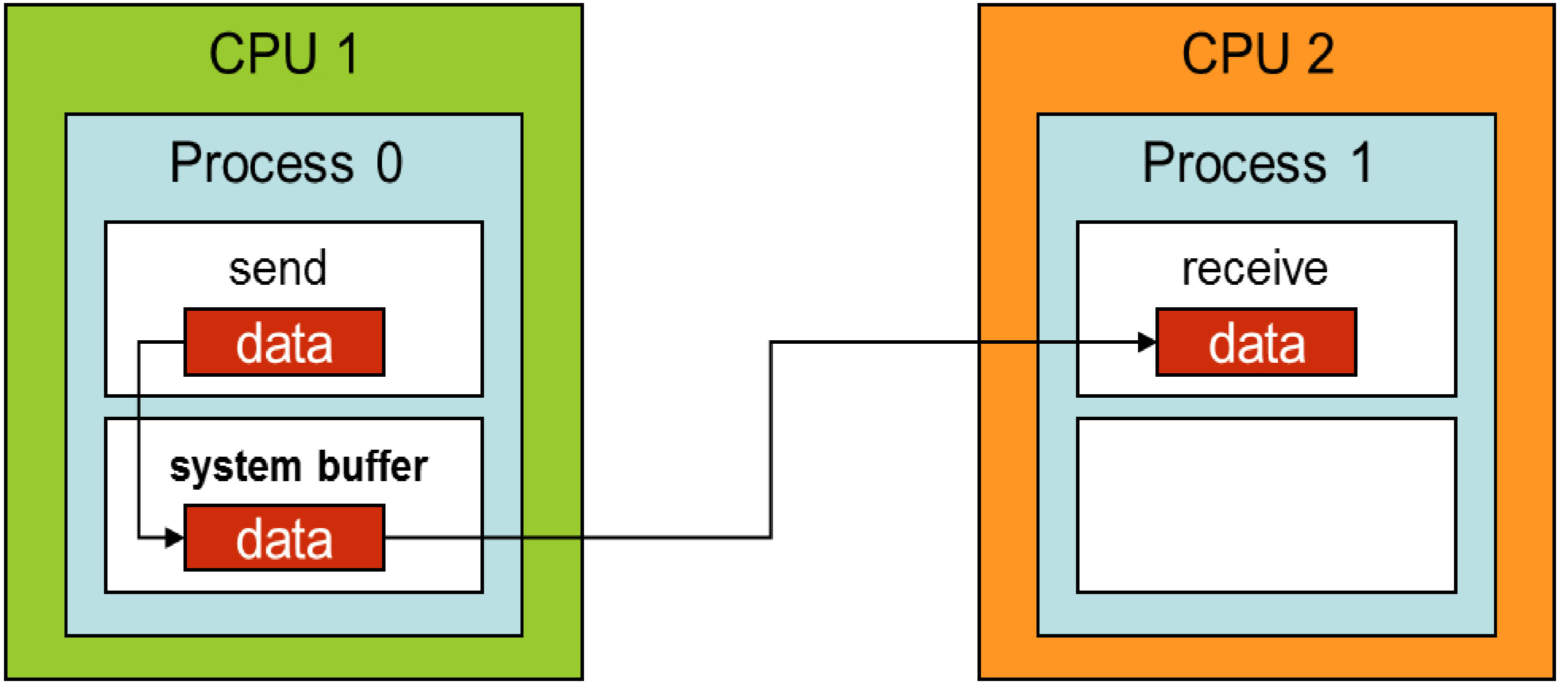
Make sure that x is available for reuse:

```
if(myrank==0)
{
x=10;
MPI_Isend(&x,1,MPI_INT,1,20,MPI_COMM_WORLD,&request);
printf("Send returned immediately\n");
MPI_Wait(request, status)
x=x+10;
}
```

Communication Modes

- **Standard Mode :** Calls block until message has been either transferred or copied to an internal buffer for later delivery. Ex: MPI_Send() and MPI_Recv()
- **Buffered Mode :** Send may start and return before a matching receive. MPI_Bsend()
- **Synchronous Mode :** Call blocks until matching receive has been posted and the message reception has started. MPI_Ssend()
- **Ready Mode :** Requires that a matching receive is already posted. MPI_Rsend().

Buffered Mode



MPI_BUFFER_ATTACH(buffer, size)	
buffer	initial buffer address (choice)
size	buffer size, in bytes (integer)

NOTE: A user may specify a buffer to be used for buffering messages sent in buffered mode.

Image Reference: https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/communication-modes

Synchronous Mode

We see that the data is not copied to system buffer.

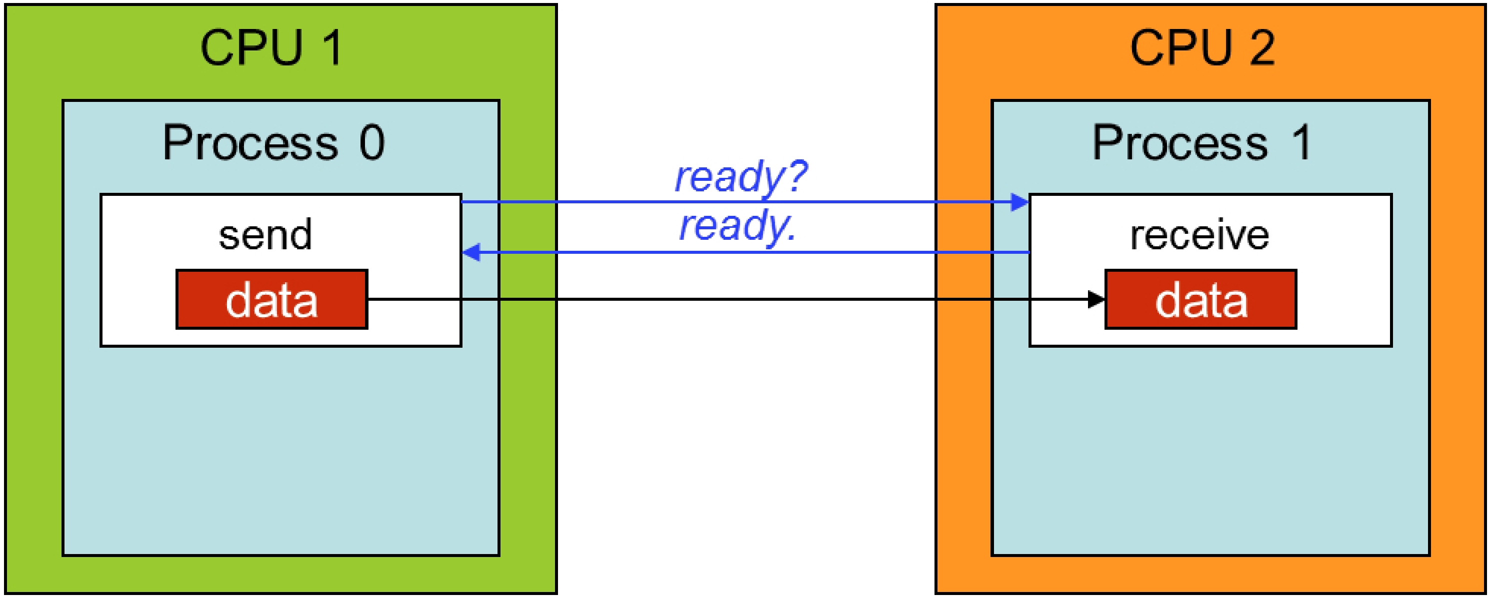


Image Reference: https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/communication-modes

Ready Mode

We make use of MPI_Barrier() to wait for the receive to be posted. This will not result in error.

