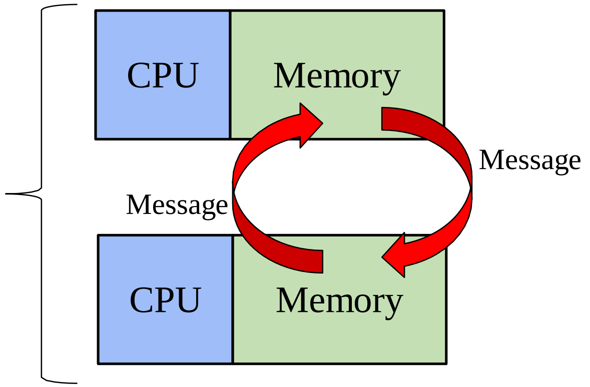
Introduction to MPI Programming
Dept. of MACS NITK

MPI and OpenMP

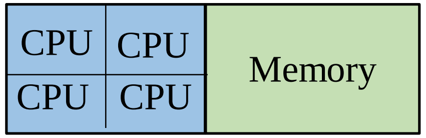
- MPI – Designed for distributed memory
  - Multiple systems
    - Send/receive messages

MPI



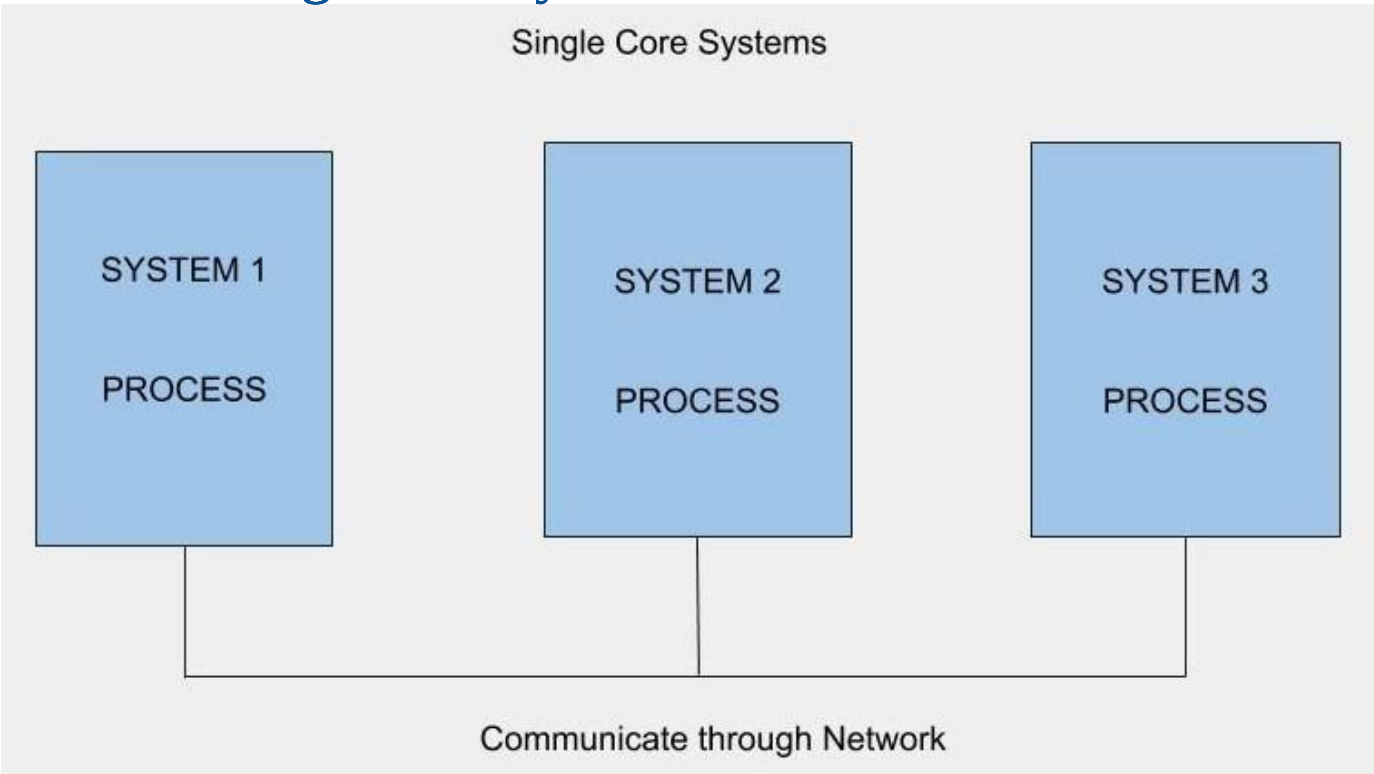
- OpenMP – Designed for shared memory
  - Single system with multiple cores
  - Sharing memory

OpenMP



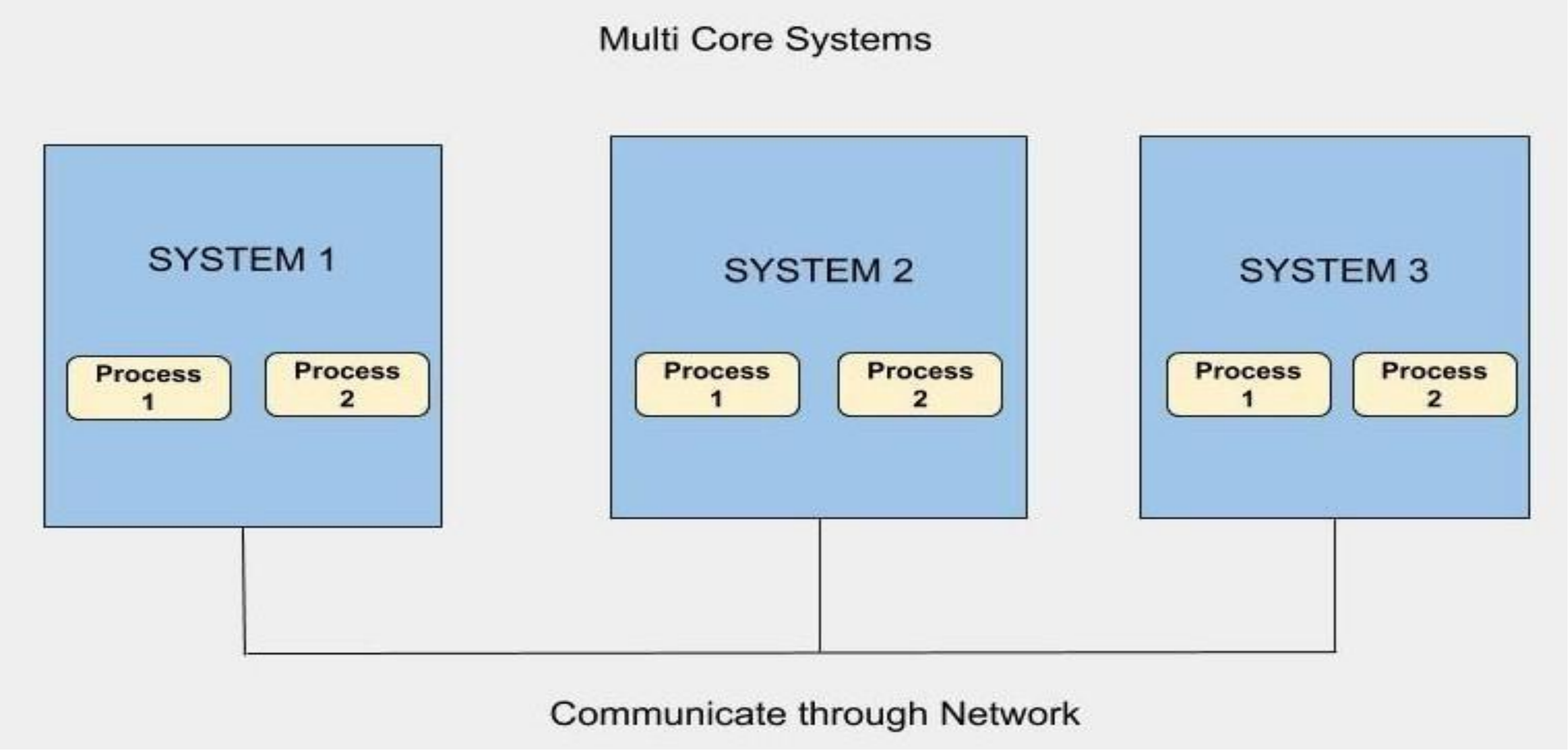
DISTRIBUTED ARCHITECTURE

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



DISTRIBUTED ARCHITECTURE

- Systems with multiple core communicating through shared and distributed memory



MPI

- Message Passing Interface

- Multiple implementations exist
  - Open MPI
  - MPICH
  - Many commercial (Intel, HP, C-DAC etc..)
  - Difference should only be in the compilation not development

