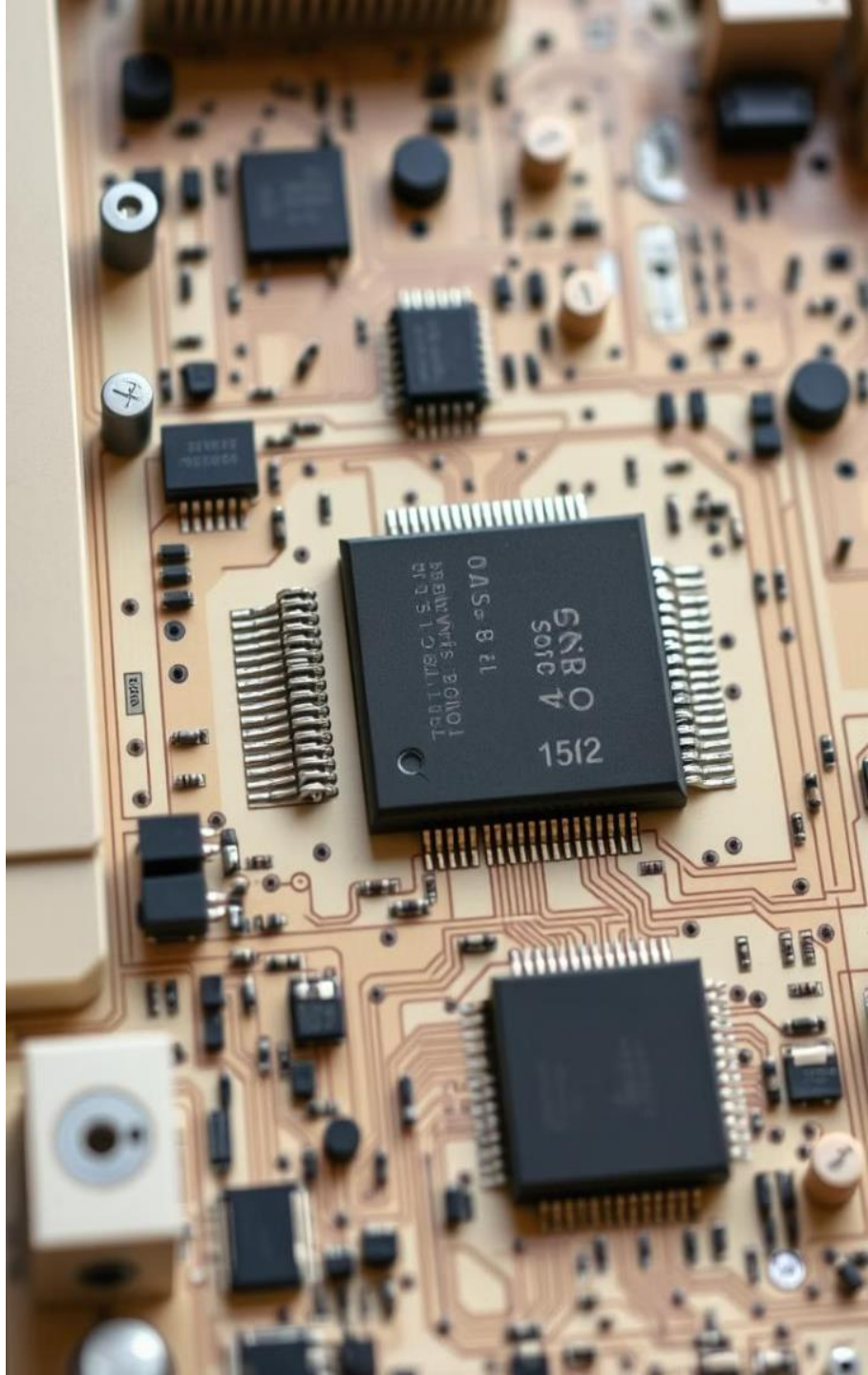
Course Title: **Parallel** and **Cloud**

Computing

Course Number: CABSXO5003

Introduction to

MPI (Message Passing Interface)



What is MPI?

MPI (Message Passing Interface) is a standardized and portable **message-passing system** designed to allow parallel programs to run on different systems ranging from multi-core processors to large-scale supercomputers.

MPI allows **parallel processes** to communicate with each other, either by sending or receiving messages, thus enabling distributed computing across different nodes.

MPI Incorporates the best ideas in a “standard” way

- Each function takes fixed arguments
- Each function has fixed semantics
  - Standardizes what the MPI implementation provides and what the application can and cannot expect
  - Each system can implement it differently as long as the semantics match

MPI is not…

- a language or compiler specification
- a specific implementation or product

Compiling and running

- MPI compilers are usually called mpicc, mpif90, mpicxx.
- These are not separate compilers, but scripts around the regular C/Fortran compiler. You can use all the usual flags.
- **Run your program with something like**

  mpiexec-n 4 hostfile ... yourprogram arguments

  mpirun-np 4 hostfile ... yourprogram arguments

MPI History

- **Version 1.0 (1994): FORTRAN 77 and C bindings**
- **Version 1.1 (1995): Minor corrections and clarifications**
- **Version 1.2 (1997): Further corrections and clarifications**
- **Version 2.0 (1997): Major enhancements**
  - → One-sided communication
  - → Parallel I/O
  - → Dynamic process creation
  - → Fortran 90 and C++ bindings
  - → Thread safety
  - → Language interoperability
- **Version 2.1 (2008): Merger of MPI-1 and MPI-2**
- **Version 2.2 (2009): Minor corrections and clarifications**
  - → C++ bindings deprecated
- **Version 3.0 (2012): Major enhancements**
  - → Non-blocking collective operations; C++ deleted from the standard

Features of MPI

- MPI is a platform for Single Program Multiple Data (SPMD) parallel computing on distributed memory architectures, with an API for sending and receiving messages
- It includes the abstraction of a "communicator", which is like an N-way communication channel that connects a set of N cooperating processes (analogous to a phaser)
- It also includes explicit datatypes in the API, that are used to describe the contents of communication buffers.

Do I need a supercomputer?

- With mpiexec and such, you start a bunch of processes that execute your MPI program.
- Does that mean that you need a cluster or a big multicore? No!
- You can start a large number of MPI processes, even on your laptop.
- The OS will use ‘time slicing’. Of course it will not be very efficient...

How does MPI know where to run?

- On a PC/laptop, it will create multiple processes and let the OS deal with it.

Advantages of MPI:

- **Scalability**: MPI is highly scalable and can be used for small clusters or large supercomputers.
- **Portability**: Since MPI is standardized, programs written with MPI can be run on different hardware and software architectures without modification.
- **Efficiency**: MPI is designed for high-performance computing, providing efficient communication mechanisms for distributed systems.
- **Flexible Communication**: MPI supports both **point-to-point** and **collective** communication, allowing it to handle various types of parallel applications.

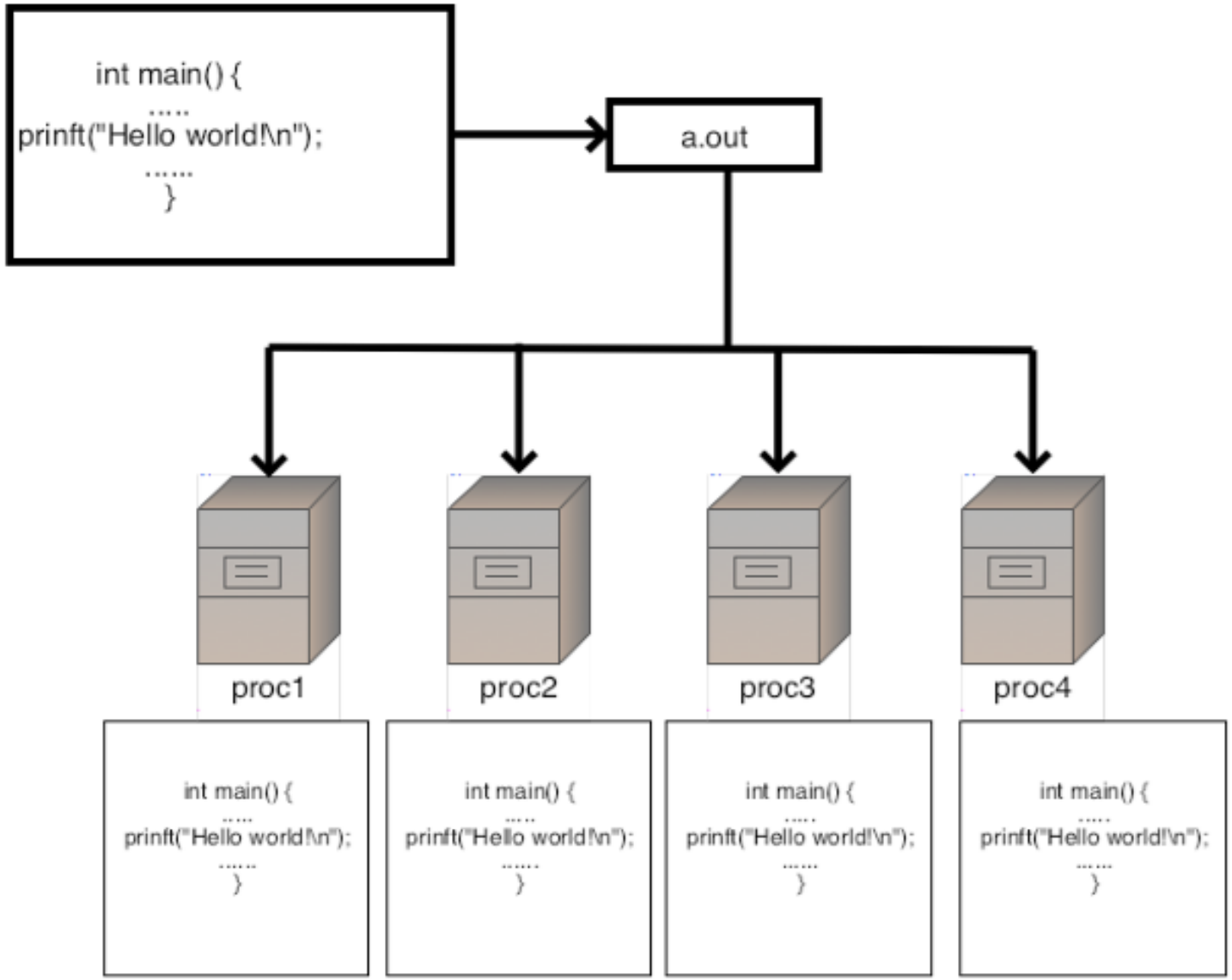
Disadvantages of MPI:

- **Complexity**: MPI requires explicit communication handling, which can increase the complexity of programming.
- **Manual Data Management**: The programmer is responsible for managing communication and synchronization between processes, unlike shared-memory models.
- **Limited Shared Memory**: MPI does not provide shared memory between processes, which can be a limitation for some parallel applications.

Applications of MPI:

- **Scientific Simulations**: MPI is commonly used in simulations that require large-scale parallelism, such as climate modeling, molecular dynamics, and astrophysics.
- **Finite Element Analysis (FEA)**: MPI is used in FEA to divide the problem into smaller parts and solve it across multiple processors.
- **Machine Learning**: MPI can be used to parallelize distributed training of large models across multiple nodes in machine learning frameworks.
- And many more ….

