FIT 3143 Parallel Computing Assignment 2

Name: Samuel Tai Meng Yao
Student ID: 32025068
Student email: stai0007@student.monash.edu

WORD COUNT:
2140 words

TABLE OF CONTENTS

The URL for the Lucid Chart is attached in the appendix of the diagram reference purpose

1. Methodology

1.1 Overall Architecture

The simulation architecture for the Wireless Sensor Network (WSN) is built upon a foundation of MPI Processes and POSIX Threads. This simulation comprises two primary components: the base station and the charging nodes, where each node will have n charging ports. In this context, the charging nodes are represented as a Cartesian grid, with each node restricted to communicating exclusively with its immediate neighbours, allowing for a maximum of four connections (as depicted in *Figure 1*).

To enhance the simulation of the charging nodes, I chose to implement MPI Virtual Topology. This decision was influenced after reading one of the articles from Spagnuolocarmine (n.d.) which highlights a common issue in parallel applications—where a linear ranking of processes fails to accurately represent the logical communication between them. The adoption of a virtual topology, specifically a Cartesian one in our case, allows the system to adapt the application for more efficient communication as the communication patterns are known to be highly localised. It leverages the hardware more effectively by mapping the processes onto the hardware, as Traff (2002) has pointed out. Such implementation might lead to a more efficient communication between the charging nodes in the WSN which is one of the main aims of this assignment.

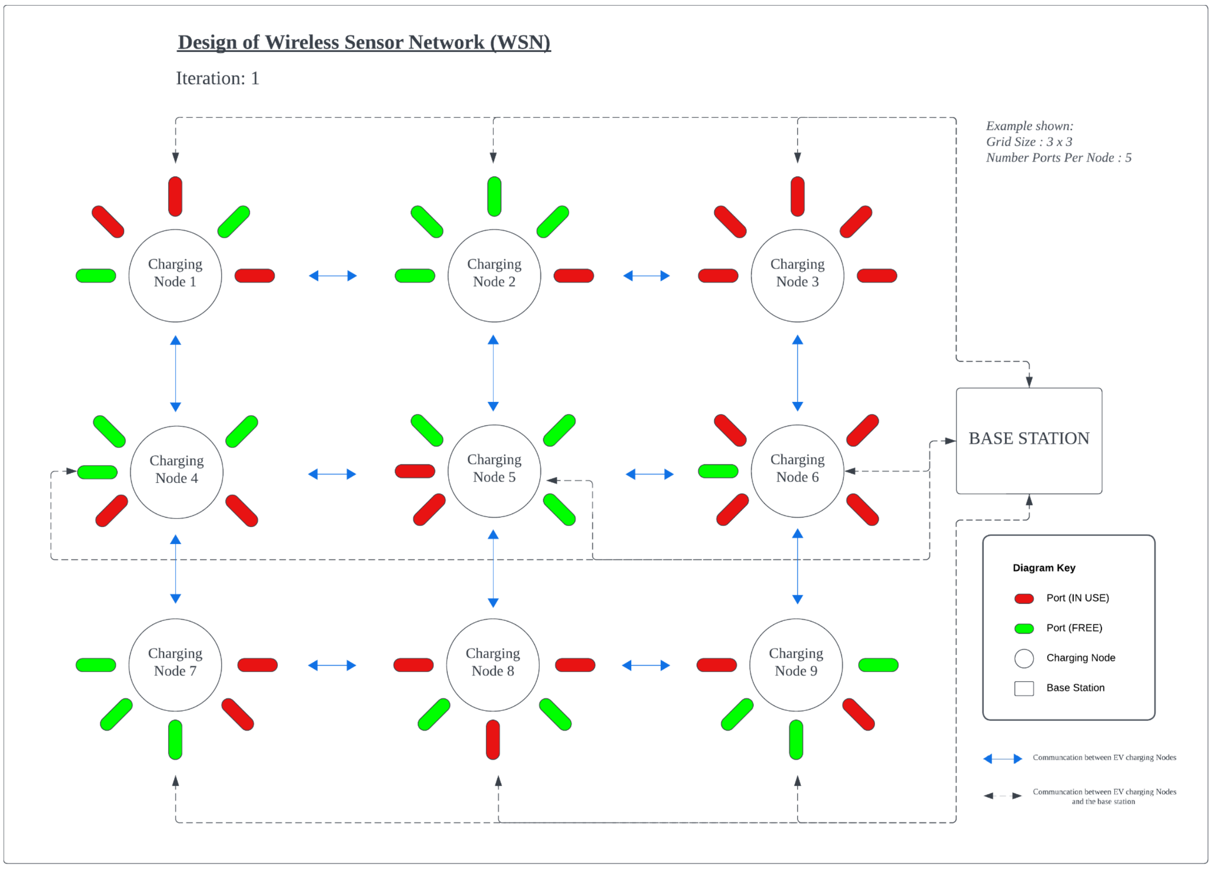


Figure 1: *Illustration of the Wireless Sensor Network (WSN) in Non-Technical Format*

In terms of the implementation (as depicted in Figure 2), each charging node and the base station are simulated using MPI processes. Additionally, each charging node is equipped with n ports, which are simulated using threads. Notably, in our implementation, an extra thread is spawned by the base station to handle incoming alerts from the charging nodes.

This combination of MPI processes and threads, known as a hybrid approach in the base station, has the potential to enhance scalability by mitigating communication costs and reducing the computational burden on the base station, as indicated by ENCCS (please provide the full name and publication date if available). This implementation holds the promise of improving communication time (comprising both communication and resolution time) between alerting nodes and the base station. It allows the base station to not only identify solution nodes but also concurrently receive alerts from other nodes.

Each charging node has a shared array of fixed size which is used to record the timestamp and the port availability for the node throughout the simulation which is updated parallelly by the ports (threads). Further details on the design and implementation of communication between the charging nodes and the base station will be discussed in the subsequent subsection (Charging Nodes & Base Station)."

