MONASH University
Information Technology

GROUP ASSIGNMENT COVER SHEET

Student ID Number	Surname	Given Names
31248403	Pay	Quan Bi
32236247	Low	Jun Jie

* Please include the names of all other group members.

Unit name and code	FIT3143 Parallel Computing	
Title of assignment	SEISMIC READINGS FOR EARTHQUAKE DETECTION IN A DISTRIBUTED WIRELESS SENSOR NETWORK (WSN)	
Lecturer/tutor	Dr Vishnu / Ms Jessie	
Tutorial day and time	Wednesday 12pm - 2pm	**Campus** Monash University Malaysia
Is this an authorised group assignment?	[✓ Yes] ☐ No	
Has any part of this assignment been previously submitted as part of another unit/course?	☐ Yes [✓ No]	
Due Date 18 / 10 / 2022 23:55 (Malaysian Time)		**Date submitted 18 / 10 / 2022**

All work must be submitted by the due date. If an extension of work is granted this must be specified with the signature of the lecturer/tutor.

Extension granted until (date) .... **Signature of lecturer/tutor** ....

Please note that it is your responsibility to retain copies of your assessments.

Intentional plagiarism or collusion amounts to cheating under Part 7 of the Monash University (Council) Regulations

Plagiarism: Plagiarism means taking and using another person's ideas or manner of expressing them and passing them off as one's own. For example, by failing to give appropriate acknowledgement. The material used can be from any source (staff, students or the internet, published and unpublished works).

Collusion: Collusion means unauthorised collaboration with another person on assessable written, oral or practical work and includes paying another person to complete all or part of the work.

Where there are reasonable grounds for believing that intentional plagiarism or collusion has occurred, this will be reported to the Associate Dean (Education) or delegate, who may disallow the work concerned by prohibiting assessment or refer the matter to the Faculty Discipline Panel for a hearing.

Student Statement:

- I have read the university's Student Academic Integrity Policy and Procedures.
- I understand the consequences of engaging in plagiarism and collusion as described in Part 7 of the Monash University (Council) Regulations http://adm.monash.edu/legal/legislation/statutes
- I have taken proper care to safeguard this work and made all reasonable efforts to ensure it could not be copied.
- No part of this assignment has been previously submitted as part of another unit/course.
- I acknowledge and agree that the assessor of this assignment may for the purposes of assessment, reproduce the assignment and:
  i. provide to another member of faculty and any external marker; and/or
  ii. submit it to a text matching software; and/or
  iii. submit it to a text matching software which may then retain a copy of the assignment on its database for the purpose of future plagiarism checking.
- I certify that I have not plagiarised the work of others or participated in unauthorised collaboration when preparing this assignment.

Signature ....Pay.. Date.........18 / 10 / 2022.....................

* delete (iii) if not applicable

Signature ____Low Jun Jie_____Date:_____18/10/2022______

Signature ______Pay_____________Date:_____18 / 10 / 2022_______

Privacy Statement
The information on this form is collected for the primary purpose of assessing your assignment and ensuring the academic integrity requirements of the University are met. Other purposes of collection include recording your plagiarism and collusion declaration, attending to course and administrative matters and statistical analyses. If you choose not to complete all the questions on this form it may not be possible for Monash University to assess your assignment. You have a right to access personal information that Monash University holds about you, subject to any exceptions in relevant legislation. If you wish to seek access to your personal information or inquire about the handling of your personal information, please contact the University Privacy Officer: privacyofficer@adm.monash.edu.au

FIT3143 Semester 2, 2022
Assignment 2 - Report

Team Name (or Number): 5

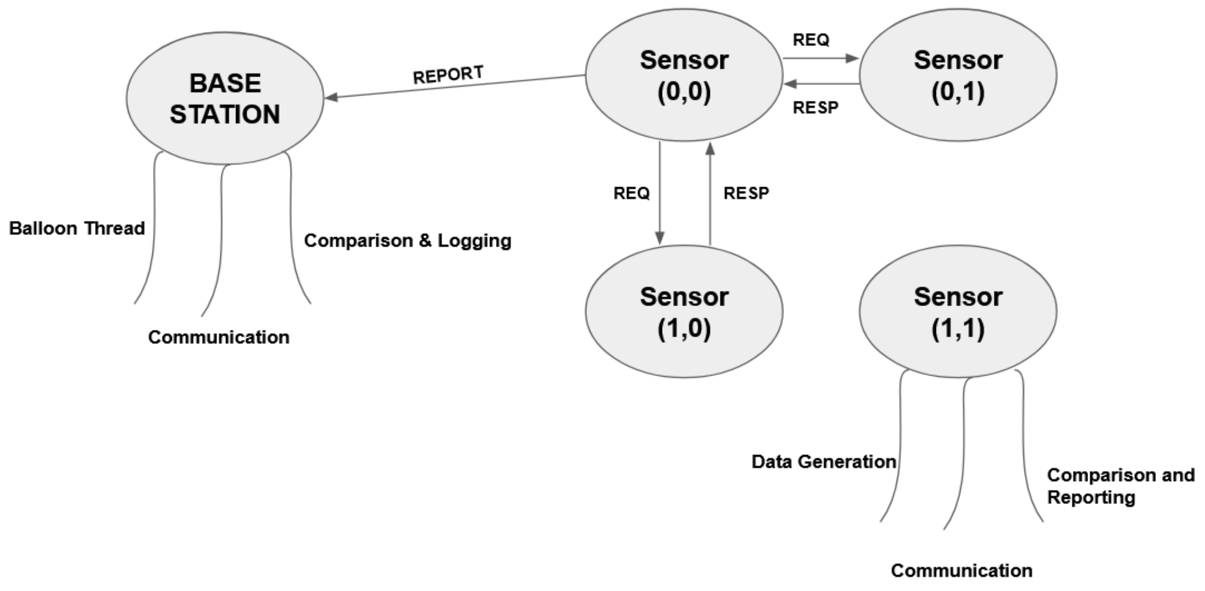
Student email address	Student First Name	Student Last Name	Contribution %	Contribution details*
qpay0001@student.monash.edu	Quan Bi	Pay	50	Base Station and Seismic Ballon Sensor and Fault Detection on Base Station. Code debugging on the overall system. Report is written and proofread together.
jlow0030@student.monash.edu	Jun Jie	Low	50	Seismic Sensor Nodes and Fault Detection on Sensor Nodes. Code debugging on the overall system. Report is written and proofread together.

**Your contribution details include the report, code, or both.*

Note: Please refer to Assignment specifications, FAQ and marking guidelines for details to be included in the following sections of this report.

Include the word count here (for Sections A to C): 1627 Words

A. Methodology



Above diagram describes the system architecture. The *Base Station* and *Sensors* spawns 3 threads each with functionalities described below. Each Thread will communicate using a shared memory paradigm. The sample communication flow is compatible with the specifications.

1) **Base Station**

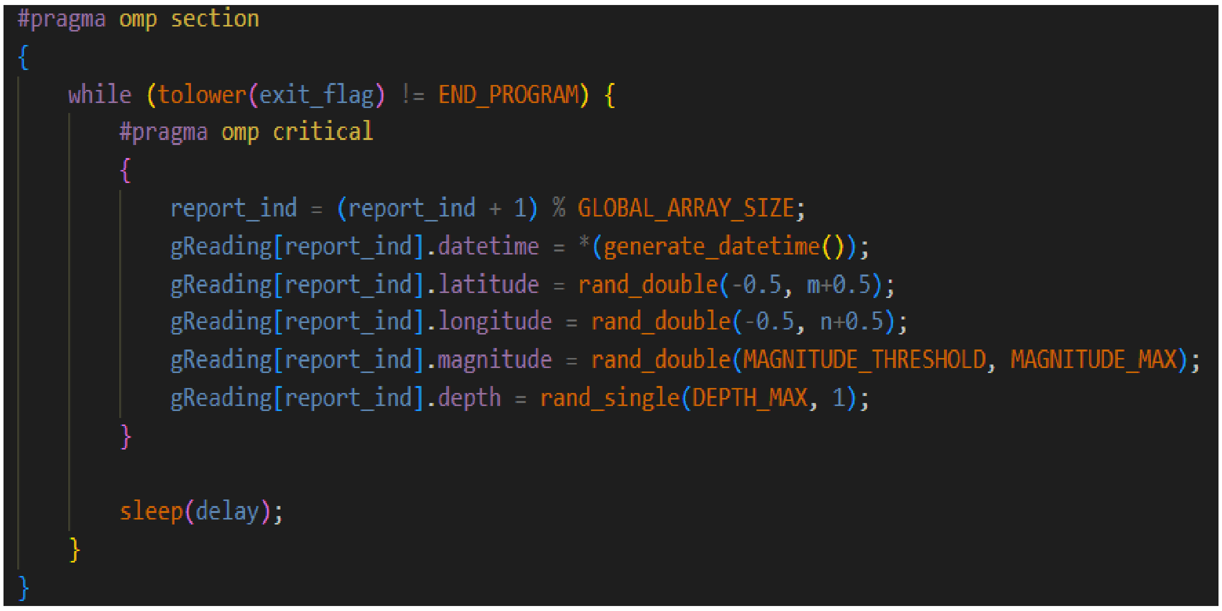
a) **Balloon Sensor**

This is a thread spawned by the base station process in order to generate the readings. Note that a circular queue-like array is implemented using 2 variables, pointer to the position of the current element in the array and the size of the array. To exhibit the circular queue property, modulus operation is used to make sure that the pointer is within the range, 0 - size of array.

