MPI (III)

Distributed-memory programming

Juan Ángel Lorenzo del Castillo

CY Cergy Paris Université

ING2-GSI-MI Architecture et Programmation Parallèle

2023 - 2024

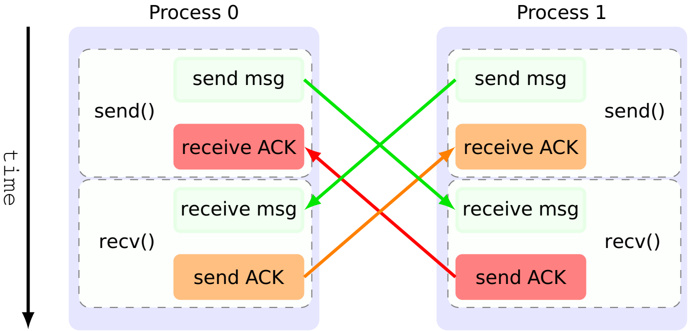
juan-angel.lorenzo-del-castillo@cyu.fr

Table of Contents

Table of Contents

Other communication modes

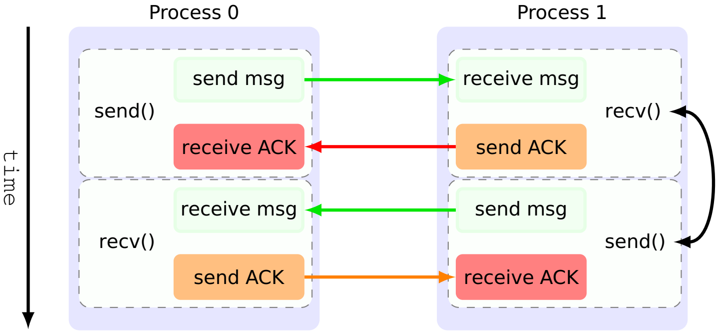
- Standard message send/receive primitives are blocking modes.
  - This can lead to an efficiency loss on some MPI implementations.
  - This can lead to **deadlocks** in some applications.



- In this example, obviously, the send/receive operations order should be switched.

Other communication modes

- Standard message send/receive primitives are blocking modes.
  - This can lead to an efficiency loss on some MPI implementations.
  - This can lead to **deadlocks** in some applications.



  - In this example, obviously, the send/receive operations order should be switched.

Other communication modes

Ring Communications

```
int MPI_Sendrecv(void *sendbuff, int sendcount,
                 MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuff, int recvcount,
                 MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPIComm comm, MPI_Status *status)
```

- It contains the parameters of the `MPI_send` and `MPI_recv` instructions
- It performs a single send & receive operation in the proper order to avoid **deadlocks**

Other communication modes

Ring communication example

```
src = myid-1;
dest = myid+1;

if (src < 0)
  src += num_nodes;

if (dest >= num_nodes)
  dest -= num_nodes;

MPI_Sendrecv(msg_out, msg_len, MPI_INT, dest, tag,
             msg_in, msg_len, MPI_INT, src, tag,
             MPI_COMM_WORLD, &status);
```

Other communication modes

Communication models

- Blocking
- Non blocking

communication modes

- Basic
- Buffered
- Synchronous
- Ready
- Persistent

Blocking model

Other communication modes

- **Buffered** communication mode
  - The message is copied in a buffer
  - This avoids blocking the sender during the communication process
  - It can be started (and completed) whether or not a matching receive has been posted

Buffered communication

- `MPI_Bsend`, with the same arguments as `MPI_Send`
- The user must assign a send buffer:

```
int MPI_Buffer_attach(void *buffer, int size)
```

  - It tells the system to attach `buffer` to the coming communications.

```
int MPI_Buffer_detach(void *buffer, int *size)
```

  - The buffer is freed to be used somewhere else.

Blocking model

Other communication modes

- **Ready** communication mode
  - A send that uses the ready communication mode may be started only if the matching receive is already posted.
    - By using this mode, the sender provides additional information to the system (namely that a matching receive is already posted), which can save some overhead.
    - If the receiver is not ready, the behaviour is unpredictable.
    - In a correct program, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.
  - `MPI_Rsend`, with the same arguments as `MPI_Send`.