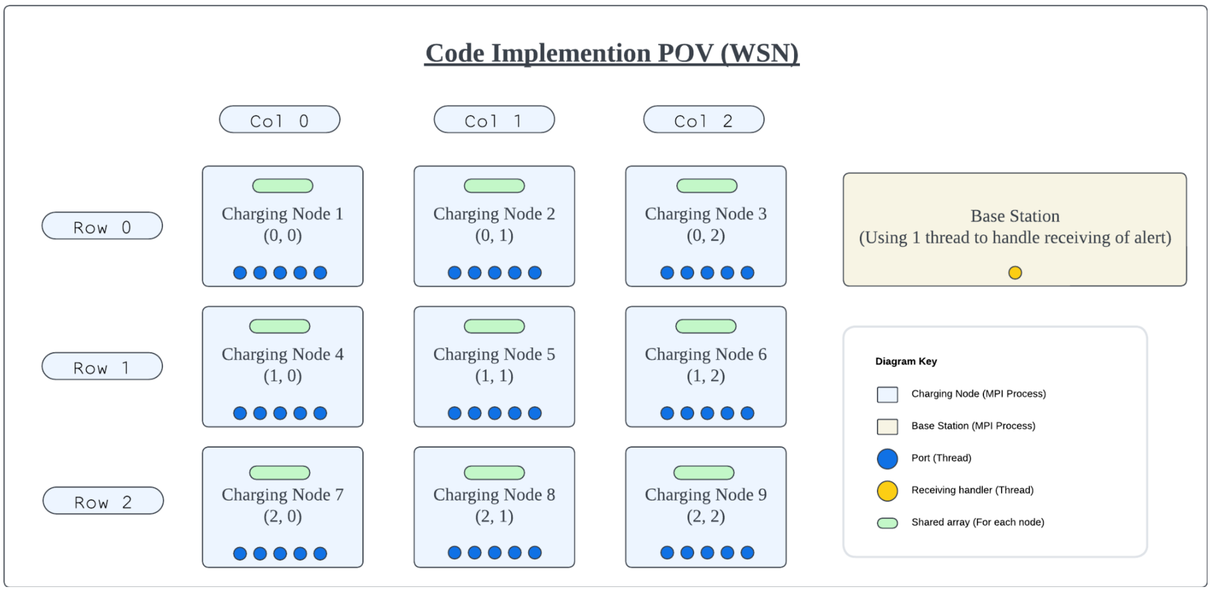


Figure 2: *Illustration of the Wireless Sensor Network (WSN) in Technical Format*

The actual cartesian topology in MPI which I have adopted in the simulation architecture looks as in Figure 2 where each charging node will be in part of the cartesian grid with its own coordinates, ports and shared array where the base station is a standalone MPI Process which communicates with the nodes.

1.2 Self-defined Structures

This section will describe the structures (struct) that have been developed for the implementation of the Wireless Sensor Network (WSN) simulation to describe some design decisions and make the design explanation in the following section easier to understand. The created structures serve primarily **2 purposes**:

1) Sharing details with the main MPI process (whether it's a node or the base station) from which the thread originates.
2) Combining details to facilitate communication through MPI Send and Recv operations.

These structures play a crucial role in the efficient exchange of information and coordination between various components of the simulation.

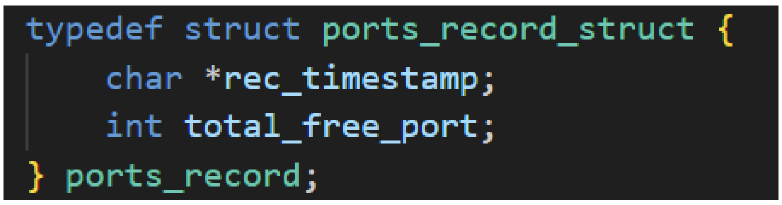
1.2.1 Shared Array Data (ports_record)

In order to record the timestamp and the availability of the ports per iteration as the details in the shared array, I decided to combine them into a new struct.

This struct consists of 2 members:

- timestamp record (char* type)
- total available port (int type)

This struct will only be used as the element of the shared array in each of the nodes, where the shared array is of type `ports_record*` where each element in the array will be pointed to the actual `ports_record` struct.



```c
typedef struct ports_record_struct {
    char *rec_timestamp;
    int total_free_port;
} ports_record;
```

***Figure 3:** Code snippet of `ports_record` struct*

1.2.2 Port Data (node_thread_data)

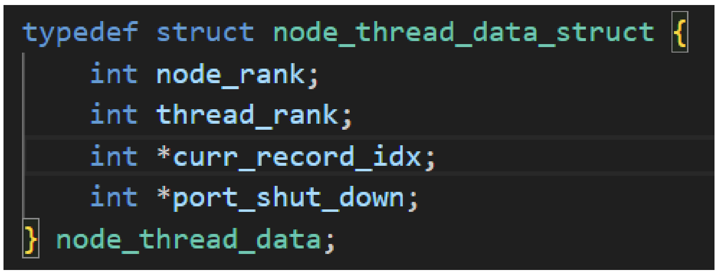
The struct is created to handle the content sharing between the port and its respective charging node instead of putting all shared variables into the global scope which provides better encapsulation and modularity in the code.

The struct contains 4 members:

- Node rank (int)
- Thread rank (int)
- Current record index (int * - shared purpose)

- Port shut down (int * - shared purpose)

This struct will be used as an argument that passed to the thread function for the charging ports. The purpose is to allow the port to keep track of when to shut down and which index of the shared array should be updated in that iteration.



```c
typedef struct node_thread_data_struct {
    int node_rank;
    int thread_rank;
    int *curr_record_idx;
    int *port_shut_down;
} node_thread_data;
```

Figure 4: *Code snippet of* `node_thread_data` *struct*

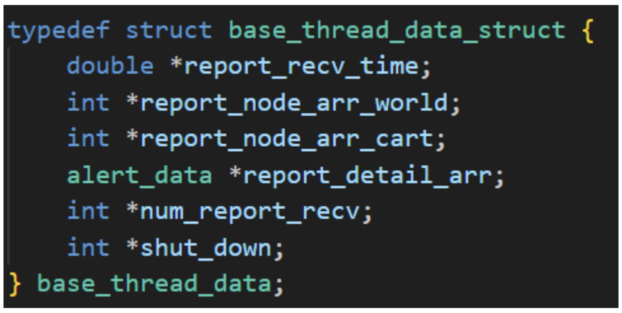
1.2.3 Base Thread Data (base_thread_data)

Serving a similar purpose as the above struct, the base_thread_data struct is also meant for sharing variables' value between the base thread and the base station to synchronise the detail related to the alert/report received.

