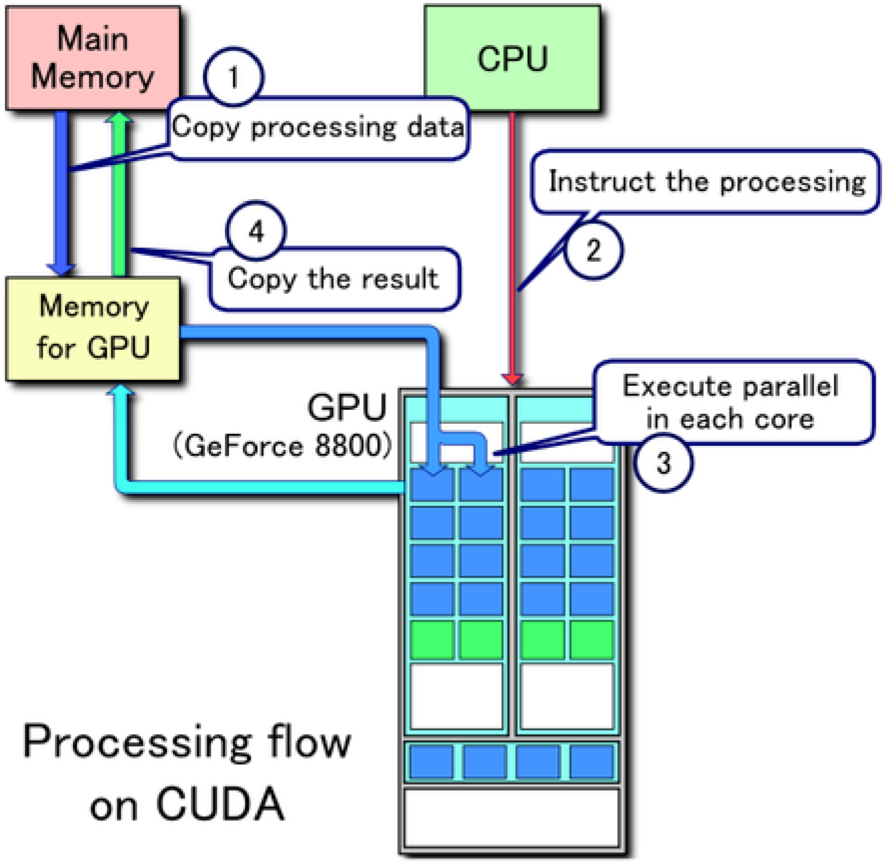
# Message Passing Programming

6 Hours


Main Memory
CPU
1
Copy processing data
Instruct the processing
2
4
Copy the result
Memory for GPU
GPU
(GeForce 8800)
Execute parallel in each core
3
Processing flow on CUDA

# Topics covered:

1. Introduction to Message Passing Interface (MPI)
2. Message Passing Model
3. MPI Basic Datatypes and Functions
4. Point-to-point Communication
5. Collective Communication
6. Benchmarking Parallel Performance
7. MPI Error Handling Functions

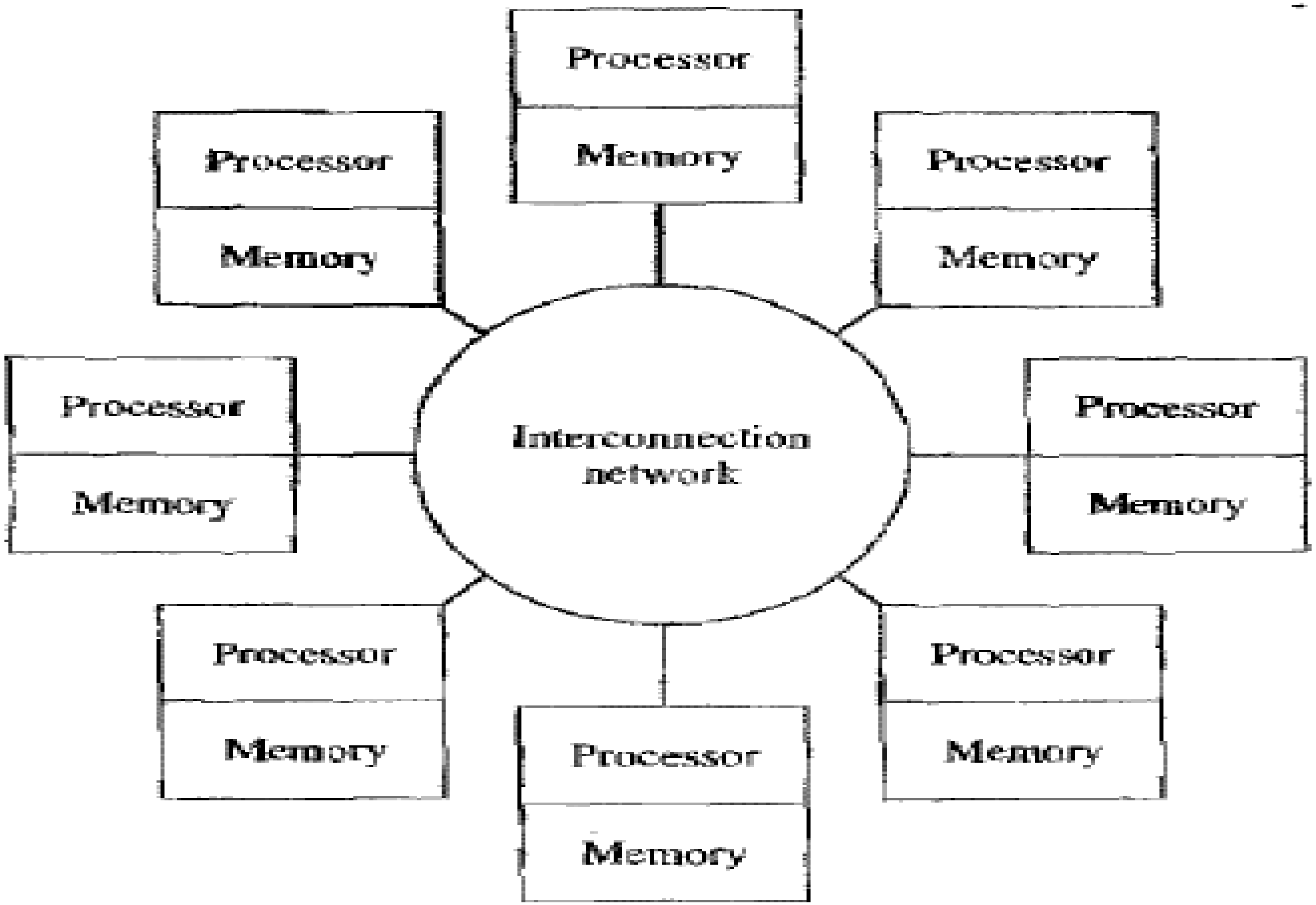
# Introduction to MPI

- The **MPI standard** is the most popular message-passing library interface specification supporting parallel programming

- MPI is a *message-passing parallel programming model*, in which data is moved from the address space of one process to that of another process through cooperative operations on each process

- **MPI is not a programming language**, and all MPI operations are expressed as functions, subroutines, or methods used by  C, C++, Fortran-77, and Fortran-95 etc which are part of MPI standard

# Message Passing Model



# Message Passing Model

**Message-passing model**

- The underlying hardware is assumed to be a collection of processors, each with its own local memory
- A processor has direct access only to the instructions and data stored in its local memory
- However, an interconnection network supports message passing between processors
- Processor **A** may send a message containing some of its local data values to processor **B**, giving processor B indirect access to these values
- The existence of the interconnection network provides an implicit communication channel between every pair of processes

# Message Passing Model

**Message-passing model**

- The user specifies the number of concurrent processes when the program begins, and the number of active processes remains constant throughout the execution of the program

- Every process executes the same program, but because each one has a unique ID number, different processes may perform different operations on the same program

- In a message-passing model, processes pass messages both to communicate and to synchronize with each other

# ***Advantages of message-passing model***

- Message-passing programs run well on a wide variety of *MIMD architectures*

- They are a natural fit for *multicomputers*, which *do not support a global address space*

- The MPI programs tend to exhibit *high cache hit rates* when executing on multiprocessors, leading to good performance

- *Debugging MPI programs is simpler* than debugging shared-variable programs. Since each process controls its own memory, it is not possible for one process to accidentally overwrite a variable controlled by another process, a common bug in shared-variable programs.

# *Key concepts of MPI programming*

- Used to create *parallel programs*
- Processors communicate *using message passing* via calls to message passing library routines
- Programmers *"parallelize"* programs by adding message calls between *manager process* and *worker process*
- No process can be *created or terminated* in the middle of program execution
- All process *stay alive* till the program terminates
- Each processor has a *local memory* to which it has exclusive access
- The MPI programs tend to exhibit *high cache hit rates* when executing on multiprocessors, leading to good performance
- The number of processes *is fixed* when starting the program

# MPI Naming Conventions, Basic Datatypes and Routines

***MPI Naming Conventions***

- The names of all **MPI entities** (routines, constants, types, etc.) begin with **MPI_** to avoid conflicts

  **Example:** **MPI_Init(&argc, &argv)**

- All **MPI constants** are strings of capital letters and underscores beginning with **MPI_**

  **Example:** **MPI_COMM_WORLD**

## *Predefined data types for MPI*

| MPI Datatype | C-Data type |
| --- | --- |
| •MPI_CHAR | signed char |
| •MPI_SHORT | signed short int |
| •MPI_INT | signed int |
| •MPI_LONG | signed long int |
| •MPI_LONG_LONG_INT | long long int |
| •MPI_UNSIGNED_CHAR | unsigned char |
| •MPI_UNSIGNED_SHORT | unsigned short int |
| •MPI_UNSIGNED | unsigned int |
| •MPI_UNSIGNED_LONG | unsigned long int |
| •MPI_UNSIGNED_LONG_LONG | unsigned long long int |
| •MPI_FLOAT | float |
| •MPI_DOUBLE | double |
| •MPI_LONG_DOUBLE | long double |
| •MPI_WCHAR | wide char |
| •MPI_PACKED | special data type for packing |
| •MPI_BYTE | single byte value |