Blocking model

Other communication modes

- **Synchronous** communication mode
  - It can be started whether or not a matching receive was posted.
  - However, the send function will not return until the receiver acknowledges the message reception and starts reading it.
  - It does not require the system buffer.
  - `MPI_Ssend`, with the same arguments as `MPI_Send`.
- All the previous modes have a non-blocking version:
  - Basic communication mode: `MPI_Isend`, `MPI_Irecv`
  - `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend`

Blocking model

Other communication modes

- **Persistent** communication mode
  - Improves performance when we need to repeatedly send messages with the same arguments.
  - Two steps:
    - Creation of a context

```
int MPI_Send_init(..., MPI_Request *request)
int MPI_Recv_init(..., MPI_Request *request)
```

      (Basic communicaton mode)
    - Send the message

```
int MPI_Start(MPI_Request *request)
```

  - `MPI_Send` = `MPI_Send_init` + `MPI_Start`
  - Same procedure for other communication modes: `MPI_Bsend_init`, `MPI_Isend_init`, `MPI_Ssend_init`

Blocking model

Other communication modes

Final remarks

- The choice of a communication type is of paramount importance for a good performance in a parallel, distributed memory system.
- That is the reason why MPI offers multiple alternatives to do the job.
- Recommended usage:
  - `MPI_Send`: The most usual alternative.
  - `MPI_Isend`: If non-blocking communications are required.
  - `MPI_Ssend`: When possible. It yields the best performance because it does not use any intermediate buffers.
  - The rest (`MPI_Bsend` and `MPI_Rsend`) for particular cases.

Non-blocking model
Other communication modes

- One can improve performance on many systems by overlapping communication and computation.
- A mechanism that often leads to good performance is to use **nonblocking** communications.
  - A nonblocking **send start** call initiates the send operation, but does not complete it. The send start call can return before the message was copied out of the send buffer.
  - A separate **send complete** call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer.
  - Similarly, a nonblocking **receive start** call initiates the receive operation, but does not complete it. The call can return before a message is stored into the receive buffer.
  - A separate **receive complete** call is needed to complete the receive operation and verify that the data has been received into the receive buffer.

Non-blocking model

Other communication modes

Non-blocking Send/Receive

```
int MPI_Isend(..., MPI_Request *request)
int MPI_Irecv(..., MPI_Request *request)
```

- These operations copy the message into a buffer and return immediately, without waiting for the communication to finish
- `MPI_Request` is like a receipt of the requested operation: it says whether the operation has finished or not

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

  - Waits for the communication to finish
  - `MPI_Isend + MPI_Wait = MPI_Send`, but allowing to perform other operations (computations, etc.) between the send and the wait operations

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
```

  - `flag` value will be 0 if the communication process has not yet finished

Non-blocking model

Other communication modes

Non-blocking communication example

```
int flag = 0;
MPI_Request req;
...

MPI_Isend(buffer, taille_buf, MPI_INT, dest, tag, MPI_COMM_WORLD,
     &req);

while(!flag && still_work_to_do) {
  /* We do other operations here */
  /* ... */

  MPI_Test(&req, &flag, &status);
}

MPI_Wait(&req, &status);
```

Non-blocking model

Other communication modes

- How to know if several communications have finished:
  - Wait functions
    - `MPI_Waitall`, wait for all
    - `MPI_Waitany`, wait for any
    - `MPI_Waitsome`, wait for a specific set
  - Check functions
    - `MPI_Testall`, check all operations
    - `MPI_Testany`, test any operation
    - `MPI_Testsome`, test a specific set

Non-blocking model

Other communication modes

- Communication test functions (without using MPI_Request)
  - MPI_Probe, MPI_Iprobe: They test if any message has arrived (without reading it)
  - Using the information given by these functions, it is possible to know the sender's identity, the message tag and its size.
  - This allows, for example, waiting simultaneously for three classes of messages, each associated to a different datatype. The class will be given by the tag value. Knowing this value we will know if we want to read that message.
  - It allows as well finding out the size of a message of unknown length, in order to reserve the memory space required to store the data that will arrive.
  - Next, we can proceed to actually read the message with the corresponding call to MPI_Recv or MPI_Irecv.

