Introduction to Parallel Programming with MPI

By
Dr. Sandhya Parasnath Dubey and Dr. Manisha
Assistant Professor
Department of Data Science and Computer Applications - MIT,
Manipal Academy of Higher Education, Manipal-576104, Karnataka, India.
Email: sandhya.dubey@manipal.edu; manisha.mit@manipal.edu

Distributed memory computers

- Distributed memory computers cannot communicate through a shared memory.
- Therefore, messages are used to coordinate parallel tasks that eventually run on geographically distributed but interconnected processors.
- Processes as well as their management and communication are well defined by a platform-independent message passing interface (MPI) specification.
- MPI is introduced from the practical point of view, with a set of basic operations that enable implementation of parallel programs.

Distributed memory computers

- Two main differences between the shared memory and distributed memory computer architectures
  - **Price of communication:** the time needed to exchange a certain amount of data between two or more processors is in favor of shared memory computers, as these can usually communicate much faster than the distributed memory computers.
  - **Number of processors:** Distributed memory computers can efficiently support a large number of processors. Useful where a high number of processors is required to reduce execution time.

Message Passing Interface (MPI)

- Enables **system independent** parallel programming. i.e., MPI is a standardized API that provide consistent set of functions for parallel programming. MPI code can be executed on different hardware platforms without modification.
- Parallel programming isn't just about dividing tasks among multiple processors. The processes must also cooperate, which means they need to exchange data.
  - The total execution time is consequently a sum of computation and communication time.
- The speed of communication becomes a crucial performance factor, especially when the number of processors is high.
- Algorithms with only local communication between neighboring processors are faster and more scalable than the algorithms with the global communication among all processors.

Message Passing Interface (MPI)

- Therefore, the programmer's view of a problem that will be parallelized has to incorporate a wide number of aspects, e.g., data independency, communication type and frequency, balancing the load among processors, balancing between communication and computation, overlapping communication and computation, synchronous or asynchronous program flow, stopping criteria, and others.

MPI includes several key features:

- Process Creation and Management: MPI provides functions to create and manage processes across different systems seamlessly.
- Language Bindings: It supports language bindings for popular programming languages like C and Fortran, making it versatile.

Message Passing Interface (MPI)

- Communication: MPI includes functions for both point-to-point and collective communications
  - **Point-to-point:** It involves direct communication between two processes. One process sends data, and another receives it. This type of communication is useful for scenarios where only specific pairs of processes need to exchange information.
  - **Collective:** This involves communication among a group of processes. All processes in a group participate in the communication, either by sending, receiving, or both.
- Groups and Communicators: These concepts help define how processes are grouped and how they communicate, providing a structured way to manage process interactions.
  - A group is an ordered set of processes. Every process within a group is identified by a unique rank. Groups allow you to manage collections of processes and determine how they interact within a program.
  - A communicator in MPI is an object that defines a group of processes that can communicate with each other. It provides the environment in which these processes operate, enabling them to send and receive messages.

Message Passing Interface (MPI)

- The MPI is not a language but rather a specification or a standard that defines how processes in a parallel computing environment can communicate with each other.
- All MPI "**operations**" are expressed as functions, subroutines, or methods that can be called from within a program written in a standard programming language like C, C++, or Fortran.
- The MPI standard defines the syntax and semantics of library operations that support the message passing model, independently of program language or compiler specification.
- An MPI program consists of autonomous processes that are able to execute their own code in the sense of multiple instruction multiple data (MIMD) paradigm.
- An MPI "**process**" can be interpreted in this sense as a program counter that addresses their program instructions in the system memory, which implies that the program codes executed by each process have not to be the same.
  - For example, one process might be reading data, another might be processing it, and yet another might be aggregating results—all within the same MPI program.

Message Passing Interface: **Prerequisites**

- To run an MPI (Message Passing Interface) program written in C on a Unix-based system (such as Linux), you typically need to follow these steps:
- **Install MPI Implementation**: Ensure that an MPI implementation (e.g., Open MPI, MPICH) is installed on your Unix system. If it's not already installed, you can install it using your system's package manager. For example, on Debian-based systems (like Ubuntu), you can use apt-get:

➢ sudo apt-get update

➢ sudo apt-get install mpich # for MPICH

➢ sudo apt-get install openmpi-bin # for Open MPI

➢ sudo apt-get install libopenmpi-dev # On Debian/Ubuntu

Example MPI Program

```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 from rank %d of %d\n", rank, size);

    MPI_Finalize();

    return 0;
}
```

Message Passing Interface:**Compiling and Running MPI Programs**

1. **Compile MPI Program**: Use the MPI compiler wrapper (mpicc for C programs) to compile your MPI program:

- mpicc -o mpi_program mpi_program.c

➢ Here, mpi_program.c is your MPI C program file, and mpi_program is the executable output file.

2. **Run MPI Program**: Use the mpirun or mpiexec command to run your MPI program with the desired number of processes (or MPI ranks):

- mpirun -np 4 ./mpi_program

Replace 4 with the number of MPI processes you want to run. -np specifies the number of MPI ranks (processes) to launch.

Message Passing Interface (MPI)

- #include <stdio.h> is needed because of printf, which is used later in the program, and
- #include "mpi.h" provides basic MPI definition of named constants, types, and function prototypes, and must be included in any MPI program.

Example MPI Program

- **MPI_Init**: Initializes the MPI execution environment.
- **MPI_Comm_rank**: Retrieves the rank (ID) of the calling process within the communicator (MPI_COMM_WORLD).
- **MPI_Comm_size**: Retrieves the total number of processes in the communicator (MPI_COMM_WORLD).
- **MPI_Finalize**: Finalizes the MPI environment before program exit.

Running the Example:

- Assuming mpi_program.c is your MPI program file, compile it using:
  - mpicc -o mpi_program mpi_program.c
- Then, run it with:
  - mpirun -np 4 ./mpi_program
- This will run the MPI program mpi_program with 4 MPI processes, each printing its rank and the total number of processes (size).

```
sandhya@telnet:~/MPI$ mpicc -o mpi_program mpi_program.c
sandhya@telnet:~/MPI$ mpirun -np 4 ./mpi_program
Hello world from rank 1 of 4
Hello world from rank 2 of 4
Hello world from rank 3 of 4
Hello world from rank 0 of 4
sandhya@telnet:~/MPI$
```

MPI Operation Syntax

- The MPI standard is independent of specific programming languages.
- Capitalized MPI operation names (MPI_SEND, MPI_RECV, and MPI_BCAST) will be used in the definition of MPI operations.
- MPI operations take various arguments. They are categorized into three types based on their role in the operation.
  - IN—IN arguments are used as inputs to an MPI operation. These values are passed into the operation, but they are not modified by it.

    IN arguments are in normal text, e.g., buf, sendbuf, MPI_COMM_WORLD, etc.
  - OUT—OUT arguments are used by an MPI operation to store the results. These are the values that the operation will update or modify, but not used as input value;

    e.g., rank, recbuf, etc.
  - INOUT—INOUT arguments are used both as inputs and outputs. The function may use the initial value of the argument and then modify it during the operation.

    e.g., inbuf,request, etc.

MPI Data Types

Table 4.1 Some MPI data types corresponding to C and Fortran data types

MPI data type	C data type	MPI data type	Fortran data type
MPI_INT	int	MPI_INTEGER	INTEGER
MPI_SHORT	short int	MPI_REAL	REAL
MPI_LONG	long int	MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_FLOAT	float	MPI_COMPLEX	COMPLEX
MPI_DOUBLE	double	MPI_LOGICAL	LOGICAL
MPI_CHAR	char	MPI_CHARACTER	CHARACTER
MPI_BYTE	/	MPI_BYTE	/
MPI_PACKED	/	MPI_PACKED	/

The type MPI_PACKED is maintained by MPI_PACK or MPI_UNPACK operations, which enable to pack different types of data into a contiguous send buffer and to unpack it from a contiguous receive buffer.

MPI Environment Management Routines:

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

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

MPI_Finalize: Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program. No other MPI routines may be called after it.

```
MPI_Finalize ();
```

MPI Environment Management Routines:

MPI_Comm_rank: Returns the rank of the calling MPI process within the specified communicator. Each process will be assigned a unique integer rank between 0 and size - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a process ID.

```
MPI_Comm_rank Comm,&rank);
```

MPI_Comm_size: Returns the total number of MPI processes to the variable size in the specified communicator, such as MPI_COMM_WORLD.

```
MPI_Comm_size(Comm,&size);
```

MPI Environment Management Routines:

Solved Example:

Write a program in MPI to print total number of process and rank of each process.

```
#include "mpi.h"
#include <stdio.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("My rank is %d in total %d processes", rank, size);
        MPI_Finalize();
        return 0;
}
```

MPI Environment Management Routines:

Lab Exercises:

1. Write a simple MPI program to find out pow (x, rank) for all the processes where 'x' is the integer constant and 'rank' is the rank of the process.
2. Write a program in MPI where even ranked process prints "Hello" and odd ranked process prints "World".
3. Write a program in MPI to simulate simple calculator. Perform each operation using different process in parallel.
4. Write a program in MPI to toggle the character of a given string indexed by the rank of the process. Hint: Suppose the string is HeLLO and there are 5 processes, then process 0 toggle 'H' to 'h', process 1 toggle 'e' to 'E' and so on.