The struct contains 6 members:

- Report received time (double *)
- Report node world rank arr (int *)
- Report node cart rank arr (int *)
- Report detail arr (alert_data *)
- Number of reports received (int *)
- Shut down (int *)

All of the members are created to do the job of shared variable in order to synchronise the process of alert handling and alert receiving without the need of locking and signalling between the thread and base station.



```c
typedef struct base_thread_data_struct {
    double *report_recv_time;
    int *report_node_arr_world;
    int *report_node_arr_cart;
    alert_data *report_detail_arr;
    int *num_report_recv;
    int *shut_down;
} base_thread_data;
```

Figure 5: *Code snippet of* `base_thread_data` *struct*

1.2.4 Alert Data (alert_data)

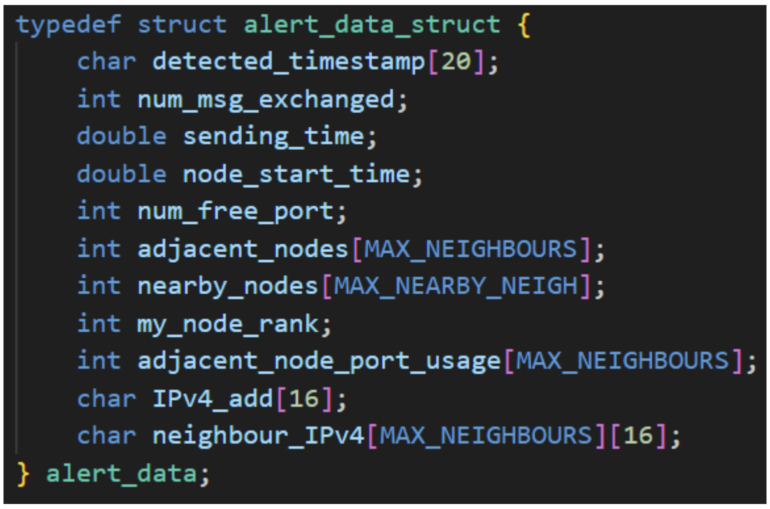
This structure serves a distinct purpose compared to the three structures mentioned earlier. Its primary role is to consolidate all the necessary information for an alert into a single struct, which is then sent to the base station for alert purposes. There are several reasons behind choosing this approach over a pack/unpack approach.

Firstly, the use of a custom datatype for MPI send and receive operations consumes less storage space, as noted by Dongarra (1995). Additionally, he highlighted that this approach also results in faster computational times. By defining a new datatype, the system can efficiently invoke a single function to send or receive the datatype, eliminating the need to call multiple functions to pack data individually. This reduction in function calls minimizes overhead, which is especially beneficial when sending and receiving large volumes of data multiple times within each iteration.

The struct contains 11 members:

- Detected timestamp (char [])
- Number of messages exchanged (int)
- Sending time (double)
- Current iteration start time (double)
- Number of free ports (int)
- Adjacent nodes (int [])
- Nearby nodes (int [])
- Cart rank (int)
- Adjacent node port usage (int [])
- IPv4 address (char [])
- Neighbour IPv4 address (char [][])

These are the details of the alert that I will be sent to the base station every time which serves the logging purpose and providing sufficient information for the base station to check whether any of the nearby nodes of the reporting node is available and provide resolution for it.

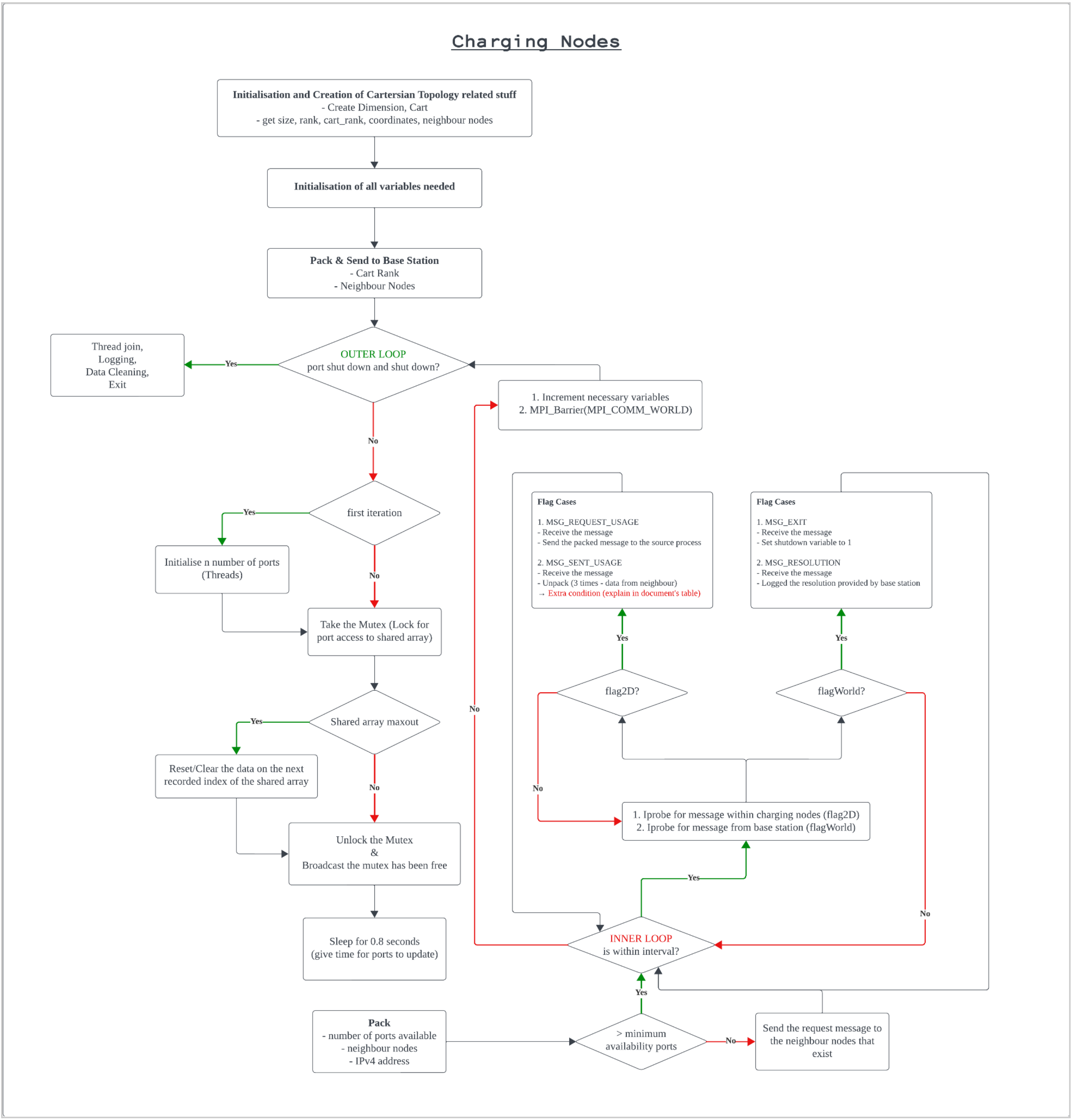


```c
typedef struct alert_data_struct {
    char detected_timestamp[20];
    int num_msg_exchanged;
    double sending_time;
    double node_start_time;
    int num_free_port;
    int adjacent_nodes[MAX_NEIGHBOURS];
    int nearby_nodes[MAX_NEARBY_NEIGH];
    int my_node_rank;
    int adjacent_node_port_usage[MAX_NEIGHBOURS];
    char IPv4_add[16];
    char neighbour_IPv4[MAX_NEIGHBOURS][16];
} alert_data;
```

