Yannick Omar (352000) | Ting-Jui, Hsu (351218) |
Abdelrahman Abdelkawi (350125)

Assignment 3

1.

a)

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

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

int me,nop;                              //me: process id
                                         // nop: number of processes
MPI_Init(&argc,&argv);                   //Init MPI session
MPI_Comm_size(MPI_COMM_WORLD,&nop);//get number processes
MPI_Comm_rank(MPI_COMM_WORLD,&me); //get process ID

//print result
printf("Hello World, I am process %i of %i\n",me,nop);

MPI_Finalize();                         //terminate MPI session

return 0;
}
```

b)

```
Hello World, I am process 1 of 8
Hello World, I am process 2 of 8
Hello World, I am process 7 of 8
Hello World, I am process 0 of 8
Hello World, I am process 4 of 8
Hello World, I am process 5 of 8
Hello World, I am process 3 of 8
Hello World, I am process 6 of 8
```

The reason why the order can differ every time, is that multiple processes are created once MPI_Init is reached. These multiple processes are work independently of each other and no order of execution was specified. Hence, the processes are run when the CPU scheduler allows them to. Therefore, the result cannot be deterministic (at least not for this program).

2.

a)

```
int PMPI_Sendrecv(
  void *sendbuf, int sendcount, MPI_Datatype sendtype,
  int dest, int sendtag, void *recvbuf, int recvcount,
  MPI_Datatype recvtype, int source, int recvtag,
  MPI_Comm comm, MPI_Status *status)
```

Yannick Omar (352000) | Ting-Jui, Hsu (351218) |
Abdelrahman Abdelkawi (350125)
Assignment 3

b)

The MPI_Sendrecv operation allows to give the command for both sending and receiving messages at the same time. This facilitates an easier way of communication when otherwise a careful order of sending and receiving messages would have been necesary. To give an example, one can consider two nodes where both want to send and receive messages from each other.
Imagine node 1 and 2 try to send at the same time using blocking sending operations then both would be locked in this state. In this case this problem can obiviously done correcting by letting e.g. node 1 only send after he receveid a message from node 2. However, this simple example shows that for more complex communication pattern, the MPI_Sendrecv operation is helpful. MPI_Sendrecv then tries to send similarly to MPI_Send but doesn't block the receiving operation. The data parameters have the same structure as the indivdual send and receive operation for sending and receiving respectively.

c)

| IN/Out | Parameter | Explanation |
|---|---|---|
| IN | *sendbuf | starting address of memory to send |
| IN | sendcount | size of data to be send |
| IN | sendtype | MPI_Datatype to be send |
| IN | dest | destination for data |
| IN | sendtag | TAG connected to the sent data |
| OUT | *recvbuf | starting address of receiving memory |
| IN | recvcount | size of data to be received |
| IN | recvtype | MPI_Datatype to be received |
| IN | source | rank of whom to receive data |
| IN | recvtag | TAG of data to be received or MPI_ANY_TAG |
| IN | comm | Communicator from which data are to be received or MPI_COMM_WORLD |
| OUT | *status | status object |

3) a)-c)

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi/mpi.h>
#include <string.h>

#define TAG_SIZE 1  //tag for sending/receiving size of the string
#define TAG_STR 2   //tag for sending/receiving the string

/*function to send string using MPI
input: drank: rank of destination
       str: string to be send*/
void send_string(int drank, char *str){
    //get the string length, +1 to condisder null terminator
    int N=strlen(str)+1;
```

```c
    //send size of string
    MPI_Send(&N, 1,MPI_INT,drank,TAG_SIZE,MPI_COMM_WORLD);
    //send string
    MPI_Send(str,N,MPI_CHAR,drank,TAG_STR,MPI_COMM_WORLD);
}

/*function to receive string, input: source from data are to be received*/
char* recv_string(int source){
    int N;              //string size
    char* str;          //string to be received
    MPI_Status status_size,status_str; //stati for receiption

    //receive size of string
    MPI_Recv(&N,1,MPI_INT,source,TAG_SIZE,MPI_COMM_WORLD,&status_size);
    //allocate memory according to exepected string size
    str=malloc(N*sizeof(char));
    //receive string
    MPI_Recv(str,N,MPI_CHAR,source,TAG_STR,MPI_COMM_WORLD,&status_str);

    return str;
}

int main(int args,char *argv[])
{
    int me,np;  //process id, number of processes
    MPI_Init(&args,&argv);                  //init MPI session
    MPI_Comm_rank(MPI_COMM_WORLD,&me);  //get own id
    MPI_Comm_size(MPI_COMM_WORLD,&np);  // get np

    if (me!=0){ //if own id is not of rank 0
        char str[MPI_MAX_PROCESSOR_NAME]; //string to be send
        int len;                          //string length
        //get processor name (includes null terminator, length
        //of returned value doesn't account for it
        MPI_Get_processor_name(str,&len);
        //send string to process 0
        send_string(0,str);
    }
    else{
        int k=0, len; //k: counter of processes, len: length of processor name
        char str[MPI_MAX_PROCESSOR_NAME]; //processor name
        //get null terminated processor name
        MPI_Get_processor_name(str,&len);
        //print result for process zero
        printf("Process %i of %i is running on %s\n",k,np,str);
        for (k=1;k<np;k++){
            //receive processor name from every process
            char *str=recv_string(k);
            printf("Process %i of %i is running on %s\n",k,np,str);
            free(str); //free alloocated memory
        }
    }

```

```
    MPI_Finalize(); //terminate MPI session
    return 0;
}
```