

ASSIGNMENT COVER SHEET


Student's name	(Family name) Wong	(Given names) Bryan Jun Kit	
ID number	32882424	E-mail	bwon0018@student.monash.edu
Unit code & name	FIT3143 Parallel Computing	Unit code	FIT3143

Title of assignment	Design and Simulation of an Electric Vehicle Charging Grid System		
Lecturer/tutor			
Is this an authorised group assignment? <input type="checkbox"/> Yes <input checked="" type="checkbox"/> No If this submission is a group assignment, each student must attach their own signed cover sheet to the assignment.			
Has any part of this assignment been previously submitted as part of another unit/course? <input type="checkbox"/> Yes <input checked="" type="checkbox"/> No			
Tutorial/laboratory day & time	Tuesday 1pm		
Due date 16/10/2023		Date submitted 18/10/2023	

All work must be submitted by the due date. If an extension of time to submit work is required, a [Special Consideration Application \(In-semester Assessment Task\)](#) must be submitted.

Has an extension been approved? Yes ☒ No ☐ If yes, please give the new submission date: 18/10/2023

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

Student Statement:	<ul style="list-style-type: none"> I have read the University's Student Academic Integrity Policy and the University's Student Academic Integrity: Managing Plagiarism and Collusion Procedures. I understand the consequences of engaging in plagiarism and collusion as described in Assessment and Academic Integrity Policy I have taken proper care of safeguarding this work and made all reasonable effort to ensure it could not be copied. I acknowledge that the assessor of this assignment may for the purposes of assessment, reproduce the assignment and: <ul style="list-style-type: none"> provide to another member of faculty; and/or submit it to a plagiarism checking service; and/or submit it to a plagiarism checking service 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. <div style="text-align: center; margin-top: 20px;">  </div> <p>Signature</p> <p>Date.....18/10/2023.....</p>
Privacy Statement	<p>The information on this form is collected for the primary purpose of assessing your assignment. 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</p>

FIT3143 Semester 2, 2023

Assignment 2 – Report

Title – Design and Simulation of an Electric Vehicle Charging Grid System.

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

Include the word count here (for Sections A to C): __2337__

A. Methodology

B. Results Tabulation

C. Analysis & Discussion

D. References

1 - Methodology

1.1 - Design Overview

This section will cover the methodology used to design and simulate a Wireless Sensor Network (WSN) for an Electric Vehicle (EV) charging grid. I implemented this simulation in C, using a combination of Message Passing Interface (MPI) and POSIX threads. I chose MPI because it provides a powerful, high-performance framework for message-based communication and coordination between processes (GeekForGeeks, *MPI - Distributed Computing made easy* 2023). MPI also provides various functions to create and manipulate virtual topologies, such as Cartesian grids (Wikipedia, *Message Passing Interface* 2023), that suit the WSN model. POSIX threads are employed because they are a lightweight and efficient way to simulate the charging ports within each node. POSIX threads also allow me to share data and synchronise among threads using mutexes and condition variables (Wikipedia, *Pthreads* 2023).

The design of the WSN consists of two main components: the charging node and the base station. Each component is simulated by a single MPI process, with a unique rank assigned by MPI. The charging node processes are arranged in a Cartesian grid topology, with dimensions specified by the user at runtime. If no dimensions are specified, it will use MPI default dimensions based on the given number of processes used. The base station process will be the last rank and is not part of the grid topology.