Point to Point Communication in MPI

Objectives:

1. Understand the different APIs used for point to point communication in MPI
2. Learn the different modes available in case of blocking send operation

- The process-to-process communication has to implement two essential tasks:
  - data movement and
  - Synchronization of processes;

  Therefore, it requires cooperation of sender and receiver processes.

Point to Point Communication in MPI

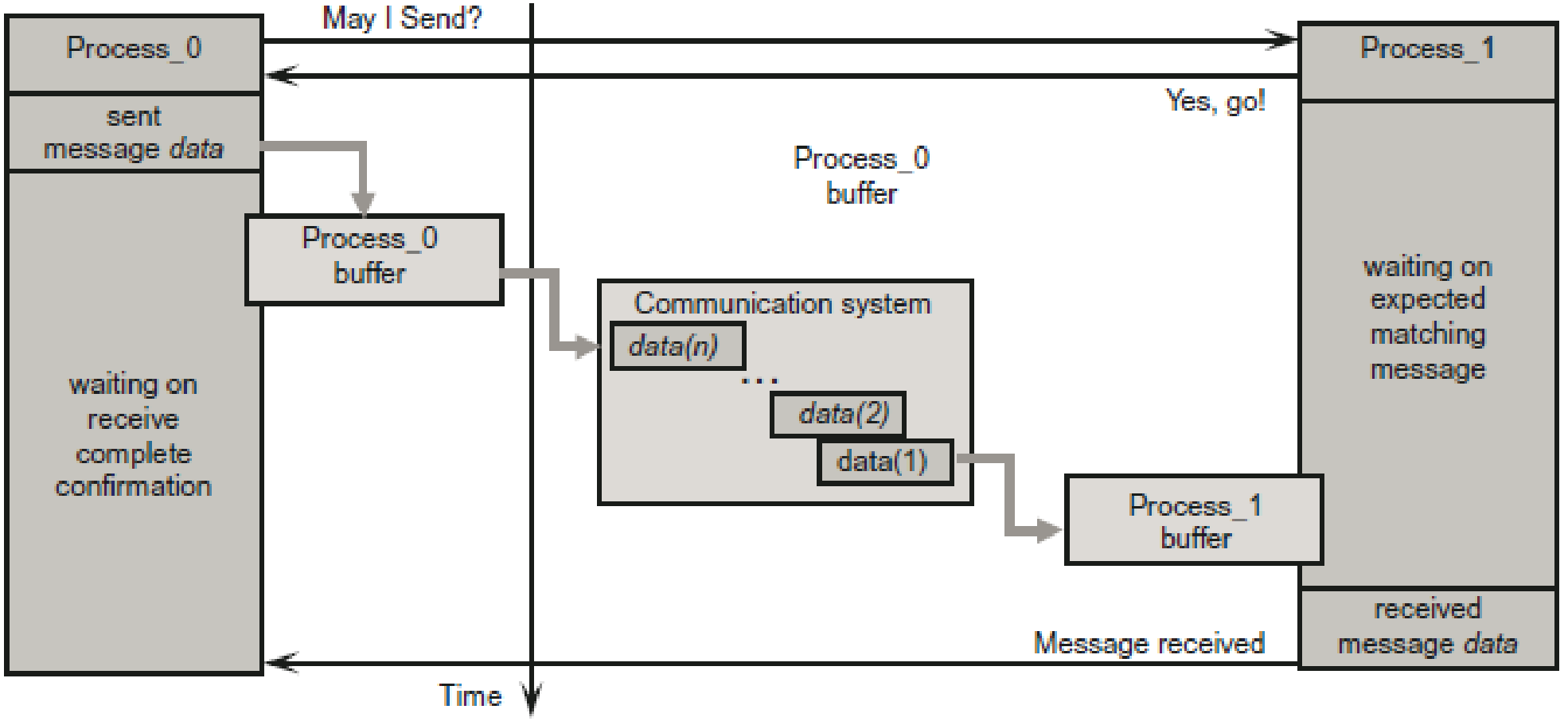


Fig. 4.1 Communication between two processes awakes both of them while transferring data from sender `Process_0` to receiver `Process_1`, possibly with a set of shorter sub-messages

Point to Point communication in MPI

- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.
- MPI provides both blocking and non-blocking send and receive operations.

- A blocking send is one where the function call does not return until the send operation is completed.
- Completion means that the message has been successfully copied out of the user's send buffer. This could either mean:
  - The message has been sent to the destination process.
  - The message has been buffered internally by the MPI system (if the message size is small and there's buffer space available).
- The blocking send ensures that the buffer used for sending the data can be safely reused or modified after the send call completes.
- A non-blocking send initiates the send operation but returns immediately, allowing the sender to proceed with other tasks, even while the message is still being sent.

Point to Point Communication in MPI

- A critical issue with blocking communication is the possibility of **deadlocks**.
- If both Process_0 and Process_1 initiate their send calls simultaneously, they will be waiting for each other to receive, leading to a deadlock where neither process can proceed.
- For example, if Process_0 and Process_1 initiate their send calls in the same time, they will be blocked forever by waiting matching receive calls.

Point to Point Communication in MPI

MPI_SEND (buf, count, datatype, dest, tag, comm)

- The send buffer is specified by the following arguments
  - buf - pointer to the send buffer,
  - count - number of data items,
  - datatype - type of data items.
- The receiver process is addressed by an envelope that consists of arguments
  - dest, which is the rank of receiver process within all processes in the communicator comm, and of a message tag.
  - **tag** provide a mechanism for distinguishing between different messages for the same receiver process identified by destination rank

Point to Point Communication in MPI

MPI_RECV (buf, count, datatype, source, tag, comm, status)

- This operation waits until the communication system delivers a message with matching datatype, source, tag, and comm.
- The entire set of arguments: count, datatype, source, tag and comm, must match between the sender process and the receiver process to initiate actual message passing.
- When a message, posted by a sender process, has been collected by a receiver process, the message is said to be completed, and the program flows of the receiver and the sender processes may continue.

Point to Point Communication in MPI

Sending message in MPI

- **Blocked Send** sends a message to another processor and waits until the receiver has received it before continuing the process. Also called as **Synchronous send**.
- **Send** sends a message and continues without waiting. Also called as **Asynchronous send.**

There are multiple communication modes used in blocking send operation:

- **Standard mode**
- **Synchronous mode**
- **Buffered mode**

both the mpi send told above and the mpi send here are same
The MPI_SEND and MPI_Ssend is both the blocking send only
the asynchronous send is MPI_Isend operation

the diff between MPI_SEND and MPI_Ssend is
MPI_Ssend is asynchronous the MPI_SEND is not meaning mpi_send
wont guarantee that the message has received it means moved to buffer or is sent

MPI_SEND
The function doesn't wait for the receiver to acknowledge that the message has started being received. The actual transmission could still be in progress after MPI_Send returns.

MPI_Ssend
The function only returns when the corresponding MPI_Recv has started, ensuring that the receiver is ready and the communication is fully synchronized.

Point to Point Communication in MPI

Standard mode

This mode blocks until the message is buffered.

```
MPI_Send(&Msg, Count, Datatype, Destination, Tag,  Comm);
```

- First 3 parameters together constitute message buffer. The **Msg** could be any address in sender's address space. The **Count** indicates the number of data elements of a particular type to be sent. The **Datatype** specifies the message type. Some Data types available in MPI are: MPI_INT, MPI_FLOAT, MPI_CHAR, MPI_DOUBLE, MPI_LONG
- Next 3 parameters specify message envelope. The **Destination** specifies the rank of the process to which the message is to be sent.
- **Tag:** The **tag** is an integer used by the programmer to label different types of messages and to restrict message reception.

