# Message Passing Interface (MPI) Basics

Usama Bin Amir

Department of Cyber Security,
Air University, Islamabad

# Distributed Systems - Definition

A **distributed system** is a **software system** in which **components** located on **networked computers communicate** and **coordinate** their actions **by passing messages** (to achieve a common goal)

**How to Program Distributed Computers?**

- **Message Passing based Programming Model**

# MPI (Message Passing Interface)?

- **Standardized message passing library specification**
  - for **parallel computers clusters**
  - not a specific product, compiler specification etc.
  - many implementations, **MPICH**, **LAM**, **OpenMPI** …

- **Portable**, with **Fortran** and **C/C++ interfaces**
- ***Real* parallel programming**

# A Brief History - MPI

- **Writing parallel applications** is **fun**!!

- **Initially,** it **was a difficult**:
  - **No single standard**
  - **Various implementations** (with **different features**)

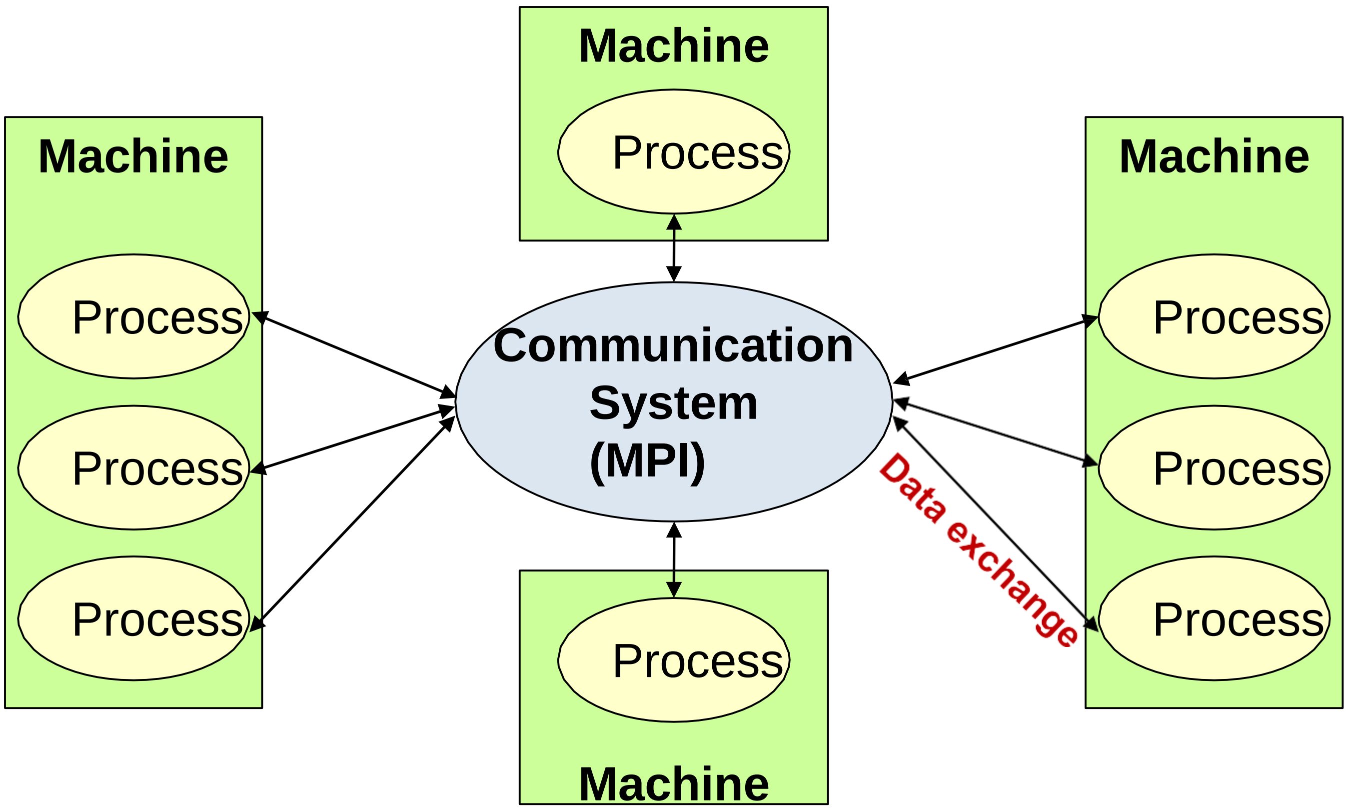
- **Solution**: a **MPI standard was defined**
  - **Supporting same features** and **semantics across implementations**
  - By **1994**, **a complete interface** and **standard** was **defined** (**MPI-1**)
- **Result**: **Portable Parallel Programs**

# The Message-Passing Model

**Two major requirements**:

1. **Creating separate processes** for **execution** on **<u>different computers</u>**

2. **Method** of **sending** and **receiving** **messages**



# The Message-Passing Model

- A ***process*** is (**traditionally**) a **program counter** and **address space**

- **Processes** may have **multiple *threads***
  - **program counters** and **associated stacks**
  - **sharing** a **single address space**

- **MPI** is for **communication among processes**
  - ➢ **separate address spaces**

- **Inter-process communication** consists of:
  - **Synchronization**
  - **Data movement** from **one process**'s **address space** to another's

# Types of Parallel Computing Models

1. **Data Parallel**
   —**Same instructions** are **carried** out **simultaneously** on **multiple data items** (e.g., **SIMD**)