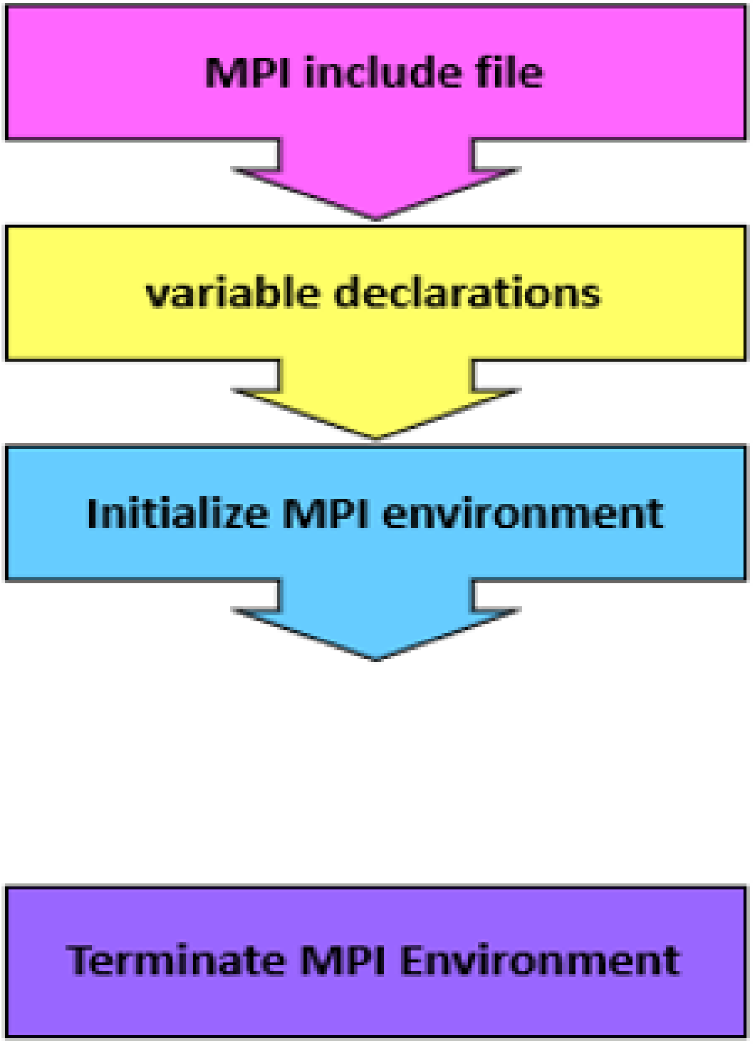
## *MPI routines*

- MPI routines are implemented as **functions** which return the *exit status* of the function call

```
int ierr;
...
ierr = MPI_Init(&argc, &argv);
```

- The *error code* returned is **MPI_SUCCESS** if the routine ran successfully or else the integer returned has an implementation-dependent value indicating the specific error

# General MPI Program Structure




```
#include <mpi.h>

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

MPI_Init(&argc, &argv);

.
.      // Run parallel code
.

MPI_Finalize(); // End MPI Envir

return 0;
}
```

# Basic Environment

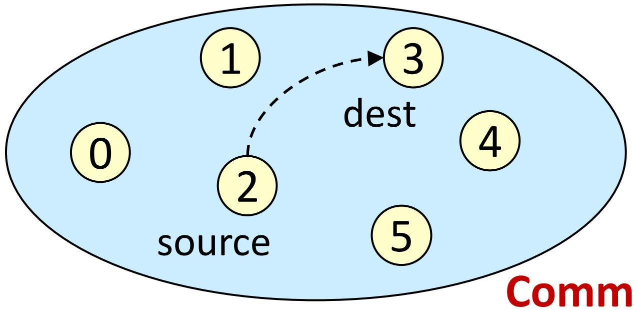
```
MPI_Init(&argc, &argv)
```

- Must be called in every MPI program
- It should be the *first MPI function* call made by every MPI process
- It initializes *MPI environment*
- Can be used to pass command line arguments to all

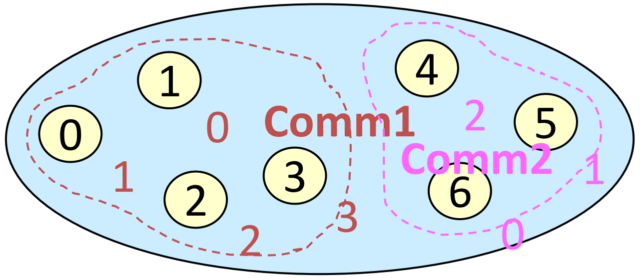
```
MPI_Finalize()
```

- It *terminates MPI environment* after releasing all the held up resources
- It should be the *last* MPI function call

# Communicators & Rank




Communicator




MPI_COMM_WORLD

```
MPI_COMM_WORLD
```

- When MPI has been initialized, every active process becomes a member of a communicator called MPI_COMM_WORLD
- A communicator is an object that provides the environment for message passing among processes
- MPI_COMM_WORLD is the *default communicator* that you get "for free”
- However, you can create your own communicators if you need to partition the processes into independent communication groups

# Communicators & Rank

```
What is rank of a process??
```

- Processes within a communicator are *always ordered*
- The rank of a process is *its position* in the overall order
- In a communicator with ***p*** processes, each process has a unique rank (ID number) between **0** and ***p* – 1**
- A process may use its rank to determine which portion of a computation and/or a dataset it is responsible for

# Communicators & Rank

int my_rank, size;

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)
```

- A process calls this function to determine *its rank* within a communicator

```
MPI_Comm_size(MPI_COMM_WORLD, &size)
```

- A process calls this function to determine *the total number of processes* in a communicator

```
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

  printf("Processor %d of %d: Hello World!\n", rank, size);

  MPI_Finalize(); //MPI cleanup

  return 0;
}

MPI_Init(int *argc, char ***argv);
```

# Hello World for MPI

- Running this code on four processors will produce a result like:

```
mpicc –o prg1 program1.c
mpirun -n 4 prg1
```

```
Processor 2 of 4: Hello World!
Processor 1 of 4: Hello World!
Processor 3 of 4: Hello World!
Processor 0 of 4: Hello World!
```

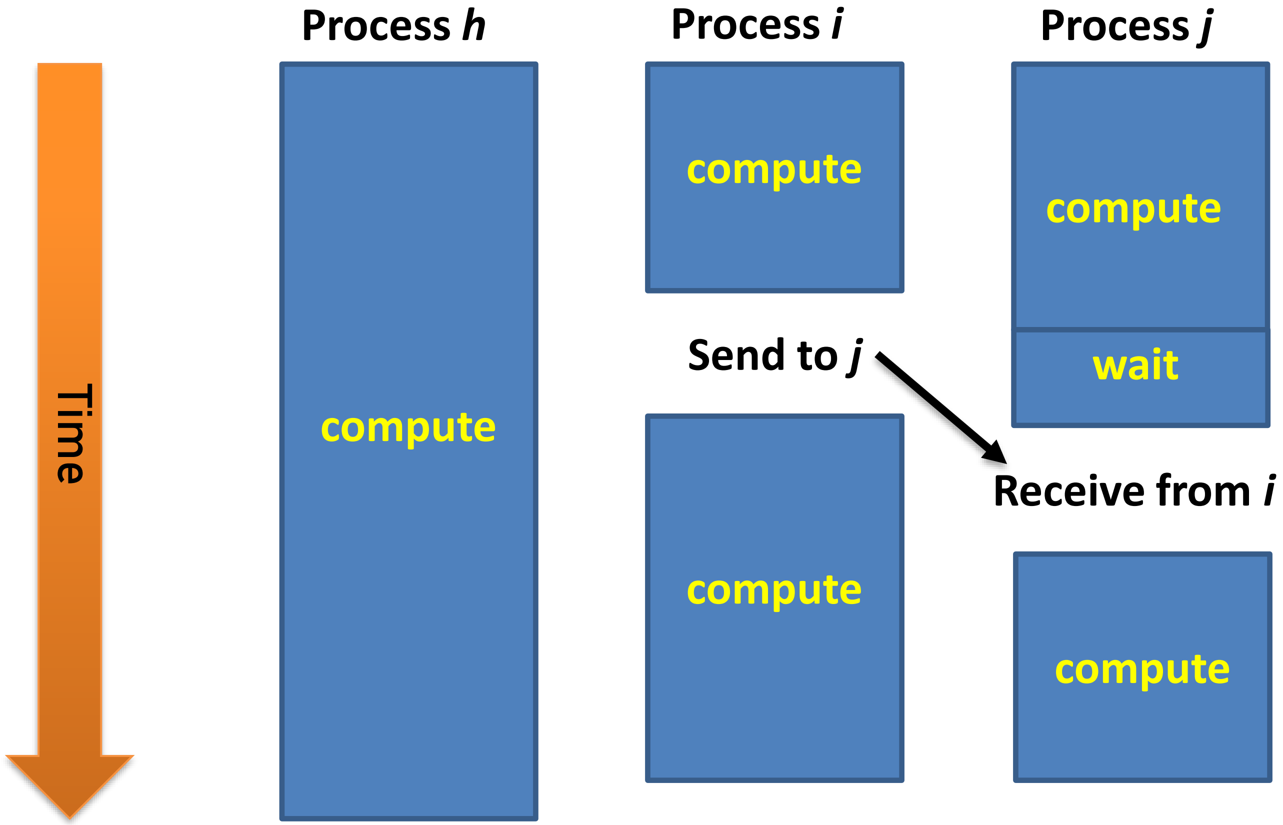
- Each processor executes the same code, including probing for its rank and size and printing the string.

- The order of the printed lines is essentially random!

# Point-to-point Communication in MPI

- A point-to-point communication involves a *pair of processes*

- In the following example, process ***h*** is not involved in a communication. It continues executing statement, manipulating its local variables. Process ***i*** performs local computations, then sends a message to process ***j.*** After the message is sent, it continues on with its computation. Process ***j*** performs local computations, then blocks until it receives a message from process ***i.***



# Blocking Message Passing Routines

## MPI_Send

```
MPI_Send(void *message,    //address of data to be transmitted
             int   count,         //number of data items
             MPI_Datatype  datatype, // type of data to be transmitted
             int   dest, // rank of the process to receive the data
             int   tag, // integer label for the message
             MPI_Comm  comm // communicator )
```

- This routine sends a message and *block* until the application buffer in the sending task is *free* for reuse
- The MPI implementation may buffer your send allowing it to return almost immediately
- If the implementation *does not buffer the send*, the send will not complete until the matching receive occurs

# Blocking Message Passing Routines

## MPI_Recv

```
MPI_Recv(void *message,   //address of where the data to be received
            int   count,         //maximum number of data items to be received
            MPI_Datatype  datatype, // type of data to be received
            int   source, // rank of the process sending the data
            int   tag, // integer label for the message
            MPI_Comm  comm // communicator
            MPI_Status  *Status // status information of data received )
```

- This routine returns *only after* the *requested data is available* in the application buffer

- The *status record* contains information about the just-completed function. In particular:

1. status- >MPI_source is the rank of the process sending the message
2. status->MPI_tag is the message's tag value
3. status- >MPI_ERROR is the error condition

MPI_ANY_SOURCE

MPI_ANY_TAG

# Synchronous Message Passing Routines

## MPI_Ssend

