MONASH University

FIT3143 Tutorial Week 6

Lecturers: ABM Russel (MU Australia) and Vishnu Monn (MU Malaysia)

DISTRIBUTED MEMORY

OBJECTIVES

- The purpose of this tutorial is to cover the basic concepts of Message Passing Interface.

Note: Tutorials are not assessed. Nevertheless, please attempt the questions to improve your unit comprehension in preparation for the labs, assignments, and final assessments.

QUESTIONS

1. With reference to the following code, describe the general coding pattern of a Message Passing Interface (MPI) program. How does this pattern differ from a shared memory parallel program?

```
#include <math.h>
#include <string.h>
#include <mpi.h>

// Point to point communication
int main()
{
	int my_rank;
	int p, i;
	//MPI_Status status;

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

	int val = -1;
	do{
	  if(my_rank == 0)
	  {
		printf("Enter a round number (>= 0 ): ");
		fflush(stdout);
		scanf("%d", &val);
		for(i = 1; i < p; i++){
			MPI_Send(&val, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
		}
	  }
	  else
	  {
```

```
            MPI_Recv(&val, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            printf("Processors: %d. Received Value: %d\n", my_rank, val);
            fflush(stdout);
      }
}while(val >= 0);

MPI_Finalize( );
return 0;
}
```

The general coding pattern is based on the following pattern:

- MPI Init()
- MPI Comm rank()–
- MPI Comm size()
- Parallel or work region and communication in this region using communication methods (e.g., MPI_Send() and MPI_Recv() functions)
- MPI Finalize()

In MPI, the entire application is executed as one or more processes. Even on the same local computer, each MPI process cannot directly access the other process's memory. Message passing functions are used to send and/or receive information between processes.

On the other hand, threads (shared memory) are forked within a process. Although each thread has its own local memory, the threads share the global or heap region of the memory for a process.

2. Describe the difference between a blocking and non-blocking message passing. Explain your answer using a sequence diagram by comparing the MPI_Send() and MPI_Isend() functions.

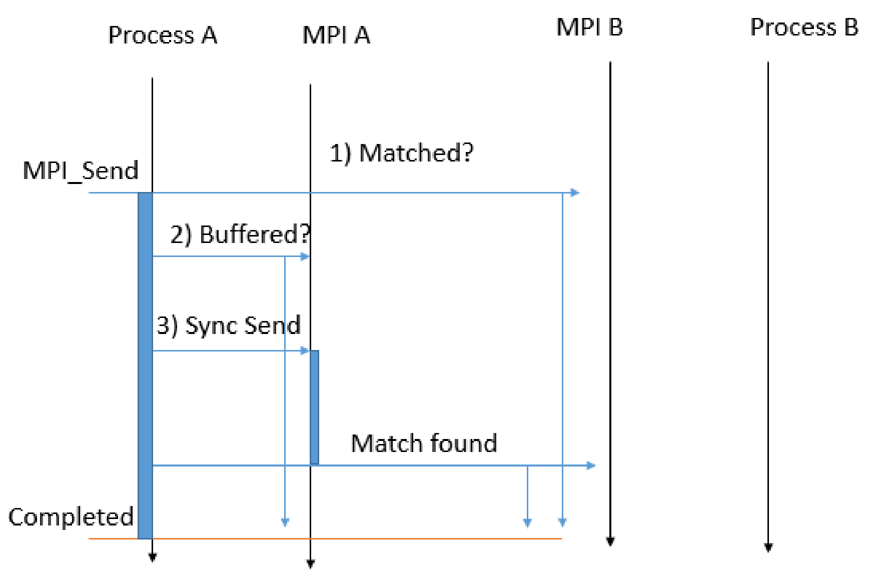
A blocking send call will halt program execution until it has been assured its message has been successfully passed to the message buffer. Alternately, the nonblocking send will continue program execution immediately after the function call. It does not care whether the call has been successfully completed or not.

The following solution is based on the online article by:

https://iamsorush.com/posts/mpi-send-types/

MPI_Send() – Blocking message passing

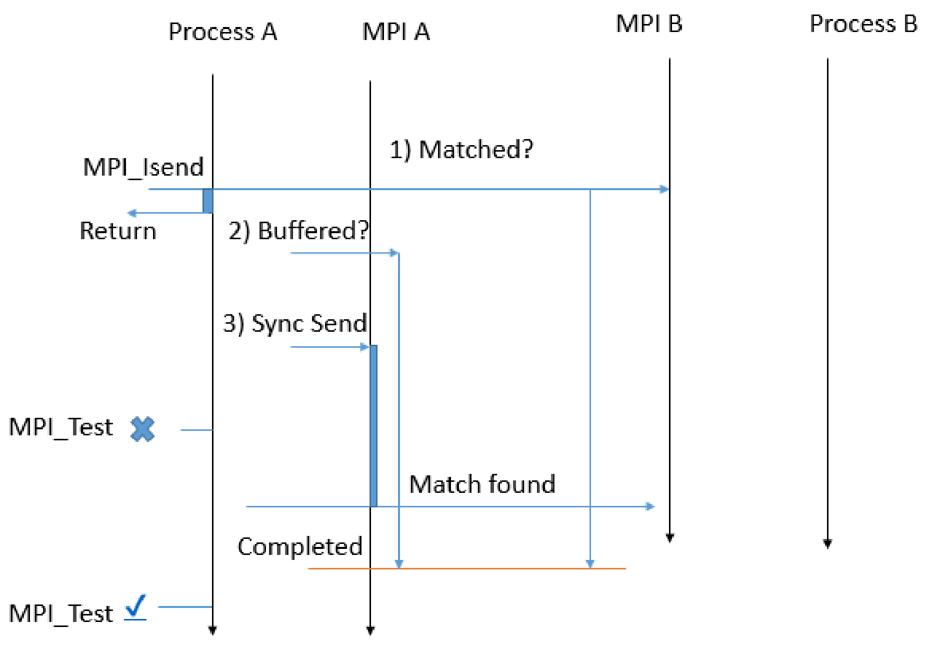
This is the **standard** mode. When it is called, (1) the message can be directly passed to the receive buffer, (2) the data is buffered (in temporary memory in the MPI implementation) or (3) the function waits for a receiving process to appear. See the picture below. Therefore, it can return quickly (1)(2) or block the process for a while (3). MPI decides which scenario is the best in terms of performance, memory, and so on. In any case, the data can be safely modified after the function returns.



MPI_Send() sequence diagram (https://iamsorush.com/posts/mpi-send-types/)

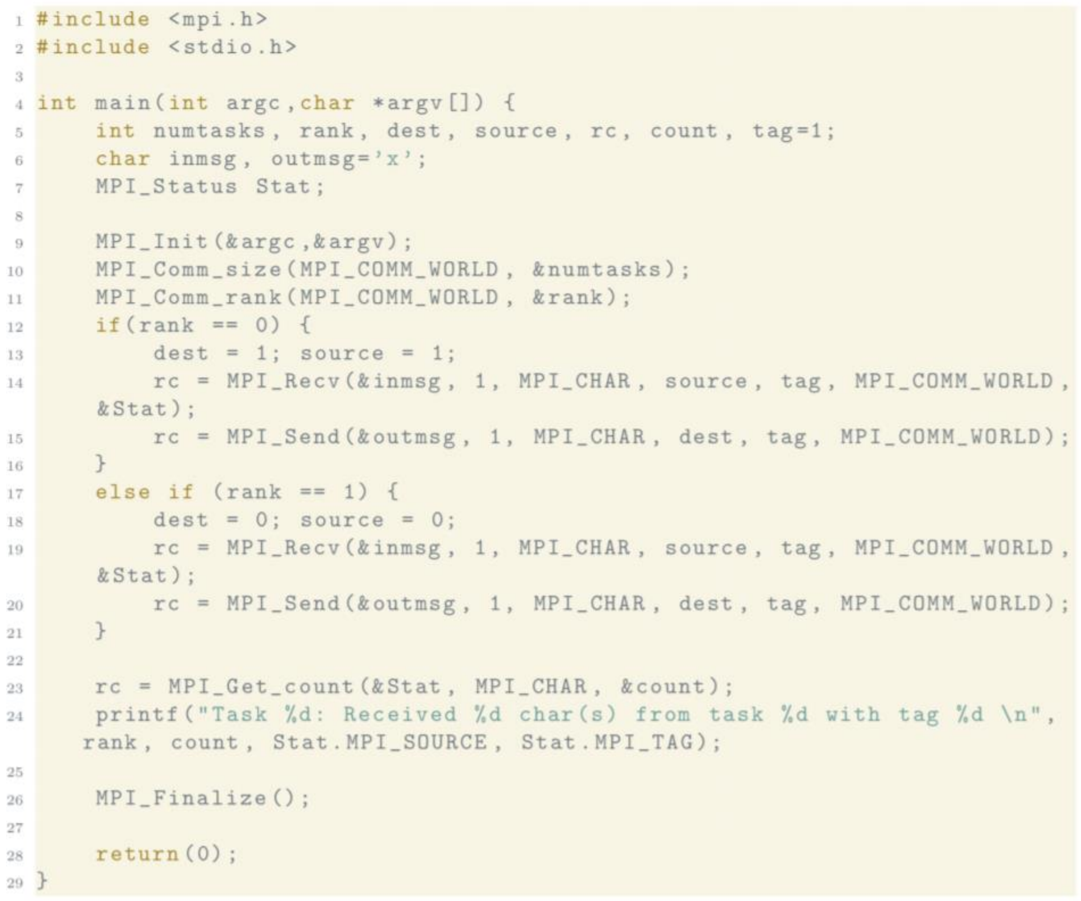
MPI_Isend() – Non blocking message passing

It is the non-blocking version of MPI_Send. When this function is called the function returns immediately but runs MPI_Send actions in the background of the process. Therefore, after the function returns, the data must not be modified unless MPI_Test and MPI_Wait confirm MPI_Isend is completed. After the completion, the data is reusable because it either is buffered in MPI or sent to the destination.