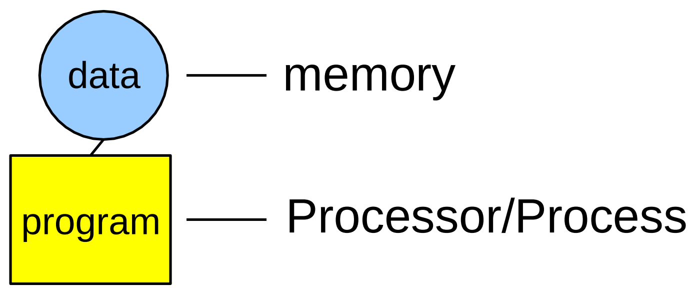
2. **Task Parallel**
   —**Different instructions** on **different data** (e.g., **MIMD**)

3. **SPMD (Single Program, Multiple Data)**
   —**Not synchronized** at **individual instruction level** (e.g., **MIMD** style execution)

→ **MPI** is for **MIMD/SPMD** type of **parallelism**

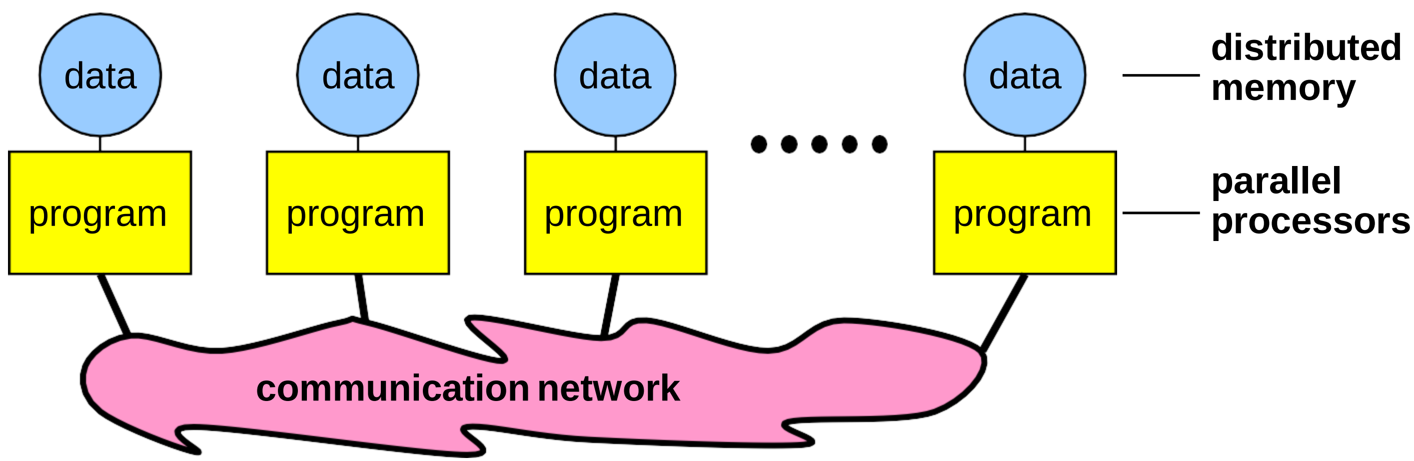
# The Message-Passing Programming Paradigm

