Bach Khoa University Of Technology
Faculty of Computer Science & Engineering

Course: Parallel Computing
Lab #3 - MPI Point-to-Point Communication

Thanh-Dang Diep

August 19, 2017

Goal: This lab helps students to get familiar with MPI point-to-point communication which is the most fundamental one to write parallel programs running distributed memory systems.

Content: The first Section introduces the MPI program's structure. The next Section shows basic environment management routines while the third Section explains point-to-point communication routines. The last one is exercises to help practice the obtained knowledge.

Result: After finishing this lab, students are able to understand and write a basic parallel program running distributed memory systems by using MPI point-to-point communication.

Contents

1 Introduction

MPI (Message Passing Interface) is a programming language specification (similarly library, API, etc.) which facilitates programmers in writing parallel programs running on distributed memory systems (as shown in Figure 1).

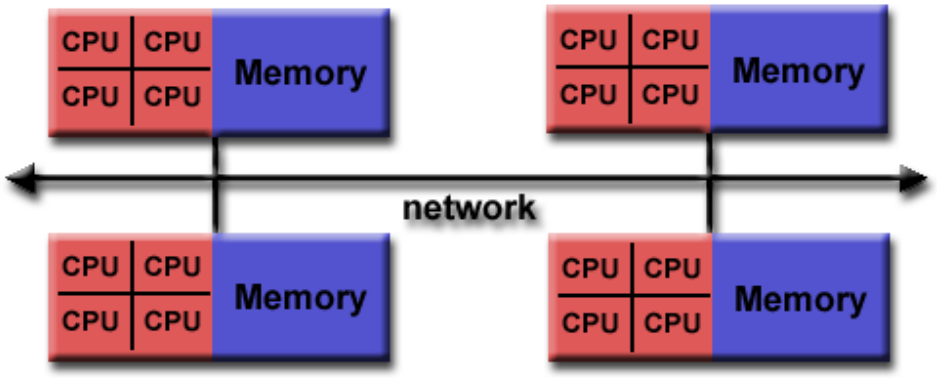

Figure 1: Distributed memory system

The MPI program's structure is commonly formatted as shown in Figure 2.

Example 1: The first program - Hello World

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

int main(int argc, char **argv)
{
     int i, rank, size;
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);

     printf("Hello world, I have rank %d out of processes \n", rank, size);
     MPI_Finalize();

     return 0;
}
```

To compile and run the program, do the following steps:

```
$ mpicc hello.c -o hello
$ mpirun -np <TheNumberOfProcesses> -hostfile <FileName> hello
```

2 Environment Management Routines

These routines are used for interrogating and setting the MPI execution environment, and covers assortment of purposes, such as initializing and terminating the MPI environment, querying a rank's prosess, etc.

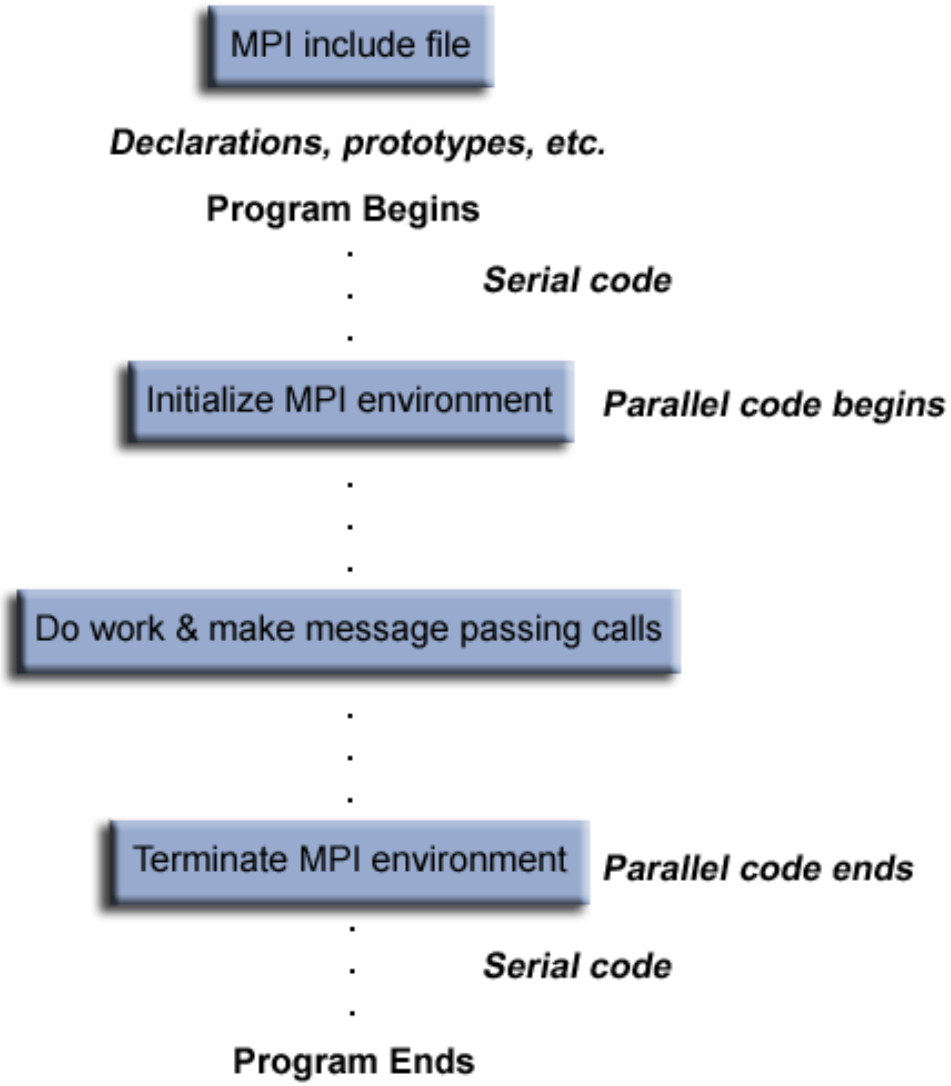

Figure 2: General MPI program structure

- **MPI_Init**

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