Implementation

1. Spawn a thread from master thread to handle the balloon sensor
2. Check whether there is a termination signal from the master thread
3. If there is a termination signal from the master thread, join the thread
4. Else, generate the readings and populate the readings to the global array
5. Update the pointer to the next position
6. Sleep the thread for specified delay to exhibit the effect of periodically generating the readings
7. Repeat step 4 to 8

The following screenshot shows the main implementation of the balloon sensor, i.e. generating reading periodically. To record the reading, a struct is then used to store the necessary information. For detailed implementation of the struct, refer to struct_data.c. Note that threads are using shared memory architecture and hence communication is done using shared variables, i.e. exit_flag for termination signal.



```
#pragma omp section
{
    while (tolower(exit_flag) != END_PROGRAM) {
        #pragma omp critical
        {
            report_ind = (report_ind + 1) % GLOBAL_ARRAY_SIZE;
            gReading[report_ind].datetime = *(generate_datetime());
            gReading[report_ind].latitude = rand_double(-0.5, m+0.5);
            gReading[report_ind].longitude = rand_double(-0.5, n+0.5);
            gReading[report_ind].magnitude = rand_double(MAGNITUDE_THRESHOLD, MAGNITUDE_MAX);
            gReading[report_ind].depth = rand_single(DEPTH_MAX, 1);
        }

        sleep(delay);
    }
}
```

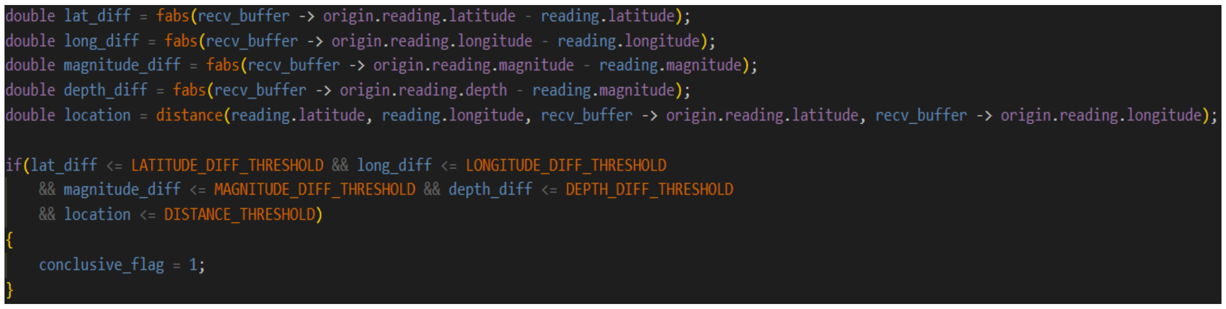
b) **Logging Thread**

This is another thread spawned by the base station process to perform comparisons on the receiving data and log the alert into a file. Similar to the balloon thread, it will continue the task until a termination signal is initiated. Note that this thread can be viewed as the master thread in the base station process since it also handles the termination signal, i.e. taking input from the user.

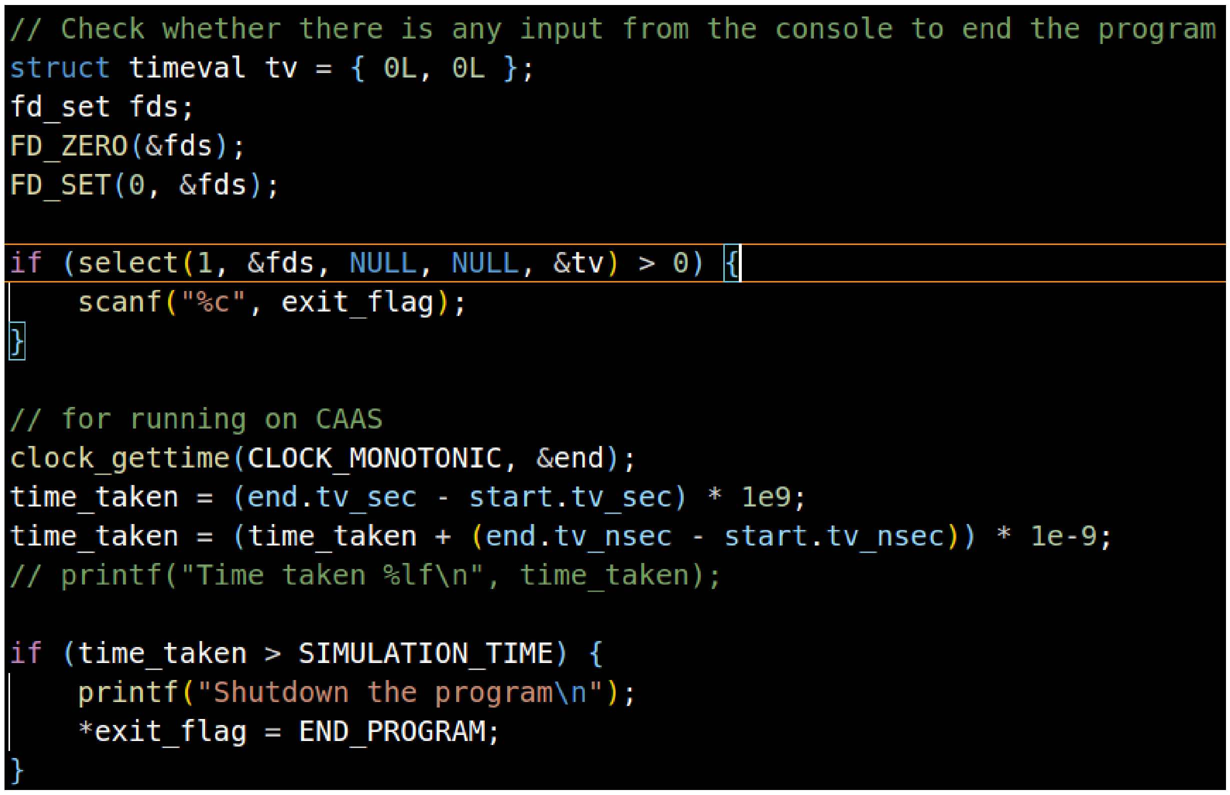
Implementation

1. Spawn a thread from the process to handle the comparison and logging task
2. Check if there is an incoming alert, marked using a flag.
3. If there is an incoming alert, compare the readings against the latest reading from the balloon sensor.
4. Mark the alert as conclusive if the incoming alert passes the comparison. Otherwise, it is inconclusive.
5. Log the incoming alert in specified format into a text file.
6. Check if there is a termination input from the user (local) / simulation time reached (CAAS). If there is a termination request, join the threads and send termination signals to the sensors. Else, repeat steps 2 to 6.

The following screenshot shows the implementation of how the comparison is done. Note that the alert is received in the form of a struct that contains all necessary information for comparison and logging. The second screenshot shows the termination signal generated by the user / time threshold. Note that to make sure input from the user does not block other operations, select() is used to set a timeout for waiting input from the user. Then, on CAAS, we simply calculate the running time against total simulation time.

