

Q1. Which are the four different modes of MPI send? Describe the behaviour of a buffered MPI send.

Ans. The four different modes of MPI Send are:

- 1) **Standard (MPI_Send)**: This is the most commonly used send mode. After executing standard send, the sending process is blocked until the data to be sent is transferred to the internal buffer of the sender or transferred to the receiver.
- 2) **Synchronous (MPI_Ssend)**: In this mode, the sending process is blocked till the receiving process starts receiving the data. The moment the destination starts receiving, it signals an *ACK* to the source.
- 3) **Buffered (MPI_Bsend)**: This mode is quite similar to Standard send except that a user-specified buffer is used instead of the standard buffer allotted by the operating system. This mode can be useful if the sender has a large amount of data to send which may not fit in an internal buffer: i.e.; if the data to be sent is too large to fit in internal buffer, then storing it in a larger user-defined memory space and sending it in one go can be better for performance as it reduces the number of communications.
- 4) **Ready mode (MPI_Rsend)**. This is seldom used. It only works if a receiving process has already called the corresponding MPI_Recv(..) for a particular send. Otherwise, it works just like the standard send.

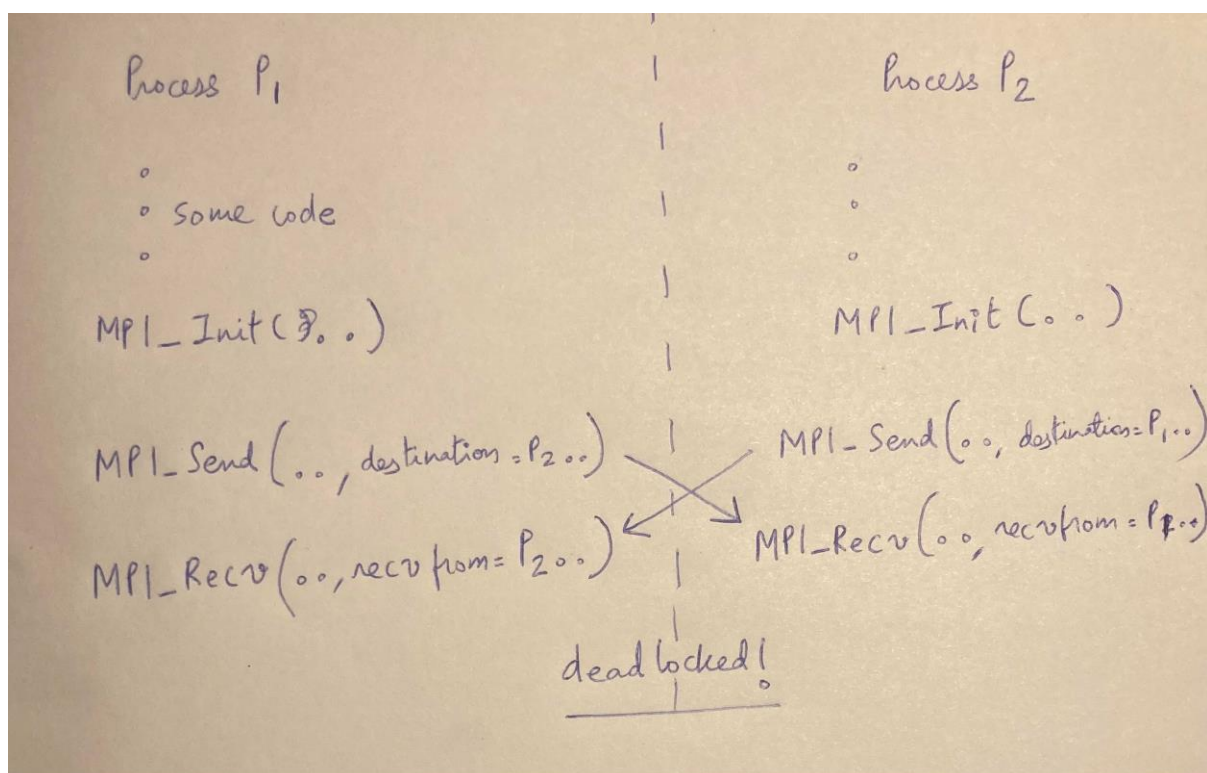
The 4 mentioned above are for the blocking version of MPI_Send (). These 4 modes exist for the non-blocking version as well:

- 1) **MPI_Isend**: It is the non-blocking version of MPI_Send. When this function is called the function returns immediately but runs MPI_Send actions in the background of the process. Therefore, After the function returns, the data **must not** be modified unless MPI_Test and MPI_Wait confirm MPI_Isend is completed. After the completion, the data is reusable because it either is buffered in MPI or sent to the destination.
- 2) **MPI_Issend**: It is the non-blocking version of synchronous send. It returns immediately, but runs MPI_Ssend actions in the background. MPI_Test or MPI_Wait **must** be used to assess if the function is completed in the background. At that point, not only the message has been sent but also the destination has started to receive the message.

- 3) **MPI_Ibsend:** This is the local non-blocking send. It blocks for neither copying the message to the buffer nor sending the message. After the test positive or wait, we can modify the source data because, if it is not sent, it is locally copied to the allocated buffer.
- 4) **MPI_Irsend:** Non-blocking version of MPI_Rsend

Q2. What is a deadlock? Give an example/situation where using MPI_Send and MPI_Recv leads to a deadlock.

Ans. In general computing, a deadlock is a situation where two different programs or processes depend on one another for completion. For example, if P1 and P2 are two processes, and both are waiting for the others to access resource R, then P1 and P2 are deadlocked. This is because P1 waits for P2 to access R and P2 is also waiting for P1 to access R, so both are mutually waiting for the other! An example of this occurring in MPI programming is shown below:



#NOTE: The above example assumes that the MPI implementation for send blocks until the intended receiving process executed receive. This may not be the case always depending on the MPI implementation.

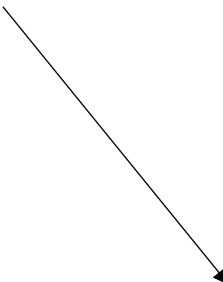
Q3. In what situation will the use of asynchronous communication be advantageous? Is there extra complication for your parallel program?

Ans. The advantage of asynchronous communication is that the functions are non-blocking, allowing for more overlap of computation and communication. A situation where this could be useful is when a process needs to transfer a large amount of data. In a blocking scenario, the process would have to wait until all this data has been copied into the internal buffers (standard blocking send) before proceeding to carry out further computations. A better option would be to allow the process to carry out the additional computations during this waiting period, instead of stalling.

However, there are some precautions to take note of when using asynchronous communication. Non-blocking calls return immediately after initiating the communication. The programmer does not know at this point whether the data to be sent have been copied out of the send buffer, or whether the data to be received have arrived. So, before using the message buffer, the programmer must check its status. This is called **race condition**. In such a scenario, the programmer can choose to block until the message buffer is safe to use, by using a call to `MPI_Wait()` or to just return the current status of the communication by using `MPI_Test()`. The following code shows how the race condition can be handled in MPI.

CODE SNIPPET FOR HANDLING RACE CONDITION

```
int i = 123;
MPI_Request myRequest;
MPI_Isend(&i, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD, &myRequest);
// do some calculations here
// Before we re-use variable i, we need to wait until the asynchronous function call is complete
MPI_Status myStatus;
MPI_Wait (&myRequest, &myStatus);
i = 234;
```



Notice that `MPI_Wait()` is called before modifying the buffer. This is because the sender does not know whether the earlier data has been successfully copied or not.