Message Passing Interface (MPI)

- MPI is a **standardized communication protocol** used to allow multiple processes (often on different machines) to communicate in a distributed or parallel system.
- It's widely used in **high-performance computing** to coordinate tasks across many processors.
- MPI works, how processes send and receive messages, and how MPI supports both **point-to-point** and **collective communication** patterns

Why It Matters

Understanding MPI is important if you're studying:

- **Distributed Systems**
- **Parallel Computing**
- **High-Performance Computing Applications**

What MPI Is

- A **standardized communication library** used to write programs that run across multiple processors or computers. MPI helps processes exchange data through *messages*.

Why MPI Matters

- MPI is essential in **high-performance computing** and parallel applications where tasks must coordinate and share results reliably and efficiently.

Core Topics Typically Explained

- **Initialization and finalization** — how MPI programs start (`MPI_Init`) and end (`MPI_Finalize`).
- **Point-to-point communication** — sending and receiving data between two processes.
- **Collective communication** — operations involving multiple processes, like broadcasting or gathering data
- **Ranks and communicators** — identifying and grouping processes in an MPI program.

Why It's Useful

MPI is widely used in real systems where many computers work together — such as weather simulations, scientific research clusters, or any application requiring **parallel computation across nodes**.

📌 What is MPI (Message Passing Interface)?

MPI stands for *Message Passing Interface*, a **standardized programming model** used for communication between processes running on *different processors or machines*, especially in **parallel and high-performance computing**.

□ Why MPI Matters

- **Parallel Computing:** It lets you distribute work across many CPUs or nodes so they can compute parts of a job simultaneously, dramatically speeding up large computations.
- **Distributed Memory:** Each process has its *own memory*—they don't automatically share data like threads do, so MPI lets them explicitly *send* and *receive* information.
- **Industry Standard:** MPI is the de-facto choice for scientific simulations, weather models, physics experiments, and other compute-intensive tasks on clusters and supercomputers.

📌 Core Concepts in MPI

Here are typical concepts that an MPI lecture covers:

🔹 *Processes and Ranks*

- Every process participating in an MPI program gets a unique **rank**.
- The set of all processes is called a **communicator** (usually `MPI_COMM_WORLD` for all processes).

🔹 *Initialization & Finalization*

- You start an MPI program with `MPI_Init(...)` and end it with `MPI_Finalize()`.
- This sets up and shuts down the MPI environment.

🔹 *Point-to-Point Communication*

- `MPI_Send(...)` and `MPI_Recv(...)` let one process send data to another.
- This is how explicit message passing works.

🔹 *Collective Communication*

- Operations that involve all processes (e.g., `MPI_Bcast` to broadcast data to all, `MPI_Reduce` to combine results).

📌 Example Usage

In a *simple parallel program*:

1. Every process starts the program (e.g., 4 processes).
2. Each process computes part of a task.
3. Processes exchange intermediate results via MPI messages.
4. One process (often rank 0) collects results and finalizes the computation.

📌 How MPI Is Used in Practice

- **Supercomputers & Clusters:** Run large simulations by distributing calculations.
- **Scientific Research:** Physics, chemistry, biology experiments that need parallel computation.
- **High-Performance Libraries:** Many advanced numerical and simulation libraries rely on MPI internally.

📌 Message Passing Interface (MPI)

🔹 What is MPI?

Message Passing Interface (MPI) is a **standard programming model** used for **communication between multiple processes** running on **different machines or processors** in a **distributed or parallel system**.

☞ In MPI:

- Each process has its **own memory**
- Processes **do NOT share memory**
- Data is exchanged using **messages**

🔹 Why MPI is Used?

MPI is used when:

- Programs need to run **in parallel**
- Tasks are distributed across **multiple computers**
- High performance is required (speed & scalability)

Real-World Uses

- Weather forecasting
- Scientific simulations
- Machine learning on clusters
- Supercomputers
- Distributed systems research

🔹 MPI vs Shared Memory

Feature	MPI	Shared Memory
Memory	Separate	Shared
Communication	Messages	Variables
Scalability	High	Limited
Used In	Clusters	Multicore CPUs

📌 MPI Architecture (How it Works)

1. Multiple processes start simultaneously
2. Each process gets a **rank (ID)**
3. Processes communicate using **send/receive messages**
4. Program ends together

📌 Core MPI Concepts (VERY IMPORTANT)

1️⃣ Process & Rank

- Every process has a **unique rank**
- Rank starts from **0**
- Rank helps identify sender/receiver

txt Copy code

```
Process 0 → Master
Process 1 → Worker
Process 2 → Worker
```

2 Communicator

A **communicator** defines a group of processes that can communicate.