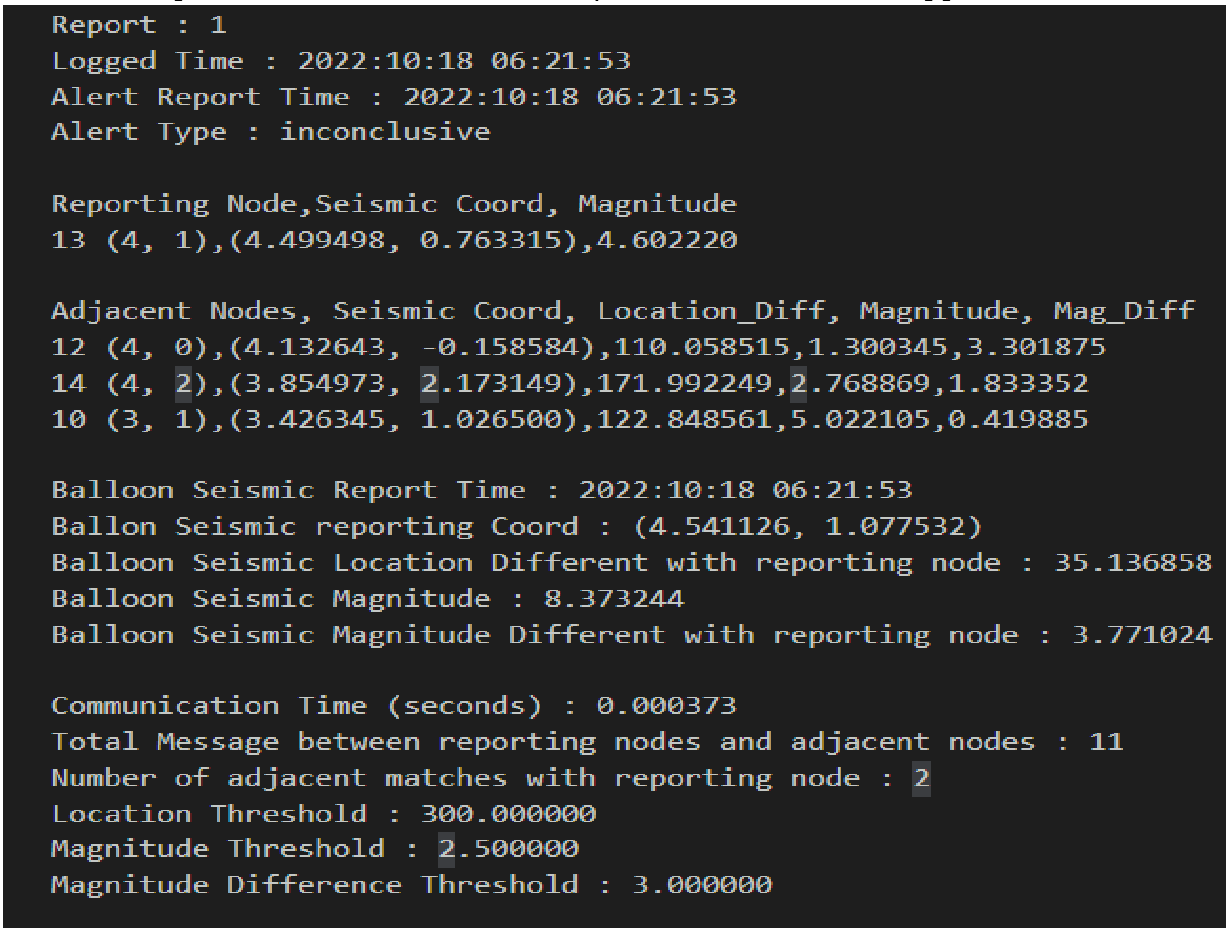


```c
double lat_diff = fabs(recv_buffer -> origin.reading.latitude - reading.latitude);
double long_diff = fabs(recv_buffer -> origin.reading.longitude - reading.longitude);
double magnitude_diff = fabs(recv_buffer -> origin.reading.magnitude - reading.magnitude);
double depth_diff = fabs(recv_buffer -> origin.reading.depth - reading.magnitude);
double location = distance(reading.latitude, reading.longitude, recv_buffer -> origin.reading.latitude, recv_buffer -> origin.reading.longitude);

if(lat_diff <= LATITUDE_DIFF_THRESHOLD && long_diff <= LONGITUDE_DIFF_THRESHOLD
    && magnitude_diff <= MAGNITUDE_DIFF_THRESHOLD && depth_diff <= DEPTH_DIFF_THRESHOLD
    && location <= DISTANCE_THRESHOLD)
{
    conclusive_flag = 1;
}
```



```c
// Check whether there is any input from the console to end the program
struct timeval tv = { 0L, 0L };
fd_set fds;
FD_ZERO(&fds);
FD_SET(0, &fds);

if (select(1, &fds, NULL, NULL, &tv) > 0) {
    scanf("%c", exit_flag);
}

// for running on CAAS
clock_gettime(CLOCK_MONOTONIC, &end);
time_taken = (end.tv_sec - start.tv_sec) * 1e9;
time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
// printf("Time taken %lf\n", time_taken);

if (time_taken > SIMULATION_TIME) {
    printf("Shutdown the program\n");
    *exit_flag = END_PROGRAM;
}
```

The following screenshot shows the example of how an alert is logged.

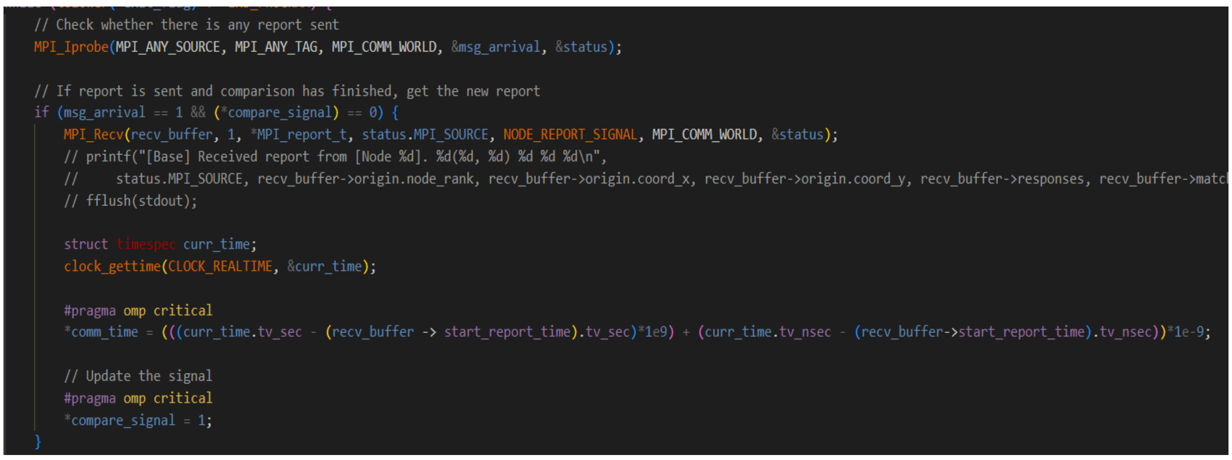


```
Report : 1
Logged Time : 2022:10:18 06:21:53
Alert Report Time : 2022:10:18 06:21:53
Alert Type : inconclusive

Reporting Node,Seismic Coord, Magnitude
13 (4, 1),(4.499498, 0.763315),4.602220

Adjacent Nodes, Seismic Coord, Location_Diff, Magnitude, Mag_Diff
12 (4, 0),(4.132643, -0.158584),110.058515,1.300345,3.301875
14 (4, 2),(3.854973, 2.173149),171.992249,2.768869,1.833352
10 (3, 1),(3.426345, 1.026500),122.848561,5.022105,0.419885

Balloon Seismic Report Time : 2022:10:18 06:21:53
Ballon Seismic reporting Coord : (4.541126, 1.077532)
Balloon Seismic Location Different with reporting node : 35.136858
Balloon Seismic Magnitude : 8.373244
Balloon Seismic Magnitude Different with reporting node : 3.771024

Communication Time (seconds) : 0.000373
Total Message between reporting nodes and adjacent nodes : 11
Number of adjacent matches with reporting node : 2
Location Threshold : 300.000000
Magnitude Threshold : 2.500000
Magnitude Difference Threshold : 3.000000
```

c) **Communication Thread**
This thread is spawned by the base station process for communication purposes. It will handle all sending and receiving messages except the termination signal which is done after joining the threads. Note that in order to handle the messages efficiently, we use a probing mechanism. We only call MPI_Recv when it is necessary. Then, we notify the master thread that there is an incoming alert. Before the master thread logs the alert, we will not be accepting any new alert from the buffer to avoid race conditions.