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();

Basic Environment

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

Communicators & Rank

- MPI uses objects called communicators
  - Defines which processes can talk
  - Communicators have a size
- MPI_COMM_WORLD
  - Predefined as ALL of the MPI Processes
  - $Size = N_{procs}$
- Rank
  - Integer process identifier
  - $0 \le Rank < Size$

Basic Environment Cont.

```
MPI_Comm_rank(comm, &rank)
```

- Returns the rank of the calling MPI process
- Within the communicator, comm
  - MPI_COMM_WORLD is set during Init(…)
  - Other communicators can be created if needed

```
MPI_Comm_size(comm, &size)
```

- Returns the total number of processes
- Within the communicator, comm

```
int my_rank, size;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Hello World for MPI

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


int main (int argc, char *argv[]) {

  int rank, size;

  MPI_Init (&argc, &argv);  //initialize MPI library

  MPI_Comm_size(MPI_COMM_WORLD, &size); //get number of processes
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); //get my process id

  //do something
  printf ("Hello World from rank %d\n", rank);
  if (rank == 0) printf("MPI World size = %d processes\n", size);

  MPI_Finalize(); //MPI finish

  return 0;
}
```

To Run

With this number of processes

Start an mpi job

Run this executable

```
[user@chandra]$ mpiexec -n 4 ./hello_world_mpi
Hello World from rank 0
MPI World size = 4 processes
Hello World from rank 1
Hello World from rank 2
Hello World from rank 3
```

Hello World Output

- 4 processes

```
Hello World from rank 3
Hello World from rank 0
MPI World size = 4 processes
Hello World from rank 2
Hello World from rank 1
```

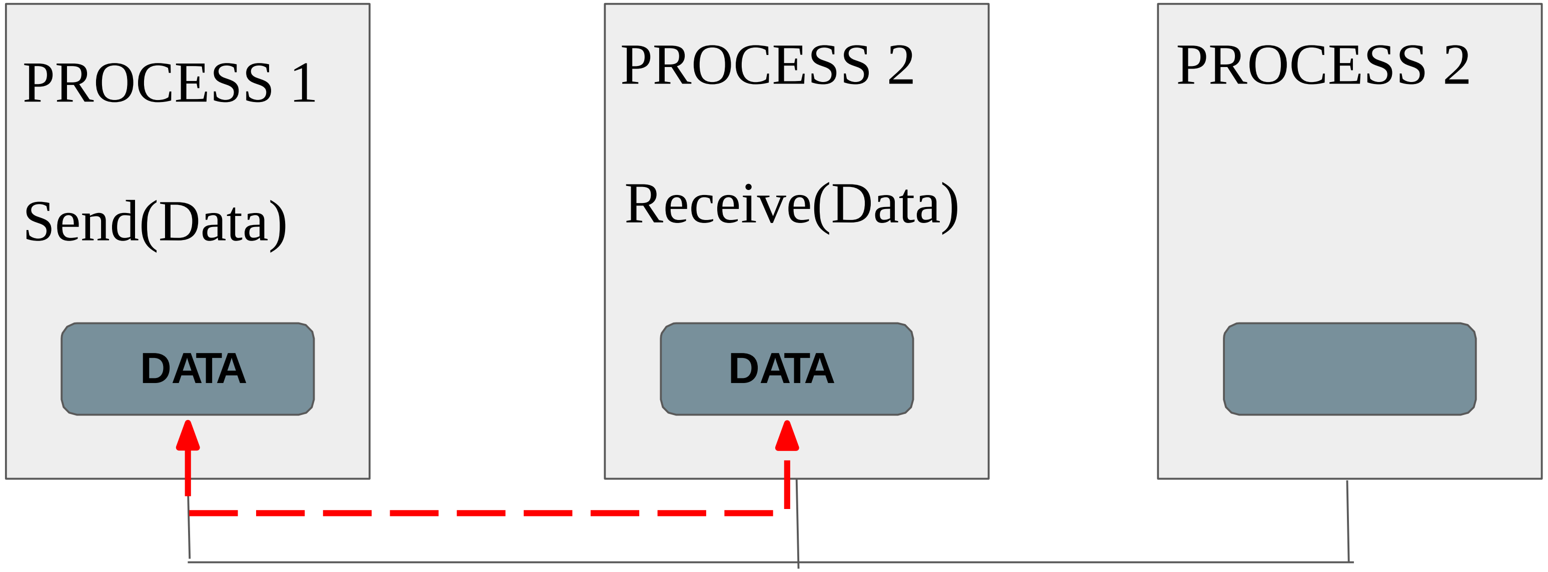
- Code ran on each process independently
- MPI Processes have *private* variables
- Processes can be on completely different machines

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, 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.

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()

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.