Figure 6: *Code snippet of* `alert_data` *struct*

1.3 Charging Node

The design and implementation details will be elaborated using the flowchart provided below (the explanation below represents a charging node which is the same for others) (*Figure 7*):



Charging Nodes
Initialisation and Creation of Cartersian Topology related stuff
- Create Dimension, Cart
- get size, rank, cart_rank, coordinates, neighbour nodes
Initialisation of all variables needed
Pack & Send to Base Station
- Cart Rank
- Neighbour Nodes
OUTER LOOP
port shut down and shut down?
Yes
Thread join,
Logging,
Data Cleaning,
Exit
1. Increment necessary variables
2. MPI_Barrier(MPI_COMM_WORLD)
No
first iteration
Yes
Initialise n number of ports
(Threads)
No
Take the Mutex (Lock for port access to shared array)
Shared array maxout
Yes
Reset/Clear the data on the next recorded index of the shared array
No
Unlock the Mutex
&
Broadcast the mutex has been free
Sleep for 0.8 seconds
(give time for ports to update)
Pack
- number of ports available
- neighbour nodes
- IPv4 address
> minimum availability ports
No
Send the request message to the neighbour nodes that exist
Yes
INNER LOOP
is within interval?
No
Yes
1. Iprobe for message within charging nodes (flag2D)
2. Iprobe for message from base station (flagWorld)
flag2D?
No
Yes
Flag Cases
1. MSG_REQUEST_USAGE
- Receive the message
- Send the packed message to the source process
2. MSG_SENT_USAGE
- Receive the message
- Unpack (3 times - data from neighbour)
→ Extra condition (explain in document's table)
flagWorld?
No
Yes
Flag Cases
1. MSG_EXIT
- Receive the message
- Set shutdown variable to 1
2. MSG_RESOLUTION
- Receive the message
- Logged the resolution provided by base station

Figure 7: *Flowchart of Charging Nodes Operation*

The charging node commences its operation with the initialization and creation of all essential variables and functions, including the establishment of a Cartesian topology and the calculation of coordinates, among other tasks. Subsequently, the charging node transmits its Cartesian rank and coordinates to the base station. This design decision is made to enhance communication efficiency and computational speed between the charging node and the base station. It allows the base station to reference records as needed, thereby eliminating redundant transmissions, especially when they are intended for logging purposes.

On the first iteration only, all ports (threads) are initialized with a self-defined struct variable passed as an argument to the thread function. The struct contains dynamically allocated memory variables serving as shared variables (as discussed in the previous section). This design choice is consistently applied throughout most of the algorithms.

Due to the presence of a fixed-size shared array, I implemented a FIFO-like approach, where the oldest data in the shared array is reset if the array is full at the start of each iteration. This shared array stores self-defined struct variables. Mutexes and conditional variables are used to prevent race conditions and lock the port from updating port availability until the next iteration starts, which is signalled by the charging node. This implementation ensures synchronization between the node and port, mirroring real-world node and port behaviour. After the check, the node broadcasts to the ports to release the mutex lock.

Subsequently, the node sleeps for 0.8 seconds to allow the ports to update their availability before proceeding.

Next, I designed the node to first pack the necessary data for sending to neighbouring nodes to save time on repeated repacking when there is a request. Pack and Unpack implementations are used to enhance communication efficiency, allowing the node to send data only once to the requested node.

If the node's port availability falls below the defined minimum, it sends a request to all its valid neighbouring nodes using Isend. Asynchronous sending is employed as the node does not need to wait for the send operation to complete. Using blocking send in this scenario, given my algorithm's design, would lead to a deadlock.

Following this, the charging node enters the inner loop, which continues until the defined interval is reached. Within this loop, the node manages the sending and receiving of data from both neighbouring nodes and the base station. It is designed to periodically listen and check for the arrival of messages that meet the node's requirements using Iprobe. The node only begins to receive when there is a message to be received, confirmed by testing with Iprobe to check the flag status. This implementation ensures efficient utilization of resources and time without blocking the code. Two Iprobe checks are employed, one for testing the arrival of messages from neighbouring nodes and another for the base station.

After determining the flag and the message's tag, the corresponding blocking receive operation is executed. Blocking receive is used as the node is certain that a message is to be received, and subsequent operations depend on the received data. Therefore, it is designed to block and receive the message. Actions are then taken based on the message tag. There are a total of four scenarios, but I will describe only two of them—MSG_EXIT and MSG_SENT_USAGE. The remaining two can be understood from the flowchart above.

Message Tag	**Operation**
MSG_EXIT	The charging node will receive this message during the second-to-last iteration. This design decision is made to prevent the scenario where the base station has closed, but the node or port is still updating values. At the end of the last iteration, once the ports have updated their availability, they will be shut down. In the subsequent iteration, when the node detects that the port is shut down, it will proceed to shut down as well.
MSG_SENT_USAGE	The charging node will receive and store data from its neighbouring nodes. Once it confirms that it has received all the data from its neighbours, it will check for the presence of any available neighbouring nodes. If there are available nodes, it will log the data and continue the loop. However, if no free neighbouring nodes are found, the charging node will construct and send the required details within an 'alert_data' structure to the base station using 'Isend.' This approach allows the node to proceed with its receiving operation without blocking it, thereby enhancing communication efficiency.

Finally, before proceeding into the next outer iteration, the node will ensure all nodes and the base station hit the MPI_Barrier to ensure all of them could have a more synchronised start time for each new iteration.

