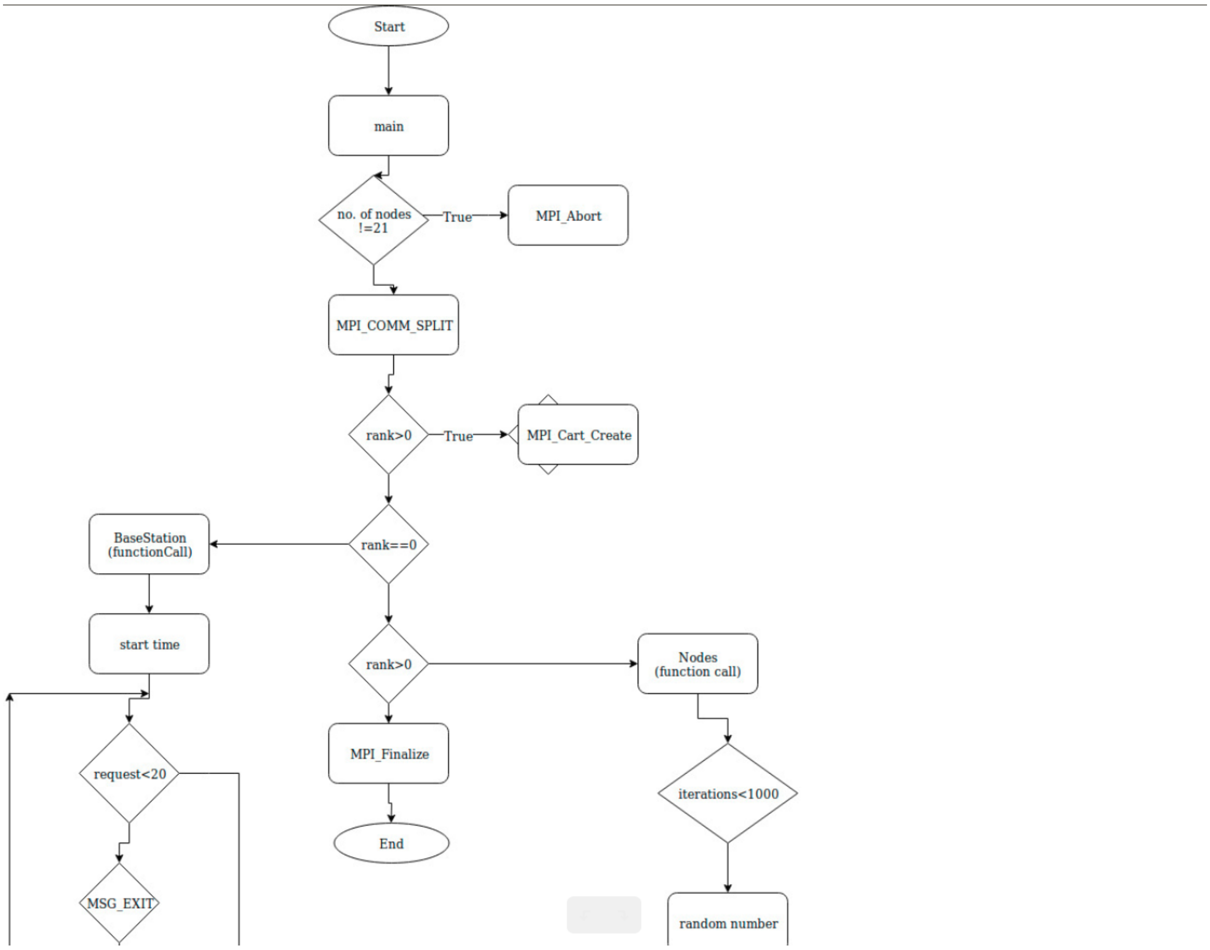
Part A

Overview

The purpose of this report is to develop a Message Passing Interface (MPI) code that stimulates the operation of the Wireless Sensor Network (WSN) including the base station in an efficient manner. This WSN consists of 20 nodes arranged in a 4x5 (rectangular-shaped) grid and a base station (Refer to Figure 7 below). These nodes are arranged in a manner that the adjacent nodes can wirelessly communicate and can exchange data through communication modes such as unicast and broadcast but communication between non-adjacent nodes is not possible.
The program runs for a number of iterations (in our case: 1000) and each node generates a random number (in our case between 1 to 10). An event is recorded in case of three adjacent nodes generating the same random number. The nodes with an event are collected at the base station where they are logged into a file.