DATA TYPES

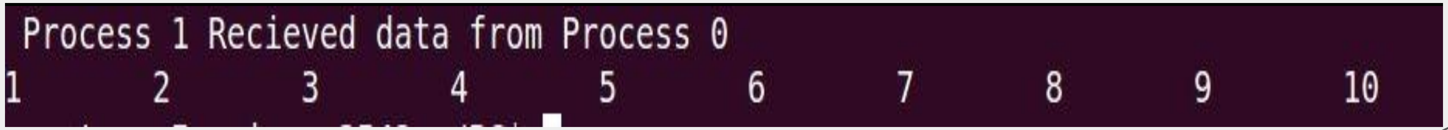
Table 1: Basic C datatypes in MPI

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	

MPI Example - 1

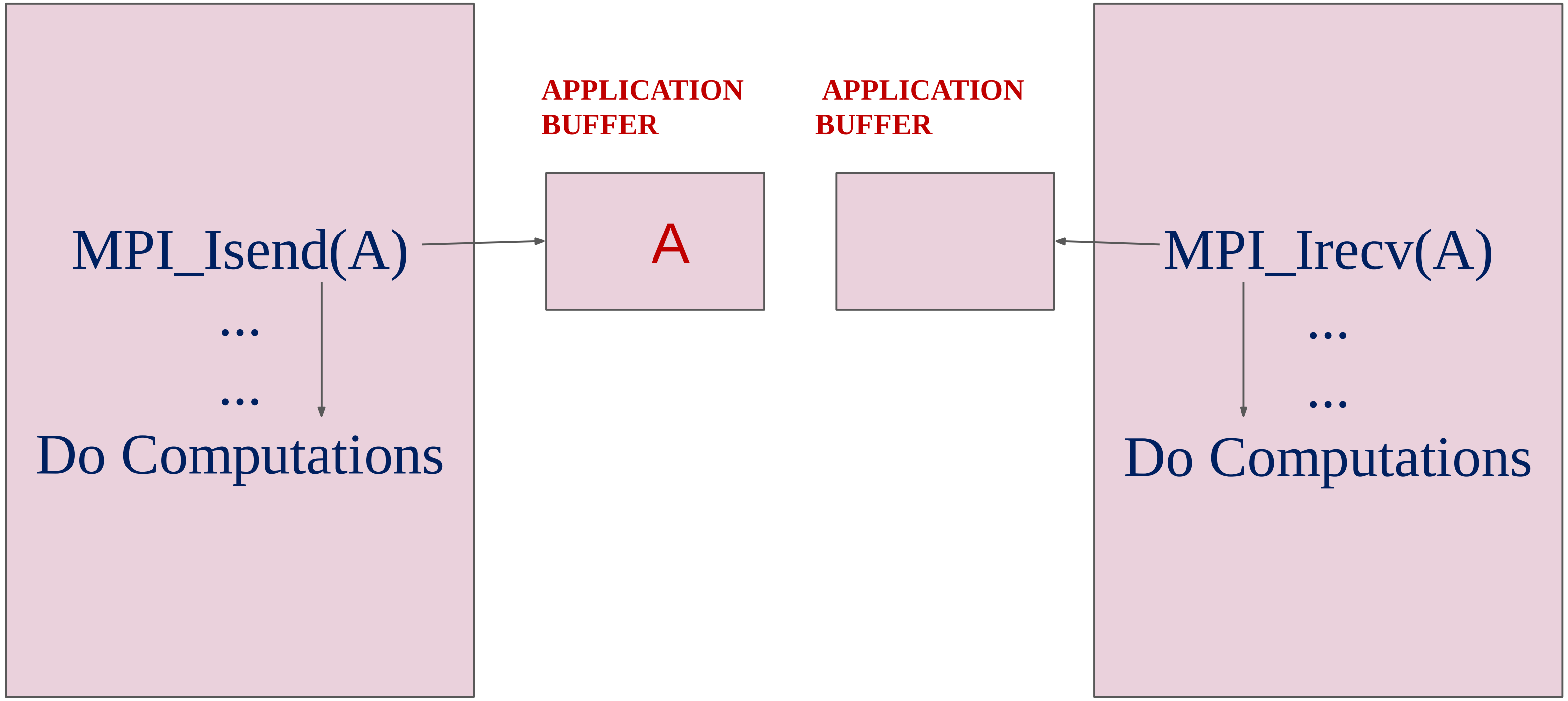
```
for(i=0;i<50;i++) //Process 0 initializes array x
x[i]=i+1; if(myrank==0)
MPI_Send(x,10,MPI_INT,1,1,MPI_COMM_WORLD);
else if(myrank==1)
{
MPI_Recv(y,10,MPI_INT,0,1,MPI_COMM_WORLD,&status);
printf("Process %d Received Data from Process %d\n",
myrank,status.MPI_SOURCE);
for(i=0;i<10;i++)

printf("%d\t",y[i]);
}
```



Non Blocking Send and Receive

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



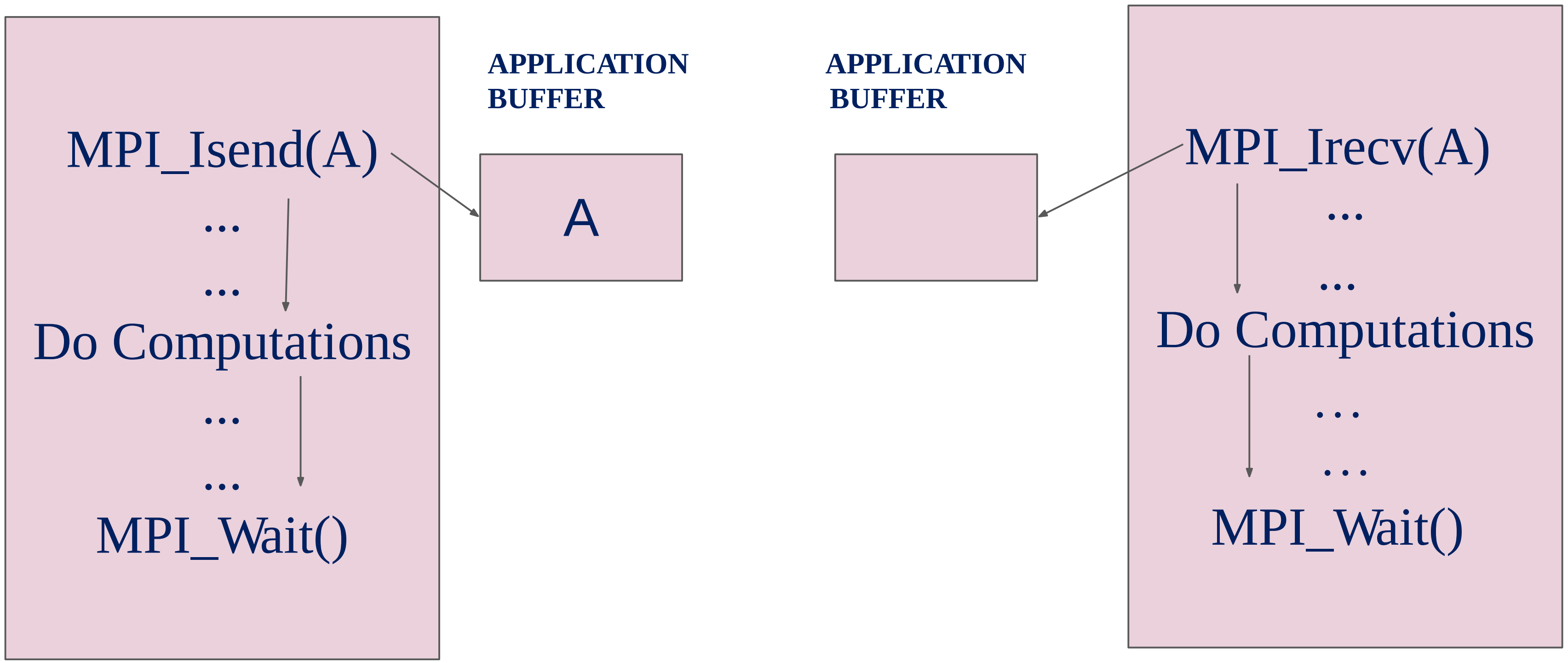
Non Blocking Send and Receive

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()**



MPI_Isend(A)
...
...
Do Computations
...
...
MPI_Wait()
APPLICATION BUFFER
A
APPLICATION BUFFER
MPI_Irecv(A)
...
...
Do Computations
...
...
MPI_Wait()

Syntax :