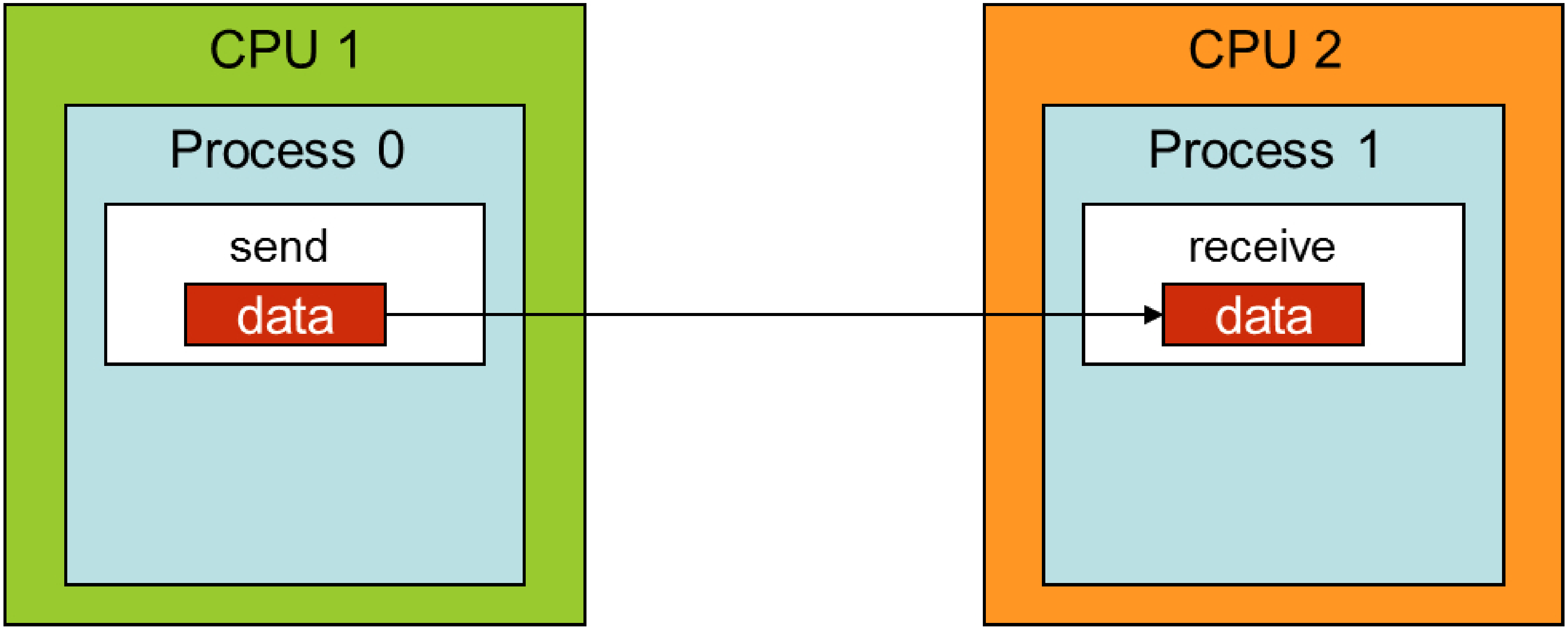


Image Reference: https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/communication-modes

MPI-Example - 3

```
if(myrank==0) {

//Blocking send will expect matching receive at the destination In Standard mode,Send will return after copying the data to the buffer

   MPI_Send(x,10,MPI_INT,1,1,MPI_COMM_WORLD);

// This send will be initiated and matching receive is already there so the program will not lead to deadlock

  MPI_Send(y,10,MPI_INT,1,2,MPI_COMM_WORLD);
}
```

```
else if(myrank==1)
{
//P1 will block as it has not received a matching send with tag 2

  MPI_Recv(x,10,MPI_INT,0,2,MPI_COMM_WORLD,&status);
  MPI_Recv(y,10,MPI_INT,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);


}
```

MPI Example 3

PROCESS 1

MPI_Send(x,10,..1,1,..);

MPI_Send(y,10,..,1,2,..);

PROCESS 2

MPI_Recv(x,10,..,0,2,..,..);

BLOCK

MPI_Recv(y,10,..,0,1,..,..);

MPI Example - 4

```
if(myrank==0) {

   MPI_Ssend(x,10,MPI_INT,1,1,MPI_COMM_WORLD);

   MPI_Send(y,10,MPI_INT,1,2,MPI_COMM_WORLD);
}
else if(myrank==1)
{
   MPI_Recv(x,10,MPI_INT,0,2,MPI_COMM_WORLD,&status);

MPI_Recv(y,10,MPI_INT,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

}
```

MPI Example - 4

if(myrank==0) {

MPI_Ssend(x,10,MPI_INT,1,1,MPI_COMM_WORLD);

// Synchronous Blocking send will expect matching receive at the destination. This results in deadlock.

*MPI_Send(y,10,MPI_INT,1,**2**,MPI_COMM_WORLD);* //This call will not be executed

}

else if(myrank==1)

{

*MPI_Recv(x,10,MPI_INT,0,**2,**MPI_COMM_WORLD,&status);* **//**P1 will block as it has not received a matching send with tag 2

MPI_Recv(y,10,MPI_INT,0,**1**,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

}

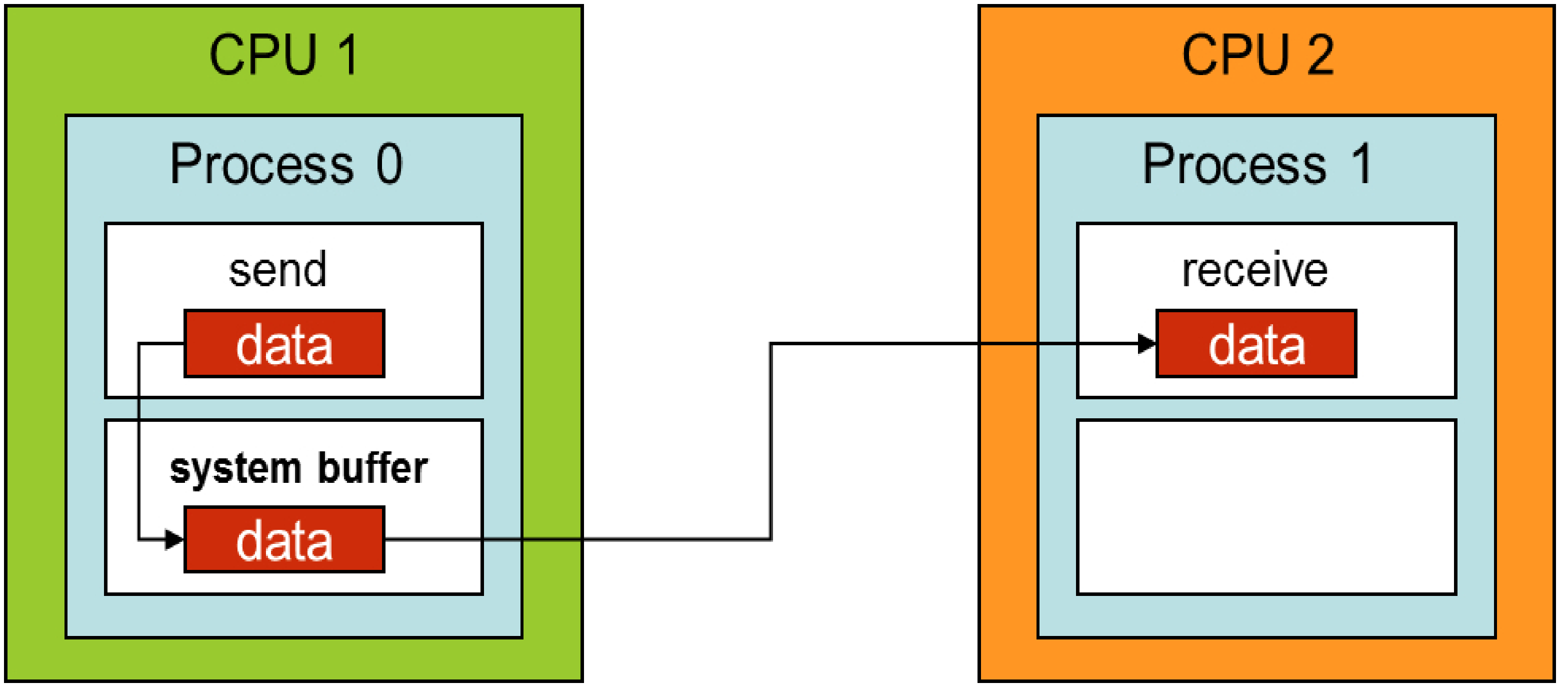
Outline

- Distributed Memory Architecture
- Introduction to MPI
- Structure of MPI program
- Types of Message Passing
- Basic Routines in Point to Point Communication
- Example programs on Point to Point Communication
- Basic Routines in Collective Communication
- Sample Programs on Collective Communication

Communication Modes

- **Standard Mode :** Calls block until message has been either transferred or copied to an internal buffer for later delivery. Ex: MPI_Send() and MPI_Recv()
- **Buffered Mode :** Send may start and return before a matching receive. MPI_Bsend()
- **Synchronous Mode :** Call blocks until matching receive has been posted and the message reception has started. MPI_Ssend()
- **Ready Mode :** Requires that a matching receive is already posted. MPI_Rsend().

Buffered Mode



MPI_BUFFER_ATTACH(buffer, size)	
buffer	initial buffer address (choice)
size	buffer size, in bytes (integer)

NOTE: A user may specify a buffer to be used for buffering messages sent in buffered mode.

Image Reference: https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/communication-modes

Synchronous Mode

We see that the data is not copied to system buffer.

