Simulation of an Alert Detection in a Distributed Wireless Sensor Network with MPI

Yin Cheng Chang
School of Information Technology
Monash University Malaysia
ycha0059@student.monash.edu

Jun Qing Lim
School of Information Technology
Monash University Malaysia
jlim0069@student.monash.edu

Word Count - 3583 Words

***Abstract*—A Wireless Sensor Network (WSN) has many important applications and is widely used in areas such as remote environmental monitoring and target tracking. The availability of sensors hardware in a cheaper, smaller, and intelligent form has assisted the society to keep track and avoid disastrous problems from occurring [1]. These sensors are interconnected within a network to communicate with one another to achieve a common desired outcome. This document aims to propose a design and analyze a simulated environment of an alert detection system with sensor devices in an inter-process communication (IPC) architecture using a Cartesian grid, implemented with Message Passing Interface [2]. A Cartesian grid is used to indicate the placement of a set of sensor nodes that monitors the temperatures at different locations. Each sensor nodes periodically measures and simulates the temperature of the ground surface and requests for neighbourhood sensor nodes readings if its temperature exceeds a pre-defined threshold. These neighbourhood values are then used to verify the alert detection before sending an alert reporting to the base station. The base station captures the alert and compares with the temperature readings obtained from an simulated infrared satellite orbiting the planet. A simulation report of the wireless sensor network is then generated for analysis.**

***Index Terms*—wireless sensor network, sensors, grid-based architecture, Cartesian topology, MPI, distributed computing**

I. Introduction

The technology advancement in wireless communications and hardware electronics have made Wireless Sensor Network (WSN) into a reality. These hardware devices and sensors have since led to a surge in demand for efficient sensor architectures [3]. To build a wireless sensor architecture or network, sensors need to communicate with one another to facilitate the process of data exchange - known as Inter-Process Communication (IPC). One method to achieve IPC is with Message Passing Interface (MPI), which provides a set of specifications for processes to send and receive messages to achieve communication within a distributed environment. As communication between nodes and base station can be located at different geographical locations and since sensors can only interact with its immediate adjacent nodes - top, bottom, left and right adjacent nodes, a Cartesian grid-based architecture is the most suitable approach to simulate a wireless sensor network. [4]

This document illustrates the application of MPI in simulating an effective communication between sensor nodes and the base station of a distributed wireless network. Each sensor node and the base station runs on a separate process. A Cartesian grid topology is constructed with each element in the grid representing a sensor node. These sensor nodes communicate directly with its neighbouring nodes and the base station. The base station constructs a thread at the start to simulate the infrared imaging satellite that periodically generates the temperature readings of the sensor nodes. These periodically generated temperatures is then used by the base station to compare with the report received by the ground sensor nodes to determine if an open burning of forest is ongoing and produces a log report that consists of the technical details of it.

In this work, we hypothesize that for an increasing number of simultaneous alerts sent by the sensor nodes, the communication time between the sensor nodes the base station increases linearly (or non-linearly). Thus, the objective of this report is to design and implement wireless sensor network with MPI and to analyze the communication time (and delays) between sensor nodes and base station for an increasing number of alerts.

The following discussion is structured such that Section II illustrates the design and algorithms incorporated in simulating the wireless sensor network, Section III discusses and analyzes the results of the simulation such as its communication time in a range of grid sizes, and the last Section IV provides an overview of the concluded analysis and the possible future work to be expanded.

II. Design Scheme for Wireless Sensor Network

A. Overview of the Design

This section discusses the overview of the design adopted for the wireless sensor network. At the start, all processes are split into its own communicator, whereby the sensor nodes will reside in a new communicator with color 0 while the base station will reside in a new communicator with color 1, illustrated in Algorithm 1. This splitting into new communicators allow the sensor nodes to construct a grid topology based on the filtered communicator as depicted in

Figure 1. Each node is able to communicate with its immediate adjacent nodes and base station only.

