Part 2 – Programming using MPI

Addisu G. Semie, PhD
Asst. Prof.,Computational Science Program,
Addis Ababa University
Email: addisugezahegn@gmail.com

MPI Communication

- In this section we will cover:
  - Blocking and non blocking communication
  - Point to point communication
  - Collective communication

Introduction to MPI communication

- MPI statements (functions and/or subroutines) can be used
  - To initialize, manage, and finalize communication
  - To manage pair wise communication (point-to-point communication)
  - To manage group wise communication (collective communication)
  - To manage deriving new data type with any complex structure
  - To manage process topology (communicator hierarchy)
  - To manage buffer zone for processes

Introduction to MPI communication

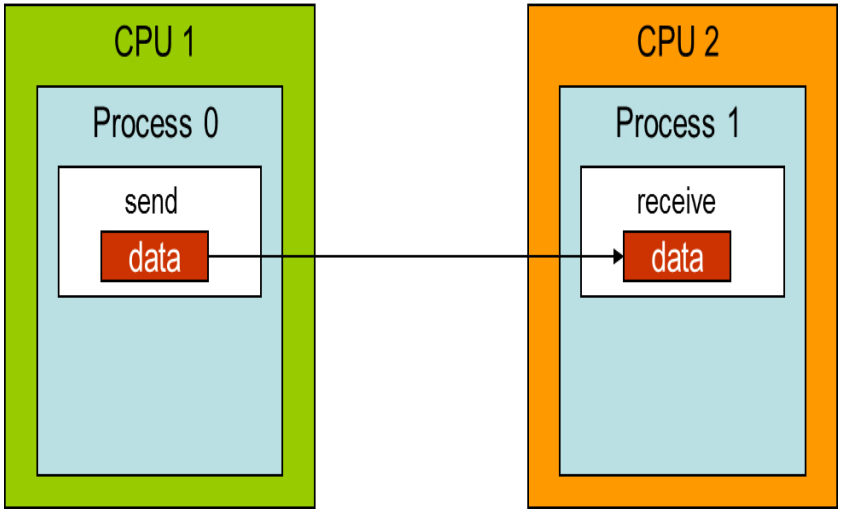
- MPI communication can be classified depending on various parameters such as:
  - The number of processes involved in a given MPI communication function
  - The implementation strategy of the MPI communication function
  - The mode of the communication function

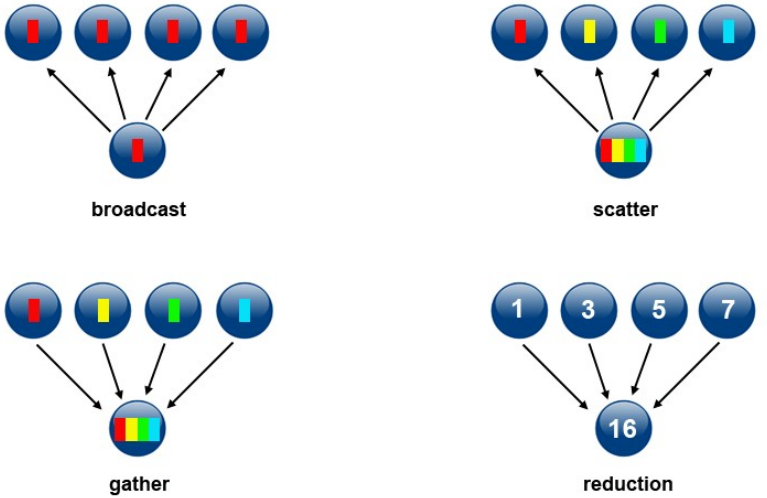
Introduction to MPI communication

Depending on the number of processes involved in a given MPI communication function, we can classify MPI communication into:

Point To Point Communication: It is a communication model between two processes

Collective Communication: It is a communication model between two or more processes

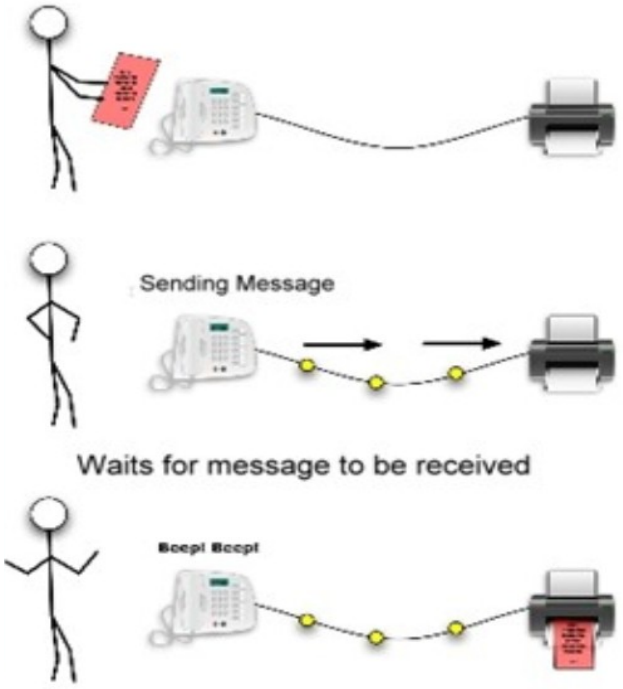




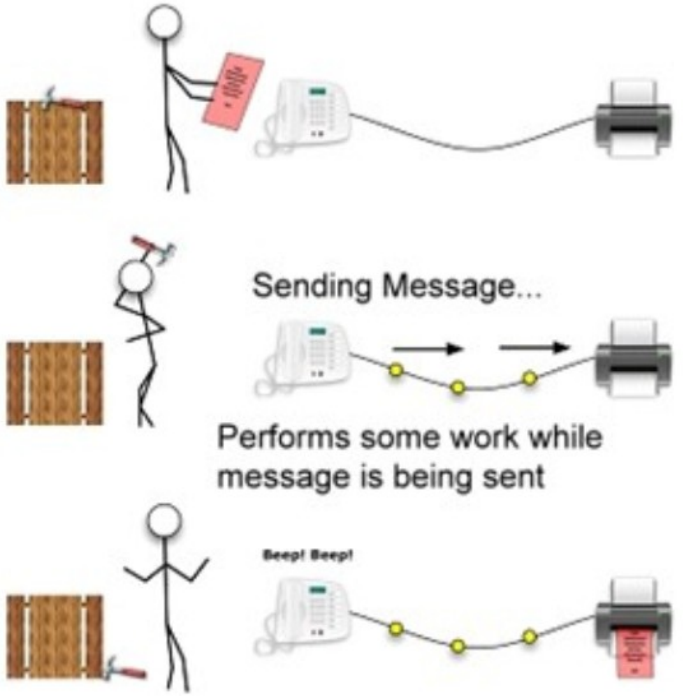
Introduction to MPI communication

Depending on the implementation strategy of the MPI communication function, we can classify MPI communication into:

Blocking: Occurs when one of the processors performs a send operation and doesn't continue unless the message buffer can be reclaimed.



Non-blocking: A processor performs a send or receive operation and immediately continues without caring whether the message received or not.



Introduction to MPI communication

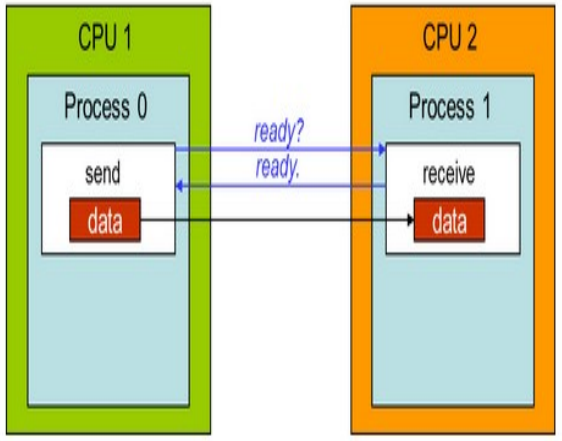
- MPI defines four communication modes which are selected via the specific SEND routine used.
- The communication mode instructs MPI on the algorithm that should be used for sending the message:

Communication Mode	Blocking Routines	Nonblocking Routines
Synchronous	MPI_Ssend	MPI_Issend
Buffered	MPI_Bsend	MPI_Ibsend
Ready	MPI_Rsend	MPI_Irsend
Standard	MPI_Send	MPI_Isend
	MPI_Recv	MPI_Irecv
	MPI_Sendrecv	
	MPI_Sendrecv_replace	

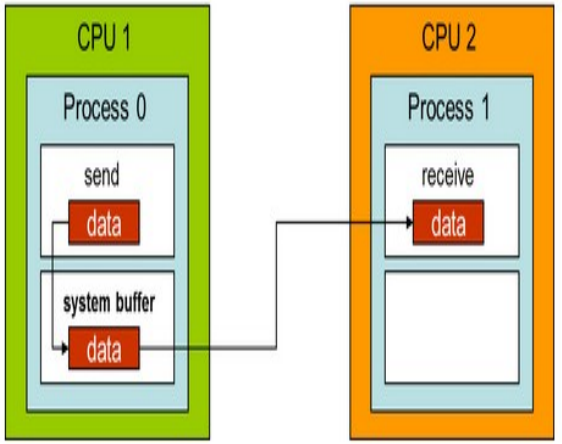
It should be noted that the RECV routine does not specify the communication mode - it is simply blocking or non blocking.

