Message Passing Interface (MPI)

1 Introduction

MPI stands for Message Passing Interface. It is a standardized and portable message-passing system designed to function on a wide variety of parallel computing architectures.

2 MPI Implementation

Every MPI implementation must provide 3 components (here only C/C++):

1. **MPI library**: with *mpi.h* header, the library to write parallel programs
2. **Compiler wrapper**: *mpicc* or *mpicxx*, to compile the programs
3. **Runtime system**: *mpirun*, to run the program

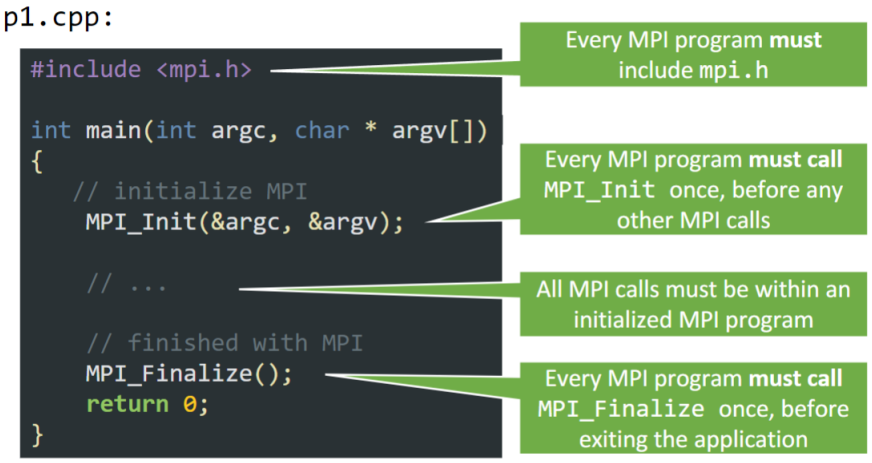

Figure 1: MPI Structure

```
$ mpicxx p1.cpp -o p1

$ mpiexec -np 4 ./p1
```

Figure 2: MPI Compiling and Running Commands

3 Communicator

A communicator is a fundamental concept that defines a group of processes that can communicate with each other. Communicators provide a context for communication and control the scope of message passing. Some characteristics:

1. In a communicator, each process has a unique rank
2. Ranks are integers from 0 to $p-1$, where p is the total number of processes in the communicator

3.1 Special Communicators

1. **MPI_COMM_WORLD**: is a communicator that includes all the processes in an MPI program. When an MPI program starts, it automatically creates this communicator. It is typically used for global communications – like broadcast, where a message from one process is sent to all processes, or global reduction operations where data from all processes are combined in some way.
2. **MPI_COMM_SELF**:is a special type of communicator that contains only one process - the process itself. This communicator is used for self-communication. While this might sound unusual, it can be useful in situations where a uniform interface for communication is needed, but sometimes the communication might be with oneself.

3.2 Communicator Operations

1. **MPI_Comm_size**: determine the size of a communicator
2. **MPI_Comm_rank**: determine the rank of a communicator
3. **MPI_Comm_split**: used for partitioning a group of processes in an existing communicator into several disjoint subcommunicators.
4. **MPI_Comm_create**: used to create a new communicator from an existing group of processes.
5. **MPI_Comm_free**: used to free up a communicator that was created using MPI communicator creation functions like MPI_Comm_create or MPI_Comm_split