- Default communicator:

```c
MPI_COMM_WORLD
```

(contains all processes)

---

3 MPI Initialization & Termination

Every MPI program **must** start and end like this:

```c
MPI_Init(&argc, &argv);
// MPI code here
MPI_Finalize();
```

4 Get Rank & Total Processes

```c
int rank, size;

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

- `rank` → ID of current process
- `size` → total number of processes

📌 Point-to-Point Communication

🔷 MPI_Send

Used to **send data**

c Copy code

```
MPI_Send(&data, count, MPI_INT, destination, tag, MPI_COMM_WORLD);
```

🔷 MPI_Recv

Used to **receive data**

c Copy code

```
MPI_Recv(&data, count, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
```

🔻 Example: Simple Send & Receive Program (C)

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

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

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

    if (rank == 0) {
        number = 100;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Process 1 received number %d\n", number);
    }

    MPI_Finalize();
    return 0;
}
```

Explanation

- Process 0 sends `100`
- Process 1 receives `100`

- Communication via message passing

📌 Collective Communication

Collective communication involves **multiple processes at once**.

1️⃣ MPI_Bcast (Broadcast)

One process sends data to **all** processes.

```c
MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

👉 Process 0 → sends
👉 All others → receive

2️⃣ MPI_Reduce

Combines values from all processes.

```c
MPI_Reduce(&send, &recv, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Operations:

- `MPI_SUM`
- `MPI_MAX`
- `MPI_MIN`

3 MPI_Gather

Collects data from all processes to one.

```c
MPI_Gather(&send, 1, MPI_INT, recv, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

📌 Blocking vs Non-Blocking Communication

🔷 Blocking

- Program waits until operation completes

```c
MPI_Send();
MPI_Recv();
```

🔷 Non-Blocking

- Program continues execution

```c
MPI_Isend();
MPI_Irecv();
```

Used for **better performance**.

📌 Advantages of MPI

✓ Highly scalable
✓ Portable
✓ Efficient
✓ Standardized
✓ Works on clusters

📌 Disadvantages of MPI

❌ Complex programming
❌ Manual message handling
❌ Debugging is difficult

📌 MPI Program Execution

MPI programs are executed using:

```bash
mpicc program.c -o program
mpirun -np 4 ./program
```

- `-np 4` → number of processes

📌 Exam-Friendly Definition

MPI is a standardized message-passing system designed to allow multiple processes running on different machines to communicate with each other in a distributed computing environment.

Message Passing Interface (MPI)

→ MPI is a specification designed for parallel application

→ The goal of MPI is to provide a widely used standard for writing message passing programs. The interface attempts to be
- Practical
- Portable
- efficient
- flexible

→ A message passing library specification is:
- message passing model
- not a compiler specification

Message Passing Interface (MPI)

→ MPI is a specification designed for parallel application

→ The goal of MPI is to provide a widely used standard for writing message passing programs. The interface attempts to be
- Practical
- Portable
- efficient
- flexible

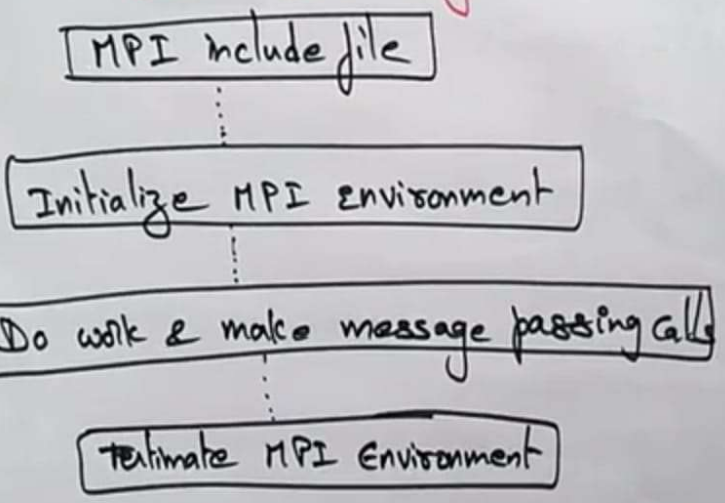
→ A message passing library specification is:
- message passing model
- not a compiler specification

- not a specific Product.

Reasons for using MPI

1. Standardization
2. Portability
3. Performance opportunities
4. Functionality - over 115 routines are defined
5. Availability - variety of implementation are available, both vendor & public domain

General MPI program structure



MPI programming

→ writing MPI Programs

→ compiling & linking ⟹ compile hello.c : mpicc -o hellohello.c

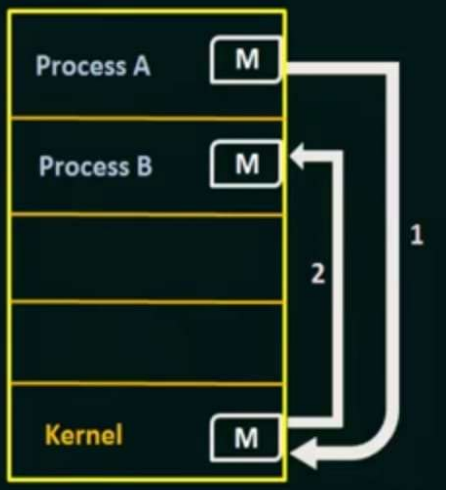
→ Running MPI programs

⇒ Example :- MPI C program (hello.c)