1.4 Base Station

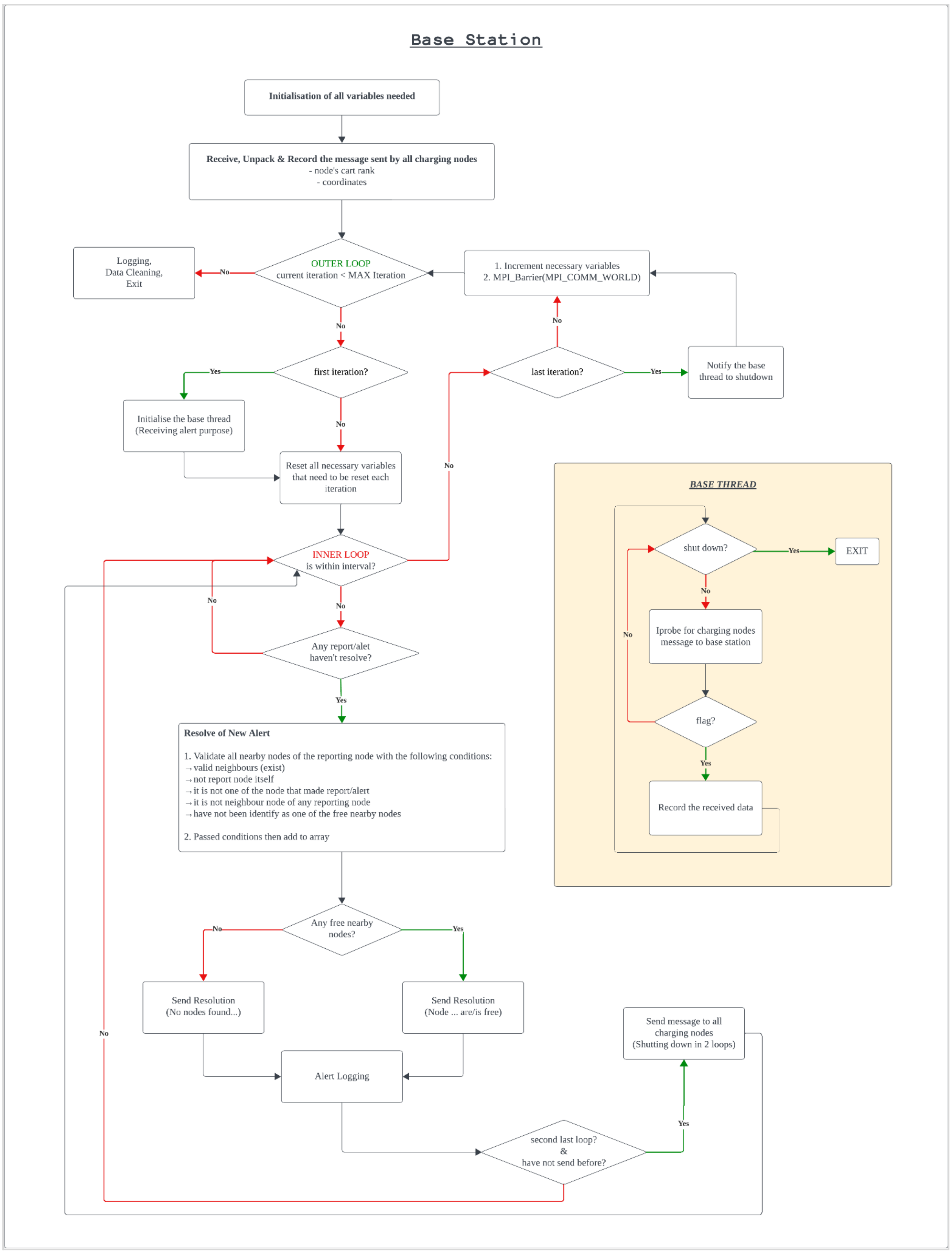

Base Station
Initialisation of all variables needed
Receive, Unpack & Record the message sent by all charging nodes
- node's cart rank
- coordinates
OUTER LOOP
current iteration < MAX Iteration
Logging,
Data Cleaning,
Exit
No
1. Increment necessary variables
2. MPI_Barrier(MPI_COMM_WORLD)
No
first iteration?
Yes
Initialise the base thread
(Receiving alert purpose)
No
Reset all necessary variables that need to be reset each iteration
last iteration?
Yes
Notify the base thread to shutdown
No
INNER LOOP
is within interval?
No
Any report/alet haven't resolve?
Yes
Resolve of New Alert
1. Validate all nearby nodes of the reporting node with the following conditions:
→ valid neighbours (exist)
→ not report node itself
→ it is not one of the node that made report/alert
→ it is not neighbour node of any reporting node
→ have not been identify as one of the free nearby nodes
2. Passed conditions then add to array
Any free nearby nodes?
No
Yes
Send Resolution
(No nodes found...)
Send Resolution
(Node ... are/is free)
Alert Logging
second last loop?
&
have not send before?
Yes
Send message to all charging nodes
(Shutting down in 2 loops)
BASE THREAD
shut down?
Yes
EXIT
No
Iprobe for charging nodes message to base station
flag?
Yes
Record the received data

Figure 8: *Flowchart of Base Station Operation*

Similar to the charging nodes, the base station initiates with the necessary variable initialization. It then employs the Recv function to receive the Cartesian rank and coordinates of the charging nodes, storing this information. Recv is chosen here because the base station anticipates incoming data and can prioritize data reception over other operations.

The primary loop begins and iterates for MAX_iteration times. At the start of each iteration, all essential variables are reset. During the first iteration, the base thread responsible for receiving alerts from charging nodes is initialized. The synchronization of receiving alerts and handling reports is facilitated by shared variables implemented through self-defined structs, as explained in the previous section. These variables are refreshed at the start of every iteration.

Subsequently, the base station enters an inner loop that repeats for the defined number of iterations. Within each iteration, if there are unresolved reports, the base station addresses them. It examines the nearby nodes of the reporting node based on the alert details sent by the reporting node. For all nearby nodes, it assesses the conditions outlined in the flowchart. If all conditions are met (indicating a valid, available, and previously unrecorded nearby node), the base station incorporates it as part of the solution.

After evaluating nearby nodes, if any free nearby node is available, the base station sends the resolution to the reporting node using Isend. This method guarantees more efficient communication, particularly because the base station may have numerous alerts to resolve. In the absence of an available nearby node, the base station follows the same process but with a different message indicating the challenges faced by the reporting node.

Subsequently, the base station logs the details of the current alert into files and checks if it is the second-to-last iteration to send a termination message to the charging node using Isend. This approach optimizes resource utilization, as discussed in the charging nodes section. Following this, it proceeds to the next inner loop.

After the inner loop, the base station instructs the base thread to shut down using a shared variable, as the base thread continues to loop throughout the entire simulation. Finally, before entering the next outer iteration, the base station ensures that all nodes, including the base station, reach an MPI_Barrier to synchronize their start times for each new iteration.

As for the base thread, a periodic listening approach is employed for receiving alerts from charging nodes. This is implemented using Iprobe to test the arrival of a message before initiating a Recv operation. Recv is used here because the subsequent operations depend on the received data.

2. Result Tabulation

2.1 Specification of the Test Platform

The test runs are done on my local machine and the CAAS platform where the specification is provided below.