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

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

}
```

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().

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 - 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);

}

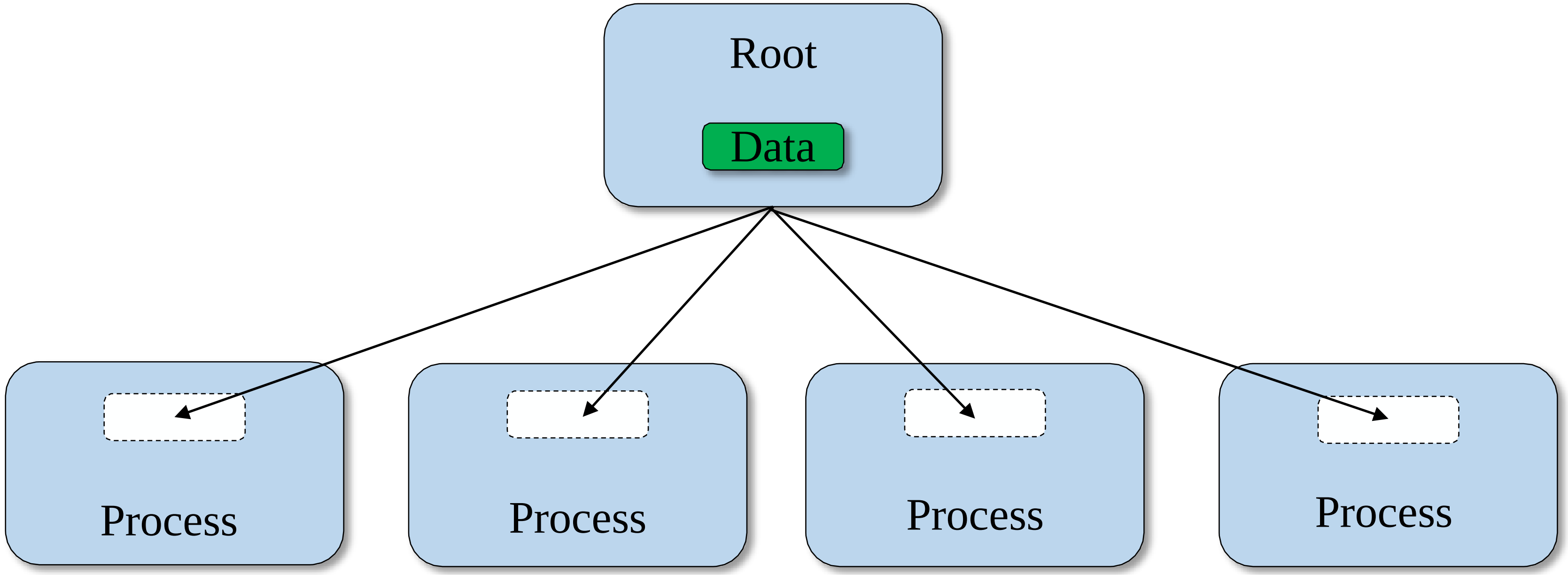
Collective Communication

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

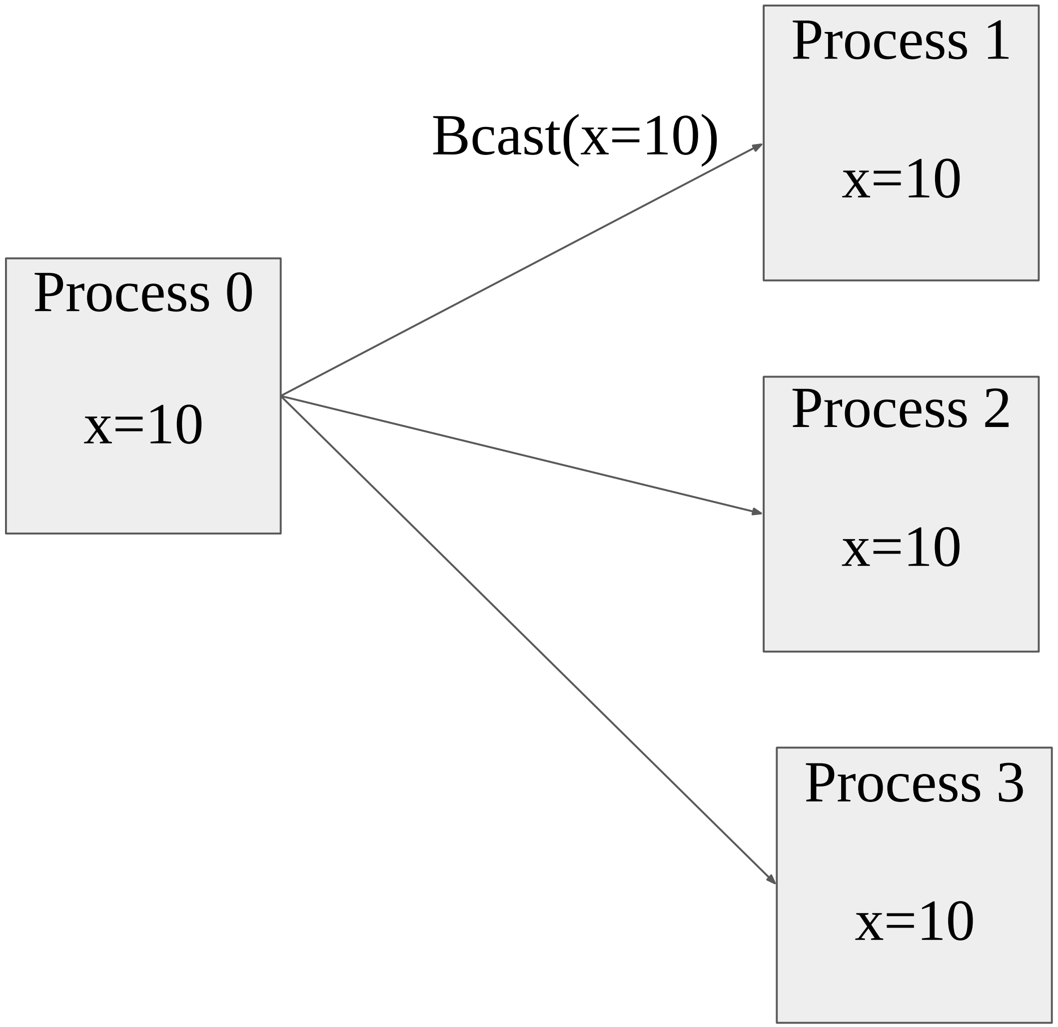
Collective Communication (Bcast)

```
MPI_Bcast(&buffer, count, datatype, root, comm)
```

- Broadcasts a message from the root process to all other processes
- Useful when reading in input parameters from file



Bcast():


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

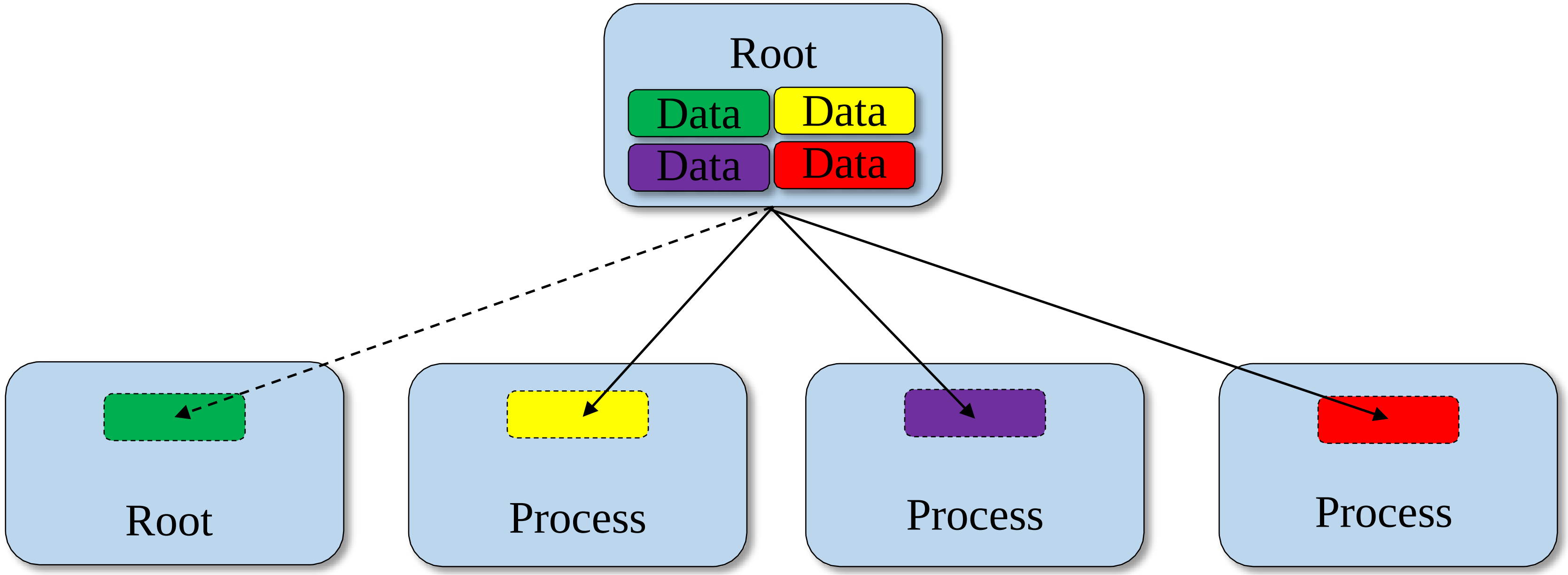
MPI Example - 5