In a picture


int main() {
.....
prinft("Hello world!\n");
.....
}
a.out
proc1
proc2
proc3
proc4
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}

Basic requirements for an MPI program

MPI Initialization and Finalization

Every MPI program starts and ends with two basic functions:

- **MPI_Init():** This function initializes the MPI environment. It must be called before any other MPI function is invoked.
- **MPI_Finalize():** This function terminates the MPI environment. After this function is called, no other MPI functions should be used.

Without these calls, MPI operations won't work.

General MPI Program Structure

- How many processes are there?
- Who am I?

C/C++

```
#include <mpi.h>
int main(int argc, char **argv)
{
  ... some code ...
  MPI_Init(&argc, &argv);
  ... other code ...
① MPI_Comm_size(MPI_COMM_WORLD,
        &numberOfProcs);
② MPI_Comm_rank(MPI_COMM_WORLD,
        &rank);
  ... computation  & communication ...
  MPI_Finalize();
  ... wrapup ...
  return 0;
}
```

① Obtain the number of processes (ranks) in the MPI program instance

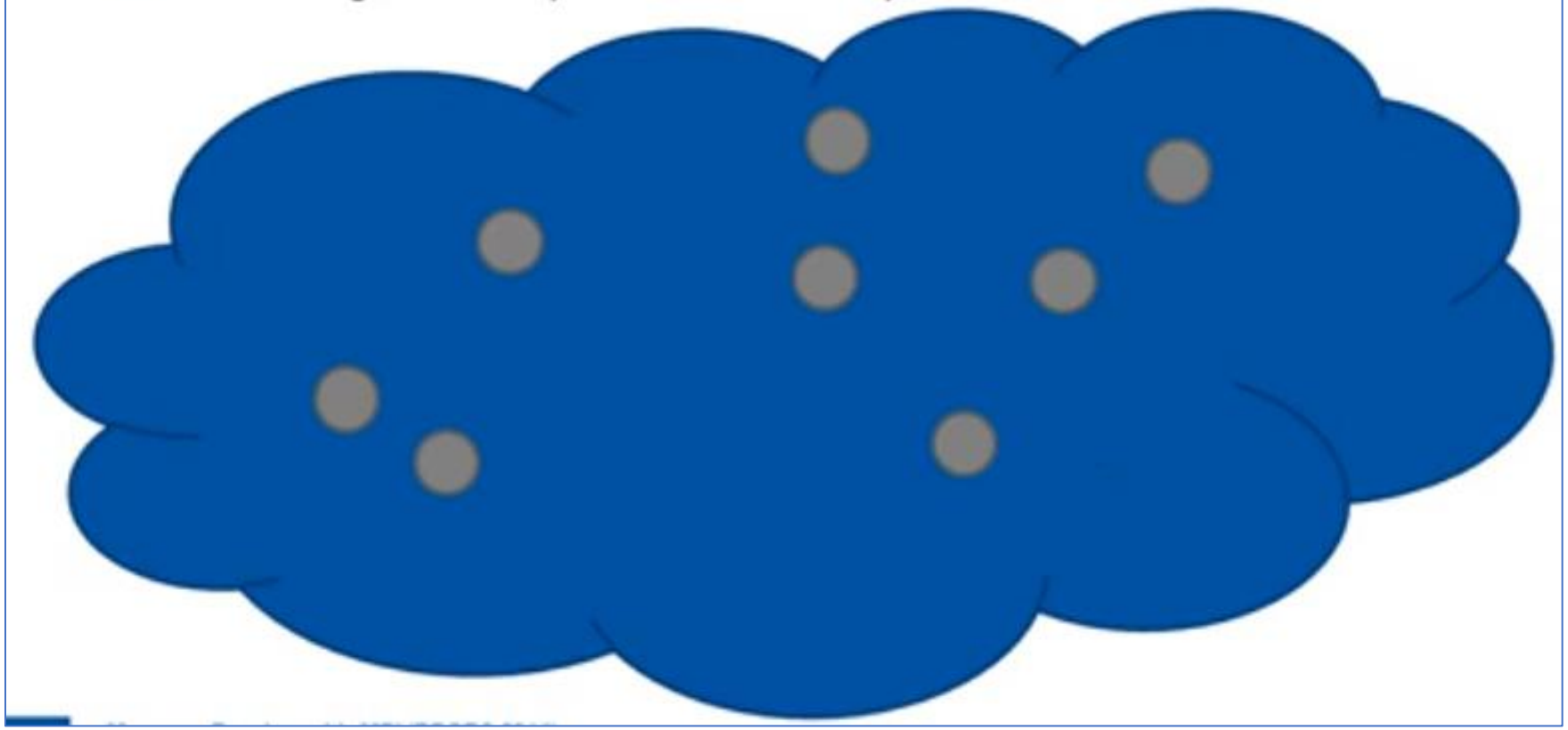
E.g. if there are 2 processes running then **numberOfProcs** will contain 2 after the call

② Obtain the identity of the calling process in the MPI program
Note: MPI processes are numbered starting from "0"

E.g. if there are 2 processes running then **rank** will be "0" in the first process and "1" in the second process after the call

MPI Basics: Ranks

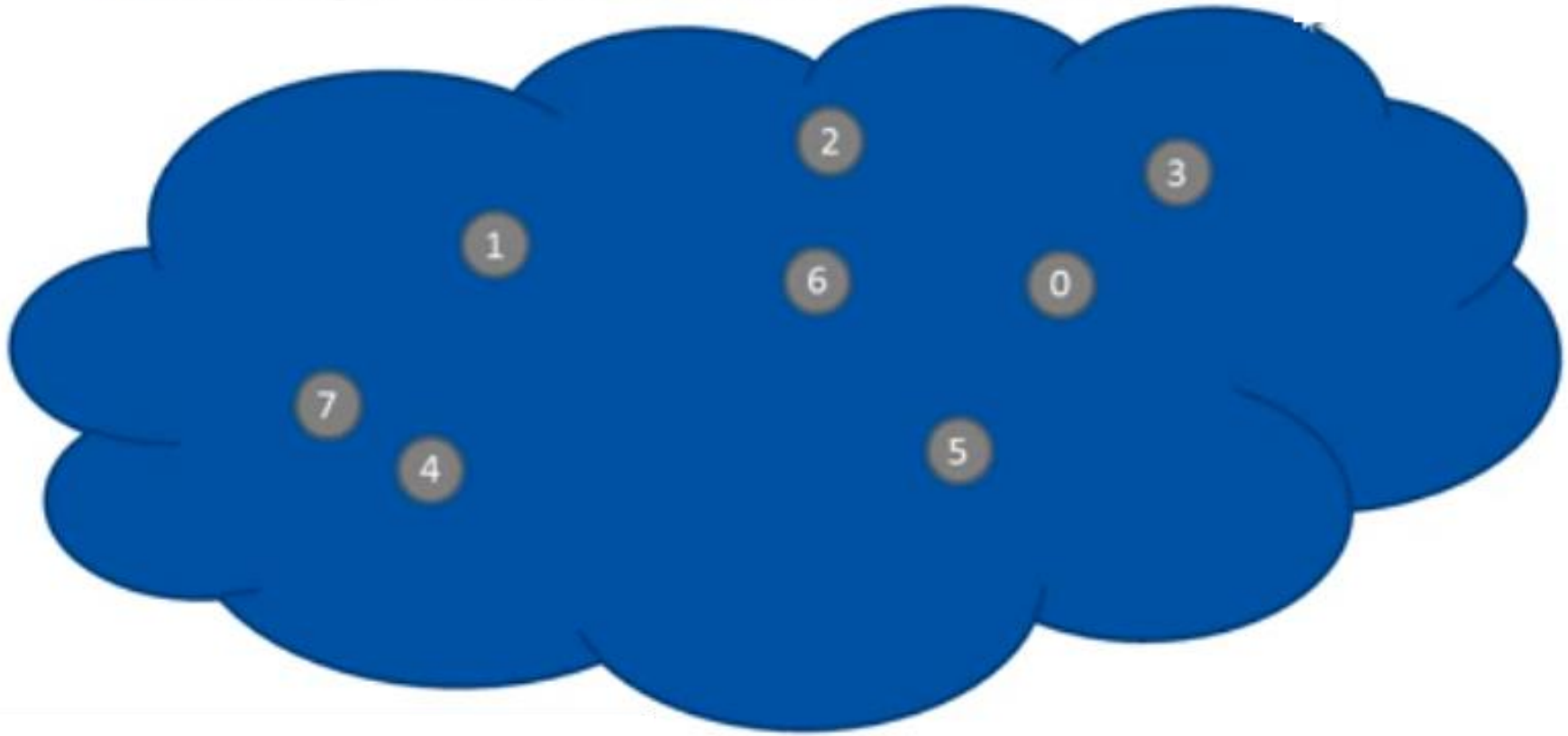
- **Processes in an MPI program are initially indistinguishable**
- **After initialization MPI assigns each process a unique identity – rank**
  - → Ranks range from 0 up to the number of processes minus 1



MPI Basics: Ranks

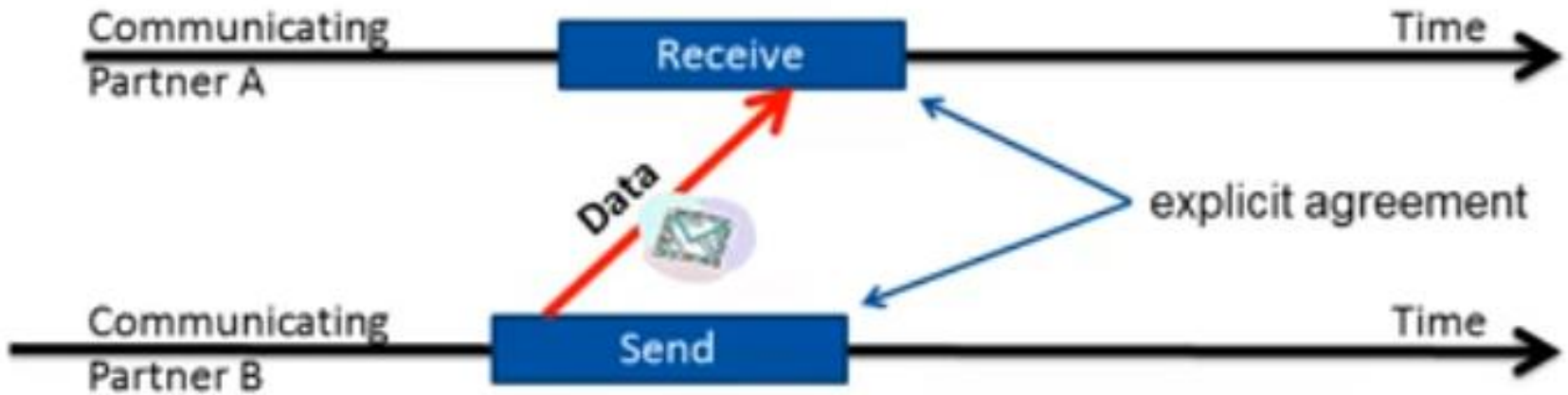
- Processes in an MPI program are initially indistinguishable
- After initialization MPI assigns each process a unique identity – rank

→ Ranks range from 0 up to the number of processes minus 1



Message Passing

- **Recall: the goal is to enable communication between processes that share no memory**



- **Required:**
  - → Send and Receive primitives (operations)
  - → Identification of both the sender and the receiver
  - → Specification of what has to be send/received

Basic methods for writing MPI program

- **int MPI_Init(int *argc, char ***argv)**

  This routine, or MPI_Init_thread, must be called before most other MPI routines are called.