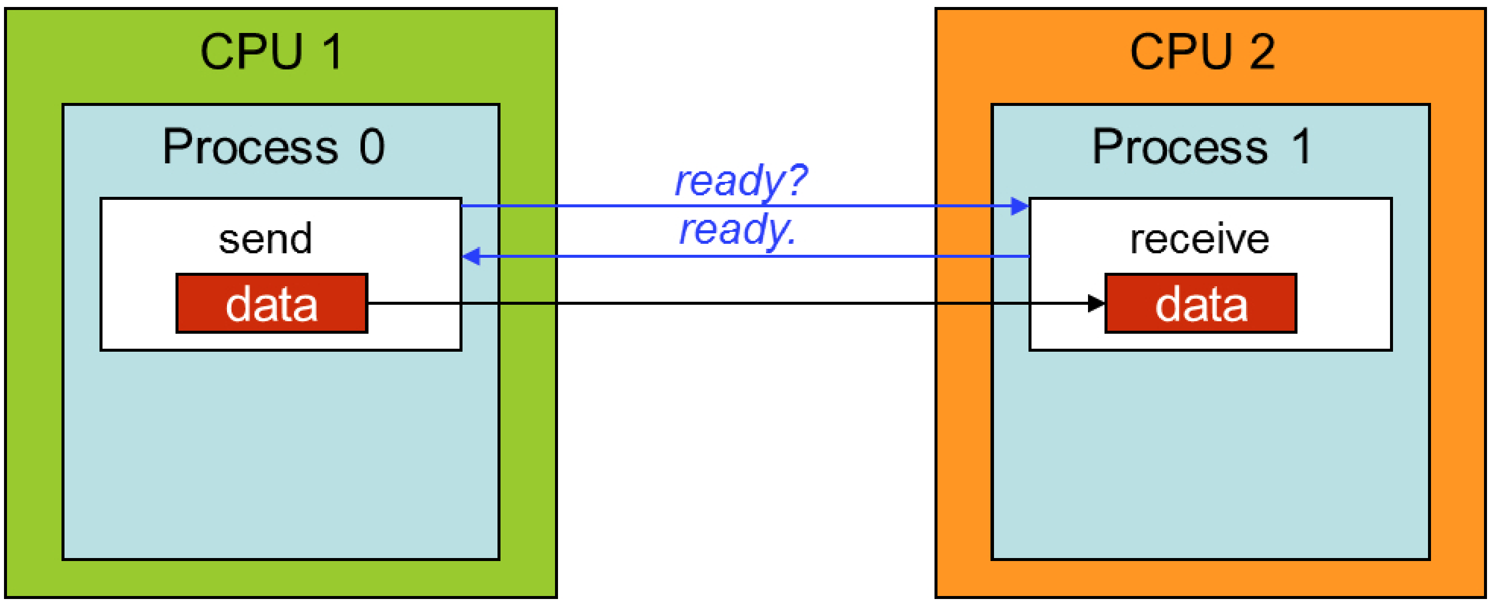


Image Reference: https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/communication-modes

Ready Mode

We make use of MPI_Barrier() to wait for the receive to be posted. This will not result in error.

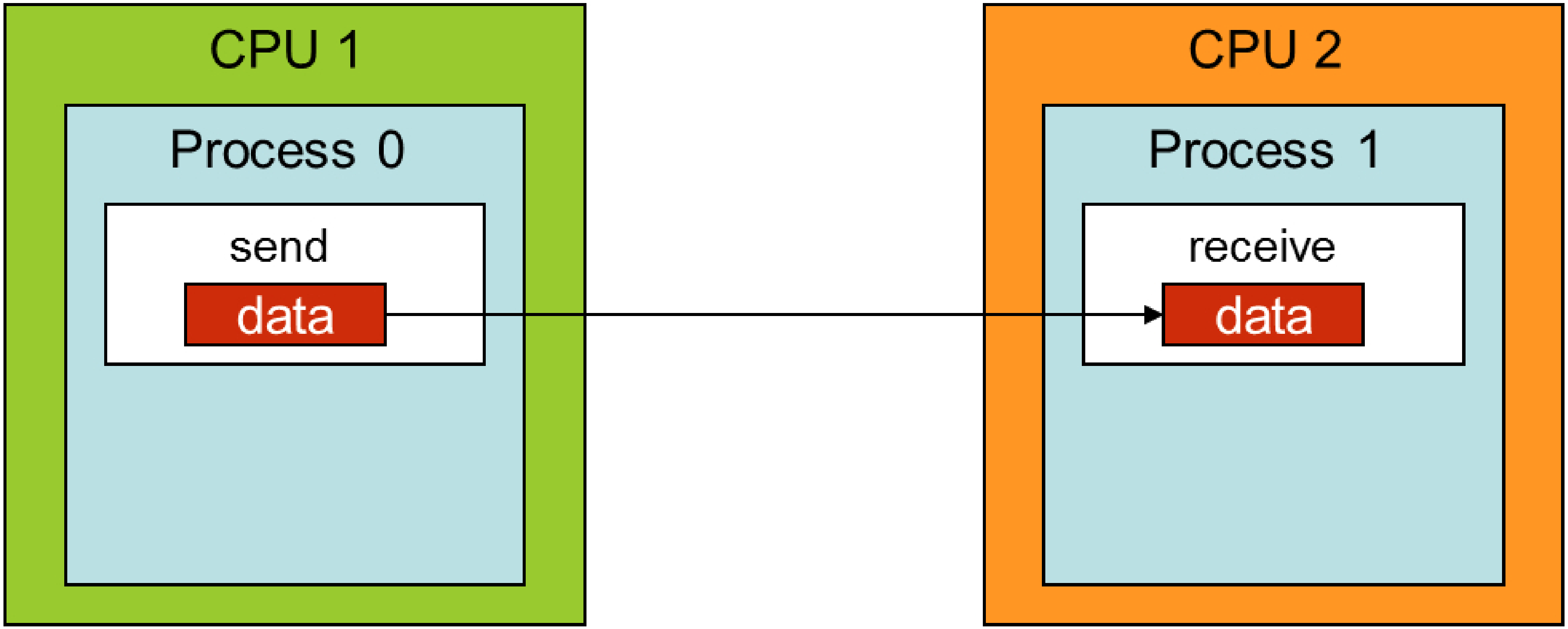


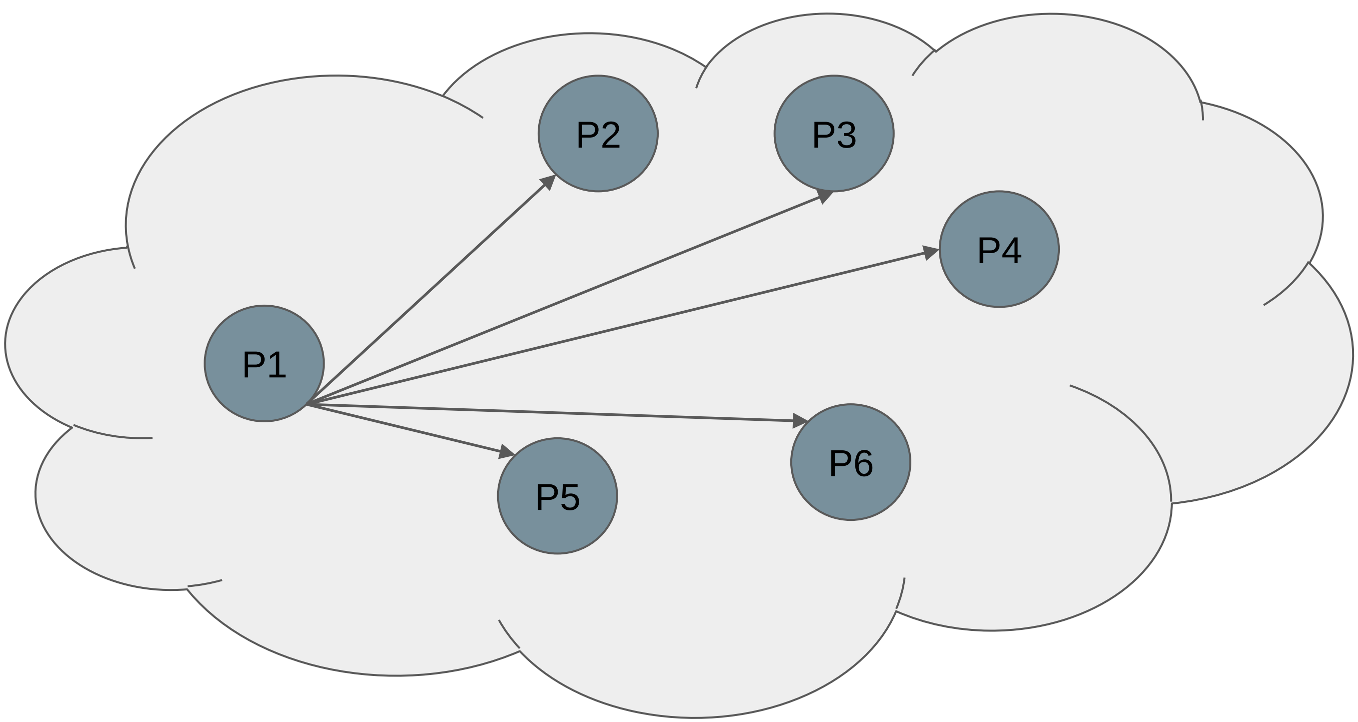
Image Reference: https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/communication-modes