```
MPI_Ssend(void *message,   //address of data to be transmitted
          int  count,         //number of data items
          MPI_Datatype  datatype, // type of data to be transmitted
          int  dest, // rank of the process to receive the data
          int  tag, // integer label for the message
          MPI_Comm  comm // communicator )
```

- This routing sends a message and *block* until the application buffer in the sending task is *free* for reuse and the destination process has started to receive the message

# Synchronous Message Passing Routines

- Write a MPI program where the master process (process 0) sends a number to each of the slaves and the slave processes receive the number and prints it.

Example: number of process :4

Root reads process : 2     P1,P2,P3 : receives 2 and prints

- Modify the above program such that slave processes increment this value by their rank and return back to root process.

P1 : receives 2          returns : 2+1 = 3

P2: receives 2          returns : 2+2 = 4

P3 : receives 2          returns : 2 + 3 = 5

# Buffered Message Passing Routines

## MPI_Bsend

```
MPI_Bsend(void *message,   //address of data to be transmitted
          int  count,         //number of data items
          MPI_Datatype  datatype, // type of data to be transmitted
          int  dest, // rank of the process to receive the data
          int  tag, // integer label for the message
          MPI_Comm  comm // communicator )
```

- This routine permits the programmer to *allocate the required amount of buffer space* into which data can be copied until it is delivered
- Insulates against the problems associated with *insufficient system buffer space*
- Routine returns after the data has been copied from application buffer space to the allocated send buffer
- It must be used with the **MPI_Buffer_attach()** and **MPI_Buffer_detach()** routines

# Buffered Message Passing Routines

## MPI_Buffer_attach

```
MPI_Buffer_attach (void *buffer, // address of the buffer
                   int size      // buffer size in  bytes )
```

## MPI_Buffer_detach

```
MPI_Buffer_detach (void *buffer, // address of the buffer
                   int *size     // buffer size in  bytes )
```

- Used by programmer to attach/detach message to the buffer space to be used by the **MPI_Bsend( )** routine
- The *size* argument is specified in actual data bytes - not a count of data elements
- *Only one buffer* can be attached to a process at a time

# Deadlock

"A process is in a deadlock state if it is blocked waiting for a condition that will never become true"

# Deadlock ()

```
int a,b,c;
int rank;
MPI_Status status;
………….
if(rank==0)
{
       MPI_Recv(&b,1,MPI_INT,1,0,MPI_COMM_WORLD,&status);
       MPI_Send(&a,1,MPI_INT,1,0,MPI_COMM_WORLD);
       c=a+b/2;
}
else if(rank==1)
{
       MPI_Recv (&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
       MPI_Send(&b,1,MPI_INT,0,0,MPI_COMM_WORLD);
       c=a+b/2;
}
```

# Deadlock (Recv-Recv)

```
int a,b,c;
int rank;
MPI_Status status;
………….
if(rank==0)
{
      MPI_Recv(&b,1,MPI_INT,1,0,MPI_COMM_WORLD,&status);
      MPI_Send(&a,1,MPI_INT,1,0,MPI_COMM_WORLD);
      c=a+b/2;
}
else if(rank==1)
{
      MPI_Recv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
      MPI_Send(&b,1,MPI_INT,0,0,MPI_COMM_WORLD);
      c=a+b/2;
}
```

# Deadlock ()

```
int a,b,c;
int rank;
MPI_Status status;
………….
if(rank==0)
{
        MPI_Send(&a,1,MPI_INT,1,1,MPI_COMM_WORLD);
        MPI_Recv(&b,1,MPI_INT,1,1,MPI_COMM_WORLD,&status);
        c=a+b/2;
}
else if(rank==1)
{
        MPI_Send(&b,1,MPI_INT,0,0,MPI_COMM_WORLD);
        MPI_Recv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
        c=a+b/2;
}
```

# Deadlock (Tag mismatch)

```
int a,b,c;
int rank;
MPI_Status status;
…………..
if(rank==0)
{
      MPI_Send(&a,1,MPI_INT,1,1,MPI_COMM_WORLD);
      MPI_Recv(&b,1,MPI_INT,1,1,MPI_COMM_WORLD,&status);
      c=a+b/2;
}
else if(rank==1)
{
      MPI_Send(&b,1,MPI_INT,0,0,MPI_COMM_WORLD);
      MPI_Recv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
      c=a+b/2;
}
```

# Deadlock ()

```
int a,b,c;
int rank;
MPI_Status status;
………….
if(rank==0)
{
      MPI_Send(&a,1,MPI_INT,2,1,MPI_COMM_WORLD);
      MPI_Recv(&b,1,MPI_INT,2,1,MPI_COMM_WORLD,&status);
      c=a+b/2;
}
else if(rank==1)
{
      MPI_Send(&b,1,MPI_INT,0,0,MPI_COMM_WORLD);
      MPI_Recv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
      c=a+b/2;
}
```

# Deadlock (Rank mismatch)

```
int a,b,c;
int rank;
MPI_Status status;
…………..
if(rank==0)
{
        MPI_Send(&a,1,MPI_INT,2,1,MPI_COMM_WORLD);
        MPI_Recv(&b,1,MPI_INT,2,1,MPI_COMM_WORLD,&status);
        c=a+b/2;
}
else if(rank==1)
{
        MPI_Send(&b,1,MPI_INT,0,0,MPI_COMM_WORLD);
        MPI_Recv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
        c=a+b/2;
}
```

# Deadlock ()

```
int a,b,c;
int rank;
MPI_Status status;
…………..
if(rank==0)
{
      MPI_Send(&a,1,MPI_INT,1,1,My_Communicator);
      MPI_Recv(&b,1,MPI_INT,1,1,MPI_COMM_WORLD,&status);
      c=a+b/2;
}
else if(rank==1)
{
      MPI_Send(&b,1,MPI_INT,0,1,MPI_COMM_WORLD);
      MPI_Recv(&a,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
      c=a+b/2;
}
```

# Deadlock (Communicator mismatch)

```
int a,b,c;
int rank;
MPI_Status status;
………….
if(rank==0)
{
      MPI_Send(&a,1,MPI_INT,1,1,My_Communicator);
      MPI_Recv(&b,1,MPI_INT,1,1,MPI_COMM_WORLD,&status);
      c=a+b/2;
}
else if(rank==1)
{
      MPI_Send(&b,1,MPI_INT,0,0,MPI_COMM_WORLD);
      MPI_Recv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
      c=a+b/2;
}
```

# Deadlock ()

```
int a,b,c;
int rank;
MPI_Status status;
………….
if(rank==0)
{
      MPI_Send(&a,1,MPI_INT,0,1,MPI_COMM_WORLD);
      MPI_Recv(&b,1,MPI_INT,1,1,MPI_COMM_WORLD,&status);
      c=a+b/2;
}
else if(rank==1)
{
      MPI_Send(&b,1,MPI_INT,0,0,MPI_COMM_WORLD);
      MPI_Recv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
      c=a+b/2;
}
```

# Deadlock (self blocking Send)

```
int a,b,c;
int rank;
MPI_Status status;
…………..
if(rank==0)
{
        MPI_Send(&a,1,MPI_INT,0,1,MPI_COMM_WORLD);
        MPI_Recv(&b,1,MPI_INT,1,1,MPI_COMM_WORLD,&status);
        c=a+b/2;
}
else if(rank==1)
{
        MPI_Send(&b,1,MPI_INT,0,0,MPI_COMM_WORLD);
        MPI_Recv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
        c=a+b/2;
}
```

# point-to-point Communication Example

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

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

    int rank, size, my_number;


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

    if(rank == 0){
            my_number = 777;
            MPI_Send(&my_number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    else if (world_rank == 1) {
            MPI_Recv(&my_number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("Process 1 received number %d from process 0\n", number);
    }

    MPI_Finalize();

    return 0;
}
```

# Collective Communication in MPI

- A collective communication is a communication operation in which *a group of processes works together* to **distribute** or **gather** together a set of one or more values

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

# Types of Collective Operations

1. **Synchronization:**
   processes wait until all members of the group have reached the synchronization point

2. **Data Movement:**
   processes send/receive data among themselves

3. **Collective Computation:**
   one or more member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data

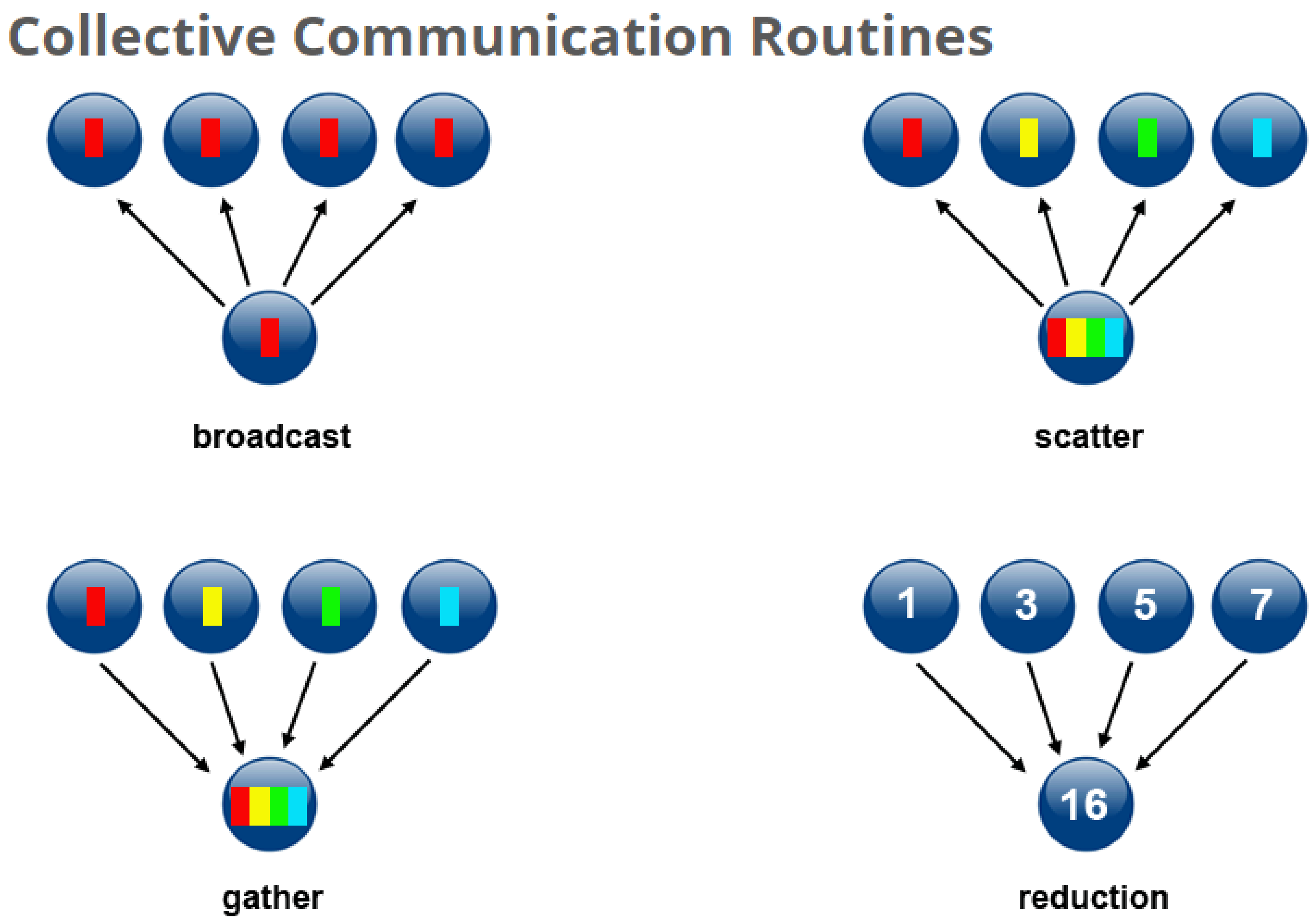
# Predefined MPI reduction operators

| Operator | Meaning |
|---|---|
| **MPI_BAND** | Bitwise and |
| **MPI_BOR** | Bitwise or |
| **MPI_BXOR** | Bitwise exclusive or |
| **MPI_LAND** | Logical and |
| **MPI_LOR** | Logical or |
| **MPI_LXOR** | Logical exclusive or |
| **MPI_ MAX** | Maximum |
| **MPI_MAXLOC** | Maximum and location of maximum |
| **MPI_MIN** | Minimum |
| **MPI_MINLOC** | Minimum and location of minimum |
| **MPI_PROD** | Product |
| **MPI SUM** | Sum |