- **int MPI_Comm_size(MPI_Comm comm, int *size)**

  This function indicates the number of processes involved in a communicator.

- **int MPI_Comm_rank(MPI_Comm comm, int *rank)**

  This function gives the rank of the process in the particular communicator's group.

- **int MPI_Finalize()**

  This routine cleans up all MPI states. Once this routine is called, no MPI routine (not even MPI_Init) may be called excepts, for MPI_Get_version, MPI_Initialized, and MPI_Finalized.

Basic requirements for an MPI program

Basic Point-to-Point Communication

In point-to-point communication, one process sends data to another process directly using MPI_Send() and MPI_Recv().

- **MPI_Send(): Sends a message from one process to another.**

  Syntax: int MPI_Send(void* buffer, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm comm);

  - buffer: The data to send.
  - count: The number of elements to send.
  - datatype: The data type of the elements (e.g., MPI_INT, MPI_FLOAT).
  - destination: The rank of the receiving process.
  - tag: A unique identifier for this message (to differentiate between messages).
  - comm: The communicator (MPI_COMM_WORLD is the default).

Message Passing: Sending Data

- To send a single message:

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

C/C++

- → **data:** location in memory of the data to be sent
- → **count:** number of data elements to be sent
- → **type:** MPI datatype of the data elements
- → **dest:** rank of the receiver
- → **tag:** additional identification of the message (color, tag, etc.)
  ranges from 0 to UB (impl. dependant but no less than 32767)
- → **comm:** communication context (communicator)

Basic requirements for an MPI program

Basic Point-to-Point Communication

- **MPI_Recv(): Receives a message from another process.**

  Syntax: int MPI_Recv(void* buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status);

  - buffer: Where the data will be stored.
  - count: The maximum number of elements to receive.
  - datatype: The data type of the elements.
  - source: The rank of the sending process.
  - tag: The unique identifier of the message (matching the sender's tag).
  - status: Information about the message received (source, length, etc.).

Message Passing: Receiving Data

- To receive a single message:

```
MPI_Recv (void *data, int count, MPI_Datatype type,
          int source, int tag, MPI_Comm comm, MPI_Status *status)
```

C/C++

- → **data:** location of the receive buffer
- → **count:** size of the receive buffer in data elements
- → **type:** MPI datatype of the data elements
- → **source:** rank of the sender or the **MPI_ANY_SOURCE** wildcard
- → **tag:** message tag or the **MPI_ANY_TAG** wildcard
- → **comm:** communication context
- → **status:** status of the receive operation or **MPI_STATUS_IGNORE**

Message Passing: MPI Datatypes

- **MPI provides many predefined datatypes for each language binding:**
  - → Fortran
  - → C/C++

MPI data type	C data type
MPI_CHAR	char
MPI_SHORT	short
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_UNSIGNED_CHAR	unsigned char
...	...
MPI_BYTE	-

  - → User-defined data types

Message Passing: A Full Example

C/C++

```
#include <mpi.h>
int main(int argc, char **argv)
{
  int numProcs, rank, data;
  MPI_Status status;
①  MPI_Init(&argc, &argv);
②  MPI_Comm_size(MPI_COMM_WORLD,
        &numProcs);
②  MPI_Comm_rank(MPI_COMM_WORLD,
        &rank);
③  if (rank == 0)
④    MPI_Recv(&data, 1, MPI_INT, 1, 0,
          MPI_COMM_WORLD, &status);
④  else if (rank == 1)
④    MPI_Send(&data, 1, MPI_INT, 0, 0,
          MPI_COMM_WORLD);
⑤  MPI_Finalize();
  return 0;
}
```

① Initialize the MPI library

② Get entity identification

③ Do smth different in each/a rank

④ Communicate

⑤ Clean up the MPI library

Example: Default Communicator (MPI_COMM_WORLD)

In a basic MPI program, MPI_COMM_WORLD is used to define the group of processes that participate in the communication.

This program initializes MPI, determines the number of processes and each process's rank, and prints this information.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {

    MPI_Init(&argc, &argv);  // Initialize MPI environment
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);  // Get the total number of processes

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

    printf("Process %d out of %d\n", world_rank, world_size);

    MPI_Finalize();  // Finalize MPI environment
    return 0;

}
```

Possible outputs

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 4 MPIExample.exe

Process 0 out of 4
Process 2 out of 4
Process 1 out of 4
Process 3 out of 4

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 8 MPIExample.exe

Process 7 out of 8
Process 5 out of 8
Process 1 out of 8
Process 3 out of 8
Process 0 out of 8
Process 4 out of 8
Process 2 out of 8
Process 6 out of 8

//Example: Simple MPI Send and Receive

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);  // Initialize the MPI environment

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

    if (world_rank == 0) {
        int number = 100;
        // Send number to process 1
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent number %d to process 1\n", number);
    }
    else if (world_rank == 1) {
        int number;
        // Receive number from process 0
        MPI_Recv(&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();  // Finalize the MPI environment
    return 0;
}
```

Possible outputs

mpiexec -n 2 MPIExample.exe

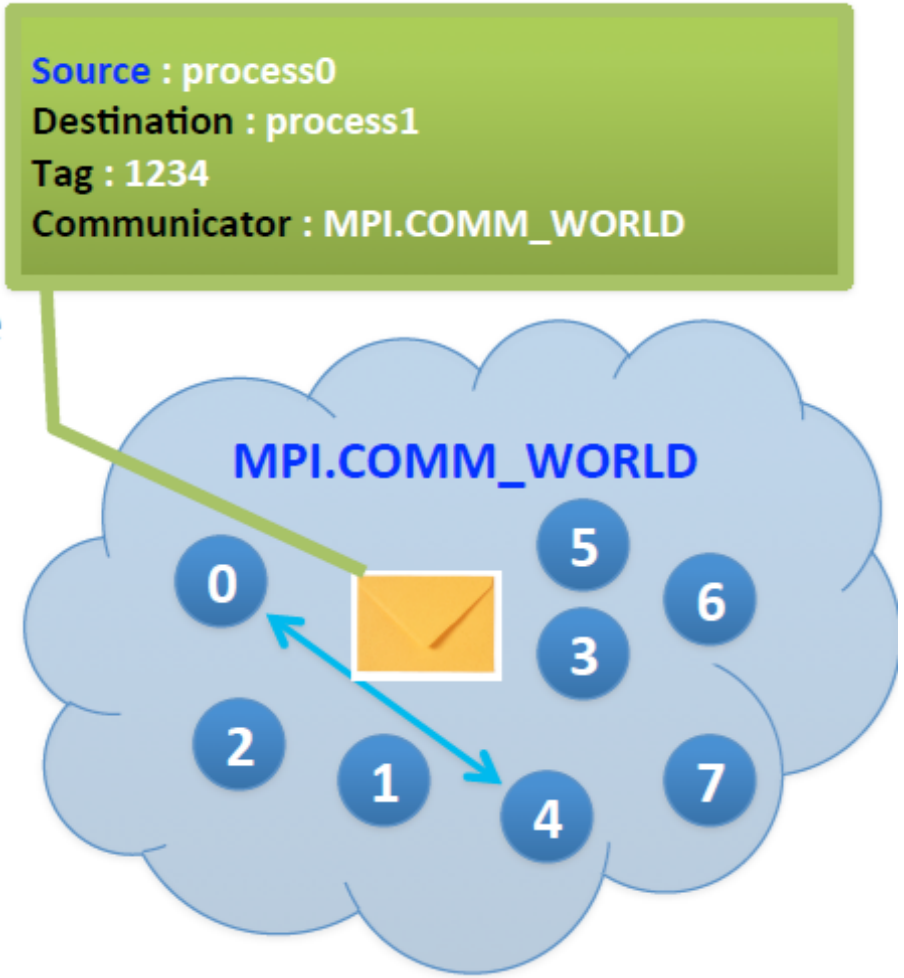
Process 0 sent number 100 to process 1
Process 1 received number 100 from process 0

mpiexec -n 4 MPIExample.exe

Process 1 received number 100 from process 0
Process 0 sent number 100 to process 1

Message Envelope

- Communication across process is performed using messages.
- Each message consists of a fixed number of fields that is used to distinguish them, called the Message Envelope :
  - —Envelope comprises source, destination, tag, communicator
  - —Message comprises Envelope + data
- Communicator refers to the namespace associated with the group of related processes



Message Passing:
Message Envelope and Matching

- Reception of MPI messages is done by matching their envelope
- Recall: MPI_Send

```
MPI_Send (void *data, int count, MPI_Datatype type,
          int dest, int tag, MPI_Comm comm)
```
C/C++

- Message Envelope:

	Sender	Receiver
Source	Implicit	Explicit, wildcard possible (MPI_ANY_SOURCE)
Destination	Explicit	Implicit
Tag	Explicit	Explicit, wildcard possible (MPI_ANY_TAG)
Communicator	Explicit	Explicit

- Recall: MPI_Recv

```
MPI_Recv (void *data, int count, MPI_Datatype type,
          int source, int tag, MPI_Comm comm, MPI_Status *status)
```
C/C++

Message Passing: Message Envelope and Matching

- Reception of MPI messages is done by matching their envelope
- Recall: MPI_Send

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

C/C++

- Message Envelope:

	Sender	Receiver
Source	Implicit	Explicit, wildcard possible (MPI_ANY_SOURCE)
Destination	Explicit	Implicit
Tag	Explicit	Explicit, wildcard possible (MPI_ANY_TAG)
Communicator	Explicit	Explicit

Message Envelope

- Recall: MPI_Recv

```
MPI_Recv (void *data, int count, MPI_Datatype type,
          int source, int tag, MPI_Comm comm, MPI_Status *status)
```

C/C++

Message Passing:
Message Envelope and Matching

- Reception of MPI messages is also dependent on the data.
- Recall:

```
MPI_Send (void *data, int count, MPI_Datatype type,
          int dest, int tag, MPI_Comm comm)                    C/C++

MPI_Recv (void *data, int count, MPI_Datatype type,
          int source, int tag, MPI_Comm comm, MPI_Status *status)   C/C++
```

- The standard expects datatypes at both ends to match

  → Not enforced by most implementations

- For a receive to complete a matching send has to be posted and v.v.
- Beware: sends and receives are atomic w.r.t. the message

Rank 0:
MPI_Send(myArr,1,MPI_INT,1,0,MPI_COMM_WORLD)
... some code ...
MPI_Send(myArr,1,MPI_INT,1,0,MPI_COMM_WORLD)

Rank 1:
MPI_Recv(myArr,2,MPI_INT,0,0,MPI_COMM_WORLD,&stat)
... some code ...