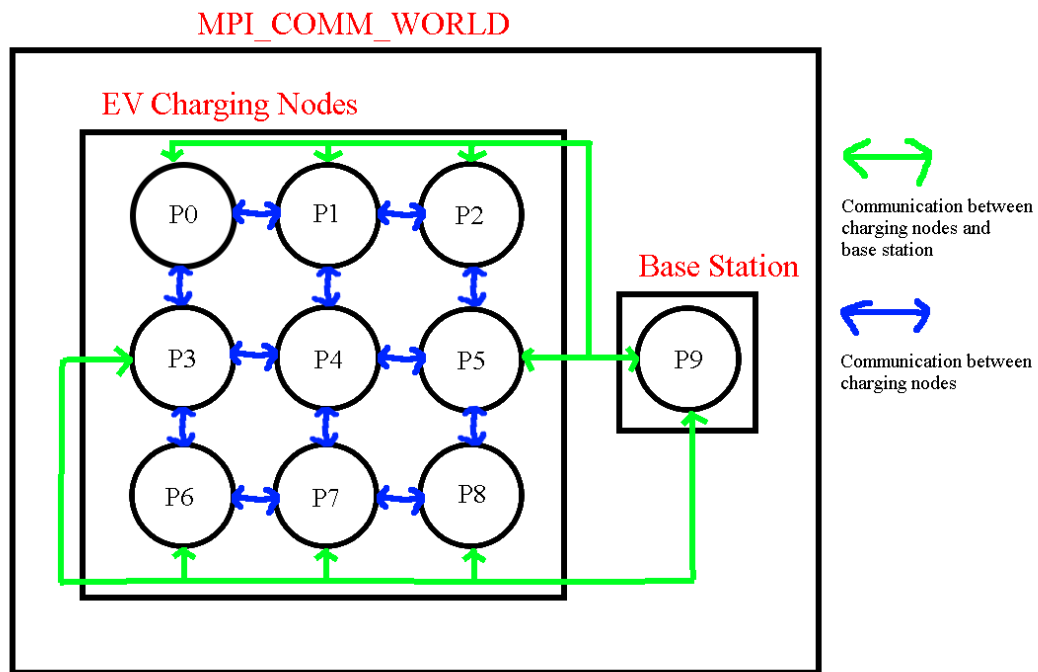


Figure 1.1: WSN architecture using MPI Virtual Topologies with 9 charging nodes and 1 base station.

As shown in Figure 1.1, the WSN architecture consists of 9 charging node processes arranged in a 3x3 Cartesian grid and a base station process. Using the master-slave architecture, the base station process writes logs and receives alerts from the charging node processes, which act as the slave processes performing the charging cycle and communication task.

1.2 – Main and General Functions

Pseudocode for main function

Function main(argc, argv):

 Initialize MPI with requested threading level MPI_THREAD_MULTIPLE

 Calculate masterRank from worldSize.

 Create new communicator by splitting MPI_COMM_WORLD master and slave groups

 Call createCustomDatatype() to create custom MPI datatypes

 If current processor is the master processor:

 Call baseStationIo() with MPI_COMM_WORLD and newComm

 Else:

 Call chargingNodeIo() with MPI_COMM_WORLD and newComm

 Free custom datatypes created with MPI_Type_free()

 Call MPI_Finalize() to terminate the MPI program and clean up processes

 Return 0 to terminate the program

End Function

This code initialises MPI with multiple threads support. It assigns the last rank of MPI_COMM_WORLD to be the master (base station). In the example in Figure 1.1, the base station rank is 9. It creates a new communicator splitting MPI_COMM_WORLD into two groups: master and slaves. The master group has only the base station and the slave group has all the charging nodes. It calls createCustomDatatype() to create custom MPI datatypes. The if-else block decides what each MPI process does. The base station calls baseStationIo() and the charging nodes call chargingNodeIo(). This way the program simulates the different WSN behaviour independently. After the simulation ends, memory is cleaned up and freed and the program exits.

Pseudocode for createCustomDatatype function

Function createCustomDatatype():

Store the MPI_Datatype as global variables

Set the displacement values for each variable in NodeInfo using the `offsetof` function.

Create an MPI datatype structure for NodeInfo using `MPI_Type_create_struct`, specifying the size of the datatype, block lengths, displacements, data types, and store it in `MPI_NODE_DATATYPE`.

Commit the MPI datatype `MPI_NODE_DATATYPE` to make it available for use.

Set the displacement values for each variable in NodeAlert using the `offsetof` function.

Create an MPI datatype structure for NodeAlert using `MPI_Type_create_struct`, specifying the size of the datatype, block lengths, displacements, data types, and store it in `MPI_ALERT_DATATYPE`.

Commit the MPI datatype `MPI_ALERT_DATATYPE` to make it available for use.