Point to Point Communication

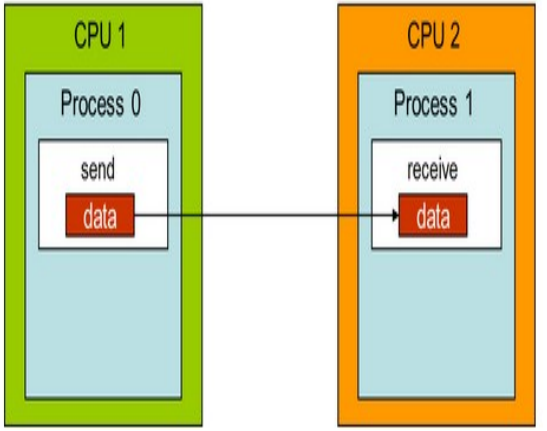
- In point to point communication, two process communicate to each other in such a way that one **send** a message and the second **receive** the message.
- MPI provides a set of **send** and **receive** function that allow the communication of message between two processes.
- Point to point communication (send and receive) can be **blocking** or **non-blocking**.



Synchronous mode



Buffered mode



Ready mode

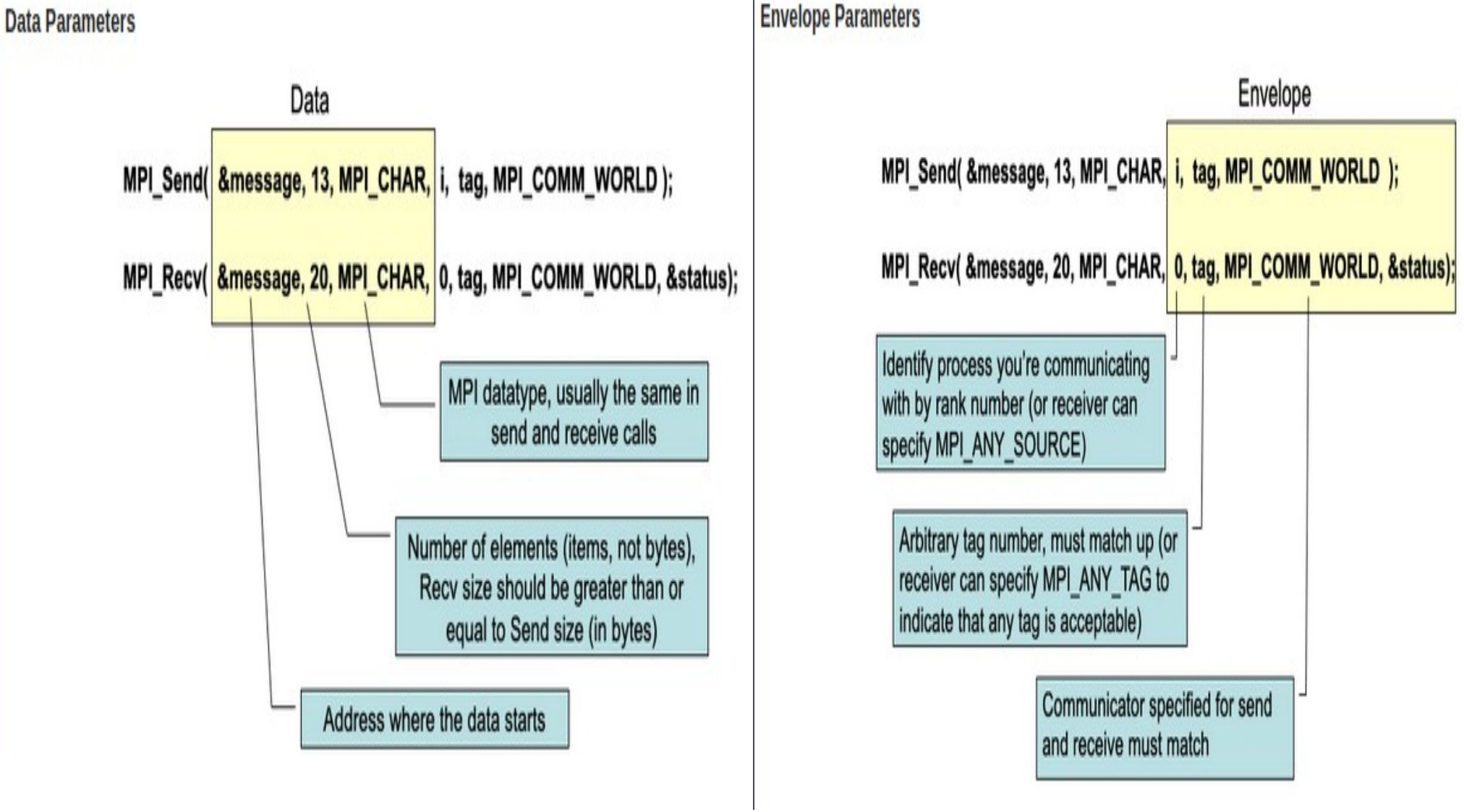
Point to Point Communication

- MPI send and receive function message has two parts:
  - **message data**
  - **message envelop**.

- **Message data** refers to the data that needs to be sent or received.
  - This usually involves address of the message data,
  - count of the message data elements and
  - the data type of the element.

- **Message envelop** refers to the information that uniquely identify the receiver and the sender.
  - This involves the sender processor (implicit in MPI send),
  - the receiver processor (implicit in MPI receive),
  - the tag (used to identify which data for what purpose),
  - the communicator (that enable processes within a group or across a group to communicate each other).

Point to Point Communication



Point to Point Communication

Blocking Send and Receive Syntax

C:

```
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)
```

Fortran:

```
MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)
MPI_RECV(buf, count, datatype, source, tag, comm, status, ierror)
```

buf	The beginning of the buffer containing the data to be sent. For Fortran, this is often the name of an array in your program. For C, it is an address.
count	The number of elements to be sent (not bytes)
datatype	The type of data, either a predefined MPI_Datatype or user-defined derivative type
dest (source)	The rank of the process which is the destination for (or source of) the message
tag	An arbitrary number which can be used to distinguish among messages
comm	The communicator id
ierror	The returned error code (Fortran only—in C, check the function's return value)
status	The array or structure of information that is returned. For example, if you specify a wildcard for source or tag, status will tell you the actual rank or tag for the message received

Point to Point Communication

- **ierror** is an integer value which shows the error returned by the MPI subroutine. In C it will be returned via its function name.

Possible Error value	Description
MPI_SUCCESS	No error; MPI routine completed successfully
MPI_ERR_COMM	Invalid communicator
MPI_ERR_COUNT	Invalid count argument. Count arguments must be non-negative; a count of zero is often valid.
MPI_ERR_TYPE	Invalid datatype argument. May be an uncommitted MPI_Datatype
MPI_ERR_TAG	Invalid tag argument. Tags must be non-negative. may also be MPI_ANY_TAG in MPI_Recv
MPI_ERR_RANK	Invalid source/destination rank. Rank must be between 0 and the size of the communicator minus one

Point to Point Communication

- **Datatype** is the type of each of the object (we are sending array of count objects each of them having the stated data type where the first object address is **buffer)**.
- The following shows primitive MPI data type and equivalent Fortan programming language type

Datatype	Fortran Equivalent	Description
MPI_CHARACTER	CHARACTER	8-bit character
MPI_COMPLEX	COMPLEX	32-bit floating point real, 32-bit floating point imaginary
MPI_COMPLEX8	COMPLEX*8	32-bit floating point real, 32-bit floating point imaginary
MPI_COMPLEX16	COMPLEX*16	64-bit floating point real, 64-bit floating point imaginary
MPI_COMPLEX32	COMPLEX*32	128-bit floating point real, 128-bit floating point imaginary
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX	64-bit floating point real, 64-bit floating point imaginary
MPI_DOUBLE_PRECISION	DOUBLE PRECISION	64-bit floating point
MPI_INTEGER	INTEGER	32-bit integer
MPI_INTEGER1	INTEGER*1	8-bit integer
MPI_INTEGER2	INTEGER*2	16-bit integer
MPI_INTEGER4	INTEGER*4	32-bit integer
MPI_INTEGER8	INTEGER*8	64-bit integer
MPI_LOGICAL	LOGICAL	32-bit logical
MPI_LOGICAL1	LOGICAL*1	8-bit logical
MPI_LOGICAL2	LOGICAL*2	16-bit logical
MPI_LOGICAL4	LOGICAL*4	32-bit logical
MPI_LOGICAL8	LOGICAL*8	64-bit logical
MPI_REAL	REAL	32-bit floating point
MPI_REAL4	READ*4	32-bit floating point
MPI_REAL8	REAL*8	64-bit floating point
MPI_REAL16	REAL*16	128-bit floating point

Point to Point Communication

- **Datatype** is the type of each of the object (we are sending array of count objects each of them having the stated data type where the first object address is **buffer)**.
- The following shows primitive MPI data type and equivalent C programming language type