## Sequential Programming Paradigm:




**A processor may run many processes**

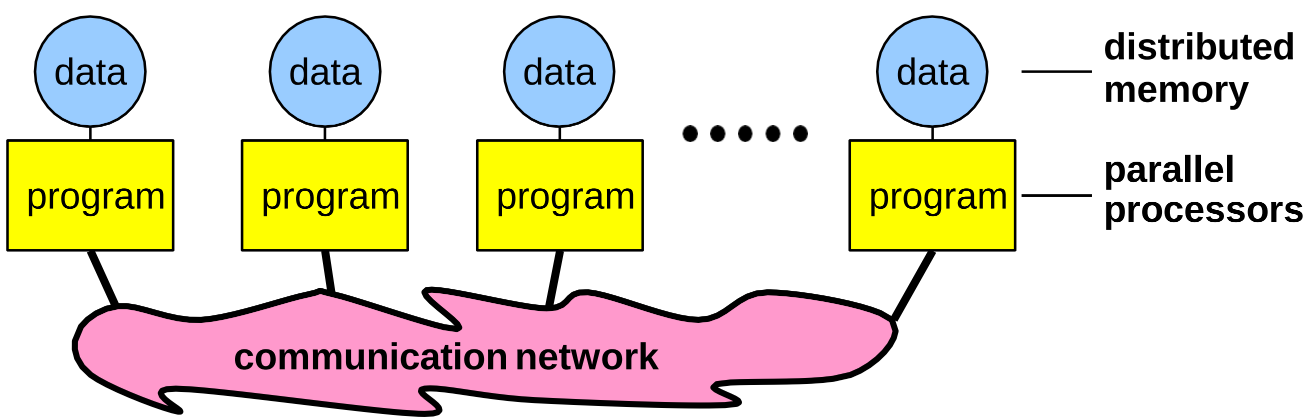
## Message-Passing Programming Paradigm



# The Message-Passing Programming Paradigm

- A **process** is a **program performing** a **task** on a **processor**

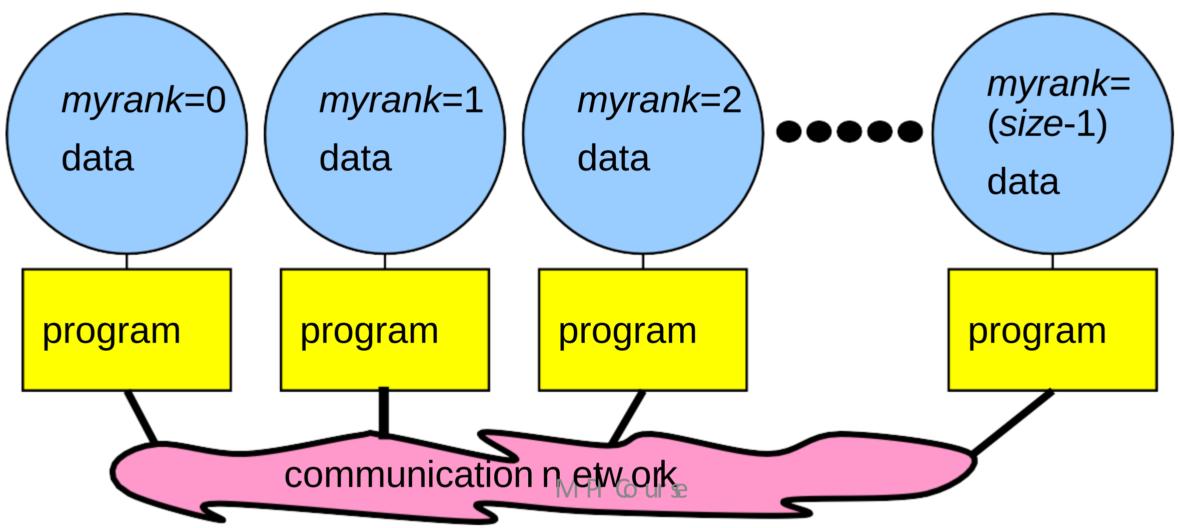
- Each **processor/process runs** a **instance/copy** of the **same program:**
  - the **variables** of **each sub-program** have:
    - the **same name** but **different locations** (**distributed memory**) and **different data**
    - i.e., **all variables** are **local to a process**
  - **communicate** via ***message passing***



# Data and Work Distribution

- To **communicate** together **MPI-processes** **need identifiers**: **rank** **=** **identifying number**

- **Processes** are **identified** using ***rank***

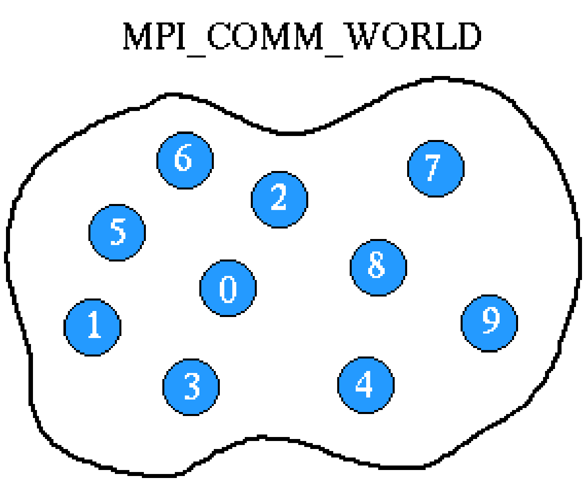


# MPI Fundamentals

- A **communicator** **defines** a **group of processes** that have the **ability** to **communicate** with **one another**

- **<u>In a group</u>**, each **processes** is **assigned** a **unique** ***rank***

- Use **rank** to **explicitly communicate** with **one another**



# Communicators

**Communicators** do **not need to contain all processes** in the system

**Every process in a communicator has an ID called as “rank”**


**The same process might have different ranks in different communicators**

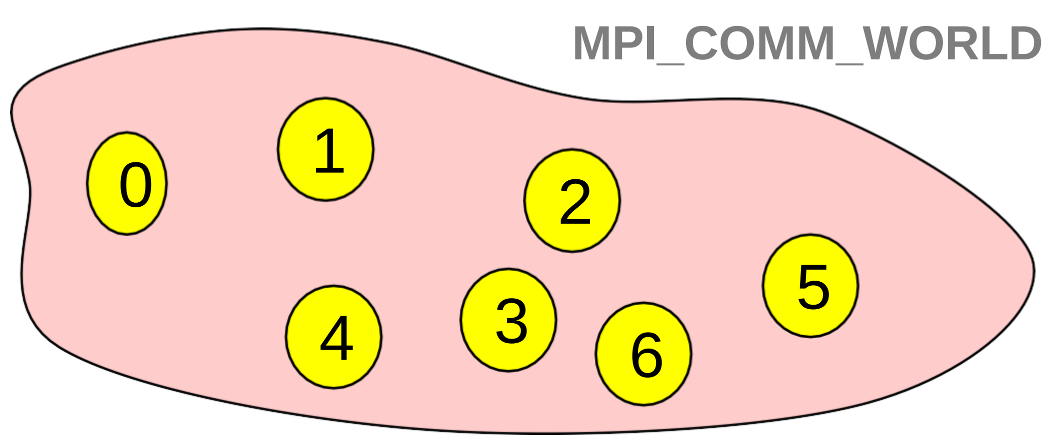
When you **start an MPI program**, there is **one predefined communicator** **`MPI_COMM_WORLD`**