# Collective Communication Routines

| | |
|---|---|
| **MPI_Bcast()** | Broadcast data from root to all other processes |
| **MPI_Alltoall()** | Sends data from every processes to all processes |
| **MPI_Reduce()** | Combine values from all processes to a single value |
| **MPI_Scatter()** | Scatters buffer in parts to group of processes |
| **MPI_Gather()** | Gather values from group of processes |
| **MPI_Allgather()** | Every process gather values from all processes in a communicator |
| **MPI_Scan()** | Computes the scan (partial reductions) of data on a collection of processes |

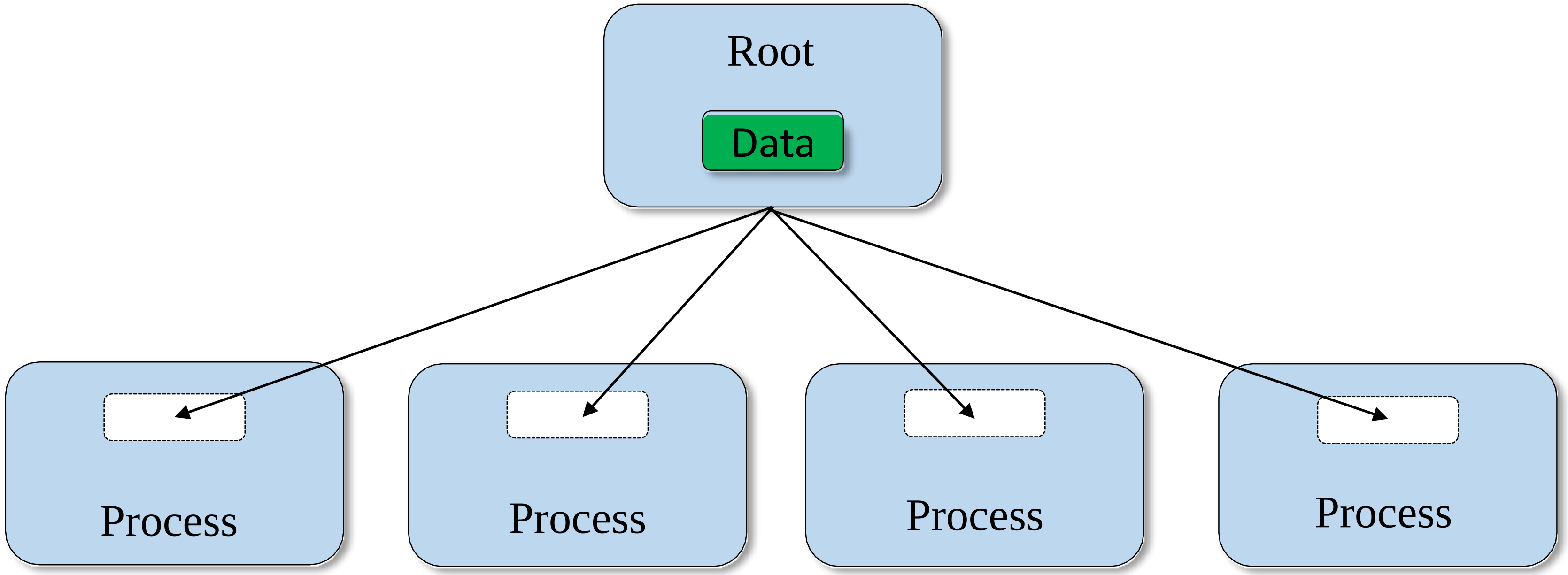
# Collective Communications



# MPI_Bcast (one-to-all)

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

- Broadcasts a message from process with rank ***root*** in ***comm*** to all other processes in *comm*.
- One process (root) sends data to all the other processes in the same communicator
- Must be called by all the processes *with the same arguments Data*

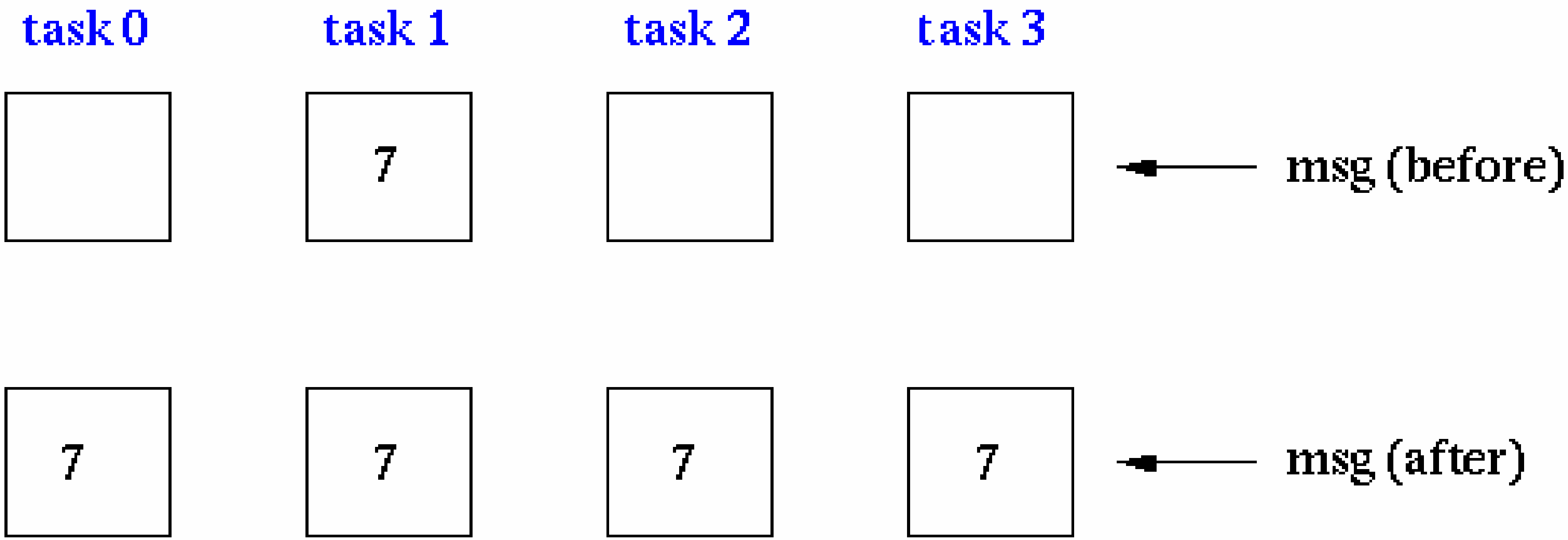



**buffer** → starting address of buffer
**count** → number of entries in buffer (integer)
**datatype** → data type of buffer
**root** → rank of broadcast root (integer)
**comm** → communicator (handle)