Possible Datatype	C Equivalence	Description
MPI_CHAR	signed int	8-bit character
MPI_SIGNED_CHAR	signed int	8-bit signed character
MPI_UNSIGNED_CHAR	unsigned int	8-bit unsigned character
MPI_SHORT	signed short	16-bit integer
MPI_UNSIGNED_SHORT	unsigned short	16-bit unsigned integer
MPI_INT	signed int	32-bit integer
MPI_UNSIGNED	unsigned int	32-bit unsigned integer
MPI_LONG	signed int	32-bit integer
MPI_UNSIGNED_LONG	unsigned int	32-bit unsigned integer
MPI_LONG_LONG	signed int	64-bit integer
MPI_LONG_LONG_INT	signed int	64-bit integer
MPI_UNSIGNED_LONG_LONG	unsigned int	64-bit unsigned integer
MPI_FLOAT	float	32-bit floating point
MPI_DOUBLE	double	64-bit floating point
MPI_LONG_DOUBLE	long double	64-bit floating point
MPI_WCHAR	wchar_t	Wide (16-bit) unsigned character (MPI-2)
MPI_BYTE		
MPI_PACKED		

Point to Point Communication

- Examples of the blocking, standard mode point to point communication in FORTRAN Programming Language

```
program send_receive
        include "mpif.h"

        integer ::numProcess, myRank, i
        integer ::num_chars
        integer ::ierror
        integer ::status(MPI_STATUS_SIZE)
        character(100) :: greeting0
        character (100) :: greetingOther

        call MPI_Init(ierror)
        call MPI_Comm_rank(MPI_COMM_WORLD, myRank,ierror)
        call MPI_Comm_size(MPI_COMM_WORLD, numProcess,ierror)

        if(myRank == 0) then
                do i = 1, numProcess-1
                        call MPI_Recv(greeting0, 200, MPI_CHARACTER, i, 1, MPI_COMM_WORLD, status,ierror)
                        write(*,*)"Master Received message with the following Detail: "
                        write(*,*)"Message = ",greeting0, " Source process ID = ",status(MPI_SOURCE)
                end do
        else
                write(greetingOther,*) "Hello world from process ", myRank, " of ", numProcess
                call MPI_Send(greetingOther,100, MPI_CHARACTER, 0, 1, MPI_COMM_WORLD,ierror)
        end if

        if(myRank == 0) then
                write(*,*) 'That is all for now!'
        end if
        call MPI_Finalize(ierror)
end
```

Point to Point Communication

- Examples of the blocking, standard mode point to point communication in C Programming Language

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
        int numProcess, myRank, i;
        int num_chars;
        char greeting[200];
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcess);
        sprintf(greeting, "Hello world from process %d of %d\n", myRank, numProcess);
        if(myRank == 0){
        //char greeting[200];
                fputs(greeting, stdout);
                for(i = 1; i < numProcess; i++)
                {
                        MPI_Recv(greeting, 200, MPI_CHAR, i, 1, MPI_COMM_WORLD, &status);
                  //fputs(greeting, stdout);
                        printf("Master Received message with the following Detail: \n\t Message = %s\n\tSource process ID = %d\n",greeting, status.MPI_SOURCE);
                }
           }
           else{
                char greeting[100];
                sprintf(greeting, "Hello world from process %d of %d ", myRank, numProcess);
                MPI_Send(greeting,100,MPI_CHAR, 0, 1, MPI_COMM_WORLD);
          }
          MPI_Finalize();
         return 0;
}
```

Point to Point Communication

- The status structure which is an argument of the MPI_Recv has an element that can be used to know the actual number of elements sent from the sender in addition to the TAG and the SOURCE information.

- However, the structure element can not be directly accessible (encapsulated) to know the actual number of elements sent from the sender.

- In order to access the actual number of elements, we use the MPI function **MPI_Get_count**
- **In FORTRAN**
  - **call MPI_Get_count(status, MPI_CHARACTER, num_chars)**
- **In C**
  - **MPI_Get_count(&status, MPI_CHAR, &num_chars)**

Point to Point Communication

- Note that in the above code the variable **greeting** has different size for process with rank 0 and others.
- MPI_Get_count function return count as the number of data elements in the sent message.
- The datatype used in MPI_Get_count must be the same as the data type specified in the MPI_Recv function.
- The function return error code or success code as an integer.

Point to Point Communication

The following example demonstrate the MPI_Get_count function in FORTRAN

```
program send_receive
        include "mpif.h"

        integer ::numProcess, myRank, i
        integer ::num_chars
        integer ::ierror
        integer ::status(MPI_STATUS_SIZE)
        character(100) :: greeting0
        character (100) :: greetingOther

        call MPI_Init(ierror)
        call MPI_Comm_rank(MPI_COMM_WORLD, myRank,ierror)
        call MPI_Comm_size(MPI_COMM_WORLD, numProcess,ierror)

        if(myRank == 0) then
                do i = 1, numProcess-1
                        call MPI_Recv(greeting0, 200, MPI_CHARACTER, i, 1, MPI_COMM_WORLD, status,ierror)
                        write(*,*)"Master Received message with the following Detail: "
                        write(*,*)"Message = ",greeting0, " Source process ID = ",status(MPI_SOURCE)
                        call MPI_Get_count(status, MPI_CHARACTER, num_chars)
                        write(*,*) "MPI_Get_count detail Tag =  ",status(MPI_TAG)
                        write(*,*) "Source Processor ID =  ",status(MPI_SOURCE)
                        write(*,*) "Number of MPI_CHAR obtained =  ", num_chars
                end do
        else
                write(greetingOther,*) "Hello world from process ", myRank, " of ", numProcess
                call MPI_Send(greetingOther,100, MPI_CHARACTER, 0, 1, MPI_COMM_WORLD,ierror)
        end if

        if(myRank == 0) then
                write(*,*) 'That is all for now!'
        end if
        call MPI_Finalize(ierror)
end
```

Point to Point Communication

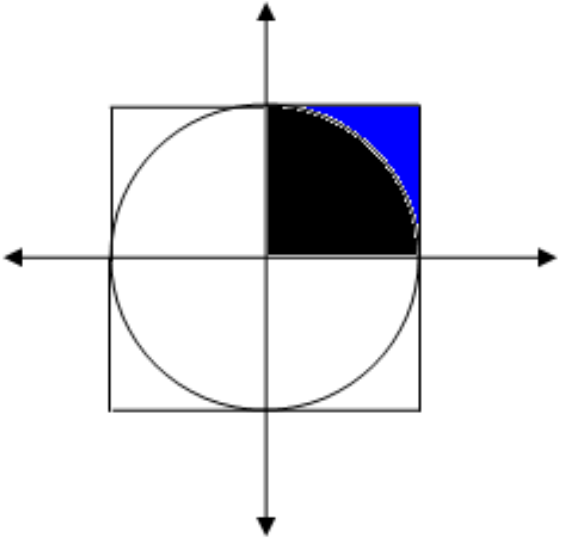
The following example demonstrate the MPI_Get_count function in C

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[])
	{
		int numProcess, myRank, i,num_chars;
		MPI_Status status;
		MPI_Init(&argc, &argv);
		MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
		MPI_Comm_size(MPI_COMM_WORLD, &numProcess);
		if(myRank == 0) {
			char greeting[200];
			for(i = 1; i < numProcess; i++) {
				MPI_Recv(greeting, 200, MPI_CHAR, i, 1, MPI_COMM_WORLD, &status);
				printf("Received message Detail: \tMessage = %s\tSource process ID = %d\n",greeting, status.MPI_SOURCE);
			MPI_Get_count(&status, MPI_CHAR, &num_chars);
			printf("MPI_Get_count detail \t Tag = %d \tSource Processor ID=%d\t Number of MPI_CHAR obtained = %d \n",status.MPI_TAG, status.MPI_SOURCE, num_chars);
			}
		}
		else {
			char greeting[100];
			sprintf(greeting, "Hello world from processed %d of %d ", myRank, numProcess);
			MPI_Send(greeting,100,MPI_CHAR, 0, 1, MPI_COMM_WORLD);
		}
		MPI_Finalize();
		return 0;
}
```

Sample Application – compute PI

Problem description

Estimate the value of PI using the ratio of the area of a 1 unit square and the area of sector of a circle



Sample Application – compute PI

Algorithmic Description

- Area of a circle is PI * R * R which is PI if the radius is one.
- Consider the unit circle whose center is at the origin and a square of 1 unit side (whose adjacent sides are the +ve x and +ve y axis).
- The part of the unit circle under the boundary of the square has area = 0.25*PI.
- The ratio of the area of the sector to the area of the square becomes 0.25PI/1 = 0.25PI
- Now randomly generate a set of N points within the boundary of the square (i.e. both x and y value are positive real numbers less than 1).
- Let Na of this points are within the sector and the rest are outside the sector.
- Na/N as the limit for N goes to infinity is the same as 0.25PI.
- This implies, PI = 4 * (Na / N) when N approaches infinity.
- Though it is difficult to approximate infinity we can perform the experiment in a parallel computing so that each will perform some part of the experiment and the result can be computed in a joint fashion.

