Parallel and Distributed Computing

Dr. Ali Sayyed
Department of Computer Science
National University of Computer & Emerging Sciences

Distributed Memory Programming Model

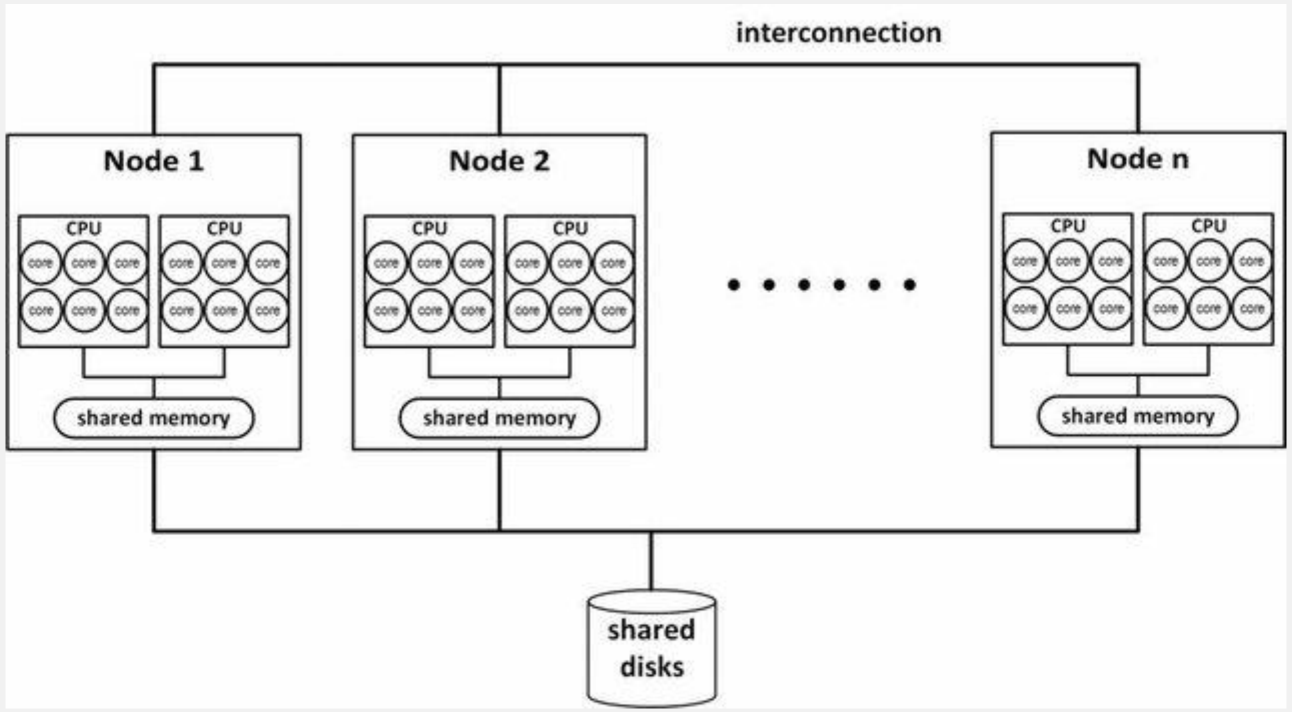
OpenMPI

Motivation

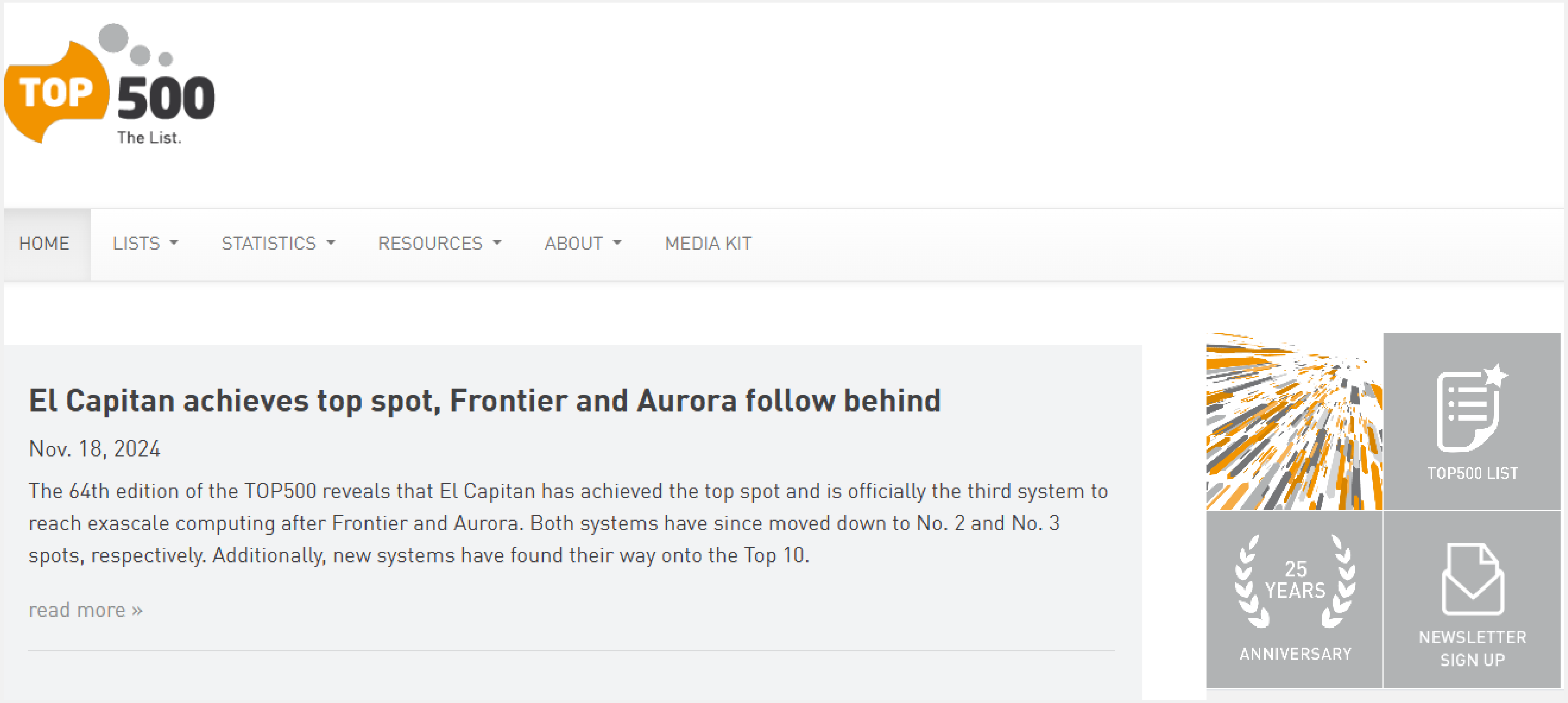
- **OpenMP** take advantage of **multiple CPU cores** on a shared memory system where **each core shares the same physical memory**, communicate data and synchronize their behavior.
- The **communication** in shared memory systems is relatively **easy**, however, its **scalability is limited** by the number of CPU cores.
- Some high-end server generally provide a **maximum of 64 cores**, however, for some tasks, even a few hundred CPU cores isn't close enough.
    - For example, to simulate the **fluid dynamics** of the Earth's oceans
- Such **massive tasks** require more physical memory and processors than any single computer can provide.
- Such applications **require systems built from multiple computers** that communicate over a network to coordinate their behavior.
- Such systems are known as **distributed memory system** (or often just distributed system).

Supercomputer

- Some distributed systems integrate hardware more closely than others.
- For example, a **supercomputer** is a high-performance system in which many compute nodes are tightly coupled (closely integrated) to a fast interconnection network.
- Each compute node contains its own CPU(s), GPU(s), and memory, but multiple nodes might share auxiliary resources like secondary storage and power supplies.
- The exact level of hardware sharing varies from one supercomputer to another.



Top500



HOME LISTS STATISTICS RESOURCES ABOUT MEDIA KIT

El Capitan achieves top spot, Frontier and Aurora follow behind

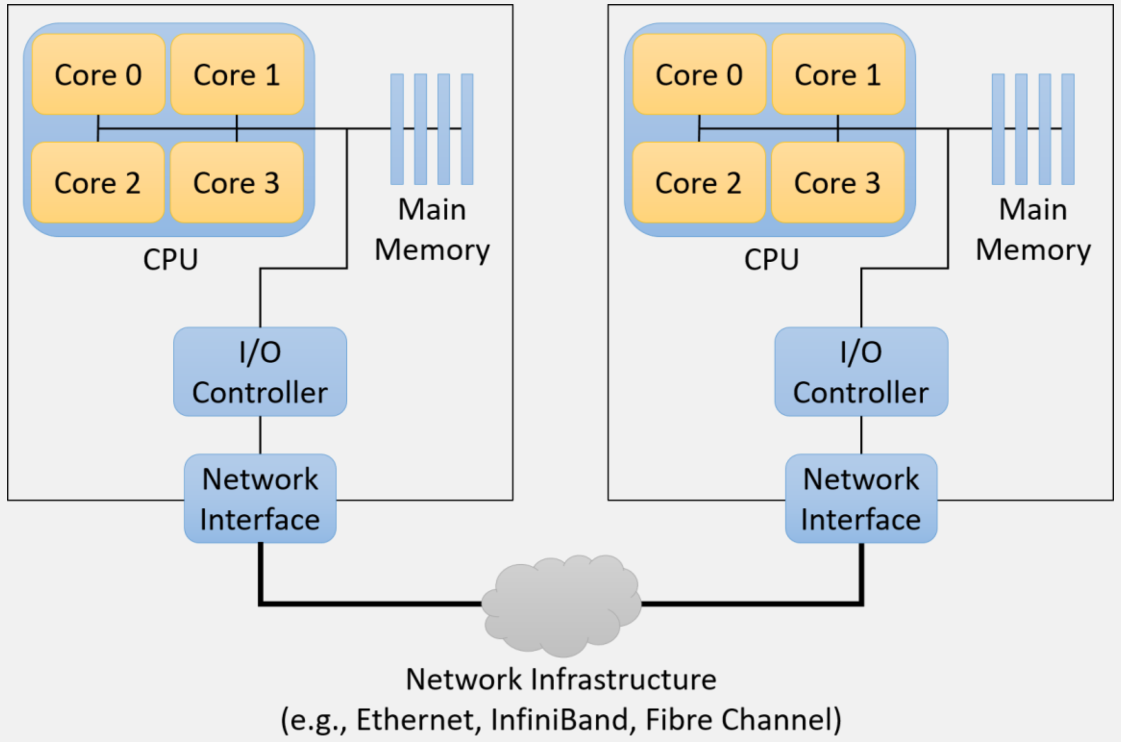
Nov. 18, 2024

The 64th edition of the TOP500 reveals that El Capitan has achieved the top spot and is officially the third system to reach exascale computing after Frontier and Aurora. Both systems have since moved down to No. 2 and No. 3 spots, respectively. Additionally, new systems have found their way onto the Top 10.

read more »

COTS Cluster

- On the other end of the spectrum, a distributed system might have a **loosely coupled** (less integrated) collection of fully autonomous computers (nodes) connected by a **traditional LAN**.
- Such a collection of nodes is called **commodity off-the-shelf (COTS)** cluster.
- COTS clusters typically employ a **shared-nothing architecture** in which each node contains its own set of computation hardware.