Implementation
1. Spawn a thread to handle communication purposes
2. Probe the incoming alert
3. If there is an alert, accept the alert only when the master thread is not logging. Else, keep the alert in the buffer.
4. Calculate the communication time for the incoming alert
5. Signal that the alert is accepted and logging can be done
6. Repeat steps 2 to 5 until the termination signal is fired.

The following shows the implementation of the above algorithm. We have flag variables in shared memory for communication between the threads, i.e. signalling.



```c
    // Check whether there is any report sent
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &msg_arrival, &status);

    // If report is sent and comparison has finished, get the new report
    if (msg_arrival == 1 && (*compare_signal) == 0) {
        MPI_Recv(recv_buffer, 1, *MPI_report_t, status.MPI_SOURCE, NODE_REPORT_SIGNAL, MPI_COMM_WORLD, &status);
        // printf("[Base] Received report from [Node %d]. %d(%d, %d) %d %d %d\n",
        //     status.MPI_SOURCE, recv_buffer->origin.node_rank, recv_buffer->origin.coord_x, recv_buffer->origin.coord_y, recv_buffer->responses, recv_buffer->matc
        // fflush(stdout);

        struct timespec curr_time;
        clock_gettime(CLOCK_REALTIME, &curr_time);

        #pragma omp critical
        *comm_time = (((curr_time.tv_sec - (recv_buffer -> start_report_time).tv_sec)*1e9) + (curr_time.tv_nsec - (recv_buffer->start_report_time).tv_nsec))*1e-9;

        // Update the signal
        #pragma omp critical
        *compare_signal = 1;
    }
```

In general, combining the above 3 algorithms, we have the complete base station function.

Implementation

1. Initialize shared variables, including the global array.
2. Spawn the threads for each task described above.
3. Join the threads once termination signal is received from the user.
4. Send the termination signal to other processes using MPI
5. Close the program

2) Seismic Sensor Node

The sensors are created and connected according to the cartesian topology. Each of them have the following threads for handling different tasks.

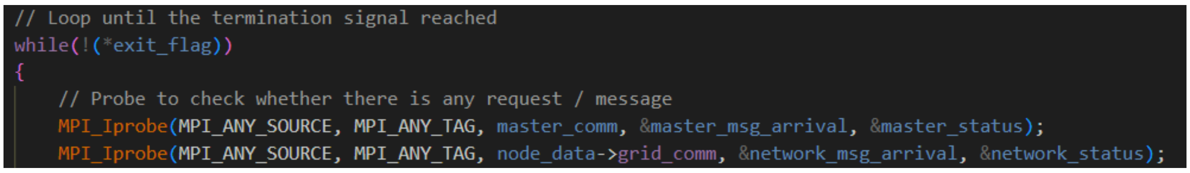
a) Communication Thread

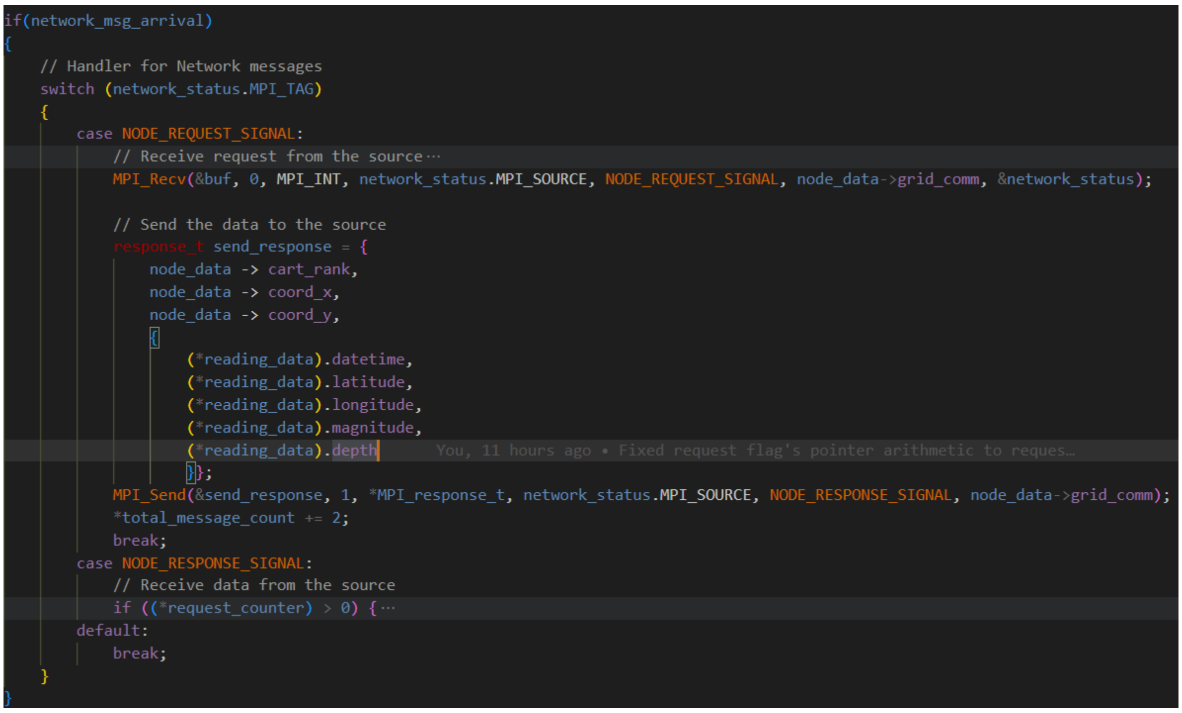
This is a thread spawned by the Seismic Sensor Nodes to handle all communication. For receiving, it will constantly probe if there are any incoming messages, and handle them accordingly. As for sending, it will constantly check the signal for sending messages and call MPI_Send when necessary.