Sample Application – compute PI

Parallization logic

- Given K+1 processors (K working and 1 managing)
- Then each worker will do the experiment for predefined value of N (Total number of experiment becomes k*N)
- They will report their finding (Na) to the master
- The master collect the responses sum up the result (say it becomes S_{Na}) and compute value of PI as (4 * S_{Na} /(K*N))
- The master display the result

Sample Application – add matrix row

Algorithmic description

for each row_index i
  sum = 0
  for each col_index j
    sum = sum + A_{ij}
  end for
  $output_i$ = sum
end for

return output

Sample Application – add matrix row

Parallization logic

- *Given K+1 processors (K working and 1 managing), either of the following is true:*

  $K > N$ *or*

  $K < N$ *or*

  $K = N$

  In this case will consider the action as shown in the table bellow

Condition	Action
K > N	send one row of data to each processor and some processor will not have thing to do
K < N	send the first K rows to each of the processors and the remaining will be sent depending on which process complete its task. The moment a process complete its task send the next row to that process. Continue this until all rows get sent and processed.
K = N	Send the i^{th} rows to the i^{th} processor

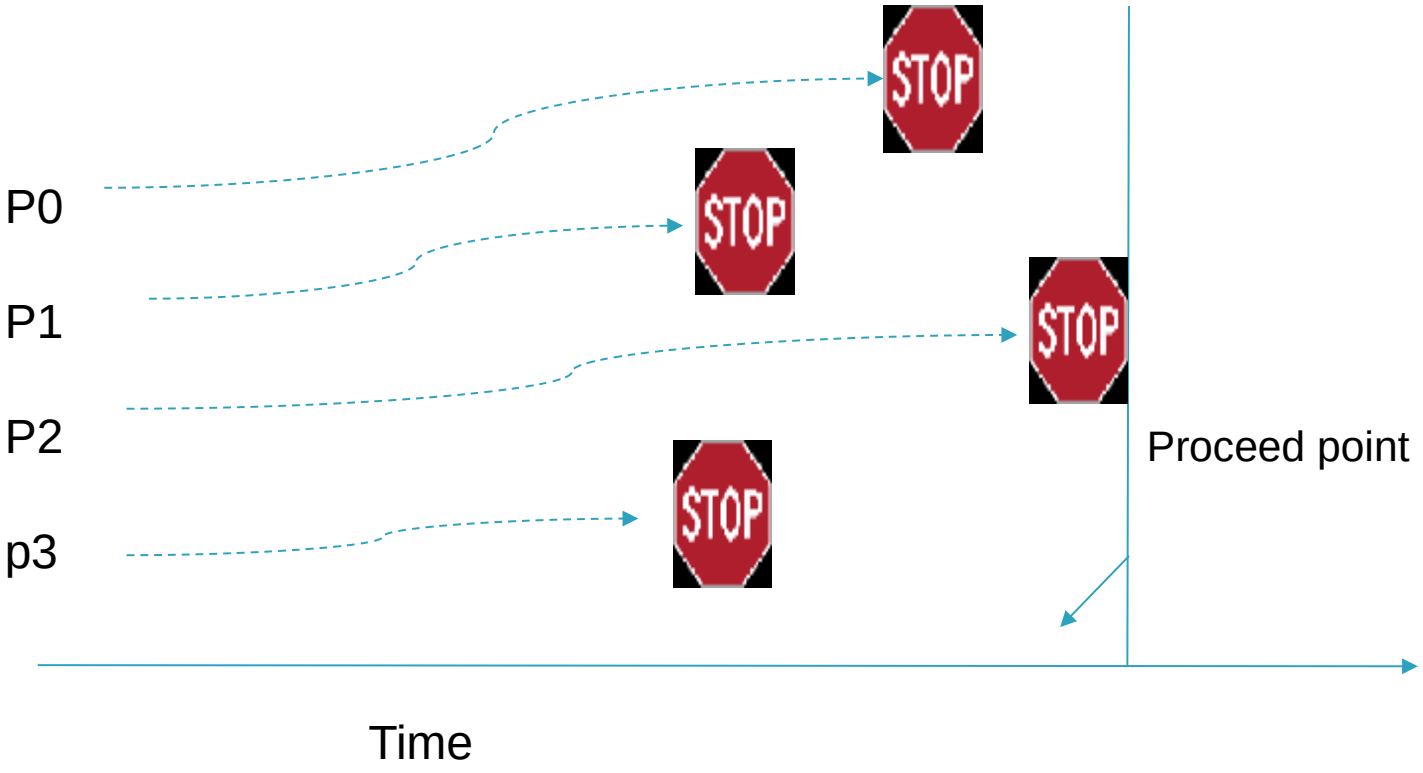
Collective Communication

- Collective communication refers to the process of transmitting message among all processes in a group
- MPI collective communication can be divided into three subsets:

  1. Synchronization
     - Barrier synchronization
  2. Data Movement
     - Broadcast from one member to all other members
     - Gather data from an array spread across processes into one array
     - Scatter data from one member to all members
     - All-to-all exchange of data
  3. Global Computation
     - Global reduction (e.g., sum, min of distributed data elements)

Collective Communications: Barrier Synchronization

- Blocks the caller until all group members have called it.
- The call returns at any process only after all group members have entered the call.



Collective Communications: Barrier Synchronization

- The syntax of MPI_BARRIER for both C and FORTRAN programs is given below:

C

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

Where:

MPI_Comm	is an MPI predefined structure for communicators, and
comm	is a communicator.

FORTRAN

```
MPI_BARRIER(comm, ierr)
```

Where:

comm	is an integer denoting a communicator, and
ierr	is an integer return error code.

Collective Communications: Barrier Synchronization

- Barrier Synchronization across all group members

FORTRAN

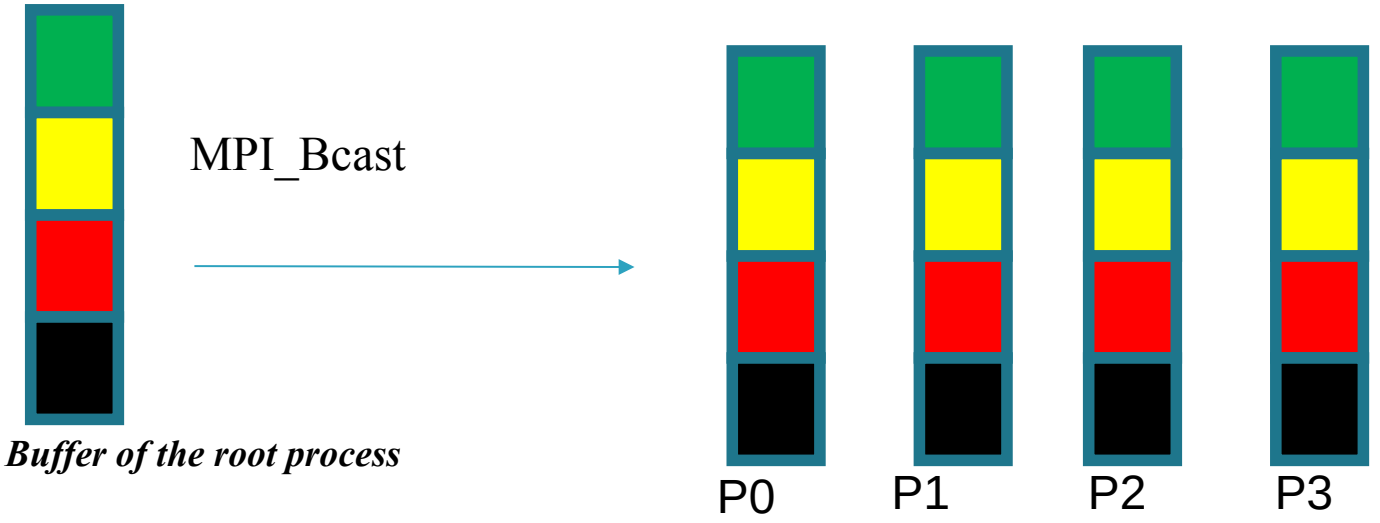
```
program send_receive
      include "mpif.h"

      integer ::numProcess, myRank
      integer ::ierror

      call MPI_Init(ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, myRank,ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, numProcess,ierror)

      write(*,*) "Hello world, I am ", myRank, "  of", numProcess

      !Blocks until all process have reached this routine

      call MPI_Barrier(MPI_COMM_WORLD, ierror)

      write(*,*) "I am", myRank, "   of", numProcess

      call MPI_Finalize(ierror)
end
```

C

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]) {
      int rank, size;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      printf("Hello, world, I am %d of %d\n", rank, size);

      //Blocks until all processes have reached this routine
      MPI_Barrier(MPI_COMM_WORLD);
      printf("\t\t\t\t I am %d of %d\n", rank, size);
      MPI_Finalize();
      return 0;
}
```

Collective Communications: Broadcast

- Broad cast from one member to all members of the group
- Broad cast the content of the buffer of the root processor to all process in comm.
- Root must be element of the **comm** group.



Collective Communications: Broadcast

C

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
     int root, MPI_Comm comm)
```

FORTRAN

```
MPI_BCAST(buffer, count, datatype, root, comm, ierr)
```

where:

`buffer`	is the starting address of a buffer,
`count`	is an integer indicating the number of data elements in the buffer,
`datatype`	is an MPI defined constant indicating the data type of the elements in the buffer,
`root`	is an integer indicating the rank of broadcast root process, and
`comm`	is the communicator.