Communication Protocols

- Whether they are part of a supercomputer or a COTS cluster, **processes in a distributed memory system communicate via message passing**, where one process explicitly sends a message to another process.
- It's up to the applications running on the system to determine how to utilize the network.
- Some applications require frequent communication to tightly coordinate the behavior of processes, whereas other applications communicate to divide up a large input among processes and then mostly work independently.
- In any case a communication protocol is required
    - When a process should send a message
    - To which process(es) it should send the message
    - How to format the message

Message Passing Interface (MPI)

- The MPI defines (but does not itself implement) a **standardized interface** that applications can use to communicate in a distributed system.
- Supported by Fortran, C, C++ (but modules also available for python, & Java)

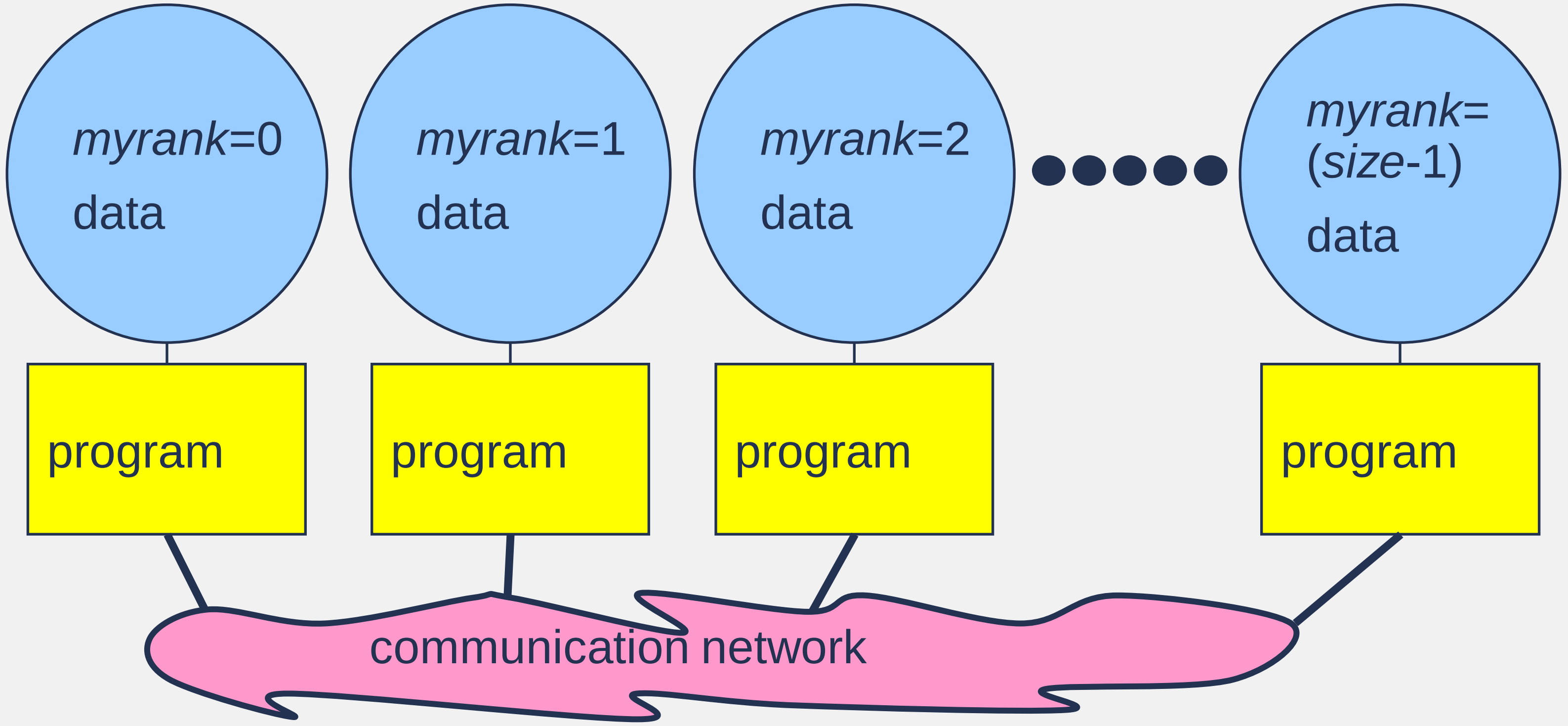
- By adopting the MPI communication **standard**, applications become **portable**, meaning that they can be compiled and executed on many different systems.

- MPI allows a programmer to **divide an application into multiple processes**.

- It assigns each of the processes a **unique identifier**, known as a **Rank**, which ranges from **0 to N-1** for an application with N processes.

Data and Work Distribution

- To communicate together mpi-processes need identifiers: rank = identifying number
- all distribution decisions are based on the rank
  - i.e., which process works on which data



Is MPI Large or Small?

- **MPI is large** (400+ functions)
  - MPI's extensive functionality requires many functions
  - Number of functions not necessarily a measure of complexity

- **MPI is small** (10-12 functions)
  - Many parallel programs (90%) can be written with just 10-12 basic functions.

MPI Process Identification

- **Two of the fundamental questions** asked in a parallel program are:
  - How many processes are there?
  - Who am I?

- How many is answered with **MPI_Comm_size(comm, &count)**
- who am I is answered with **MPI_Comm_rank(comm, &rank)**

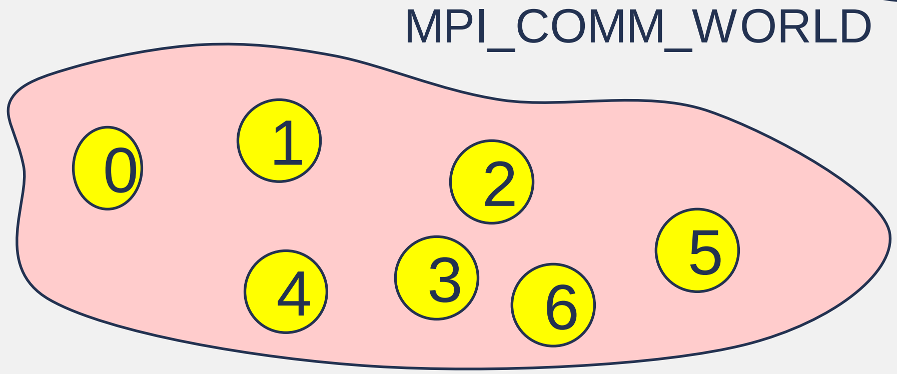
- The rank is a number between zero and size-1.

Starting & 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_”.

MPI Communicators

- A **communicator** defines a communication domain - a set of processes that are allowed to communicate with each other (also called **Groups**)
- Each message is **sent in a context** and **must be received in the same context**.
- Communicators are used **as arguments** to **all message transfer MPI routines**.
- A Process is identified by its **Rank** in the Group associated with a Communicator
- A process can belong to many different (possibly overlapping) communication domains.
- There is a default Communicator whose group contain all initial Processes, called **MPI_COMM_World**



The minimal set of MPI routines.

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

MPI Hello World

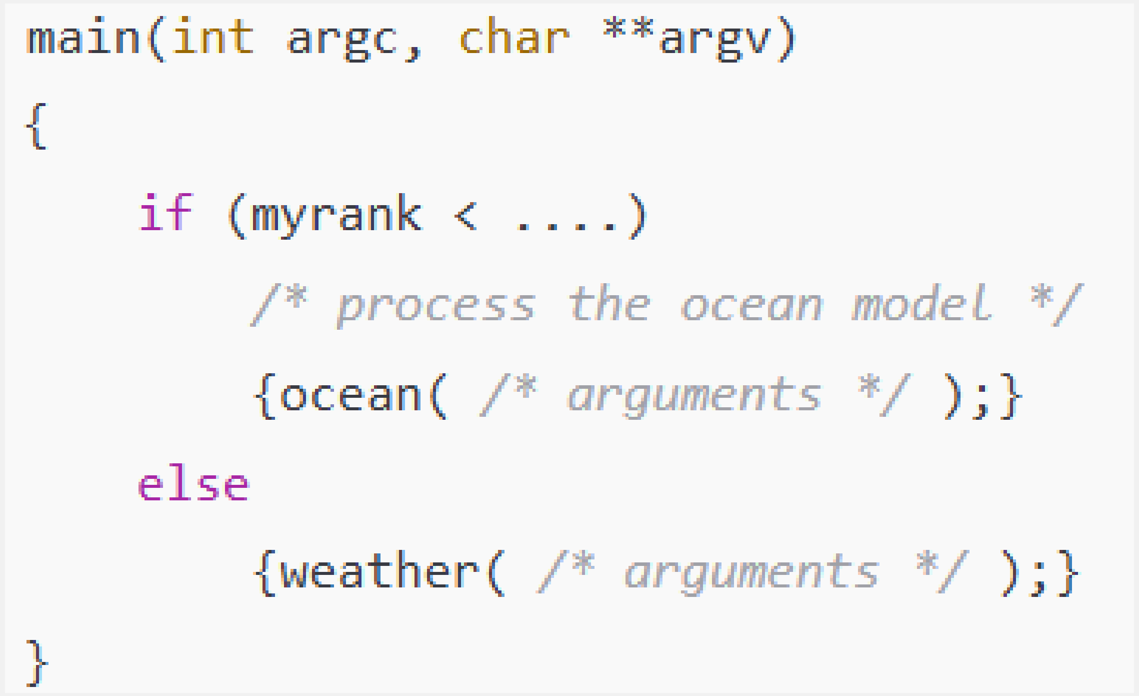
```
#include "mpi.h"
int main(int argc, char **argv) {
    int rank, process_count;
    MPI_Init(&argc, &argv);  // Initialize and start MPI.
  /* Determine how many processes there are and which one this is. */
  MPI_Comm_size(MPI_COMM_WORLD, &process_count);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  /* Print a message, identifying the process and machine it comes from. Note
that the printf will run on each process*/
 printf("Hello from process %d of %d\n", rank, process_count);
 MPI_Finalize();  // Clean up and exit MPI.
 return 0;
}
```

MPI is primarily for SIMD/MIMD

- **SIMD (Single Instruction, Multiple Data):**
  - SIMD refers to a parallel computing model where **the same instruction** (or program) runs on **multiple processors**, but each processor works on **different pieces of data**.
  - **Example**: In image processing, the same filter operation can be applied to multiple pixels simultaneously.
- **MIMD (Multiple Instruction, Multiple Data):**
  - MIMD refers to a parallel computing model where **different processors execute different instructions** (or programs) on **different data**.
  - **Example**: One processor runs a weather simulation while another runs an ocean simulation.

MIMD Can Be Emulated with SIMD

- While SIMD inherently requires all processors to run the same program
  - MIMD behavior can be emulated in SIMD.
  - **Key Idea**: The same program runs on all processors, but each processor behaves differently based on its unique identifier (myrank).



```
main(int argc, char **argv)
{
    if (myrank < ....)
        /* process the ocean model */
        {ocean( /* arguments */ );}
    else
        {weather( /* arguments */ );}
}
```

Advantages of This Approach

- **Simplicity of Code Distribution**
  - Only one program is distributed across all processors.

- **Flexibility**
  - Different tasks can be assigned to processors dynamically using the if condition.

Running MPI Code

To run MPI examples, you'll need an MPI implementation

- How to Install and **Use OpenMPI** for Parallel Computing in **Linux (Fedora)**
  - https://www.youtube.com/watch?v=NpDKxFO6fHM

- How to setup **VS Code** for Parallel Programming using **MS-MPI** in Windows
  - https://www.youtube.com/watch?v=bkfCrj-rBjU

mpicc

- Make sure **OpenMPI** is installed
  - You can check by typing in terminal, If the following command shows the version, OpenMPI is installed. If not, you need to install it first.

  -
    ```
    mpicc --version
    ```

- Use **mpicc** to compile:
  - This will create an executable file called **prog1**

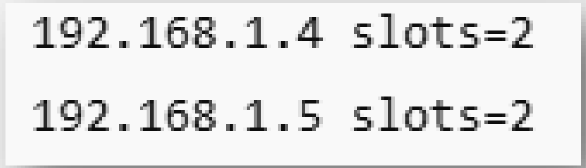
  -
    ```
    mpicc prog1.c –o prog1
    ```

mpirun

- The **mpirun** command supports a large number of command line options.
- To see a complete list of these options
  - `mpirun --help`
- The following are commonly-used options for executing applications with mpirun:
- One process on the local machine
  - `mpirun prog1`
- Two processes on the local machine
  - `mpirun -np 2 prog1`

Running MPI code on a Network

- **Step 1:** Setup the network
    - You need at least two computers connected in a network.
- **They should:**
    - Be able to ping each other (they must "see" each other)
    - Have OpenMPI installed on both
    - Have the same MPI program copied to both computers
- **Step 2:** Allow passwordless SSH
    - You need SSH access without password between the computers.
- **Step 3:** Prepare the program
    - Compile the MPI program on all computers
- **Step 4:** Create a "hostfile“
    - Make a simple text file (call it hosts.txt) with the IP addresses or names of computers.



```
192.168.1.4 slots=2
192.168.1.5 slots=2
```

Running MPI code on a Network

- Step 5: Run your MPI code
- Start one instance of prog1 on h1 node and another on the h2 node
  - `mpirun –np 2 -host h1,h2 prog1`
- Start one instance (total 3) of prog1 on each host in hosts.txt
  - `mpirun -np 3 --hostfile hosts.txt prog1`

- mpirun command can be issues from any computer in the network as long as it can SSH into the other computers without password and It has the MPI program ready (cmpiled).