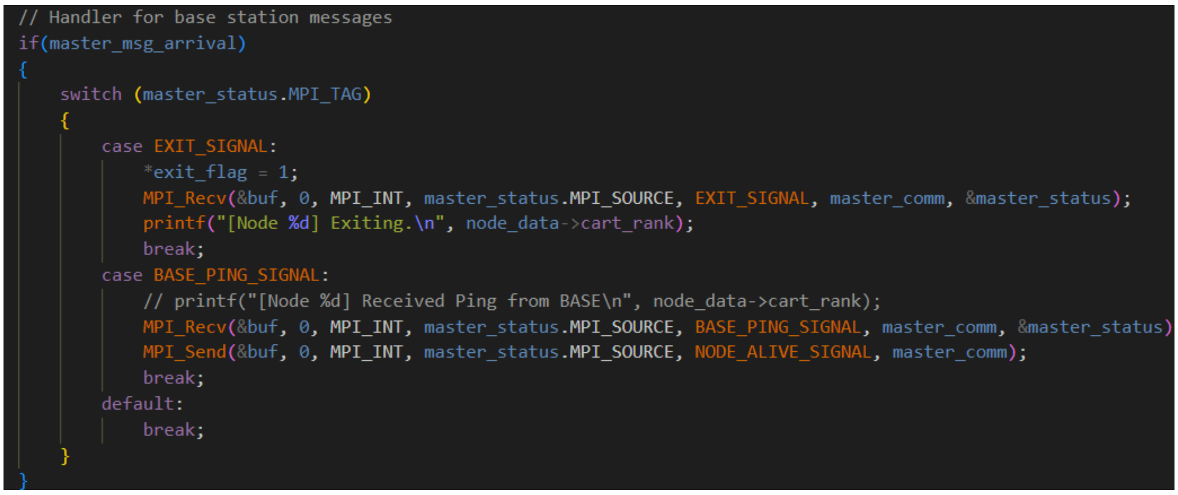
Implementation

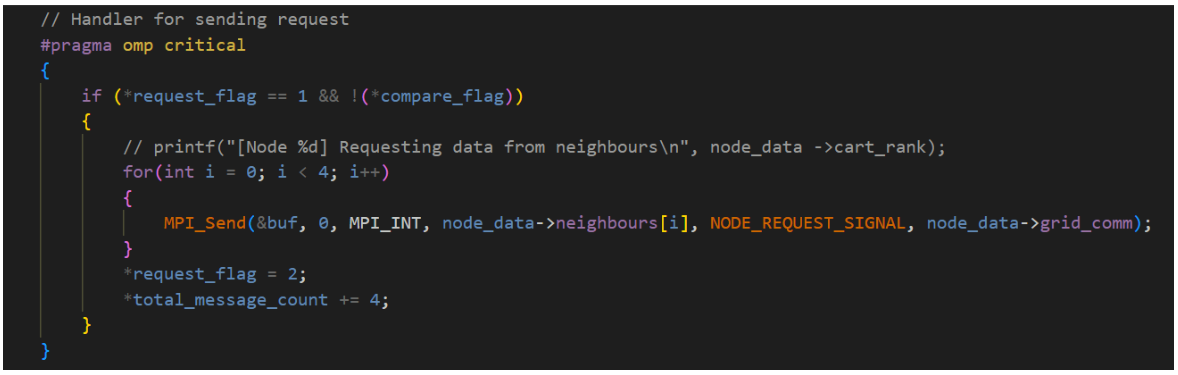
1. Non-blocking probe for incoming messages from neighbours and Base Station
    a. If incoming message
        i. Receive message and handle based on message tag.
    b. Else continue on
2. Check flags for sending messages
    a. If flags active
        i. Send the buffers that are presumably filled. Reset flag.
    b. Else continue on

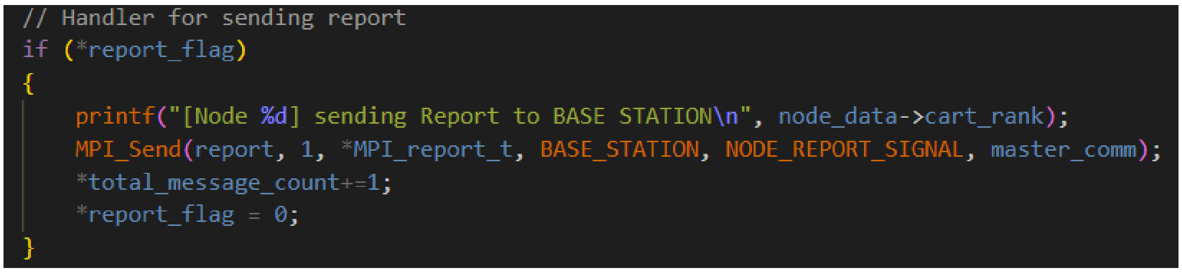
The following screenshot shows the implementation. We would have to probe for 2 types of communication, i.e. from the base station or from the neighbours and handle them accordingly. The remaining screenshots tell how we handle different messages according to the specifications, including sending messages.


```
// Loop until the termination signal reached
while(!(*exit_flag))
{
    // Probe to check whether there is any request / message
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, master_comm, &master_msg_arrival, &master_status);
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, node_data->grid_comm, &network_msg_arrival, &network_status);
```


```
if(network_msg_arrival)
{
    // Handler for Network messages
    switch (network_status.MPI_TAG)
    {
        case NODE_REQUEST_SIGNAL:
            // Receive request from the source···
            MPI_Recv(&buf, 0, MPI_INT, network_status.MPI_SOURCE, NODE_REQUEST_SIGNAL, node_data->grid_comm, &network_status);

            // Send the data to the source
            response_t send_response = {
                node_data -> cart_rank,
                node_data -> coord_x,
                node_data -> coord_y,
                {
                    (*reading_data).datetime,
                    (*reading_data).latitude,
                    (*reading_data).longitude,
                    (*reading_data).magnitude,
                    (*reading_data).depth       You, 11 hours ago • Fixed request flag's pointer arithmetic to reques…
                }};
            MPI_Send(&send_response, 1, *MPI_response_t, network_status.MPI_SOURCE, NODE_RESPONSE_SIGNAL, node_data->grid_comm);
            *total_message_count += 2;
            break;
        case NODE_RESPONSE_SIGNAL:
            // Receive data from the source
            if ((*request_counter) > 0) {···
        default:
            break;
    }
}
```


```
// Handler for base station messages
if(master_msg_arrival)
{
    switch (master_status.MPI_TAG)
    {
        case EXIT_SIGNAL:
            *exit_flag = 1;
            MPI_Recv(&buf, 0, MPI_INT, master_status.MPI_SOURCE, EXIT_SIGNAL, master_comm, &master_status);
            printf("[Node %d] Exiting.\n", node_data->cart_rank);
            break;
        case BASE_PING_SIGNAL:
            // printf("[Node %d] Received Ping from BASE\n", node_data->cart_rank);
            MPI_Recv(&buf, 0, MPI_INT, master_status.MPI_SOURCE, BASE_PING_SIGNAL, master_comm, &master_status)
            MPI_Send(&buf, 0, MPI_INT, master_status.MPI_SOURCE, NODE_ALIVE_SIGNAL, master_comm);
            break;
        default:
            break;
    }
}
```


```
    // Handler for sending request
    #pragma omp critical
    {
        if (*request_flag == 1 && !(*compare_flag))
        {
            // printf("[Node %d] Requesting data from neighbours\n", node_data ->cart_rank);
            for(int i = 0; i < 4; i++)
            {
                MPI_Send(&buf, 0, MPI_INT, node_data->neighbours[i], NODE_REQUEST_SIGNAL, node_data->grid_comm);
            }
            *request_flag = 2;
            *total_message_count += 4;
        }
    }
```



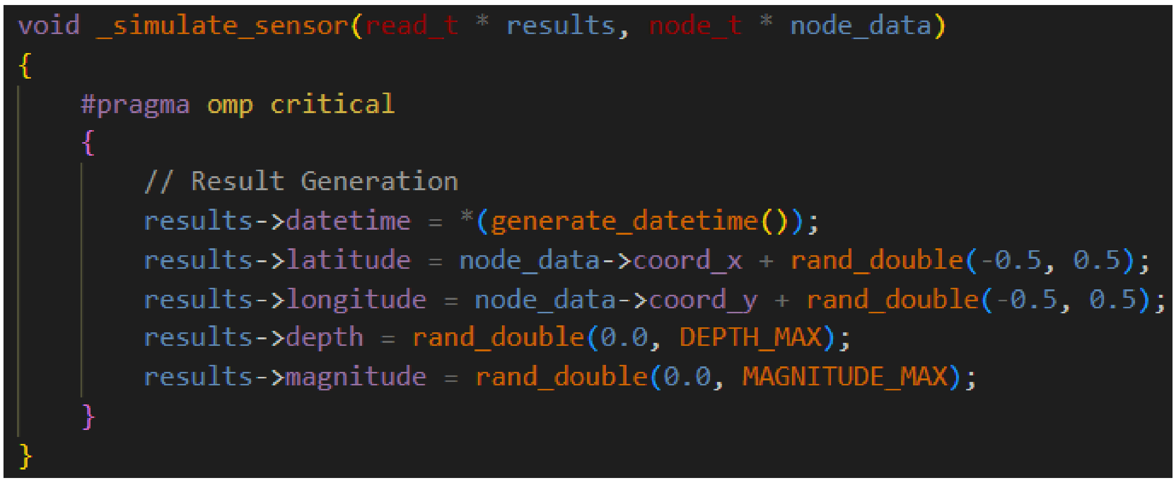
```
// Handler for sending report
if (*report_flag)
{
    printf("[Node %d] sending Report to BASE STATION\n", node_data->cart_rank);
    MPI_Send(report, 1, *MPI_report_t, BASE_STATION, NODE_REPORT_SIGNAL, master_comm);
    *total_message_count+=1;
    *report_flag = 0;
}
```

b) Data Generation Thread

This is a thread that handles data generation for Seismic Sensors. It generates readings into a shared memory for the threads. To prevent race conditions, the update is encapsulated in a critical region. Then it will initiate a request to the neighbours if the readings pass the threshold.