**Simple programs typically only use the predefined communicator**

**`MPI_COMM_WORLD`**

# Communicator MPI_COMM_WORLD

- **All processes** of an **MPI program** are **members** of the **default communicator MPI_COMM_WORLD**
- **MPI_COMM_WORLD** is **predefined**
- **Additional communicators can** also be **defined**
- Each **process** has its **own rank** in a **communicator**:
  - **starting** with **0**
  - **Ending** with (**size-1**)



# How big is the MPI library?

- **Huge** (**125 Functions**) !!
- **Good news** – you can write **useful MPI programs** only using **6 basic functions**

| | |
|---|---|
| `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. |

# MPI – Start & Termination

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

- **initialization**: must **call** this **prior** to **other MPI routines** **(by main thread)**
- **initializes MPI environment**

**int MPI_Finalize( )**

- **Must call at the end of the computation (by the main thread)**
- **performs various clean-up tasks to terminate MPI environment**

- **Return codes** (for both ***MPI_Init*** & ***MPI_Finalize***)
  - **MPI_SUCCESS**
  - **MPI_ERROR**

# Communicators

**Communicator: MPI_Comm**

- Group of **processes** that could **communicate** with **one another**
- **Communication domains could overlap**
- **A process may be part of multiple communicators**
- **MPI_COMM_WORLD: root communicator (all the processes)**

# Communicators

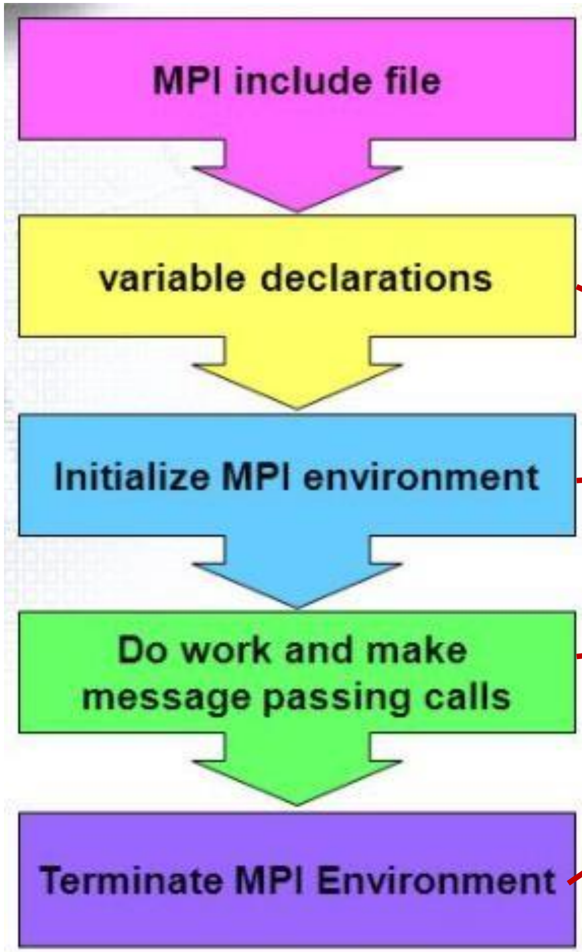
**int MPI_Comm_size(MPI_Comm comm, int *size)**

- **Determine the number of processes (in a particular communicator)**

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

- **Index of the calling process (in a particular communicator)**
- **0 ≤ rank < communicator size**

# MPI Program – A Generic Structure




```c
#include <mpi.h>
void main (int argc, char *argv[])
{
int np, rank, ierr;
ierr = MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&np);
/*      Do Some Works          */
ierr = MPI_Finalize();
}
```

*Credits: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/ MPI Program Structure*

# A minimal MPI Program

Demo:
hello.c

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
      MPI_Init(&argc, &argv);
      printf("Hello PDC Class!\n");
      MPI_Finalize();
      return 0;
}
```

MPI header file

MPI **environment initialization**, all of MPI's **global** and **internal variables** are constructed. For example, a **communicator** is **formed** around all of the **processes** that were **spawned**, and **unique ranks** are **assigned** to each process.

MPI_Comm_size returns the **size** of a **communicator**. Here, MPI_COMM_WORLD encloses all of the processes, so this call should return the amount of processes that were requested for the job.

MPI_Comm_rank returns the rank of a process in a communicator. Ranks are incremental starting from zero and are primarily used for identification purposes during send/receive.

MPI_Get_processor_name obtains the actual name of the processor on which the process is executing.

**Demo: processorName.c**

clean up the MPI environnent

# Compiling MPICH Program

- **Regular C applications:**

```
gcc hello_world.c -o hello_world
```

- **MPI based C applications**

```
mpicc mpi_hello_world.c -o mpi_hello_world
```

# Running MPICH Program

- **Regular C applications**

```
./hello_world
```

- **MPI based C applications** (running with 16 processes)

