PMC Lecture 04

Gianmaria Romano

7 October 2025

Chapter 2

Message passing interfaces

2.1 The main aspects message passing interfaces

Implemented for multiple-instruction-multiple-data structures with a distributed memory, a message passing interface is a library that is commonly described as a "single-program-multiple-data" computational model because it compiles one program that gets executed by multiple processes, whose individual functionalities are specified using selective branches with respect to their ranks.
Example: The following code, which makes use of the "mpi.h" library, prints "Hello, World!" using a message passing interface:

```c
#include <stdio.h>
#include <mpi.h> // This library is needed to implement message passing interfaces.

int main(void) {
    MPI_Init(NULL, NULL); // MPI_Init() sets up the program.
    printf("Hello, World!\n");
    MPI_Finalize(); // MPI_Finalize() terminates the program and provides its exit code.
    return 0;
}
```

2.2 Grouping processes in communicators

In the context of message passing interfaces, the term "communicator" refers to a collection of processes that can send messages to each other.
Generally speaking, a message passing interface always creates a "communicator world" that will contain all the processes that are created when the corresponding program is started.
Most particularly, if more communicators are created, each process will have different IDs relative to each communicator.
Example: The following code makes use of a communicator that allows four processes to print "Hello, World!":

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

int main(void) {
    int comm_sz, my_rank;
    MPI_Init(NULL,NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz); // MPI_Comm_size() returns the number of processes in the communicator.
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // MPI_Comm_rank() returns the rank of the caller process.
    printf("Hello, World - From process %d out of %d \n", my_rank, comm_sz); //
    MPI_Finalize();
    return 0;
}
```

Observe that, since process scheduling will be non-deterministic, it might be the case that these processes do not print their messages in order.
However, it is possible to handle this issue by calling some functions that deal with message passing, as shown in the following code:

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

const int MAX_STRING = 100;

int main(void) {
    char greeting[MAX_STRING];
    int comm_sz;
    int my_rank;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Here, process 0 will receive and print all the greetings.
    // On the other hand, the other processes will send a greeting to process 0.
    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d out of %d!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting) + 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
}
```

2.3 Handling messages

When implementing a message passing interface, processes can communicate via message passing by using dedicated functions to 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 by q should be available to r before the second one is.
N.B.: The constraint on messages being non-overtaking does not necessarily apply whenever the messages are sent from different processes.