End Function

This function creates two custom MPI datatypes for sending and receiving data between MPI processes, one for the NodeInfo structure and another for the NodeAlert structure. It sets the respective arguments required for each datatype, then calls `MPI_Type_create_struct` to create the custom datatype with said arguments. Then, this function calls `MPI_Type_commit` to commit the MPI datatype for use in subsequent MPI functions. The MPI_Datatype is saved as a global variable so that all MPI processes can access it.

1.3 – Charging Node

Each charging node process simulates an EV charging station with a k number of charging ports. In this scenario, k is 5. Each charging port is simulated by a POSIX thread, created, and joined by the `chargingStationIo` function.

Pseudocode for creating Cartesian Grid and creation of POSIX threads

Function `chargingNodeIo(worldComm, newComm)`:

- Initialise required variables

- Create array `threadID` of length `k`

- Get current rank of MPI process

- Set `dims` to 0 and `ndims` to 2 for 2D grid

- Call `MPI_Dims_create` with size of communicator, `ndims` and `dims`

- Set `periods` to 0 for no wrap-around of grid

- Set `reorder` to 0 so that ranks are preserved for each MPI node

- Call `MPI_Cart_create` to create the Cartesian Grid, storing the new communicator into `chargingNodeComm`

- Get neighbours of current node

- Assign instance variables of `NodeInfo` structure

- For `i = 0` to 3 inclusive:

 - Count number of neighbours and save their respective coordinates in `NodeInfo` structure

- Allocate memory for shared array of `portReports` with size `MAX_REPORTS`

- Call `pthread_mutex_init()` to initialise mutex with `NULL`

- for `i = 0` to `k - 1` inclusive:

 - Multiply `i` with current rank and assign it to variable `seed`

 - Create a new thread with `threadID[i]` and run the `runChargingPort` method with the `seed`

This code creates a 2D grid of charging node processes, each an MPI process with dimensions from `MPI_Dims_create`. It sets the period variable to 0 to indicate that there is no wrap-around for the grid. It also sets the reorder variable to 0 to ensure that the ranks of MPI processes are preserved. Then, this block of code creates a Cartesian grid communicator called `chargingNodeComm` using arguments created earlier. It also creates a `NodeInfo` structure with fields from its rank, coordinates, neighbours, and neighbour ranks and coordinates. Then it allocates memory for the shared `portReports` array. Finally, this block of code uses a loop to create and start `k` threads, each running the `runChargingPort` function with a different seed value. The seed value is generated using the current rank and the loop counter, which are used to generate port availability.

Pseudocode for runChargingPort thread function

Function *runChargingPort(*pArg):

while true:

 Lock mutex

 If termination signal received, break out of the while loop

 Else continue

 Unlock mutex

 Calculate the new seed, taken from pArg and multiplied with the current time

 Use the seed in rand_r to generate a random number (0 or 1) to represent availability

 Lock the mutex

 Increment the nPortsRan variable

 If availability is one:

 Increment the number of available charging ports for this cycle (nAvailability)

 If nPortsRan is divisible by 0:

 Enqueue the nAvailability into the shared portReports array

 Unlock the mutex

 Sleep for 10 seconds

End Function

This function loops until it gets the termination signal. It locks the mutex to check the signal variable. If true, it exits the loop and function. Otherwise, it generates the port availability randomly using a seed from the arguments and current time. This gives each thread a different and dynamic seed for each iteration. The availability is 0 or 1, with the former meaning occupied and the latter meaning available. It locks the mutex again to access the shared data, minimising race conditions. It increments nPortsRan, a global variable for how many ports updated their availability in this cycle. A cycle is 10 seconds long. If the port is free, it increments nAvailability, a global variable for how many ports are free in this cycle. If all ports are updated in this cycle, it enqueues nAvailability into portReports, an array for port availability history. It unlocks the mutex and sleeps for 10 seconds before repeating.