Collective Communications: Broadcast

Fortran Example:

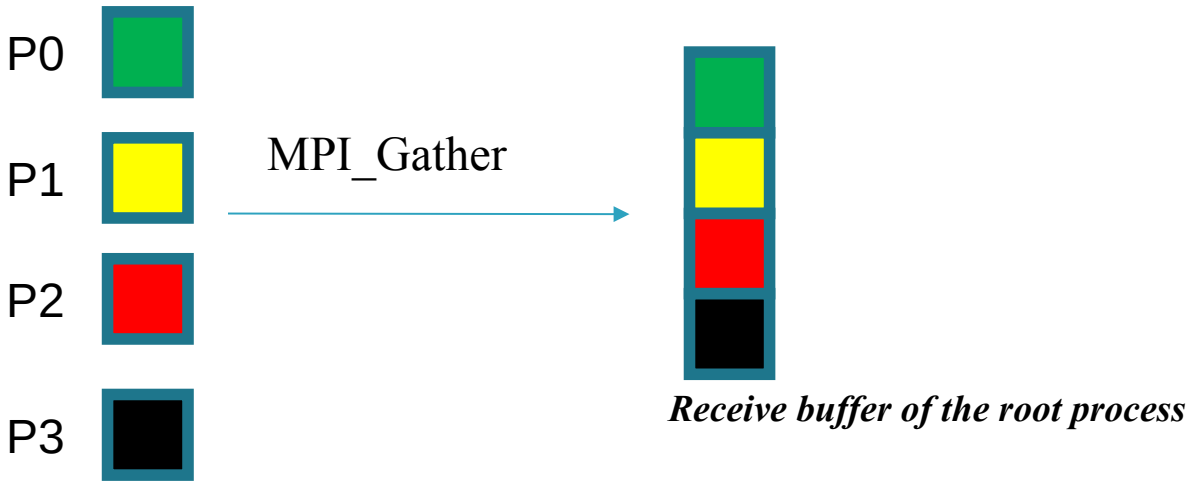
```
use MPI
character(12) message
integer rank,root
data root/0/
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
if (rank .eq. root) then
    message = "Hello, world"
endif
call MPI_BCAST(message, 12, MPI_CHARACTER, root, &
        MPI_COMM_WORLD, ierr)
print*, "node", rank, ":", message
call MPI_FINALIZE(ierr)
end
```

C Example:

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int  i, rank, size;
  MPI_Status status;
  int root = 0;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == root)
  {
    strcpy(message, "Hello, world");
  }
  MPI_Bcast(message, 13, MPI_CHAR, root, MPI_COMM_WORLD);
  printf( "Message from process %d : %s\n", rank, message);

  MPI_Finalize();
}
```

Collective Communications: Gather Data

- Gather data from all group members to one member
- Each process (including the root) sends the content of its ***sendbuffer*** to the ***root*** process.
- The ***root*** process receives the message and store the message into the ***receive buffer*** according to the rank of the sender



Collective Communications: Gather Data

C

```
int MPI_Gather(const void* sbuf, int scount, \
    MPI_Datatype stype, void* rbuf, int rcount, \
    MPI_Datatype rtype, int root, MPI_Comm comm )
```

sbuf	is the starting address of the send buffer,
scount	is the number of elements to be sent,
stype	is the data type of send buffer elements,
rbuf	is the starting address of the receive buffer,
rcount	is the number of elements for any single receive,
rtype	is the data type of the receive buffer elements,
root	is the rank of receiving process, and
comm	is the communicator.

FORTRAN

```
MPI_GATHER(sbuf, scount, stype, rbuf, rcount, rtype,
    root, comm, ierr)
```

- ***recvcount*** must be greater than or equals to ***sendcount***
- ***recvBuffer*** should have sufficiently large space (for ***N* Recvcount*** objects where ***N*** is the number of processes in the communicator)

Collective Communications: Gather Data

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[]) {
        int myRank, numProcess,value, *data;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcess);
        value = (myRank + 1) *10; //set what each process to contribute
        //arrange memory space for the gathered data
        if(myRank==0) data = (int *) malloc(numProcess*sizeof(int));

        //the value in value of each process will be gathered into the array data of process zero
        MPI_Gather(&value, 1, MPI_INT, data, 1, MPI_INT, 0,MPI_COMM_WORLD);

        printf("I am %d contributing data  value %d to be gathered \n",myRank, value);

        //wait the rest action until all display their contribution
        MPI_Barrier(MPI_COMM_WORLD);

        //let the first process to display the result
        if(myRank ==0) {
                int i;
                printf("I am %d getting the following data gathered from different processes\n", myRank);
                for(i = 0; i < numProcess; i++)
                        printf("%d\t",data[i]);
                printf("\n");
        }
        MPI_Finalize();
        return 0;
}
```

Collective Communications: Gather Data

Fortran

```
program send_receive
        include "mpif.h"
        integer ::numProcess, myRank, i
        integer ::num_chars
        integer ::ierror
        integer ::value
        integer ::status(MPI_STATUS_SIZE)
        integer, allocatable, dimension(:)  :: pdata

        call MPI_Init(ierror)
        call MPI_Comm_rank(MPI_COMM_WORLD, myRank,ierror)
        call MPI_Comm_size(MPI_COMM_WORLD, numProcess,ierror)
        !set what each processor contribute
        value = (myRank + 1)*10
        !arrange memory space for the gathered data
        if(myRank == 0)  allocate(pdata(numProcess))
        !the value in value of each process will be gathered into the array data of process zero
        write(*,*) "I am ", myRank, "contributing data value", value, "to be gathered"
        call MPI_Gather(value, 1, MPI_INT, pdata, 1, MPI_INT, 0, MPI_COMM_WORLD, ierror)
        !Wait the rest action until all display their contribution
        call MPI_Barrier(MPI_COMM_WORLD, ierror)
        !let the first process display the result
        if(myRank == 0) then
                write(*,*) "I am ", myRank, "getting the following data gathered from different processes"
                do i=1, numProcess
                        write(*,*) '    ', pdata(i)
                end do
        end if
        call MPI_Finalize(ierror)
end
```

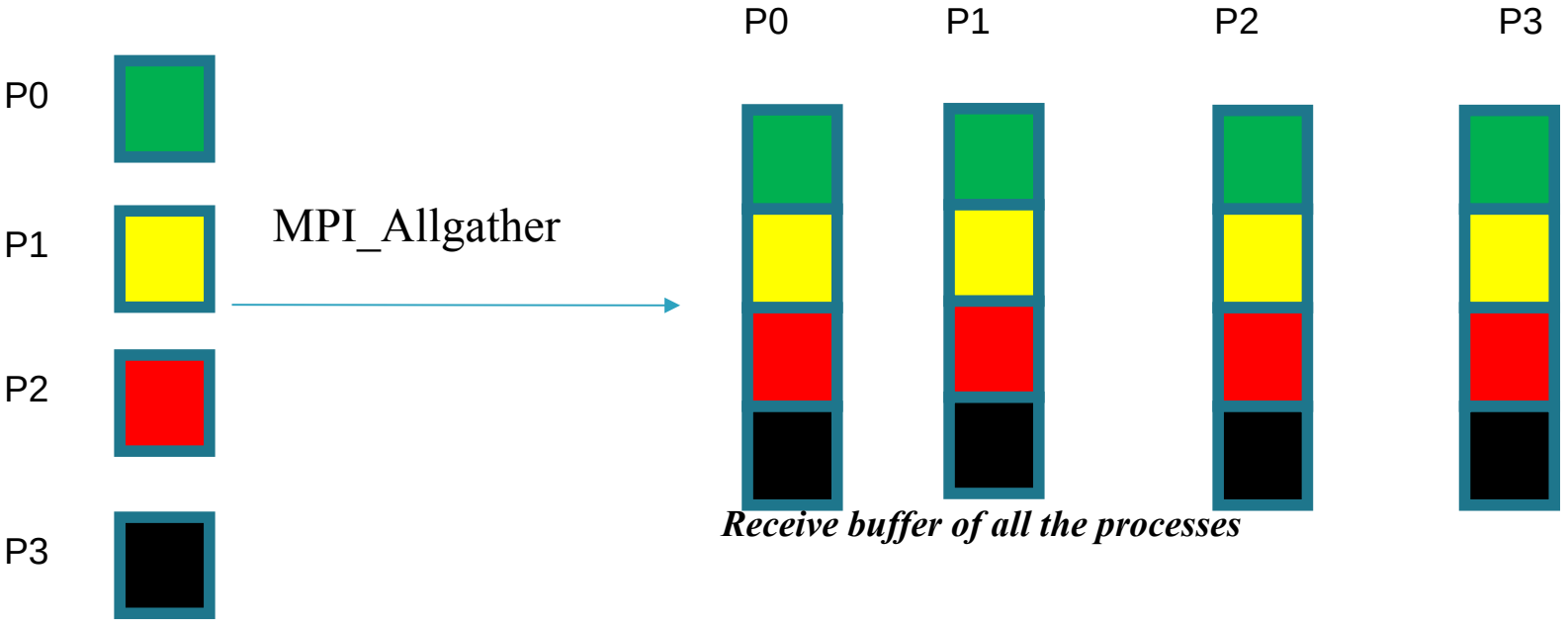
Collective Communications: Gather Data

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[]) {
        int myRank, numProcess,value, *data;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcess);
        value = (myRank + 1) *10; //set what each process to contribute
        //arrange memory space for the gathered data
        if(myRank==0) data = (int *) malloc(numProcess*sizeof(int));

        //the value in value of each process will be gathered into the array data of process zero
        MPI_Gather(&value, 1, MPI_INT, data, 1, MPI_INT, 0,MPI_COMM_WORLD);

        printf("I am %d contributing data  value %d to be gathered \n",myRank, value);