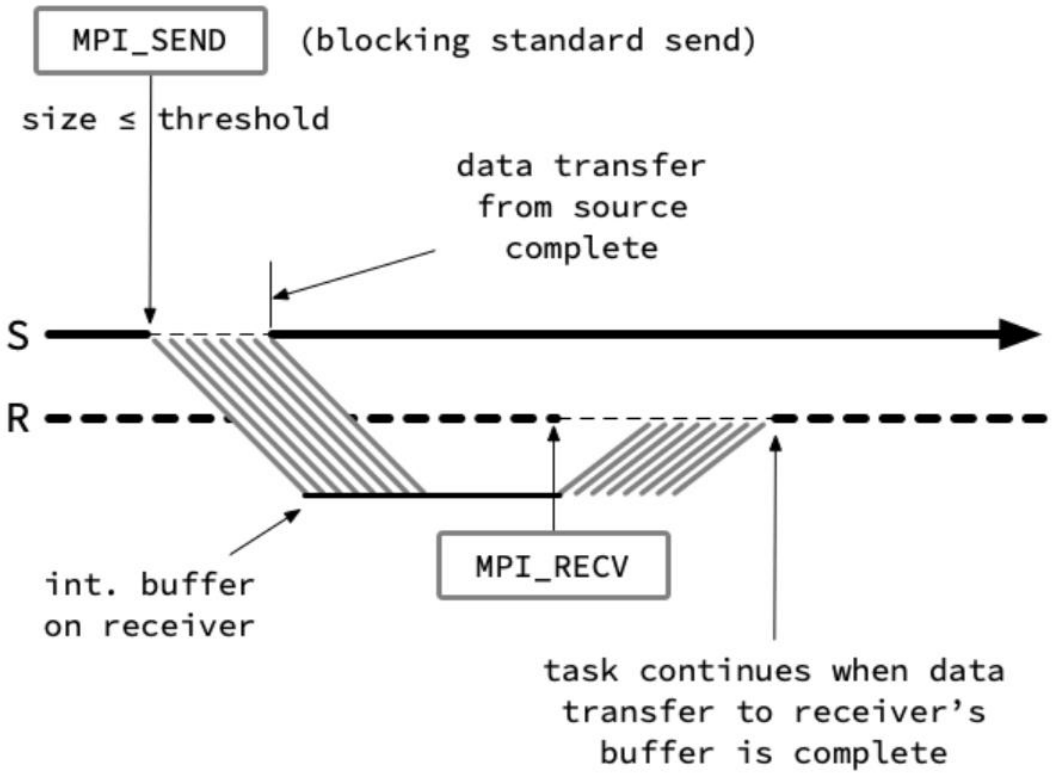
Point to Point Communication in MPI

- **Communicator:** Major problem with tags is that they are specified by users who can make mistakes. **Context** are allocated at run time by the system in response to user request and are used for matching messages. The notions of **context and group** are combined in a single object called a communicator (**Comm**).
- The default process group is **MPI_COMM_WORLD**.

- In MPI's Standard Mode (MPI_Send), the behavior of the send operation can vary depending on whether the size of the data being sent is less than or greater than a certain threshold.
- This threshold is implementation-dependent and determines whether the message is sent using buffering or synchronization .

MPI Standard Send will have two implementations based on size of the message being > the buffer size or less than the buffer size

Point to Point Communication in MPI

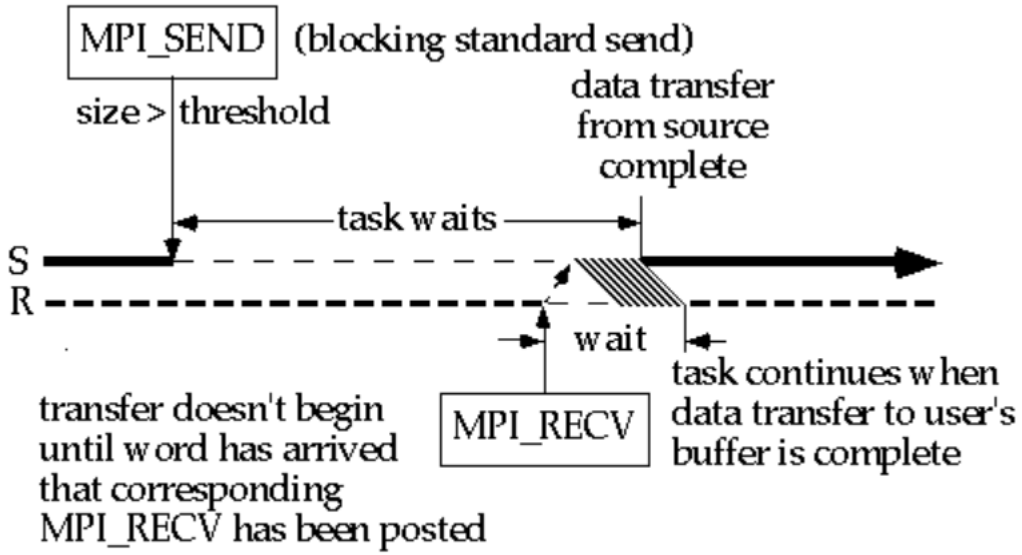


- The heavy horizontal line marked S represents the execution time of the sending task (on one node)
- The heavy dashed line marked R represents the execution time of the receiving task (on a second node).
- Breaks in these lines represent interruptions due to the message-passing event.

Message size less than threshold:

- The blocking standard send MPI_Send copies the message over the network into a system buffer on the receiving node.
- The standard send then returns, and the sending task can continue computation.
- The system buffer is attached when the program is started- the user does not need to manage it in any way.
- There is one system buffer per task that will hold multiple messages.
- The message will be copied from the system buffer to the receiving task when the receive call is executed.
- This protocol is like buffered mode, but the buffering is all done on the receiving side rather than the sending side.

Point to Point Communication in MPI



Message size greater than threshold: **If messages are large, the penalty of putting extra copies in a buffer may be excessive and may even cause the program to crash. Buffering may turn out to be beneficial for small messages.**

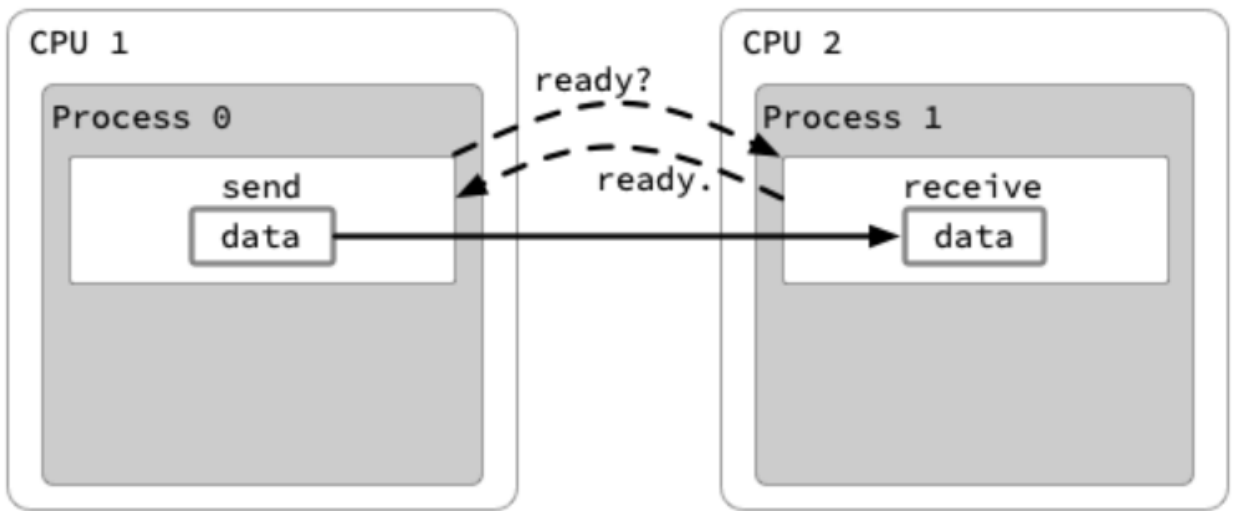
- When the size of the data being sent is larger than the threshold, MPI typically does not use internal buffering.
- Instead, the MPI_Send operation may block until the receiver is ready to receive the data.
- This blocking behavior occurs because large messages cannot be buffered efficiently.
- Therefore, MPI waits for the receiver to post a matching MPI_Recv before the data transfer can proceed.

Point to Point Communication in MPI

Synchronous mode

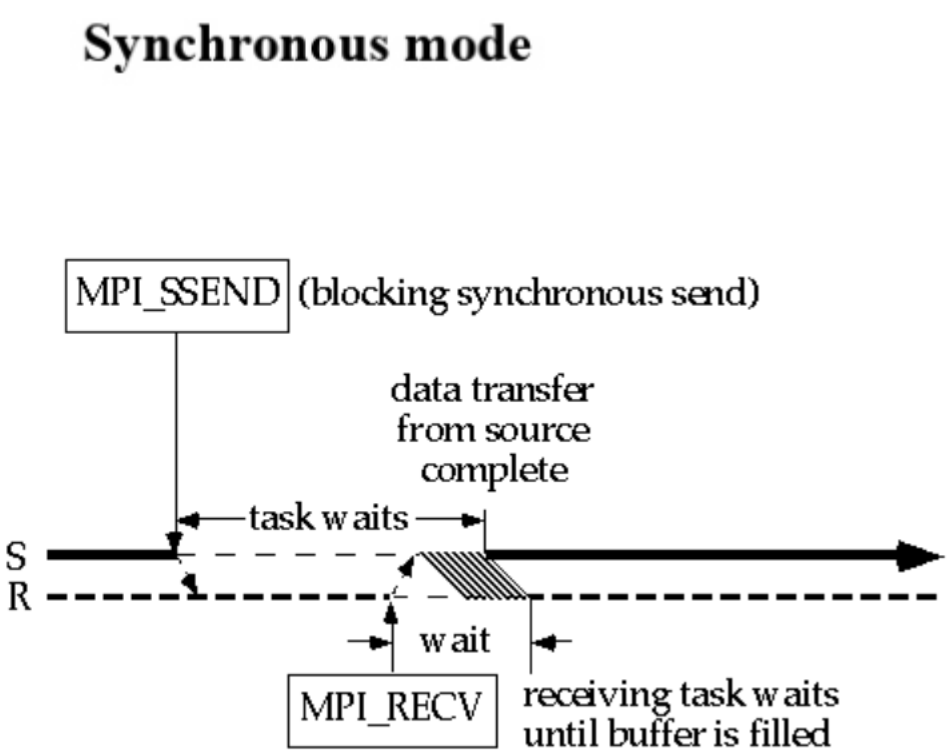
This mode requires a send to block until the corresponding receive has occurred.

```
MPI_Ssend(&Msg, Count, Datatype, Destination, Tag,  Comm);
```