# MPI_Bcast

**Broadcasts a message to all other processes of that group**

```
count = 1;
source = 1;                 broadcast originates in task 1
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

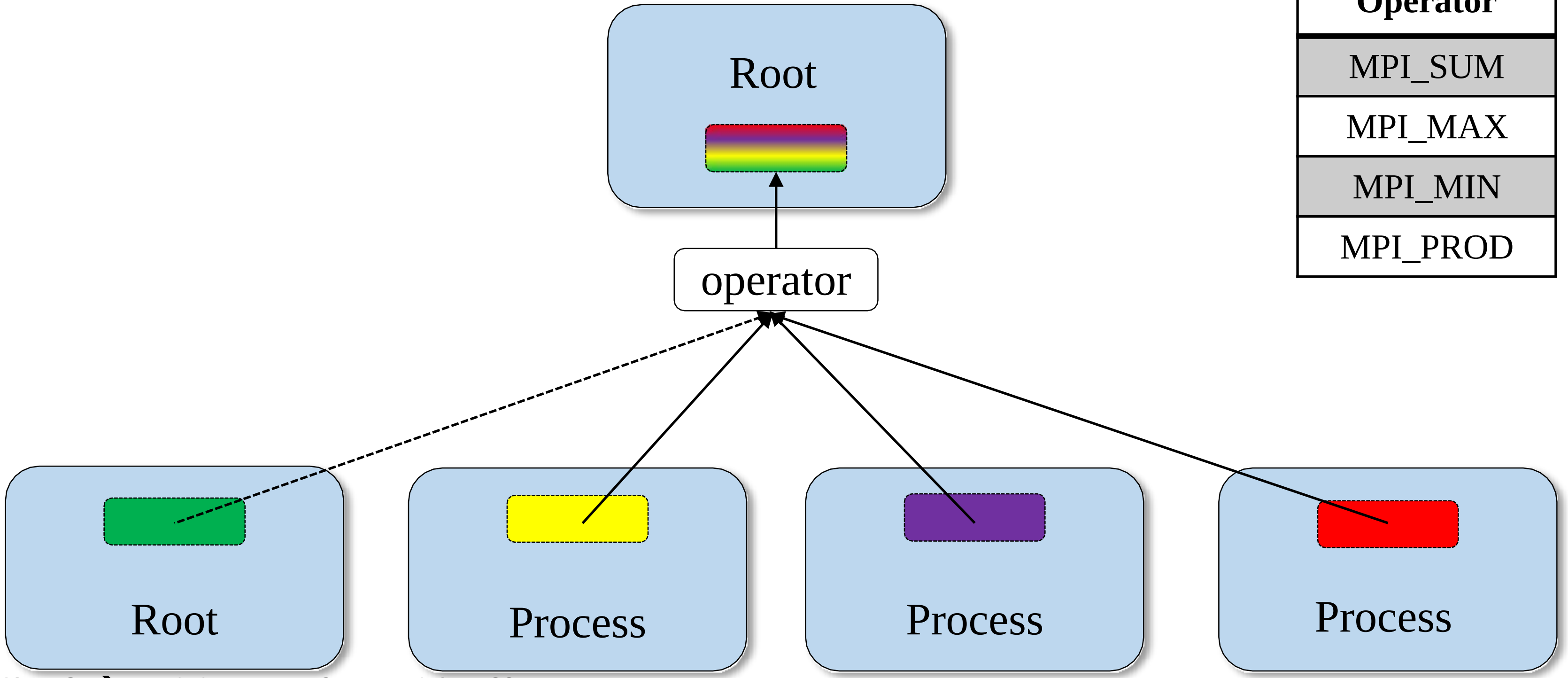


# MPI_Reduce

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

- One process (root) collects data from all the other processes in the same communicator, and performs an operation on the data (i.e combines elements provided by input buffer of each process in the group using operation *op*.)
- Returns combined value in the output buffer of process with rank *root*

| Operator |
|---|
| MPI_SUM |
| MPI_MAX |
| MPI_MIN |
| MPI_PROD |



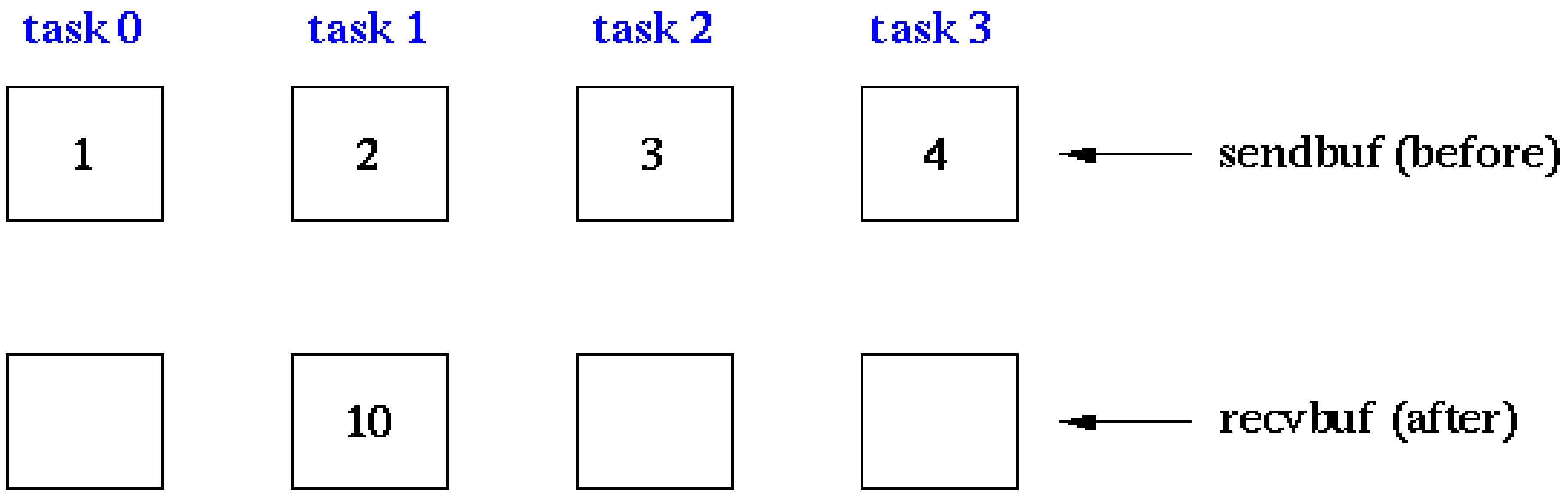

**sendbuf** → address of send buffer
**recvbuf** → address of result buffer
**count** → number of elements in send buffer (integer)
**datatype** → data type of send buffer
**op** → reduction operation
**root** → rank of broadcast root (integer)

# MPI_Reduce

Perform and associate reduction operation across all tasks in the group and place the result in one task

```
count = 1;
dest = 1;
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
           dest, MPI_COMM_WORLD);
```

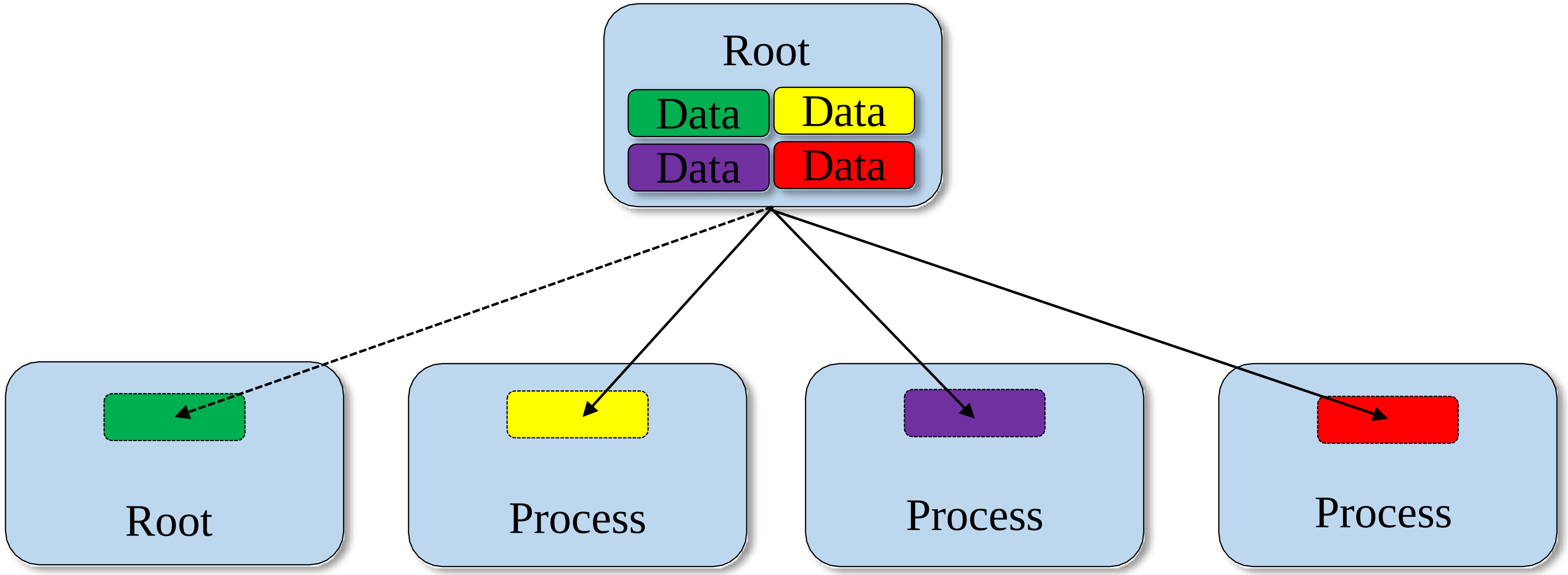
result will be placed in task 1



# MPI_Scatter

```
MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
             int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- Sends individual messages from the root process to all other processes
- Inverse to MPI_Gather
- *sendbuf* is ignored by all non-*root* processes




**sendbuf** → address of send buffer (significant only at root)
**sendcount** → number of elements sent to each process (significant only at root)
**sendtype** → data type of send buffer elements (significant only at root)

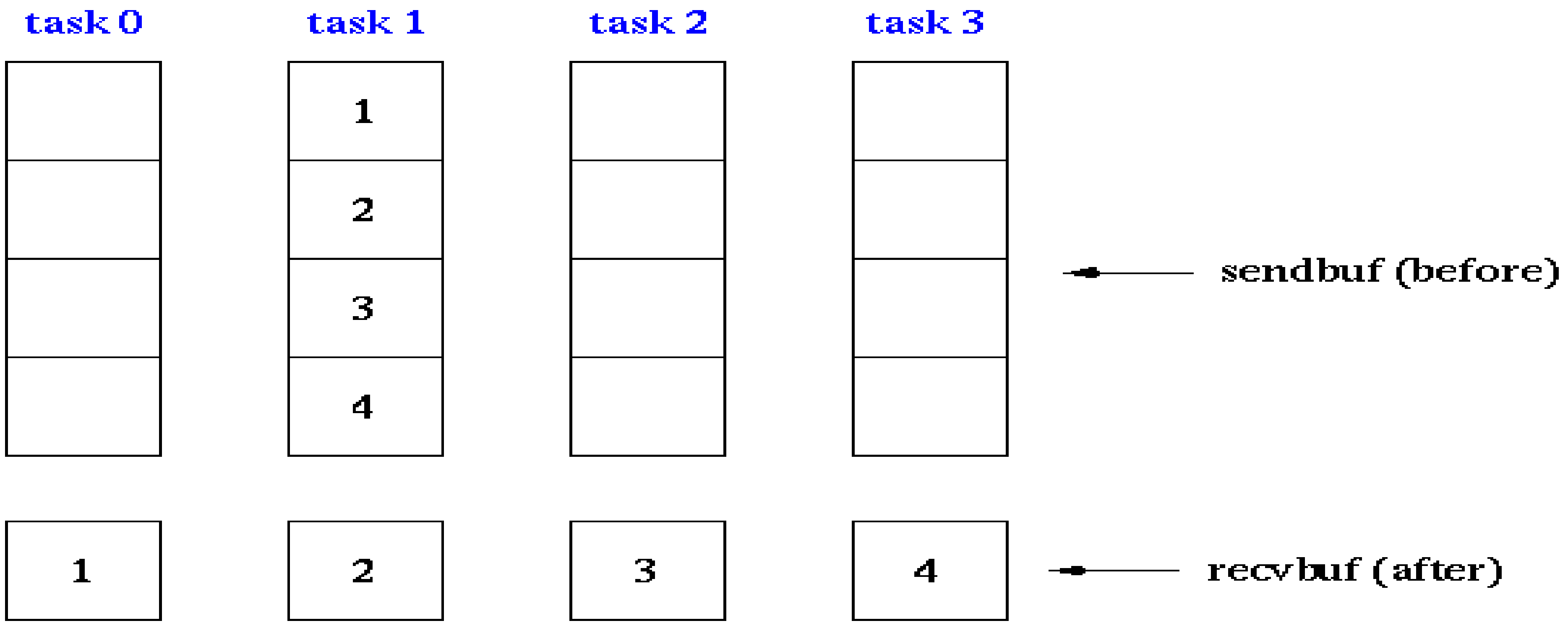
**recvcount** → number of elements in receive buffer (integer)
**recvtype** → data type of receive buffer elements
**sendtype** → data type of send buffer elements (significant only at root)
**root** → rank of sending process (integer)

# MPI_Scatter

**Sends data from one task to all other tasks in a group**

```
sendcnt = 1;
recvcnt = 1;
src  = 1;
MPI_Scatter(sendbuf, sendcnt, MPI_INT,
            recvbuf, recvcnt, MPI_INT,
            src, MPI_COMM_WORLD);
```