Message Passing: Message Reception

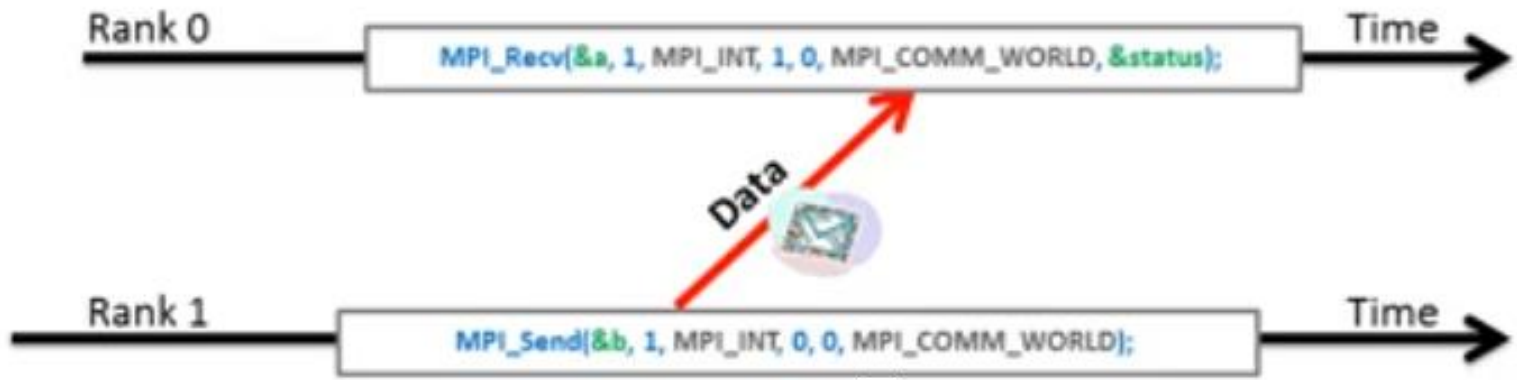
- **The receive buffer must be able to hold the whole message**
  - → send count ≤ receive count -> **OK** (but check status)
  - → send count > receive count -> **ERROR** (message truncated)
- **The MPI status object holds information about the received message**
- **C/C++: MPI_Status status;**
  - → **status.MPI_SOURCE** message source rank
  - → **status.MPI_TAG** message tag
  - → **status.MPI_ERROR** receive status code

MPI Basics:
Return Values and Error Handling

- Almost every C/C++ MPI call returns an integer error code:
  - → int MPI_Send(...)
- Fortran MPI calls take an extra INTEGER output argument (always last in the argument list) where the error code is returned:
  - → SUBROUTINE MPI_SEND(..., ierr)
- Error codes indicate the success of the operation:
  - → C/C++ MPI_SUCCESS == MPI_Send(...)
  - → Failure indicated by error code different from MPI_SUCCESS
- If an error occurs, an MPI error handler is called before the operation returns. The default MPI error handler aborts the MPI job!
- Note: MPI error code values are implementation specific.

Message Passing: The MPI Way

- **Message passing in MPI is explicit:**



- **These two calls transfer the value of the *b* variable in rank 1 into the *a* variable in rank 0.**

- **For now assume that *comm* is fixed as MPI_COMM_WORLD.**

Understanding Communicators in MPI

- A communicator in MPI is a powerful concept that defines a group of processes that can communicate with each other.
- The communicator manages message-passing contexts and process grouping, ensuring messages are routed to the correct destinations.
- All MPI communication functions (like MPI_Send, MPI_Recv, etc.) involve communicators to specify the group of processes involved in the communication.

Understanding Communicators in MPI

Key Concepts Related to Communicators

- **Default Communicator: MPI_COMM_WORLD**

  When you initialize an MPI program using MPI_Init, all processes become part of the default communicator, called MPI_COMM_WORLD. This communicator includes all the processes launched for the program, and each process has a unique rank within this communicator.

  a) Rank: A unique identifier for each process within a communicator. The rank starts from 0 and increments by 1 for each process. If there are 8 processes, ranks will range from 0 to 7.

  b) Size: The total number of processes in the communicator.

Understanding Communicators in MPI

Key Concepts Related to Communicators

- **Creating Communicators: MPI_Comm_split() and MPI_Comm_create()**

  Often, you'll want to divide the processes into smaller groups or subsets. MPI allows you to create new communicators using two main functions:

  - MPI_Comm_split(): Splits an existing communicator into multiple, smaller communicators based on color or key values. All processes with the same color end up in the same new communicator.
  - MPI_Comm_create(): Creates a new communicator based on an existing group of processes.

- **Communicators and Process Isolation**

  Processes in one communicator are isolated from processes in another communicator. This means:

  - A process can only communicate with other processes within the same communicator.
  - Messages sent within one communicator cannot be received by processes in another communicator, even if their ranks match.

Example: Splitting a Communicator (MPI_Comm_split)

Suppose you have 8 processes, and you want to split them into two groups based on their ranks—one for even-numbered processes and one for odd-numbered processes. Here's how you can do it using MPI_Comm_split():

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);  // Initialize MPI environment

    int world_rank, world_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);  // Get rank
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);  // Get size

    int color = world_rank % 2;  // Split based on whether the rank is even or odd

    // Create a new communicator based on color
    MPI_Comm new_comm;
    MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &new_comm);

    // Get the rank and size of the new communicator
    int new_rank, new_size;
    MPI_Comm_rank(new_comm, &new_rank);
    MPI_Comm_size(new_comm, &new_size);

    printf("World Rank: %d, New Rank: %d, New Size: %d\n", world_rank, new_rank, new_size);

    MPI_Comm_free(&new_comm);  // Free the communicator when no longer needed
    MPI_Finalize();  // Finalize MPI environment
    return 0;
}
```

Explanation:

- color = world_rank % 2: Processes are split based on whether their rank is even or odd.
  - Even ranks get a color of 0, and odd ranks get a color of 1.
- MPI_Comm_split(): Creates two new communicators, one for even-ranked processes and another for odd-ranked processes.
  - All processes with the same color form a new communicator.
- After splitting, processes in the new communicator are assigned new ranks (starting from 0) within their respective groups.

Possible outputs

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 8 MPIExample.exe

World Rank: 2, New Rank: 1, New Size: 4
World Rank: 7, New Rank: 3, New Size: 4
World Rank: 1, New Rank: 0, New Size: 4
World Rank: 4, New Rank: 2, New Size: 4
World Rank: 0, New Rank: 0, New Size: 4
World Rank: 3, New Rank: 1, New Size: 4
World Rank: 6, New Rank: 3, New Size: 4
World Rank: 5, New Rank: 2, New Size: 4

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 4 MPIExample.exe

World Rank: 0, New Rank: 0, New Size: 2
World Rank: 2, New Rank: 1, New Size: 2
World Rank: 1, New Rank: 0, New Size: 2
World Rank: 3, New Rank: 1, New Size: 2

MPI Deadlock

In MPI (Message Passing Interface) programming, **deadlock** occurs when two or more processes are stuck, waiting for each other to send or receive a message, and none of them can proceed. This happens when the communication pattern creates a situation where processes are dependent on each other's actions in a way that none can continue.

How Deadlock Occurs in MPI

Deadlock typically arises in scenarios involving blocking communication routines like MPI_Send and MPI_Recv. In these routines, a process calling MPI_Send may wait until the corresponding MPI_Recv is called by the destination process, and vice versa. If both processes are trying to send and receive from each other simultaneously, and both are waiting, they enter a deadlock.

MPI Deadlock

Example 1: Simple Pairwise Deadlock

Here is a classic example of deadlock involving two processes:

Code for Process 0

```
MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
MPI_Recv(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

Code for Process 1

```
MPI_Send(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

What's Happening:

- **Process 0** is attempting to send a message to **Process 1**.
- **Process 1** is attempting to send a message to **Process 0**.
- Both processes are waiting for each other to receive their message, but neither of them can proceed until the other completes its send. Therefore, both are blocked.

```c
/*Explanation of the Deadlock
Process 0 executes MPI_Send to send data to Process 1, but it cannot proceed until Process 1
receives the message.
Process 1 also executes MPI_Send to send data to Process 0 and similarly cannot proceed
until Process 0 receives its message.
Both processes are blocked, waiting for the other to receive, resulting in a deadlock.*/

#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);

    int send_data = rank;    // Each process sends its rank
    int recv_data;

    if (size != 2) {
        printf("This program requires exactly 2 processes.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
```

```c
/*Deadlock*/

if (rank == 0) {
        // Process 0 sends a message to Process 1 and waits to receive from Process 1
        MPI_Send(&send_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&recv_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    else if (rank == 1) {
       // Process 1 sends a message to Process 0 and waits to receive from Process 0
        MPI_Send(&send_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
       MPI_Recv(&recv_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    printf("Process %d received %d\n", rank, recv_data);

    MPI_Finalize();
    return 0;
}
```

Possible outputs

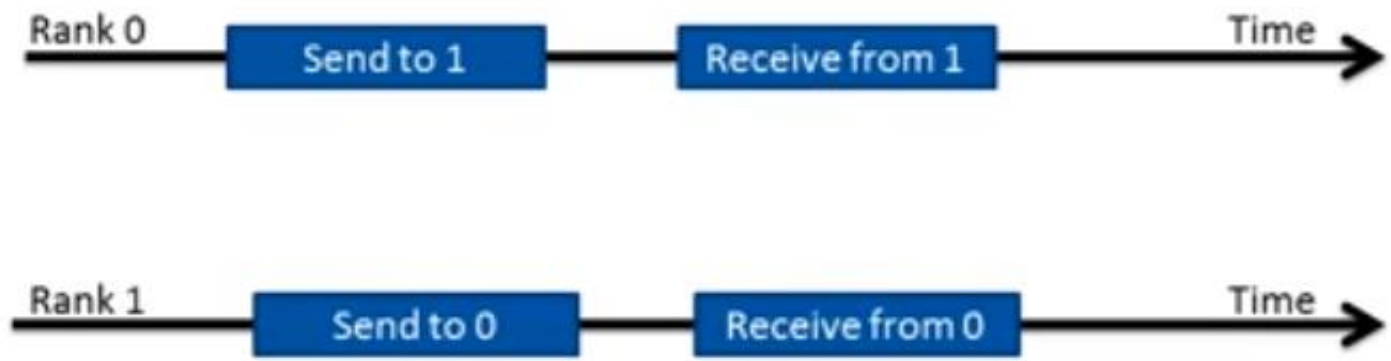
E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 4 MPIExample.exe

```
job aborted:
[ranks] message

[0] application aborted
aborting MPI_COMM_WORLD (comm=0x44000000), error 1, comm rank 0

[1] application aborted
aborting MPI_COMM_WORLD (comm=0x44000000), error 1, comm rank 1

[2] application aborted
aborting MPI_COMM_WORLD (comm=0x44000000), error 1, comm rank 2

[3] application aborted
aborting MPI_COMM_WORLD (comm=0x44000000), error 1, comm rank 3

---- error analysis -----

[0-3] on DESKTOP-R1J709G
MPIExample.exe aborted the job. abort code 1
```

Message Passing: Deadlocks

- **Both MPI_Send and MPI_Recv calls are blocking:**
  - → Blocking calls only return once the operation has completed
  - → The receive operation only returns after a matching message has arrived
  - → The send operation might be buffered and return before the message is sent

- **Deadlock in a typical data exchange scenario:**



Example 2: Multi-process Deadlock

In programs with more than two processes, deadlock can happen in more complex communication patterns, such as a **ring** communication:

Code Example (3 processes):

```
if (rank == 0) {
   MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
   MPI_Recv(&data, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
   MPI_Send(&data, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
   MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
} else if (rank == 2) {
   MPI_Send(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
   MPI_Recv(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
}
```

What's Happening:

Each process is waiting for the next process to receive the data it is sending, creating a circular wait. If the processes try to execute their MPI_Send operations first, deadlock occurs.

How to Avoid Deadlock:

There are several strategies to avoid deadlock in MPI.

Use MPI_Sendrecv

MPI_Sendrecv is a handy routine that combines sending and receiving into one operation, ensuring no deadlock occurs.Code using Non-blocking:

```
MPI_Sendrecv(&data_send, 1, MPI_INT, 1, 0, &data_recv, 1, MPI_INT, 1, 0,
                                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

This function sends and receives data simultaneously in a way that avoids the need for manual pairing of sends and receives, preventing deadlocks.