```
MPI_Init(&argc, &argv);
```

- **MPI_Comm_size**

Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD. If the communicator is MPI_COMM_WORLD, then it presents the number of MPI tasks available to your application.

```
MPI_Comm_size(comm, &size);
```

- **MPI_Comm_rank**

Returns the rank of the calling MPI process within the specified communicator. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD. This rank is often offered to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

```
MPI_Comm_rank(comm, &rank);
```

- **MPI_Wtime**

Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

```
MPI_Wtime();
```

- **MPI_Finalize**

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

```
MPI_Finalize();
```

3 Point to Point Communication Routines

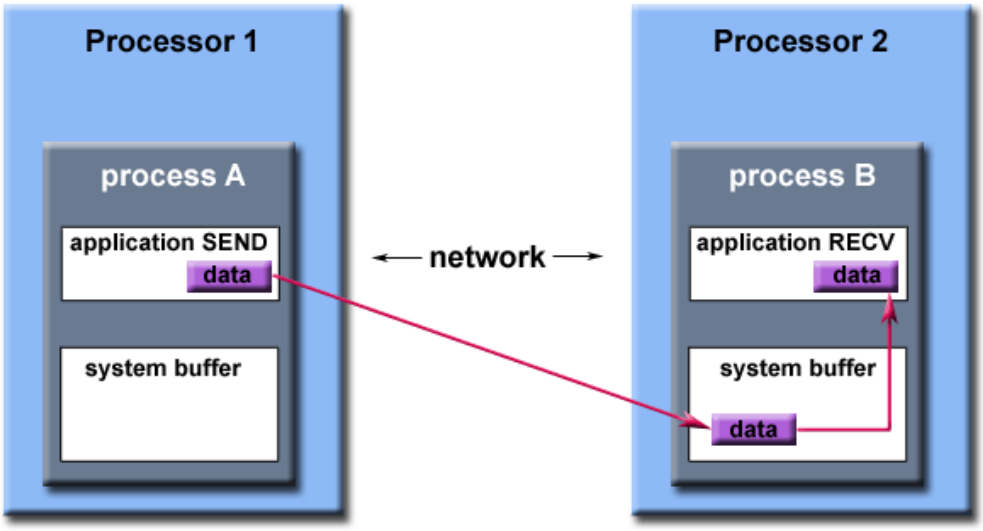

Figure 3: Path of a message buffered at the receive process

MPI point-to-point operations typically involve message passing between two, and only two, different MPI processes. One process is performing a send operation and the other process is performing a matching receive operation as shown in Figure 3.
There are different types of send and receive routines used for different purposes. However, to figure out easily, we only consider some basic MPI point-to-point routines used in either blocking or non-blocking mode. Figure 4 shows the essential difference between these modes.

Blocking Send	Non-blocking Send

myvar = 0;
 	
for (i=1; i<ntasks; i++) {
task = i;
MPI_Send (&myvar ... ... task ...);
myvar = myvar + 2
 	
/* do some work */
 	
}
	myvar = 0;
	
	for (i=1; i<ntasks; i++) {
	task = i;
	MPI_Isend (&myvar ... ... task ...);
	myvar = myvar + 2;
	
	/* do some work */
	
	MPI_Wait (...);
	}

Safe. Why?	Unsafe. Why?

Figure 4: The difference between blocking and non-blocking modes

3.1 Blocking Rountines

- **MPI_Send**

Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse.