Outline

- Distributed Memory Architecture
- Introduction to MPI
- Structure of MPI program
- Types of Message Passing
- Basic Routines in Point to Point Communication
- Example programs on Point to Point Communication
- **Basic Routines in Collective Communication**
- **Sample Programs on Collective Communication**

Collective Communication

- **Multiple processes in same communicator involve in collective communication.**
- **They are blocking calls.**
- **No tags required.**



Collective Communication

- Barrier
- Broadcast
- Scatter
- Gather
- Reduce
- Scatterv
- Gatherv

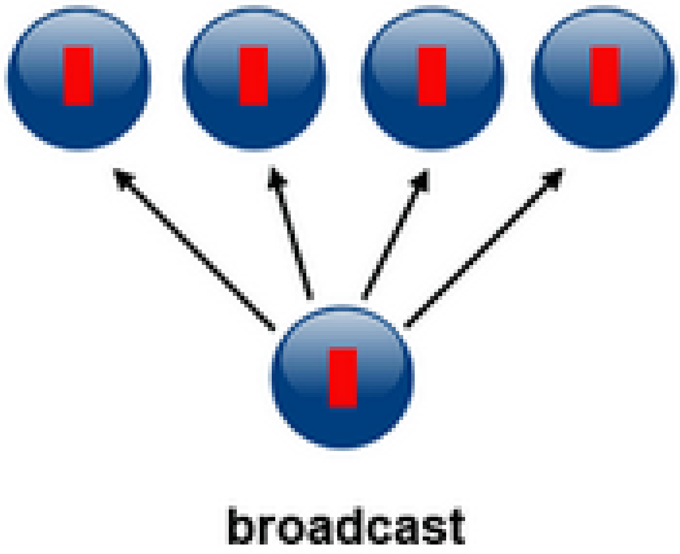
Collective communication: MPI_Barrier

- **Mainly used for synchronization**
- **The call returns only after all the processes have called Barrier function.**
- **Uses:**
  - Access to files
  - Achieve consistency

Syntax: MPI_Barrier(MPI_COMM_WORLD)

Collective Communication: Broadcast

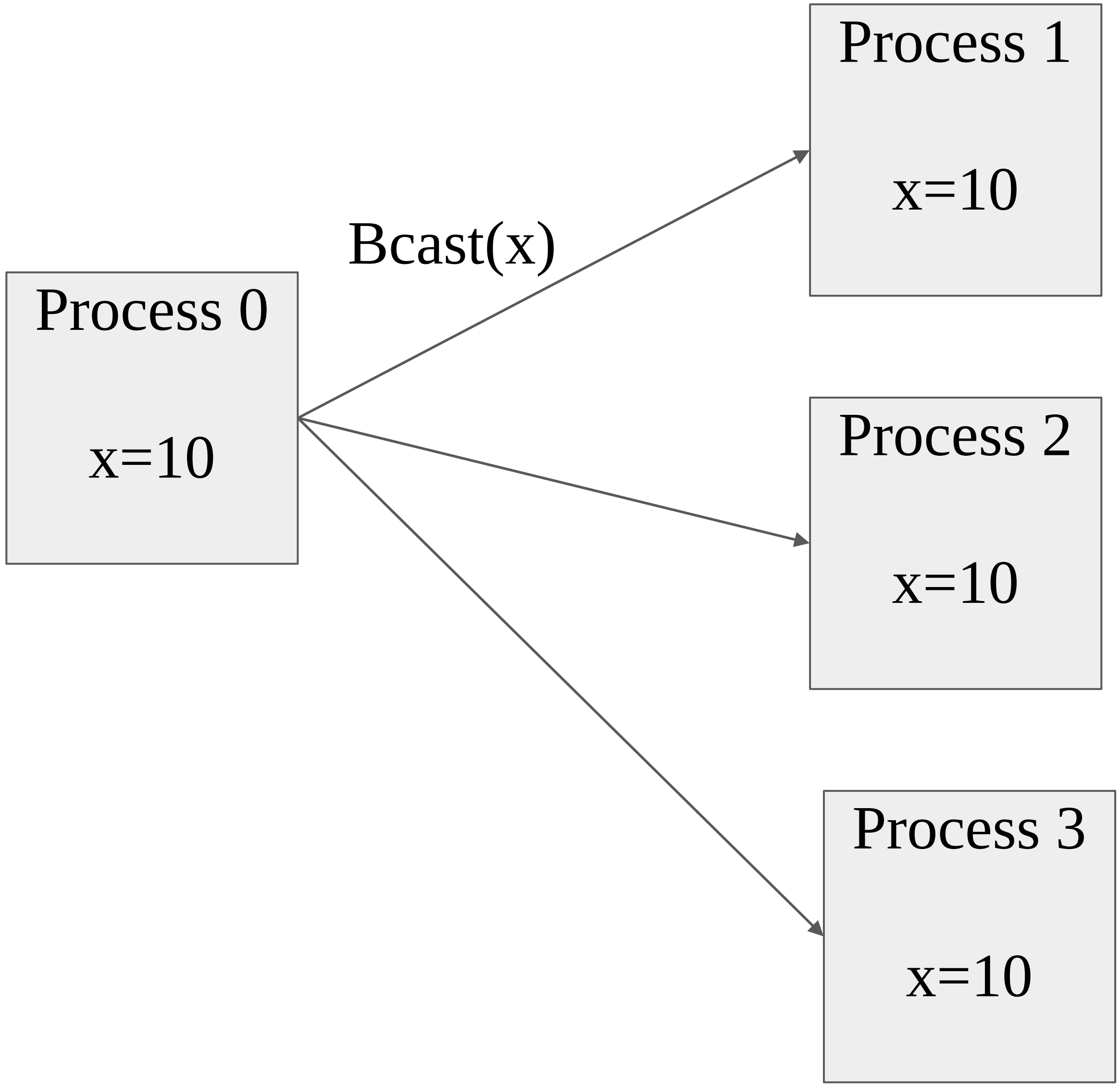
- **MPI_Bcast(buf, count, datatype, source, comm)**
  - buf : send buffer of sender and receive buffer of receiver
  - source : process which sends data to others



MPI Example - 5

```
if(myrank==0)
{
scanf("%d",&x);
}
MPI_Bcast(&x,1,MPI_INT,0,MPI_COMM_WORLD);
printf("Value of x in process %d : %d\n",myrank,x);
MPI_Finalize();
return 0;
}
```

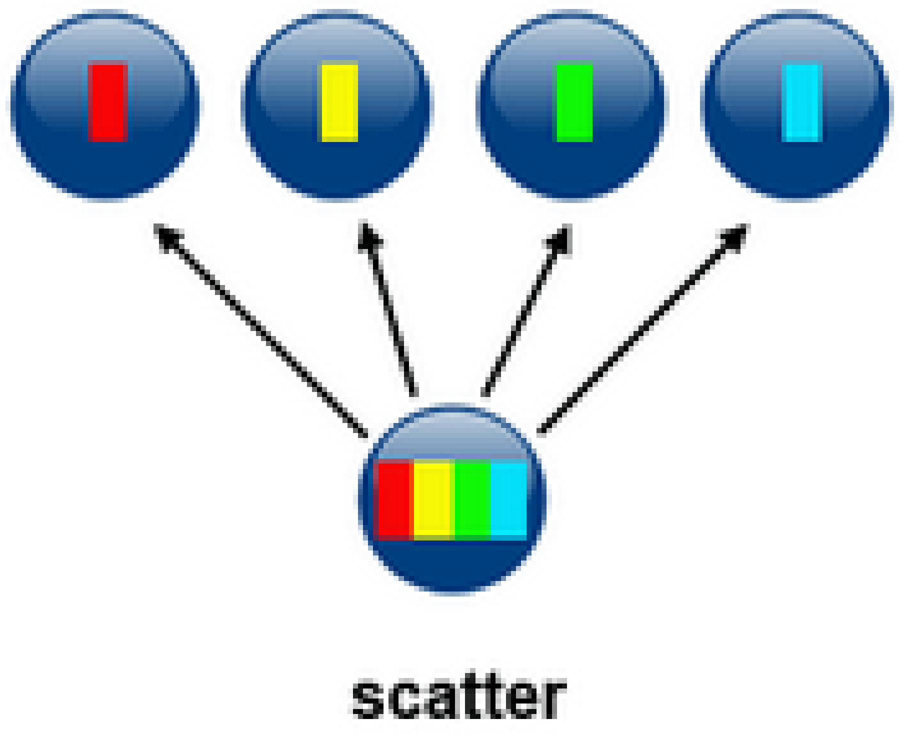
Bcast():