```
#include<mpi.h>   // provides basic MPI definition & types
#include<stdio.h>
int main(int argc, char **argv)
{
    MPI_Init(&argc,&argv);   // MPI_init starts MPI
    Printf("Hello world\n");
    MPI_Finalize();          // MPI_Finalize exit MPI
```

Message-Passing Systems (Part-1)

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.



A message-passing facility provides at least two operations:

- send (message)

and

- receive (message)

Messages sent by a process can be of either fixed or variable size.

FIXED SIZE: The system-level implementation is straightforward.
But makes the task of programming more difficult.

VARIABLE SIZE: Requires a more complex system-level implementation.
But the programming task becomes simpler.

NESO ACADEMY

Message-Passing Systems (Part-2)

If processes P and Q want to communicate, they must send messages to and receive messages from each other.

A communication link must exist between them.

This link can be implemented in a variety of ways. There are several methods for logically implementing a link and the send( ) /receive( ) operations, like:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

There are several issues related with features like:

- Naming
- Synchronization
- Buffering

NESO ACADEMY

Naming

Processes that want to communicate must have a way to refer to each other.
They can use either **direct** or **indirect** communication.

Under direct communication- Each process that wants to communicate must explicitly name the recipient or sender of the communication.

- send (P, message) - Send a message to process P.
- receive (Q, message) - Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

This scheme exhibits **symmetry in addressing**; that is, both the sender process and the receiver process must name the other to communicate

Another variant of Direct Communication- Here, only the sender names the recipient; the recipient is not required to name the sender.

- send (P, message) - Send a message to process P.
- receive (id, message) - Receive a message from any process; *the variable id is set to the name of the process with which communication has taken place.*

This scheme employs **asymmetry in addressing**.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions.
Changing the identifier of a process may necessitate examining all other process definitions.

With indirect communication:

The messages are sent to and received from **mailboxes**, or ports.

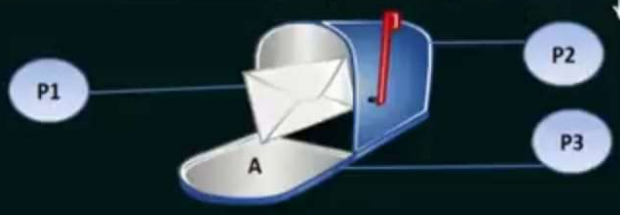
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique identification.
- Two processes can communicate only if the processes have a shared mailbox

  - send (A, message) — Send a message to mailbox A.
  - receive (A, message) — Receive a message from mailbox A.

A communication link in this scheme has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

Now suppose that processes P1, P2, and P3 all share mailbox A



Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. **Which process will receive the message sent by P1?**

Message-Passing Systems (Part-3)

If processes P and Q want to communicate, they must send messages to and receive messages from each other.

A communication link must exist between them.

This link can be implemented in a variety of ways. There are several methods for logically implementing a link and the **send( ) /receive( )** operations, like:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

There are several issues related with features like:

- **Naming**
- **Synchronization**
- **Buffering**

Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. **Which process will receive the message sent by P1?**

The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.
- Allow at most one process at a time to execute a receive () operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, round robin where processes take turns receiving messages). The system may identify the receiver to the sender.

A **mailbox** may be **owned** either by a **process** or by the **operating system.**

Synchronization

Communication between processes takes place through calls to send() and receive () primitives. There are different design options for implementing each primitive.

Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous.**

Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous**.

Blocking send: The sending process is blocked until the message is received by the receiving process or by the mailbox.

Nonblocking send: The sending process sends the message and resumes operation.

Blocking receive: The receiver blocks until a message is available.

Nonblocking receive: The receiver retrieves either a valid message or a null.

Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
Basically, such queues can be implemented in three ways:

Zero capacity: The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

Bounded capacity: The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

NESO ACADEMY

Zero capacity: The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

Bounded capacity: The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

Unbounded capacity: The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

NESO ACADEMY

MPI – MESSAGE PASSING INTERFACE

- MPI is a standard library for message passing used in parallel programming with C and Fortran.
- It defines both the syntax and semantics of core library routines.
- MPI implementations are available on various parallel computing platforms.
- It is possible to write fully-functional message-passing programs by using only the six routines.

MPI – MESSAGE PASSING INTERFACE

- Initializing and Finalizing MPI:
  - `MPI_Init` initializes the MPI environment before any other MPI routines.

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

  - `MPI_Finalize` is called at the end of computation to clean up MPI resources.

    ```
    int MPI_Finalize()
    ```

  - These functions also handle command-line arguments related to MPI.

MPI – MESSAGE PASSING INTERFACE

- MPI_Init also strips off any MPI related command-line arguments.
- All MPI routines, data-types, and constants are prefixed by "MPI_".
- The return code for successful completion is MPI_SUCCESS.

MPI – MESSAGE PASSING INTERFACE

- Querying Information:
  - `MPI_Comm_size` determines the number of processes.

    int MPI_Comm_size(MPI_Comm comm, int *size)
  - `MPI_Comm_rank` identifies the rank (label) of the calling process.

    int MPI_Comm_rank(MPI_Comm comm, int *rank)
  - Ranks are integers ranging from 0 to the communicator's size minus one.

MPI – MESSAGE PASSING INTERFACE

- Communicators:
  - Communicators define sets of processes allowed to communicate.
  - MPI uses variables of type `MPI_Comm` to store information about communication domains.
  - The default communicator is `MPI_COMM_WORLD`, which includes all processes.
  - Communicators are used as arguments to all message transfer MPI routines.
  - A process can belong to many different (possibly overlapping) communication domains.

MPI – MESSAGE PASSING INTERFACE