Message Passing: Combining Send and Receive

C/C++

```
MPI_Sendrecv (void *senddata, int sendcount, MPI_Datatype sendtype,
              int dest, int sendtag, void *recvdata, int recvcount,
              MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
              MPI_Status *status)
```

- Sends one message and receives one message evading deadlocks (unless unmatched).
- Send and receive buffers must not overlap!

C/C++

```
MPI_Sendrecv_replace (void *data, int count, MPI_Datatype datatype,
              int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
              MPI_Status *status)
```

- First sends a message to *dest*, then receives a message from *source*, using the same memory location, elements count and datatype for both operations.

How to Avoid Deadlock:

There are several strategies to avoid deadlock in MPI.

Use Non-blocking Communication

Non-blocking communication, such as MPI_Isend and MPI_Irecv, allows processes to continue computation or progress through the code even if the message hasn't been sent or received yet.

Code using Non-blocking:

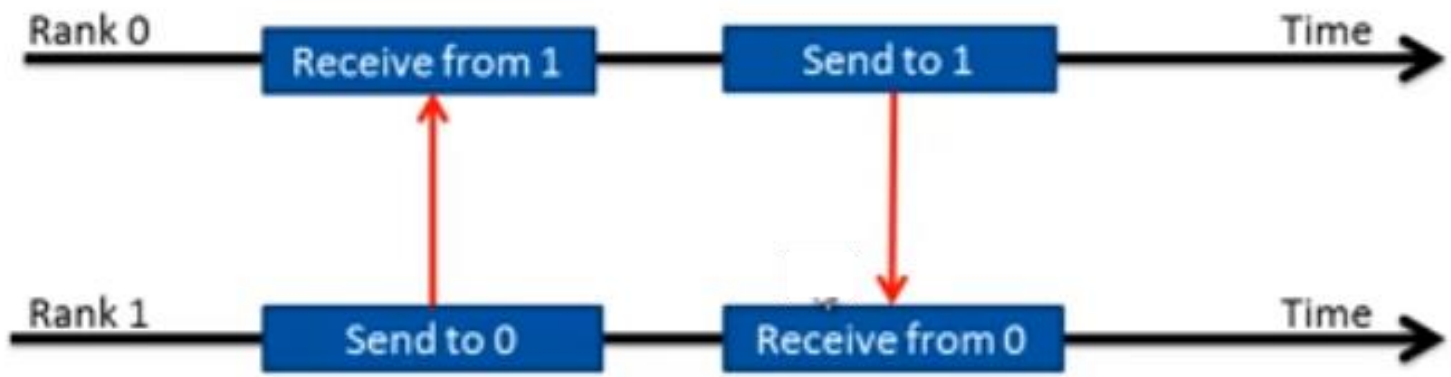
```
MPI_Request request;
MPI_Isend(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
MPI_Irecv(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);  // Wait for communication to complete
```

In this case, the send and receive operations are initiated, and then MPI_Wait is called to ensure they finish. Since the processes are not forced to wait immediately, deadlock is avoided.

Message Passing: Deadlocks

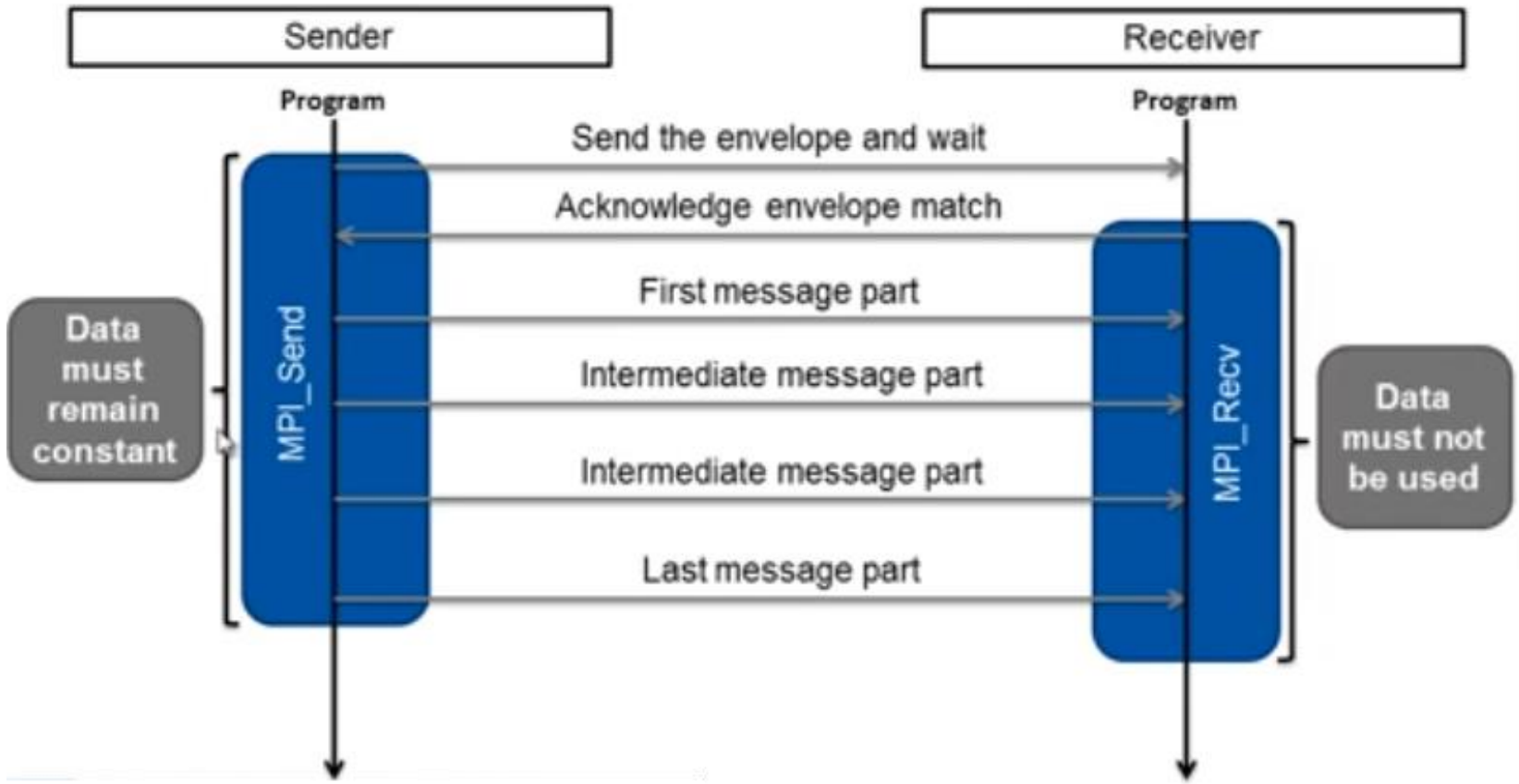
- **Both MPI_Send and MPI_Recv calls are blocking:**
  - → Blocking calls only return once the operation has completed
  - → The receive operation only returns after a matching message has arrived
  - → The send operation might be buffered and return before the message is sent

- **Deadlock resolution in a typical data exchange scenario:**



Non-Blocking Operations: Overview

- **Blocking send (w/o buffering) and receive operate as follows:**



Non-blocking messages

MPI also offers non-blocking versions.

These functions all return immediately, and provide a “request” object that we can then either wait for completion with or inspect to check if the message has been sent/received.

The function signatures for MPI_Isend and MPI_Irecv are:

```
int MPI_Isend(const void *buffer, int count, MPI_Datatype dtype, int
dest, int tag, MPI_Comm comm, MPI_Request *request);

int MPI_Irecv(void *buffer, int count, MPI_Datatype dtype, int dest,
int tag, MPI_Comm comm, MPI_Request *request);
```

Notice how the send gets an extra output argument (the request), and the receive loses the `MPI_Status` output argument and gains a request output argument.

```c
/*To avoid deadlock in MPI, there are several approaches you can use:
Use Non-Blocking Communication (MPI_Isend / MPI_Irecv)*/

#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);

    int send_data = rank;
    int recv_data;
    MPI_Request req_send, req_recv;
    MPI_Status status;

    if (size != 2) {
        printf("This program requires exactly 2 processes.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
```

```c
/*To avoid deadlock in MPI, there are several approaches you can use:
Use Non-Blocking Communication (MPI_Isend / MPI_Irecv)*/

    if (rank == 0) {
        // Non-blocking send and receive
        MPI_Isend(&send_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req_send);
        MPI_Irecv(&recv_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req_recv);
    }
    else if (rank == 1) {
        MPI_Isend(&send_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &req_send);
        MPI_Irecv(&recv_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &req_recv);
    }

    // Wait for both send and receive to complete
    MPI_Wait(&req_send, &status);
    MPI_Wait(&req_recv, &status);

    printf("Process %d received %d\n", rank, recv_data);

    MPI_Finalize();
    return 0;
}
```

Possible outputs

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 2 MPIExample.exe
Process 1 received 0
Process 0 received 1

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 2 MPIExample.exe
Process 1 received 0
Process 0 received 1

With the blocking versions (MPI_Send, MPI_Ssend, MPI_Bsend), the buffer argument is safe to reuse *as soon as the function returns*. Equally, as soon as MPI_Recv returns, we know the message has been received and we can inspect the contents.

This is not the case for non-blocking calls.

We are not allowed to reuse the buffer (or rely on its contents being ready) until we have “waited” on the request handle.