MPI_Isend() sequence diagram (https://iamsorush.com/posts/mpi-send-types/)

3. Briefly discuss what is wrong with the following code snippet?



```c
#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;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == 0) {
        dest = 1; source = 1;
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    else if (rank == 1) {
        dest = 0; source = 0;
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }

    rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
    printf("Task %d: Received %d char(s) from task %d with tag %d \n", rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

    MPI_Finalize();

    return(0);
}
```

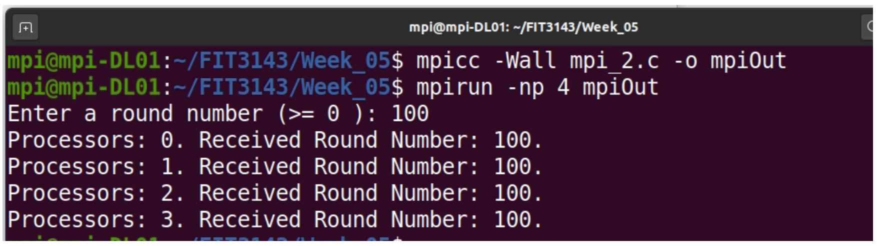
All ranks post a blocking receive before their send. This means all ranks will be blocking and waiting for a receive and no ranks will be sending, i.e., deadlock.

4. The following code attempts to broadcast a value from the root MPI process to other processes within the MPI group.

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

int main(int argc, char* argv[])
{
     int my_rank;
     int p;

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
     MPI_Comm_size(MPI_COMM_WORLD, &p);

     int val = -1;

     if(my_rank == 0)
     {
      printf("Enter a round number (>= 0 ): ");
```

```
         fflush(stdout);
         scanf("%d", &val);
         MPI_Bcast(&val, 1, MPI_INT, 0, MPI_COMM_WORLD);
        }else {
         MPI_Bcast(&val, 1, MPI_INT, 0, MPI_COMM_WORLD);
        }
        printf("Processors: %d. Received Round Number: %d.\n", my_rank, val);
        fflush(stdout);

        MPI_Finalize();
        return 0;
    }
```

When compiling the code above and executing, the following outcome is observed:



```
mpi@mpi-DL01:~/FIT3143/Week_05$ mpicc -Wall mpi_2.c -o mpiOut
mpi@mpi-DL01:~/FIT3143/Week_05$ mpirun -np 4 mpiOut
Enter a round number (>= 0 ): 100
Processors: 0. Received Round Number: 100.
Processors: 1. Received Round Number: 100.
Processors: 2. Received Round Number: 100.
Processors: 3. Received Round Number: 100.
```

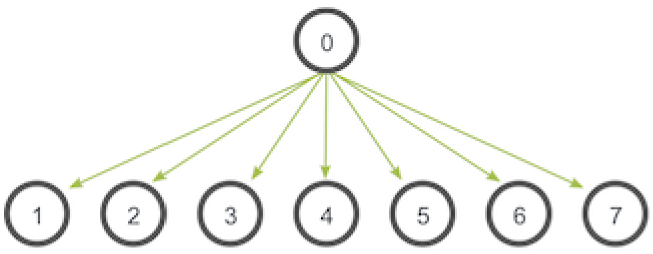
Answer the following:

a) Describe the concept of MPI broadcast.

Solution from: https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

A ***broadcast*** is one of the standard collective communication techniques. During a broadcast, one process sends the same data to all processes in a communicator. One of the main uses of broadcasting is to send out user input to a parallel program or send out configuration parameters to all processes.

The communication pattern of a broadcast looks like this:



In this example, process zero is the ***root*** process, and it has the initial copy of data. All the other processes receive the copy of data.

b) In the code above, the function, the MPI_Bcast() function is called twice, i.e., once inside the if condition for root process and the second time in the else condition. Yet, the produced answer remains consistent. Why is this so?

The MPI_Bcast represents a wrapper of the MPI send and recv functions. In the code above, the root process (i.e., rank = 0) calls the send function insider MPI_Bcast. At the same time, the other processes would call the receive function inside MPI_Bcast. Therefore, the code should work. Nevertheless, this is not an ideal approach. Instead, MPI_Bcast should not be called within a rank condition.

c) What happens if the root MPI process rank (i.e., my_rank = 0) does not call the MPI_Bcast() function after prompting the user for a round number?

The non-root MPI processes that calls the MPI_Bcast() function will wait to receive the value which is supposed to be broadcasted by the root rank. However, since the root MPI process rank did not call MPI_Bcast() function, the other processes will continue to wait for value that will not be broadcasted. This leads to a program failure.