```
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;
}
```

Collective Communication (Scatter)

```
MPI_Scatter(&sendbuf, sendcnt, sendtype, &recvbuf,
            recvcnt, recvtype, root, comm)
```

- Sends individual messages from the root process to all other processes



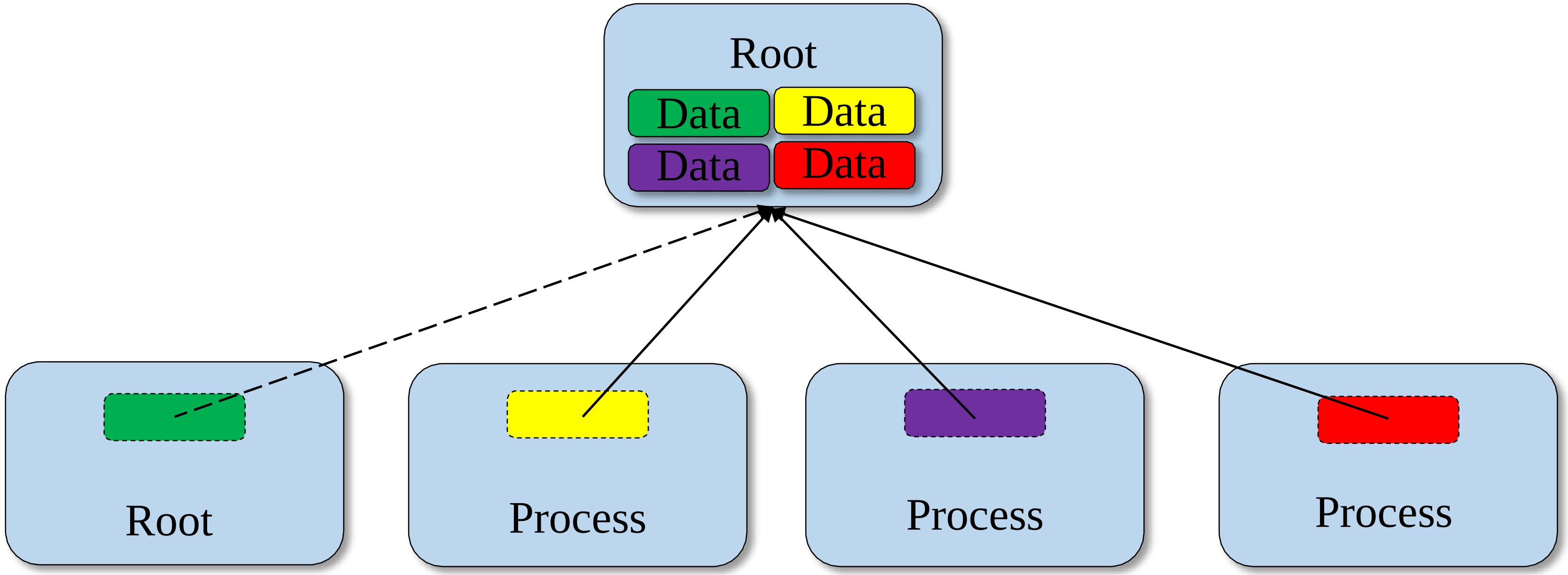
MPI Example - 6

```
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]);
```

Collective Communication (Gather)

```
MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf,
           recvcnt, recvtype, root, comm)
```

- Opposite of Scatter



MPI-Example 7

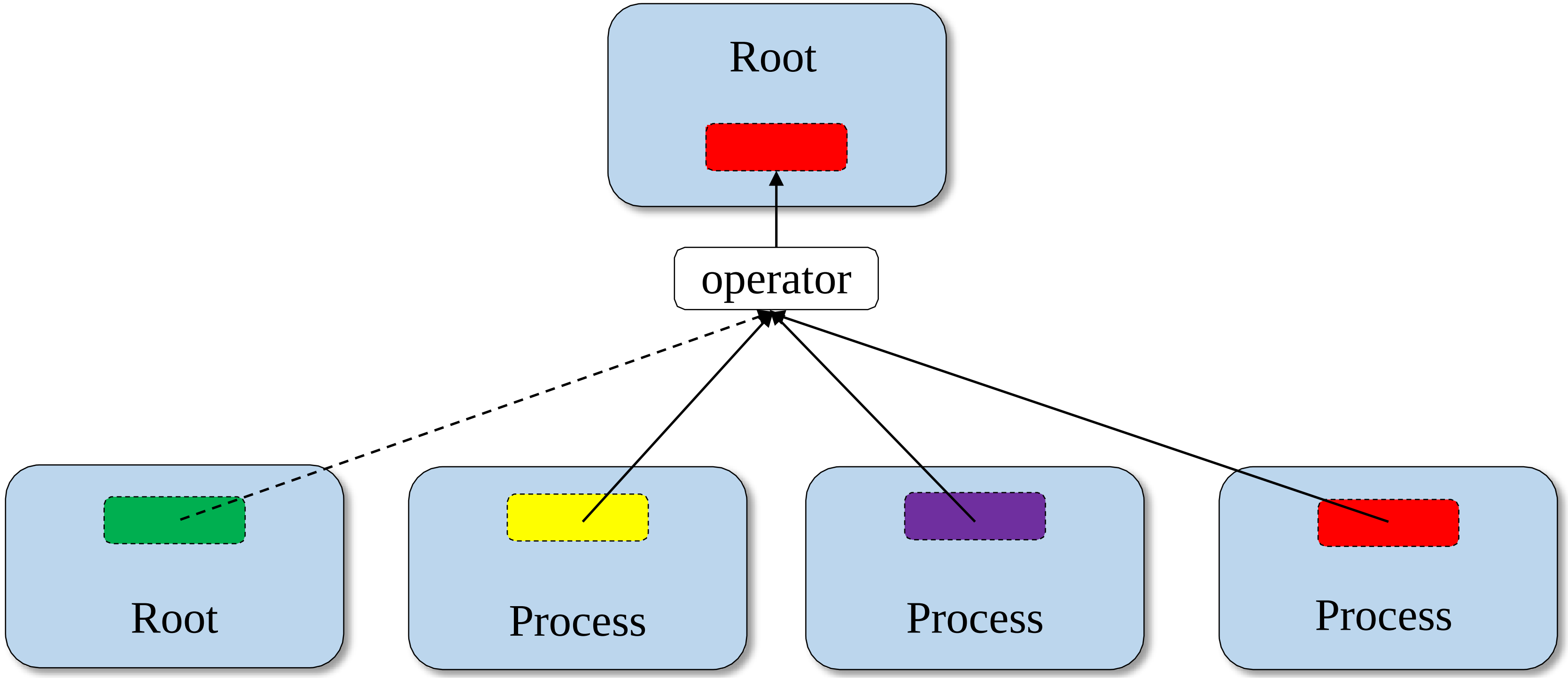
```
x=10, y[50]