Design



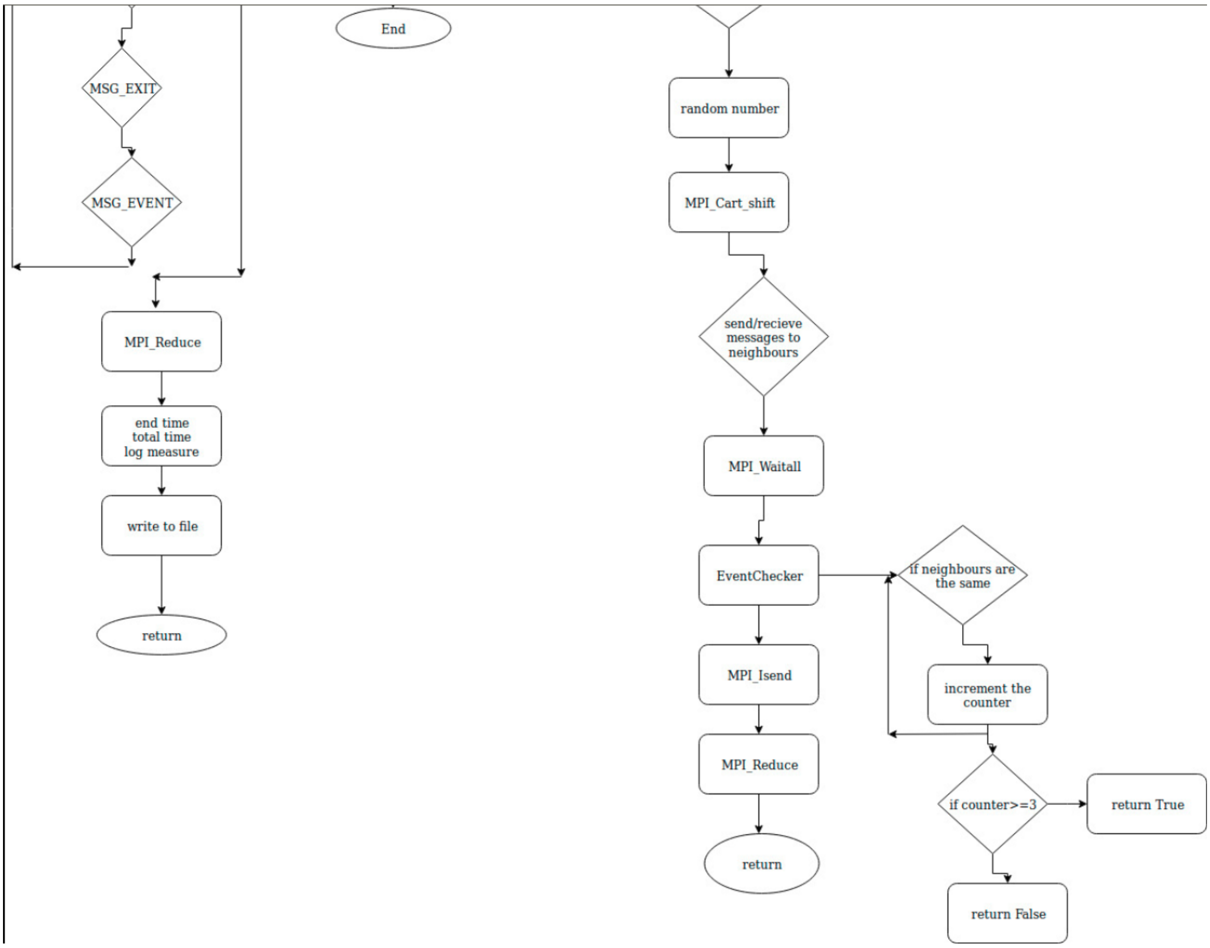


Figure 1: Flowchart of MPI structure

The way the program is structured is:
The header files are included for example **#include “mpi.h”** which contains the MPI-specific definitions and function prototypes. Figure 2 below displays this information.

```
#include <stdio.h>
#include "mpi.h"
#include <time.h>
#include <stdlib.h>
#include <string.h>
#define MAXRND 10
#define MSG_EXIT 0
#define MSG_EVENT 1
#define MAXN 256
```

Figure 2: Header files

Main function:
Following the header files, the variables are declared and the MPI execution environment is initialized.
All MPI process can only communicate if they share a communicator therefore MPI_COMM_WORLD is used.
Each process has its own rank and size.

A check is done at the start to see if the number of nodes is equal to 21 as per assignment specification. If the size is not equal to 21 (the number of total nodes including the base station and the slave nodes), the MPI environment is aborted.
Further initializations are made for the splitting the communicators between base-station (master) and nodes (slaves).
In order to create a 4x5 grid, MPI_Cart_Create was used. Refer to Figure 7 below for an example. What MPI_Cart_Create does is, it “makes a new communicator to which topology information has been attached.” Note that MPI_Cart_Create was only used for rank >0 that is for all the nodes except the base station.
Following the creation of the grid, the base station function is called if rank == 0 else the slave node function is called.
Finally, MPI_Finalize is used to terminate the mpi execution environment.
The flow of the program can be seen in Figure 1 above.
Figure 3 below shows the code for the main function.

```c
int sum; //total messages of system
void baseStation(MPI_Comm master_comm);
void node(MPI_Comm master_comm, MPI_Comm comm);
int eventChecker(int array[4]);

int main(int argc, char** argv){
    int rank;
    int size;
    MPI_Init(&argc, &argv); //Initialize the MPI execution environment