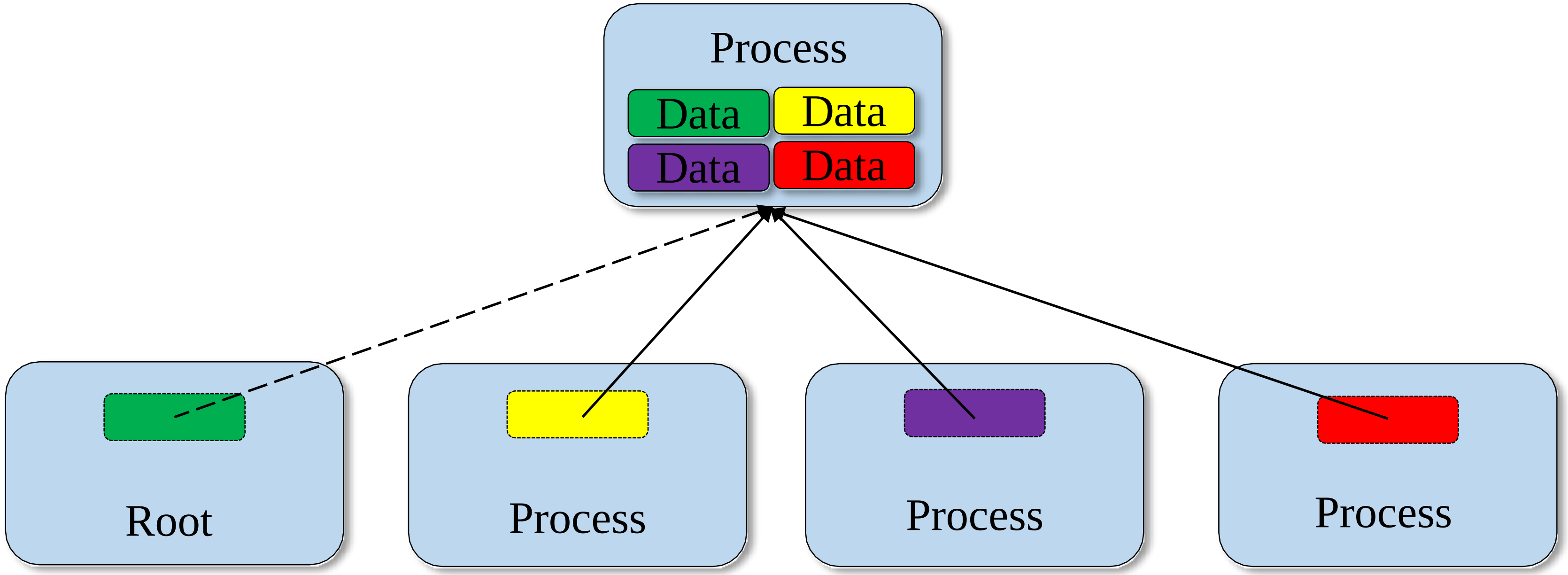
task 1 contains the message to be scattered



# MPI_Gather

```
MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
            int recvcnt, MPI_Datatype recvtype,  int root, MPI_Comm comm)
```

- One process (root) collects data from all the other processes in the same communicator (i.e each process in *comm* (including *root* itself) sends its *sendbuf* to *root*.)
- The *root* process receives the messages in *recvbuf* **in rank order**
- Must be called by all the processes with the same arguments

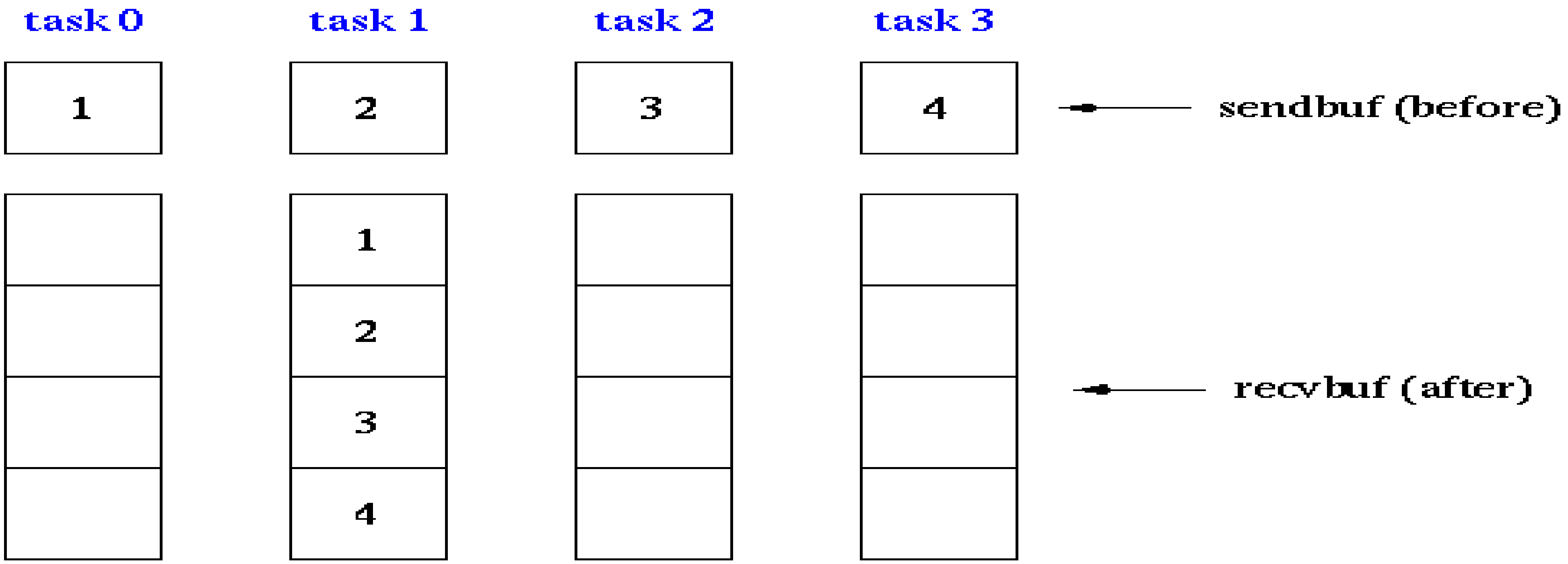



**Inverse to MPI_Scatter**

# MPI_Gather

**Gathers together values from a group of processes**

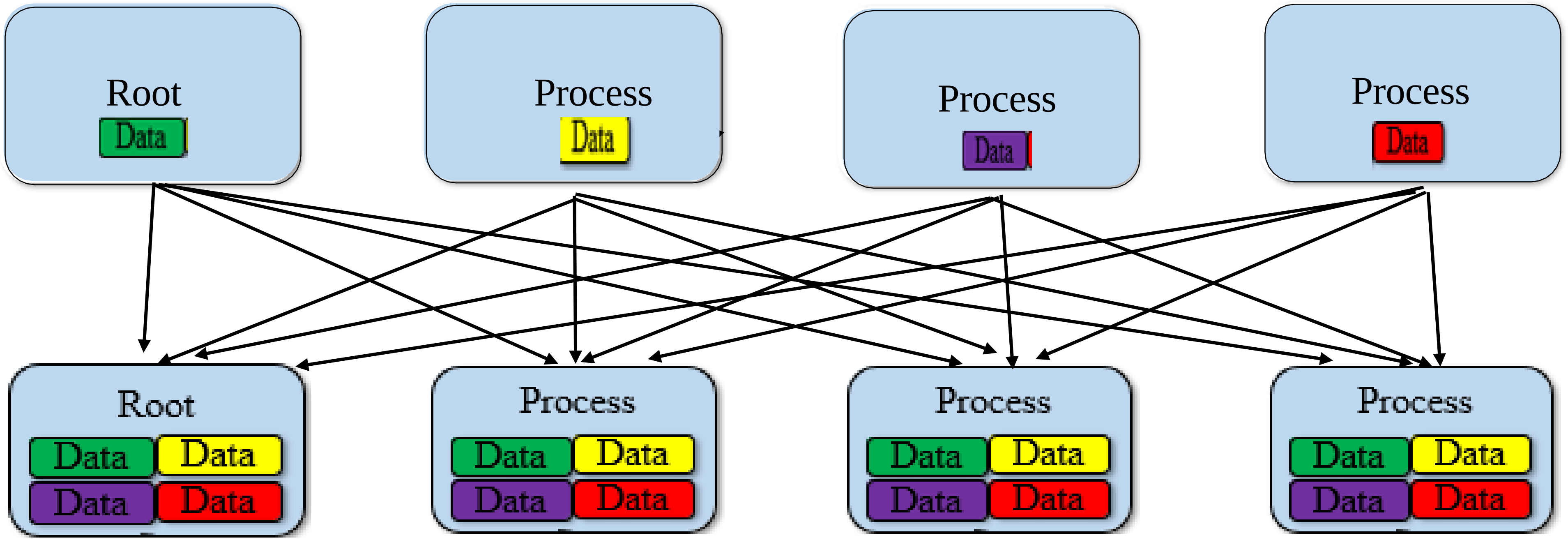
```
sendcnt = 1;
recvcnt = 1;
src  = 1;                  messages will be gathered in task 1
MPI_Gather(sendbuf, sendcnt, MPI_INT,
           recvbuf, recvcnt, MPI_INT,
           src, MPI_COMM_WORLD);
```



# MPI_Allgather

```
MPI_Allgather (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf,
               int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)
```

- All the processes collects data from all the other processes in the same communicator (i.e similar to MPI_Gather except now all processes receive the result.)
- *recvbuf* is **NOT ignored**
- Must be called by all the processes with the same arguments