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]);
}
```

Collective Communication (Reduce)

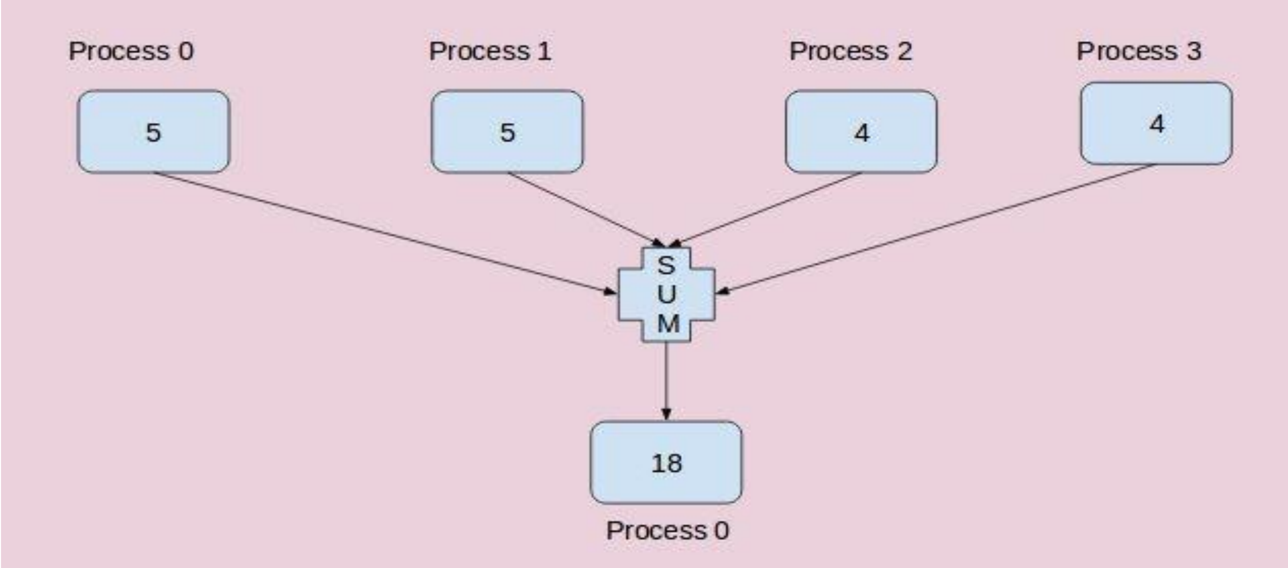
```
MPI_Reduce(&sendbuf, &recvbuf, count, datatype,
           mpi_operation, root, comm)
```

- Applies reduction operation on data from all processes
- Puts result on root process



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

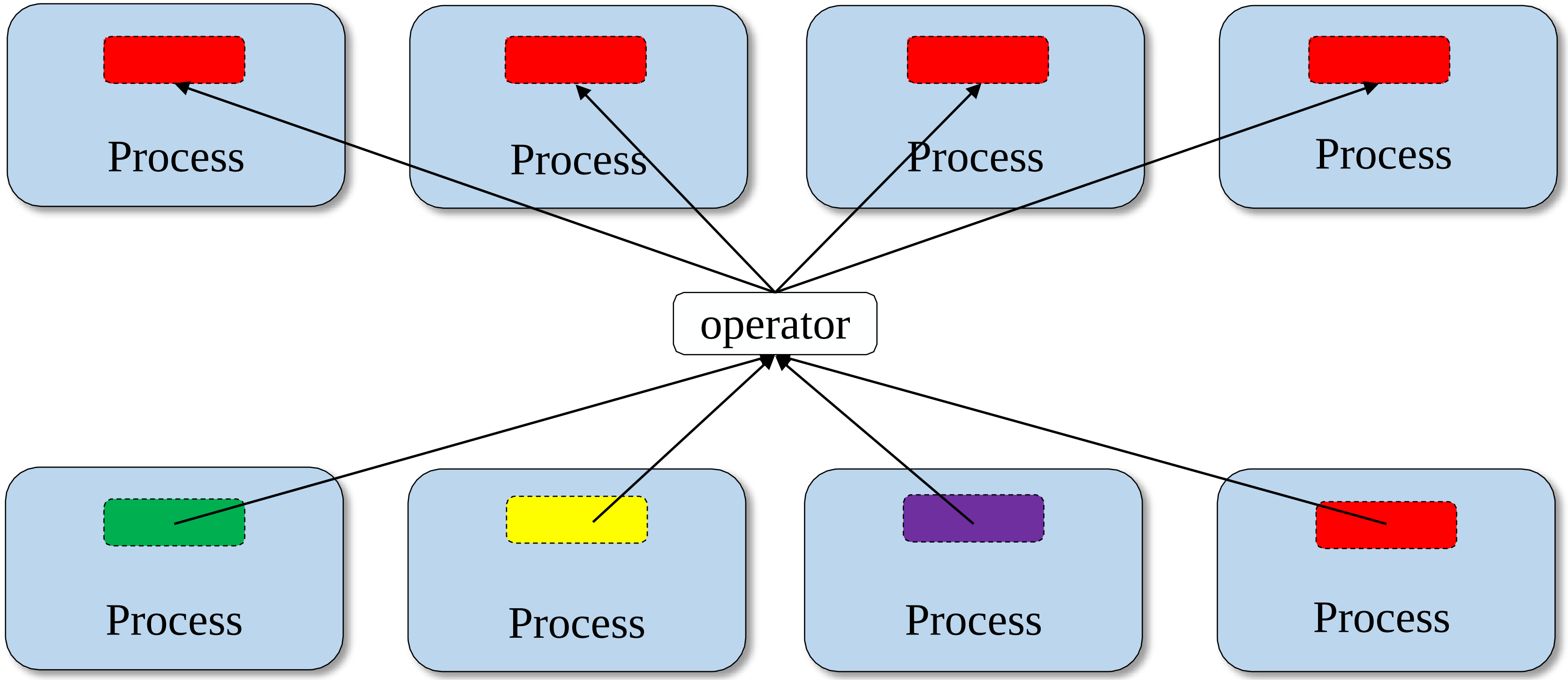
MPI Example - 8

```
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);
}
```

Collective Communication (Allreduce)