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); //Determines the rank of the calling process in the communicator
    MPI_Comm_size(MPI_COMM_WORLD, &size);//Determines the size of the group associated with a communicator

    if (size != 21){ //if the number of nodes is not eqaul to 21, abort.
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    int ndims = 2; //number of dimensions of cartesian grid
    int dimension[2]; //size of the the dimension is 2, to make a 4x5 grid
    int periods[2]; //logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
    int reorder = 0;//ranking may be reordered (true) or not (false) (logical)
    MPI_Comm comm_cart; //communicator with new cartesian topology (handle)
    MPI_Comm comm_old; //input communicator (handle)

    dimension[0]=5;
    dimension[1]=4;
    periods[0]=0; //stored as False
    periods[1]=0; //stored as False


    MPI_Comm_split(MPI_COMM_WORLD, rank==0, 0, &comm_old); //split into masters(base station) & slaves(nodes)

    if (rank > 0){ //4x5 grid for the slaves
        MPI_Cart_create(comm_old, ndims, dimension, periods, reorder, &comm_cart);// Makes a new communicator to which topology information has been attached
    }

    if (rank==0){
        baseStation(MPI_COMM_WORLD); //rank 0 executes base station
    }else if (rank>0){
         node(MPI_COMM_WORLD, comm_cart); //other ranks nodes
    }

    MPI_Finalize(); //Terminates MPI execution environment
    return 0;
}
```

Figure 3: main function

Base Station (Master) function:

The base station’s main function is to receive the messages from the slave nodes (where an event occurs) whose three or more adjacent nodes generate the same random number. Below is the program structure.
Initializes the variables.
Timer is started.
For every node (slaves) checks whether MSG_EXIT or if it’s MSG_EVENT. Increment and writing to a file is done according to the type of message.
Total number of messages is calculated.
Timer is stopped.
Total time is calculated.
The performance measures: ‘number of events’, ‘time taken’ and ‘messages’ are recorded and written into a file.