Bcast(x)
Process 0
x=10
Process 1
x=10
Process 2
x=10
Process 3
x=10

Broadcast Output:

```
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 4 ./a.out
3
Value of x in process 0 : 3
Value of x in process 1 : 3
Value of x in process 2 : 3
Value of x in process 3 : 3
```

Collective Communication: Scatter



Collective Communication: Scatter

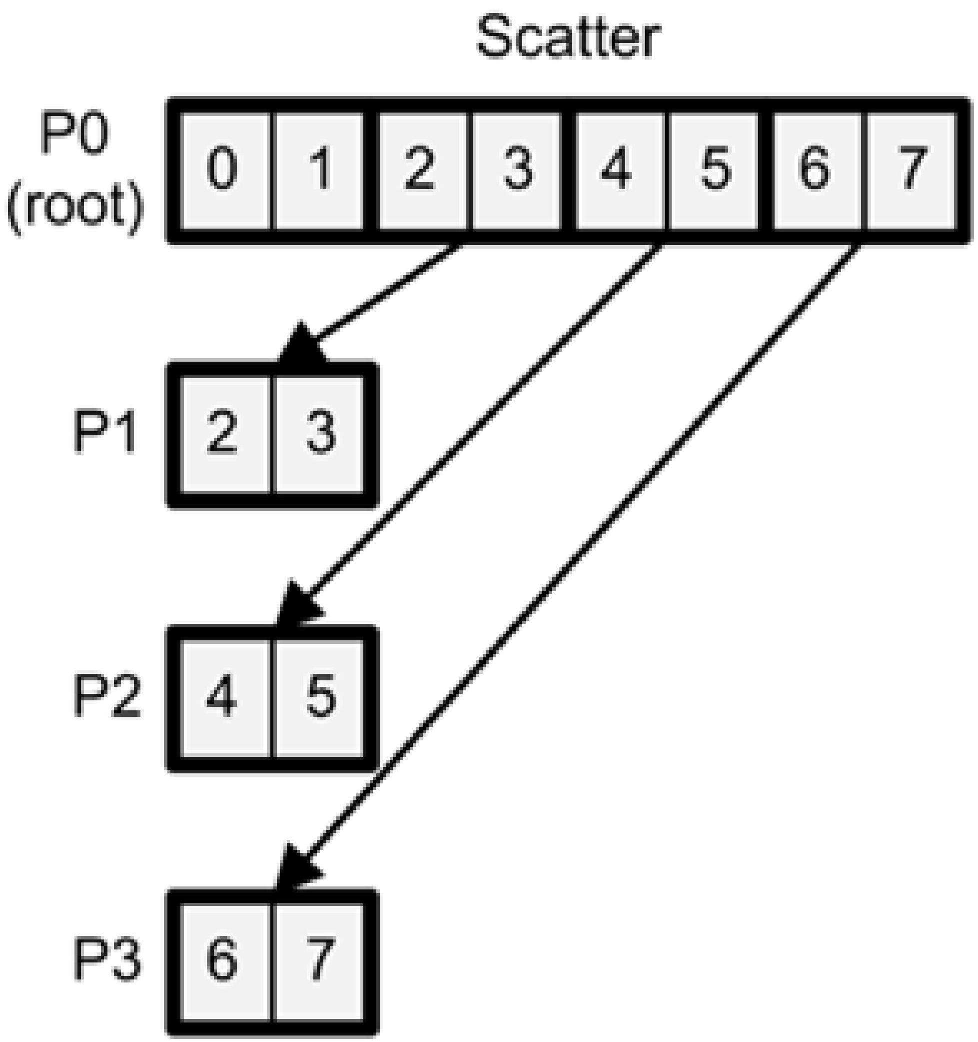
MPI_Scatter(sendbuf, sendcount, datatype, recvbuf, recvcount, datatype, root, comm)

Parameters:

- sendbuf : sender buffer
- sendcount : specify the number of elements to be sent. recvcount should be same as sendcount
- recvbuf : recv buffer
- root : Sender

MPI_Scatter

Example:



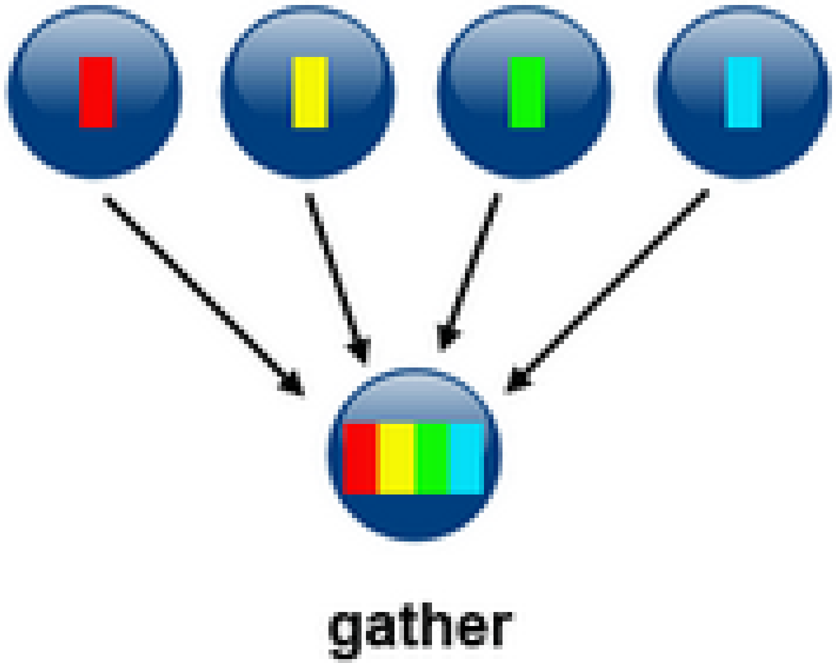
MPI Example - 6

```
if(myrank==0)
{
printf("Enter values into array x:\n");
for(i=0;i<8;i++)
scanf("%d",&x[i]);
}
MPI_Scatter(x,2,MPI_INT,y,2,MPI_INT,0,MPI_COMM_WORLD);
for(i=0;i<2;i++)
printf("\nValue of y in process %d : %d\n",myrank,y[i]);
```

Output

```
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 4 ./a.out
Enter values into array x:
1 2 3 4 5 6 7 8

Value of y in process 0 : 1

Value of y in process 0 : 2

Value of y in process 1 : 3

Value of y in process 1 : 4

Value of y in process 2 : 5

Value of y in process 2 : 6

Value of y in process 3 : 7

Value of y in process 3 : 8
```

Collective Communication: Gather



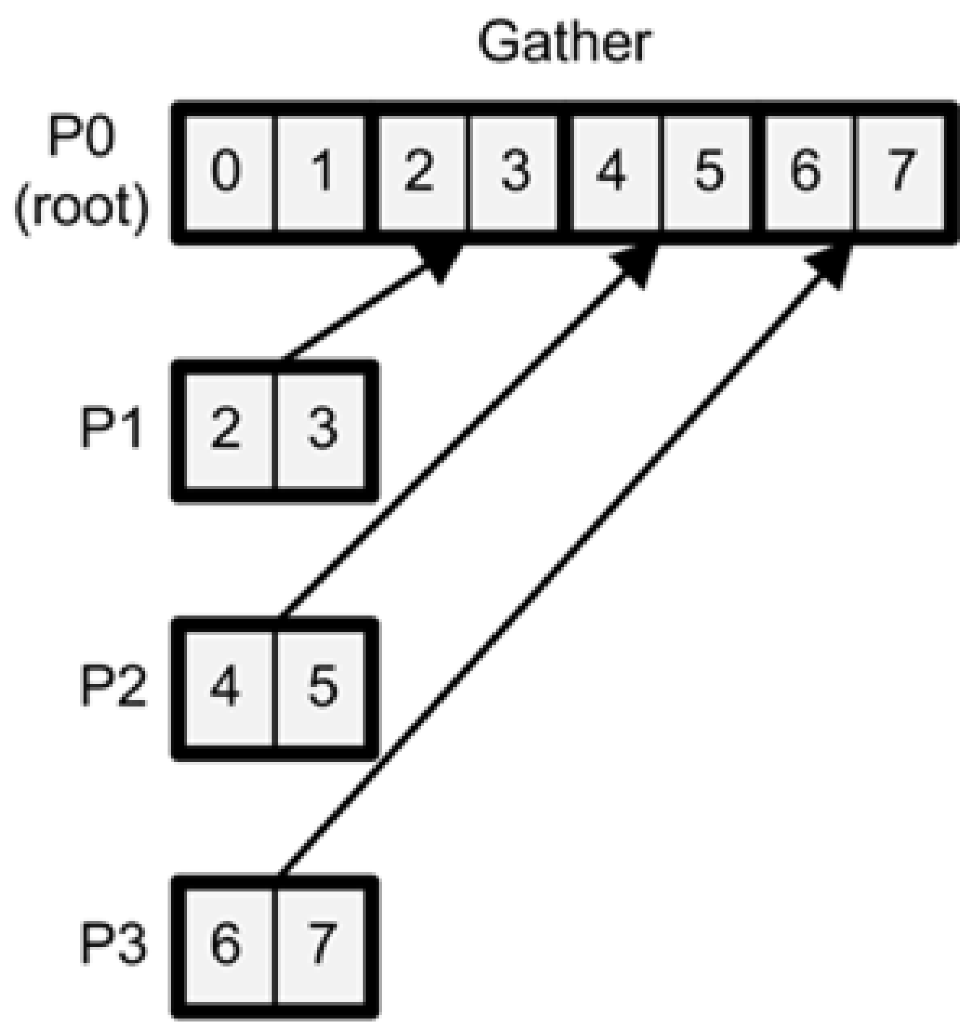
Collective Communication: Gather

MPI_Gather(sendbuf, sendcount, datatype, recvbuf, recvcount, datatype, root, comm)