- MPI_Allgather

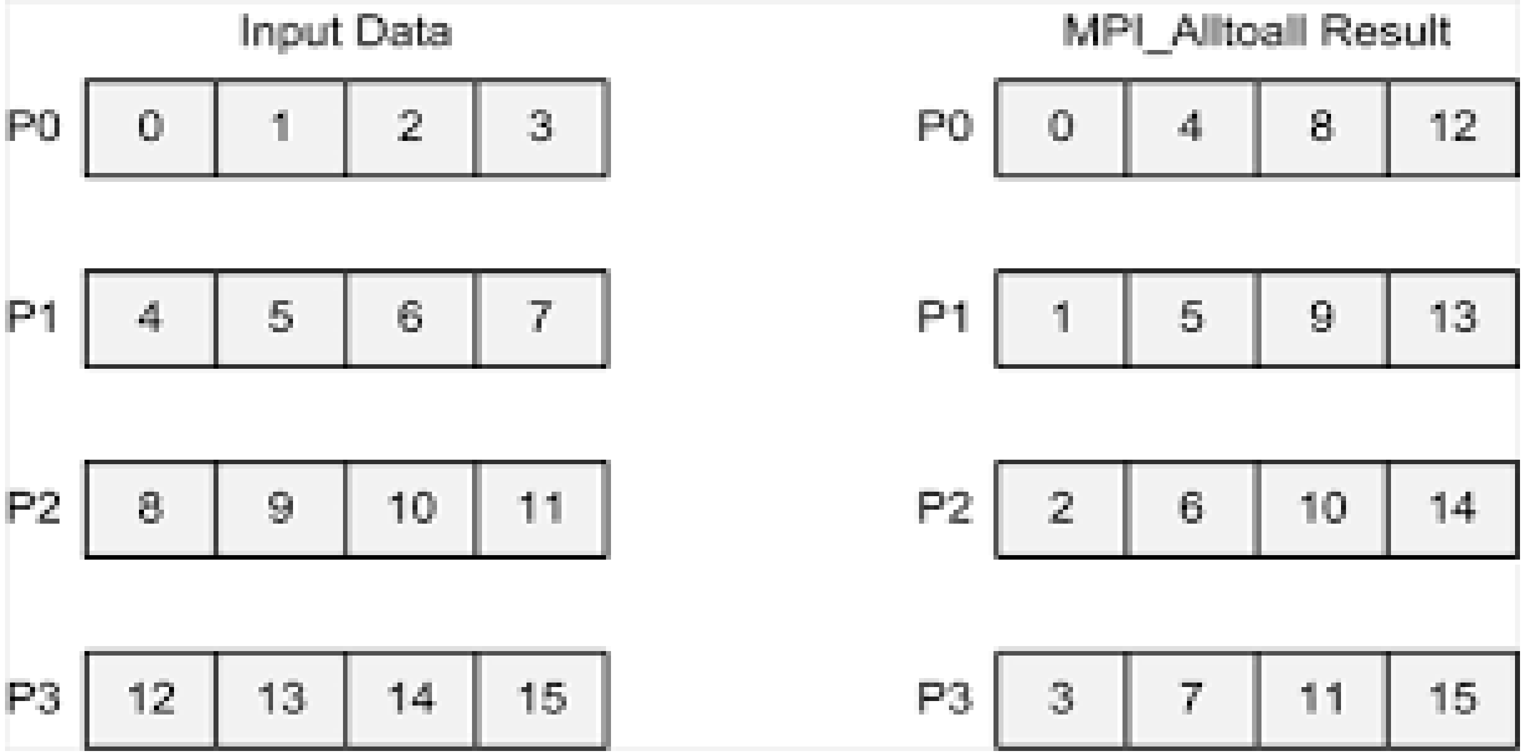
| PO | A | | | |
|---|---|---|---|---|
| **P1** | B | | | |
| **P2** | C | | | |
| **P3** | D | | | |

| PO | A | B | C | D |
|---|---|---|---|---|
| **P1** | A | B | C | D |
| **P2** | A | B | C | D |
| **P3** | A | B | C | D |

# MPI_Alltoall

```
MPI_Alltoall (void *sendbuf, int sendcount, MPI_Datatype sendtype, void
              *recvbuf, int recvcount, MPI_Datatype recvtyp, MPI_Comm comm )
```

- It is a combination of **MPI_Scatter** and **MPI_Gather**
- It is an extension of the [MPI_Allgather]() function
- Each process sends distinct data to each of the receivers. **The *j*th block that is sent from process *i* is received by process *j* and is placed in the *i*th block of the receive buffer**



- MPI_Alltoall

| PO | A0 | B0 | C0 | D0 |
|---|---|---|---|---|
| **P1** | A1 | B1 | C1 | D1 |
| **P2** | A2 | B2 | C2 | D2 |
| **P3** | A3 | B3 | C3 | D3 |

| PO | A0 | A1 | A2 | A3 |
|---|---|---|---|---|
| **P1** | B0 | B1 | B2 | B3 |
| **P2** | C0 | C1 | C2 | C3 |
| **P3** | D0 | D1 | D2 | D3 |

# MPI_Allgather vs MPI_Alltoall

```
rank     send buf                          recv buf
----     --------                          --------
 0       a,b,c          MPI_Allgather      a,b,c,A,B,C,#,@,%
 1       A,B,C        ---------------->    a,b,c,A,B,C,#,@,%
 2       #,@,%                             a,b,c,A,B,C,#,@,%
```

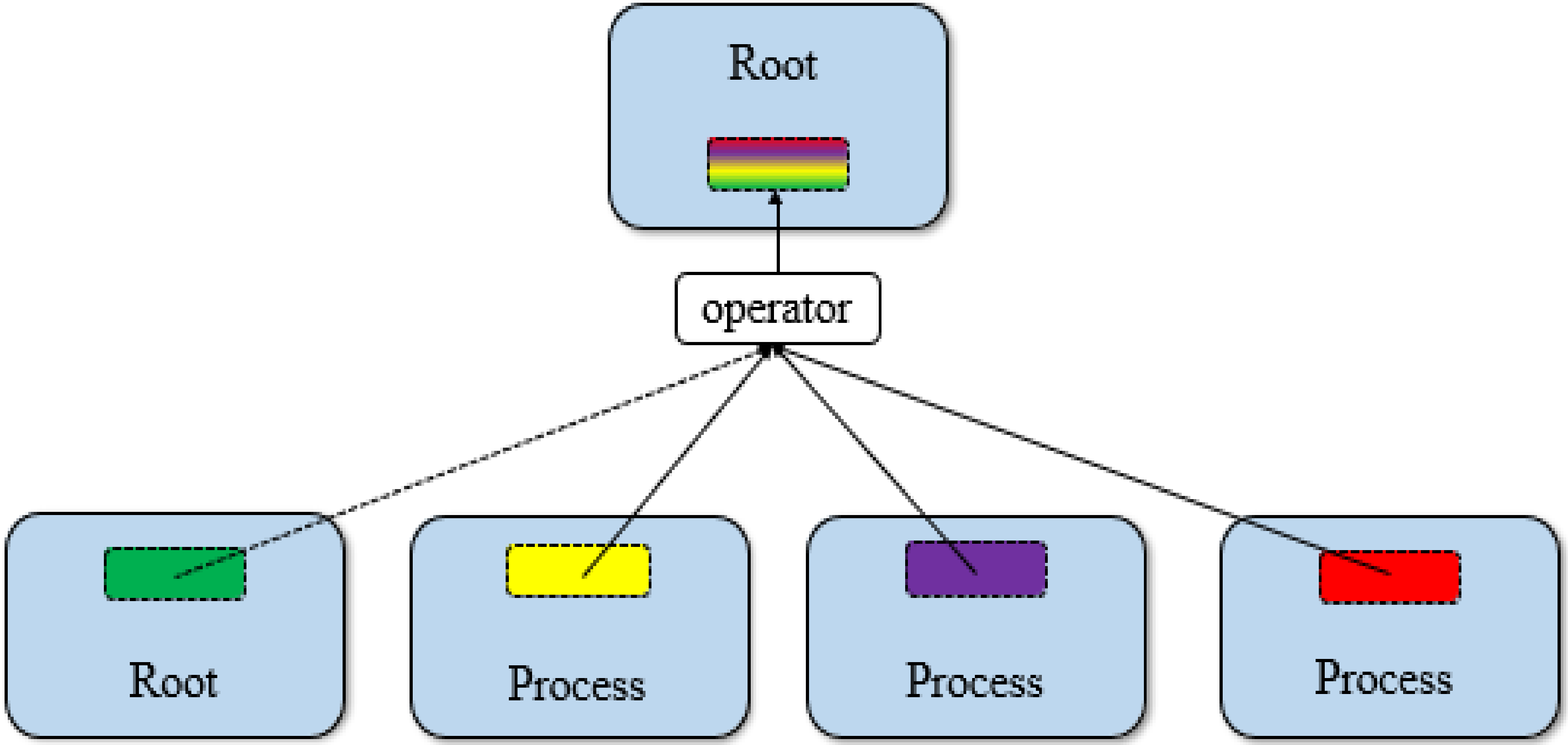
This is just the regular `MPI_Gather`, only in this case all processes receive the data chunks, i.e. the operation is root-less.

```
rank     send buf                          recv buf
----     --------                          --------
 0       a,b,c           MPI_Alltoall      a,A,#
 1       A,B,C        ---------------->    b,B,@
 2       #,@,%                             c,C,%

(a more elaborate case with two elements per process)

rank     send buf                          recv buf
----     --------                          --------
 0       a,b,c,d,e,f     MPI_Alltoall      a,b,A,B,#,@
 1       A,B,C,D,E,F  ---------------->    c,d,C,D,%,$
 2       #,@,%,$,&,*                       e,f,E,F,&,*
```

# MPI_Scan

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

- It returns the *partial operation results* on **each processor**



# *MPI_Reduce* vs *MPI_Scan*

- A **reduction** means all processors get the same value while **scan** returns the partial operation results on each processor

- **For example:**

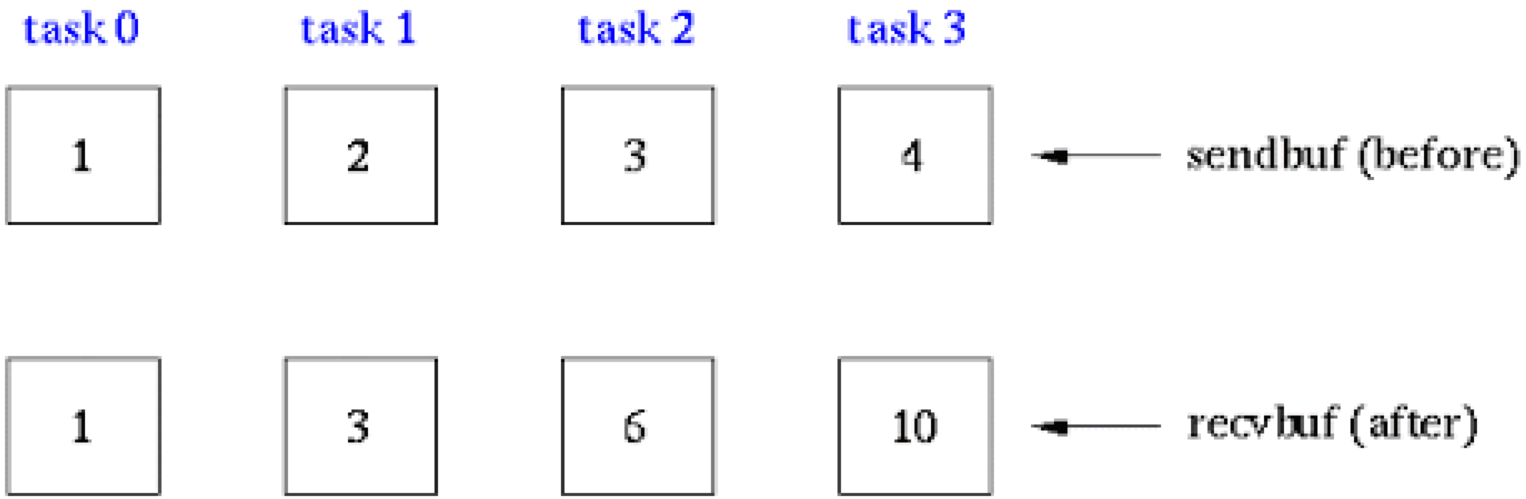
  - if you had **10** processors and you were taking the sum of their rank, **MPI_Reduce** would give you the scalar **45 (0+1+2+3+4+5+6+7+8+9)** on the root process,