Order of Execution in Processes

- Never make **any assumptions** about the **order** in which MPI processes will execute.
- The processes start up on multiple machines, each of which has its own OS and process scheduler.
- If the correctness of your program requires that processes run in a particular order, you **must ensure that the proper order occurs**
    - for example, by forcing certain processes to pause until they receive a message.

Distributed Processing Models

- Application designers often **organize distributed applications** using different designs.
- Each model has its unique benefits and drawbacks — there's no one-size-fits-all solution.
- We briefly characterize a few (not presenting an exhaustive list).
  - Client/Server
  - Boss/Worker
  - Peer-to-Peer

Client/Server

- The client/server model is an **extremely common** application model that divides an application's responsibilities among **two actors**:
    - client processes and server processes.
- The **server process provides a service** to clients.
- A client sends requests to the server process, which either satisfies those requests (e.g., by fetching a requested file) or reports an error (e.g., the file not existing or the client can't be properly authenticated).
- Although you may not have considered it, you access your course material from Google Classroom via the client/server model

Boss/Worker

- In the boss/worker model, **one process** acts as a central coordinator and distributes work among the processes at other nodes.
- This model works well for problems that require processing a **large, divisible input**. The boss divides the input into smaller pieces and assigns one or more pieces to each worker.
- In some applications, the boss might statically assign each worker exactly one piece of the input. In other cases, the workers might repeatedly finish a piece of the input and then return to the boss to dynamically retrieve the next input chunk.

Peer-to-Peer

- Unlike the boss/worker model, a peer-to-peer application avoids relying on a centralized control process.
- Instead, peer processes self-organize the application into a structure in which they each take on roughly the same responsibilities.
- For example, in the BitTorrent file sharing protocol, each peer repeatedly exchanges parts of a file with others until they've all received the entire file.
- Lacking a centralized component, peer-to-peer applications are generally robust to node failures. On the other hand, peer-to-peer applications typically require complex coordination algorithms, making them difficult to build.

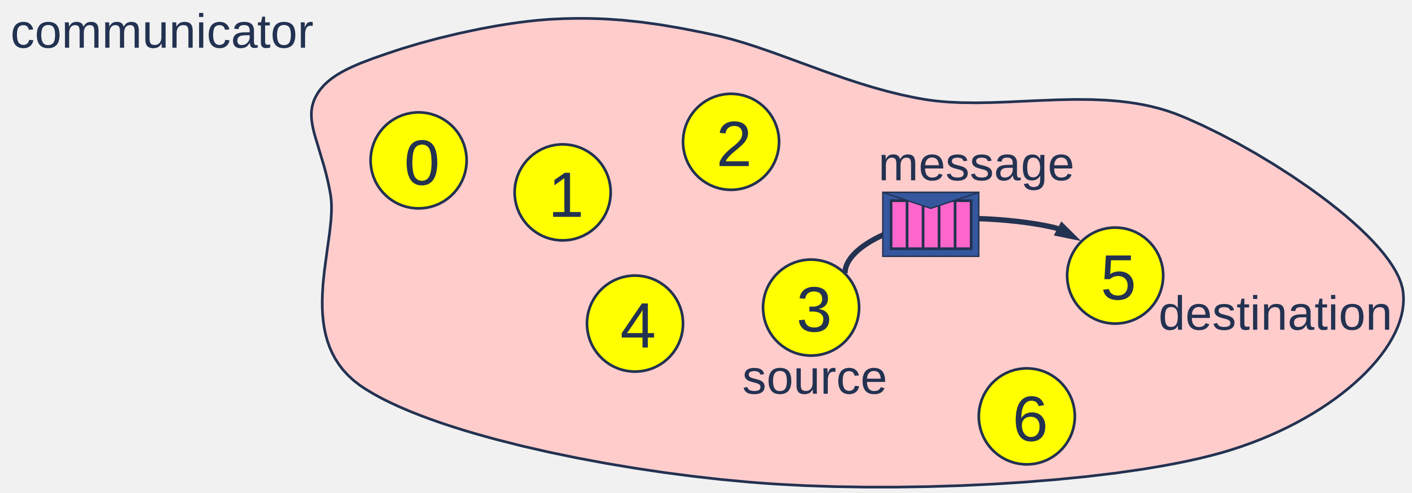
MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message.

- Messages can be screened at the receiving end by specifying a specific tag or not screened by specifying **MPI_ANY_TAG** as the tag in a receive.

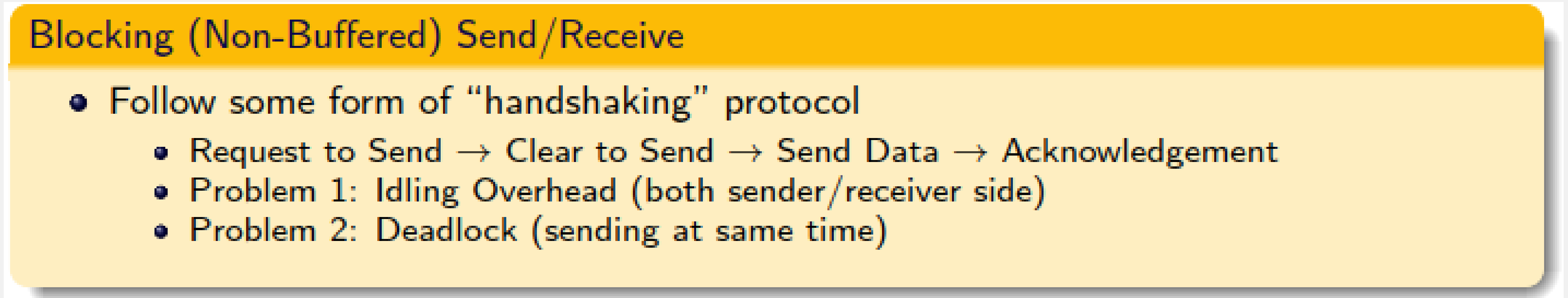
Point to Point Message Passing

- It is the simplest form of message passing.
  - **One process sends a message to another**.
  - Communication takes place within a communicator, e.g., MPI_COMM_WORLD.
  - Processes are identified by their ranks in the communicator.



Point to Point Message Passing

- **Synchronous Send (Blocking Send):**
  - The call waits until the data transfer is done
  - The sender sends and wait for a response from receiver. The sender gets an information that the message is received.
  - The sending process waits until all data are transferred
  - The receiving process waits until all data are transferred from the system buffer to the receive buffer

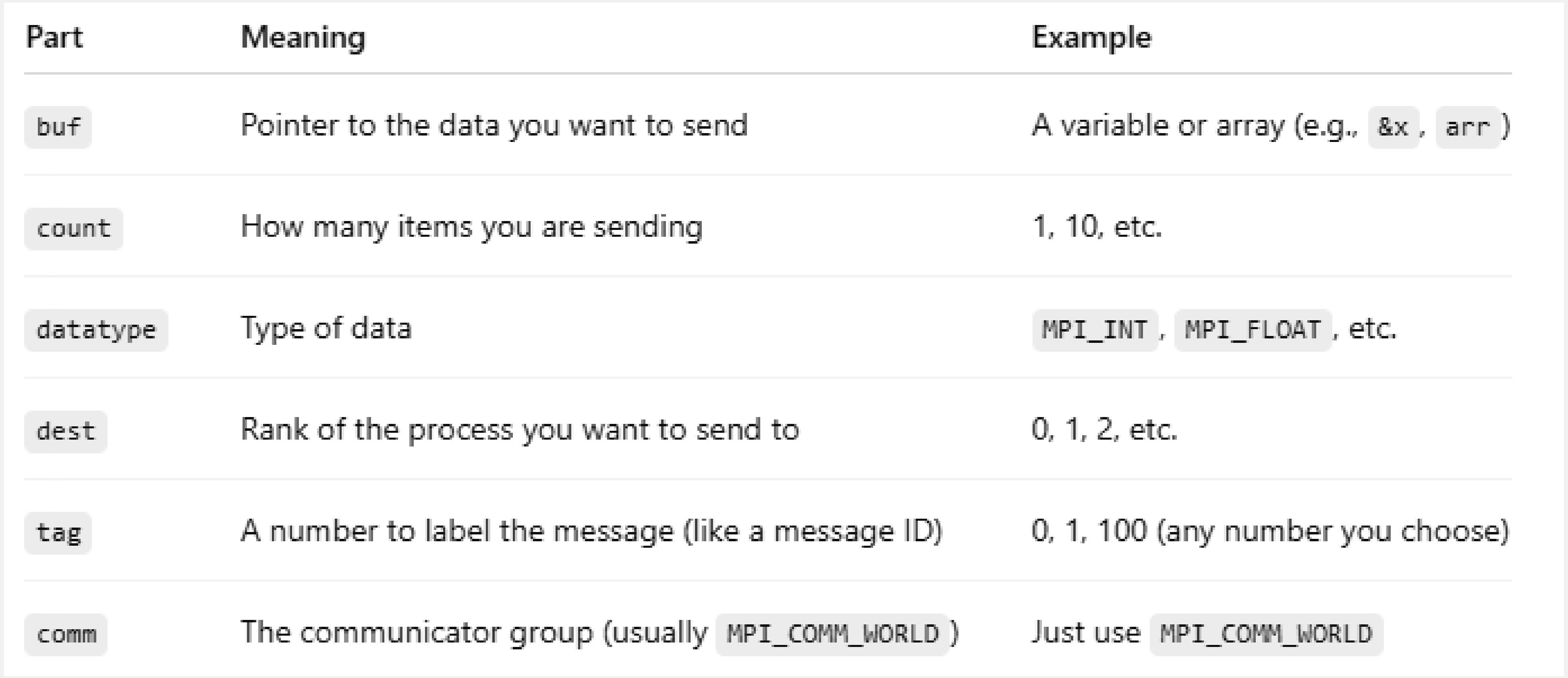


Blocking (Non-Buffered) Send/Receive

- Follow some form of "handshaking" protocol
  - Request to Send → Clear to Send → Send Data → Acknowledgement
  - Problem 1: Idling Overhead (both sender/receiver side)
  - Problem 2: Deadlock (sending at same time)