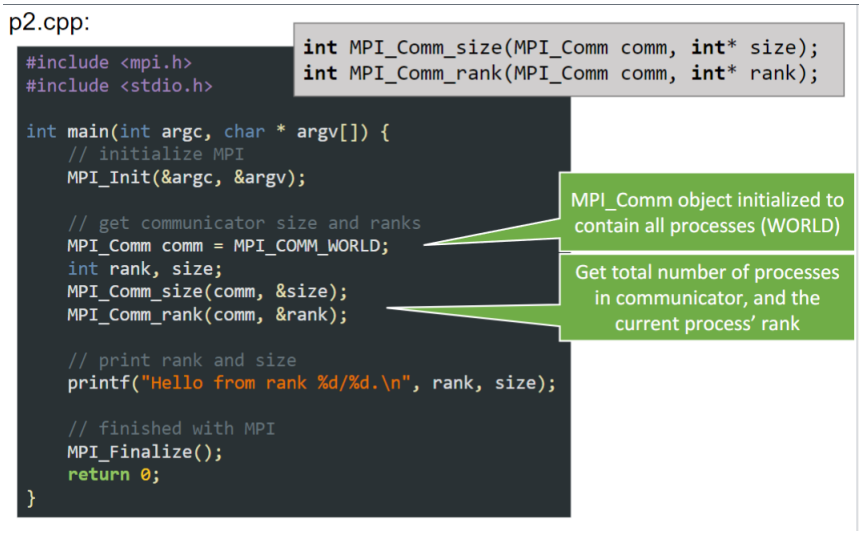

Figure 3: MPI Operations Example

4 Point to Point Communication

4.1 Overview

Point to point communication means sending messages from one process to another. This process always involves sending and receiving processes, including source and destination, as shown below:

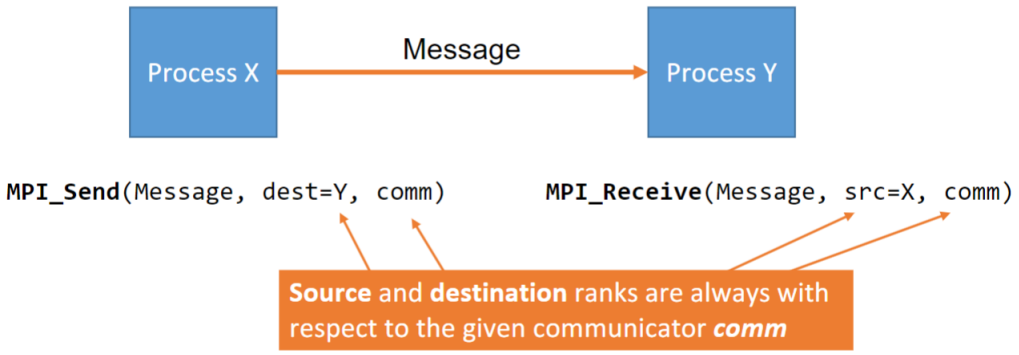

Figure 4: Overview

Every message consists of a **Message Envelope** and **Message Data**, as shown below:

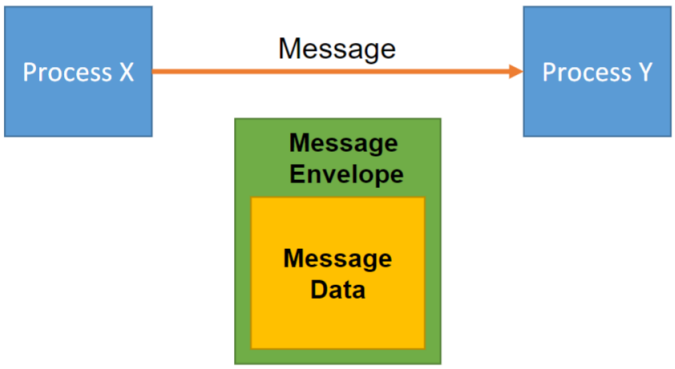

Figure 5: Message Envelope and Data

4.2 Message Data

The message data for each message is described by:

1. Memory buffer storing data: **void*** buf
2. Number of objects in the buffer: **int** count
3. Data type of the data in the buffer: **MPI_Datatype**: type

Notice here MPI defines **MPI_Datatype** for all built in C/C++ types. The message data is described as a **block of memory**, where **buf** is a pointer to where the data lies in memory, **count** and **type** define **how large** the memory block is:

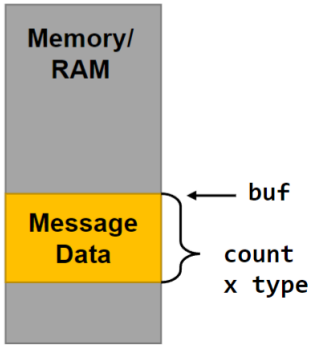

Figure 6: Message Data in Memory

4.3 Message Envelope

The message envelope contains:

1. Source rank: **int** src
2. Destination rank: **int** dest
3. Tag: **int** tag, integer used to distinguish messages
4. Communicator: **MPI_Comm comm**, universe of communication

4.4 MPI_Send/MPI_Recv

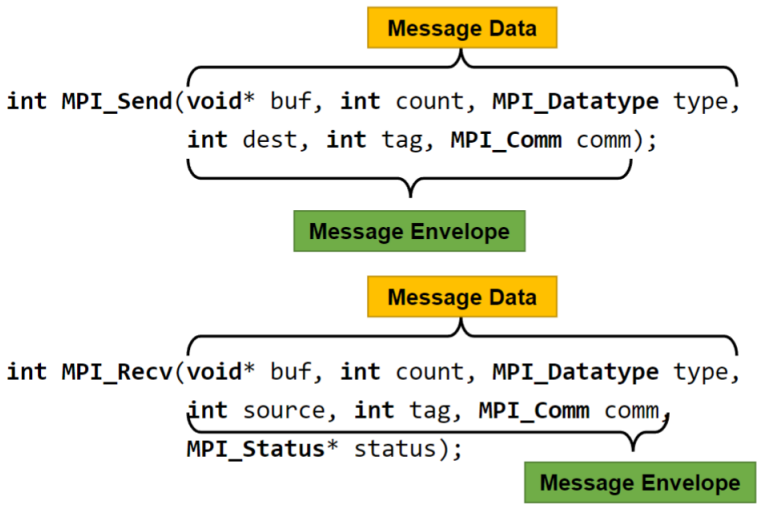

Figure 7: Send/Receive Functions

p3.cpp:

```cpp
#include <mpi.h>
#include <iostream>

int main(int argc , char* argv []) {
    int rank;
    MPI_Init (&argc , &argv);
    MPI_Comm_rank (MPI_COMM_WORLD , &rank );
    const int N = 32;
    int tab[N];
    if (rank == 0) {
        tab[N - 1] = 13;
        MPI_Send(tab, N, MPI_INT,
                 1, 111, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status stat;
        MPI_Recv(tab, N, MPI_INT,
                 0, 111, MPI_COMM_WORLD, &stat);
        std::cout << tab[N - 1] << std::endl;
    }
    return MPI_Finalize();
}
```

Send N integers from rank 0 to rank 1.
Blocks till everything is sent.

Receive N integers.
Blocks till everything is received.

```
Output: 13
```

Figure 8: Send/Receive Example

5 Message Matching

5.1 Overview

Each send operation must be matched by a corresponding receive operation. This matching is via the envelope, including:

1. Communicator
2. Source rank
3. Tag

4. **Not by type or size**

5.2 Blocking

Another concept is **blocking** in parallel computing. Blocking refers to a situation where the execution of certain processes or tasks is delayed due to the need to synchronize or communicate with other processes. An **MPI_Recv** operation blocks until a matching message is received. An **MPI_Send** operation may block until the message has been received by the destination process. An example is shown below:

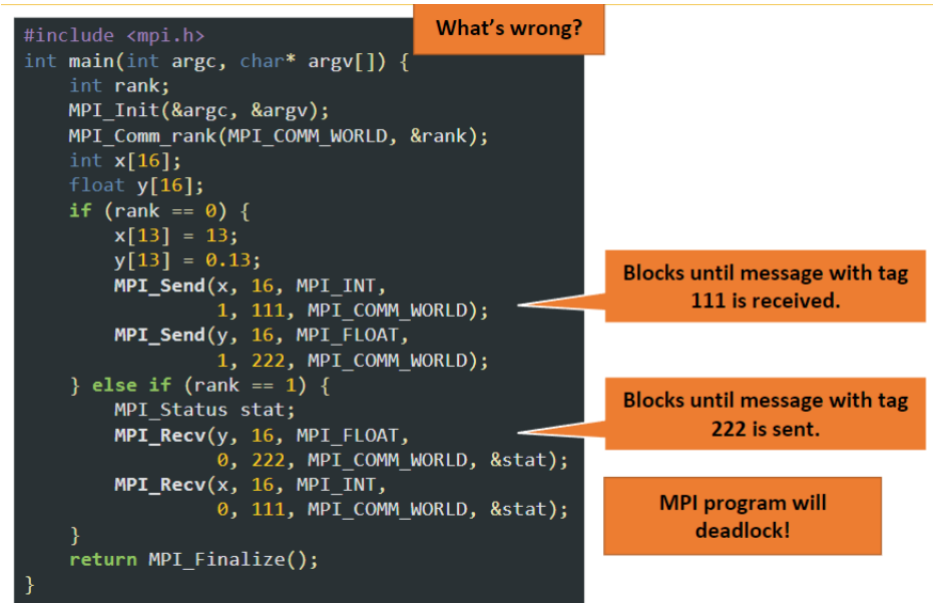

Figure 9: Blocking Example

5.3 ANY

We can use **ANY** in MPI_Recv function to receive any message from any processor. And we can use MPI_Status structure to check envelope of the received message.

```
int x;
MPI_Status stat;
MPI_Recv(&x, 1, MPI_INT,
         MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
int source = stat.MPI_SOURCE;
int tag = stat.MPI_TAG;
```

Figure 10: ANY Example 1

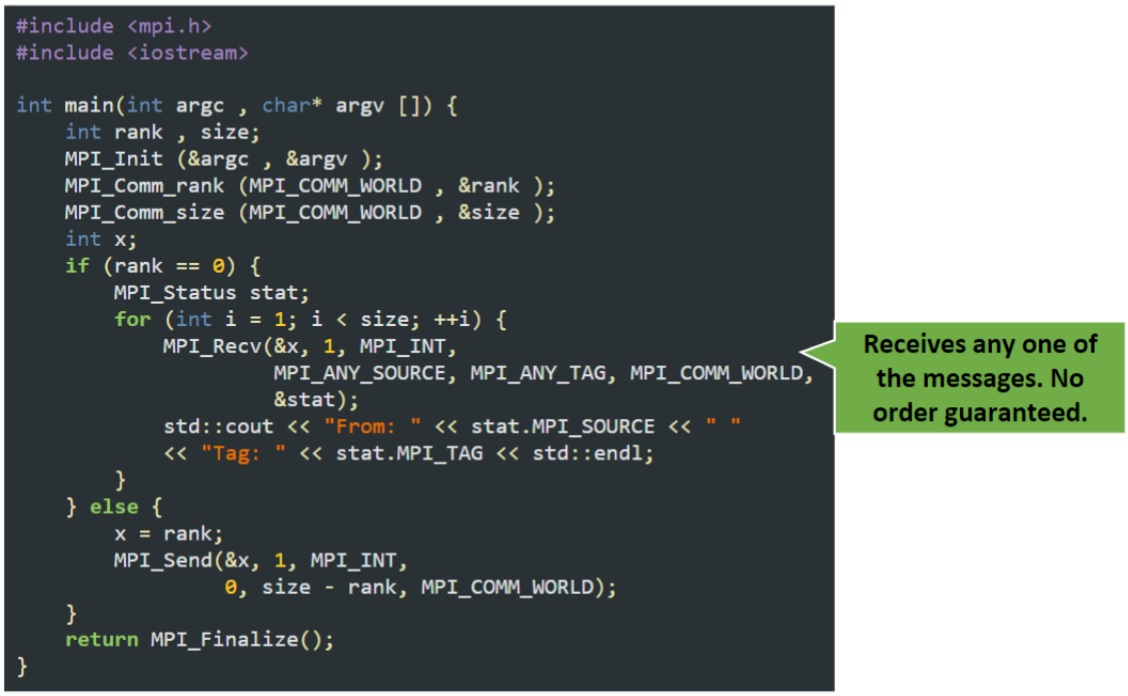

Figure 11: ANY Example 2

5.4 Message Ordering

There are three main rules for message ordering:

1. If messages originate from **different processors**, the order in which they are received is **arbitrary**.
2. If messages have **different tags**, the order in which they are received is **arbitrary**
3. Only if two or more messages have the **same source AND the same tag**, their order is preserved. They are received in the same order in which they were sent.