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

int main(int argc, char* argv[]) {
    int rank, size, send_data, recv_data;
    int left, right;

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

    // Get the rank (process ID) and size (number of processes)
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Seed the random number generator
    srand(time(NULL) + rank);  // Different seed for each process

    // Generate random data to send
    send_data = rand() % 100;  // Random number between 0 and 99
```

/* MPI Program, where, each process sends a randomly generated number to its right neighbor and receives one from its left neighbor */

Explanation of Changes:

1. **Random Data Generation**:
   - We seed the random number generator using srand(time(NULL) + rank) to ensure each process generates different random numbers.
   - Each process generates a random number between 0 and 99 using rand() % 100.

2. **MPI_Sendrecv** remains the same, except now it sends and receives the random data, not the rank.

/* MPI Program, where, each process sends a randomly generated number to its right neighbor and receives one from its left neighbor */

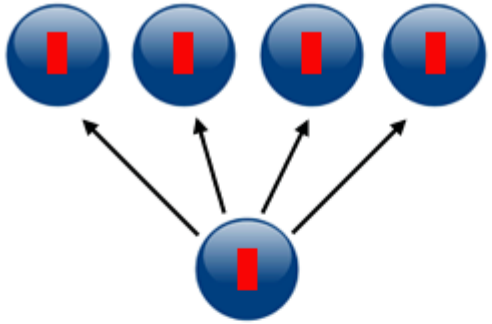
```
    // Define the neighbor processes in a circular topology (ring)
    left = (rank - 1 + size) % size;   // Neighbor on the left
    right = (rank + 1) % size;         // Neighbor on the right

    // Send random data to the right neighbor, receive data from the left neighbor
    MPI_Sendrecv(&send_data, 1, MPI_INT, right, 0,      // Send to the right
        &recv_data, 1, MPI_INT, left, 0,      // Receive from the left
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // Output the result
    printf("Process %d sent %d to process %d and received %d from process %d\n",
        rank, send_data, right, recv_data, left);

    // Finalize MPI environment
    MPI_Finalize();

    return 0;
}
```

Possible outputs

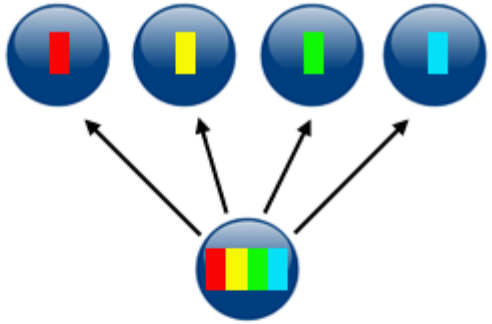
E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n **4** MPIExample.exe
Process 3 sent 11 to process 0 and received 8 from process 2
Process 2 sent 8 to process 3 and received 5 from process 1
Process 0 sent 1 to process 1 and received 11 from process 3
Process 1 sent 5 to process 2 and received 1 from process 0

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n **8** MPIExample.exe
Process 3 sent 78 to process 4 and received 75 from process 2
Process 5 sent 85 to process 6 and received 81 from process 4
Process 4 sent 81 to process 5 and received 78 from process 3
Process 1 sent 72 to process 2 and received 68 from process 0
Process 6 sent 88 to process 7 and received 85 from process 5
Process 0 sent 68 to process 1 and received 91 from process 7
Process 7 sent 91 to process 0 and received 88 from process 6
Process 2 sent 75 to process 3 and received 72 from process 1

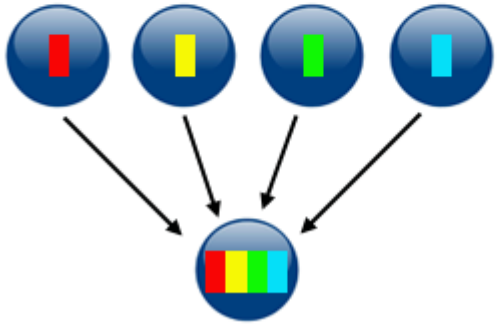
MPI Broadcast and Collective Communication



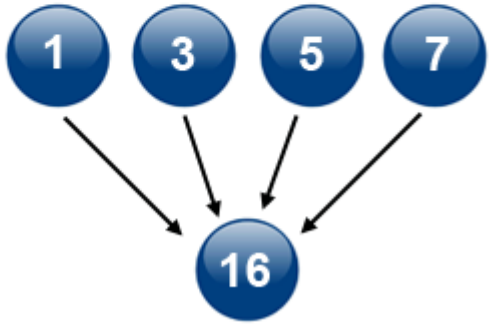
broadcast



scatter



gather



reduction

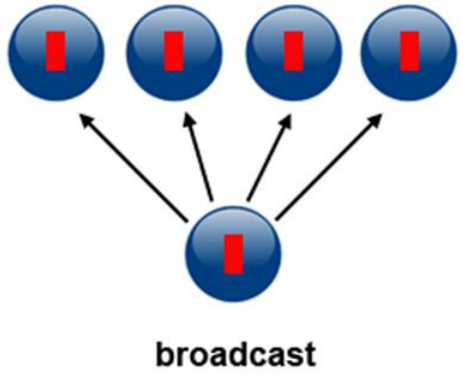
In MPI (Message Passing Interface), **group communication** or **collective communication** involves multiple processes and ensures synchronized communication among them.

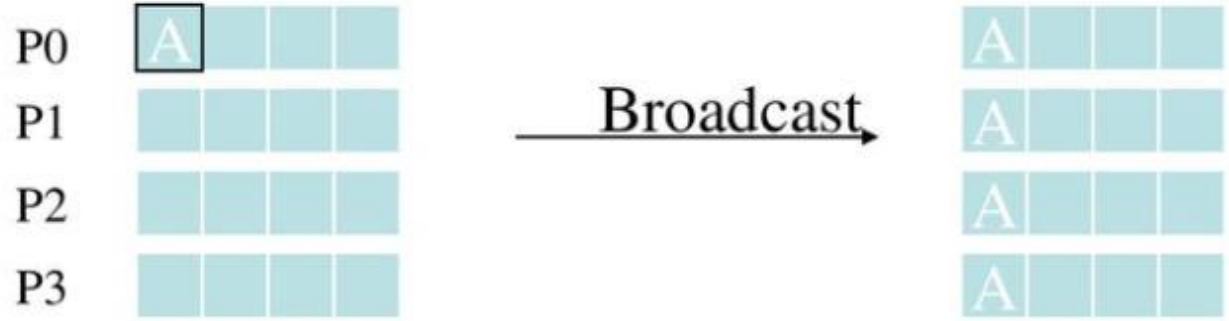
Types of Collective Operations:

- **Synchronization** - processes wait until all members of the group have reached the synchronization point.
- **Data Movement** - broadcast, scatter/gather, all to all.
- **Collective Computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

MPI_Bcast (Broadcast)

MPI_Bcast is used to broadcast data from one process (the root process) to all other processes in a communicator.

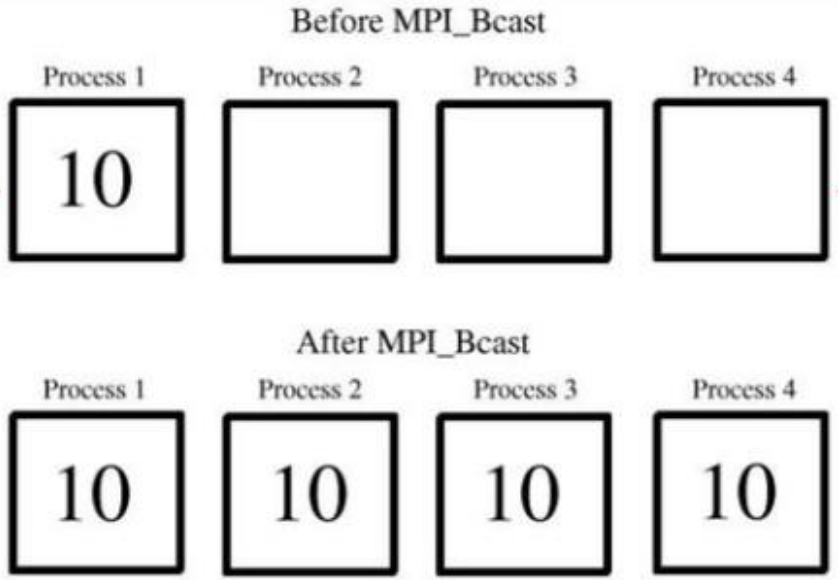




int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

buffer: The starting address of the buffer (where the data is).
count: Number of elements in the buffer.
datatype: Data type of each element (e.g., MPI_INT, MPI_FLOAT).
root: The rank of the root process that sends the data.
comm: The communicator (usually MPI_COMM_WORLD).

Broadcast



- **Data belonging to a single process is sent to all of the processes in the communicator.**

```
int MPI_Bcast(
        void*         data_p        /* in/out */,
        int           count         /* in     */,
        MPI_Datatype  datatype      /* in     */,
        int           source_proc   /* in     */,
        MPI_Comm      comm          /* in     */);
```

Example: Suppose process 0 has a variable data that needs to be sent to all other processes.

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

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

    int data;
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        data = 100;  // Process 0 initializes data
    }

    // Broadcast data from process 0 to all other processes
    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Process %d received data: %d\n", rank, data);

    MPI_Finalize();
    return 0;
}
```

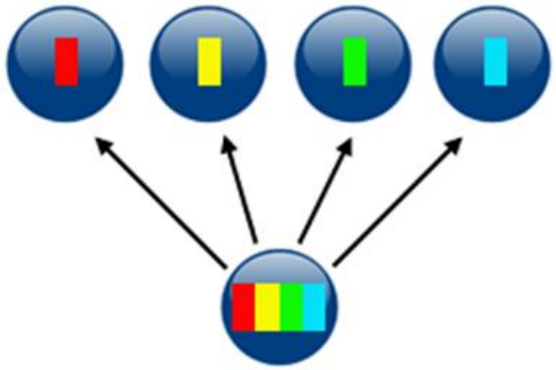
// In this case, process 0 sends data to all other processes. The same data is received by each process.

Possible outputs

```
E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 4 MPIExample.exe
Process 3 received data: 100
Process 1 received data: 100
Process 2 received data: 100
Process 0 received data: 100
```

MPI_Scatter (Scatter)

MPI_Scatter divides the data into parts and distributes them to all processes, where each process receives a unique portion of the data.



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

sendbuf: Starting address of the send buffer (on the root process).
sendcount: Number of elements to send to each process.
sendtype: Data type of the send buffer elements.
recvbuf: Starting address of the receive buffer (on each process).
recvcount: Number of elements received by each process.
recvtype: Data type of the receive buffer elements.
root: Rank of the root process.
comm: Communicator.

Example:
Process 0 has an array of integers, and each process will receive one element of this array.

```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);

    int send_data[4];  // Data to be scattered from process 0
    int recv_data;     // Each process will receive one element

    if (rank == 0) {
        // Process 0 initializes the array
        for (int i = 0; i < size; i++) {
            send_data[i] = i + 10;
        }
    }
    // Scatter the array to all processes
    MPI_Scatter(send_data, 1, MPI_INT, &recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Process %d received %d\n", rank, recv_data);
    MPI_Finalize();
    return 0;
}
```

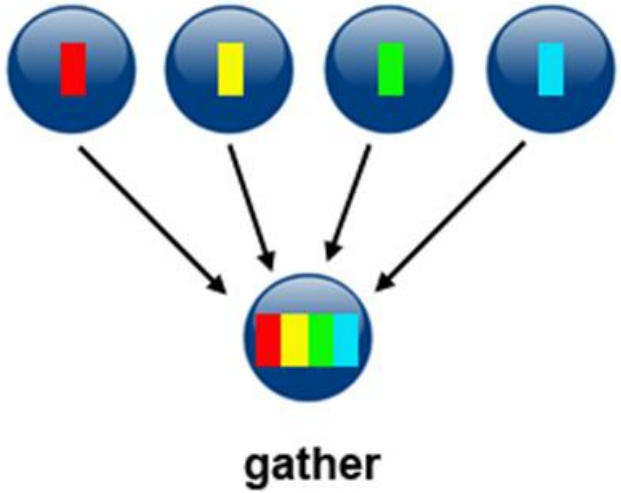
Possible outputs

```
E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 4 MPIExample.exe
Process 0 received 10
Process 3 received 13
Process 2 received 12
Process 1 received 11
```

```
E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 4 MPIExample.exe
Process 2 received 12
Process 3 received 13
Process 1 received 11
Process 0 received 10
```

MPI_Gather (Gather)

MPI_Gather is the reverse of MPI_Scatter. It gathers data from all processes and collects it into a single array on the root process.