Implementation

1. Generate data into the preallocated readings buffer periodically.
2. If generated data is above the threshold set
    a. Set request flag to mark for **Communication Thread** to send request signals to neighbours



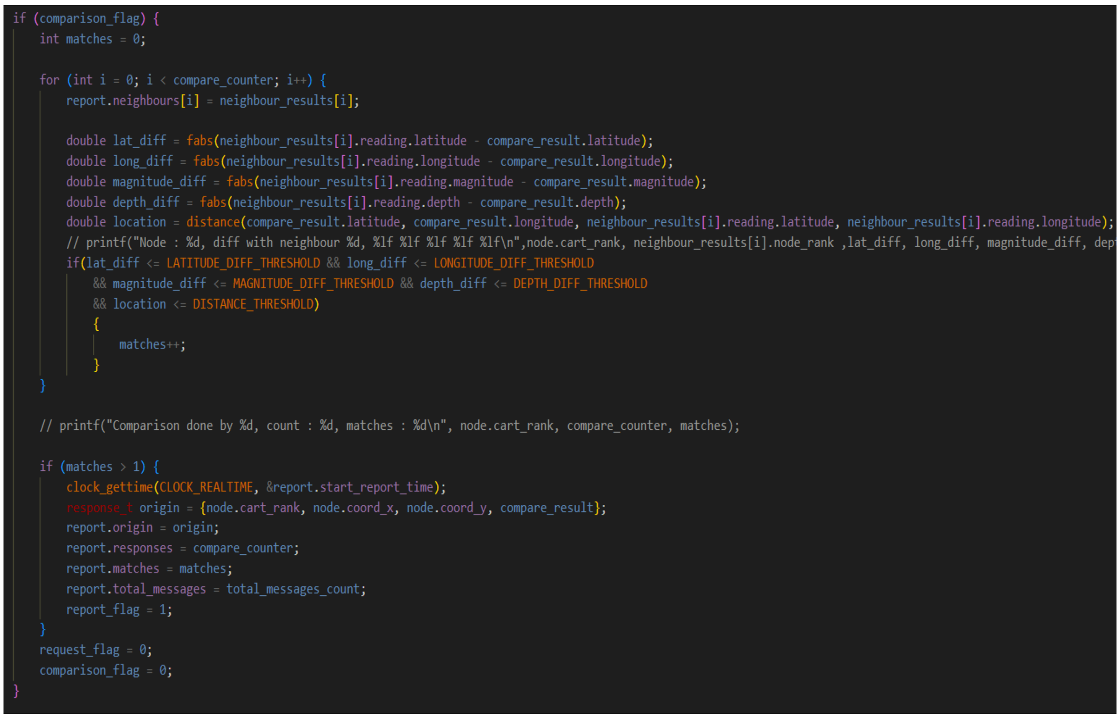
```
void _simulate_sensor(read_t * results, node_t * node_data)
{
    #pragma omp critical
    {
        // Result Generation
        results->datetime = *(generate_datetime());
        results->latitude = node_data->coord_x + rand_double(-0.5, 0.5);
        results->longitude = node_data->coord_y + rand_double(-0.5, 0.5);
        results->depth = rand_double(0.0, DEPTH_MAX);
        results->magnitude = rand_double(0.0, MAGNITUDE_MAX);
    }
}
```

c) Comparing Thread

This thread handles all the comparison of results and prepares the report to send to Base Station if necessary. If there are readings available, it will compare the values against its own reading. If there are more than 1 neighbouring node with compatible results, it will send the report to Base Station.

Implementation

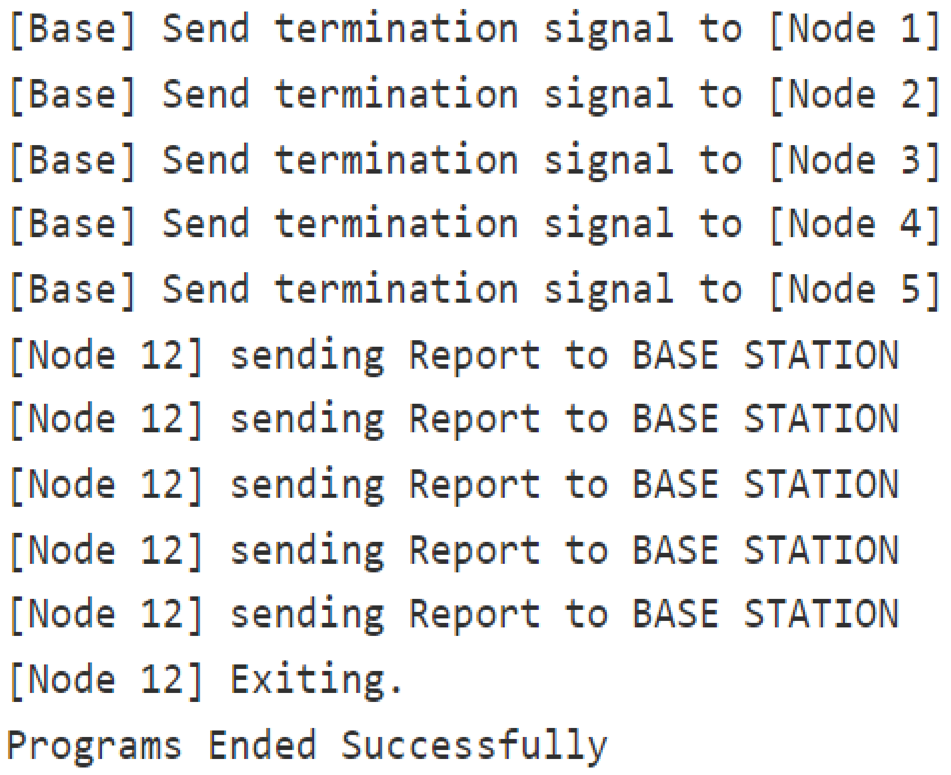
1. Check Compare Flag
    a. If set
        i. Compare with local readings, count if values within threshold.
        ii. Prepare a report and set Report flag to wait for **Communication Thread** to send the report to Base Station.
    b. not set continue next iteration



```c
if (comparison_flag) {
    int matches = 0;

    for (int i = 0; i < compare_counter; i++) {
        report.neighbours[i] = neighbour_results[i];

        double lat_diff = fabs(neighbour_results[i].reading.latitude - compare_result.latitude);
        double long_diff = fabs(neighbour_results[i].reading.longitude - compare_result.longitude);
        double magnitude_diff = fabs(neighbour_results[i].reading.magnitude - compare_result.magnitude);
        double depth_diff = fabs(neighbour_results[i].reading.depth - compare_result.depth);
        double location = distance(compare_result.latitude, compare_result.longitude, neighbour_results[i].reading.latitude, neighbour_results[i].reading.longitude);
        // printf("Node : %d, diff with neighbour %d, %lf %lf %lf %lf %lf\n",node.cart_rank, neighbour_results[i].node_rank ,lat_diff, long_diff, magnitude_diff, dep
        if(lat_diff <= LATITUDE_DIFF_THRESHOLD && long_diff <= LONGITUDE_DIFF_THRESHOLD
            && magnitude_diff <= MAGNITUDE_DIFF_THRESHOLD && depth_diff <= DEPTH_DIFF_THRESHOLD
            && location <= DISTANCE_THRESHOLD)
            {
                matches++;
            }
    }

    // printf("Comparison done by %d, count : %d, matches : %d\n", node.cart_rank, compare_counter, matches);

    if (matches > 1) {
        clock_gettime(CLOCK_REALTIME, &report.start_report_time);
        response_t origin = {node.cart_rank, node.coord_x, node.coord_y, compare_result};
        report.origin = origin;
        report.responses = compare_counter;
        report.matches = matches;
        report.total_messages = total_messages_count;
        report_flag = 1;
    }
    request_flag = 0;
    comparison_flag = 0;
}
```