Algorithm 1 Splitting processes into its own communicator

```
MPI_Comm newComm
color ← (rank = 0)
MPI_Comm_split(MPI_COMM_WORLD, color, 0, &newComm)
if rank == 0 then
  base(MPI_COMM_WORLD, newComm)
else
  node(MPI_COMM_WORLD, newComm)
end if
```

The number of nodes used in the simulation depends on the number of processes initialized by the user at the start of the program. If a user were to allocate 17 processes, 17 nodes will be generated to simulate the WSN where a single rank will be used as the base station while the remaining ranks as sensor nodes. If size n is the number of processes initiated, then rank 1 to rank $(n-1)$ will be used as the sensor nodes and the first rank 0 will be used as the base station. Base station is selected as rank 0 because MPI allows only rank 0 to accept inputs from standard inputs. Upon executing the program, user will also be required to comply with the rule of $(\text{rows} \times \text{columns}) + 1 = \text{number of processes initiated}$. This is to facilitate the process of creating a grid topology based on the specified layout of the grid. Figure 1 uses an example of a rows $= 4$ and columns $= 3$ layout. *Note: For diagram simplicity, although the actual implementation considers rank 0 as base station, Figure 1 still displays the sensor nodes as from rank 0 to rank* $(n-2)$*, this is to facilitate the following discussions.*

After initializing the grid, each node in the WSN will be able to communicate with the immediate neighbouring nodes (top, right, left, bottom) along with the base station. The following subsections will discuss on the design implementation of the sensor nodes, the infrared imaging satellite, the base station and in simulating an alert detection.

B. Wireless Sensor Network Alert Detection Algorithm

Algorithm 2 Creating a Cartesian Grid Communicator

```
Input: MPI_Comm nodeCommunicator, int rows, int cols
MPI_Comm cartComm
int n ← 2
int dims[2] ← {row, cols}
int size ← size of nodeCommunicator
int wrapAround[2] ← {0,0}
int reorder ← 1
MPI_Dims_create(size, 2, dims)
MPI_Cart_create(nodeCommunicator, n, dims,wrapAround,
reorder, cartComm)
```

1) Creating a Cartesian Grid Communicator: Each node utilizes the new communicator `nodeCommunicator` (which consists of the total number of processes in the simulation minus one) to construct a 2D Cartesian grid topology. `MPI_Dims_create` is used to distribute the processes into the dimensions (rows and columns) specified by the user. In our case as of Figure 1, the dimensions is set to `[4,3]`. After the dimensions of the grid are set, `MPI_Cart_create` is then used to create a 2D topology communication. All communication between sensor nodes will subsequently utilize this new topology `cartComm` communicator.

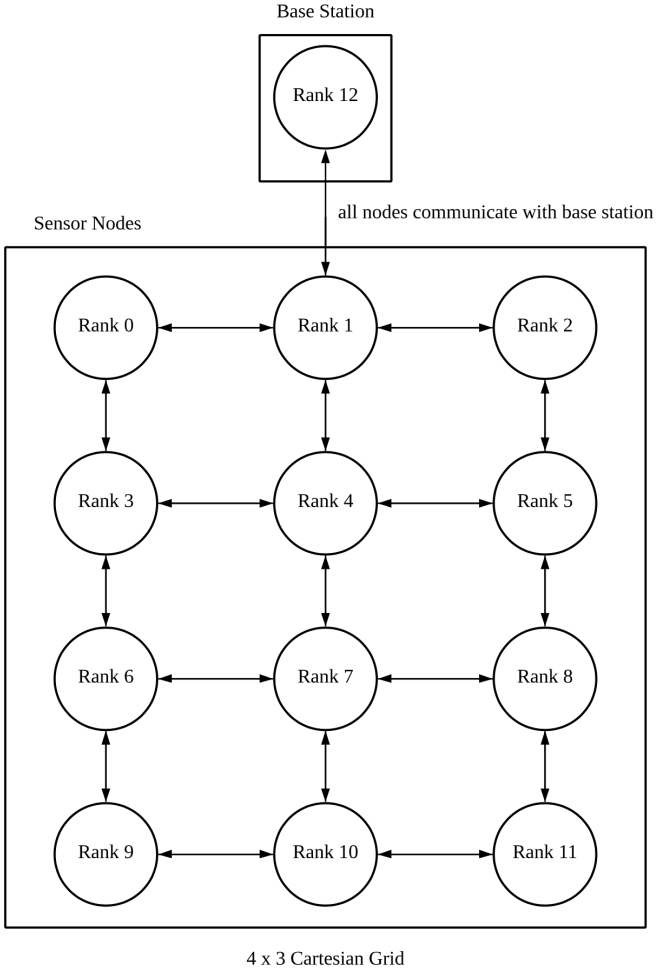


Fig. 1: An example of a wireless sensor network with a dimension of [4,3], the double sided arrows in the figure shows the message exchanging paths when the temperature generated by a sensor is above the threshold.