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

sendbuf: Starting address of the send buffer (on each process).
sendcount: Number of elements sent by each process.
sendtype: Data type of the send buffer elements.
recvbuf: Starting address of the receive buffer (on the root process).
recvcount: Number of elements received from each process.
recvtype: Data type of the receive buffer elements.
root: Rank of the root process.
comm: Communicator.

```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);

    int send_data = rank + 100;  // Each process has unique data
    int recv_data[4];            // Array on process 0 to gather data

    // Gather data from all processes to process 0
    MPI_Gather(&send_data, 1, MPI_INT, recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Process 0 gathered: ");
        for (int i = 0; i < size; i++) {
            printf("%d ", recv_data[i]);
        }
        printf("\n");
    }
    MPI_Finalize();
    return 0;
}
```

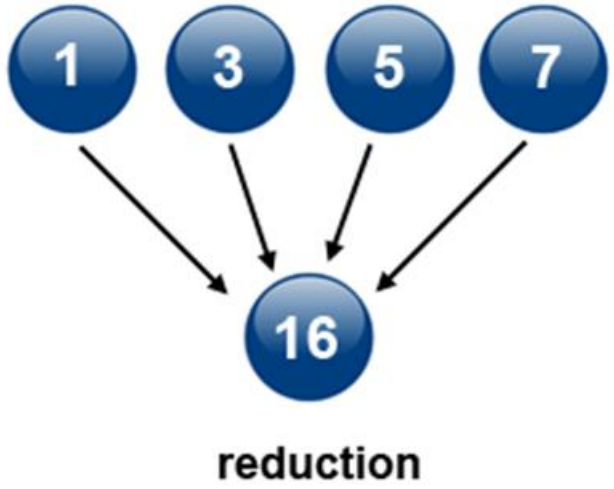
//Each process sends a value, and process 0 gathers these values into an array.

Possible outputs

mpiexec -n 4 MPIExample.exe
Process 0 gathered: 100 101 102 103

MPI_Reduce (Reduce)

MPI_Reduce performs a reduction operation (like sum, max, min, etc.) on data from all processes and stores the result in the root process.




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

sendbuf: Starting address of the send buffer (on each process).
recvbuf: Starting address of the receive buffer (on the root process).
count: Number of elements in the send buffer.
datatype: Data type of buffer elements.
op: The operation to be applied (e.g., MPI_SUM, MPI_MAX).
root: Rank of the root process.
comm: Communicator.

Example: Each process contributes a number, and process 0 calculates the sum of all the numbers.

```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);

    int send_data = rank + 1;  // Each process contributes a number
    int recv_data;

    // Reduce the values from all processes by summing them
    MPI_Reduce(&send_data, &recv_data, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Sum of ranks: %d\n", recv_data);  // Process 0 prints the sum
    }
    MPI_Finalize();
    return 0;
}
```

In this example:
Each process sends a number (its rank plus 1), and process 0 receives the sum of all these numbers.

Possible outputs

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 8 MPIExample.exe
Sum of ranks: 36

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 4 MPIExample.exe
Sum of ranks: 10

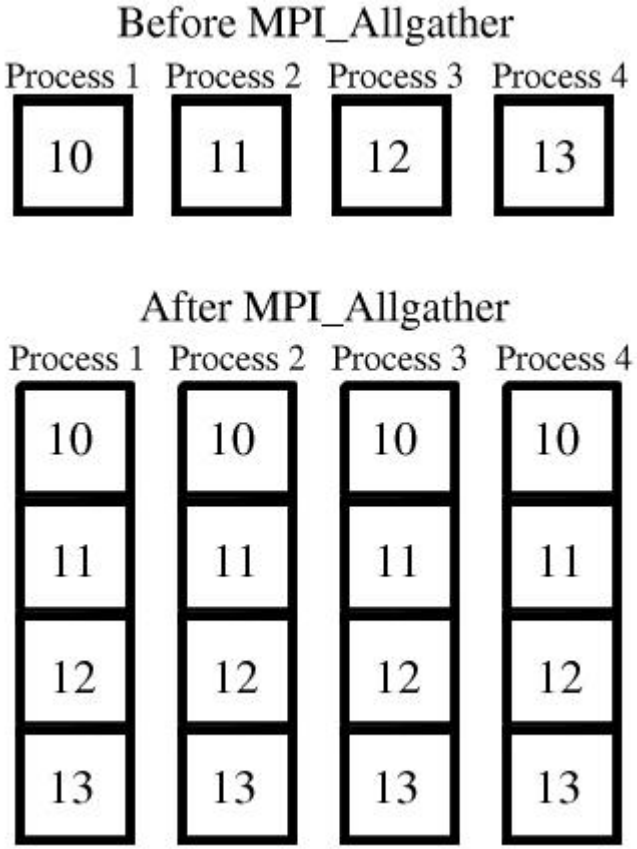
Summary of MPI Collective Communication:

Collective communication functions are blocking, meaning that all processes must call the function and will wait until every process reaches the communication point.

Function	Description
MPI_Bcast	Broadcast data from one process to all processes.
MPI_Scatter	Distribute distinct parts of data from one process to all processes.
MPI_Gather	Collect data from all processes to one process.
MPI_Reduce	Combine data from all processes using a reduction operation (e.g., sum, max).

MPI_Allgather

MPI_Allgather is similar to MPI_Gather, but instead of gathering the data at a root process, the gathered data is distributed to all processes. Every process collects data from all other processes.



MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);

sendbuf	starting address of send buffer (choice)
sendcount	number of elements in send buffer (integer)
sendtype	data type of send buffer elements (handle)
recvbuf	address of receive buffer (choice)
recvcount	number of elements received from any process (integer)
recvtype	data type of receive buffer elements (handle)
comm	communicator (handle)

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

int main(int argc, char** argv) {
    int rank, size, send_data, recv_data[10];

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

    send_data = rank + 1;  // Each process will send its rank+1
    // Allgather: each process sends its value and receives values from all other processes
    MPI_Allgather(&send_data, 1, MPI_INT, recv_data, 1, MPI_INT, MPI_COMM_WORLD);

    printf("Process %d received data: ", rank);
    for (int i = 0; i < size; i++) {
        printf("%d ", recv_data[i]);
    }
    printf("\n");

    MPI_Finalize();
    return 0;
}
```

Explanation:

- Each process sends a value (rank + 1), and every process receives the gathered values from all processes.
- For a communicator of size n, each process ends up with an array of size n filled with data from all processes.

Possible outputs

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 4 MPIExample.exe

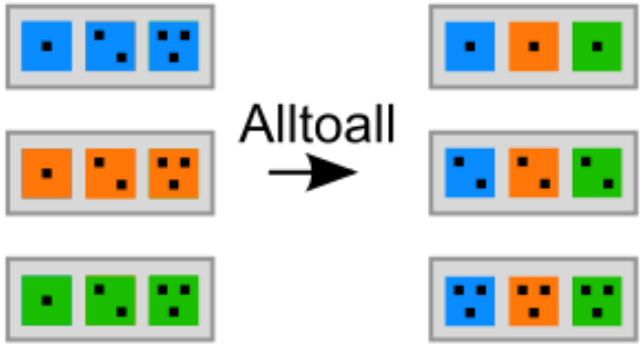
Process 3 received data: 1 2 3 4
Process 2 received data: 1 2 3 4
Process 1 received data: 1 2 3 4
Process 0 received data: 1 2 3 4

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 8 MPIExample.exe

Process 2 received data: 1 2 3 4 5 6 7 8
Process 7 received data: 1 2 3 4 5 6 7 8
Process 0 received data: 1 2 3 4 5 6 7 8
Process 4 received data: 1 2 3 4 5 6 7 8
Process 6 received data: 1 2 3 4 5 6 7 8
Process 5 received data: 1 2 3 4 5 6 7 8
Process 3 received data: 1 2 3 4 5 6 7 8
Process 1 received data: 1 2 3 4 5 6 7 8

MPI_Alltoall

MPI_Alltoall is a powerful collective communication routine where each process sends data to every other process and receives data from every other process. Each process sends a unique value to every other process.

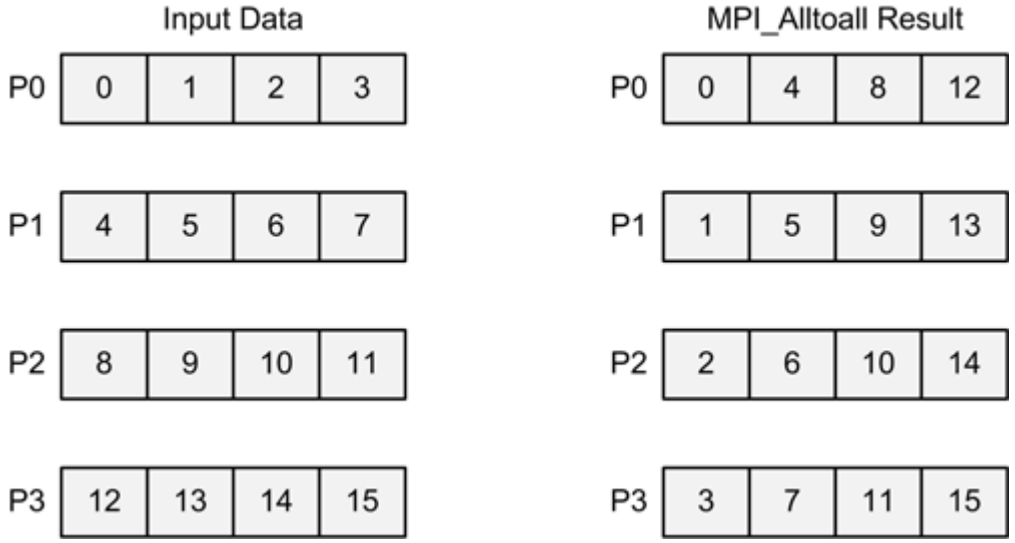


MPI_Alltoall(const void* src_buff,

int src_count, MPI_Datatype src_type,

void* dst_buff, int dst_count,

MPI_Datatype dst_type, MPI_Comm comm**)**



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

int main(int argc, char** argv) {
    int rank, size, send_data[10], recv_data[10];

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

    // Initialize the send_data array, where each process sends data to every other process
    for (int i = 0; i < size; i++) {
        send_data[i] = rank * 10 + i;  // Each process will send different values to other processes
    }

    // Alltoall: each process sends a value to every other process
    MPI_Alltoall(send_data, 1, MPI_INT, recv_data, 1, MPI_INT, MPI_COMM_WORLD);

    printf("Process %d received data: ", rank);
    for (int i = 0; i < size; i++) {
        printf("%d ", recv_data[i]);
    }
    printf("\n");

    MPI_Finalize();
    return 0;
}
```

Explanation:

- Each process sends one value to every other process and receives one value from each process.
- If there are n processes, each process sends an array of size n and receives an array of size n. Each element in the send array is destined for a specific process.