        //wait the rest action until all display their contribution
        MPI_Barrier(MPI_COMM_WORLD);

        //let the first process to display the result
        if(myRank ==0) {
                int i;
                printf("I am %d getting the following data gathered from different processes\n", myRank);
                for(i = 0; i < numProcess; i++)
                        printf("%d\t",data[i]);
                printf("\n");
        }
        MPI_Finalize();
        return 0;
}
```

Collective Communications: Allgather

- Allgather: is a variation of Gather where all members of the group receive the gather result
- Each process send ***sendcount*** number of data, each having ***sendtype*** data type to the root from their send buffer and the root gather the data according to the process rank order and the final gathered data will be copied into the ***recvbuffer*** of all the processes.



Collective Communications: Allgather

Fortran

```
program send_receive
        include "mpif.h"
        integer ::numProcess, myRank, i
        integer ::num_chars
        integer ::ierror
        integer ::value
        integer ::status(MPI_STATUS_SIZE)
        integer, allocatable, dimension(:)  :: pdata

        call MPI_Init(ierror)
        call MPI_Comm_rank(MPI_COMM_WORLD, myRank,ierror)
        call MPI_Comm_size(MPI_COMM_WORLD, numProcess,ierror)
        !set what each processor contribute
        value = (myRank + 1)*10
        !arrange memory space for the gathered data
        allocate(pdata(numProcess))
        write(*,*) "I am ", myRank, "contributing data value", value, "to be gathered"
        !the value in value of each process will be gathered into the array data of all members of the group
        call MPI_Allgather(value, 1, MPI_INT, pdata, 1, MPI_INT, MPI_COMM_WORLD, ierror)
        !Wait the rest action until all display their contribution
        call MPI_Barrier(MPI_COMM_WORLD, ierror)
        !let the first process display the result
        write(*,*) "I am ", myRank, "getting the following data gathered from different processes"
        do i=1, numProcess
                print*,  '    ', pdata(i)
        end do
        write(*,*) "================================================"
        call MPI_Finalize(ierror)
end
```

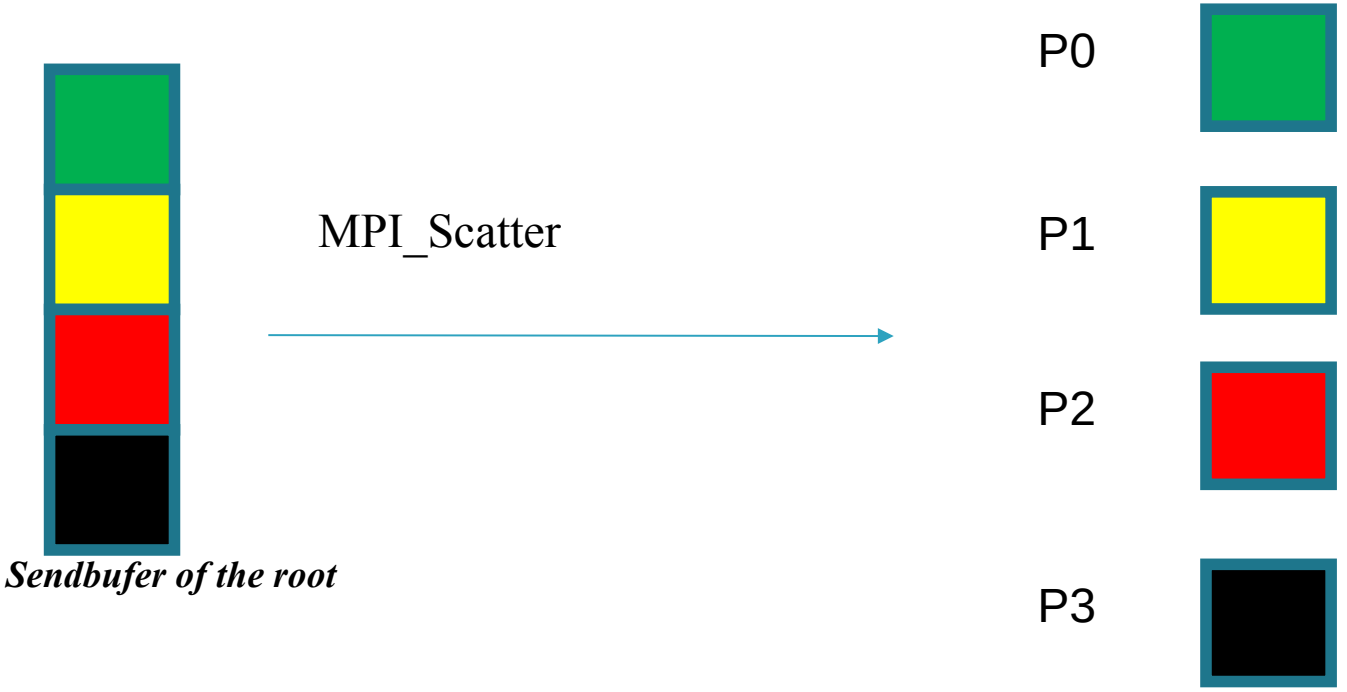
Collective Communications: Allgather

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[]) {
        int myRank, numProcess,value, *data;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcess);
        //set what each process to contribute
        value = (myRank + 1) *10;
        //arrange memory space for the gathered data
        data = (int *) malloc(numProcess*sizeof(int));
        printf("I am %d contributing data value %d to be gathered \n",myRank, value);
        //the value in value of each process will be gathered into the array data of process zero
        MPI_Allgather(&value, 1, MPI_INT, data, 1, MPI_INT, MPI_COMM_WORLD);
        //printf("I am %d contributing data value %d to be gathered \n",myRank, value);
        //wait the rest action until all display their contribution
        MPI_Barrier(MPI_COMM_WORLD);
        //let all the process to display the result
        int i;
        printf("I am %d getting the following data gathered from different processes\n", myRank);
        for(i = 0; i < numProcess; i++)
                printf("%d\t",data[i]);
        printf("\n==============================================\n\n");
        MPI_Finalize();
        return 0;
}
```

Collective Communications: Scatter

- Scatter data from one member to all member of a group
- Each process (including the root) receive ***sendcount*** objects each having ***sendtype*** data type and received by a buffer called ***recvbuffer***. Each processor prepared ***recvcount*** object space for the receive operation with data type ***recvtype***.



Collective Communications: Scatter

Fortran

```
MPI_SCATTER(sbuf, scount, stype, rbuf, rcount, rtype,
     root, comm, ierr)
```

C

```
int MPI_Scatter(const void* sbuf, int scount, \
     MPI_Datatype stype, void* rbuf, int rcount, \
     MPI_Datatype rtype, int root, MPI_Comm comm)
```

`sbuf`	is the address of the send buffer,
`scount`	is the number of elements to be sent to each process,
`stype`	is the data type of the send buffer elements,
`rbuf`	is the address of the receive buffer,
`rcount`	is the number of elements in the receive buffer,
`rtype`	is the data type of the receive buffer elements,
`root`	is the rank of the sending process, and
`comm`	is the communicator.

Note: sbuf, scount, stype are significant for the root process only.

Collective Communications: Scatter

- Sufficiently large data is prepared to be scattered
  Fortran

```
program scatter
        include "mpif.h"
        integer ::numProcess, myRank, i
        integer ::num_chars
        integer ::ierror
        integer ::value
        integer, allocatable, dimension(:)  :: pdata

        write(*,*) "Scatter example"
        call MPI_Init(ierror)
        call MPI_Comm_rank(MPI_COMM_WORLD, myRank,ierror)
        call MPI_Comm_size(MPI_COMM_WORLD, numProcess,ierror)

        if(myRank == 0)  then
                allocate(pdata(numProcess))
                do i = 1, numProcess
                        pdata(i) = 10 + 10*i
                end do
        end if
        call MPI_Scatter(pdata, 1, MPI_INT, value, 1, MPI_INT, 0, MPI_COMM_WORLD, ierror)
        write(*,*) "I am ", myRank, "getting the data value", value
        call MPI_Finalize(ierror)
end
```