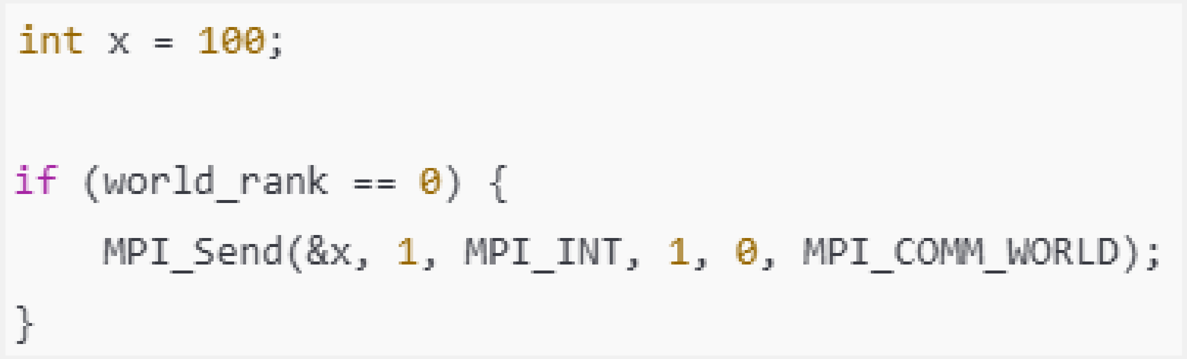
MPI_Send

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



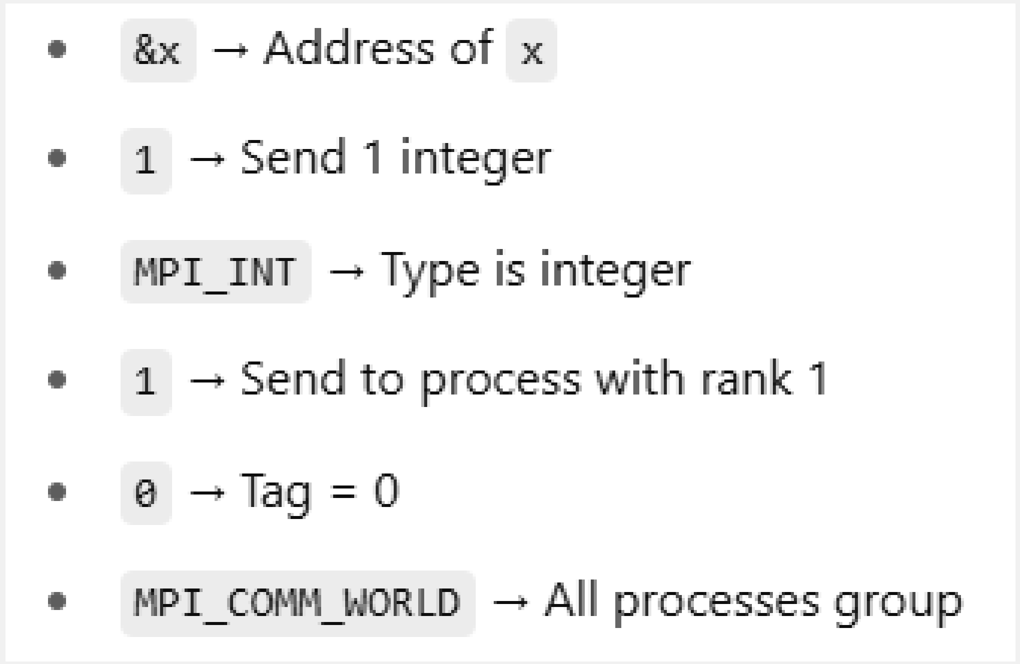
Part	Meaning	Example
`buf`	Pointer to the data you want to send	A variable or array (e.g., `&x`, `arr`)
`count`	How many items you are sending	1, 10, etc.
`datatype`	Type of data	`MPI_INT`, `MPI_FLOAT`, etc.
`dest`	Rank of the process you want to send to	0, 1, 2, etc.
`tag`	A number to label the message (like a message ID)	0, 1, 100 (any number you choose)
`comm`	The communicator group (usually `MPI_COMM_WORLD`)	Just use `MPI_COMM_WORLD`

MPI_Send Example



```
int x = 100;

if (world_rank == 0) {
    MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
}
```

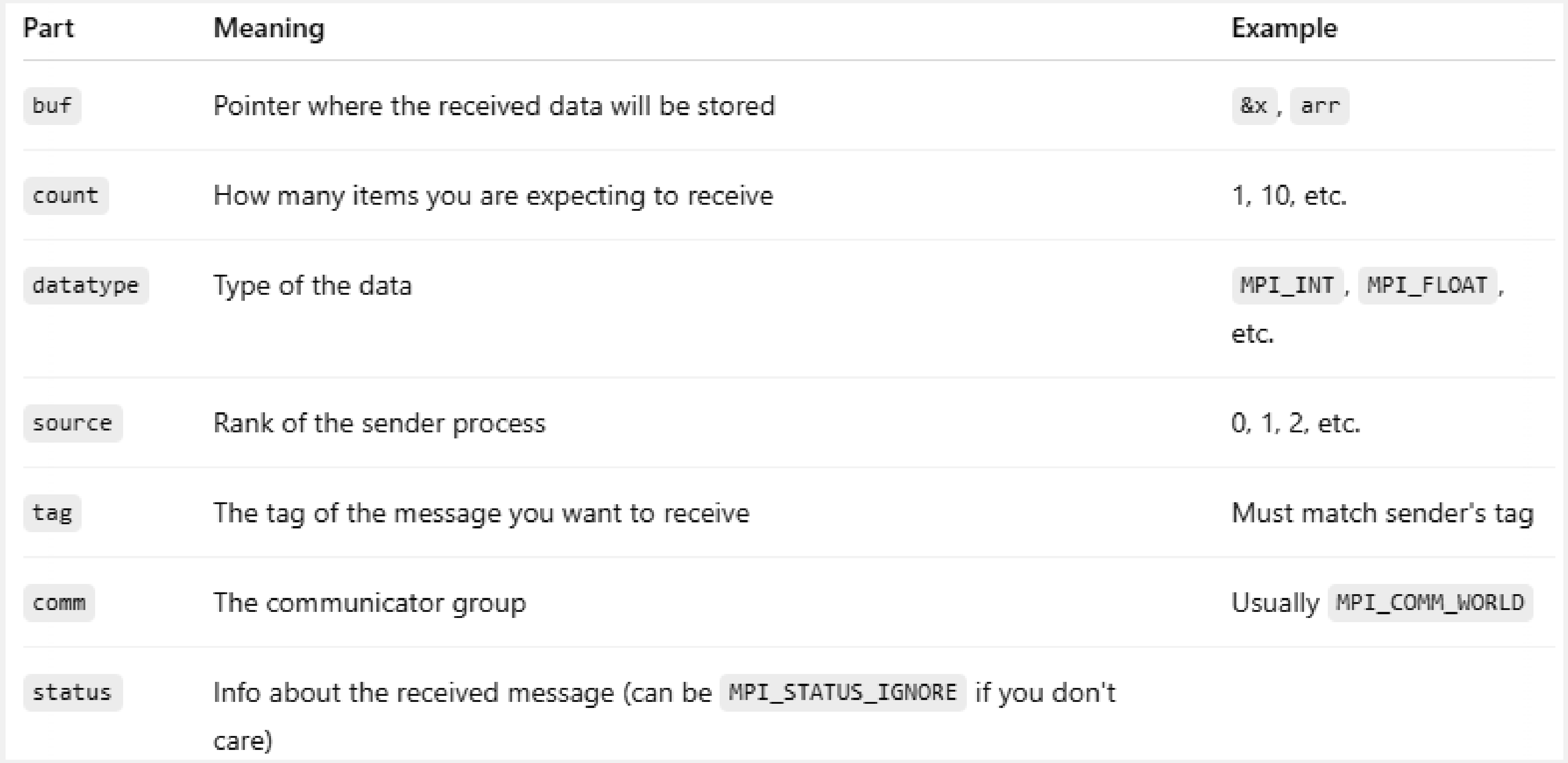


- `&x` → Address of `x`
- `1` → Send 1 integer
- `MPI_INT` → Type is integer
- `1` → Send to process with rank 1
- `0` → Tag = 0
- `MPI_COMM_WORLD` → All processes group

- **Important**
- If you use MPI_Send, there must be a matching MPI_Recv (receive) at the destination!

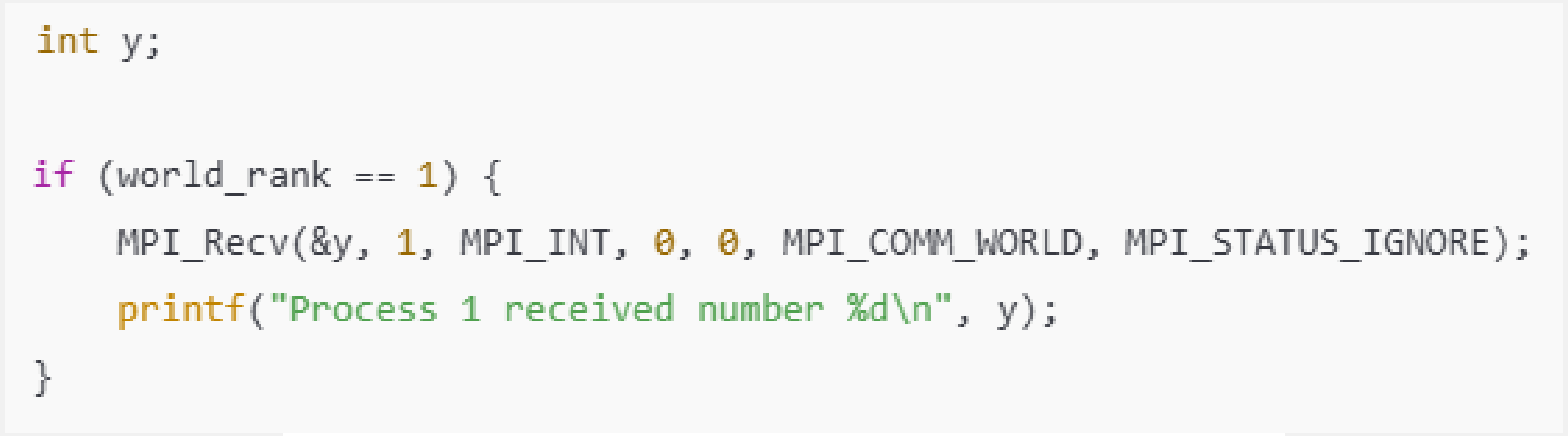
MPI_Recv

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



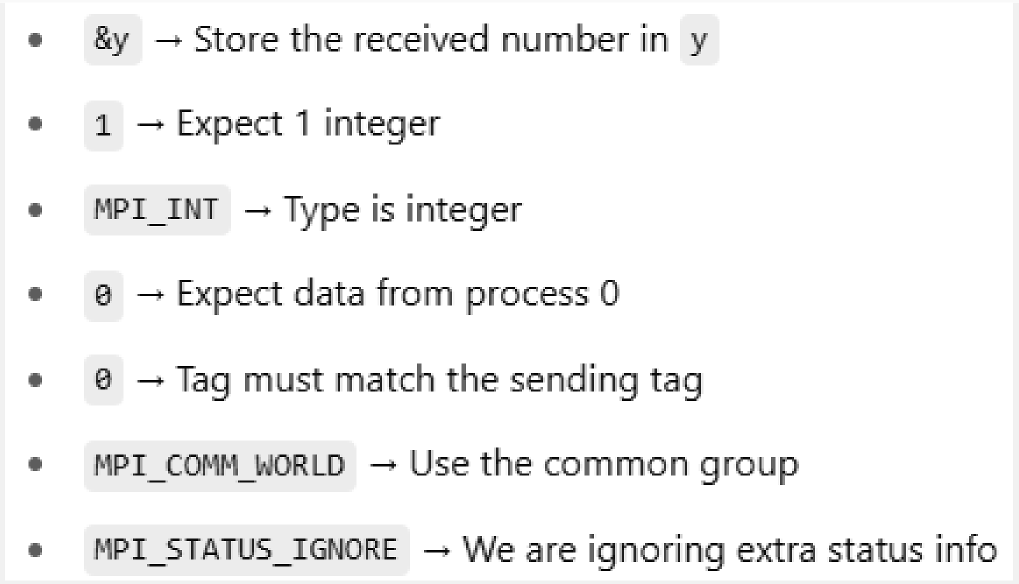
Part	Meaning	Example
`buf`	Pointer where the received data will be stored	`&x`, `arr`
`count`	How many items you are expecting to receive	1, 10, etc.
`datatype`	Type of the data	`MPI_INT`, `MPI_FLOAT`, etc.
`source`	Rank of the sender process	0, 1, 2, etc.
`tag`	The tag of the message you want to receive	Must match sender's tag
`comm`	The communicator group	Usually `MPI_COMM_WORLD`
`status`	Info about the received message (can be `MPI_STATUS_IGNORE` if you don't care)	

MPI_Recv Example



```c
int y;

if (world_rank == 1) {
    MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received number %d\n", y);
}
```



- `&y` → Store the received number in `y`
- `1` → Expect 1 integer
- `MPI_INT` → Type is integer
- `0` → Expect data from process 0
- `0` → Tag must match the sending tag
- `MPI_COMM_WORLD` → Use the common group
- `MPI_STATUS_IGNORE` → We are ignoring extra status info

Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message datatypes must match.

Receiver can wildcard.