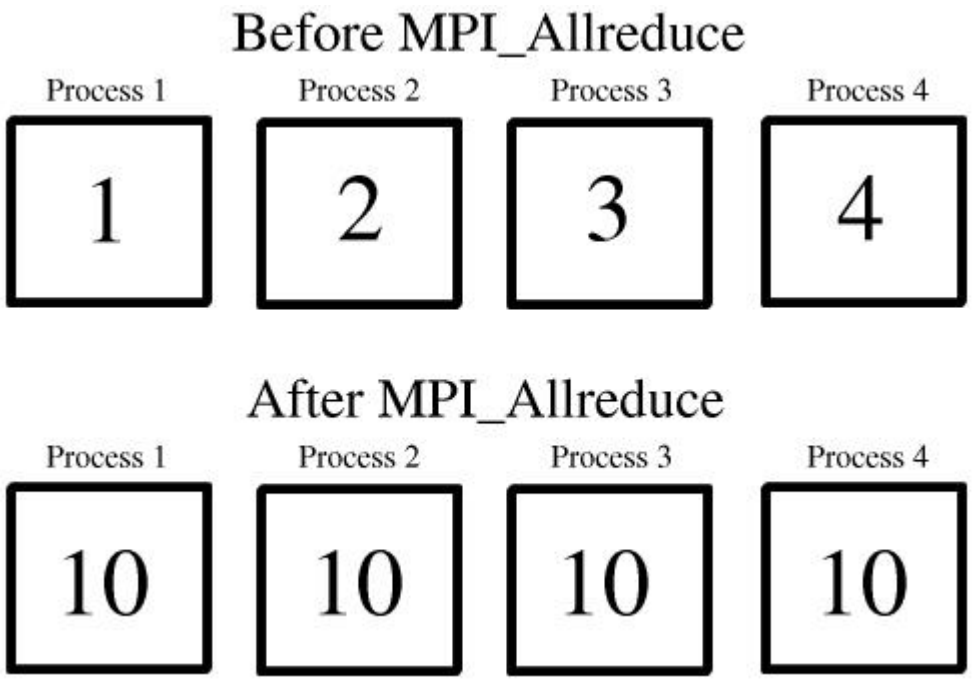
Possible outputs

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 8 MPIExample.exe

Process 6 received data: 6 16 26 36 46 56 66 76
Process 4 received data: 4 14 24 34 44 54 64 74
Process 1 received data: 1 11 21 31 41 51 61 71
Process 3 received data: 3 13 23 33 43 53 63 73
Process 2 received data: 2 12 22 32 42 52 62 72
Process 5 received data: 5 15 25 35 45 55 65 75
Process 0 received data: 0 10 20 30 40 50 60 70
Process 7 received data: 7 17 27 37 47 57 67 77

MPI_Allreduce

MPI_Allreduce is similar to MPI_Reduce, but instead of storing the result in a root process, the result is made available to all processes. It performs a reduction operation (such as sum, max, min, etc.) across all processes and distributes the result to all processes.



MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);

sendbuf	address of send buffer (choice)
recvbuf	starting address of receive buffer (choice)
count	number of elements in send buffer (integer)
datatype	data type of elements in send buffer (handle)
op	operation (handle)
comm	communicator (handle)

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

int main(int argc, char** argv) {
    int rank, size, send_data, result;

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

    send_data = rank + 1;  // Each process has a unique value

    // Allreduce: all processes sum their data and each receives the result
    MPI_Allreduce(&send_data, &result, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    printf("Process %d received result of sum: %d\n", rank, result);

    MPI_Finalize();
    return 0;
}
```

Explanation:

- Each process provides its local data (rank + 1 in this case).
- MPI_Allreduce applies the reduction operation (sum) to the data from all processes and stores the result (in this case, the sum) in each process.
- After execution, all processes have the sum of the values from all processes.

Possible outputs

E:\MPIExamples\MPIExample\x64\Debug>mpiexec -n 8 MPIExample.exe

Process 0 received result of sum: 36
Process 1 received result of sum: 36
Process 7 received result of sum: 36
Process 6 received result of sum: 36
Process 5 received result of sum: 36
Process 4 received result of sum: 36
Process 3 received result of sum: 36
Process 2 received result of sum: 36

Summary of Functions

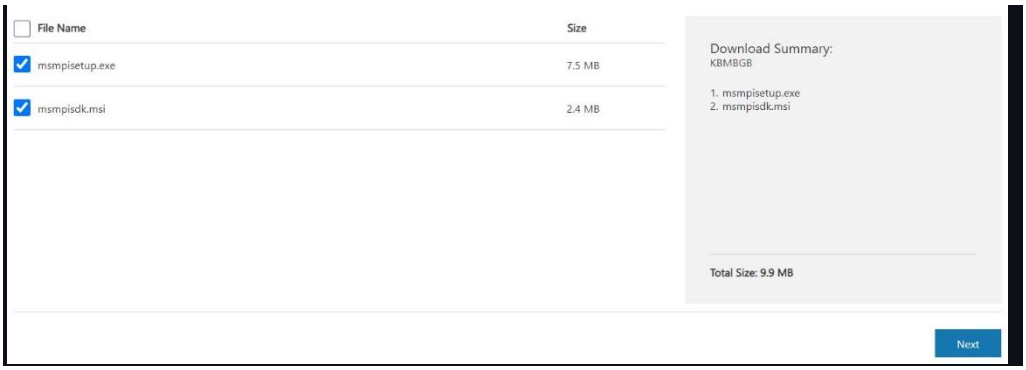
These functions provide efficient mechanisms for collective data exchange and processing across multiple processes in parallel applications.

Function	Description
MPI_Allgather	Every process sends its data to all other processes and receives the data from all other processes.
MPI_Alltoall	Each process sends a unique data element to every other process and receives unique data from all other processes.
MPI_Allreduce	Performs a reduction (like sum, max, etc.) on data from all processes and distributes the result to all processes.

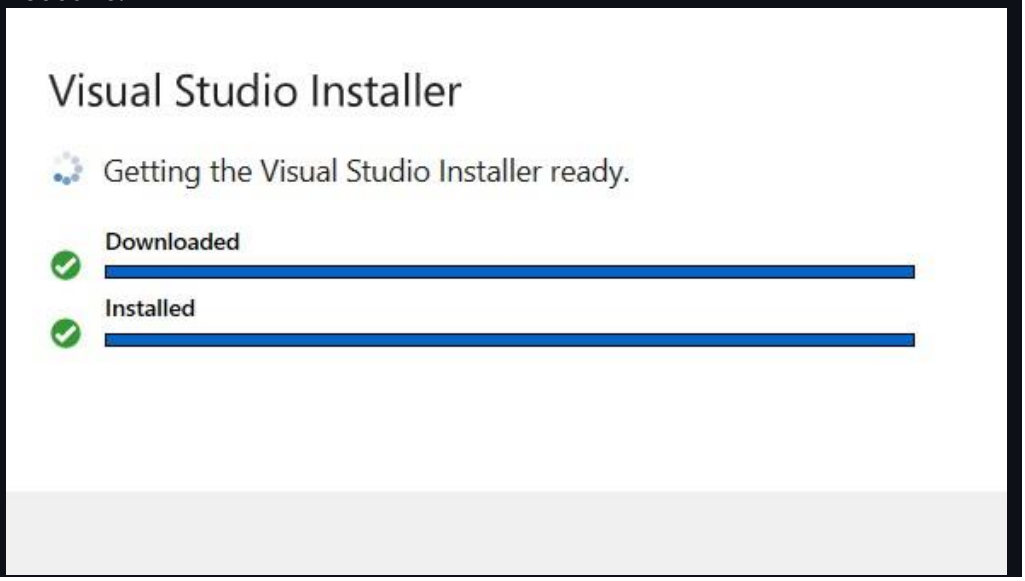
Message Passing Interface (MPI):

Steps:

1. Download Visual Studio from the following link :
    - Visual Studio Community 2022
2. Download useful files for MPI from the following link:
    - Microsoft MPI v10.0
    - Select both files and click on `next` button



3. After downloading install `msmpisetup.exe` application:
    - Leave all as default and click on next and once the installation is done click on finish
4. Run `msmpisdk.msi` installer leaving all as default
5. Now run the `Visual Studio installer` file which was downloaded above.
    - Click `Continue` to accept the `terms and conditions`.
    - Once the downloading and installation is done, you are required to sign in using your `Microsoft Acoount`.

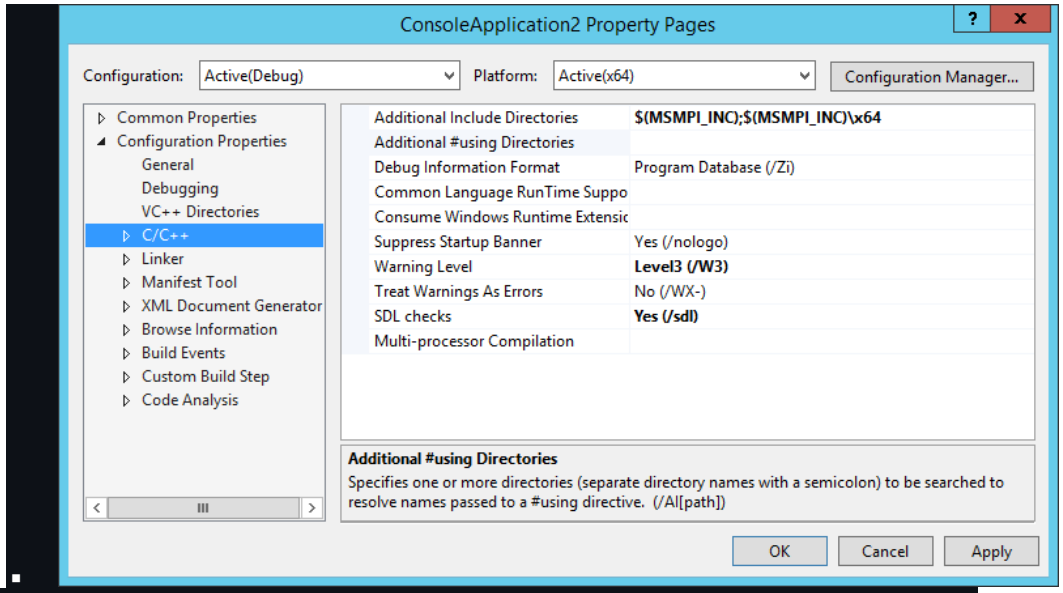


- After successful signing in, you will be directed to choose the required workloads. Here we need to select the required workloads.
- Check the box of required workloads and Click on `Install/Modity` by leaving the option as `Install while downloading`.
- In my case `Desktop Development for C++` is enough.
- It requires nearly 3GB of data to finish whole installation process.
- Once the installation is done `reboot the application`.

6. Now after getting restarted, Click on `Create a new project` and then
   - Select `C++` in `All Programs` drop-down.
   - Now select `Console App` as shown below and click `next`

   - Now choose your `Project Name` and click `Create`.
   - Call MPI header file: `#include "mpi.h"` as shown below:

7. Configuration: After creating project:
   - Head to **Project**->**Properties** and go to C/C++ and add **`$(MSMPI_INC);$(MSMPI_INC)\x64`** as shown below:



- Now go to **Linker**->**All Options** and click scroll to find adding Additional Dependencies and add `msmpi.lib` and press `Apply`
- Now in the additional library directories row add `$(MSMPI_LIB64)`.