The flow of the program can be seen in Figure 1 above.
Figure 4 below shows the code for the main function.

```
void baseStation(MPI_Comm master_comm){
    char buffer[MAXN]="NULL"; //initialize the buffer
    char measure[MAXN]; //measure array of size 256
    int requests = 0;
    int events=0;
    int numMsg=0;
    MPI_Status status;
    FILE *file;

    file = fopen("file.txt", "w+");
    double start = MPI_Wtime();//start time
    //iteration for the nodes
    while (requests < 20){
        MPI_Recv(buffer, MAXN, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, master_comm, &status);
        switch(status.MPI_TAG){
            case EXIT: //case for message exit
                requests++;
                break;
            case EVENT: //case for message event
                fputs(buffer, file); //write
                events++;
                break;
        }
    }
        //to calculate the total messages
    MPI_Reduce(&numMsg, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD); //Reduces values on all processes to a single value
    double end = MPI_Wtime(); //end time
    double total = end - start; //get the total time
    sprintf(measure, "Time Taken: %f\nEvents:%d\nMessages: %d\n", total, events, sum);
    fputs(measure , file); //log performance metrics
    fclose(file);
}
```

Figure 4: base station (master) function

Node (Slaves) function:

The node function’s job is to generate a random number for **1000 iterations**. The range of random numbers is kept between 1 to 10. This range was chosen so that the number of events at a node are often.

MPI_Cart_shift is used to get the adjacent nodes and messages are sent and received between every node and its adjacent nodes.

MPI_Waitall is used to wait for all given MPI Requests to complete.

The next step is to check if there is an event at a particular node. The ‘EventChecker’ function is called. If it returns ‘True’ (1), the node at which an event occurred, together with its adjacent nodes is printed. A non-blocking send is made to the master communicator. Finally, to end simulation, a finish request is sent and the total messages(num messages) are collected in MPI_reduce. Figure 5 below shows this.

```c
void node(MPI_Comm master_comm, MPI_Comm comm){
    int numRequests;
    int rank;
    int size;
    MPI_Request array_of_requests[8];
    MPI_Status array_of_statuses[8];
    int adjNodes[4];
    int recieve_buffer[4];
    char buf[MAXN];
    MPI_Request request;
    int numMsg=0;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);

    //generate a random number for 1000 iterations
    for  (int j = 0; j < 1000; j++){
        srandom(time(NULL) + rank + j);
        int min = 1;
        int max=10;
        int ranNum = (random()%(max-min+1))+min;
        numRequests = 0;
        MPI_Cart_shift(comm, 0, 1, &adjNodes[0], &adjNodes[1]); //get adjacent nodes
        MPI_Cart_shift(comm, 1, 1, &adjNodes[2], &adjNodes[3]); //get adjacent nodes

        //sending messages the adjacent nodes
        for (int i=0; i<4; i++){
                if(adjNodes[i] >= 0){
                    MPI_Isend(&ranNum, 1, MPI_INT, adjNodes[i], 0, comm, &array_of_requests[numRequests++]);
                    numMsg++;
                    MPI_Irecv(&recieve_buffer[i], 1, MPI_INT, adjNodes[i], 0, comm, &array_of_requests[numRequests++]);
                }
            }

        MPI_Waitall(numRequests, array_of_requests, array_of_statuses);//Waits for all given MPI Requests to complete


        if (eventChecker(recieve_buffer)==1){ //if theres an event there
            sprintf(buf, "Node: %d   Neigbhours:[%d, %d, %d, %d]\n",
                rank, adjNodes[0], adjNodes[1], adjNodes[2], adjNodes[3]);
            MPI_Isend(buf, MAXN, MPI_CHAR, 0, EVENT, master_comm, &request); //non-blocking send to base station
            numMsg++; //increment the total messages
        }
    }

    //to end simulation, a finish request is sent
    //the total messages(num messages) are collected in MPI_reduce
    MPI_Isend(buf, MAXN, MPI_CHAR, 0, EXIT, master_comm, &request);
    numMsg++;
    MPI_Reduce(&numMsg, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

Figure 5: node function (slaves)

EventChecker Function:

This function is used to check if an 'event' occurred at any particular node. The node function calls this function to check if there is an event. An event occurs at a node if three or more adjacent nodes output the same random generated number.

Comparison is done twice where the first node is compared to the other adjacent nodes and and returns 'True' (1) if it the number of events is greater than or equal to three. The process is repeated a second time. Figure 6 below demonstrates the code.

