Introduction to Parallel Programming with MPI

Lecture #2*: Point to Point Communications*

Andrea Mignone[1]

[1]Dipartimento di Fisica - Turin University, Torino (TO), Italy

Data Communication

- So far, each process has performed an independent task.
- Data communication between different processes is, however, a key point for employing MPI.
- In MPI, we distinguish between two main classes of communication:

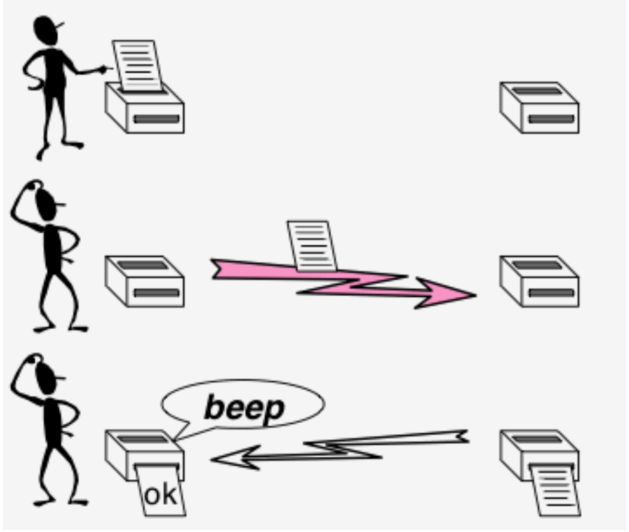
  - *Point to point (P2P)*: one process sends a message to another one. This is the simplest form of message passing (like email exchange). There're two different P2P communications:

    - **Synchronous** (or also blocking) such as `MPI_Send()` and `MPI_Recv()`;
    - **Asynchronous** (non-blocking), such as `MPI_Isend()` and `MPI_Irecv()`.

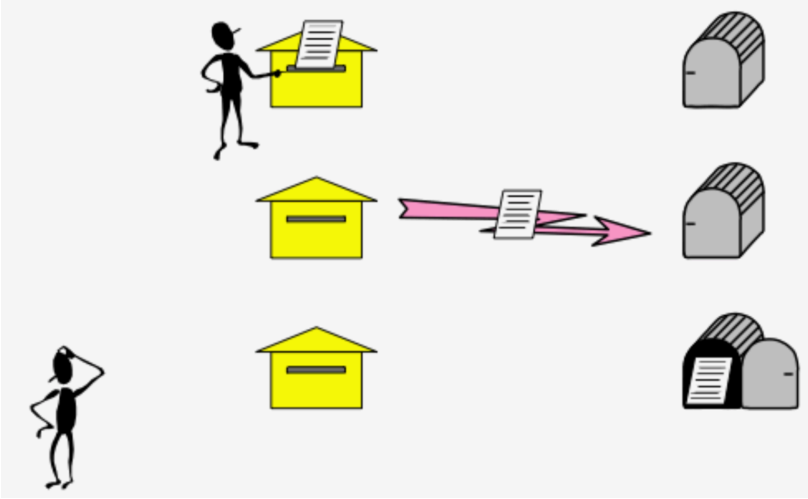
  - *Collective*: composed of several point-to-point operations, e.g., one-to-all or all-to-one such as `MPI_Broadcast()`, `MPI_Reduce()`, `MPI_Barrier()` and others.

Point to Point Communications

- Blocking (Synchrounous) calls:
  - Message arrives at the same time as it is sent;
  - Sender has to wait if receiver is not ready
  - Sender gets confirmation of receipt
  - Analogy: fax-machine



- Non-Blocking (Asynchrounous) calls:
  - Message arrives whenever receiver is ready;
  - Sender does not have to wait;
  - Sender only knows that message has left;
  - analogy: postal mail



Blocking Point to Point Communication

- The most basic forms of P2P communication are called **blocking:** the process that sends a message will be waiting until the process that receives has finished receiving all the information.
- This is the easiest form of communications but not necessarily the fastest one.
- Communication occurs between two processors in the same communicator.
- The **source** process sends a message with a certain **length** and **data type** to the **destination** process. Perhaps, a good place to start is the `MPI_Send()` function:

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

MPI_Send()

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

- `*buf`: pointer to a memory buffer (e.g. an array) containing the data that you wish to send from the current process to another;
- `count`: the number of elements in the buffer;
- `datatype`: the kind of data that we are sending (e.g. char, float, double, and so forth). MPI has predefined data-types that must correspond precisely to the data stored in the buffer (see next slide);
- `dest`: the rank of the destination process;
- `tag`: an integer that identifies the "type" of communication (this is an informal value: think of it as an email subject; allows the receiver to understand what type of data is being received);
- `comm`: the communicator on which to send the data to.

MPI DataTypes

- MPI datatypes can be defined as:
  - predefined: corresponds to a data type inherited from the language, e.g.

MPI Datatype	C correspondence
`MPI_CHAR`	`char`
`MPI_INT`	`int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG`	`long`

  - A contiguous array of MPI datatypes;
  - An indexed array of blocks of datatypes;
  - An arbitrary structure of datatypes;

- MPI provides functions to construct custom datatypes.

MPI_Send() and MPI_Ssend()

- Both `MPI_Send()` and `MPI_Ssend()` are blocking calls. There's a small difference between the two:

  - With a regular `MPI_Send()`, the implementation will return to the application when the buffer is available for reuse: this could be before the receiving process has actually posted the receive: for instance, it could be when a small message has been copied into an internal buffer and the application buffer is no longer needed. However, for large messages that may not be buffered internally, the call may not return until enough of the message has been sent to the remote process that the buffer is no longer needed.

  - `MPI_Ssend()`, on the other hand, will always wait until the receive has been posted on the receiving end. Even if the message is small and can be buffered internally, it will still wait until the message has started to be received on the other side.

Receiving Data with `MPI_Recv()`

- Receiving data is very similar to sending them.
- The corresponding basic function used for blocking P2P communication is

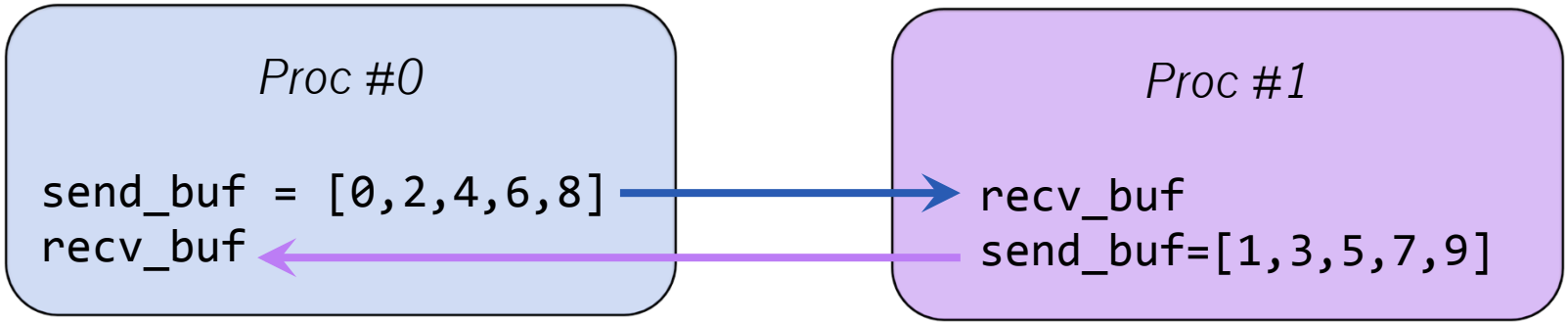
```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source,  int tag, MPI_Comm comm, MPI_Status *status)
```

- The arguments to `MPI_Recv()` match those used by `MPI_Send()`
  - `*buf`: the receiving buffer
  - `count`: the number of elements in the buffer;
  - `datatype`: the kind of data that we are receiving;
  - `source`: the rank of the sending process;
  - `tag`: should match the source tag;
  - `comm`: the communicator used;

- One additional argument is present, `MPI_Status`: this is a structure that containing information on the message you just received.