2) Setting Up Communication: Once the 2D communicator is created, `MPI_Cart_shift` is used to obtain a list of neighbouring ranks where each node will have an array of neighbouring ranks, allowing the sensor nodes to be able to iterate through this list of neighbours and perform all required communication on sending and receiving. As an illustration, Node 4 will have an array `neighbours = [1,3,5,7]`. MPI 2D Cartesian topology provides us not just the flexibility of communicating with the neighbours with least effort but

provides an ability of integrating with physical hardware. This suits the condition of combining the software of a alert detection with a sensor hardware [5].

All sensor nodes will then obtain and send its own MAC and IP address to the base station with `MPI_Pack` and `MPI_Send` for the base station to store the identification information of all sensor nodes. `MPI_Pack` is chosen because it is able to handle complex data more easily for instance multiple strings, which is a list of characters (strings). When the base station receives this packed data of MAC and IP address, it is able store them easily with less hassle, giving an optimized advantage to both sides [6].

Algorithm 3 Setting Up Communication with NodeInfo Structure

```
Struct NodeInfo:
    int rank
    int coord[2]
    int temperature
NodeInfo nodeInfo
nodeInfo.rank = rank of the node
nodeInfo.coord = coordinates of the node
nodeInfo.temperature = an initial default temperature 0
int neighboursCount = get the number of neighbours
NodeInfo neighboursNodeInfo[neighboursCount]
for all available neighbours do
  MPI_Sendrecv(nodeInfo, ..., neighbour, neighboursNode-
  Info, ..., neighbour)
end for
```

Algorithm 3 illustrate the communication setup between sensor nodes. Each node initializes a NodeInfo structure with its own rank, coordinate and a default temperature of 0. The total number of neighbours `neighboursCount` is obtained to construct an array of NodeInfo structure to store the information of all neighbourhood nodes. Each node then sends its own NodeInfo structure to the neighbourhood ranks while receiving the neighbour's NodeInfo structure and storing them in the `neighboursNodeInfo` array. Having this NodeInfo structure allows the data management of each node to become easier. This nearest neighbouring architecture was adopted because it is one of the scalable topologies in distributed systems [7]. Hence, when the number of sensor nodes increases according to the grid size, the performance of the communication between nodes will not be affected, ensuring an efficient system. Note that the usage of `MPI_Sendrecv` can prevent the program goes into a deadlock as this function allows processes to send and receive data simultaneously [8].

3) Detecting and Sending Alerts to Base Station: Algorithm 4 illustrate the alert detection and sending of reports to the base station. Each node generates a random temperature and stores them in the NodeInfo structure. If the temperature exceeds the threshold, it sends a temperature request to all its neighbours and perform a non-blocking receive to obtain its neighbours' temperature readings. Non-blocking send and receive is used

Algorithm 4 Detecting and Sending Alerts to Base Station

```
terminated = false
while !terminated do
  temperature = getRandomNumber()
  nodeInfo.temperature = temperature
  if temperature > threshold then
    for all available neighbours do
      MPI_Isend(..., REQUEST_TAG, ...)
      MPI_Irecv(..., TEMPERATURE_TAG, ...)
    end for
    waiting = true
  end if
  if MPI_Iprobe(...., REQUEST_TAG, ...) then
    MPI_Recv(..., REQUEST_TAG, ...)
    MPI_Isend(temperature, ..., TEMPERATURE_TAG,
    ...)
  end if
  if MPI_Iprobe(...., TERMINATION_TAG, ...) then
    MPI_Recv(..., TERMINATION_TAG, ...)
    clearPendingCommunications()
    terminated = true
    break
  end if
  while waiting do
    if MPI_Iprobe(...., REQUEST_TAG, ...) then
      MPI_Recv(..., REQUEST_TAG, ...)
      MPI_Isend(temperature, ..., TEMPERATURE_TAG,
      ...)
    end if
    if MPI_Iprobe(...., TERMINATION_TAG, ...) then
      MPI_Recv(..., TERMINATION_TAG, ...)
      clearPendingCommunications()
      terminated = true
      break
    end if
    MPI_Testall(all neighbour's temperatures are received)
    if allReceived then
      matchCount = getMatchingCount()
      if matchCount ⩾ 2 then
        sendReport(to base station)
      end if
    end if
  end while
end while
```

for the communication between the adjacent nodes because blocking send has a chance of creating deadlock if the sending message is too large to be sent in once and blocking receive would not return until a message has been received. While the node is waiting to receive the temperature readings from its neighbours, it performs 2 checks:

- If any neighbouring node requests for its temperature, it sends its temperature readings
- If any termination signal is received from the base station,

it terminates and clear all pending communications

When a requesting node has received all neighbouring nodes' temperature readings, it gets the number of nodes that matches its generated temperature by a tolerance range. If the number of matches is $\geqslant 2$, it sends a report with the details of an alert to the base station.

Algorithm 5 Sending Report to Base Station

```
Struct Alert:
    timestamp: alerted time
    matchCount: number of reporting node's matches
    commStartTime: communication start time
MPI_Pack(alert, ..., buffer, ...)
MPI_Pack(nodeInfo, ..., buffer, ...)
for all available neighbours do
  MPI_Pack(neighboursNodeInfo, ..., buffer, ...)
end for
MPI_Send(buffer, ..., baseRank, ...)
```

Algorithm 5 describes the working process of sending a report to the base station. Upon sending an alert report to the base station, it sets the alerted time `timestamp` to be the current time and the `MPI_Wtime()` is set to the `commStartTime` to measure the start of the communication time between sensor node and the base station. The `matchCount` is also set to the number of matching of the reporting node. The Alert message, NodeInfo of reporting node (and its neighbours) are packed into a buffer and sent to the base station. Here, the communication between sensor node and the base station is achieved with blocking communication to ensure the report data is being received by the base station.

C. Base Station Algorithm

Algorithm 6 Simulating Infrared Imaging Temperatures

```
Struct SatelliteData
    timestamp simulated time
    nodeValues an array of temperatures for each node
simulatedValues ← create an array of SatelliteData of size
the number of time units
while true do
  for all time_unit in simulatedValues do
    timestamp ← get current time
    for all node in the grid do
      value ← getRandomNumber()
      simulatedValues[time_unit].nodeValues[node_rank]
      ← value
    end for
    sleep for 500 milliseconds
  end for
end while
```

1) Simulating Infrared Imaging Readings: The base station creates a separate running thread that simulates the infrared imaging satellite temperature readings. As illustrated in Algorithm 6 with Figure 2, a `simulatedValues` array of `SatelliteData`, is created where each element in the array signifies the temperature simulated for each node along with the timestamp of the simulation. These temperature readings are populated continuously until it is stopped by the main caller `baseFunction`. Each infrared simulation is separated with a medium interval of 500 milliseconds to increase the probability for base station to map the values while preventing the thread from fast simulation.

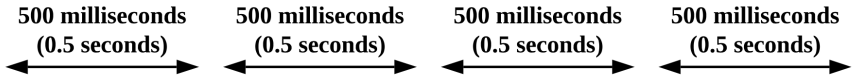
2) Listening for Reports from Sensor Nodes: Base station runs for as long as the number of iterations as provided by the user (or the default no. of iterations if user does not provide any). It executes a blocking receive with a `REPORT_TAG` to ensure that the base station receives the complete report data from the sensor nodes. These reports are packed in the buffer with `MPI_PACKED` type, hence upon receiving the buffer, the base station unpacks the content - which includes the Alert, Reporting Node's information and Neighbouring Node's information. The base station then utilizes the data unpacked to perform computations including:

- calculating the communication time with `MPI_Wtime()` and the timestamp from the alert
- summing the total communication time
- getting the longest communication time
- getting the shortest communication time
- verifying the authenticity of the alert (i.e., true or false alert)

Note: As the clocks are assumed to be synchronized, using MPI_Wtime() to compute the communication time may result in negative values in certain occasions. Under such a circumstance, the communication time is converted to zero.

3) Comparing Reporting Node's Temperature with Infrared's Temperature: After the base station unpacks the temperature of the reporting node, it compares it with the infrared's temperature readings. It searches for the alert's timestamp within the infrared simulated list that is within a time tolerance range (i.e., tolerance of 2 seconds) and returns true if there exists a simulated temperature that is within the tolerance range when compared with the actual reporting node's temperature and returns false otherwise, as described in Algorithm 7. This allows the base station to determine the correctness of the alert report sent by the sensor nodes.