Parameters:

- **sendbuf: buffer of sending processes**
- **sendcount and recvcount value is same**
- **recvbuf: root process's buffer**
- **root : process where the data is gathered**

MPI_Gather



MPI-Example 7

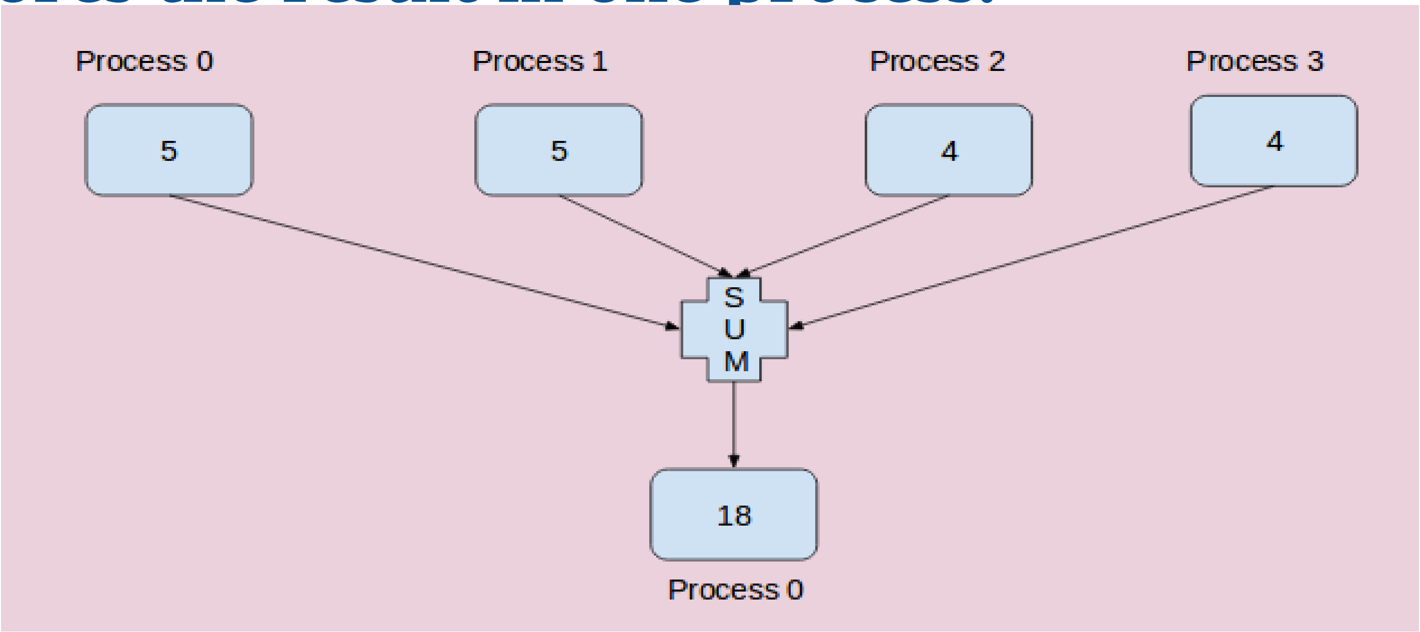
```
x=10, y[50]

MPI_Gather(&x,1,MPI_INT,y,1,MPI_INT,0,MPI_COMM_WORLD);
// Value of x at each process is copied to array y in Process 0
if(myrank==0)
{
for(i=0;i<size;i++)
printf("\nValue of y[%d] in process %d : %d\n",i,myrank,y[i]);
}
```

Output

```
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 4 ./a.out

Value of y[0] in process 0 : 10

Value of y[1] in process 0 : 10

Value of y[2] in process 0 : 10

Value of y[3] in process 0 : 10
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 6 ./a.out

Value of y[0] in process 0 : 10

Value of y[1] in process 0 : 10

Value of y[2] in process 0 : 10

Value of y[3] in process 0 : 10

Value of y[4] in process 0 : 10

Value of y[5] in process 0 : 10
```

Collective Communication: Reduce

- **Allows to perform computations on data present at multiple processes.**
- **Computations like : Sum, Product, Maximum, Minimum**
- **Stores the result in one process.**



Collective Communication: Reduce

MPI_Reduce(sendbuf, recvbuf, count, datatype, operation, dest, comm)

Parameters:

count: size of receive buffer

operation:

MPI name	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Summation
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_LOR	Logical OR
MPI_LXOR	Logical XOR

MPI Example - 8

```
x=myrank;
MPI_Reduce(&x,&y,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD)
;
if(myrank==0)
{
printf("Value of y after reduce : %d\n",y);
}
```

Output

```
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 3 ./a.out
Value of y after reduce : 3
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 4 ./a.out
Value of y after reduce : 6
```

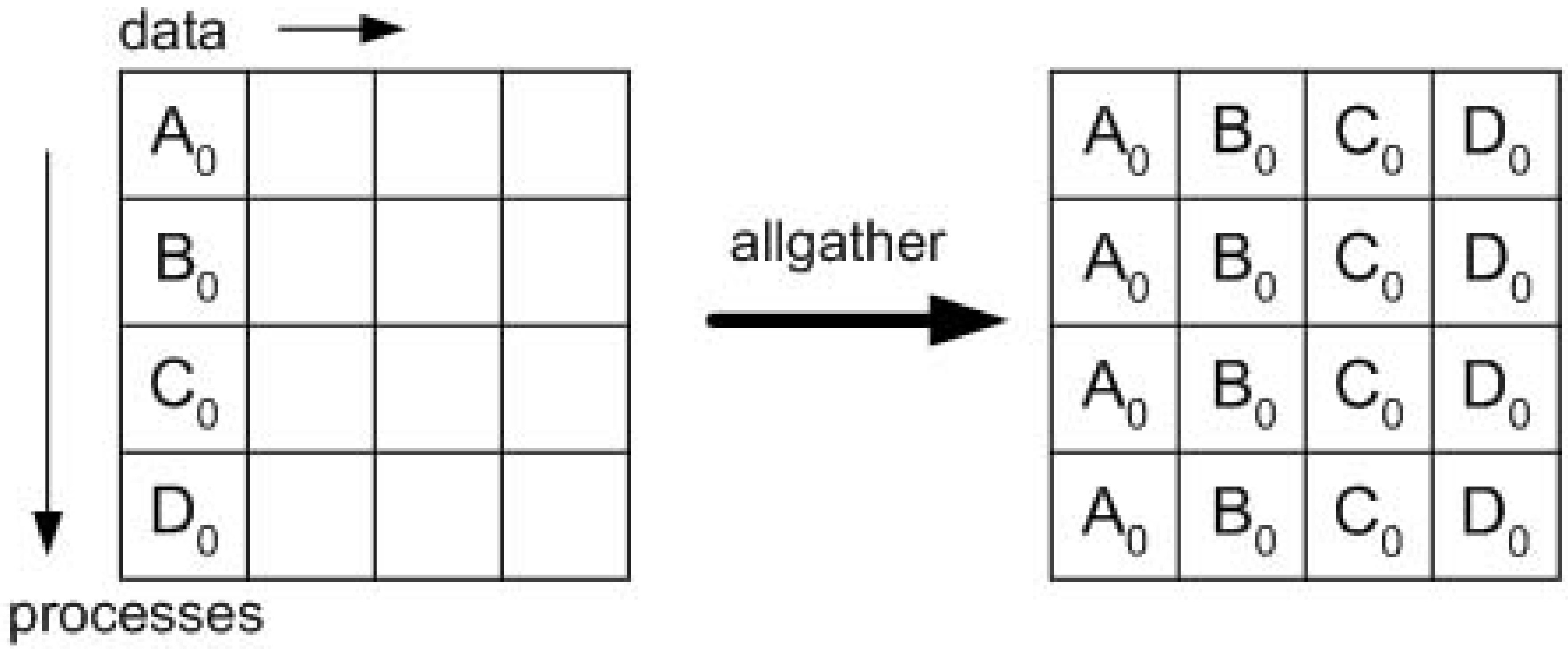
Outline

- Distributed Memory Architecture
- Introduction to MPI
- Structure of MPI program
- Types of Message Passing
- Basic Routines in Point to Point Communication
- Example programs on Point to Point Communication
- **Basic Routines in Collective Communication**
- **Sample Programs on Collective Communication**