Example #1: odd_even.c

- Let's write a program where process `#0` creates a table of even numbers (e.g. 0,2,4,6,8) while process `#1` creates a table of odd numbers (1,3,5,7,9).
- Now we want these processes to exchange the information using `MPI_Send()` and `MPI_Recv()`:



- This program involves only 2 processor and should be executed with

```
> mpirun -np 2 ./my_program          # run on with 2 processors
```

- We will consider 3 different versions of this program.

Example #1: odd_even.c

- The beginning part of the program will be the same for all versions.
- Here we initialize the MPI environment and create buffers containing either even or odd (integer) numbers, arranged in arrays:

```
#include <mpi.h>
#include <stdio.h>
#define NELEMENTS 5

int main(int argc, char ** argv)
{
  int i, rank;
  int send_buf[NELEMENTS], recv_buf[NELEMENTS];
  MPI_Request req;

  MPI_Init(&argc, &argv); // Initialize the MPI execution environment
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* -- Create buffers of odd or even number -- */

  for (i = 0; i < NELEMENTS; i++) send_buf[i] = 2*i + rank;
```

- Note that `rank = 0` for process #0 while `rank = 1` for process #1.

odd_even.c: version 1

- In the first version we will arrange the calls in the following way:

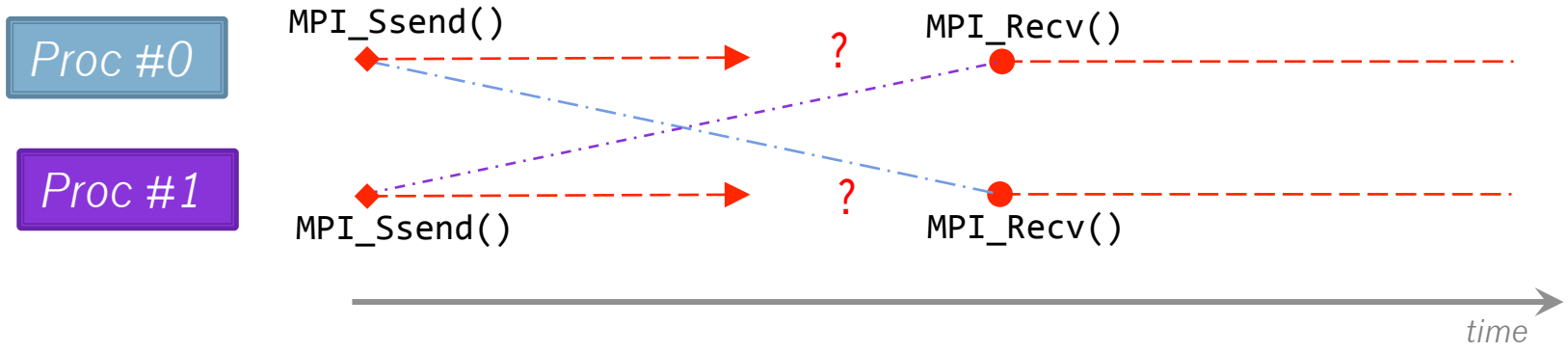
```
if (rank == 0){
  MPI_Ssend(send_buf, NELEMENTS, MPI_INT, 1, 0, MPI_COMM_WORLD);
  MPI_Recv (recv_buf, NELEMENTS, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

if (rank == 1){
  MPI_Ssend(send_buf, NELEMENTS, MPI_INT, 0, 0, MPI_COMM_WORLD);
  MPI_Recv (recv_buf, NELEMENTS, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

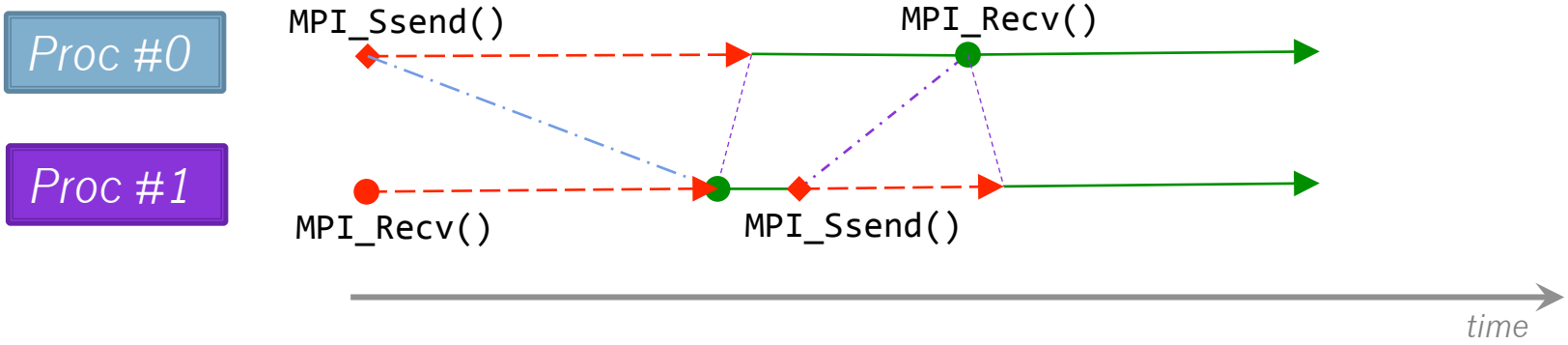
- Here `MPI_STATUS_IGNORE` tells that the status fields are not to be filled in.
- This version of the code has, however, a problem: may run into a *deadlock* or *race condition*:
  - `MPI_Send()` (or `MPI_Ssend()`): does not return until buffer is emptied so the process will be blocked until then;
  - `MPI_Recv()` does not return until buffer is full: process will be blocked until then.

Deadlocks: Attention!

- A blocking send will complete when receive is done on the destination process. In our case:



- In practice neither `MPI_Recv()` is never reached since `MPI_Ssend()` calls cannot complete in this way.
- A possible remedy is to **reverse the order** of one of the send/receive pairs:



odd_even.c:version 2 (with MPI_Sendrecv())

- A second possibility is to use `MPI_Sendrecv()` which is a combination of both send and receive process:

```
int MPI_Sendrecv(void* send_buf, int send_count, MPI_Datatype send_type, int dst, int send_tag,
                 void* recv_buf, int recv_count, MPI_Datatype recv_type, int src, int recv_tag,
                 MPI_Comm comm, MPI_Status* status);
```

- `send_buf`: buffer to send.
- `send_count`: number of elements to send.
- `send_type`: type of one send buffer element.
- `dst`: The rank of the recipient MPI process.
- `send_tag`: tag to assign to the send message.
- `recv_buf`: buffer in which receive the message.
- `recv_count`: number of elements to receive.
- `recv_type`: type of one receive buffer element.
- `src`: rank of the sender MPI process.
- `recv_tag`: tag to require from the message.
- `comm`: the communicator in which the send receive takes place.
- `status`: the variable in which store the reception status returned.

odd_even.c: version 2

- Using `MPI_Sendrecv()`, we can replace the previous lines with

```
int dst = 1 - rank;
MPI_Sendrecv(send_buf, NELEMENTS, MPI_INT, dst, 0,
             recv_buf, NELEMENTS, MPI_INT, dst, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
```

- Here process #0 is sending to *and* receiving from the same process #1 (and viceversa).
- Note that these are still blocking calls.

Asynchronous (non-blocking) Communications

- Non-blocking operations return immediately and allow the calling program to continue;
- Computations can proceed while communication can take place in the background;
- One must of course wait for the communication to complete before you need the new data.
- The code becomes slightly more complex.
- The prototype functions to be used in this case are `MPI_Isend()` and `MPI_Irecv()`:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dst,
              int tag, MPI_Communicator comm, MPI_Request *request)
```

- `MPI_Irecv()` shares the same prototype.
- Notice the new argument, `MPI_Request`.