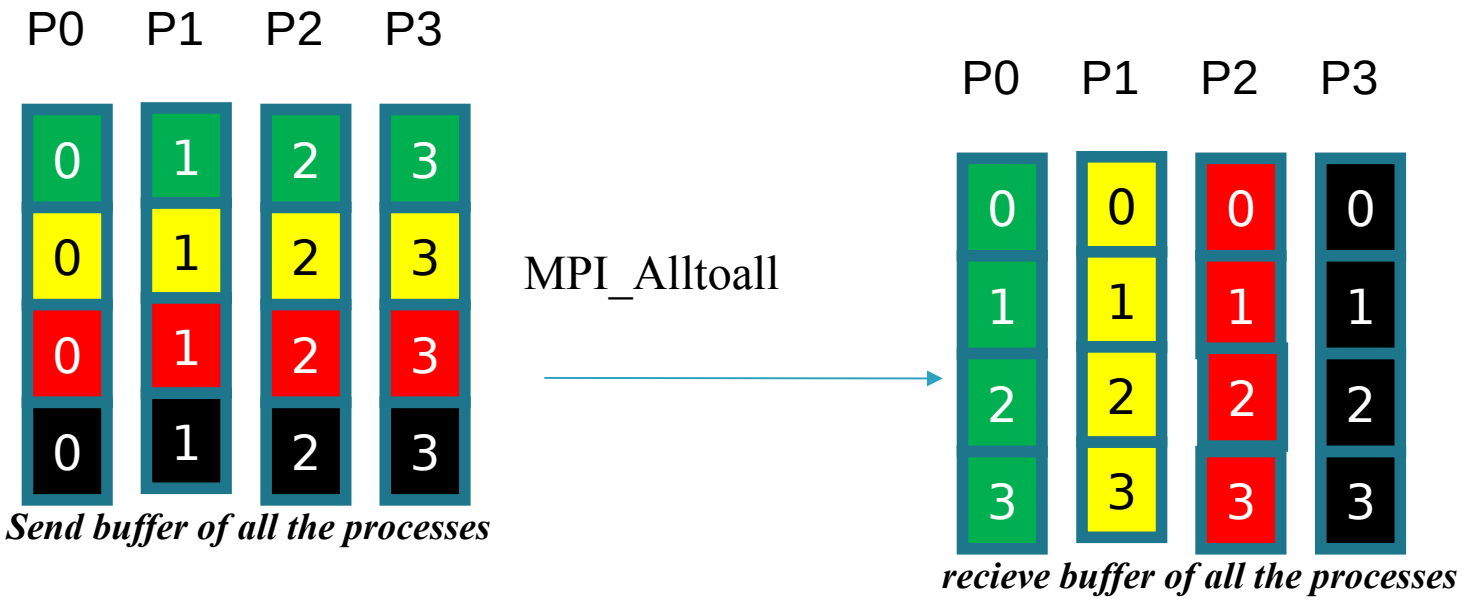
Collective Communications: Scatter

- Sufficiently large data is prepared to be scattered
  C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[]) {
        int myRank, numProcess, value , *data;
        printf("Scatter example \n");
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcess);
        if(myRank == 0) {
                int i;
                data = (int *) malloc(sizeof(int)*numProcess);
                for(i = 0; i < numProcess; i++){
                        data[i] = 10 + 10*i;
                }
        }
        MPI_Scatter(data, 1, MPI_INT, &value, 1, MPI_INT, 0, MPI_COMM_WORLD);
        printf("I am %d getting the data value %d \n", myRank, value);
        MPI_Finalize();
        return 0;
}
```

Collective Communications: Alltoall

- Scatter/Gather data from all members of a group (also called complete exchange or all-to-all)
- Process **i** sends the j**th** ***sendcount*** number of data elements to process **j** and process **j** receive data from process **i** and keep in ***recvbuffer*** after escaping **(i-1)****recvcount*** element spaces where each element has type ***recvtype***.



Collective Communications: Alltoall

Fortran

```
program alltoall
        include "mpif.h"
        integer ::numProcess, myRank, i
        integer ::num_chars
        integer ::ierror
        integer, allocatable, dimension(:)  ::value
        integer, allocatable, dimension(:)  :: pdata

        call MPI_Init(ierror)
        call MPI_Comm_rank(MPI_COMM_WORLD, myRank,ierror)
        call MPI_Comm_size(MPI_COMM_WORLD, numProcess,ierror)

        !arrange memory space for send buffer to all process
        allocate(value(numProcess))
        !arrange memory space for recieve buffer
        allocate(pdata(numProcess))
        do i = 1, numProcess
                value(i) = (2*myRank) + 10*(i)
                print*, "for process ", myRank, "Value[",i,"]=", value(i)
        end do
        call MPI_Alltoall(value, 1, MPI_INT, pdata, 1, MPI_INT, MPI_COMM_WORLD, ierror)
        call MPI_Barrier(MPI_COMM_WORLD, ierror)
        print*, "data after MPI_Alltoall"
        do i = 1, numProcess
                 print*, "for process ", myRank, "data[",i,"]=", value(i)
         end do

        call MPI_Finalize(ierror)
end
```

Collective Communications: Alltoall

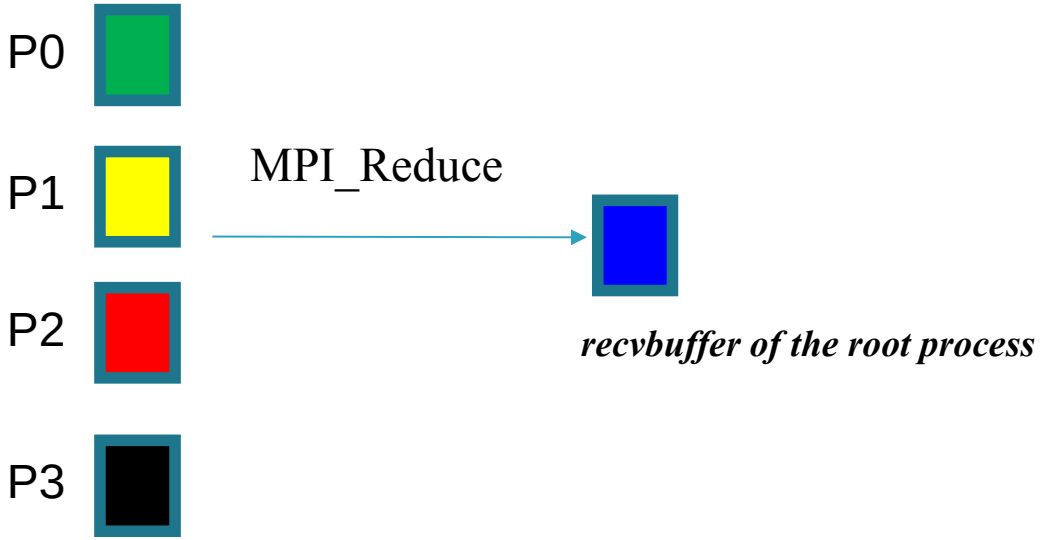
C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[]) {
        int myRank, numProcess, *value, *data;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcess);
        //arrange memory space for send buffer to all process
        value = (int *) malloc(numProcess*sizeof(int));
        //arrange memory space for receive buffer
        data = (int *) malloc(numProcess*sizeof(int));
        int i;
        for(i =0; i < numProcess; i++) {
                value[i] = (2*myRank) + 10 * (i+1);
                printf("for process %d Value[%d]= %d\n",myRank, i, value[i]);
        }
        MPI_Alltoall(value, 1, MPI_INT, data, 1, MPI_INT, MPI_COMM_WORLD);
        //wait the rest action until all display their contribution
        MPI_Barrier(MPI_COMM_WORLD);
        printf("data after MPI_Alltoall\n");
        for(i =0; i < numProcess; i++)
                printf("for process %d data[%d]= %d\n",myRank, i, data[i]);
        MPI_Finalize();
return 0;
}
```

Collective Communications: Reduce

- Reduction where the result is returned to all group members
- The i^{th} element of the send buffer from each process will be reduced into one element using the reduction operation **op** and kept into the i^{th} element of the receive buffer of the root process.
- Count is the number of elements in each processors buffer to be reduced and the number of elements obtained in the root processor after reduction

recvBuff[i] = op(A_0[i], A_1[i], …, A_n[i]) of the root



Collective Communications: Reduce

- Global Reduction Operation, Op can be

OP	Description	OP	Description
MPI_MAX	Maximum	MPI_MIN	Minimum
MPI_SUM	Sum	MPI_PRODUCT	Product
MPI_LAND	Logical and	MPI_BAND	Bitwise and
MPI_LOR	Logical or	MPI_BOR	Bitwise or
MPI_LXOR	Logical XOR	MPI_BXOR	Bitwise XOR
MPI_MAXLOC	maximum value and location		
MPI_MINLOC	minimum value and location		

Collective Communications: Reduce

C

```
int MPI_Reduce(const void* sbuf, void* rbuf, int count, \
     MPI_Datatype stype, MPI_Op op, int root, MPI_Comm comm)

int MPI_Allreduce(const void* sbuf, void* rbuf, int count \
     MPI_Datatype stype, MPI_Op op, MPI_Comm comm)

int MPI_Reduce_scatter_block(const void* sbuf, void* rbuf, \
     int rcount, MPI_Datatype stype, MPI_Op op, MPI_Comm comm)

int MPI_Reduce_scatter(const void* sbuf, void* rbuf, \
     const int[] rcount, MPI_Datatype stype, MPI_Op op, \
     MPI_Comm comm)
```

FORTRAN

```
MPI_REDUCE(sbuf, rbuf, count, stype, op, root, comm, ierr)

MPI_ALLREDUCE(sbuf, rbuf, count, stype, op, comm, ierr)

MPI_REDUCE_SCATTER(sbuf, rbuf, rcount, stype, op, comm, ierr)
```

In the above:

`sbuf`	is the address of send buffer,
`rbuf`	is the address of receive buffer,
`count`	is the number of elements in send buffer, OR
`rcount`	are the numbers of elements to be scattered back,
`stype`	is the data type of elements of send buffer,
`op`	is the reduce operation (MPI predefined, or your own),
`root`	is the rank of the root process,
`comm`	is the group communicator.

Notes:

- `rbuf` is significant only at the root process for MPI_Reduce.