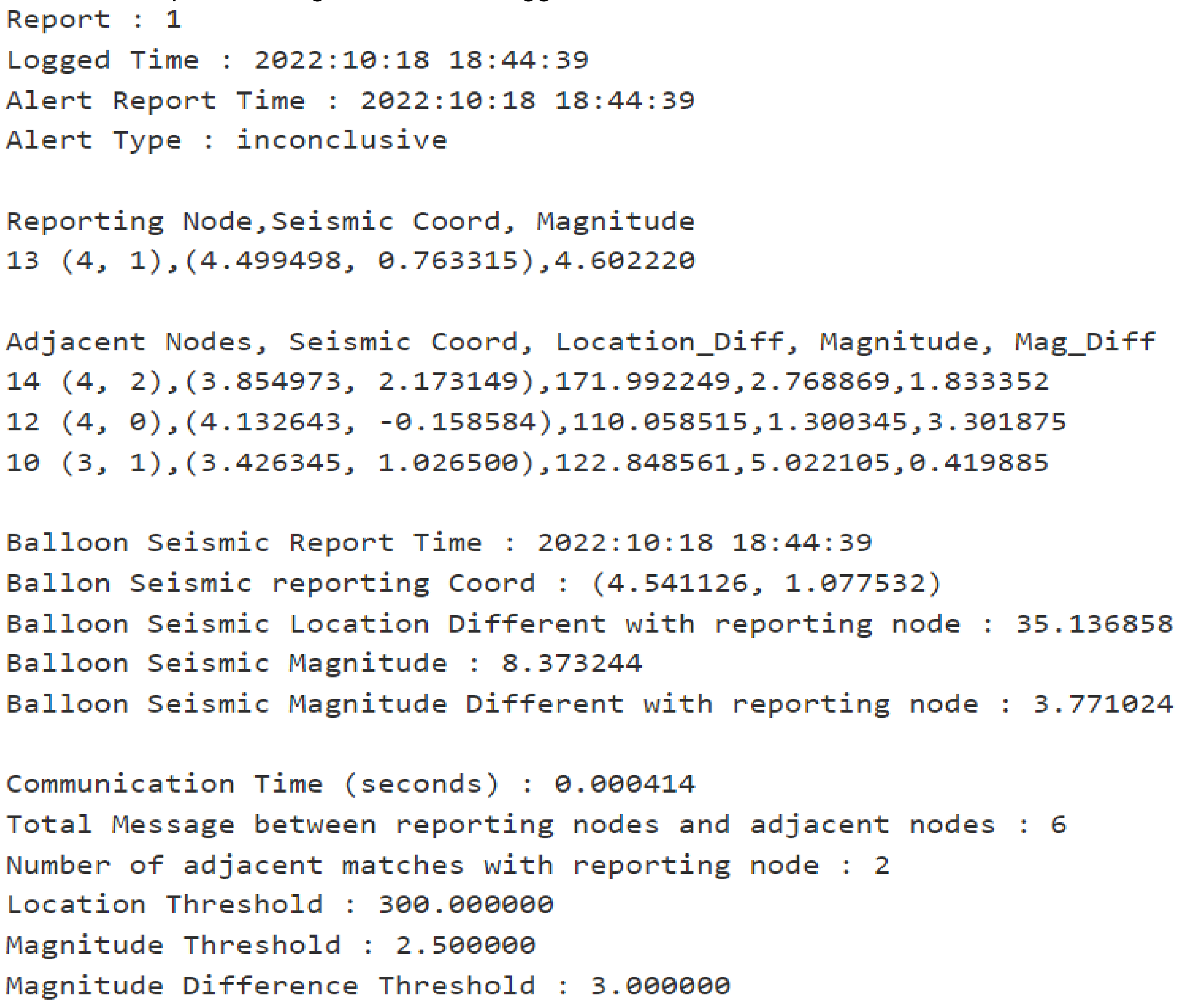
B. Results Tabulation

Below shows part of the stdout output when running on CAAS with configuration of 16 total processes and **m x n** of 5 x 3 with a timer of 120 seconds and delay of 2 seconds per reading generation. This result is run 5 times and the best is selected



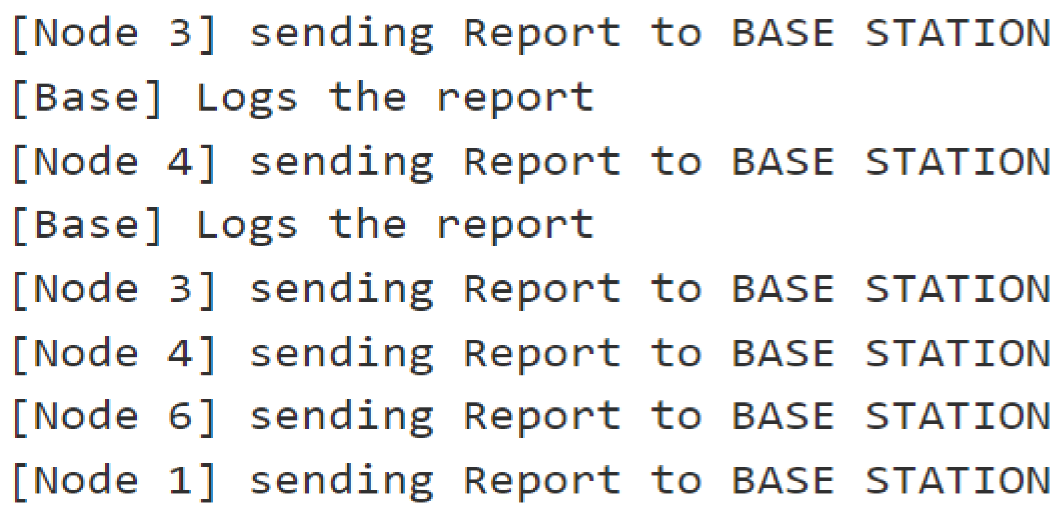
```
[Base] Send termination signal to [Node 1]
[Base] Send termination signal to [Node 2]
[Base] Send termination signal to [Node 3]
[Base] Send termination signal to [Node 4]
[Base] Send termination signal to [Node 5]
[Node 12] sending Report to BASE STATION
[Node 12] sending Report to BASE STATION
[Node 12] sending Report to BASE STATION
[Node 12] sending Report to BASE STATION
[Node 12] sending Report to BASE STATION
[Node 12] Exiting.
Programs Ended Successfully
```

Below shows the report log file after running the program on CAAS with above configurations. A total of 154 reports were generated and logged.