Suppose we have the processors in cyclic dependencies:

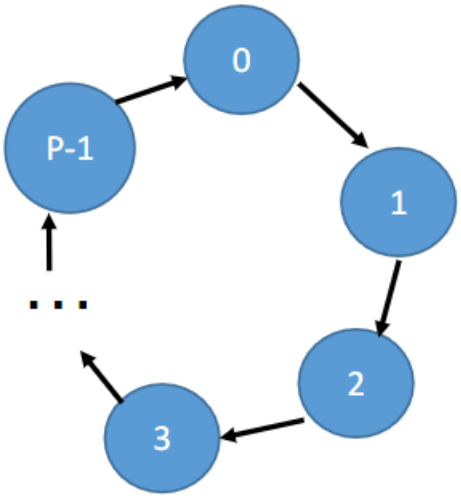

Figure 12: Cyclic Dependencies

Now we want to send messages according to right shift permutation. Recall the formula:

$$i \to (i+1) \bmod p \tag{1}$$

Then the blocking will cause the issue:

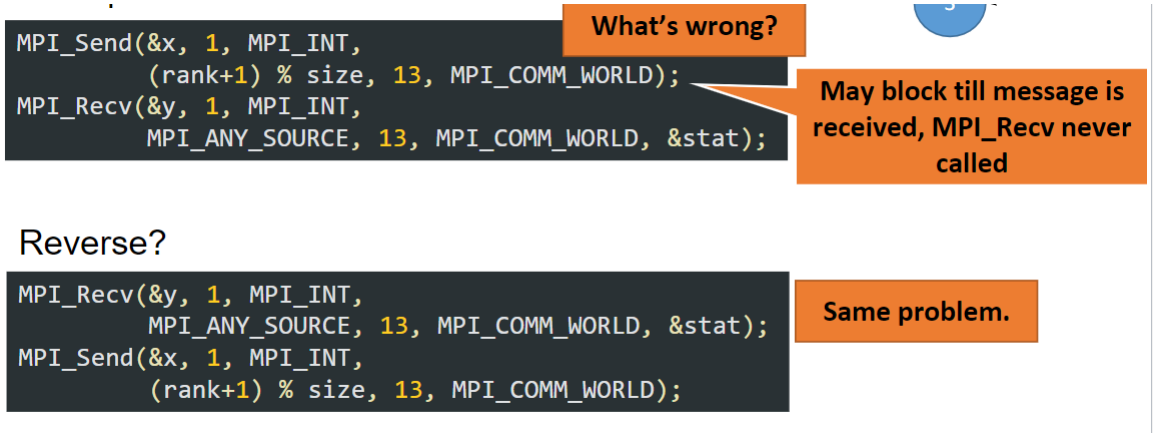

Figure 13: Blocking Issue

Notice that here using **MPI_ANY_SOURCE** is valid, because in the sending function the destination is already specified. However, in cyclic dependencies, all the processors need to send the message, the receive function may not be called at all. Therefore in this case, we need **non-blocking communication**, which is realized by functions **MPI_Isend** and **MPI_Irecv**:

```
int MPI_Isend(void* buf, int count, MPI_Datatype type,
              int dest, int tag, MPI_Comm comm,
              MPI_Request* req);

int MPI_Irecv(void* buf, int count, MPI_Datatype type,
              int source, int tag, MPI_Comm comm,
              MPI_Request* req);
```

Figure 14: Isend and Irecv funtions

And two other functions could be used to check for communication status:

```
int MPI_Wait(MPI_Request* req, MPI_Status* status);
int MPI_Test(MPI_Request* req, int* flag, MPI_Status* status);
```

Figure 15: Status Checking

MPI_Wait will block till the send/receive operation is completed. **MPI_Test** returns a flag of either 0 or 1 depending on whether the send/receive operation is completed. Now the cyclic dependencies could be implemented as:

Using Non-blocking receive:

Correct!

```
MPI_Request req;
MPI_Irecv(&y, 1, MPI_INT,
          MPI_ANY_SOURCE, 13, MPI_COMM_WORLD, &req);
MPI_Send(&x, 1, MPI_INT,
          (rank+1) % size, 13, MPI_COMM_WORLD);
MPI_Wait(&req, &stat);
```

Figure 16: Correct Example