```
int MPI_Probe(int source, int tag, MPI_Comm comm,
              MPI_Status *status)

int MPI_Iprobe(int source, int tag, MPI_Comm comm,
               int *flag, MPI_Status *status)
```

Non-blocking model Other communication modes

Example:

```c
// Author: Wes Kendall
// Copyright 2011 www.mpitutorial.com
// Example of using MPI_Probe to dynamically
      allocated received messages

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char** argv) {
  MPI_Init(NULL, NULL);

  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  if (world_size != 2) {
    fprintf(stderr, "Must use two processes for
          this example\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
  }
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  int number_amount;
  if (world_rank == 0) {
    const int MAX_NUMBERS = 100;
    int numbers[MAX_NUMBERS];
```

```c
    // Pick a random amont of integers to send to process one
    srand(time(NULL));
    number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;
    // Send the amount of integers to process one
    MPI_Send(numbers, number_amount, MPI_INT, 1, 0,
          MPI_COMM_WORLD);
    printf("0 sent %d numbers to 1\n", number_amount);
  } else if (world_rank == 1) {
    MPI_Status status;
    // Probe for an incoming message from process zero
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
    // When probe returns, the status object has the size and
          other
    // attributes of the incoming message. Get the size of the
          message.
    MPI_Get_count(&status, MPI_INT, &number_amount);
    // Allocate a buffer just big enough to hold the incoming
          numbers
    int* number_buf = (int*)malloc(sizeof(int) * number_amount
          );
    // Now receive the message with the allocated buffer
    MPI_Recv(number_buf, number_amount, MPI_INT, 0, 0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("1 dynamically received %d numbers from 0.\n",
          number_amount);
    free(number_buf);
  }
  MPI_Finalize();
}
```

Table of Contents

Communicators

- A *communicator* is a subset of processes that can exchange information. Each subset will have its own communication space.
- A communicator comprises, at least, a group of processes and a **context**.
  - A *context* defines an well-known, identified communication space.
  - A process can belong to several communicators, thus it will have a different id for each of the contexts where it belongs.
  - A communicator may include multiple associated information, such as a *topology*.
- The `MPI_COMM_WORLD` communicator is predefined and comprises all processes.
- New communicators can be defined using subsets of existing processes.

Motivation

Suppose that we want to broadcast a collective message to all processes with a pair rank, and another message to all processes with odd rank.

- Writing a loop that iterates over pairs of *send/recv* instructions can be detrimental for the performance, particularly if the number of processes is high. In addition, it will be necessary to include a test in the loop to check whether the receiving process has a pair or odd rank.
- A possible solution is to create two communicators: one that contains all pair processes and a second one with the odd processes. Then we can run collective communications, such as a broadcast, inside each communicator.

Creating communicators

- A communicator is always created from a previous existing communicator. The first one will be created from MPI_COMM_WORLD.
- After invoking MPI_INIT(), a communicator is created during the program execution lifetime.
- Its identifier MPI_COMM_WORLD is an integer defined in the header file.
- This global communicator will be destroyed by invoking MPI_FINALIZE().
- All communicators created by the developer can be dynamically managed, and they can be destroyed by the function MPI_COMM_FREE().

Creating communicators

- Three steps:

Get a group associated to an existing communicator

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

Create a group from another existing group

```
int MPI_Group_incl(MPI_Group group, int newsize, int *ids,
                   MPI_Group *newgroup)
```

Create a communicator associated to the new group

```
MPI_Comm_create(MPI_Comm comm, MPI_Group newgroup,
                MPI_Comm *newcomm)
```

Creating communicators

Example: Communicator with *c* processes

```
MPI_Comm C_row0;
MPI_Group group, group_row0;

// List of processes of the new communicator
for(proc=0; proc<c; proc++)
    pids[proc] = proc;

MPI_Comm_group(MPI_COMM_WORLD, &group);
MPI_Group_incl(group, c, pids, &group_row0);
MPI_Comm_create(MPI_COMM_WORLD, group_row0, &C_row0);
...
MPI_Comm_rank(C_row0, &pid_f0);
```

Creating communicators

Creation of multiple communicators simultaneously

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

- Collective function
- All processes with the same *colour* are attached to the same communicator.
- Processes are sorted on each communicator by the *key*.

Creating communicators

Example: One communicator for each row of a mesh of $f \times c$ processes.

```
// pid = id in the global communicator

my_row = pid/c;
MPI_Comm_split(MPI_COMM_WORLD, my_row, pid, &C_my_row);
```

- Everybody executes the function.
- Processes that are not going to be be part of the new communicators will call the function with the `MPI_UNDEFINED` colour value.