In the synchronous send mode, both the sender and receiver signal their readiness to communicate before passing the message

Point to Point Communication in MPI



- When the blocking synchronous send **MPI_Ssend** is executed, the sending task sends the receiving task a "ready to send" message. When the receiver executes the receive call, it sends a "ready to receive" message. The data are then transferred.
- There are two sources of overhead in message-passing. **System overhead** is incurred from copying the message data from the sender's message buffer onto the network, and from copying the message data from the network into the receiver's message buffer.
- **Synchronization overhead** is the time spent waiting for an event to occur on another task. In the figure, the sender must wait for the receive to be executed and for the handshake to arrive before the message can be transferred.
- The receiver also incurs some synchronization overhead in waiting for the handshake to complete. Synchronization overhead can be significant in synchronous mode.

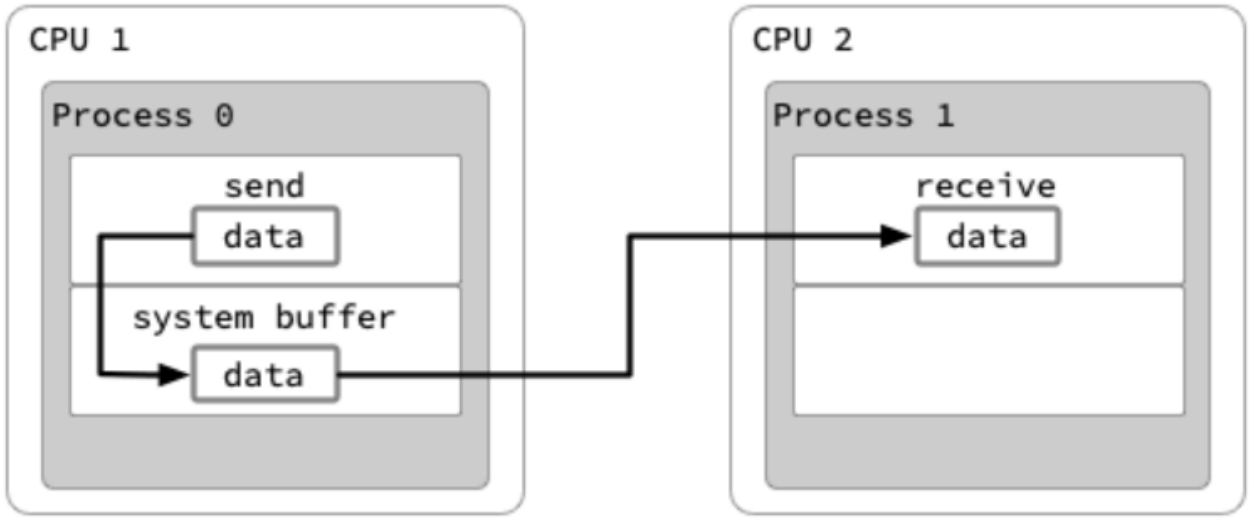
Point to Point Communication in MPI

Buffered mode

```
MPI_Bsend(&Msg, Count, Datatype, Destination, Tag,  Comm);
```

In this mode a send assumes availability of a certain amount of buffer space, which must be previously specified by the user program through a routine call that allocates a user buffer.

In Buffered mode, the user must provide a buffer space explicitly using MPI_Buffer_attach before calling MPI_Bsend



Buffered Send

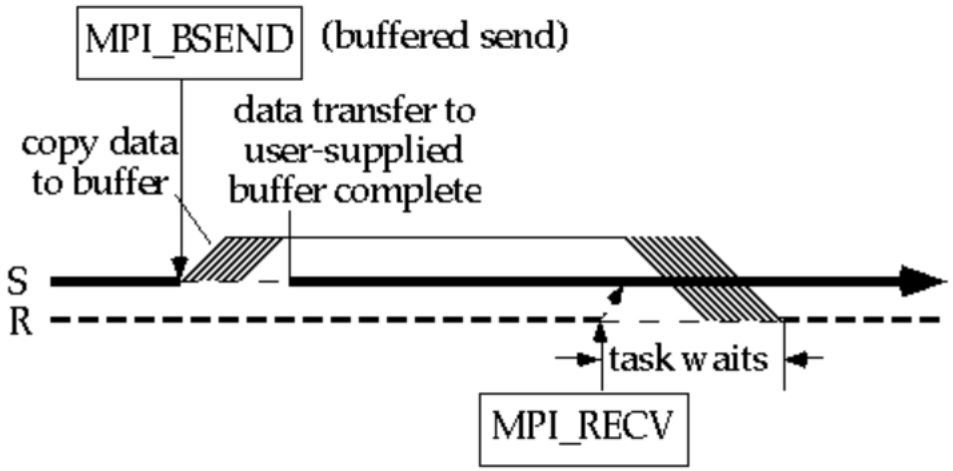
```
MPI-Buffer_attach(buffer, size);
```

This buffer can be released by

```
MPI_Buffer_detach(*buffer, *size);
```

Point to Point Communication in MPI

Buffered mode



- The blocking buffered send **MPI_BSEND** copies the data from the message buffer to a user-supplied buffer, and then returns.
- The sending task can then proceed with calculations that modify the original message buffer, knowing that these modifications will not be reflected in the data actually sent.
- The data will be copied from the user-supplied buffer over the network once the "ready to receive" notification has arrived.
- Buffered mode incurs extra system overhead, because of the additional copy from the message buffer to the user-supplied buffer.
- Synchronization overhead is eliminated on the sending task -- the timing of the receive is now irrelevant to the sender.
- Synchronization overhead can still be incurred by the receiving task. Whenever the receive is executed before the send, it must wait for the message to arrive before it can return.

Point to Point Communication in MPI

Feature/Mode	Standard Mode (`MPI_Send`)	Synchronous Mode (`MPI_Ssend`)	Buffered Mode (`MPI_Bsend`)
Completion	May complete before the message is received	Completes only when the receiver is ready	Completes as soon as the message is buffered
Synchronization	No guarantee of synchronization	Strict synchronization	No synchronization, but requires user-provided buffer
Buffering	May or may not be buffered internally	Not buffered; waits for receiver	Explicitly buffered using user-provided space
Use Case	General-purpose, when no strict sync is needed	When sender must wait for receiver	When sender should continue immediately
Data Size	Small messages likely buffered, large may block	Data size independent	Requires sufficient user-provided buffer space

Point to Point Communication in MPI

Receiving message in MPI

```
MPI_Recv(&Msg, Count, Datatype, Source, Tag, Comm, &status);
```

- Receive a message and block until the requested data is available in the application buffer in the receiving task.
- The **Msg** could be any address in receiver's address space. The **Count** specifies number of data items. The **Datatype** specifies the message type. The **Source** specifies the rank of the process which has sent the message. The **Tag** and **Comm** should be same as that is used in corresponding send operation. The status is a structure of type status which contains following information: Sender's rank, Sender's tag and number of items received
- The status argument allows the receiver to examine the properties of the message once it has been received. If the user doesn't need this information, MPI_STATUS_IGNORE can be passed.