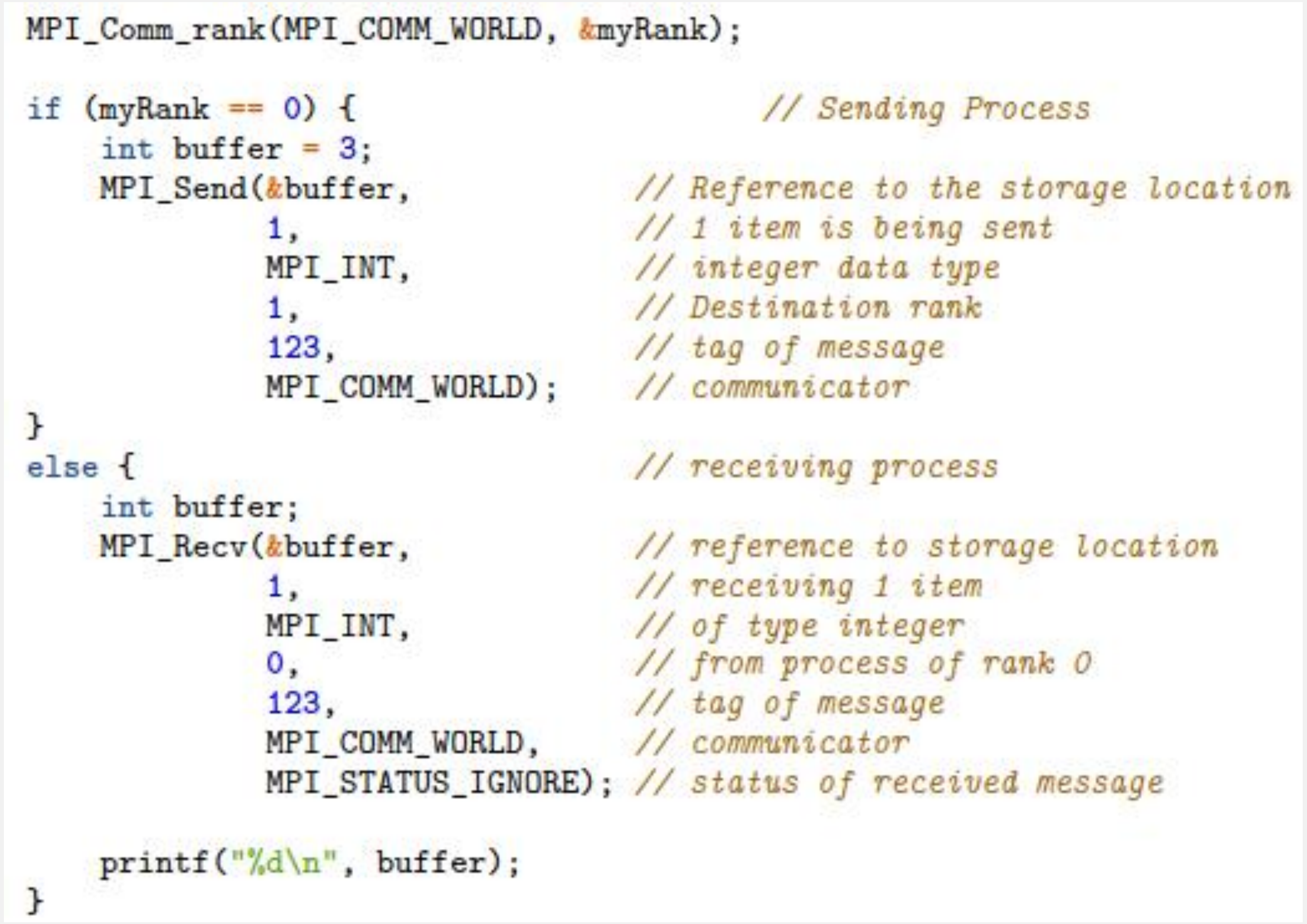
- ○ To receive from any source
- ✓ source = MPI_ANY_SOURCE
- ○ To receive with any tag
- ✓ tag = MPI_ANY_TAG

MPI_Status

- On the receiving end, the **status** variable can be used to get information about the MPI_Recv operation.
- The corresponding data structure contains:
  - `typedef struct MPI_Status {`
  - `int MPI_SOURCE;`
  - `int MPI_TAG;`
  - `int MPI_ERROR; };`

Example

Blocking Send/Receive between Two Processes



```c
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

if (myRank == 0) {                               // Sending Process
    int buffer = 3;
    MPI_Send(&buffer,                   // Reference to the storage location
             1,                         // 1 item is being sent
             MPI_INT,                   // integer data type
             1,                         // Destination rank
             123,                       // tag of message
             MPI_COMM_WORLD);           // communicator
}
else {                                  // receiving process
    int buffer;
    MPI_Recv(&buffer,                   // reference to storage location
             1,                         // receiving 1 item
             MPI_INT,                   // of type integer
             0,                         // from process of rank 0
             123,                       // tag of message
             MPI_COMM_WORLD,            // communicator
             MPI_STATUS_IGNORE);        // status of received message

    printf("%d\n", buffer);
}
```

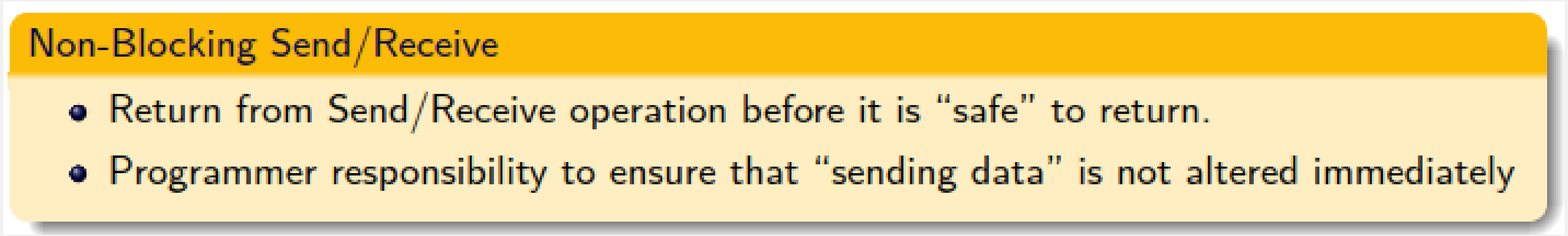
Deadlocks

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

If MPI Send is blocking, there is a deadlock.

Point to Point Message Passing

- **Asynchronous Send (Non-Blocking Send):**
  - It starts sending the data, but does not wait for it to finish.
  - The sender sends the message without waiting for a response from receiver. The operation will complete WITHOUT waiting for the message reception by the receiver.



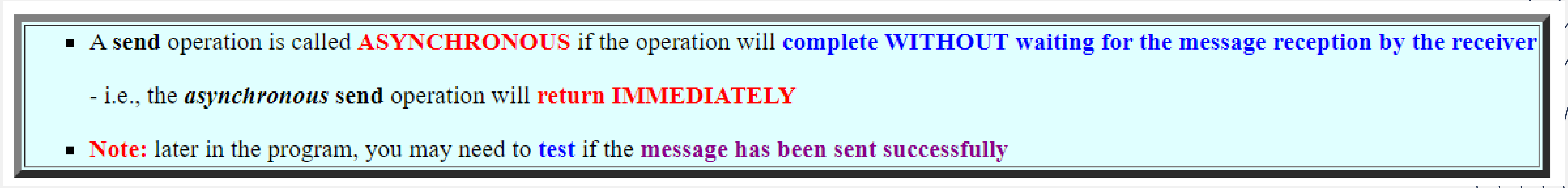
Non-Blocking Send/Receive

- Return from Send/Receive operation before it is "safe" to return.
- Programmer responsibility to ensure that "sending data" is not altered immediately

Non-blocking Message Passing

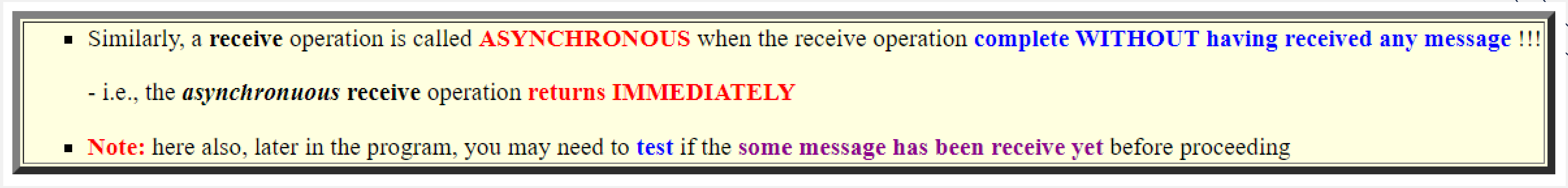
- Returns immediately after the data transferred is initiated
- Allows to overlap computation with communication
  - returns Immediately
  - routine name starting with MPI_I…
- Do some work
  - “latency hiding”
- Wait for non-blocking communication to complete
- Need to be careful though
  - When send and receive buffers are updated before the transfer is over, the result will be wrong

Asynchronous Send and Receive



- A **send** operation is called **ASYNCHRONOUS** if the operation will **complete WITHOUT waiting for the message reception by the receiver**

  - i.e., the ***asynchronous* send** operation will **return IMMEDIATELY**
- **Note:** later in the program, you may need to **test** if the **message has been sent successfully**



- Similarly, a **receive** operation is called **ASYNCHRONOUS** when the receive operation **complete WITHOUT having received any message** !!!

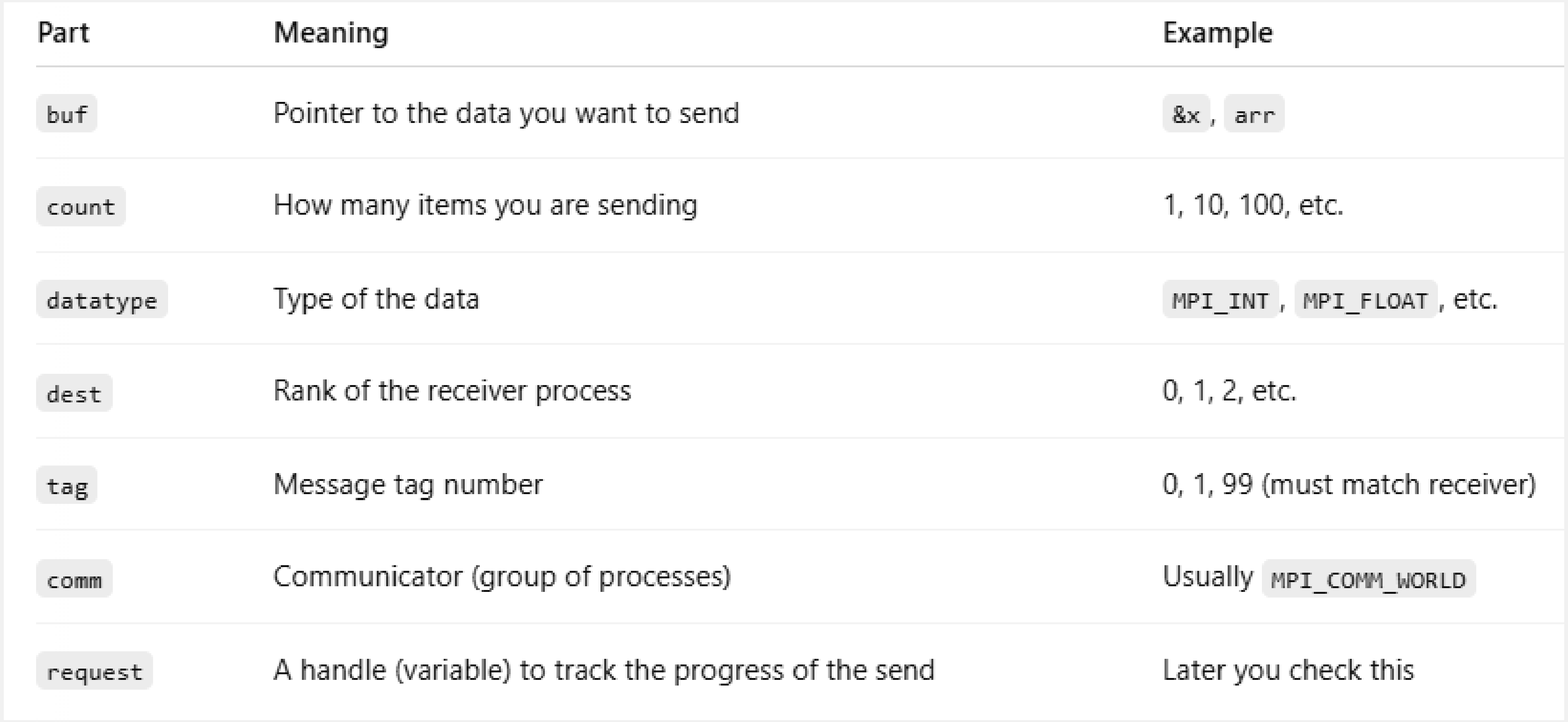
  - i.e., the ***asynchronuous* receive** operation **returns IMMEDIATELY**
- **Note:** here also, later in the program, you may need to **test** if the **some message has been receive yet** before proceeding