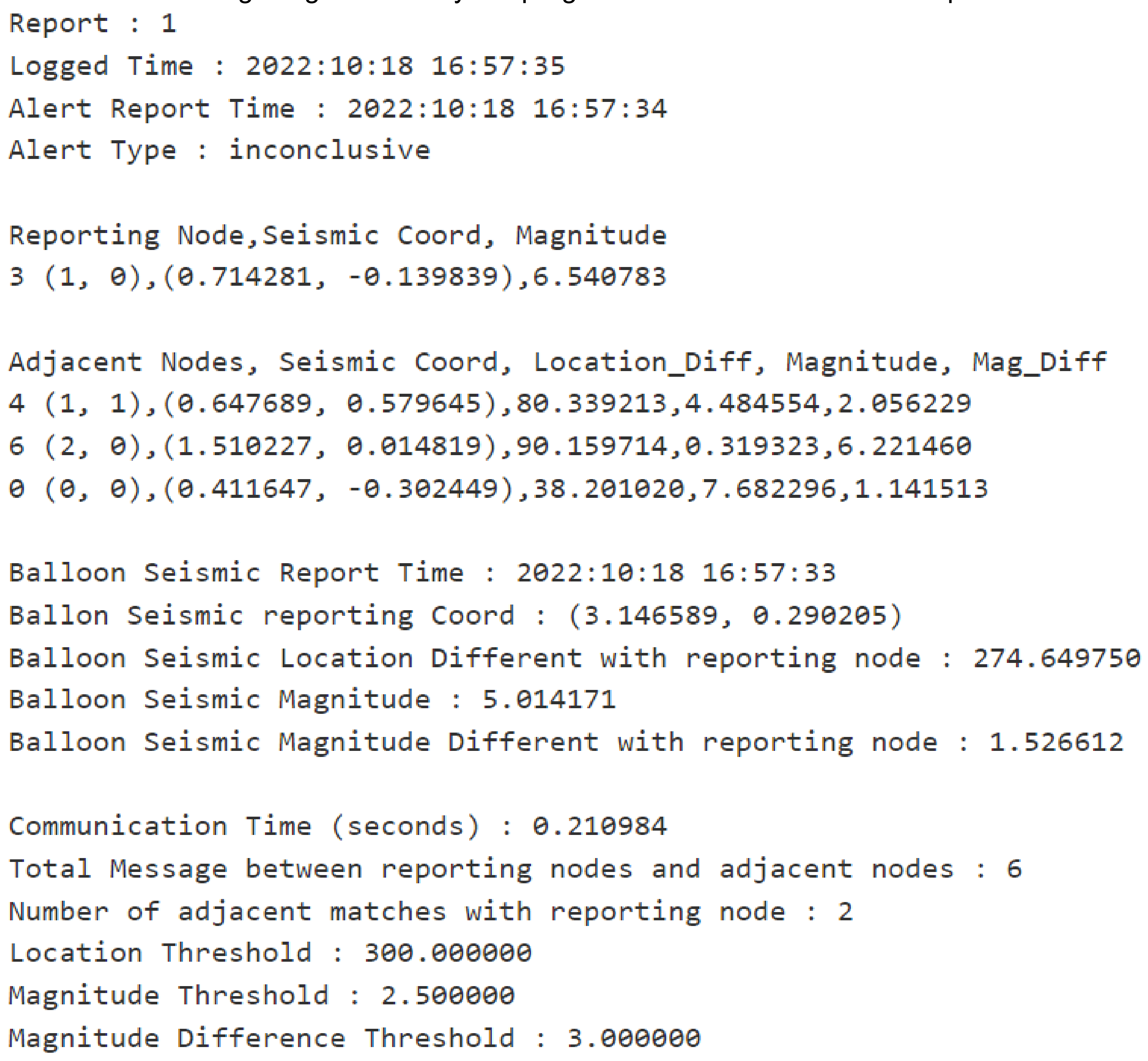
```
Report : 1
Logged Time : 2022:10:18 18:44:39
Alert Report Time : 2022:10:18 18:44:39
Alert Type : inconclusive

Reporting Node,Seismic Coord, Magnitude
13 (4, 1),(4.499498, 0.763315),4.602220

Adjacent Nodes, Seismic Coord, Location_Diff, Magnitude, Mag_Diff
14 (4, 2),(3.854973, 2.173149),171.992249,2.768869,1.833352
12 (4, 0),(4.132643, -0.158584),110.058515,1.300345,3.301875
10 (3, 1),(3.426345, 1.026500),122.848561,5.022105,0.419885

Balloon Seismic Report Time : 2022:10:18 18:44:39
Ballon Seismic reporting Coord : (4.541126, 1.077532)
Balloon Seismic Location Different with reporting node : 35.136858
Balloon Seismic Magnitude : 8.373244
Balloon Seismic Magnitude Different with reporting node : 3.771024

Communication Time (seconds) : 0.000414
Total Message between reporting nodes and adjacent nodes : 6
Number of adjacent matches with reporting node : 2
Location Threshold : 300.000000
Magnitude Threshold : 2.500000
Magnitude Difference Threshold : 3.000000
```

Below shows part of the standard output when running on a local computer with configuration of 10 processes and **m x n** of 3 x 3 with a timer of 120 seconds and delay of 5 seconds each node. This is run 5 times and the best is selected.



```
[Node 3] sending Report to BASE STATION
[Base] Logs the report
[Node 4] sending Report to BASE STATION
[Base] Logs the report
[Node 3] sending Report to BASE STATION
[Node 4] sending Report to BASE STATION
[Node 6] sending Report to BASE STATION
[Node 1] sending Report to BASE STATION
```

Below shows the log file generated by the program. There are a total of 26 reports.



```
Report : 1
Logged Time : 2022:10:18 16:57:35
Alert Report Time : 2022:10:18 16:57:34
Alert Type : inconclusive

Reporting Node,Seismic Coord, Magnitude
3 (1, 0),(0.714281, -0.139839),6.540783

Adjacent Nodes, Seismic Coord, Location_Diff, Magnitude, Mag_Diff
4 (1, 1),(0.647689, 0.579645),80.339213,4.484554,2.056229
6 (2, 0),(1.510227, 0.014819),90.159714,0.319323,6.221460
0 (0, 0),(0.411647, -0.302449),38.201020,7.682296,1.141513

Balloon Seismic Report Time : 2022:10:18 16:57:33
Ballon Seismic reporting Coord : (3.146589, 0.290205)
Balloon Seismic Location Different with reporting node : 274.649750
Balloon Seismic Magnitude : 5.014171
Balloon Seismic Magnitude Different with reporting node : 1.526612

Communication Time (seconds) : 0.210984
Total Message between reporting nodes and adjacent nodes : 6
Number of adjacent matches with reporting node : 2
Location Threshold : 300.000000
Magnitude Threshold : 2.500000
Magnitude Difference Threshold : 3.000000
```

C. Analysis & Discussion

Note that the simulation would run on parallel processes. The behavior of the entities could be summarized as follows.

1) Base Station

a) Only starts the comparison when there is an incoming alert and the reading from the balloon reading. If the balloon reading comes behind the alert, the thread would wait until the balloon reading is obtained. This would drag the efficiency but is a must for synchronization to ensure correctness.

b) The communication thread also lacks efficiency where it only probes until the message is sent to the buffer without doing other things. This could be a waste of resources. However, when there is a large amount of alerts sent in, this thread is working at its maximum capacity. Hence, the trade-off between such scenarios is not well designed. A possible solution could be checking the number of messages in the buffer and handing the resources to other threads if there are less messages in the buffer. This is the same for seismic sensors.

c) Balloon sensor thread is also not using the resources to its full capacity since it would sleep for a specified period after generating the data. This is useful when the threads are running concurrently since it would hand the resources to other threads. However, for parallelism, it is not.

2) Seismic Sensors

a) As mentioned above, constant probing is a waste of resources. It could be possible to have blocking probes on different threads for each source of message, which will block the thread until there is a message incoming, it may save computation resources, but sending and receiving messages will have to be separated into different threads as well.

b) Note that after each generation of data, the thread would sleep for a specified period. The longer the period, the less communication between the neighbours and hence the probing mechanism is inefficient in the sense that it does not utilize the resources for useful work. However, sleep is only for simulation purposes where in reality, the thread can handle other computations.

c) Probing is a good mechanism here since it can help to resolve deadlock situations. We would not have thread doing blocking receive when there is no message sent. Also, we would not want the thread not handling the messages when it is sent. Hence, the probing mechanism provides a good trade-off between them while maintaining the parallelism of the codes.

d) Due to different speeds in different processes, the sensor node may request readings before one can generate. Hence, some rubbish value from the memory would be sent over and the comparison would definitely fail. This can be solved by having all the threads generate values before sending the data according to the request at the start of the program. However, such a naive solution would always compensate for the efficiency.

Overall, the simulation works as described in the specifications. However, some details such as synchronization between the clocks and sensor nodes and comparison on readings according to the timestamp is not developed in this simulation. These topics could be explored in the future for better simulation purposes.

The following shows numerical results from the above simulation done on CAAS and Laptop.

CAAS

$$Average\ Comm\ Time\ = \frac{0.040476}{154} = 0.0002628$$

Using the above simulation results, we know that the communication time is rather subtle. This is mainly due to the long delay of each sensor node after generating the data. If the delay is shortened, the communication time would ultimately increase since the threads may not be able to handle large amounts of requests efficiently and there is a long waiting time for each request. There is some waiting time for threads to finish the comparison and log the report. Hence, the shorter the delay, the longer the communication time.

On the other hand, we note that there are only 25 conclusive reports out of 154 reports based on the simulation result. This shows that synchronization between the clocks and the nodes are extremely important in getting the correct decision. Note that we only compare the values of the nodes with the latest reading from the balloons. Due to different speed in processes, the comparison may be done on data with different timestamps and hence getting inconsistent results.

Laptop

$$Average\ Comm\ Time = \frac{12.810233}{26} = 0.4927012$$

Note that the average communication time is about 150 times slower than running on CAAS. This is because our laptop has only 4 cores and hence the threads are running concurrently using context switching instead of parallel paradigm. Hence, the communication time is longer due to the waiting time during context switching.

6 out of 26 reports are actually conclusive. This shows similar results with CAAS and it is uncertain due to the random number generator. Nevertheless, the synchronization problems persist.

D. References

Note: You may opt to customize this report to include additional sub-sections or any additional formatting where necessary.

MONASH University
Information Technology

Declaration:

I declare that this assignment report and the submitted code represent work within my team. I have not copied from any other teams' work or from any other source except where due acknowledgment is made explicitly in the report and code, nor has any part of this submission been written for me by another person outside of my team.

Signature of student 1: Pay Quan Bi

Signature of student 2: Low Jun Jie