```
mpiexec -n 16 ./mpi_hello_world
```

# Running MPICH Program on a Cluster

- **On Clusters**, you will **have to set up a host file** (named machinefile in our earlier demo)

- The **host file** contains **names** of all of the **nodes** on which your **MPI job will execute**

- Example (*machinefile* contents):

  *slave1:4 # this will spawn 4 processes on slave1*

  *master:2  # this will spawn 2 processes on master*

- and executed it using

  **mpiexec -n 6 -f machinefile ./mpi_hello**

# Running MPICH Program on a Cluster

- As shown, the **MPI program** was **launched across all of the hosts** in ***machinefile***

- **Each process** was **assigned** a **unique rank**, which was printed off **along with** the **process name**.

- The **output** of the **processes** is in an **arbitrary order** since there is **no synchronization involved before printing**

# Congratulations !!

- You now have a **basic understanding** about **how MPI works**

- **Even better, you already have a cluster** and you can write parallel programs !!

# Point-to-Point Communication

# Data Messages

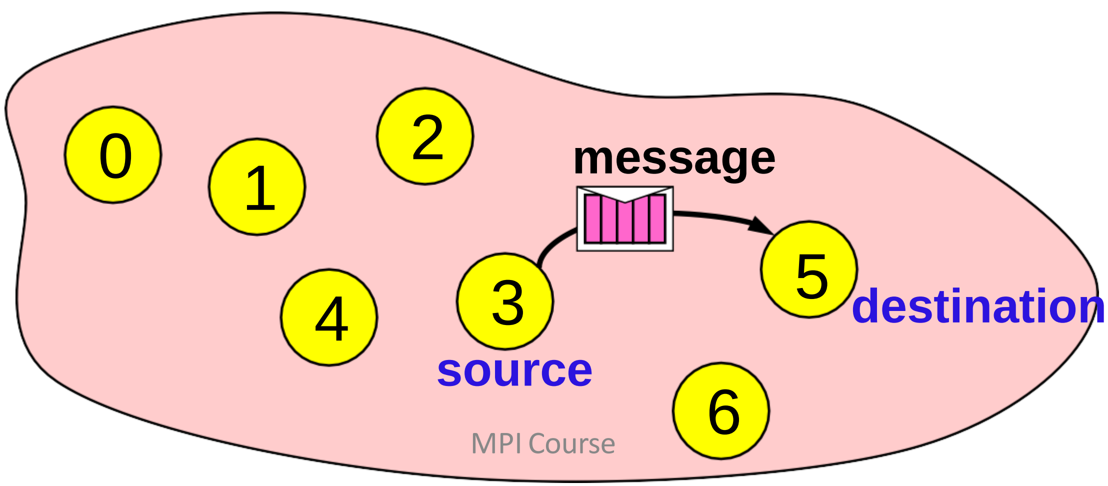
- A **message contains** a **number of elements** of some **particular datatype**

**Example: message with 5 integers**

| | | | | |
|---|---|---|---|---|
| 2345 | 654 | 96574 | -12 | 7676 |

# Point-to-Point Communication

- **Communication between two processes**
- **Source** process **sends message** to **destination** process
- **Communication** takes place **within a communicator**, e.g., MPI_COMM_WORLD
- **Processes** are **identified** by their **ranks** in the communicator

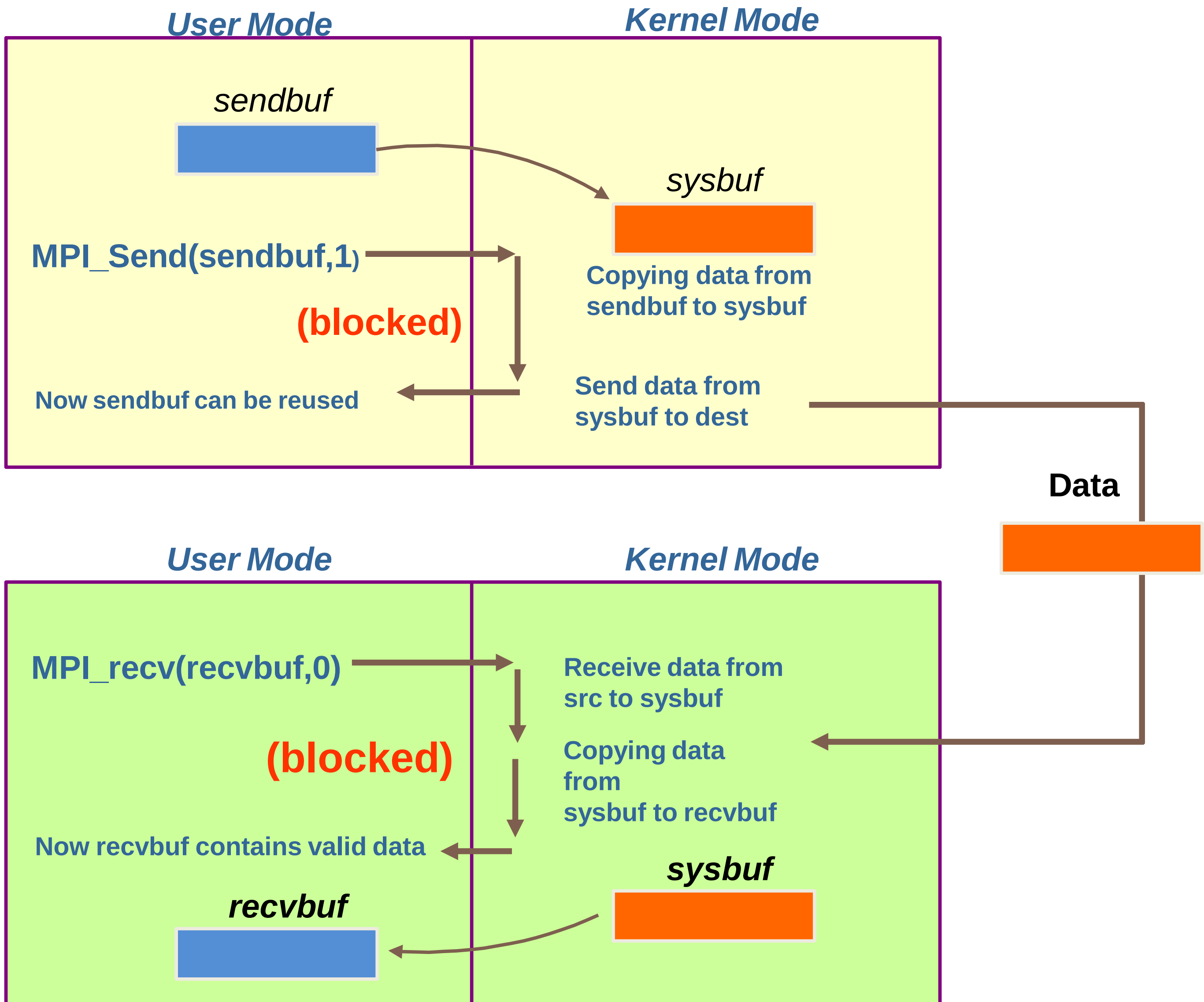


# Point to Point Communication

- **Communication** is **done** using **send** and **receive** among processes:
  - To **send** a **message**, sender **provides** the **rank** of the **process** **and** a **unique *tag*** to **identify** the **message**
  - The **receiver** can then **receive** a **message** with a **given tag** (or it may not even care about the tag), and then **handle** the **data accordingly**
  - **Two basic (and simple) functions**, **MPI_Send** and **MPI_Recv**

# MPI_Send & MPI_Recv

**Sending process waits** until all data are **transferred** to the **system buffer**




**Receiving process waits** until all **data** are **transferred** from the **system buffer** to the **receive buffer**

# Data Communication in MPI

- **Communication requires the following information:**
  - **Sender has to know**:
    - **Whom to send** the **data** to (***receiver's process rank***)
    - What **kind of data** to **send** (***100 integers*** or ***200 characters***,..)
    - A **user-defined** "**tag**" (**distinguish different messages**)
  - **Receiver "might" have to know:**
    - **Who** is **sending** Or **wildcard**: **MPI_ANY_SOURCE** (**meaning anyone can send**)
    - **kind of data is being received** (may be **partial info, e.g., upper bound**)
    - **Message "tag"** Or **wildcard**: **MPI_ANY_TAG** (**any message**)

# MPI_Send

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

*data*: **pointer** to **data**

*count*: number of **elements to send**

*type*: **data type** of data

*dest*: **destination process (rank)**

*tag*: identifying **tag**

*comm*: **communicator**

- **Blocking operation**
- When `MPI_Send` **returns** the **message is sent** and the **data buffer can be reused** (the **message may not have been received** by the **target process** yet)

# MPI_Recv

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

*data*: **pointer** to **data**

*count*: **number** of **elements** to be **received (upper bound)**

*type*: **data type**

*source*: **source process** of the **message**

*tag*: identifying **tag**

*comm*: **communicator**

*status*: i.e., ***sender***, ***tag***, and ***message size***

- When `MPI_Recv` **returns** the **message** has been **received**
- **Waits** until a **matching** (on `source`, `tag`, `comm`) **message** is **received**

# Elementary MPI datatypes

**Similar** to **C datatypes, portable MPI datatypes**

**int → MPI_INT**

**double → MPI_DOUBLE**

**char → MPI_CHAR**

- **Complex datatypes** also **possible**:
  - E.g., a **structure datatype** that **comprises** of other **datatypes** → a **char**, an **int** and a **double**

# Elementary MPI datatypes

| MPI data type | C data type |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | 8 bits |
| MPI_PACKED | packed sequence of bytes |

# Simple Communication in MPI

Demo: P2P.c

Process rank or id

Message tag

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv)
{
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0){
        char sdata[] = "Hello PDC";
        MPI_Send(sdata, 9, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
    else if (rank == 1) {
       char rdata[]="";
       MPI_Recv(rdata,9,MPI_CHAR,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
       printf("\nI am a slave,received %s message from master\n", rdata);
     }

    MPI_Finalize();
    return 0;
}
```