  - while **MPI_scan** would give you the scalar of the reduction *up to the rank of the processor on each processor*. So processor **0** would get **0**, processor **1** would get **1**, processor **2** would get **3**, and so on. Processor **9** would get **45**

# MPI_Scan

Computes the scan (partial reductions) of data on a collection of processes

```
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
                  MPI_COMM_WORLD);
```



Row scatter

| | b[0] | b[1] | b[2] |
|---|---|---|---|
| **P0** | 1 | 2 | 3 |
| **P1** | 4 | 5 | 6 |
| **P2** | 7 | 8 | 9 |

original matrix(a)

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

Column sum (method 1)

| | b[0] | b[1] | b[2] | Column Sum |
|---|---|---|---|---|
| **P0** | 1 | 4 | 7 | 12 |
| **P1** | 2 | 5 | 8 | 15 |
| **P2** | 3 | 6 | 9 | 18 |

Original Matrix

| | | | |
|---|---|---|---|
| a[0][0] | 1 | 2 | 3 |
| a[1][0] | 4 | 5 | 6 |
| a[2][0] | 7 | 8 | 9 |

Column sum (method 2)

| | b[0] | b[1] | b[2] |
|---|---|---|---|
| P0 | 1 | 2 | 3 |
| P1 | 4 | 5 | 6 |
| P2 | 7 | 8 | 9 |

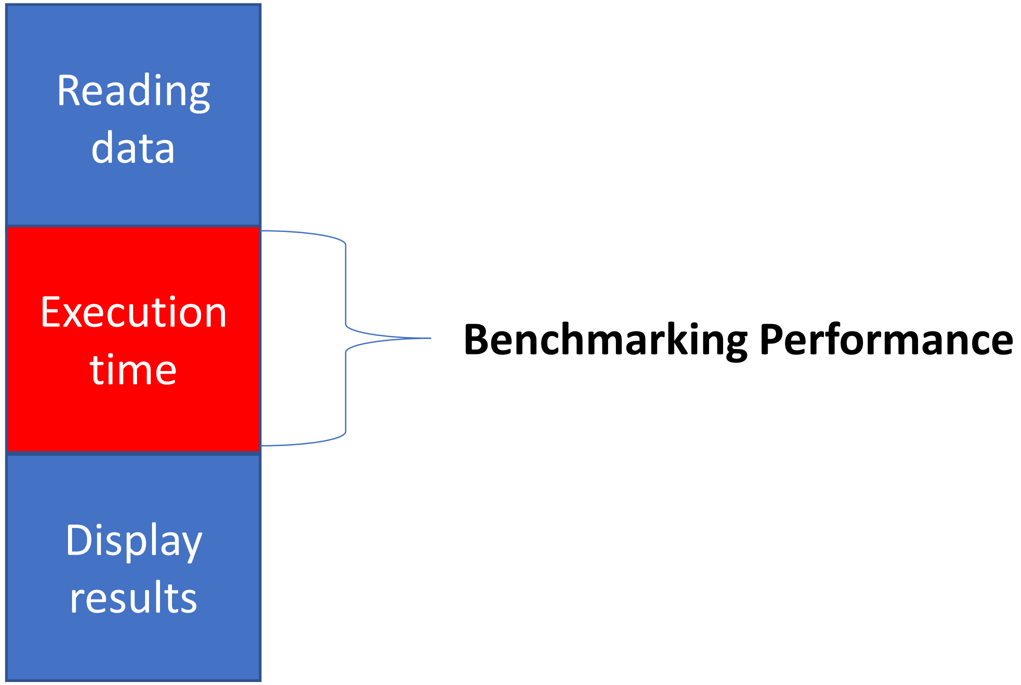
Rank=0 csum[0] csum[1] csum[2]

Original matrix

| | | | |
|---|---|---|---|
| a[0][0] | 1 | 2 | 3 |
| a[1][0] | 4 | 5 | 6 |
| a[2][0] | 7 | 8 | 9 |

# Benchmarking Parallel Performance

- Benchmarking parallel program performance measure *how well* parallel programs perform against their *sequential counterparts* in the "middle area" between reading the dataset and writing the results




- Typically, we are going to **ignore** the time spent *initiating MPI processes*, *establishing communications sockets* between them, and *performing I/O* on sequential devices

# Benchmarking Parallel Performance

- MPI provides a function called **MPI_Wtime** that returns the *number of seconds that have elapsed* since some point of time in the past

- Function **MPI_Wtick** returns the precision of the result returned by **MPI_Wtime**

**double MPI_Wtime (void)**

**double MPI_Wtick (void)**

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

```
double elapsed_time;
………….
MPI_Init(&argc, &argv);
elapsed_time = - MPl_Wtime();

 …….parallel execution code…..

elapsed_time += MPI_Wtime();
```

# *Setting barrier for process synchronization*

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

- MPI processes executing on different processors may begin **executing seconds apart**. *This can throw off timings significantly*

- We address this problem by **introducing a barrier synchronization** **before the first call to MPI_Wtime**.

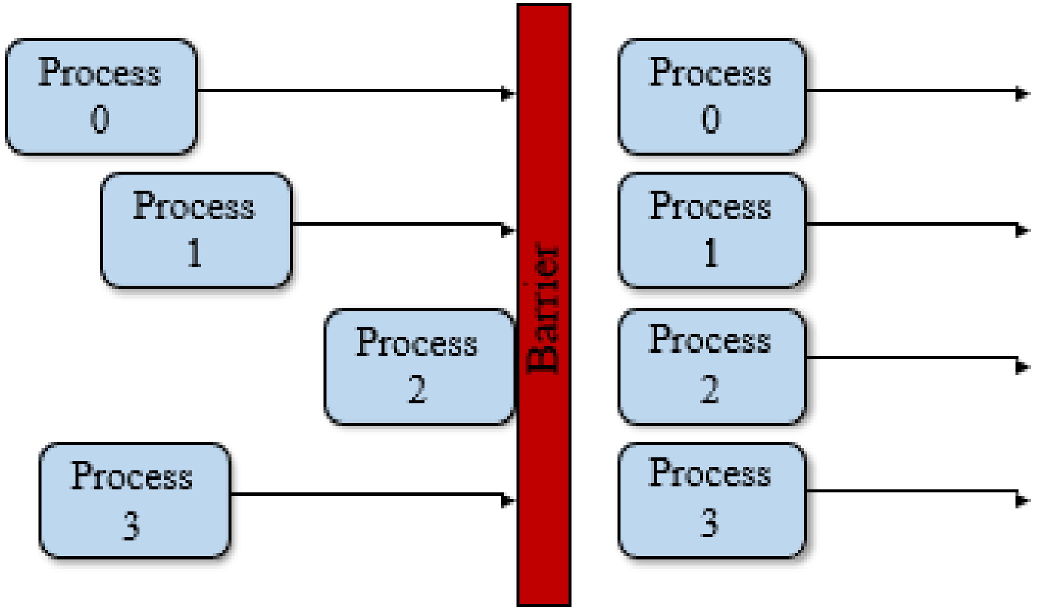
- No process can proceed beyond a barrier until all processes have reached it

- Hence a barrier ensures that all processes are going into the measured section of code at more or less the same time

# Setting barrier for process synchronization

```
MPI_Barrier(MPI_COMM_WORLD)
```

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




```
double elapsed_time;
………….
MPI_Init(&argc, &argv);
MPI_Barrier (MPI_COMM_WORLD)
elapsed_time = - MPl_Wtime();

 …….parallel execution code…..

elapsed_time += MPI_Wtime();
```

# MPI Error Handling Functions

- When an error is occurred while executing a MPI program typically, the program aborts
- MPI calls a *default error handler* **MPI_ERRORS_ARE_FATAL** every time an MPI error is detected within the communicator
- **MPI_ERRORS_ARE_FATAL** *abort the whole parallel program* as soon as any MPI error is detected
- There is another predefined error handler **MPI_ERRORS_RETURN** which is used to return the generated error for custom handling
- The default error handler **MPI_ERRORS_ARE_FATAL** can be replaced with **MPI_ERRORS_RETURN** by calling function **MPI_Errhandler_set ()**

```
MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN)
```

# MPI Error Handling Functions

- Once we've called **MPI_Errhandler_set ( )** in our MPI code, the program will no longer abort on having detected an MPI error, instead the error will be returned and we will have to handle it

- MPI standard defines ***error classes***. *Every error code, must belong to some error class*, and the error class for a given error code can be obtained by calling function **MPI_Error_class ()**

```
MPI_Error_class(int errorcode, int *errorclass)
```

- Error code can be converted to comprehensible error messages by calling function **MPI_Error_string ()**

```
MPI_Error_string(int errorcode, char *string, int *resultlen)
```

# MPI Error Handling Functions

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

void ErrorHadler(int error_code);

int main(int argc,char *argv[]){
      int  C=3;
      int  numtasks, rank, len, error_code;
      MPI_Init(&argc,&argv);
      MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
      error_code = MPI_Comm_size(C,&numtasks);
      ErrorHadler(error_code);
      printf ("Number of tasks= %d My rank= %d \n", numtasks,rank);
      MPI_Finalize();
   }
```

# MPI Error Handling Functions

```
void ErrorHadler(int error_code){

  if (error_code != MPI_SUCCESS){
     char error_string[BUFSIZ];
     int length_of_error_string, error_class;
     MPI_Error_class(error_code, &error_class);
     MPI_Error_string(error_code, error_string, &length_of_error_string);
     printf( "%d %s\n",  error_code, error_string);

     MPI_Error_string(error_class, error_string, &length_of_error_string);
     printf("%d %s\n",  error_class, error_string);
   }
}
```

# Useful MPI Routines

| Routine | Purpose/Function |
|---|---|
| MPI_Init | Initialize MPI |
| MPI_Finalize | Clean up MPI |
| MPI_Comm_size | Get size of MPI communicator |
| MPI_Comm_Rank | Get rank of MPI Communicator |
| MPI_Reduce | Min, Max, Sum, etc |
| MPI_Bcast | Send message to everyone |
| MPI_Allreduce | Reduce, but store result everywhere |
| MPI_Barrier | Synchronize all tasks by blocking |
| MPI_Send | Send a message (blocking) |
| MPI_Recv | Receive a message (blocking) |
| MPI_Isend | Send a message (non-blocking) |
| MPI_Irecv | Receive a message (non-blocking) |
| MPI_Wait | Blocks until message is completed |

MPI Documentation

MPI Reference