```
MPI_Send(&buf, count, datatype, dest, tag, comm);
```

- **MPI_Recv**

Receive a message and block until the requested data is available in the application buffer in the receiving process.

```
MPI_Recv(&buf, count, datatype, source, tag, comm, &status);
```

- **MPI_Wait**

MPI_Wait blocks until a specified non-blocking send or receive operation has completed. For multiple non-blocking operations, the programmer can specify all completions.

```
MPI_Wait(&request, &status);
MPI_Waitall(count, &array_of_requests, &array_of_statuses);
```

Example 2: Blocking Message Passing Routines
Process 0 pings process 1 and awaits return ping

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

int main(int argc, char **argv)
{
     int numtasks, rank, dest, source, rc, count, tag = 1;
     char inmsg, outmsg = 'x';
     MPI_Status Stat;    // required variable for receive routines
```

```
10      MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        /* task 0 sends to task 1 and waits to receive a return message */
15      if (rank == 0)
        {
             dest = 1;
             source = 1;
             MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
20           MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
        }

        /* task 1 waits for task 0 message then returns a message */
        else if (rank == 1)
25      {
             dest = 0;
             source = 0;
             MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
             MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
30      }

        /* query receive Stat variable and print message details */
        MPI_Get_count(&Stat, MPI_CHAR, &count);
        printf("Task %d: Receive %d char(s) from task %d with tag %d \n", rank, count,
            Stat.MPI_SOURCE, Stat.MPI_TAG);
35

        MPI_Finalize();
        return 0;
}
```

3.2 Non-blocking Routines

- **MPI_Isend**

Identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to MPI_Wait indicate that the non-blocking send has completed.

```
MPI_Isend(&buf, count, datatype, dest, tag, comm, &request);
```

- **MPI_Irecv**

Identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the application buffer. A communication request handle is returned for handling the pending message status. The program must use calls to MPI_Wait to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

```
MPI_Irecv(&buf, count, datatype, source, tag, comm, &request);
```

Example 3: Non-blocking Message Passing Routines
Nearest neighbor exchange in a ring topology as shown in Figure 5

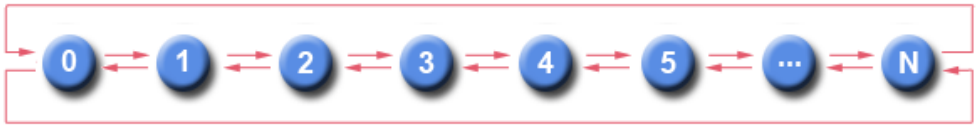

Figure 5: Data exchange in a ring topology

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

int main(int argc, char **argv)
{
     int numtasks, rank, next, prev, buf[2], tag1 = 1, tag2 = 2;
     MPI_Request reqs[4];     // required variable for n non-blocking calls
     MPI_Status stats[4];     // required variable for Waitall routines

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

     /* determine left and right neighbors */
     prev = rank - 1;
     next = rank + 1;
     if (rank == 0) prev = numtasks - 1;
     if (rank == (numtasks - 1)) next = 0;

     /* post non-blocking receives and sends for neighbors */
     MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
     MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

     MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
     MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

     /* do some work while sends/receives progress in background */

     /* wait for all non-blocking operations to complete */
     MPI_Waitall(4, reqs, stats);

     /* continue - do mork work */

     MPI_Finalize();
     return 0;
}
```

4 Exercises

1. Write a program that prints out prime numbers in the first one billion of positive integers.

2. Write a program that measures the network bandwidth between any two node in the cluster by exchanging data packages of size 128, 256, 512 and 1024 KB among these nodes. Each data package will be exchanged 5 times to attain the average value.

3. Write a program that computes the value of Pi using Monte Carlo simulation. The program samples points inside the rectangle delimited by (0,0) and (1,1) and counts how many of these are within a circle with a radius of 1. The ratio between the number of points inside the circle and the total number of samples is Pi/4.

4. Write a program multiplying two square matrices whose sizes of each are 1000x1000 and 10000x10000.