Point to Point Communication in MPI

Finding execution time in MPI

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

- **MPI_Wtime ()**

```
double start, finish;
start = MPI_Wtime ();
   ... //MPI program segment to be clocked
finish = MPI_Wtime ();
printf ("Elapsed time is %f\n", finish - start);
```

Solved Example:

Write a MPI program using standard send. The sender process sends a number to the receiver. The second process receives the number and prints it.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
        int rank,size,x;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Status status;
        if(rank==0)
         {
          Printf("Enter a value in master process:");
          scanf("%d",&x);
          MPI_Send(&x,1,MPI_INT,1,1,MPI_COMM_WORLD);
          fprintf(stdout,"I have sent %d from process 0\n",x);
          fflush(stdout);
         }
        else
        {
         MPI_Recv(&x,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
         fprintf(stdout,"I have received %d in process 1\n",x);
         fflush(stdout);
         }
       MPI_Finalize();
       return 0;
}
```

Point to Point Communication in MPI

Seven basic MPI operations

Many parallel programs can be written and evaluated just by using the following seven MPI operations that have been overviewed in the previous sections:

```
MPI_INIT,
MPI_FINALIZE,
MPI_COMM_SIZE,
MPI_COMM_RANK,
MPI_SEND,
MPI_RECV,
MPI_WTIME.
```

Collective MPI Communication

Objectives:

1. Understand the usage of collective communication in MPI
2. Learn how to broadcast messages from root
3. Learn and use the APIs for distributing values from root and gathering the values in the root

Collective MPI Communication

- Collective Communication routines
  - When **all processes** in a group (communicator) participate in a global communication operation, the resulting communication is called a collective communication.
- Crucial for coordinating tasks across multiple processes, ensuring data consistency.
- Performs tasks like synchronization, broadcasting, gathering, and scattering data.
- The MPI collective communication routines:
  - MPI_BARRIER (comm)
  - MPI_BCAST (inbuf, incnt, intype, root, comm)
  - MPI_GATHER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)
  - MPI_SCATTER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

Collective MPI Communication

- **MPI_Bcast (inbuf, incnt, intype, root, comm)**
  - The operation implements a ***one-to-all broadcast operation*** whereby a single process named ***root*** sends its data to ***all other processes*** in the communicator, including to ***itself*** .
  - Each process receives this data from the root process, which can be of any rank.
  - The input data are located in ***inbuf*** of root process and consists of ***incnt*** data items of a specified ***intype.***
  - Each process, including the root, receives the broadcasted data. After the broadcast, the **inbuf** in every process (including the root) contains the same data.
  - After the call, the data are replicated in ***inbuf*** as output data of all remaining processes. As inbuf is used as an *input argument at the root process, but as an output argument in all remaining processes*, it is of the INOUT type.

Collective MPI Communication

- **MPI_Bcast(inbuf, incnt, intype, root, comm)**
  - **inbuf**: The buffer containing the data to be broadcasted.
    - At the root process, it serves as an input buffer.
    - In all other processes, it serves as an output buffer, where the received data will be stored.
  - **incnt**: The number of elements to be broadcasted from the root process.
  - **intype**: The data type of the elements in inbuf (e.g., MPI_INT, MPI_FLOAT).
  - **root**: The rank of the process within the communicator that serves as the source of the broadcast.
  - **comm**: The communicator that defines the group of processes participating in the broadcast.

Collective MPI Communication

- **MPI_Bcast (inbuf, incnt, intype, root, comm)**

```
int data = 100;
MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

  - In this example, the process with rank 0 (the root) broadcasts the integer 100 to all processes in MPI_COMM_WORLD.
  - After the broadcast, every process in MPI_COMM_WORLD will have data set to 100.

Collective MPI Communication

so always the root process or process we want take the input and the value is stored for it then is bcast it and the bcast shd be by all process called and all those will take this as the output so cant have this in the if() inside and all shd be outside only and all will get that value

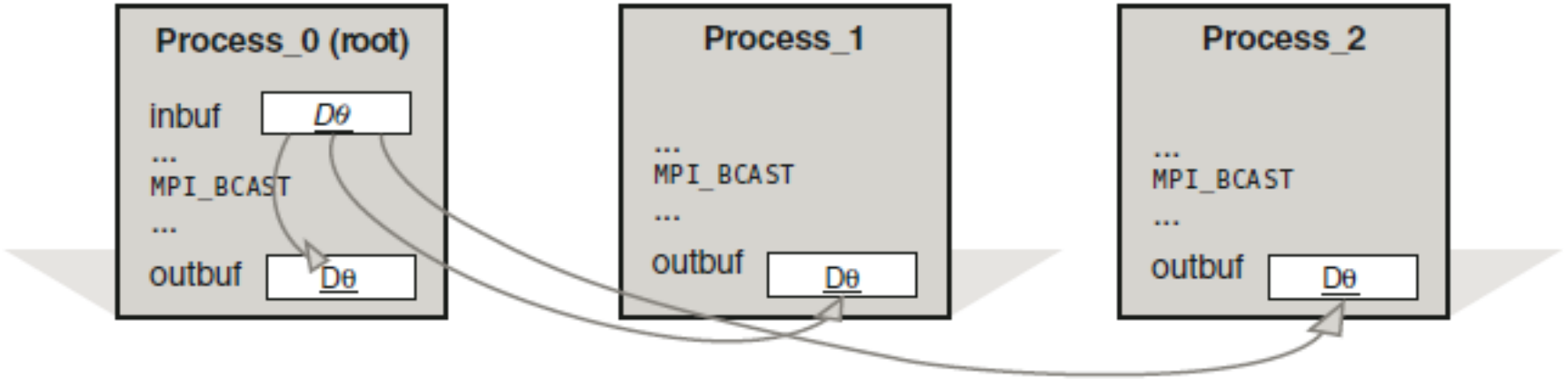


Fig. 4.4 Root process broadcasts the data from its input buffer in the output buffers of all processes

- Note that all processes have to call MPI_BCAST to complete the requested data relocation.

Collective MPI Communication

- Note that the functionality of MPI_BCAST could be implemented, in the above example, by three calls to MPI_SEND in the root process and by a single corresponding MPI_RECV call in any remaining process.
- Usually, such an implementation will be less efficient than the original MPI_BCAST.
- All collective communications could be *time-consuming*. Their *efficiency* is strongly related with the *topology and performance of interconnection network.*
- *The process ranked **Root** send the same message whose content is identified by the triple (Address, Count, Datatype) to **all** processes (including itself) in the communicator **Comm.***

Collective MPI Communication

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

int main(int argc, char *argv[]) {
    int rank, size;
    int data = 0;

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Get the rank of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // The root process initializes the data
    if (rank == 0) {
        data = 100;  // Root process sets the data
    }

    // All processes call MPI_Bcast
    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // After MPI_Bcast, all processes have the value of data
    printf("Process %d received data = %d\n", rank, data);

    // Finalize the MPI environment
    MPI_Finalize();

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ vi broadcast.c
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpicc broadcast.c -o broadcast
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpirun -np 2 ./broadcast
Process 0 received data = 100
Process 1 received data = 100
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpirun -np 4 ./broadcast
Process 0 received data = 100
Process 1 received data = 100
Process 3 received data = 100
Process 2 received data = 100
```

Collective MPI Communication

- **MPI_Gather(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)**
  - *All-to-one collective communication* is implemented by MPI_GATHER.
  - It is a collective communication operation used to collect data from all processes within a communicator and gather it to a single root process.
  - This operation is commonly used when multiple processes produce individual pieces of data that need to be aggregated.
  - This operation is also *called by all processes* in the communicator. Each process, including *root process*, sends its *input data located in inbuf that consists of incnt data items of a specified intype, to the root process*, which can be of any rank.

Collective MPI Communication

- **MPI_Gather (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)**
  - Note that the communication data can be different in count and type for each process.
  - However, the root process has *to allocate enough space, through its output buffer, that suffices for all expected data*.
  - After the return from MPI_GATHER in all processes, the data are collected in *outbuf of the root processes*.
  - After the gather operation, the root process's outbuf will contain all the gathered data, ordered by process rank.

Collective MPI Communication

- **MPI_Gather (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)**
  - **inbuf**: The address of the buffer containing the data to be sent by each process.
  - **incnt**: The number of elements in the input buffer to be sent from each process.
  - **intype**: The data type of elements in the input buffer (e.g., MPI_INT, MPI_FLOAT).
  - **outbuf**: The address of the buffer where the root process will store the gathered data. This buffer is only significant at the root process.
  - **outcnt**: The number of elements per process in the output buffer.
  - **outtype**: The data type of elements in the output buffer.
  - **root**: The rank of the process within the communicator that will receive all the gathered data.
  - **comm**: The communicator that defines the group of processes participating in the gather operation.

CollectiveMPI Communication

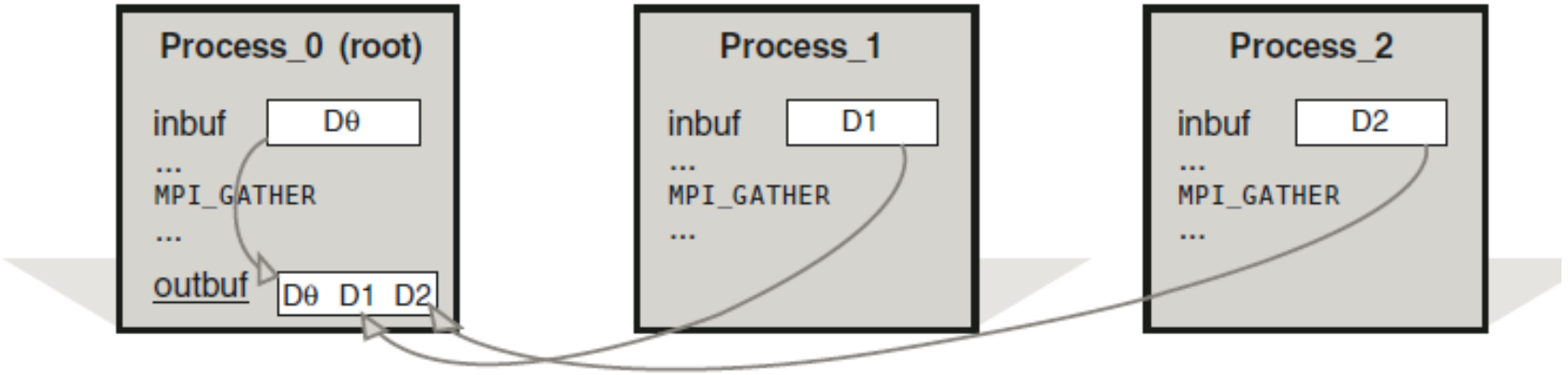


Fig. 4.5 Root process gathers the data from input buffers of all processes in its output buffer

- A schematic presentation of data relocation after the call to MPI_GATHER is shown in Fig. 4.5 for the case of three processes, where process with rank = 0 is the root process.
- Note again that all processes have to call MPI_GATHER to complete the requested data relocation.

Collective MPI Communication

```
int data = rank + 10; // Example data in each process
int recv_data[3]; // Buffer at root to receive the gathered data


MPI_Gather(&data, 1, MPI_INT, recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

- Process 0 has data = 10
- Process 1 has data = 11
- Process 2 has data = 12
- Each process sends its data (data) to the root process (rank 0). After the gather operation, **recv_data** in the root process will contain the values [10, 11, 12], assuming the ranks are 0, 1, and 2.