Collective Communications: Reduce

Fortran

```
program mpi_reduce
        include "mpif.h"
        integer ::numProcess, myRank, i
        integer ::num_chars
        integer ::ierror
        integer :: value, pdata

        call MPI_Init(ierror)
        call MPI_Comm_rank(MPI_COMM_WORLD, myRank,ierror)
        call MPI_Comm_size(MPI_COMM_WORLD, numProcess,ierror)

        !set what each process to contribute
        value = (myRank + 1)*10

        ! the value in value of each process will be reduced into data of process zero using the MPI_SUM
        ! aggregate value
        call MPI_Reduce(value, pdata, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD, ierror)
        print*, "I am ", myRank, "contributing data value",value,"for reduction"
        !wait the rest action until all display their contribution
        call MPI_Barrier(MPI_COMM_WORLD, ierror)
        !let the first process to display the result
        if(myRank == 0) then
                print*, "I am ", myRank, "getting the reduced value which is ",pdata
        end if

        call MPI_Finalize(ierror)
end
```

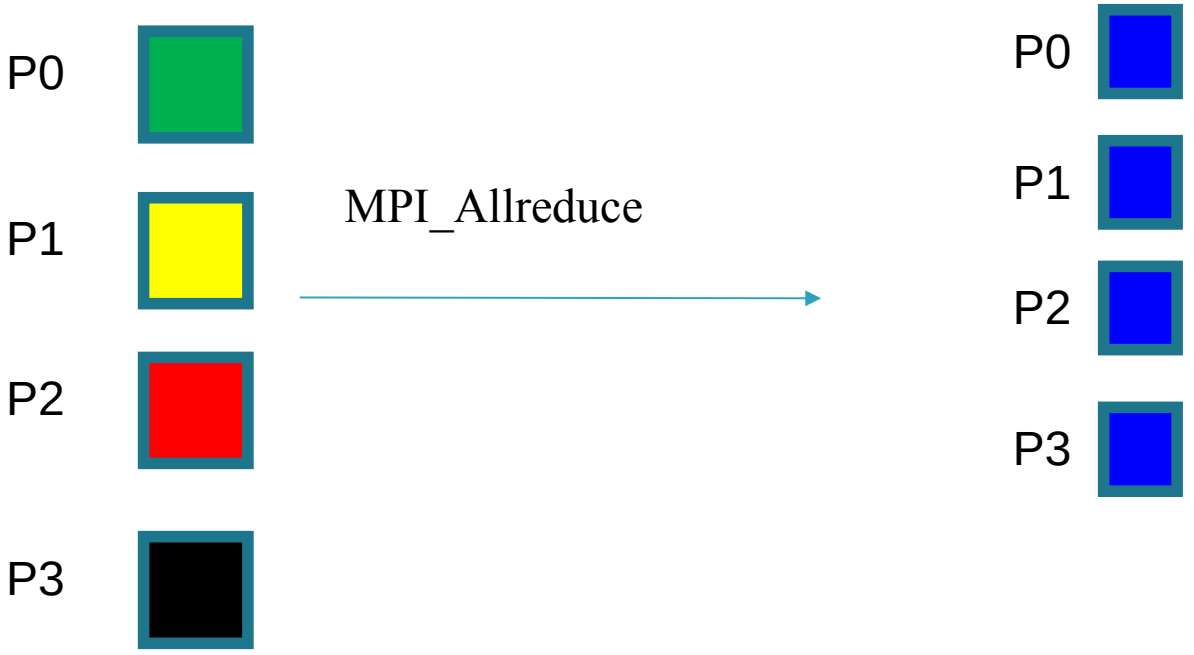
Collective Communications: Reduce

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[]) {
        int myRank, numProcess,value, data;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcess);

        //set what each process to contribute
        value = (myRank + 1) *10;

        //the value in value of each process will be reduced into data of process zero using the MPI_SUM aggregate value
        MPI_Reduce(&value, &data, 1, MPI_INT, MPI_SUM, 0,MPI_COMM_WORLD);
        printf("I am %dcontributing data value %d for reduction \n",myRank, value);

        //wait the rest action until all display their contribution
        MPI_Barrier(MPI_COMM_WORLD);

        //let the first process to display the result
        if(myRank ==0)
        {
                printf("I am %d getting the reduced value which is %d \n", myRank, data);
        }
        MPI_Finalize();
        return 0;
}
```

Collective Communications: Allreduce

- A variation of reduce where all member of the group receive the reduced data
- This does the same as above except all process will have the same value copy of receive buffer.



recvbuffer of the root process

recvBuff[i] = op(A_0[i], A_1[i], …, A_n[i])
of all process

Collective Communications: Allreduce

Fortran

```
program mpi_reduce
        include "mpif.h"
        integer ::numProcess, myRank, i
        integer ::num_chars
        integer ::ierror
        integer :: value, pdata

        call MPI_Init(ierror)
        call MPI_Comm_rank(MPI_COMM_WORLD, myRank,ierror)
        call MPI_Comm_size(MPI_COMM_WORLD, numProcess,ierror)

        !set what each process to contribute
        value = (myRank + 1)*10

        ! the value in value of each process will be reduced into data of process zero using the MPI_SUM
        ! aggregate value and the reduce value scattered to all process
        print*, "I am ", myRank, "contributing data value",value,"for reduction"
        call MPI_Allreduce(value, pdata, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD, ierror)
        !wait the rest action until all display their contribution
        call MPI_Barrier(MPI_COMM_WORLD, ierror)
        !let the all process display the result
        print*, "I am ", myRank, "getting the reduced value which is ",pdata

        call MPI_Finalize(ierror)
end
```

Collective Communications: Allreduce

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[]) {
        int myRank, numProcess,value, data;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcess);
        value = (myRank + 1) *10; //set what each process to contribute
   //the value in value of each process will be reduced into data of process zero using the MPI_SUM aggregate value
   MPI_Allreduce(&value, &data, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
        printf("I am %d contributing data value %d for reduction \n",myRank, value);
        MPI_Barrier(MPI_COMM_WORLD);          //wait the rest action until all display their contribution
        printf("I am %d  getting the reduced value which %d \n", myRank, data);
        MPI_Finalize();
        return 0;
}
```

Collective Communications: Reduce_scatter

- A combined reduce and Scatter operation

  int MPI_Reduce_scatter(void sbuff, void* rbuff, int rcount, MPI_Datatype stype, MPI_Op op, MPI_Comm comm)*

  MPI_Reduce_scatter(sbuff, rbuff, rcount, stype, op, comm, ierr)

- This function takes the first ***rcount[0]*** data element from each process's ***sbuff*** and reduce each of the corresponding elements using MPI_OP and keep into process 0 ***rbuff***.
- Take the next ***rcount[1]*** data element and do the same to ***rbuffer*** of process 1.
- Continue likewise to all processors

Collective Communications: Reduce_scatter

Consider we have 4 processor each having the list of data shown bellow

P0:	1	2	3	4	5	6	7	8	9	10
P1:	2	3	4	5	6	7	8	9	10	11
P2:	3	4	5	6	7	8	9	10	11	12
P3:	4	5	6	7	8	9	10	11	12	13

Consider recvCount[0]= recvCount[1]= recvCount[2]= recvCount[3]= 2

MPI_Reduce_scatter first reduce each element (Assuming sum, it results)

10	14	18	22	26	30	34	38	42	46

Then each of the will be scattered to all processors two for one processor

P0:	10	14
P1:	18	22
P2:	26	30
P3:	34	38

Collective Communications: Reduce_scatter

Fortran

```
program mpi_reduce
        include "mpif.h"
        integer ::numProcess, myRank, i
        integer, parameter :: SIZE = 16
        integer ::num_chars
        integer ::ierror
        integer :: pdata(SIZE), value(size), recCount(size)

        call MPI_Init(ierror)
        call MPI_Comm_rank(MPI_COMM_WORLD, myRank,ierror)
        call MPI_Comm_size(MPI_COMM_WORLD, numProcess,ierror)

        !set what each process to contribute
        do i = 1, SIZE
                value(i)    = (myRank + 1) + i
                recCount(i) = Size/numProcess
        end do
        print*, "I am", myRank, "having the data value "
        do i = 1, SIZE
                print*, value(i)
        end do
        print*, "I am ", myRank, "contributing data value",value,"for reduction"
        call MPI_Reduce_scatter(value, pdata, recCount, MPI_INT, MPI_SUM, MPI_COMM_WORLD, ierror)
        !wait the rest action until all display their contribution
        call MPI_Barrier(MPI_COMM_WORLD, ierror)
        !let the first process to display the result
        do i = 1, SIZE/numProcess
                write(6,100)  pdata(i)
        end do
        100 FORMAT(1X,2I5)
        call MPI_Finalize(ierror)
end
```

Collective Communications: Reduce_scatter

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
#define SIZE 16
int main(int argc, char* argv[]) {
        int i,myRank, numProcess;
        int value[SIZE], data[SIZE],recCount[SIZE];
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcess);
        for(i = 0; i < SIZE; i++){
                value[i] = (myRank + 1) + i;
                recCount[i] = SIZE/numProcess;
        }
        printf("I am %d having the data value \n", myRank);
        for(i = 0; i < SIZE; i++) printf("%d  ", value[i]);
  printf("\n ");
MPI_Reduce_scatter(&value, &data, recCount, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
        //wait the rest action until all display their contribution
        MPI_Barrier(MPI_COMM_WORLD);
        printf("I am %d getting the reduced and scattered value \n", myRank);
        for(i =0; i < SIZE/numProcess; i++)printf("%d ",data[i]);
        printf("\n");
        MPI_Finalize();
        return 0;
}
```