Asynchronous (non-blocking) Communications

- `MPI_Isend()` is preparing a request but it does not actually transfer any data.
- The request is going to be executed when both processes are ready to synchronize.
- The new argument - `MPI_Request` - is a handle on a non-blocking operation, used by `MPI_Wait()` or `MPI_Test()` (or similar) to know when the non-blocking operation handled completes.
- All non-blocking operations *must have a matching wait operation* (some system resources can only be freed after the operation has completed).
- A non-blocking operation that is immediately followed by a wait is equivalent to a blocking operation.
- `MPI_Wait()` will force the process to go in blocking mode until the request is fulfilled.
- `MPI_Test()`: checks if the request can be completed. If it can, the request is automatically completed and the data transferred.

Non-Blocking Calls: An example

- Let's consider two processes, waiting and testing on only one request:

```
if (rank == 0) {

  MPI_Isend(...)

  // do dome work here
  while (has_work) {
    do_work();

    // We only test if the request is not already fulfilled
    if (!request_complete)
       MPI_Test(&request, &request_complete, &status);
  }

  // No more work, wait for the request to be complete if it's not the case
  if (!request_complete) MPI_Wait(&request, &status);

} else {

  MPI_Irecv(...);

  // Here we just wait for the message to come
  MPI_Wait(&request, &status);

}
```

odd_even.c: version 3

- Let's devise a 3rd version of odd_even.c using asynchronous communication;
- Using `MPI_Isend()` and `MPI_Recv()` we can replace the previous lines with

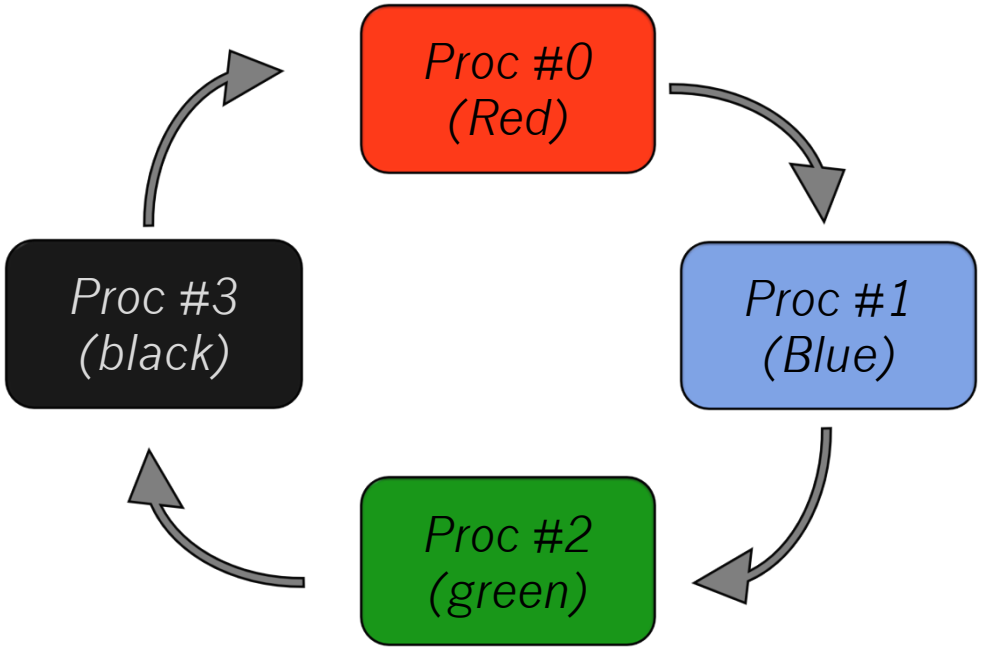
```
int dst = 1 - rank;
MPI_Isend(send_buf, NELEMENTS, MPI_INT, dst, 0, MPI_COMM_WORLD, &req);
MPI_Irecv(recv_buf, NELEMENTS, MPI_INT, dst, 0, MPI_COMM_WORLD, &req);

MPI_Wait (&req, MPI_STATUS_IGNORE);
```

- Note that the order is now irrelevant since completion is done at the `MPI_Wait()` signal.