# The MPI_Status structure

- **Information** is **returned** from **MPI_RECV** in ***status***

```
typedef struct MPI_Status {
   int MPI_SOURCE;
   int MPI_TAG;
   int MPI_ERROR;
 };
```

- **For the messages received using wildcards (MPI_ANY_TAG, MPI_ANY_SOURCE)**
  - **Receiver can check actual "Source" and "Tag" of the message**
  - **MPI_STATUS_IGNORE** can be **used** <u>**if we don't need any**</u> additional <u>**information**</u>

# The MPI_Status structure

```
int MPI_Get_count(MPI_Status* status,
                              MPI_Datatype type,
                               int* count)
```

- In **MPI_Get_count**, the user **passes** the **MPI_Status structure**, the **datatype** of the **message**, and **<u>count</u>** is **<u>returned</u>**

- ***count variable*** is the **total number** of **elements** that were **received**

**Demo: P2PStatus.c**

```c
const int MAX_NUMBERS = 100;
int numbers[MAX_NUMBERS];
int number_amount;
if (world_rank == 0) {
    // Pick a random amont of integers to send to process one
    srand(time(NULL));
    number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

    // Send the amount of integers to process one
    MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("0 sent %d numbers to 1\n", number_amount);
} else if (world_rank == 1) {
    MPI_Status status;
    // Receive at most MAX_NUMBERS from process zero
    MPI_Recv(numbers, MAX_NUMBERS, MPI_INT, 0, 0, MPI_COMM_WORLD,
             &status);

    // After receiving the message, check the status to determine
    // how many numbers were actually received
    MPI_Get_count(&status, MPI_INT, &number_amount);

    // Print off the amount of numbers, and also print additional
    // information in the status object
    printf("1 received %d numbers from 0. Message source = %d, "
           "tag = %d\n",
           number_amount, status.MPI_SOURCE, status.MPI_TAG);
}
```

# Summary (Blocking Send/Recv)

**MPI_SEND** **<u>does not return</u> until buffer is empty (available for reuse)**

**MPI_RECV** **<u>does not return</u> until buffer is full (available for use)**

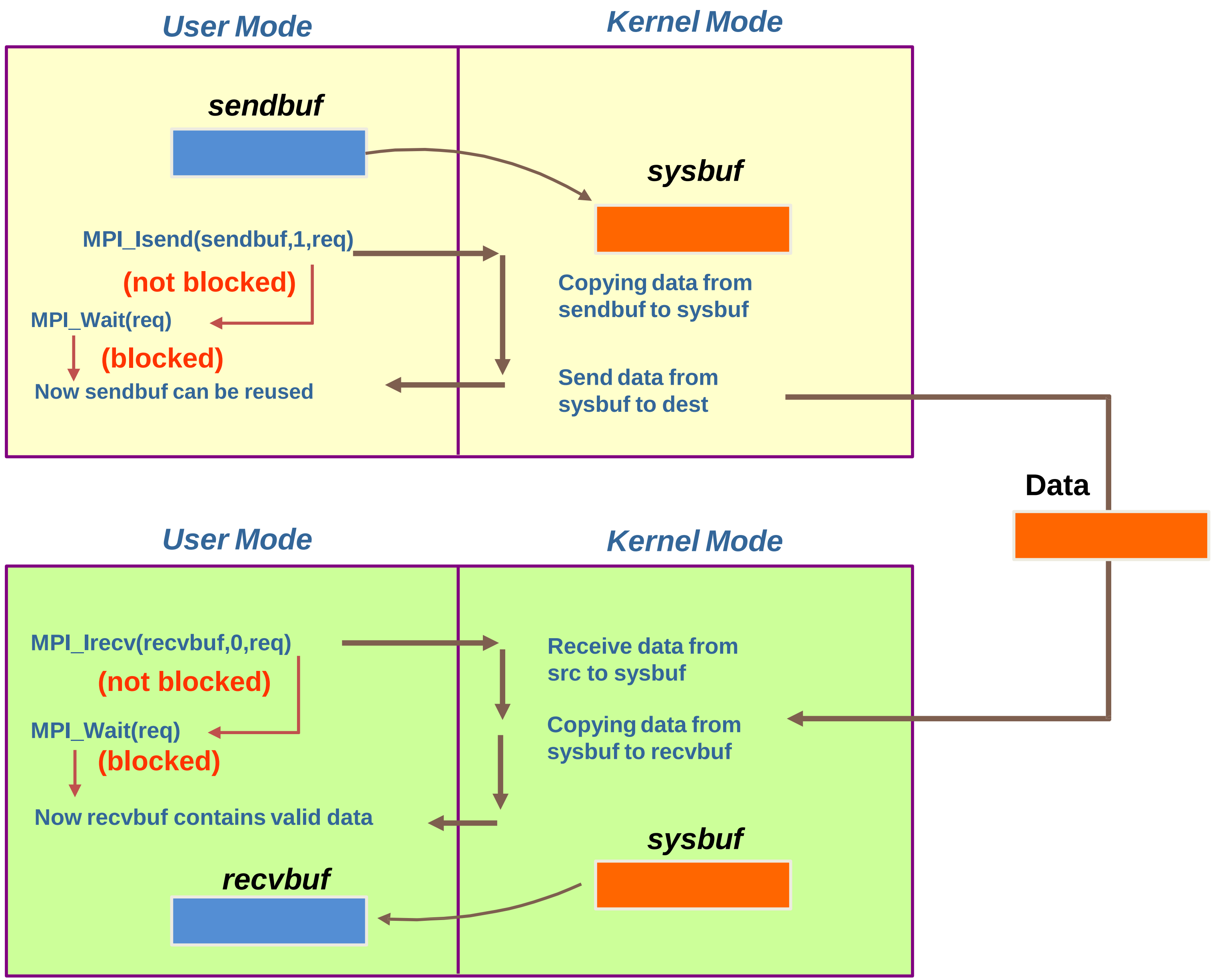
**MPI_Send** is a **blocking operation**. It **<u>may not complete until a matching receive is posted</u>**.