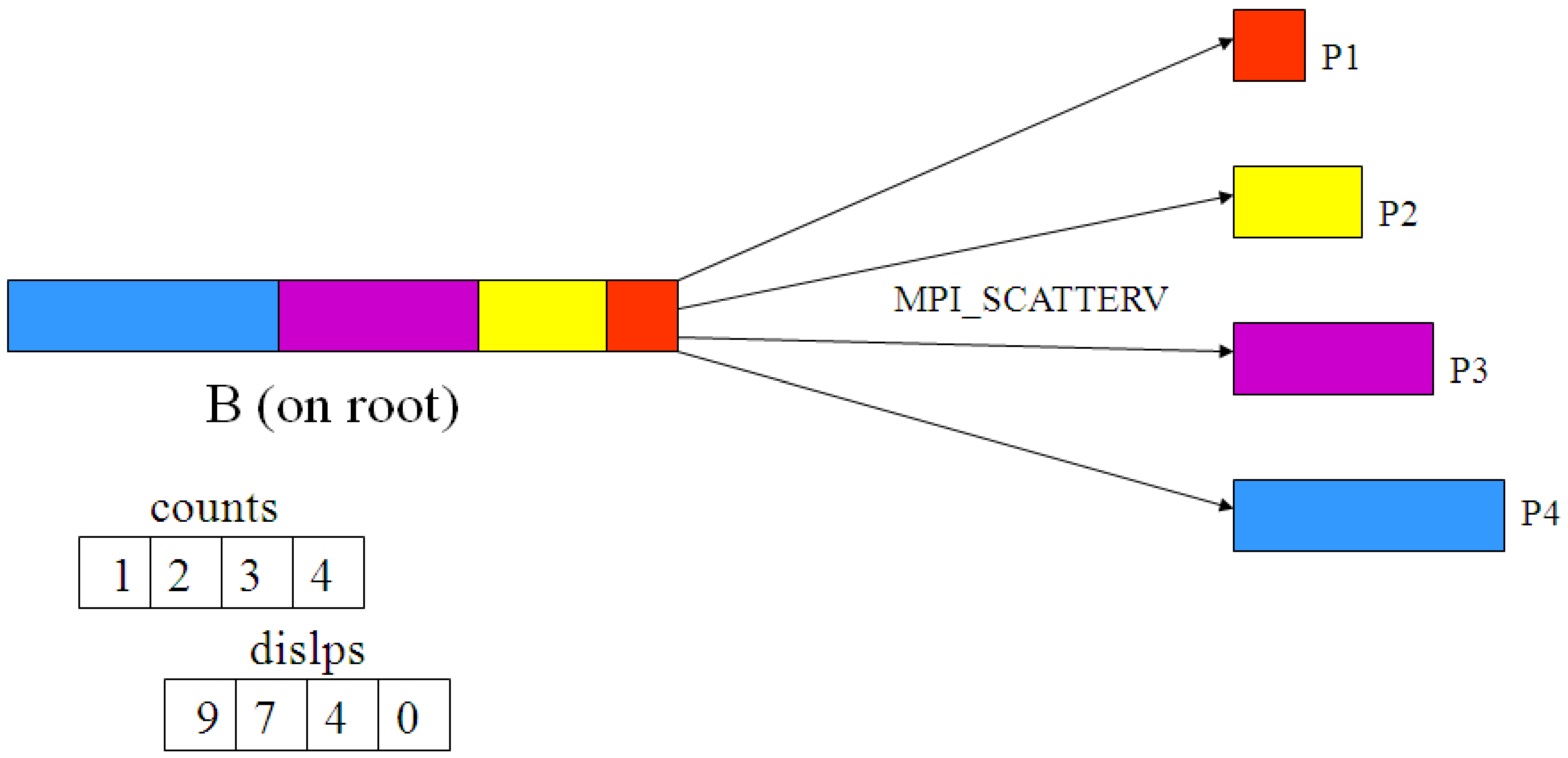
MORE Collective Communication Routines:

- MPI_Gatherv()
- MPI_Scatterv()
- MPI_Allgather
- MPI_AllReduce()
- MPI_Scan()
- MPI_Comm_Split()

MPI_Allgather



MPI_Scatterv



MPI_Scatterv():

MPI_Scatterv(sendbuf, sendcounts, displacement, datatype, recvbuf, recvcount, datatype, root, comm)

Parameters:

sendcounts : array with number of elements to be sent to each process. ex: sendcount[0]=10 means send 10 elements to Process zero. sendcount[1]=20 means send 20 elements to Process one.

displacement: array which holds the index from where the data is to be sent to each process. Ex: disp[0]=0 means Process zero gets elements starting with index zero. disp[1]=10 means Process 1 will get elements starting from index 10.

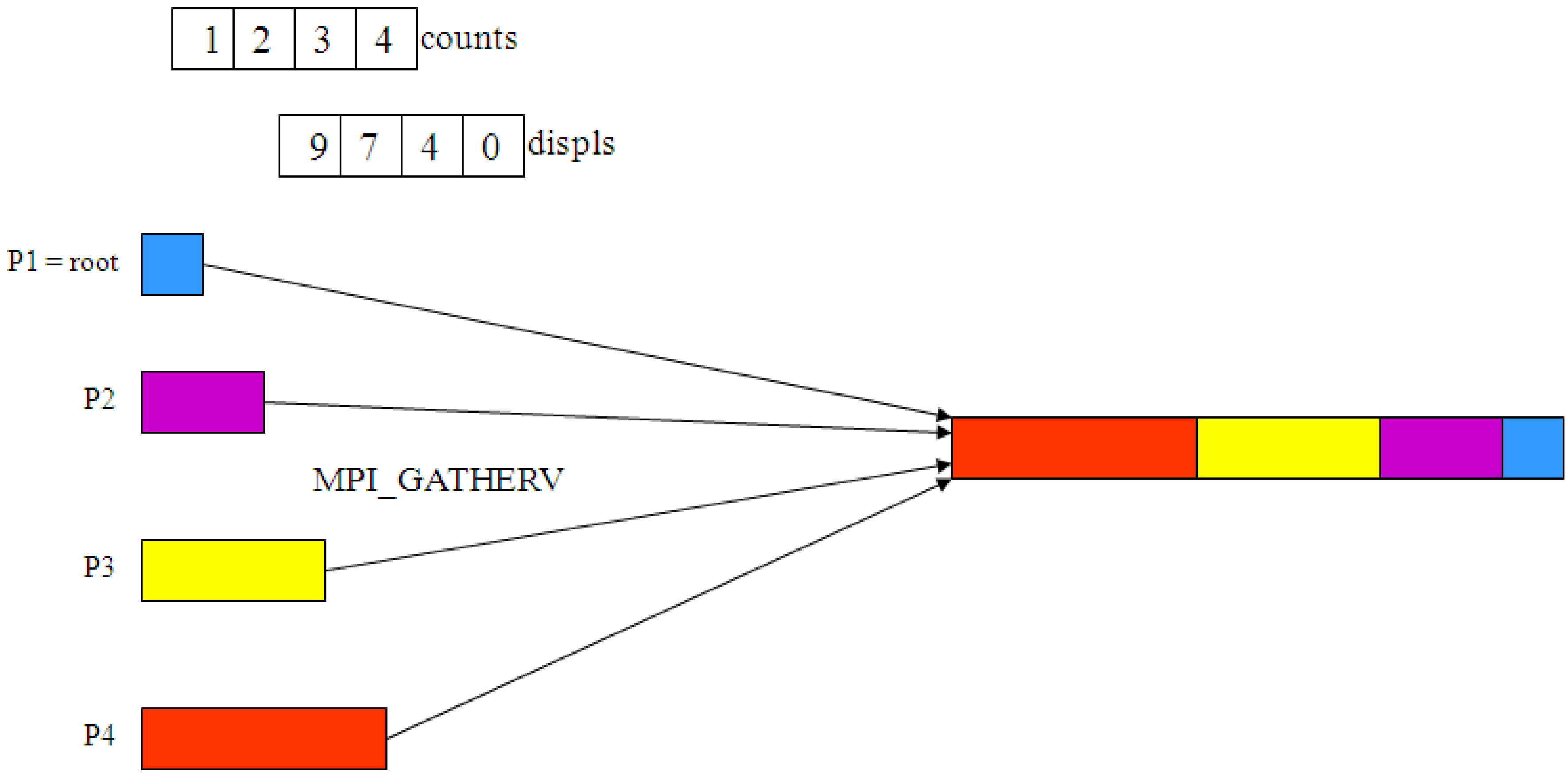
MPI_Scatterv

```
if(myrank==0)
        {
        printf("Enter the number of elements:\n");
        scanf("%d",&m);
        printf("Initializing array x:\n");
        for(i=0;i<m;i++)
        x[i]=i+1;
        disp[0]=0;
        for(i=0;i<size;i++)
        {
        z[i]=i+2; // Process 0 gets 2 elements, Process 1 gets 3, Process 2 gets 4 and so on
        disp[i+1]=disp[i]+z[i]; // disp[1]=2, disp[2]=5,disp[3]=9 and so on
        }
}

MPI_Scatterv(x,z,disp,MPI_INT,y,myrank+2,MPI_INT,0,MPI_COMM_WORLD);
```