```
#include <mpi.h>
main(int argc, char *argv[])
{
    int npes, myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("From process %d out of %d, Hello World!\n",
    myrank, npes);
    MPI_Finalize();
}
```

A Simple MPI Program:

- Demonstrates a basic MPI program using `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, and `MPI_Finalize`.
- Shows how to print a message with process rank and total processes.

SENDING AND RECEIVE MESSAGES

- MPI provides `MPI_Send` for sending messages and `MPI_Recv` for receiving them.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

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

SENDING AND RECEIVE MESSAGES

- MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons.
- The datatype MPI_BYTE corresponds to a byte (8 bits) and MPI_PACKED corresponds to a collection of data items that has been created by packing non-contiguous data.
- The message-tag can take values ranging from zero up to the MPI defined constant MPI_TAG_UB.

SENDING AND RECEIVE MESSAGES

- `MPI_Recv` provides a status variable for obtaining information about the received message.
- The `MPI_Status` structure contains fields like source, tag, and error information.
- `MPI_Get_count` retrieves the precise count of received data items.

AVOIDING DEADLOCKS

- Deadlocks can occur when processes block each other with sends and receives.
- Strategies like breaking circular waits can prevent deadlocks.

AVOIDING DEADLOCKS

Consider the following piece of code, in which process i sends a message to process i + 1 (modulo the number of processes) and receives a message from process i - 1 (module the number of processes).

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
       MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
       MPI_COMM_WORLD);
...
```

Once again, we have a deadlock if MPI_Send is blocking.

SIMULTANEOUS MESSAGE EXCHANGE

- MPI offers `MPI_Sendrecv` to exchange messages between processes simultaneously.

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
   MPI_Datatype senddatatype, int dest, int
   sendtag, void *recvbuf, int recvcount,
   MPI_Datatype recvdatatype, int source, int recvtag,
   MPI_Comm comm, MPI_Status *status)
```

- For using the same buffer for both send and receive, `MPI_Sendrecv_replace` is available.

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

MPI: the Message Passing Interface

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran.
- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.
- Vendor implementations of MPI are available on almost all commercial parallel computers.
- It is possible to write fully-functional message-passing programs by using only the six routines.

MPI: the Message Passing Interface

The minimal set of MPI routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

Starting and Terminating the MPI Library

- `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- The prototypes of these two functions are:

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

- `MPI_Init` also strips off any MPI related command-line arguments.
- All MPI routines, data-types, and constants are prefixed by "`MPI_`". The return code for successful completion is `MPI_SUCCESS`.

Communicators

- A communicator defines a *communication domain* - a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.

Querying Information

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.
- The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

Our First MPI Program