MPI_Isend

```
int MPI_Isend(
    void *buf,          // Starting address of the send buffer
    int count,          // Number of elements to send
    MPI_Datatype datatype, // Type of each element (e.g., MPI_INT)
    int dest,           // Rank of destination process
    int tag,            // Message tag (for identification)
    MPI_Comm comm,      // Communicator (usually MPI_COMM_WORLD)
    MPI_Request *request // Request handle to track the communication
);
```

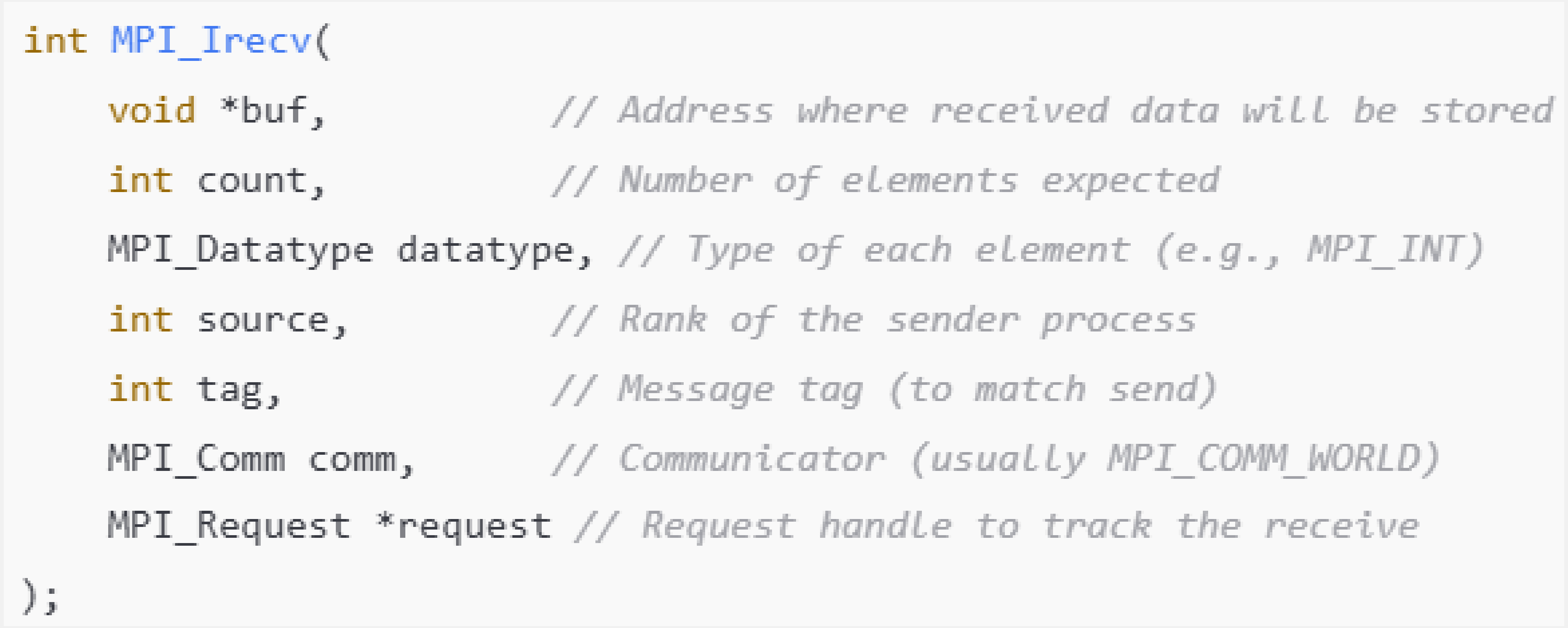
NOTE: DO NOT change/update the variable **buff** until the message has been sent !!!

MPI_Isend



Part	Meaning	Example
`buf`	Pointer to the data you want to send	`&x`, `arr`
`count`	How many items you are sending	1, 10, 100, etc.
`datatype`	Type of the data	`MPI_INT`, `MPI_FLOAT`, etc.
`dest`	Rank of the receiver process	0, 1, 2, etc.
`tag`	Message tag number	0, 1, 99 (must match receiver)
`comm`	Communicator (group of processes)	Usually `MPI_COMM_WORLD`
`request`	A handle (variable) to track the progress of the send	Later you check this

MPI_Irecv

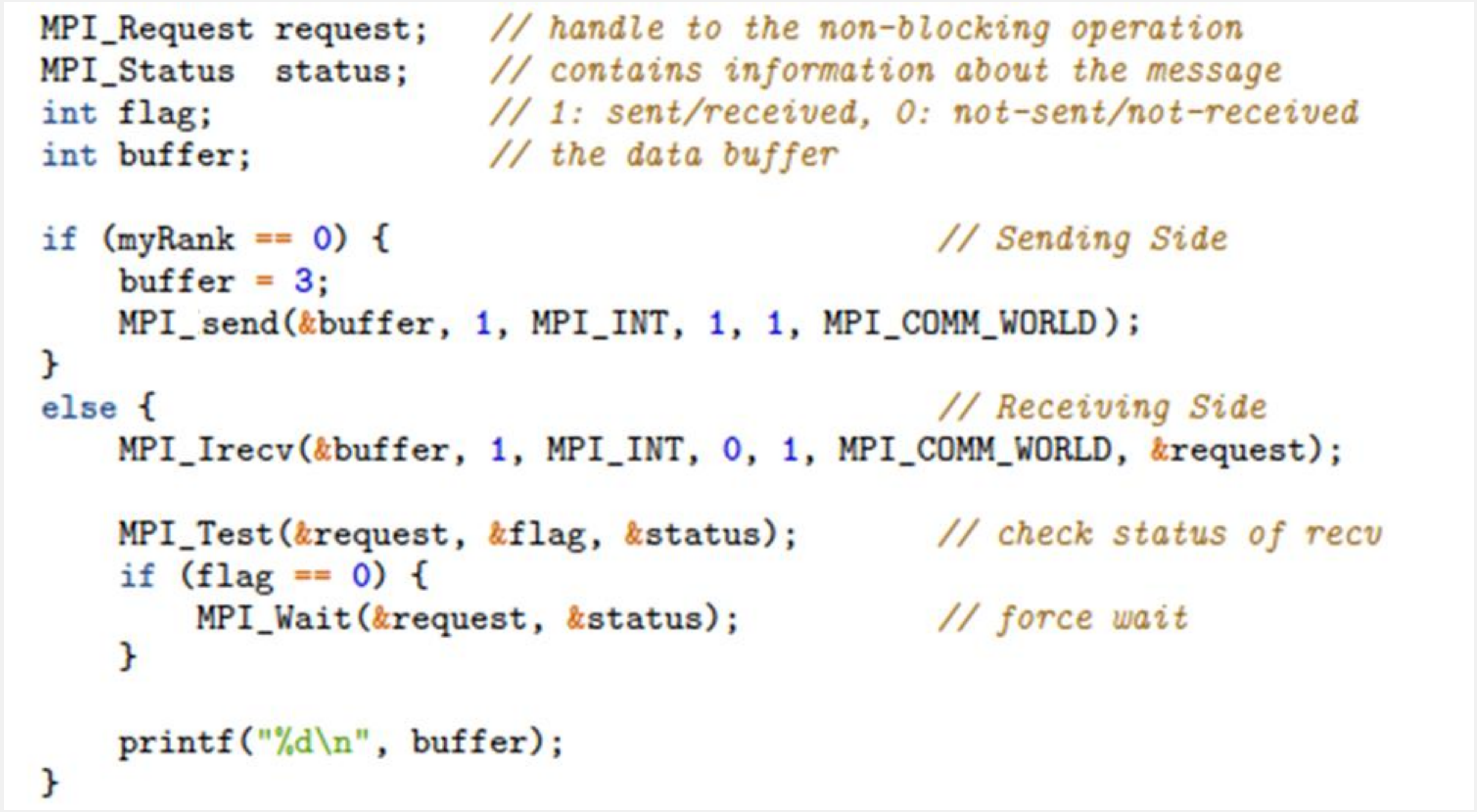


```c
int MPI_Irecv(
    void *buf,          // Address where received data will be stored
    int count,          // Number of elements expected
    MPI_Datatype datatype, // Type of each element (e.g., MPI_INT)
    int source,         // Rank of the sender process
    int tag,            // Message tag (to match send)
    MPI_Comm comm,      // Communicator (usually MPI_COMM_WORLD)
    MPI_Request *request // Request handle to track the receive
);
```

MPI request variable

- MPI_Request is like a "ticket" or "receipt" that helps you check or wait until the communication is really done.
- Because non-blocking operations happen in the background. You need a way to:
    - Wait until the send or receive is complete (MPI_Wait).
    - Check if it is complete without stopping (MPI_Test).
- MPI_Request stores the information MPI needs to manage these communications.

Example



```
MPI_Request request;   // handle to the non-blocking operation
MPI_Status  status;    // contains information about the message
int flag;              // 1: sent/received, 0: not-sent/not-received
int buffer;            // the data buffer

if (myRank == 0) {                                 // Sending Side
    buffer = 3;
    MPI_send(&buffer, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
}
else {                                             // Receiving Side
    MPI_Irecv(&buffer, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &request);

    MPI_Test(&request, &flag, &status);            // check status of recv
    if (flag == 0) {
        MPI_Wait(&request, &status);               // force wait
    }

    printf("%d\n", buffer);
}
```

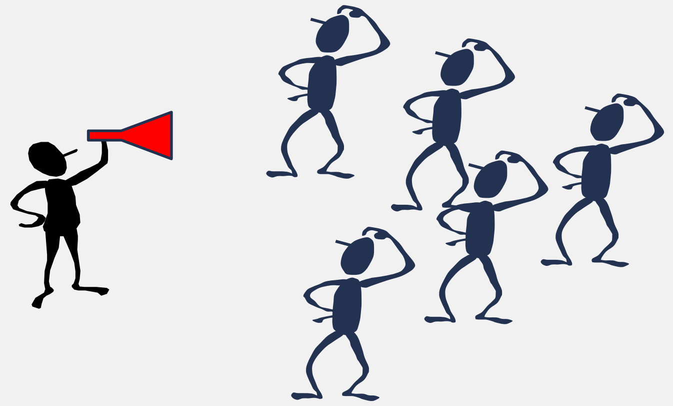
Blocking and Non-Blocking

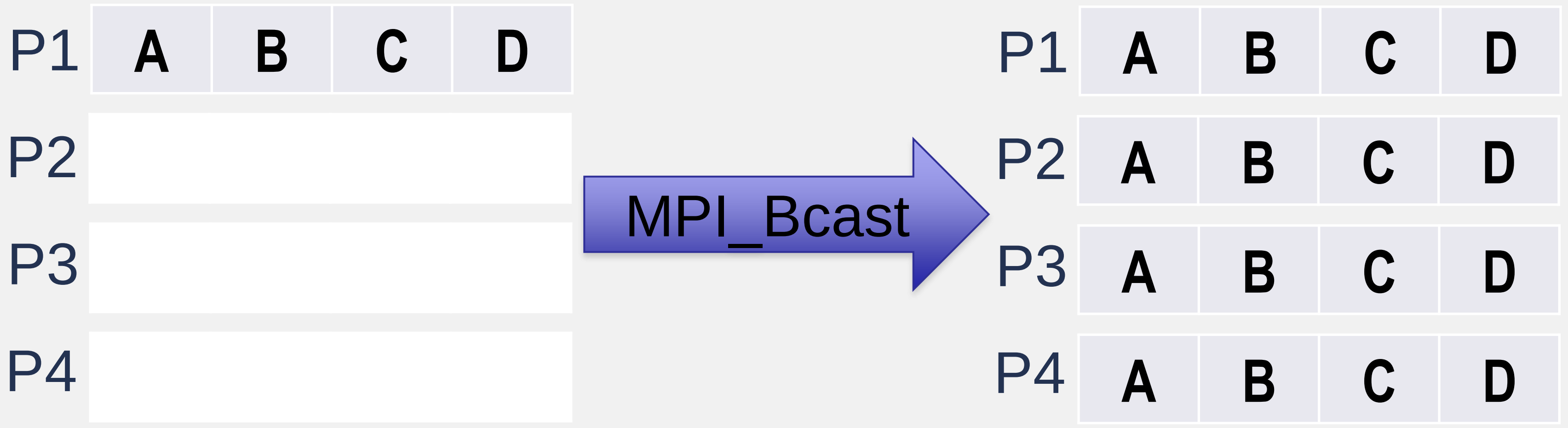
- Send and receive can be blocking or non-blocking.
- A blocking send can be used with a non-blocking receive, and vice-versa.
- So, you can mix and match these behaviors. For example, you could have a situation where sending is blocking but receiving is non-blocking, or the other way around. It depends on what you need for your program to work smoothly.