```
MPI_Allreduce(&sendbuf, &recvbuf, count,
              datatype, mpi_operation, comm)
```

- Applies reduction operation on data from all processes
- Stores results on all processes



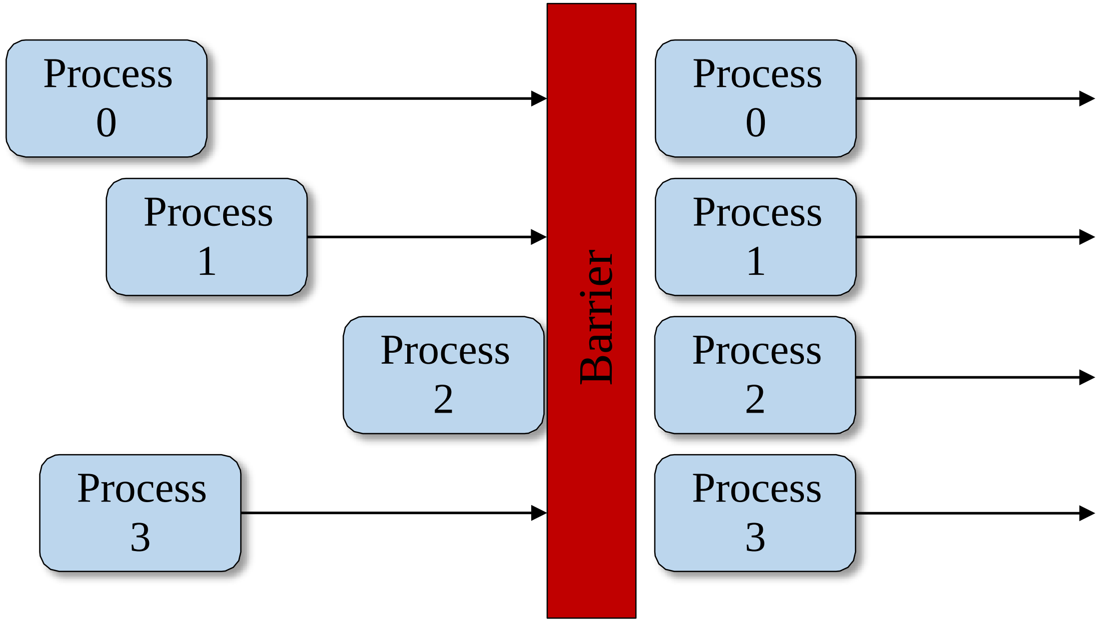
MORE Collective Communication Routines:

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

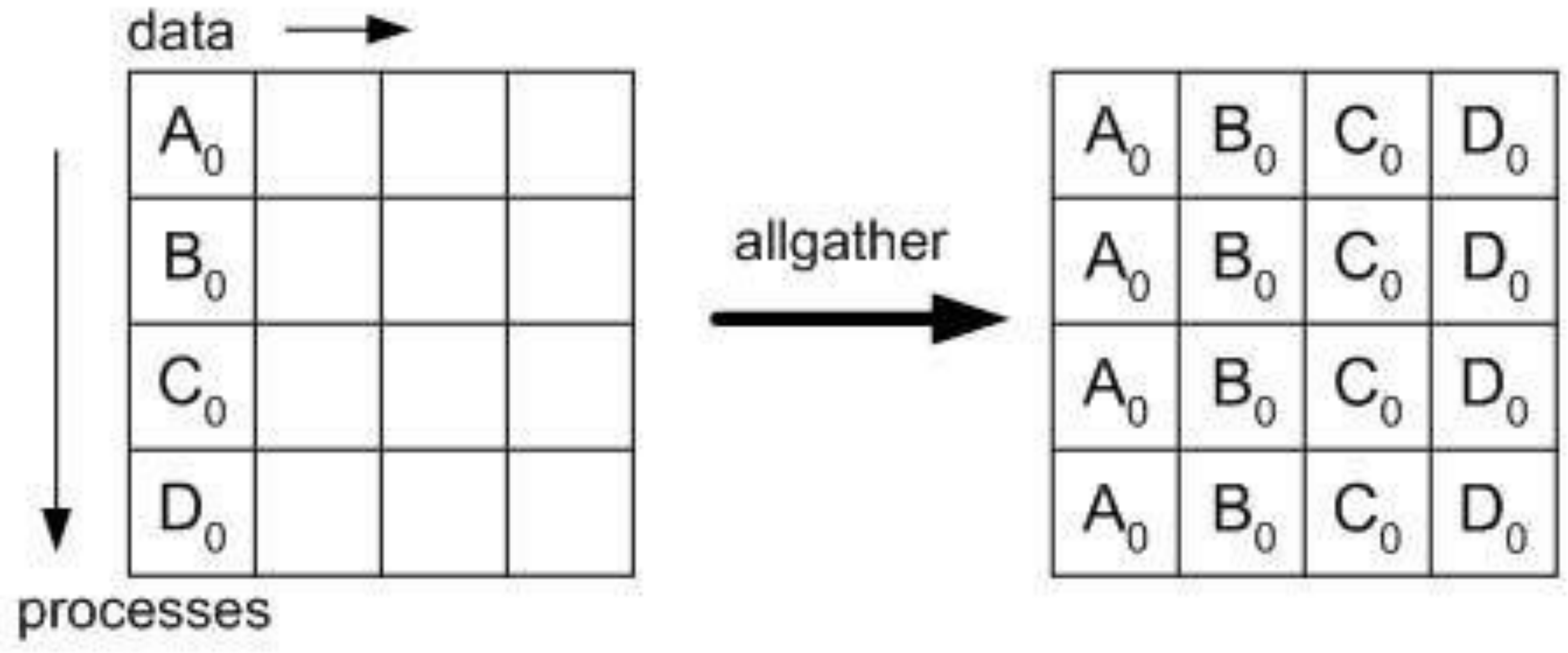
Collective Communication (Barrier)