- **Improper use** may lead to <u>**Deadlocks**</u> too:

```
If (rank == 0) Then
        Call mpi_send(..)
        Call mpi_recv(..)
Usually deadlocks →   Else
        Call mpi_send(..)    ← UNLESS you reverse send/recv
        Call mpi_recv(..)
Endif
```

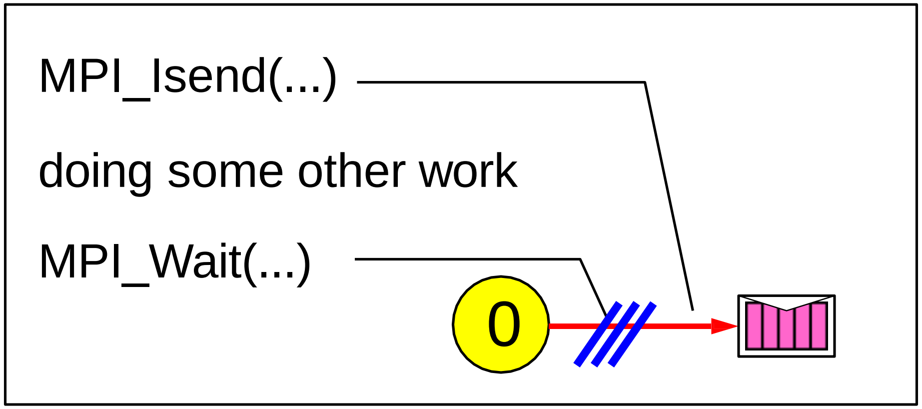
# Non-Blocking Point-to-Point Communication

# Non-Blocking Communication - Overview

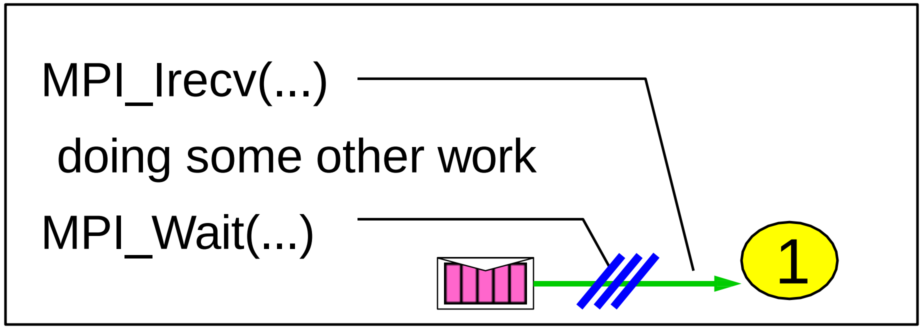


# Non-Blocking Communication - Overview

**Non-blocking Send**




**Non-blocking Receive**




= waiting until operation locally completed

# Non-Blocking Send and Receive

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
MPI_IRECV(buf, count, datatype, dest, tag, comm, request)
```

***request*** is a **request handle** which can be **used to query**:

- **Check** the **status** of the **communication**
- **Or wait for the completion**

# Non-blocking Communication

- **`MPI_Isend`** or **`MPI_Irecv`** **starts communication** and **returns** ***`request`*** **data structure**

- **`MPI_Wait`** (also **`MPI_Waitall`**, **`MPI_Waitany`**) uses **`request`** as an **argument** and **blocks until communication** is **complete**

- **`MPI_Test`** uses **`request`** as an **argument** and **checks for completion (non-blocking)**

- **Advantages:**
  - **No deadlocks (using MPI_Test for completion check)**
  - **Overlap communication with computation**
  - **Exploit bi-directional communication**

# Non-Blocking Send and Receive (Cont.)

```
MPI_WAIT (request, status)
MPI_TEST (request, flag, status)
```

Demo: P2PNonBlock.c

- **MPI_WAIT** will **block** until the **non-blocking send/receive** with the **desired request is done**

- The **MPI_TEST** is **simply queried** to see if the communication has completed (**TRUE** or **FALSE**) is returned immediately in **flag**

# Non-blocking Message Passing - Example

Demo:
P2PNonBlock.c

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

int main(int argc, char* argv[])
{
    int rank, size;
    int tag, destination, count;
    int buffer;

    tag = 1234;
    destination = 1;
    count = 1;
    MPI_Status status;
    MPI_Request request = MPI_REQUEST_NULL;

    MPI_Init(&argc, &argv);

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

# Non-blocking Message Passing - Example

```
 if (rank == 0) { /* master process */
      buffer = 9999;
      MPI_Isend(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD,&request);
     }

if (rank == destination)  /* slave process */
      MPI_Irecv(&buffer, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &request);

MPI_Wait(&request, &status); //Everyone wait here (both sender & receiver)

if (rank == 0)
      printf("process %d sent %d\n", rank, buffer);

if (rank == destination)
      printf("process %d rcv %d\n", rank, buffer)

MPI_Finalize();
return 0;
}
```

# Non-blocking Message Passing - Example

## Standard Output

```
Compiling
Compilation is OK
Execution ...
processor 0 sent 9999
processor 1 rcv 9999
Done.
```

# MPI_Probe

- Instead of **posting a receive** and simply **providing** a **really large buffer** to **handle** all **possible sizes of messages**

- You can use **MPI_Probe** to **query** the **message size** before **actually receiving it**:

*MPI_Probe(*
    *int* ***source****,*
    *int* ***tag****,*
    *MPI_Comm* ***comm****,*
    *MPI_Status** ***status****)*

# An Example

```
// Probe for an incoming message from process 0, tag 0
MPI_Probe(0, 0, MPI_COMM_WORLD, &status);

// When probe returns, the status object has the size and other
// attributes of the incoming message. Get the message size
MPI_Get_count(&status, MPI_INT, &number_amount);

// Allocate a buffer to hold the incoming numbers
int* number_buf = (int*)malloc(sizeof(int) * number_amount);

// Now receive the message with the allocated buffer
MPI_Recv(number_buf, number_amount, MPI_INT, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

**Demo:**
**`messageProbe.c`**

# **Any Questions**