One-to-Many or Many-to-One Message Communication

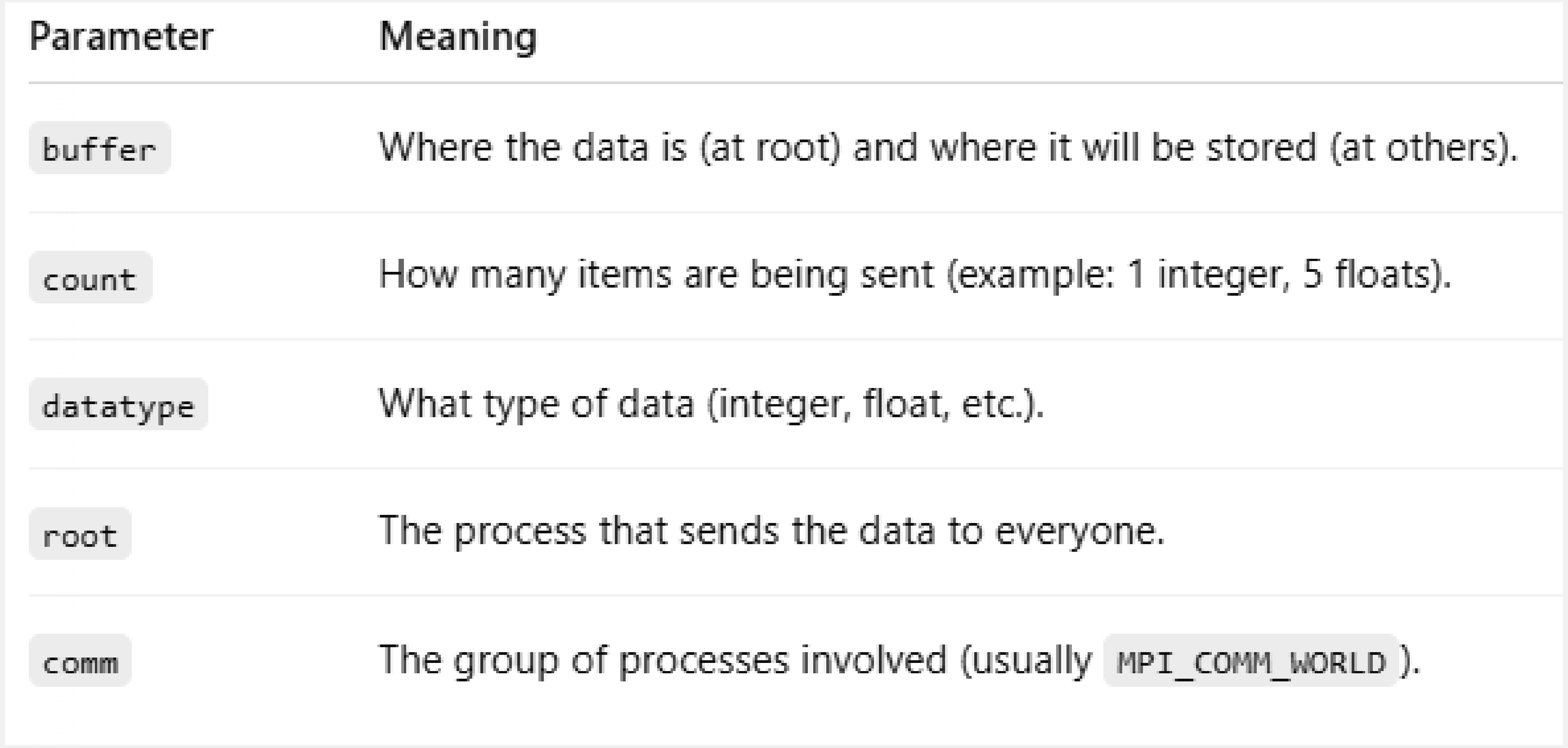
Broadcast

- `int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);`
    - One process (root) sends data to all the other processes in the same communicator
    - Must be called by all the processes with the same arguments





Bcast Parameters



Parameter	Meaning
`buffer`	Where the data is (at root) and where it will be stored (at others).
`count`	How many items are being sent (example: 1 integer, 5 floats).
`datatype`	What type of data (integer, float, etc.).
`root`	The process that sends the data to everyone.
`comm`	The group of processes involved (usually `MPI_COMM_WORLD`).

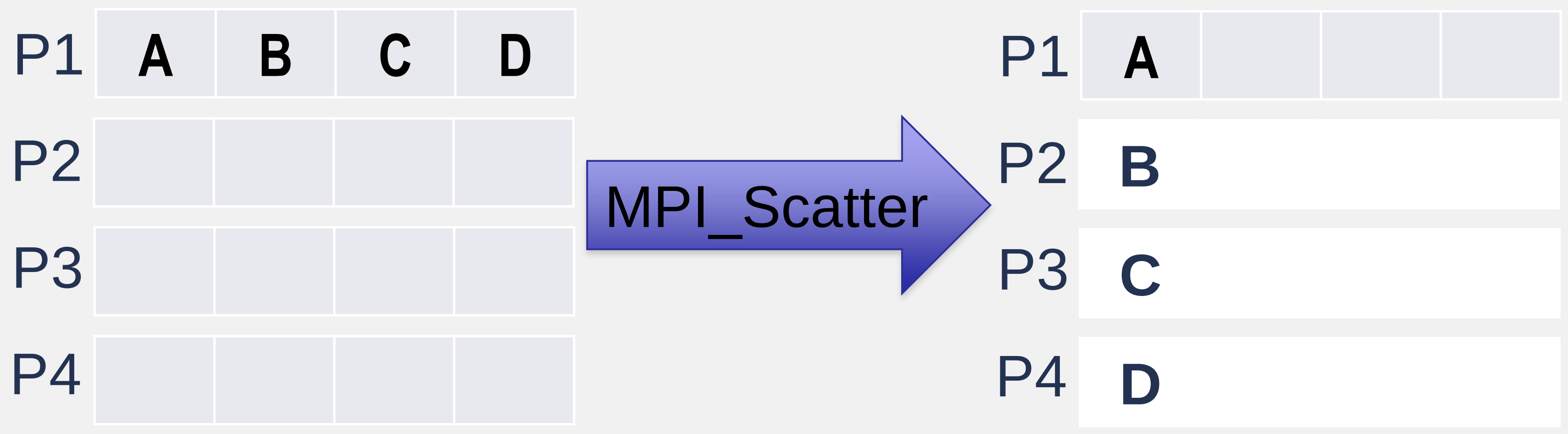
- So, when you call MPI_Bcast(buffer, count, datatype, root, comm), the root process sends the data in the buffer to all other processes in the communicator comm. All other processes receive this data. After the call, **everyone's buffer** contains the **same data**.

Example

```
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int data;
    if (world_rank == 0) {
        // Only process 0 sets the data
        data = 100;
    }
    // Broadcast data from process 0 to all other processes
    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    // Now all processes have the same value of data
    printf("Process %d received data: %d\n", world_rank, data);
    MPI_Finalize();
    return 0;}
```

Scatter

- `int MPI_Scatter(void *inbuf, int incount, MPI_Datatype sendtype, void *outbuf, int outcount, MPI_Datatype outtype,  int root, MPI_Comm comm)`
  - One process (root) scatter data to all the other processes in the same communicator
  - Must be called by all the processes with the same arguments



Scatter Parameters

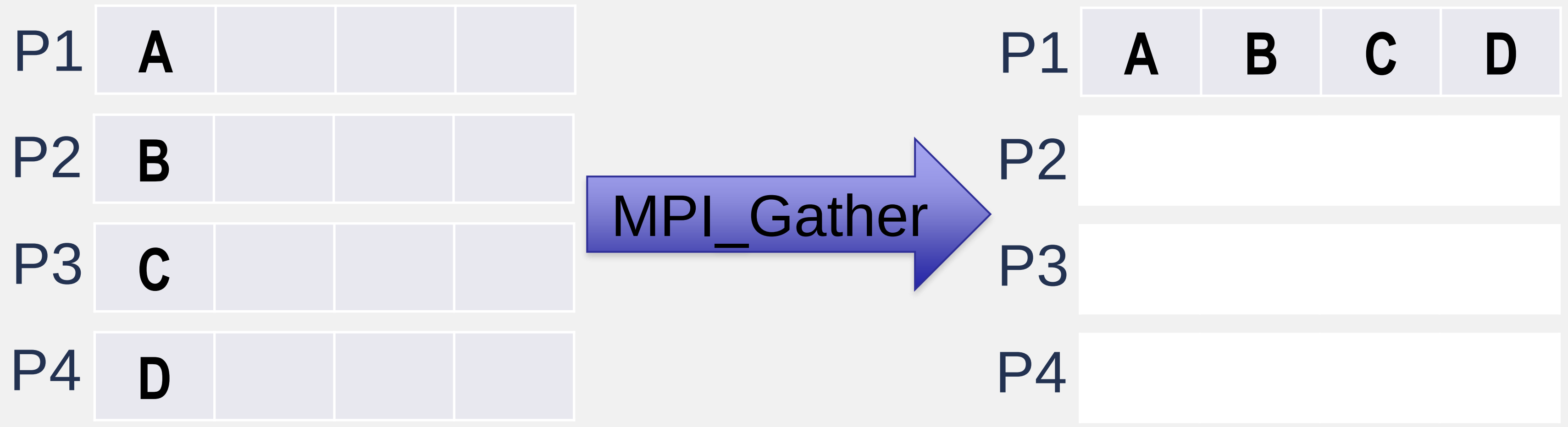
- **inbuf**: This is the address of the data you want to scatter. It's only significant at the root process.
- **incount**: It specifies the number of elements in the input buffer (inbuf).
- **sendtype**: This parameter specifies the datatype of the elements in the input buffer.
- **outbuf**: This is the address of the memory location where the received data will be stored in each process.
- **outcount**: It specifies the number of elements each process expects to receive.
- **outtype**: This parameter specifies the datatype of the elements in the output buffer. It should match the datatype of inbuf.
- **root**: This is the rank of the root process. Only the root process specifies the data in inbuf.
- **comm**: This is the communicator.
- So, when you call MPI_Scatter(inbuf, incount, sendtype, outbuf, outcount, outtype, root, comm), the root process scatters the data from its inbuf to all other processes' outbuf in the communicator comm.