Example #2: Sending Messages in a Ring

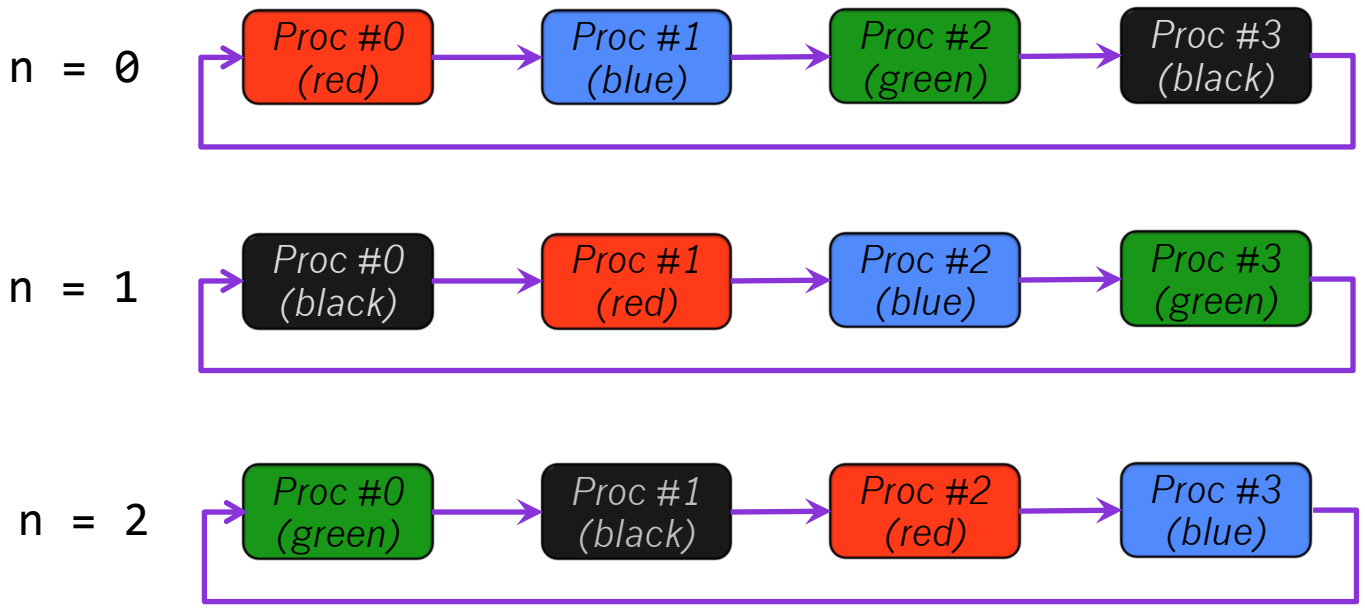
- In this exercise we will allow communications between more than two processes.
- In a ring, each process “talks” to its neighbors on the left and on the right in a circular topology:



- We now wish write a program that makes cyclic permutations of colors.

Example #2: Shuffling Colors

- For each processor, define a unique string defining the process color.
- Perform n cyclic permutations by transferring the color's name to the process to the right:



- Let proc #0 print its color at each iteration n. The output should look like

```
Proc #0, My color is black  [n=1]
Proc #0, My color is green  [n=2]
Proc #0, My color is blue   [n=3]
Proc #0, My color is red    [n=4]
...
```

Example #2: Shuffling Colors

- Write three versions of the program using

1. `MPI_Send()` and `MPI_Recv()`;
2. `MPI_Sendrecv()`;
3. `MPI_Isend()` and `MPI_Irecv()`;