chargingNodeIo Function Loop Logic

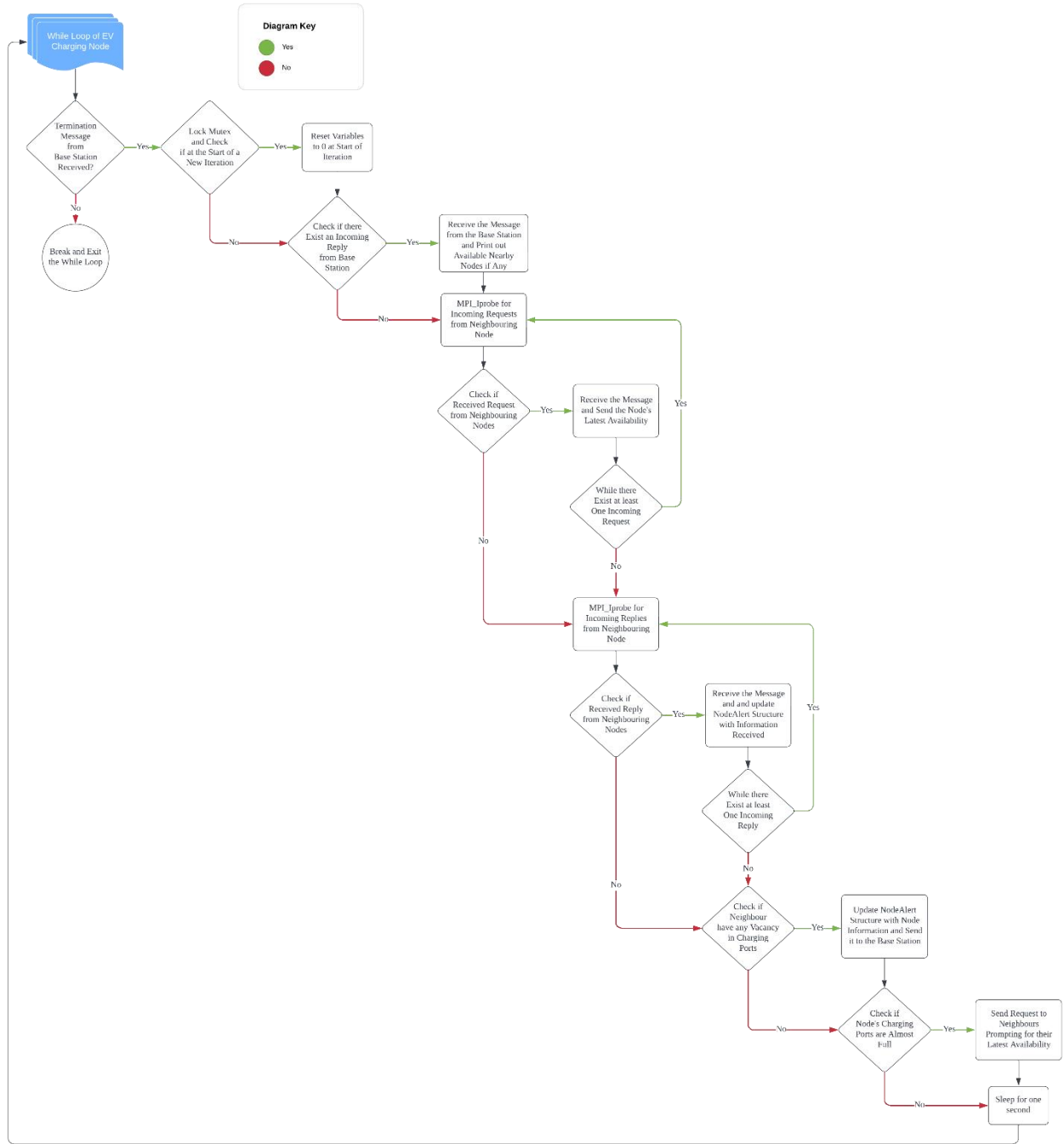


Figure 2.1: The loop logic in chargingNodeIo function simulating the charging node at each iteration

Link to LucidChart for bigger view: [here](#)