2.1.1 Local Computer

Hardware Component	Category	Description
CPU	Name	AMD Ryzen 9 5900HS
	Base Speed	3.30Hz
	Cores	8
	Logical Processor	16
Memory	RAM	16GB
Network Bandwidth	Name	-

2.1.2 CAAS

Hardware Component	Category	Description
CPU	Name	Mix of Xeon-Gold and Platinum Processors of different variants
	Base Speed	2.00Hz to 2.70Hz
	Cores	256+
	Logical Processor	-
Memory	RAM	256 GB+
Network Bandwidth	Name	Gigabit ethernet

2.2 Test Specification

The test specs 1 and 2 will be run on both my local machine and CAAS platform but the test spec 3 will only be running on the CAAS platform due to the grid size.

2.2.1 Test Spec 1

Number of attempts run: 2 (one time on each platform)

Aspect	Configuration
Grid Size	3 x 3
Number of Charging Port	5
Number of Iteration	30
Interval per Iteration	5
Minimum Availability Port	2

2.2.2 Test Spec 2

Number of attempts run: 2 (one time on each platform)

Aspect	Configuration
Grid Size	5 x 5
Number of Charging Port	7
Number of Iteration	50
Interval per Iteration	10
Minimum Availability Port	2

2.2.3 Test Spec 1

Number of attempts run: 1 (one time on CAAS platform)

Aspect	Configuration
Grid Size	6 x 6
Number of Charging Port	5
Number of Iteration	50
Interval per Iteration	10
Minimum Availability Port	2

2.3 Results

The tabulated results are attached below where only the average communication time in the test runs in local machine is calculated.

2.3.1 Test 1

2.3.1.1 Local Machine

Aspect	Result
Simulation Duration (seconds)	150.031569
Total Alert Detected	63
Total Messages Exchanged	438
Total Communication time (seconds) (Node to base station)	0.040638
Average Communication Time (seconds) (Node and Node)	0.000424
Average Communication Time (seconds) (Node and Base Station)	0.000687

2.3.1.2 CAAS

Aspect	Result
Simulation Duration (seconds)	150.016051
Total Alert Detected	47
Total Messages Exchanged	330
Total Communication time (seconds) (Node to base station)	0.017339

2.3.2 Test 2

2.3.2.1 Local Machine

Aspect	Result
Simulation Duration (seconds)	250.468511
Total Alert Detected	173
Total Messages Exchanged	1376
Total Communication time (seconds) (Node to base station)	0.196197
Average Communication Time (seconds) (Node and Node)	0.004228
Average Communication Time (seconds) (Node and Base Station)	0.004427

2.3.2.2 CAAS

Aspect	Result
Simulation Duration (seconds)	250.044681
Total Alert Detected	108
Total Messages Exchanged	836
Total Communication time (seconds) (Node to base station)	0.023145

2.3.3 Test 3

2.3.3.1 CAAS

Aspect	Result
Simulation Duration (seconds)	250.050566
Total Alert Detected	211
Total Messages Exchanged	1742
Total Communication time (seconds) (Node to base station)	0.051311

2.3.4 Graph 1

This graph is about the relationship between the number of alerts detected and the number of messages exchanged.

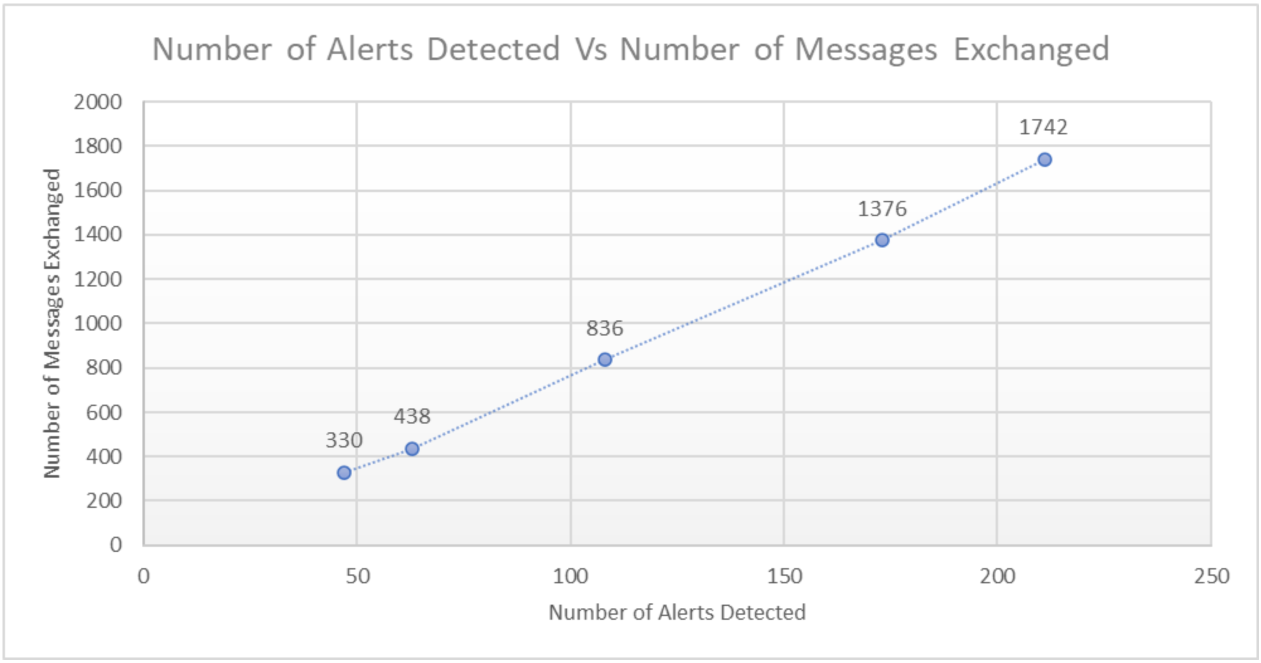


Figure 9: *Graph 1 (No.of Alerts Detected Vs No.of Messages Exchanged)*

2.3.5 Graph 2

This graph shows the relationship between the average number of messages exchanged per alert and the grid size.