Collective MPI Communication

```c
int main(int argc, char *argv[]) {
    int rank, size;
    int send_data;
    int recv_data[4];  // Assume we have 4 process

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Get the rank of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Each process sets its send_data to its rank
    send_data = rank;

    // Gather all send_data values at the root process (rank 0)
    MPI_Gather(&send_data, 1, MPI_INT, recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Only the root process will print the gathered data
    if (rank == 0) {
        printf("Gathered data at root process:\n");
        for (int i = 0; i < size; i++) {
            printf("recv_data[%d] = %d\n", i, recv_data[i]);
        }
    }

    // Finalize the MPI environment
    MPI_Finalize();

    return 0;
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ vi gather.c
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpicc gather.c -o gather
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpirun -np 4 ./gather
Gathered data at root process:
recv_data[0] = 0
recv_data[1] = 1
recv_data[2] = 2
recv_data[3] = 3
```

Collective MPI Communication

- **MPI_Scatter (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)**
  - Used to distribute (scatter) data from a single root process to all processes in the communicator, including the root itself.
  - This operation works *inverse to MPI_GATHER*, i.e., it scatters data from *inbuf of process root to outbuf of all remaining processes, including itself*.
  - Used when a large data set needs to be divided among multiple processes for parallel processing.
  - Note that the count *outcnt and type outtype of the data in each of the receiver processes are the same, so, data is scattered into equal segments*.

Collective MPI Communication

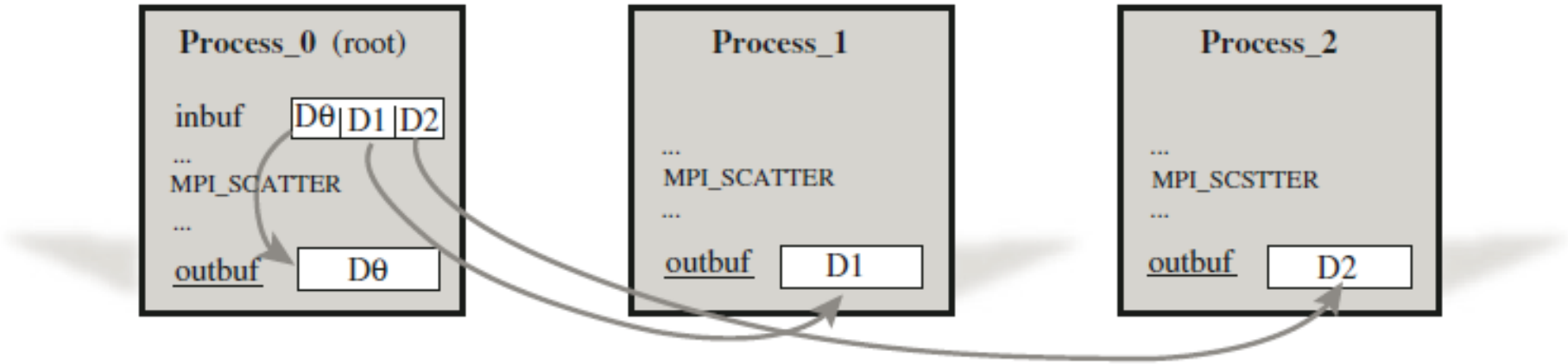


Fig. 4.6 Root process scatters the data from its input buffer to output buffers of all processes in its output buffer

A schematic presentation of data relocation after the call to MPI_SCATTER is shown in Fig. 4.6 for the case of three processes, where process with rank = 0 is the root process.

- The root process, specified by **root**, holds the data to be scattered in its **inbuf**. The data is divided into equal segments, and each segment is sent to a different process in the communicator.
- Each process, including the root, receives one segment of the data in its **outbuf**.

Collective MPI Communication

- **MPI_SCATTER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)**
  - **inbuf**: The address of the buffer in the root process containing the data to be scattered. Only significant at the root process.
  - **incnt**: The number of elements to be sent to each process from the inbuf.
  - **intype**: The data type of elements in the inbuf (e.g., MPI_INT, MPI_FLOAT).
  - **outbuf**: The address of the buffer where the scattered data will be stored in each process, including the root.
  - **outcnt**: The number of elements received by each process, which is equal to incnt.
  - **outtype**: The data type of elements in the outbuf.
  - **root**: The rank of the process within the communicator that holds the original data and scatters it to the others.
  - **comm**: The communicator that defines the group of processes participating in the scatter operation.

CollectiveMPI Communication

```
int send_data[3] = {10, 20, 30}; // Data at the root process
int recv_data; // Buffer to receive the scattered data


MPI_Scatter(send_data, 1, MPI_INT, &recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Example: The root process has an array of integers [10, 20, 30] that it wants to distribute among the three processes. After the scatter operation:

- Process 0 (root) receives 10
- Process 1 receives 20
- Process 2 receives 30

- The root process (rank 0) scatters its array **send_data** so that each process receives one element. After the scatter, **recv_data** will contain 10, 20, or 30 depending on the rank of the process.

Collective MPI Communication

- All *processes have to call MPI_SCATTER* to complete the requested data relocation.
- **MPI_GATHERV** and **MPI_SCATTERV** are more flexible variants of **MPI_GATHER** and **MPI_SCATTER**.
- These allow each process to send/receive varying amounts of data.
- incnt and outcnt become arrays of integers, specifying the number of elements for each process.
- Such extensions are possible by changing the incnt and outcnt arguments from a single integer to an array of integers, and by providing a new array argument displs for specifying the displacement relative to root buffers at which to place the processes' data.

Collective MPI Communication

```c
int main(int argc, char *argv[]) {
    int rank, size;
    int send_data[4];  // Assume we have 4 processes for this example
    int recv_data;

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Get the rank of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Root process initializes the send_data array
    if (rank == 0) {
        for (int i = 0; i < size; i++) {
            send_data[i] = i * 10;  // For example: [0, 10, 20, 30]
        }
    }

    // Scatter the data from the root process to all processes
    MPI_Scatter(send_data, 1, MPI_INT, &recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Each process prints the data it received
    printf("Process %d received data = %d\n", rank, recv_data);

    // Finalize the MPI environment
    MPI_Finalize();

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpicc scatter.c -o scatter
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpirun -np 4 ./scatter
Process 1 received data = 10
Process 2 received data = 20
Process 3 received data = 30
Process 0 received data = 0
```

Collective MPI Communication

- **MPI_Barrier(comm)**

```
MPI_Barrier(MPI_Comm comm);
```

  - Used to *synchronize the execution of a group of processes* specified within the communicator ***comm***. When a process reaches this operation, *it has to wait until all other processes have reached the MPI_BARRIER*.
  - In other words, no process returns from MPI_BARRIER until all processes have called it.
  - This mechanism ensures that all processes are aligned and can be crucial in scenarios where certain operations must be completed across all processes before moving forward.
  - *Used to ensure that messages generated in different phases do not interfere.* Prevents messages generated in one phase from interfering with messages generated in a different phase by ensuring that all processes are synchronized.
  - MPI_BARRIER is a global operation that invokes all processes; therefore, it could be **time-consuming**. In many cases, the call to MPI_BARRIER should be avoided by ***an appropriate use of explicit addressing options***, e.g., tag, source, or comm, which can provide more fine-grained control and reduce the need for global synchronization

Collective MPI Communication

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

int main(int argc, char *argv[]) {
    int rank, size;
    int sum_before = 0,sum_after=0;

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Get the rank of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Each process adds its rank to the sum
    sum_before+= rank;
    printf("Process %d added its rank. Current sum = %d\n", rank, sum_before);

    MPI_Barrier(MPI_COMM_WORLD);

    sum_after = sum_before+rank;
    printf("Process %d added its rank. Current sum = %d\n", rank, sum_after);

    // Finalize the MPI environment
    MPI_Finalize();

    return 0;
}
```