Figure 2.1 shows a flowchart of the charging node simulation. It starts by checking if it has received a termination signal from the base station; if found, it exits. If not, it proceeds. It then checks if it is the start of a new cycle (every 10 seconds). If yes, it resets some variables to 0 and it uses a mutex lock to update the latest availability of the node, preventing data race conditions. The node then checks for incoming base station messages using MPI_Iprobe. The non-blocking probe is used as it allows the node to continue its tasks without waiting for a message. If a message arrives, it prints nearby available nodes. Otherwise, it enters a do-while loop to check for requests from neighbouring nodes. If MPI_Iprobe returns true, it uses MPI_Recv to receive the request and replies

with its latest availability to the sender. Blocking receive is utilised because the charging node should wait for and respond to requests from its neighbouring nodes. After processing these requests, it repeats this until there are no more requests. It then enters another do-while loop to check for replies from neighbouring nodes. Similarly, it receives the reply and updates the NodeAlert structure with the information. It repeats this until there are no more replies. If the replies indicate that there are no free ports in the neighbouring nodes, it sends an alert to the base station using MPI_Isend. Like the probe, non-blocking sending is used to improve the efficiency of the code and reduce wait time. Lastly, the node checks if it's almost full (2/5 ports remaining). In this case, it requests availability data from neighbours, waits for a second, and repeats the loop.

1.4 – Base Station

The base station acts as the central hub for the EV charging navigation system, collecting data from all WSN charging nodes and recording key performance metrics. These metrics include:

- Current simulation iteration/cycle.
- Timestamp of alert dispatch.
- Timestamp of data logging.
- Reporting node's rank, coordinates, and availability.
- Neighbouring nodes' rank, coordinates, and availability.
- Nearby adjacent nodes' rank and coordinates.
- Suggested nearby available nodes.
- Communication time between the reporting node and the base station.
- The total number of messages between the base station and the reporting node.
- The total number of messages between the reporting node and its neighbours.

A single POSIX thread manages message sending and receiving, while the main function handles data logging, ensuring efficient data collection and logging at the base station. The base station will run a maximum of 10 iterations, where each iteration is 10 seconds long. Each iteration will only have one charging cycle.

Pseudocode for *receiveAlerts thread function

Function * receiveAlerts (*pArg):

Allocate memory for two dynamic arrays to store the last report time and time elapsed

for i = 0 to number of charging nodes:

Update each slave's time to the current clock time initially

While true until termination signal is sent:

do:

Probe for incoming alerts from charging nodes

If there exist at least one incoming alert:

Blocking Receive to get alert from reporting node

Calculate the communication time between the two processors

Set the corresponding clock time for the reporting node in the array

for i = 0 to number of charging nodes:

Calculate elapsed time and update the elapsed time array for each slave node

Find nearby adjacent available nodes by calling processAlert()

Non-blocking send the nearby available nodes back to the reporting node

while until probe result is false (no more incoming alerts)

Enqueue the alert with other vital informations into the baseLogs queue

Clean and free memory

End Function

The receiveAlerts function allocates memory for arrays to store the last report time and elapsed time, initialises the nodes' clock times and enters a do-while loop that probes for incoming alerts and processes them upon receipt. For each received alert, it calculates communication time and records the reporting node's time. It then updates the elapsed time for each node, identifies nearby available nodes using the processAlert function, and sends this information back to the reporting node non-blocking. The process repeats until there are no more incoming alerts. The corresponding data is enqueued into the baseLogs array. The entire outer while loop will terminate upon the termination signal being sent.

Pseudocode for processAlerts function

Function processAlerts(int *availableNearbyNodes, NodeAlert *receivedNodeAlert):

for i = 0 to reporting node's number of neighbours:

for j = 0 to 3 inclusive:

Get neighbours of each neighbouring node.

If the node is not the reporting node itself, and is unique (has not been added yet):

Add it to the availableNearbyNodes array

Increment the index

return index - 1

End Function

The processAlerts function iterates through the neighbours of the reporting node, and for each neighbouring node, it further checks its adjacent nodes. For each adjacent node, it checks if it's not the reporting node itself and if it hasn't been added to the list of available nearby nodes yet (ensuring uniqueness). If these conditions are met, the adjacent node's rank is added to the availableNearbyNodes array, and the index is incremented. The function returns the index reduced by 1, providing the number of available nearby nodes based on the received node alert.