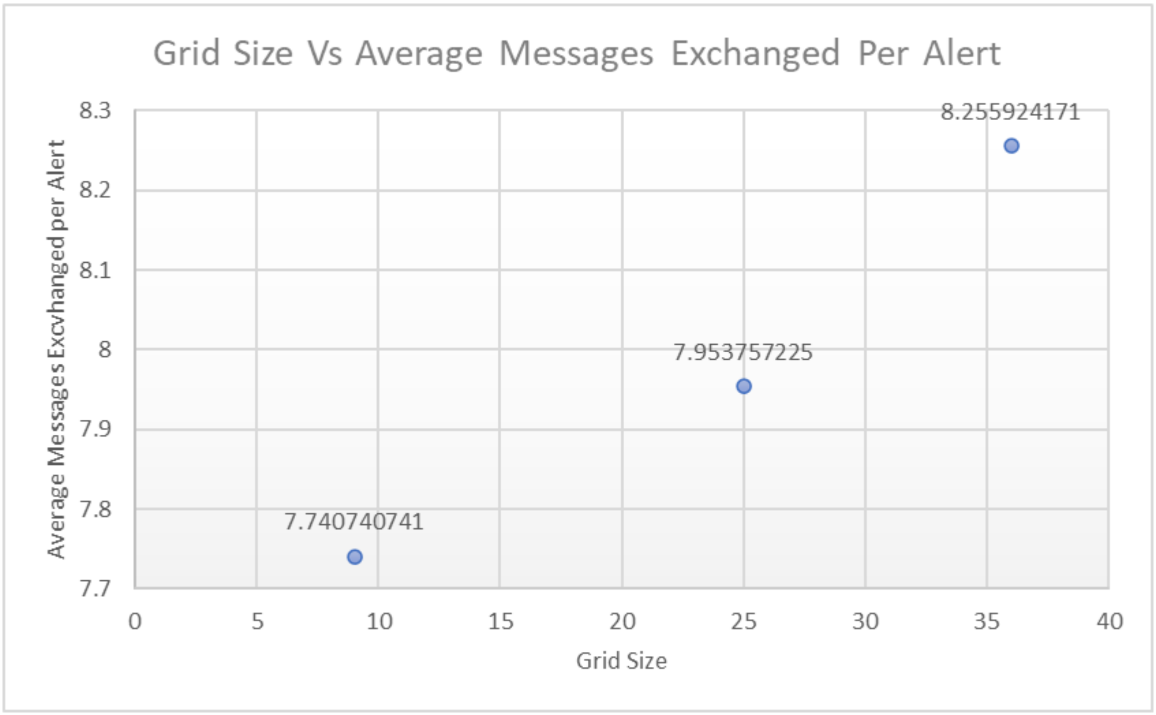


Figure 9: *Graph 2 (Grid size Vs Average Messages Exchanged Per Alert)*

2.3.6 Graph 3

This graph shows the relationship between the total communication time between the nodes and base station and the number of alerts detected.

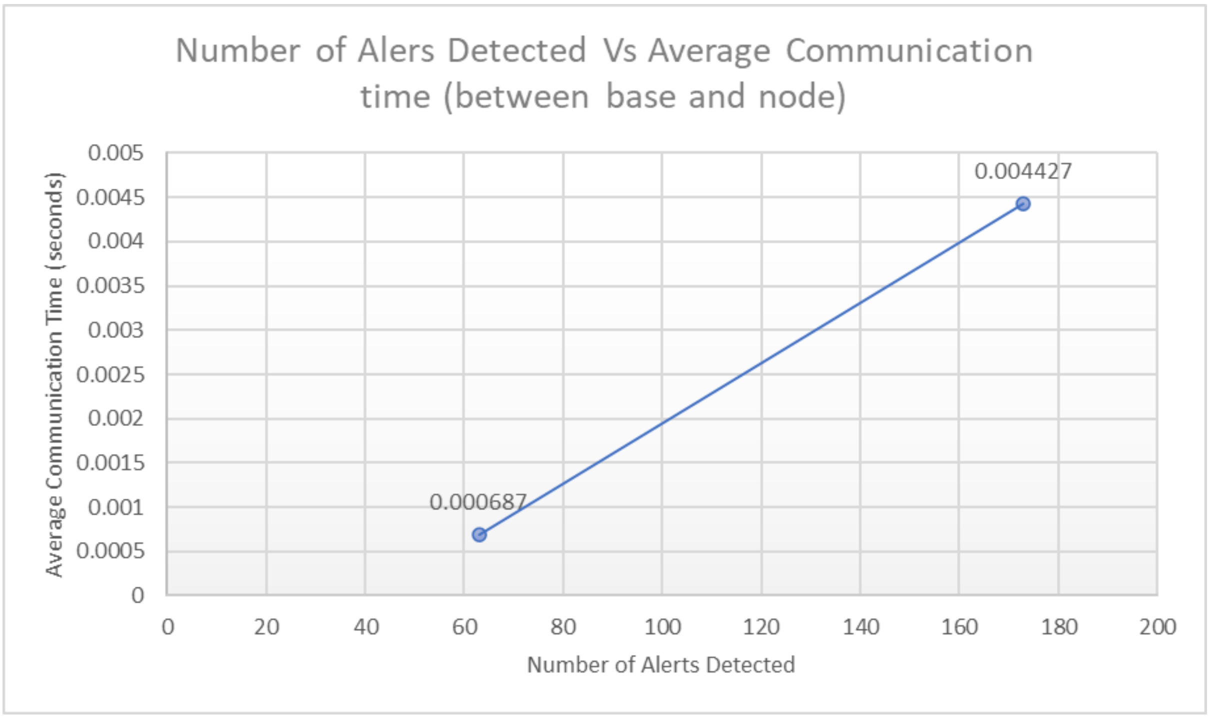


Figure 10: *Graph 3 (No.of Alerts Detected Vs Average Communication Tme between Base Station and Charging Nodes)*

3. Analysis & Discussion

3.1 Analysis against derived hypothesis

The first hypothesis under consideration relates to the increase in the number of alerts detected, leading to a corresponding increase in the number of exchanged messages. This hypothesis finds support in Graph 1 as depicted in Figure 9. The rationale behind this assertion is that when the number of alerts rises, it signifies that more charging nodes have reached their capacity and are sending requests for availability confirmation to their neighbouring nodes. Consequently, this results in a higher volume of messages exchanged between nodes and between the nodes and the base station.

The second hypothesis I would like to discuss pertains to the increase in grid size, resulting in a higher average number of messages exchanged per alert. This relationship is visually represented in Graph 2 (Figure 9). The rationale behind this observation is that with a larger grid size, more charging nodes tend to be positioned closer to the centre of the Cartesian grid. This, in turn, leads to an increase in the number of valid neighbouring nodes compared to smaller grid sizes.

When the number of neighbouring nodes increases, there is a possibility that the nodes triggering the alerts are located nearer to the grid's centre. This proximity necessitates exchanging information with more neighbouring nodes before deciding whether to send an alert to the base station.

The final hypothesis under consideration is the relationship between the increase in the number of alerts and the average communication time required (back and forth) between the base station and the charging nodes. Graph 3 (Figure 10) provides support for this hypothesis.