OUTPUT:

```
Enter the number of elements:
20

Value of y in process 1 : 3

Value of y in process 1 : 4

Value of y in process 1 : 5

Value of y in process 2 : 6

Value of y in process 2 : 7

Value of y in process 2 : 8

Value of y in process 2 : 9

Value of y in process 3 : 10

Value of y in process 3 : 11

Value of y in process 3 : 12

Value of y in process 3 : 13

Value of y in process 3 : 14

Value of y in process 4 : 15

Value of y in process 4 : 16

Value of y in process 4 : 17

Value of y in process 4 : 18

Value of y in process 4 : 19

Value of y in process 4 : 20
Initializing array x:

Value of y in process 0 : 1

Value of y in process 0 : 2
```

MPI_Gatherv



MPI_Gatherv():

MPI_Gatherv(sendbuf, sendcount, datatype, recvbuf, recvcounts, displacements, datatype, root, comm)

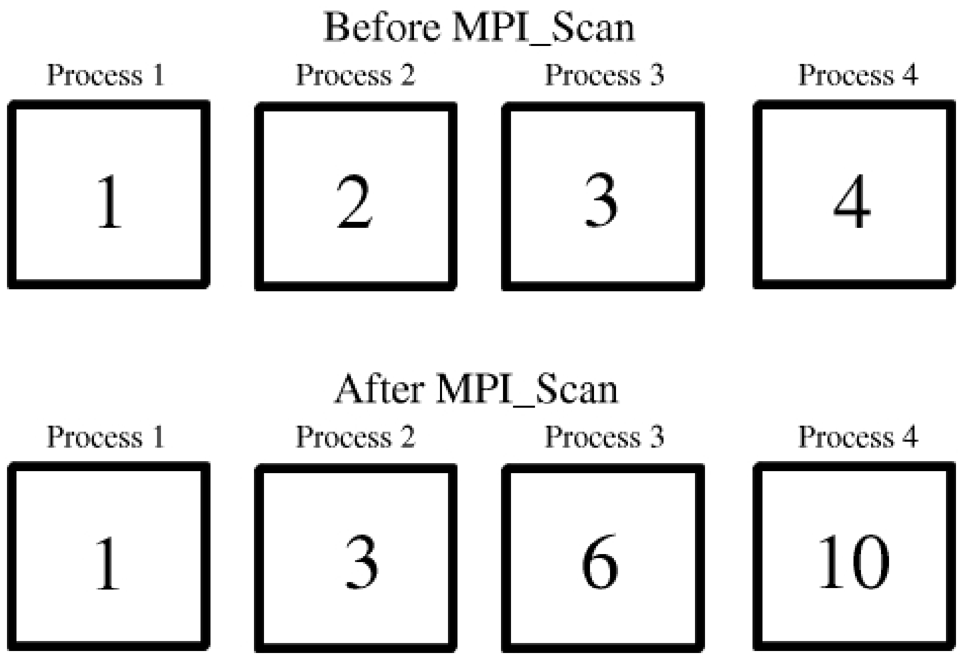
Parameters:

recvcounts : array with number of elements to be received from each process.

displacement: array which holds the beginning index where the data is to be received from each process.

MPI_Scan

int MPI_Scan(sendbuf, recvbuf, int count, datatype, MPI_Op, comm)



MPI_Comm_Split : Split the communication Domain

MPI_Comm_Split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm);

color: controls subset assignment

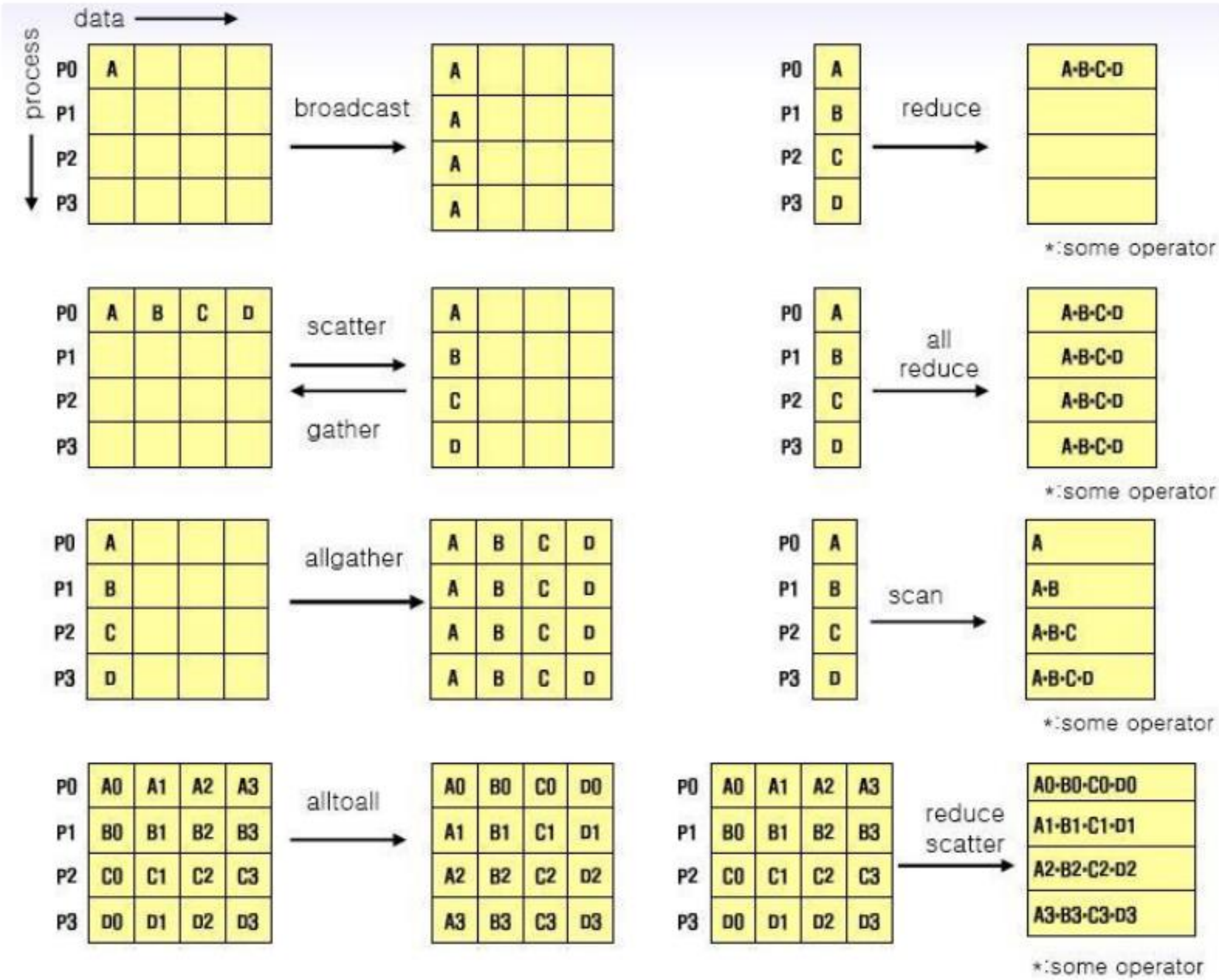
key: controls rank assignment of processes in different group

Ex: MPI_Comm_split(MPI_COMM_WORLD,0,0,&comm1);

MPI_Comm_split(MPI_COMM_WORLD,1,0,&comm2);

MPI_Comm_split(MPI_COMM_WORLD,2,0,&comm3);

MPI Collective Routine



Reference: Introduction to MPI and OpenMP (with Labs) Brandon Barker Computational Scientist Cornell University Center for Advanced Computing (CAC) . https://www.cac.cornell.edu/

Summary

- MPI provides a simplified way for sending and receiving messages
- MPI rich set of collective functions
- MPI helps for developing Scalable and Portable Parallel Programs
- MPI is the defacto standard for Distributed Memory Parallelism

Thank You