Pseudocode for baseStationIo function

Function baseStationIo (worldComm, newComm):

- Get the threadID for master

- Open file for writing

- Allocate sufficient memory for queue to store information of each log (baseLogs)

- Initialise mutex lock for master

- Create thread with threadID which will run the receiveAlerts thread function

- While iteration less than maximum iterations

 - If current iteration surpassed maximum iterations:

 - Exit loop

 - Increment iteration at the start of each iteration

- While there exist unlogged alerts in queue

 - Log the performance metrics

 - Sleep for 5 seconds to simulate one cycle/iteration

- for i = 0 up to size of the world communicator:

 - Non-blocking send to every charging node the termination signal

- Rejoin threads

- Free memory and destroy mutex lock

- Close the file

End Function

The baseStationIo function begins by initialising necessary resources, including thread management, file handling, and memory allocation for the queue to hold information for the logs. A thread is created to run the receiveAlerts function that handles incoming alerts and replies to them. It runs a loop simulating base station operations for a set number of maximum iterations. Inside this loop, it exits if the current iteration exceeds the maximum limit, else it increments the iteration counter at the beginning of each iteration. Within each iteration, it processes unlogged alerts in the queue, logs performance metrics, and then sleeps to simulate a cycle/iteration. Once the maximum iterations are reached, it sends a termination signal to all charging nodes, waits for threads to complete, and cleans up resources, ensuring proper shutdown of the base station.

2 - Results Tabulation

2.1 – Local Machine

The simulation program was tested on a local machine with the following specifications:

- Number of cores: 8
- Number of logical processors: 12
- System memory: 16 GB RAM
- CPU base speed: 2.0 GHz

The simulation program was run with different grid sizes, ranging from 2x2 to 5x5. Each grid size was tested with 5 charging ports per node and 10 iterations before termination by the base station. The simulation program was run 10 times for each grid size and the average results were recorded.

Performance Metrics and Summary of Events between Reporting Node and Base Station

Grid Size	Total Number of Alerts	Communication Time (seconds)		Number of Messages Exchanged
		Total	Average	Total
2 x 2	9	13.022	1.447	18
3 x 3	13	30.012	2.309	26
4 x 4	13	25.559	1.996	26
5 x 5	25	49.395	1.976	50

Performance Metrics and Summary of Events between Reporting Node and Neighbouring Nodes

Grid Size	Total Number of Alerts	Communication Time (seconds)		Number of Messages Exchanged
		Total	Average	Total
2 x 2	9	34.007	3.779	72
3 x 3	13	46.011	3.539	126
4 x 4	13	37.273	2.867	140
5 x 5	25	84.093	3.364	284

```
-----
Iteration :                5
Logged time :              Wed 2023-10-18 03:49:39
Alert reported time :      Wed 2023-10-18 03:49:30
Number of adjacent node(s): 2
Availability to be considered full: 2

Reporting Node   Coord      Port Value   Available Port
0               (0,0)         5             0

Adjacent Nodes   Coord      Port Value   Available Port
1               (0,1)         5             2
2               (1,0)         5             2

Nearby Nodes     Coord
3               (1,1)

Available station nearby (no report received in last 5 seconds): 3
Communication Time (seconds) between reporting node and base station : 1.0011
Total Messages exchanged between reporting node and base station: 2
Total Messages exchanged between reporting node and its neighbours: 8
-----
```

Figure 3.1: Base log on a single alert for a 2 x 2 grid

```

-----
Iteration :                2
Logged time :              Wed 2023-10-18 03:59:18
Alert reported time :      Wed 2023-10-18 03:59:09
Number of adjacent node(s): 3
Availability to be considered full: 2

Reporting Node    Coord      Port Value    Available Port
1                (0,1)         5             0

Adjacent Nodes    Coord      Port Value    Available Port
2                (0,2)         5             0
4                (1,1)         5             2
0                (0,0)         5             0

Nearby Nodes      Coord
5                (1,2)
7                (2,1)
3                (1,0)

Available station nearby (no report received in last 5 seconds): 5, 7, 3
Communication Time (seconds) between reporting node and base station : 1.9925
Total Messages exchanged between reporting node and base station: 2
Total Messages exchanged between reporting node and its neighbours: 12
-----