Collective MPI Communication

```
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpirun -np 4 ./barrier
Process 2 added its rank. Current sum = 2
Process 3 added its rank. Current sum = 3
Process 0 added its rank. Current sum = 0
Process 1 added its rank. Current sum = 1
Process 2 added its rank. Current sum = 4
Process 1 added its rank. Current sum = 2
Process 0 added its rank. Current sum = 0
Process 3 added its rank. Current sum = 6
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpirun -np 4 ./barrier
Process 0 added its rank. Current sum = 0
Process 1 added its rank. Current sum = 1
Process 2 added its rank. Current sum = 2
Process 3 added its rank. Current sum = 3
Process 0 added its rank. Current sum = 0
Process 1 added its rank. Current sum = 2
Process 3 added its rank. Current sum = 6
Process 2 added its rank. Current sum = 4
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpirun -np 4 ./barrier
Process 1 added its rank. Current sum = 1
Process 2 added its rank. Current sum = 2
Process 3 added its rank. Current sum = 3
Process 0 added its rank. Current sum = 0
Process 0 added its rank. Current sum = 0
Process 1 added its rank. Current sum = 2
Process 2 added its rank. Current sum = 4
Process 3 added its rank. Current sum = 6
```

With MPI_Barrier

```
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpirun -np 4 ./barrier
Process 0 added its rank. Current sum = 0
Process 0 added its rank. Current sum = 0
Process 2 added its rank. Current sum = 2
Process 2 added its rank. Current sum = 4
Process 3 added its rank. Current sum = 3
Process 3 added its rank. Current sum = 6
Process 1 added its rank. Current sum = 1
Process 1 added its rank. Current sum = 2
```

Without MPI_Barrier

Collective MPI Data Manipulations

- MPI provides a set of operations that perform several simple manipulations on the transferred data.
- These operations represent a combination of *collective communication* and *computational manipulation* in a single call and therefore simplify MPI programs.
- Collective MPI operations for data manipulation are based on data *reduction paradigm* that involves *reducing a set of numbers into a smaller set of numbers via a data manipulation using operation like summation, finding a maximum, etc*.
- For example, three pairs of numbers: {5, 1}, {3, 2}, {7, 6}, each representing the local data of a process.
- Can be reduced in a pair of maximum numbers, i.e., {7, 6}
- Or in a sum of all pair numbers, i.e., {15, 9}

Collective MPI Data Manipulations

- Reduction operations defined by MPI:
  - MPI_MAX: Returns the maximum value across all processes.
  - MPI_MIN: Returns the minimum value across all processes.
  - MPI_SUM: Returns the sum of all values across processes.
  - MPI_PROD: Returns the product of all values.
  - MPI_LAND: Performs a logical AND across all values.
  - MPI_LOR: Performs a logical OR across all values.
  - MPI_BAND: Performs a bitwise AND across all values.
  - MPI_BOR: Performs a bitwise OR across all values.
  - MPI_MAXLOC: Returns the maximum value and the rank of the process that owns it.
  - MPI_MINLOC: Returns the minimum value and the rank of the process that owns it.

Collective MPI Data Manipulations

- MPI_Reduce is a collective communication routine in MPI that enables data reduction across multiple processes.
- MPI_Reduce (inbuf, outbuf, count, type, op, root, comm).
  - The MPI_Reduce operation implements manipulation **op** on matching data items in input buffer inbuf from all processes in the communicator comm.
  - The results of the manipulation are stored in the output buffer outbuf of root process.
  - The functionality of MPI_Reduce is in fact an ***MPI_GATHER followed by manipulation op in root process.***
  - Reduce operations are implemented on a per-element basis, i.e., i^{th} elements from each process' inbuf are combined into the i^{th} element in outbuf of root process.

Collective MPI Data Manipulations

- **MPI_Reduce (inbuf, outbuf, count, type, op, root, comm)**
- inbuf: The input buffer, which contains the data to be reduced. Each process in the communicator provides its own data.
- outbuf: The output buffer, where the result of the reduction operation will be stored. This buffer is only relevant for the process designated as the root.
- count: The number of elements in the inbuf that will be reduced.
- type: The data type of the elements in inbuf (e.g., MPI_INT, MPI_FLOAT).
- op: The reduction operation to be performed (e.g., MPI_SUM, MPI_MAX). This operation is applied element-wise across the input data from all processes.
- root: The rank of the process that will store the final result in its outbuf. The reduction operation will be performed across all processes, but only the root process will contain the final result.
- comm: The communicator that defines the group of processes involved in the reduction operation.

Collective MPI Data Manipulations

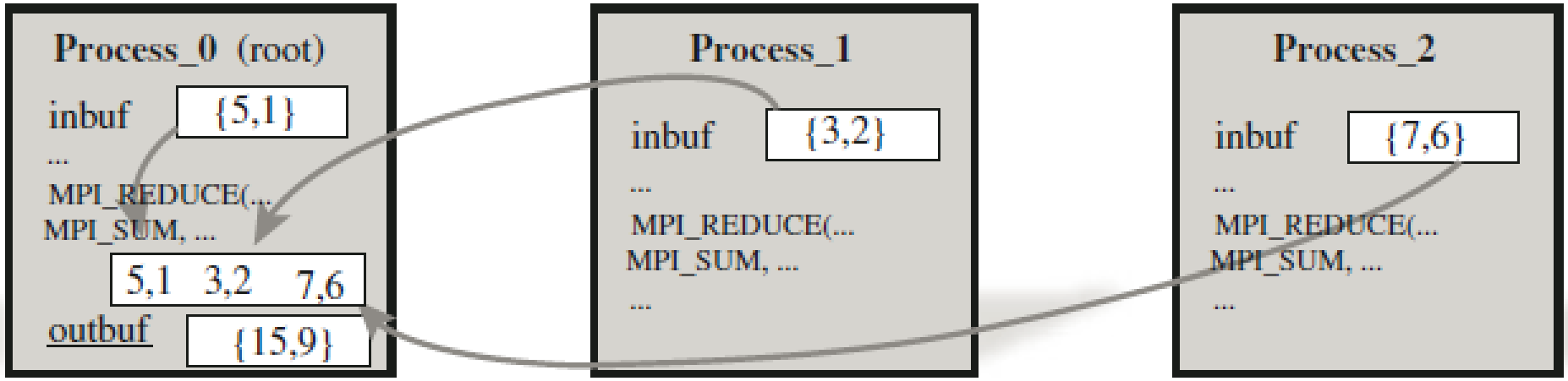


Fig. 4.7 Root process collects the data from input buffers of all processes, performs per-element `MPI_SUM` manipulation, and saves the result in its output buffer

MPI_Reduce (inbuf,outbuf,2,MPI_INT, MPI_SUM,0,MPI_COMM_ WORLD)

Before the call, inbuf of three processes with ranks 0, 1, and 2 were: {5, 1}, {3, 2}, and {7, 6}, respectively. After the call to the MPI_Reduce the value in outbuf of root process is {15, 9}

Collective MPI Data Manipulations

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Each process has a pair of numbers
    int data[2];
    data[0] = rank + 1; // First element is the rank + 1
    data[1] = rank + 2; // Second element is the rank + 2

    int sum[2];
    int max[2];

    // Perform the reduction operation to sum values
    MPI_Reduce(data, sum, 2, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Perform the reduction operation to find the maximum values
    MPI_Reduce(data, max, 2, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        // Root process prints the results
        printf("Sum of all pairs: {%d, %d}\n", sum[0], sum[1]);
        printf("Max of all pairs: {%d, %d}\n", max[0], max[1]);
    }

    MPI_Finalize();
    return 0;
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpirun -np 2 ./mpi_data_manipulation
Sum of all pairs: {3, 5}
Max of all pairs: {2, 3}
```

Each process has a pair of numbers:

{rank + 1, rank + 2}.

For example:

- Process 0 has {1, 2}
- Process 1 has {2, 3}

Collective MPI Data Manipulations

- MPI_ALLREDUCE operation is a collective communication routine in MPI that combines the functionality of MPI_REDUCE and MPI_BCAST into a single operation.
- Reduction: First, MPI_ALLREDUCE performs a reduction operation across the data in inbuf from all processes.
- Broadcast: The result of the reduction operation is then broadcast to all processes, so that every process in the communicator has the final reduced result in its outbuf.
- It works as *MPI_REDUCE followed by MPI_BCAST*.
- MPI_ALLREDUCE (inbuf, outbuf, count, type, op, comm), which *improves simplicity and efficiency of MPI programs.*
- Efficiency: MPI_ALLREDUCE is more efficient than using separate calls to MPI_REDUCE followed by MPI_BCAST because it combines these operations into one.
- Simplicity: It simplifies code by eliminating the need for a separate MPI_BCAST call.
- Note that the argument *root is not needed* anymore because the final result has to be available to all processes in the communicator.
- For the same inbuf data as in Fig. 4.7 and with MPI_SUM manipulation, a call to MPI_ALLREDUCE will produce the result {15, 9}, in output buffers of all processes in the communicator.

Collective MPI Data Manipulations

- **MPI_ALLREDUCE (inbuf, outbuf, count, type, op, comm)**
  - inbuf: The input buffer, containing the data to be reduced. Each process in the communicator provides its own data.
  - outbuf: The output buffer, where the result of the reduction operation will be stored. After the operation, all processes will have the result in their outbuf.
  - count: The number of elements in inbuf that will be reduced.
  - type: The data type of the elements in inbuf (e.g., MPI_INT, MPI_FLOAT).
  - op: The reduction operation to be performed (e.g., MPI_SUM, MPI_MAX). This operation is applied element-wise across the input data from all processes.
  - comm: The communicator that defines the group of processes involved in the operation.

Collective MPI Data Manipulations

- Example: Suppose you have multiple processes working on different parts of a calculation and you want to sum their results and then make that sum available to all processes. Using MPI_ALLREDUCE allows you to do this efficiently in a single step.
- For instance, if three processes have local data {5, 1}, {3, 2}, and {7, 6}, calling MPI_ALLREDUCE with MPI_SUM will result in every process having {15, 9} in its output buffer.

Basic MPI operations:

```
MPI_INIT, MPI_FINALIZE,
MPI_COMM_SIZE, MPI_COMM_RANK,
MPI_SEND, MPI_RECV,
```

MPI operations for collective communication:

```
MPI_BARRIER,
MPI_BCAST, MPI_GATHER, MPI_SCATTER,
MPI_REDUCE, MPI_ALLREDUCE,
```

Control MPI operations:

```
MPI_WTIME, MPI_STATUS,
MPI_INITIALIZED.
```

Communication and Computation Overlap

- Contemporary computers have separate communication (e.g., network interfaces) and calculation resources (e.g., CPUs or GPUs).
- This separation allows them to handle communication and computation tasks simultaneously.
- Therefore, they are able to execute both tasks in parallel, which is a significant potential for improving an MPI program efficiency.
- For example, instead of just waiting for a data transmission to be completed, a process can perform computations that do not depend on the communication operation. This parallel execution can reduce the overall execution time.
- If a process can perform useful work while some long communication is in progress, the overall execution time might be reduced. This approach is often termed as a **hiding latency**.

Communication and Computation Overlap

- **Example**: Process A needs to send an array of numbers to Process B and then calculate the sum of a different array of numbers.
- **Without Overlap**:
  - Process A sends the array to Process B.
  - Process A waits for confirmation that Process B has received the data.
  - Process A then starts calculating the sum of the other array.
- **With Overlap**:
  - Process A starts sending the array to Process B.
  - While the data is being transmitted, Process A starts calculating the sum of the other array (because this calculation doesn't depend on the first array).
  - By the time the sum is calculated, the data transmission is also complete, so Process A can immediately move on to the next task.
- In the second scenario, Process A is utilizing the time during communication to do other computations, effectively reducing the total time needed to complete both tasks.
- This overlapping strategy allows the process to be more efficient by reducing idle time, which is especially important in high-performance computing where tasks are distributed across many processes.

Communication and Computation Overlap

- **Communication Modes**
  - MPI processes can communicate in four different communication modes:
    - Standard
    - Buffered
    - Synchronous
    - Ready
  - Each of these modes can be performed in blocking or in non-blocking type, first being less eager to the amount of required memory for message buffering, and second being often more efficient, because of an ability to overlap the communication and computation tasks.

Communication and Computation Overlap

- **Communication Modes**
  - **Blocking Communication**
  - A **standard mode** send call, with operation MPI_SEND, should be assumed as a blocking send, which will not return until the message data and envelope have been safely stored away. The sender process can access and overwrite the send buffer with a newmessage.
  - However, depending on the MPI implementation, short messages might still be buffered while longer messages might be split and sent in shorter fragments, or they might be copied into a temporary communication buffer

Communication and Computation Overlap

- **Communication Modes**
  - **Blocking Communication**
- Because the message buffering requires extra memory space and memory-to memory copying, implementations of MPI libraries do not guarantee the amount of buffering; therefore, one has always to count on the possibility that send call will not complete until a matching receive has been posted, and the data has been moved to the receiver.
- In other words, the standard send call is **nonlocal**, i.e., may require execution of an MPI operation in another process.

Communication and Computation Overlap

- **Communication Modes**
  - **Blocking Communication**
- There are three blocking send call modes, indicated by a single-letter prefix: MPI_BSEND, MPI_SSEND, MPI_RSEND, with B for buffered, S for synchronous, and R for ready, respectively.
- The send operation syntax is the same as in the standard send, e.g., MPI_BSEND (buf, count, datatype, dest, tag, comm).

Communication and Computation Overlap

- **Communication Modes**
- The **buffered mode** send is a standard send with a user-supplied message buffering.
- It will start independent of a matching receive and can complete before a matching receive is posted.
- However, unlike the standard send, this operation is **local**, i.e., its completion is independent on the matching receive. Thus, if a buffered send is executed and no matching receive is posted, then the MPI will buffer the outgoing message, to allow the send call to complete.
- It is a responsibility of the programmer to allocate enough buffer space for all subsequent MPI_BSEND by calling MPI_BUFFER_ATTACH (bbuf, bsize).
- The buffer space bbuf cannot be reused by subsequent MPI_BSENDs if they have not been completed by matching MPI_RECVs; therefore, it must be large enough to store all subsequent messages.

Communication and Computation Overlap

- **Communication Modes**
- The **synchronous** mode send can start independently of a matching receive. However, the send will complete successfully only if a matching receive operation has started to receive the message sent by the synchronous send.
- Thus, the completion of a synchronous send not only indicates that the send buffer can be reused but also indicates that the receiver has reached a certain point in its execution, i.e., it has started executing the matching receive.
- If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication.
- As end executed in this mode is **nonlocal**, because its competition requires a cooperation of sender and receiver processes.

Communication and Computation Overlap

- **Communication Modes**
- The **ready mode** send may be started only if the matching receive has been already called.
- Otherwise, the operation is erroneous and its outcome is undefined.
- On some systems, this allows the removal of a handshake operation that is otherwise required, which could result in improved performance.
- In a correct program, a ready send can be replaced by a standard send with no effect on the program results, but with eventually improved performances.
- The receive call MPI_RECV is always blocking, because it returns only after the receive buffer contains the expected received message.

Communication and Computation Overlap

- Consider a scenario where Process A wants to send data to Process B using MPI_Rsend, and Process B is expected to receive this data with MPI_Recv.
- Correct use of Ready mode:
  - Process B first posts the MPI_Recv call.
  - Process A then calls MPI_Rsend, knowing that Process B is already waiting for the data.
- Incorrect Use of Ready Mode:
  - Process A calls MPI_Rsend without knowing whether Process B has posted the MPI_Recv.
  - If Process B has not posted the receive, the behavior of Process A's MPI_Rsend is undefined, and the program might crash.

Communication and Computation Overlap

- **Non-blocking Communication:** Non-blocking communication in MPI allows processes to initiate communication operations and then continue performing other tasks while the communication is still in progress. This can lead to better parallelism and utilization of computational resources.
- Non-blocking send start calls are denoted by a leading letter I in the name of MPI operation.
- They can use the same four modes as blocking sends: standard, buffered, synchronous, and ready, i.e., MPI_ISEND, MPI_IBSEND, MPI_ISSEND, MPI_IRSEND.
- Sends of all modes, except ready, can be started whether a matching receive has been posted or not; a non-blocking ready send can be started only if a matching receive is posted.
- In all cases, the non-blocking send start call is **local**, i.e., it returns immediately, irrespective of the status of other processes.

Communication and Computation Overlap

- Each non-blocking communication operation returns a request handle, which is a reference to the communication operation. This handle can be used later to check if the communication has completed or to wait for its completion.
- Non-blocking operations require explicit completion.
- MPI provides operations like MPI_WAIT and MPI_TEST to check the status of the communication
- MPI_WAIT: Blocks until the specified non-blocking communication operation is complete.
- MPI_TEST: Checks if the specified non-blocking communication operation is complete without blocking.
- The syntax of the non-blocking MPI operations are the same as in the standard communication mode, e.g.:
- MPI_ISEND (buf, count, datatype, dest, tag, comm, request), or
- MPI_IRECV (buf, count,datatype, dest, tag, comm, request),
- except with an additional request handle that is used for later querying.

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data[100];
    MPI_Request request;
    MPI_Status status;

    if (rank == 0) {
        // Non-blocking send
        for(int i = 0; i < 100; i++) data[i] = i;
        MPI_Isend(data, 100, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

        // Perform some computations
        for(int i = 0; i < 100000; i++) {
            // Simulating computation
        }

        // Wait for the non-blocking send to complete
        MPI_Wait(&request, &status);
    } else if (rank == 1) {
        // Non-blocking receive
        MPI_Irecv(data, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);

        // Perform some computations while waiting for the data
        int sum = 0;
        for(int i = 0; i < 100000; i++) {
            sum += i;
        }

        // Wait for the non-blocking receive to complete
        MPI_Wait(&request, &status);

        // Use the received data
        printf("Process 1 received data, first element = %d\n", data[0]);
    }

    MPI_Finalize();
    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/MPI$ mpirun -np 2 ./non_block
Process 1 received data, first element = 0
```