Example

```c
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);
    int world_rank, world_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // Scatter data from root (rank 0) to all processes
    MPI_Scatter(sendbuf, 1, MPI_INT, &recvbuf, 1, MPI_INT, 0, MPI_COMM_WORLD);
    // Each process prints its received value
    printf("Process %d received value: %d\n", world_rank, recvbuf);
    MPI_Finalize();
    return 0;
}
```

Gather

- `int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)`
    - One process (root) collects data to all the other processes in the same communicator
    - Must be called by all the processes with the same arguments

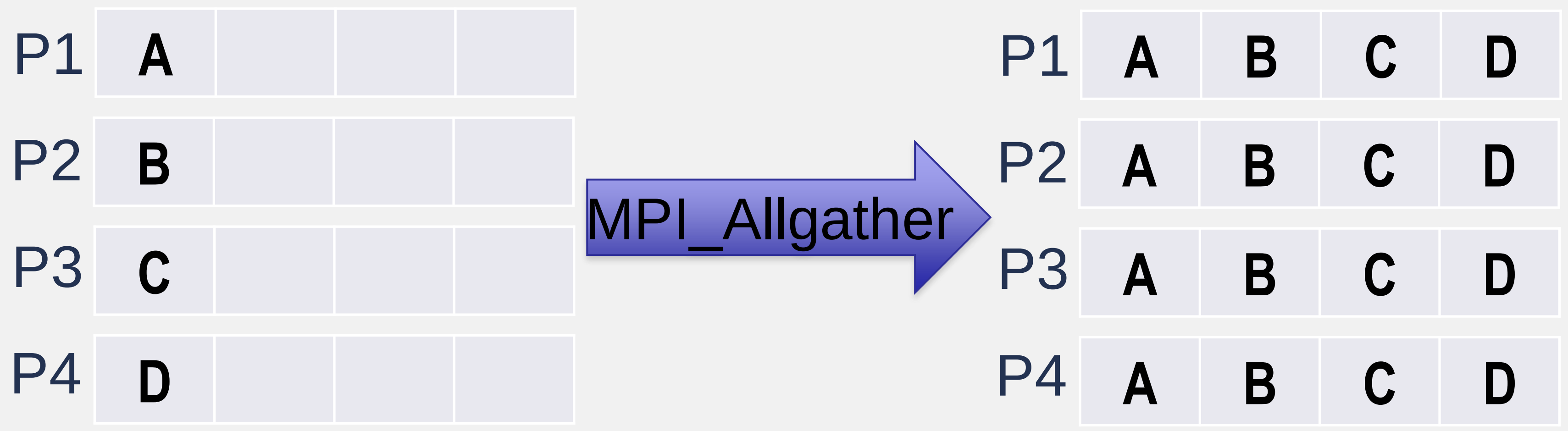


Gather Parameters

- **sendbuf**: This is the address of the data you want to send. Each process specifies its own data in this buffer.
- **sendcnt**: It specifies the number of elements each process is sending from its sendbuf.
- **sendtype**: This parameter specifies the datatype of the elements in the send buffer.
- **recvbuf**: This is the address of the memory location where the received data will be stored in the root process.
- **recvcnt**: It specifies the number of elements that each process is expecting to receive at the root process.
- **recvtype**: This parameter specifies the datatype of the elements in the receive buffer.
- **root**: This is the rank of the root process. Only the root process will receive data from all other processes.
- **comm**: This is the communicator.
- So, when you call MPI_Gather(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm), all processes send their data from sendbuf to the root process, which gathers this data into recvbuf.

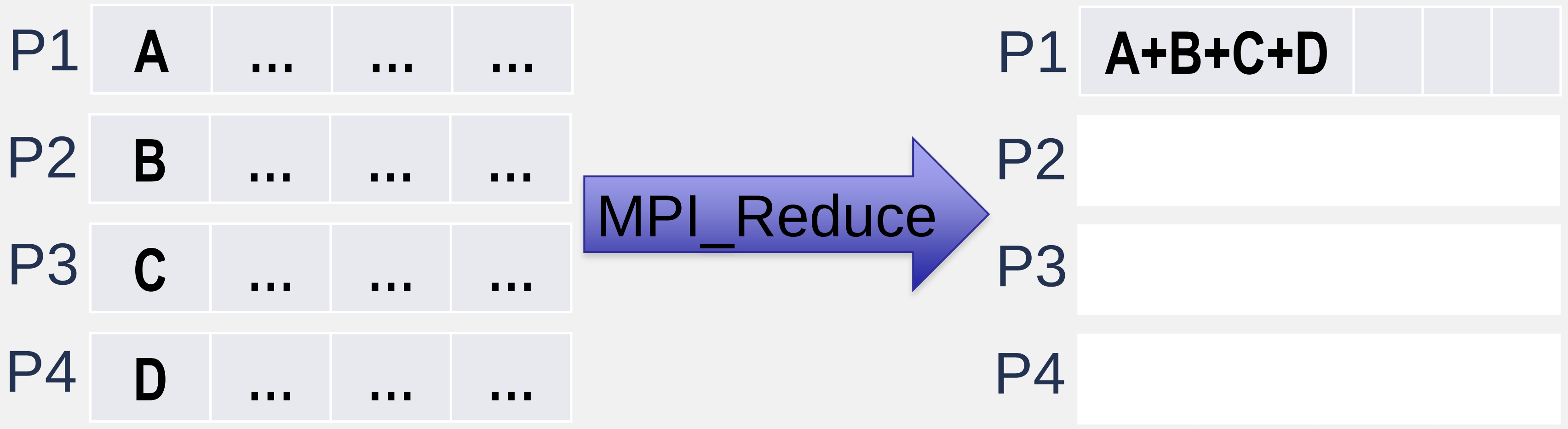
Gather to All

- `int MPI_Allgather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)`
    - All the processes collects data to all the other processes in the same communicator
    - Must be called by all the processes with the same arguments



Reduction

- Combine data from several processes to produce a single result.
- `int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
    - One process (root) collects data to all the other processes in the same communicator, and performs an operation on the data
    - MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
    - MPI_Op_create(): User defined operator

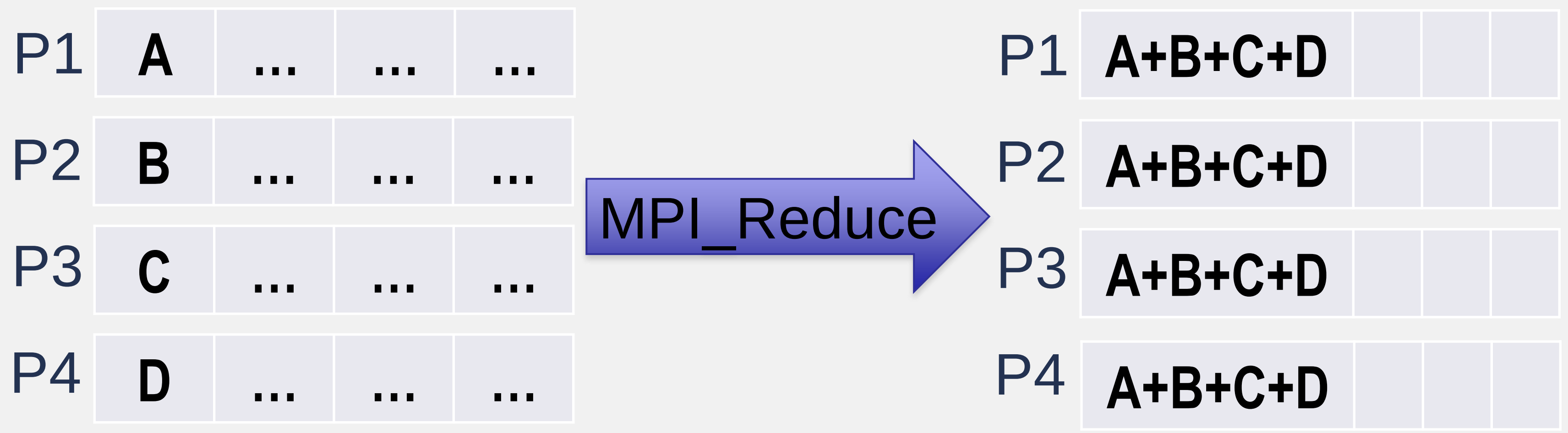


Reduce Parameters

- The MPI_Reduce function in MPI is used to combine data from all processes in a communicator and store the result at the root process.
- **sendbuf**: This is the address of the data you want to send. Each process specifies its own data in this buffer.
- **recvbuf**: This is the address of the memory location where the result will be stored at the root process. Other processes don't need to specify this parameter.
- **count**: Number of elements in the send buffer and the receive buffer.
- **datatype**: This parameter specifies the datatype of the elements.
- **op**: This is an operation that defines how data should be combined. It could be MPI_SUM, MPI_MAX, MPI_MIN, etc.
- **root**: This is the rank of the root process. Only the root process receives the result.
- **comm**: This is the communicator.
- So, when you call MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm), all processes combine their data according to the specified operation (op) and store the result in recvbuf at the root process.

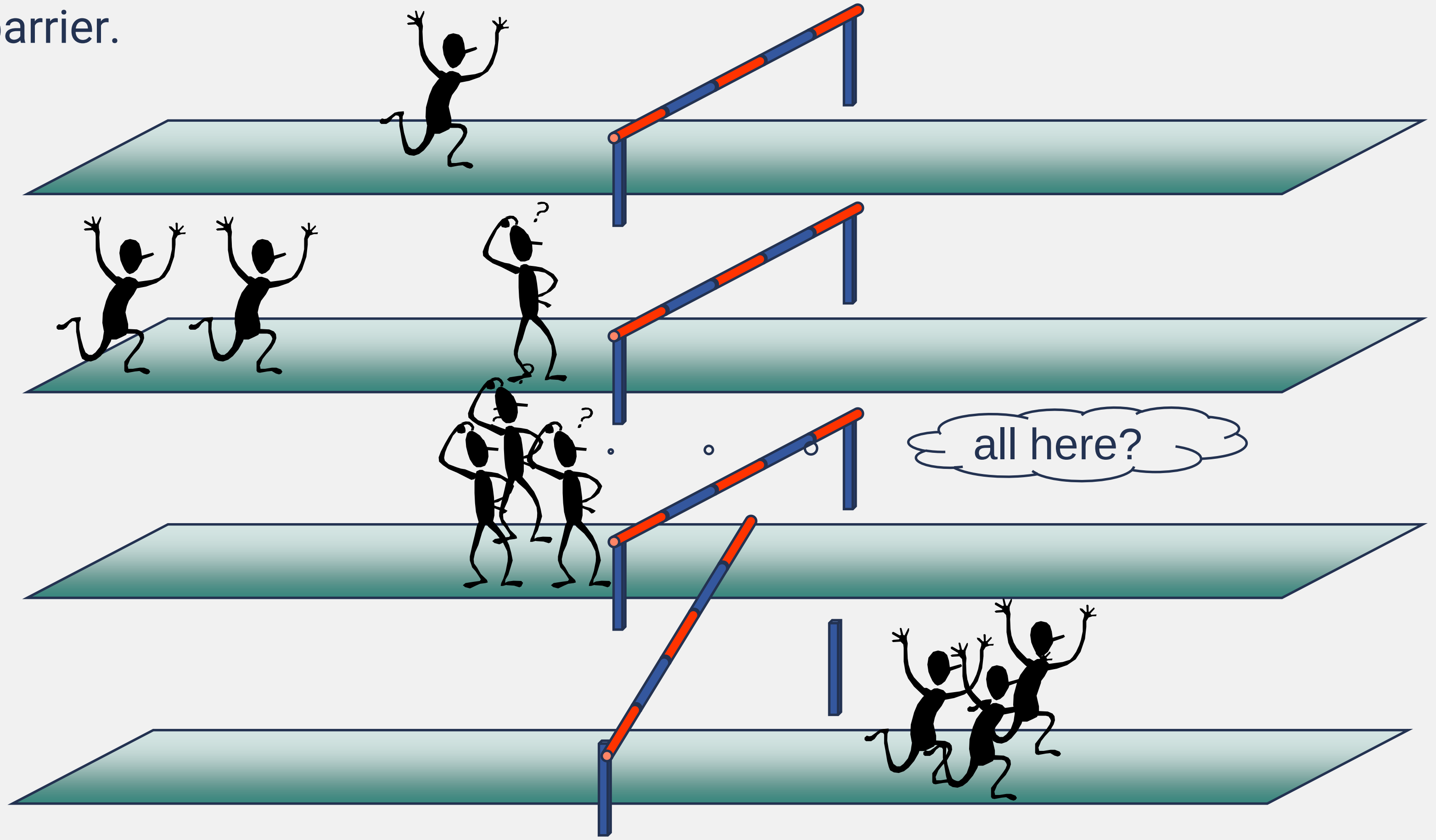
Reduction to All

- int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
    - All the processes collect data to all the other processes in the same communicator, and perform an operation on the data
    - MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
    - MPI_Op_create(): User defined operator



Barrier

- The MPI_Barrier(comm) function in MPI (Message Passing Interface) is like a checkpoint for processes in a group. All processes wait until all have arrived at the barrier.



Example Code

The following sample code illustrates MPI_Barrier.

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

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf("Hi.  I am %d of %d\n", rank, nprocs);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("How are you.  I am %d of %d\n", rank, nprocs);
    MPI_Finalize();
    return 0;
}
```