```
MPI_Barrier(comm)
```

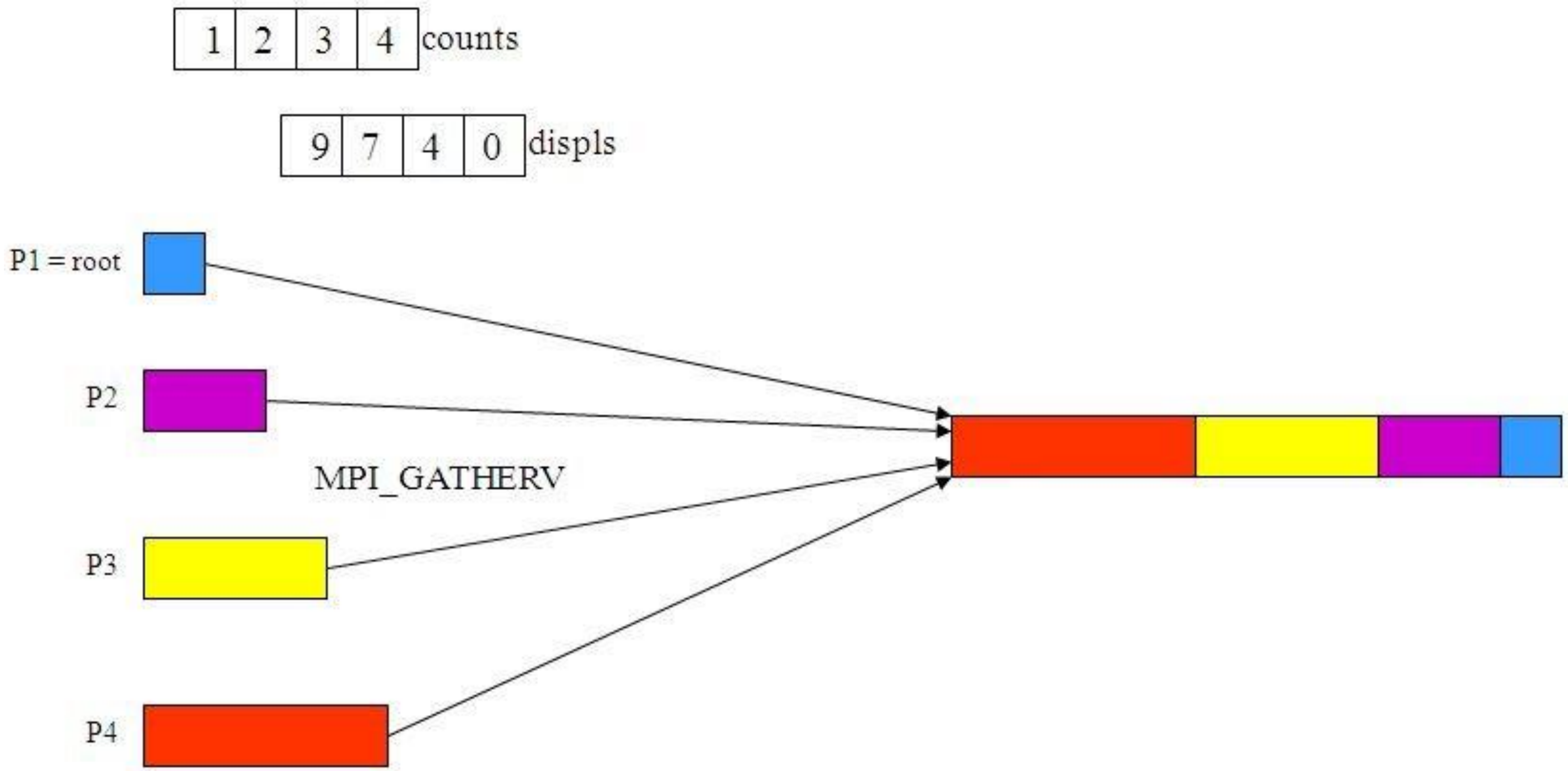
- Process synchronization (blocking)
  - All processes forced to wait for each other
- Use only where necessary
  - Will reduce parallelism



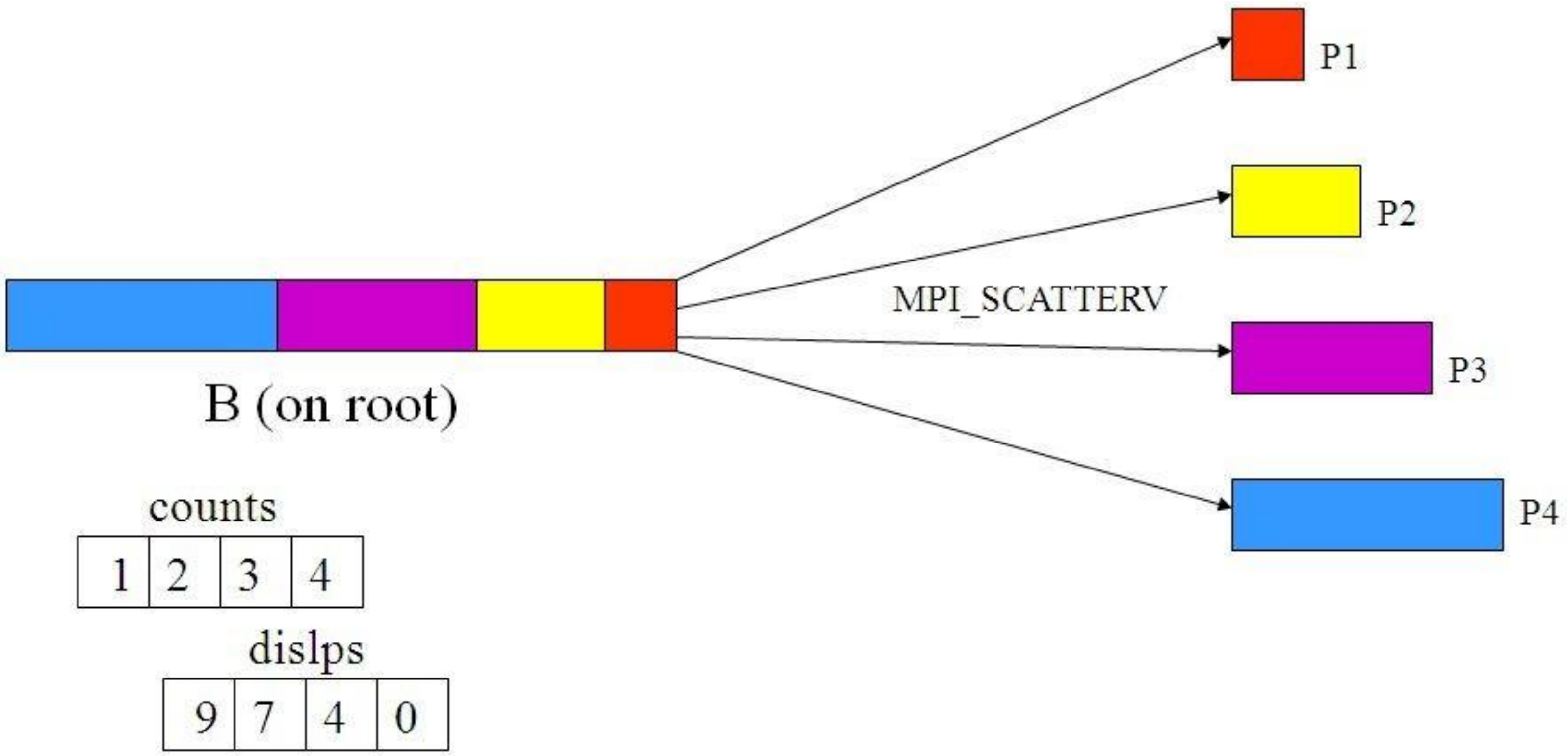
MPI_Allgather



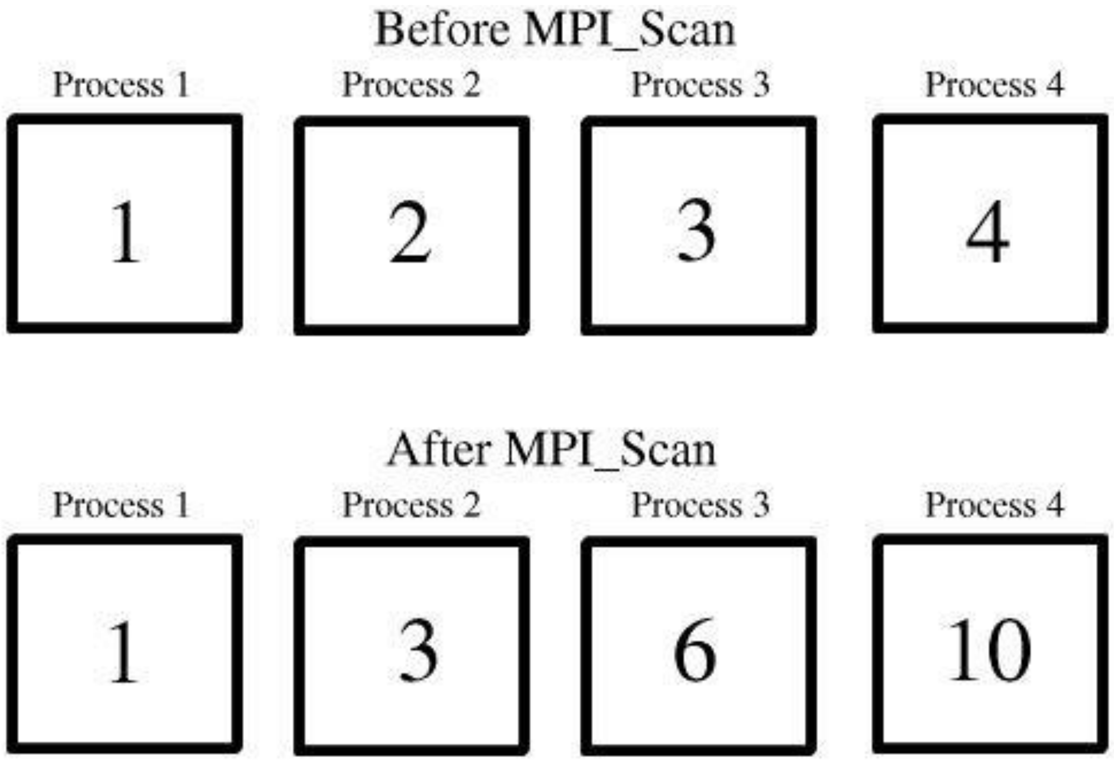
MPI_Gatherv



MPI_Scatterv



MPI_Scan



Before MPI_Scan
Process 1
Process 2
Process 3
Process 4
1
2
3
4
After MPI_Scan
Process 1
Process 2
Process 3
Process 4
1
3
6
10

Resources

http://www.mpi-forum.org (location of the MPI standard)
http://www.llnl.gov/computing/tutorials/mpi/
http://www.nersc.gov/nusers/help/tutorials/mpi/intro/
http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html
http://www-unix.mcs.anl.gov/mpi/tutorial/
MPICH (http://www-unix.mcs.anl.gov/mpi/mpich/)
Open MPI (http://www.open-mpi.org/)

THANK YOU