PMC Lecture 05

Gianmaria Romano

8 October 2025

Chapter 2

Message passing interfaces

2.1 Handling messages

When implementing a message passing interface, processes can communicate via message passing by using dedicated functions that coordinate both the sender and the receiver.
In particular, message passing interfaces require messages to be non-overtaking, which means that, if process q sends two messages to another process r, then the first message that was sent first by q should be available to r before the second one.

2.2 Message matching

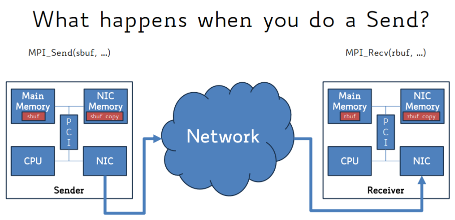
When two processes communicate over a message passing interface, a message will be successfully received when all of the following conditions are satisfied:

- The source and destination ranks match between sender and receiver.
  If the sender is unknown, the receiver can refer to a wildcard that allows it to receive a message without setting a constraint on the sender's rank: whenever this happens, the receiver can recover the rank of the sender from the "status" structure of the receiving function.
- The sender and the receiver refer to the same communication tag.
  If the tag is unknown, the receiver can refer to a wildcard that allows it to receive a message without setting a constraint on the sender's rank: whenever this happens, the receiver can recover the tag of the sender from the "status" structure of the receiving function.
- The sender and the receiver agree on the data type contained in the message.
- The receiving buffer is larger than the sending buffer.

Generally speaking, however, the message matching procedure may behave differently depending on the implementation of the message passing interface.
For this reason, when working with a message passing interface, it is more convenient to coordinate message matching according to the default standard to preserve portability.

2.2.1 Hanging processes

When dealing with message passing, a process is said to ”hang” if it stops functioning and does not respond to user inputs.
Generally speaking, a sending process will hang if its call blocks and there is no matching receiver: if, under this condition, the call is then buffered, the message will be lost.
On the other hand, a receiving process will hang if it tries to receive a message but there is no matching sender.



For this reason, when implementing a message passing interface, it is a good practice to define as few senders as possible.

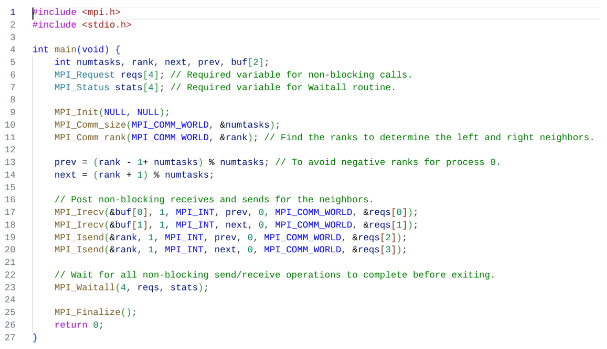
2.3 Communication modes

When sending a message, the sender typically uses the standard communication mode, which lets the process choose whether to block globally, meaning that the sender will block until the receiver collects the message, or locally, meaning that the sending call will return before a matching receiver is found.
Alternatively, in the context of blocking communication, the sender can also use one of the following communication modes:

- **Buffered:** the sending call is always locally blocking as it will return before the message is copied in a user-provided buffer.
- **Synchronous:** the sending call is always globally blocking as it will return after the receiver has started retrieving the message.
- **Ready:** the sending call is successful if and only if the receiver has already started retrieving the message.

Generally speaking, this communication mode comes in handy for reducing the communication overhead.

However, since buffered communication is typically considered to be inefficient, most programs now opt for non-blocking communication modes that maximize concurrency by letting communication and computation to overlap: this strategy is faster, but it requires explicit control of operations to let the sender re-use or modify the buffer and to allow the receiver to extract the message.
Example: The following code provides a non-blocking implementation of a ring topology where each process sends and receives something from its left and right neighbours:



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

int main(void) {
    int numtasks, rank, next, prev, buf[2];
    MPI_Request reqs[4]; // Required variable for non-blocking calls.
    MPI_Status stats[4]; // Required variable for Waitall routine.

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Find the ranks to determine the left and right neighbors.

    prev = (rank - 1+ numtasks) % numtasks; // To avoid negative ranks for process 0.
    next = (rank + 1) % numtasks;

    // Post non-blocking receives and sends for the neighbors.
    MPI_Irecv(&buf[0], 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf[1], 1, MPI_INT, next, 0, MPI_COMM_WORLD, &reqs[1]);
    MPI_Isend(&rank, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD, &reqs[3]);

    // Wait for all non-blocking send/receive operations to complete before exiting.
    MPI_Waitall(4, reqs, stats);

    MPI_Finalize();
    return 0;
}
```