```

Figure 3.2: Base log on a single alert for a 3 x 3 grid

```

-----
Iteration :                6
Logged time :              Wed 2023-10-18 04:15:57
Alert reported time :      Wed 2023-10-18 04:15:48
Number of adjacent node(s): 3
Availability to be considered full: 2

Reporting Node    Coord      Port Value    Available Port
11               (2,3)         5             0

Adjacent Nodes    Coord      Port Value    Available Port
10               (2,2)         5             1
15               (3,3)         5             0
7                (1,3)         5             2

Nearby Nodes      Coord
6                (1,2)
14               (3,2)
9                (2,1)
3                (0,3)

Available station nearby (no report received in last 5 seconds): 6, 14, 9, 3
Communication Time (seconds) between reporting node and base station : 1.9640
Total Messages exchanged between reporting node and base station: 2
Total Messages exchanged between reporting node and its neighbours: 12
-----

```

Figure 3.3: Base log on a single alert for a 4 x 4 grid

```

-----
Iteration :                7
Logged time :              Wed 2023-10-18 04:27:33
Alert reported time :      Wed 2023-10-18 04:27:24
Number of adjacent node(s): 4
Availability to be considered full: 2

Reporting Node    Coord      Port Value    Available Port
7                (1,2)         5             0

Adjacent Nodes    Coord      Port Value    Available Port
6                (1,1)         5             0
8                (1,3)         5             0
12               (2,2)         5             0
2                (0,2)         5             0

Nearby Nodes      Coord
1                (0,1)
11               (2,1)
5                (1,0)
3                (0,3)
13               (2,3)
9                (1,4)
17               (3,2)

Available station nearby (no report received in last 5 seconds): 11, 13, 9, 17
Communication Time (seconds) between reporting node and base station : 1.9736
Total Messages exchanged between reporting node and base station: 2
Total Messages exchanged between reporting node and its neighbours: 16
-----

```

Figure 3.4: Base log on a single alert for a 5 x 5 grid

2.2 – CAAS Platform

The simulation program was tested on a Monash CAAS platform with the following specifications:

- 146.96 Mbps Download Speed.
- 66.13 Mbps Upload Speed.

The simulation program was run with different grid sizes, ranging from 2x2 to 5x5. Each grid size was tested with 5 charging ports per node and 10 iterations before termination by the base station. The simulation program was run 10 times for each grid size and the average results were recorded.

Performance Metrics and Summary of Events between Reporting Node and Base Station

		Communication Time (seconds)		Number of Messages Exchanged
Grid Size	Total Number of Alerts	Total	Average	Total
2 x 2	10	17.021	1.702	20
3 x 3	13	21.904	1.991	22
4 x 4	17	49.012	2.883	34
5 x 5	17	32.033	1.884	34

Performance Metrics and Summary of Events between Reporting Node and Neighbouring Nodes

		Communication Time (seconds)		Number of Messages Exchanged
Grid Size	Total Number of Alerts	Total	Average	Total
2 x 2	10	25.7	2.57	80
3 x 3	13	35.012	3.183	100
4 x 4	17	70.01	4.119	170
5 x 5	17	72.014	4.178	184

3 – Analysis and Discussion

3.1 – Analysis of Results

The results of the simulation program can be analysed and discussed based on the following hypothesis:

The communication time measured by the base station increases linearly for an increasing number of reported alerts by charging nodes.

This hypothesis can be observed within the results of the Local Machine and CAAS where both Average and Total Communication Time between reporting node and base station generally increases as grid size increases. This is because the increase in grid size results in a higher number of charging nodes, leading to a higher chance of more reported alerts. Consequently, the base station must process a larger volume of incoming messages, increasing the communication time as a result.