```
int eventChecker(int array[4]){
    int simulation1=0;
    int simulation2=0;

    for (int i=0; i<=3;i++){
        if (array[i] == array[0]){ //compare each nodes values with the first value
            simulation1++; //increment the simualation counter
        }
    }

    if (simulation1 >=3){
        return 1; //return true if there are 3 or more events
    }

    for (int i=0; i<=3;i++){
        if (array[i] == array[1]){ //compare each nodes values with the second value
            simulation2++; //increment the simulation counter
        }
    }

    if (simulation2 >=3){
        return 1; //return true if there are 3 or more events
    }else{
        return 0; //return false if not
    }
}
```
Figure 6: Event Checker

	A	B	C	D	E
1	A1	B1	C1	D1	E1
2	A2	B2	C2	D2	E2
3	A3	B3	C3	D3	E3
4	A4	B4	C4	D4	E4

Figure 7: An example of a 4x6 grid

Inter-process communication scheme

Communication scheme between:

a) Adjacent nodes in a WSN: In order to make a 4x5 grid, MPI_Cart_create was used which simply makes a new communicator to which topology information has been attached.
The nodes further use MPI_Cart_shift in order to determine their adjacent nodes and to exchange messages. An asynchronous communication method like the MPI_Isend is used for non-blocking send request and MPI_Irecv for receiving a non-blocking request. MPI_Waitall is used to wait for all given MPI Requests to complete.

b) Node and Base station: The node (slaves) and the base station (master) were split using MPI_Comm_split.

MPI_Isend is used for non-blocking send request to send messages from the nodes to the base station. MPI_Reduce is used to convert into a single value and send to the Base station.

The Base station receives messages via synchronous blocking: MPI_Recv. It reduces to a single value and outputs it into a file.

Compilation and usage instructions

In order to compile the program, in the terminal use the command:
mpicc -o assign2 assign2.c

To run the program, use the command:
Mpirun -np 21 assign2

```
Node Event: 14   Adjacent Nodes:[10, 18, 13, 15]
Node Event: 13   Adjacent Nodes:[9, 17, 12, 14]
Node Event: 12   Adjacent Nodes:[8, 16, -2, 13]
Node Event: 10   Adjacent Nodes:[6, 14, 9, 11]
Node Event: 9   Adjacent Nodes:[5, 13, 8, 10]
Node Event: 8   Adjacent Nodes:[4, 12, -2, 9]
Node Event: 6   Adjacent Nodes:[2, 10, 5, 7]
Node Event: 5   Adjacent Nodes:[1, 9, 4, 6]
Node Event: 4   Adjacent Nodes:[0, 8, -2, 5]
Time Taken: 0.019901
Events:365
Messages: 62385
```

Figure 8: Log File Output

From the Figure 8 above, the following information can be seen:
a) Number of events occurred throughout the network.
b) Number of adjacent nodes which activated an event at the reference node.
c) Time Taken
d) Details of nodes involved in each of the events (reference node and its adjacent nodes).
e) Number of messages