The reason behind this phenomenon is that when the number of alerts increases, while it is feasible for the base station to continue receiving reports, there's only one thread or process responsible for the validation computations aimed at identifying free nearby nodes to offer as alternatives for the reporting node. This computational limitation ensures that only one report is processed at a time. Consequently, as the number of alerts escalates, each node must wait for a longer duration before receiving its resolution for the alert

3.2 Observation of behaviour of WSN on different platforms

From my tests on different platforms using the same specifications, a notable observation is that the simulation runs faster on the CAAS platform compared to my local machine when the grid size is sufficiently large. The reason for this performance gap is due to the hardware limitations on my local machine. Each MPI process on the local machine utilizes a CPU core, and when threads need to be spawned within that context, it leads to context switching among threads, even for those that don't physically exist. This context switching generates extra overhead and results in slower computational speed.

On the other hand, with HPC resources on CAAS, I can allocate a sufficient amount of resources where each thread or process is directly handled by dedicated hardware CPU cores or threads, which are significantly more powerful than those on my local machine. Additionally, without the need for context switching, the processors can focus entirely on their tasks, leading to improved performance.

Moreover, the presence of superior hardware architecture, including enhanced memory and network bandwidth, on the HPC infrastructure further contributes to the enhanced performance.

3.3 Discussion on limitations and possible solutions

One potential solution to mitigate the issue of extended communication time between the base station and the charging node is to allocate a dedicated thread for handling the sending operation of resolutions. The current implementation assigns a separate thread only for the receiving part, but this adjustment could help balance the workload.

Additionally, rather than relying on a single thread for handling alerts, we could explore parallel computing techniques to concurrently process multiple alerts and find resolutions simultaneously. This parallel approach holds the promise of improving performance, which becomes especially crucial in larger grid sizes for Wireless Sensor Networks (WSN).

However, it's important to bear in mind that implementing these suggestions will require increased hardware resources and meticulous synchronization between threads to prevent race conditions and data corruption.

4. References

Dongarra, J. (1995). Dervied Datatypes vs Pack/Unpack. Retrieved from https://netlib.org/utk/papers/mpi-book/node90.html

ENCSS (n.d.). Introducing MPI and threads. Retrieved from https://enccs.github.io/intermediate-mpi/mpi-and-threads-pt1/

Spagnuolocarmine (n.d.). MPI Process Topologies. Retrieved from https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/mpi-process-topologies

Traff, J.L. (2002). Implementing the MPI Process Topology Mechanism. Retrieved from https://ieeexplore.ieee.org/abstract/document/1592864/authors#authors

5. Appendix

5.1 Assumption Made

I have made certain assumptions about the solution provided by the base station to the reporting node where the solution provided can include the occupied node. It appears that the provided solution may include the nearby node that is already full. This is due to an implementation flaw where the specification asked the node not to report when its quadrant is not full. Consequently, the reporting node may not be able to obtain data about nearby nodes that are full.

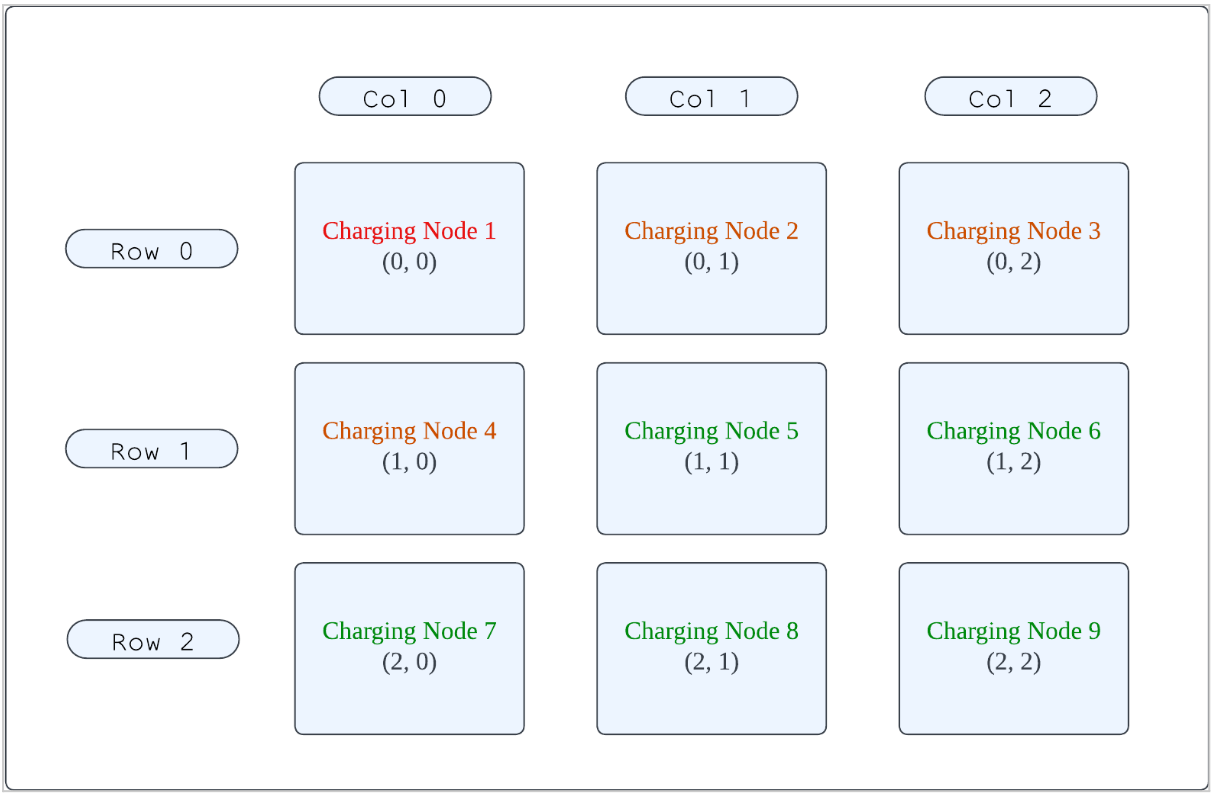


Figure ?: *Example of the assumption scenario*

RED - Full (alert) | ORANGE - Full (with free neighbour nodes) | GREEN - free

In the aforementioned example, when Node 1 is at full capacity and issues a report, its neighbouring nodes, specifically Node 2 and Node 4, refrain from reporting, given the availability of their own neighbours. Consequently, when assessing the nearby nodes of Node 1, namely Node 3, Node 5, and Node 7, we encounter a deficiency in information. This lack of data prevents us from accurately determining whether Node 3 is also at full capacity. As a result, the base station includes Node 3 as one of the 'available' nearby nodes that a vehicle at Node 1 may choose as an alternative charging station. This predicament persists without the introduction of supplementary communication or additional embedded data, a prospect that may prove unfeasible without the allocation of extra time or computational resources.

5.2 External Link

Lucid Chart link