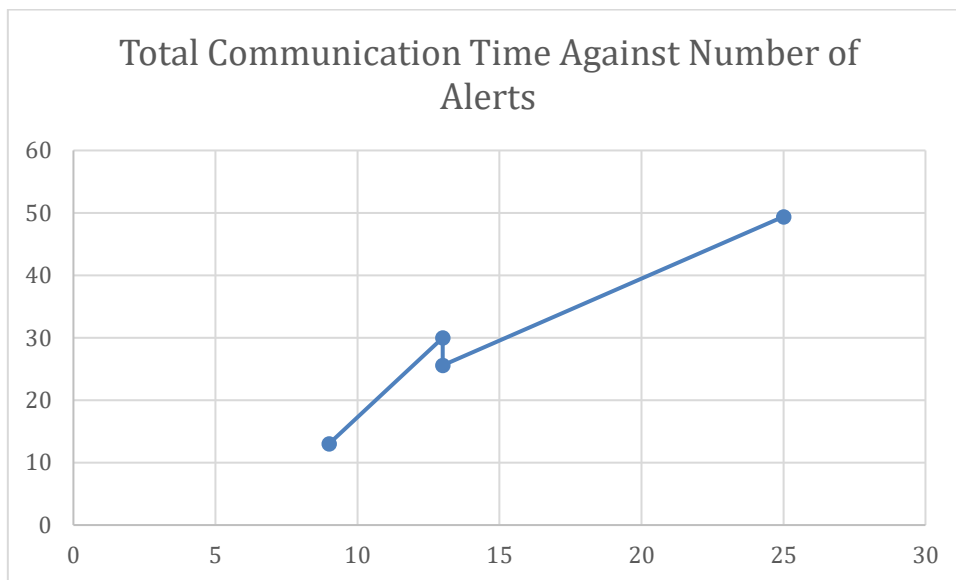


Figure 5.1: Total communication time versus number of alerts for all grid sizes

Figure 5.1 shows that there is a positive linear relationship between the communication time and the number of alerts, as indicated by the upward-sloping trend lines. This means that as more charging nodes report their full or almost full status to the base station, the communication time increases proportionally. This is expected, as more messages need to be sent and received by the base station, which adds to the communication overhead and latency.

However, some results deviate from this hypothesis, such as the communication time for grid size 4x4 on the Local Machine, which decreased from 2.3 to 1.9 seconds. This can also be shown in Figure 5.1 with the sudden drop in between 20 – 30 seconds. This can be attributed to the randomness of the port availability in each iteration. Due to this, there is a possibility that the base station could receive few or no reports for many iterations, reducing the overall communication time.

Furthermore, based on the results tabulated above, it can be observed that:

The total communication time between the reporting node and its neighbours increases as the grid size increases.

This behaviour can be attributed to the growing number of neighbours in larger grids, necessitating more communication and message exchanges. For example, in a 2x2 grid, all nodes will at most have 2 neighbours, with a total communication time of 34 seconds on the local machine. However, in a 3x3 grid as shown in Figure 1.1, node 4 has 4 neighbours while the rest have 2 or 3, resulting in an increase in communication time to 46 seconds. In essence, as the grid expands, the complexity of communication within the network increases, resulting in longer communication times between nodes, as supported by the results presented in the tabulated data.

As for the CAAS Platform, the tabulated results support both hypotheses. This can be shown in the increase in total communication time from 17 seconds to 32 seconds for 2x2 to 5x5 grids respectively. Similarly, the tabulated results for the reporting node and its neighbours exhibit the same pattern. However, it can be noted that the overall communication time is longer in comparison to the results obtained from the local machine. This could be caused by network latency, as shown in my upload and download speeds when connected to the VPN. This may result in delays in message exchange between nodes and the base station, contributing to an increase in communication time. Another possible cause is network congestion. The CAAS platform shares resources with Monash staff and students, creating network congestion which might lead to variable communication times.

3.2 – Discussion of Limitations

One limitation mentioned earlier is the inconsistencies in results which arise from the random generation of port availability. Future work should focus on enhancing the consistency of results. One approach is to refine the randomness mechanisms to ensure more consistent reporting. Additionally, conducting more iterations to gather statistically significant data can help mitigate the impact of randomness.

Another limitation is that simulation results are based on a model and lack real-world validation. For example, the port availability may be influenced by factors such as time of day, day of week, season, weather, traffic, and user preferences. Therefore, the simulation program may not capture the true dynamics and variability of the electric vehicle charging demand and supply. Future work can include real-world data collection and validation of the simulation results. Comparing the simulation outcomes with real-world scenarios can enhance the accuracy and applicability of the model.

4 – References

GeeksforGeeks. (2023). *MPI - Distributed Computing made easy*. <https://www.geeksforgeeks.org/mpi-distributed-computing-made-easy/>

Wikipedia. (2023). *Message Passing Interface*. https://en.wikipedia.org/wiki/Message_Passing_Interface

Wikipedia. (2023). *Pthreads*. <https://en.wikipedia.org/wiki/Pthreads>