```
#include <mpi.h>

main(int argc, char *argv[])
{
    int npes, myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("From process %d out of %d, Hello World!\n",
        myrank, npes);
    MPI_Finalize();
}
```

Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively.
- The calling sequences of these routines are as follows:

```
int MPI_Send(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype
datatype, int source, int tag,          MPI_Comm
comm, MPI_Status *status)
```

- MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons.
- The datatype `MPI_BYTE` corresponds to a byte (8 bits) and `MPI_PACKED` corresponds to a collection of data items that has been created by packing non-contiguous data.
- The message-tag can take values ranging from zero up to the MPI defined constant `MPI_TAG_UB`.

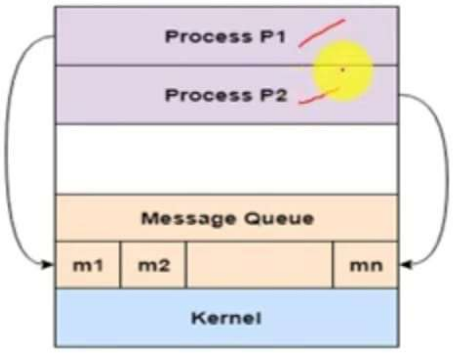
P & D Computing by Aized Amin

Message Passing Model

- In a message-passing model, parallel processes exchange data through passing messages to one another.
- These communications can be asynchronous, *where a message can be sent before the receiver is ready*, or synchronous, *where the receiver must be ready.*
- Message passing model allows multiple processes to read and write data to the message queue without being connected to each other.

Cont..

- Messages are stored on the queue until their recipient retrieves them.
- Message queues are quite useful for inter-process communication and are used by most operating systems.



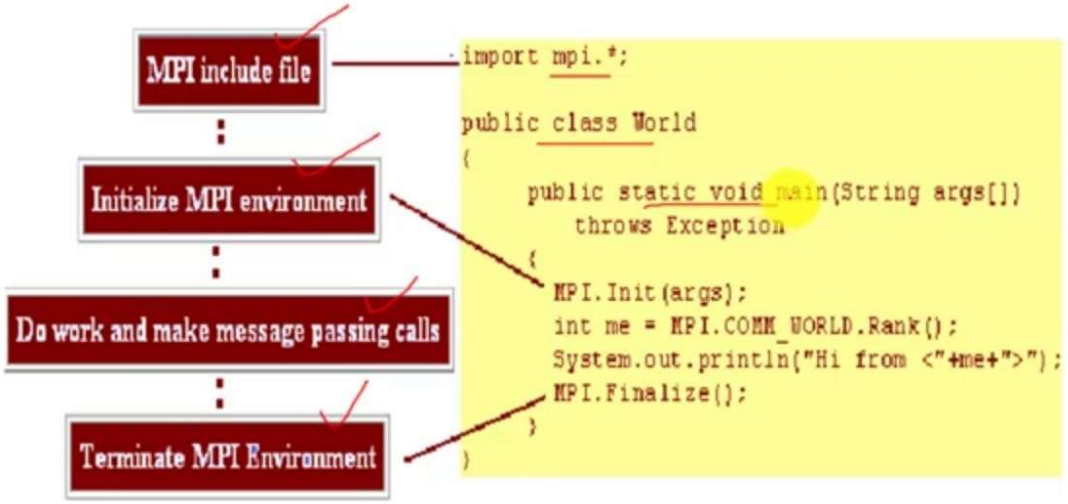
Message Passing Model

Message Passing Interface (MPI)

- MPI is specification design for parallel applications.
- The goal of MPI is to provide widely used standard for writing message passing programs.
- The interface attempt to be:
  - Practical
  - Portable
  - Efficient
  - flexible
- A message passing library specification is:
  - Message passing model
  - Not a compiler specification
  - Not a specific product

P & D Computing by Aized Amin

General MPI Program Structure



P & D Computing by Aized Amin

Reasons for Using MPI

➢The main reasons of using MPI are:

- Standardization
- Portability
- Performance opportunities
- Functionality
- availability

Parallel Virtual Machine (PVM)

- It is a S/W package that allows heterogeneous collection of workstations to function as a single high performance parallel machine (virtual).

P & D Computing by
Aized Amin

PVM Programming Model

- PVM application consist of collection of cooperating tasks
  - Each of which is responsible for some workload of big problem
- PVM is inherently dynamic in nature, and it has rich set of resources control functions.
  - Host can be added or deleted
- Main features of PVM Includes:
  - Load balancing
  - Task migration
  - Fault tolerance
  - efficiency