4) Writing to Log File and Completion: When the base station completes the temperature's verification, it writes all the data received into a log file. This behavior is repeated until a specified number of iterations have reached. Then, it sends a termination signal to all sensor nodes and the sensor nodes will receive the termination signal and gracefully end its process after it clears all pending (non-blocking) communications.



Time = Monday 24/10/2020 15:50:01	Time = Monday 24/10/2020 15:50:01	Time = Monday 24/10/2020 15:51:02	Time = Monday 24/10/2020 15:50:02	Time = Monday 24/10/2020 15:50:03
node[0] = 86	node[0] = 78	node[0] = 91	node[0] = 67	node[0] = 111
node[1] = 79	node[1] = 94	node[1] = 71	node[1] = 118	node[1] = 60
node[2] = 80	node[2] = 96	node[2] = 60	node[2] = 98	node[2] = 71
node[3] = 94	node[3] = 105	node[3] = 111	node[3] = 78	node[3] = 91
node[4] = 98	node[4] = 118	node[4] = 100	node[4] = 58	node[4] = 120
node[5] = 117	node[5] = 108	node[5] = 80	node[5] = 109	node[5] = 69
node[6] = 118	node[6] = 108	node[6] = 69	node[6] = 89	node[6] = 80
node[7] = 67	node[7] = 98	node[7] = 120	node[7] = 104	node[7] = 100
node[8] = 79	node[8] = 78	node[8] = 73	node[8] = 84	node[8] = 93
node[9] = 93	node[9] = 86	node[9] = 53	node[9] = 64	node[9] = 113
node[10] = 104	node[10] = 106	node[10] = 113	node[10] = 115	node[10] = 53
node[11] = 108	node[11] = 113	node[11] = 93	node[11] = 95	node[11] = 73
simulatedValues[0]	simulatedValues[1]	simulatedValues[2]	simulatedValues[3]	simulatedValues[4]

Fig. 2: Structure of the satellite infrared imaging simulation with random temperatures being populated iteratively and continously

Algorithm 7 Listening for Reports from Sensor Nodes

```
iterationsCount ← get number of iterations
count ← 0
while count < iterationsCount do
  if userStop then
    break
  end if
  MPI_Recv(buffer, ..., REPORT_TAG, ...)
  MPI_Unpack(buffer, ..., alert, ...)
  MPI_Unpack(buffer, ..., reportingNode, ...)
  MPI_Unpack(buffer, ..., neighboursCount, ...)
  neighboursNodeInfo ← create an array of NodeInfo to
  store the reporting node's neighbours information
  for all available neighbour do
    MPI_Unpack(buffer, ..., neighboursNodeInfo, ...)
  end for
  commTime ← take current time - timestamp in alert
  trueAlert ← isWithinInfraredThreshold(...)
  writeToLogFile(...)
  sleep for 1 second
  increment count
end while
terminated ← true
for all available neighbour do
  MPI_Send(terminated, ..., neighbour, TERMI-
  NATED_TAG, ...)
end for
```

Algorithm 8 Comparing Reporting Node's Temperature with Infrared's Temperature

```
for all time unit in simulatedValues do
  nodeTimestamp ← reporting alert's timestamp
  pthread_mutex_lock(...)
  infraredTimestamp ← timestamp in the simulatedValues
  pthread_mutex_unlock(...)
  if abs(nodeTimestamp - infraredTimestamp) < time's
  threshold then
    nodeTemperature ← reporting node's temperature
    pthread_mutex_lock(...)
    infraredTemperature ← temperature in simulatedVal-
    ues
    pthread_mutex_unlock(...)
    if abs(nodeTemperature - infraredTemperature) < tol-
    erance then
      return true
    end if
  end if
end for
return false
```

III. Results and Discussion

A. Simulation Experiment Setup

This section discusses the experiment setup on simulating the alert detection. To ensure program consistency, all test runs are executed with the same buffer size for communication and same program specifications (but with varying rows, columns and grid sizes).

Note: the command lshw and lscpu are used to obtain the hardware specifications of the machine

TABLE I: Hardware Specification

Platform	MonARCH	Local Virtual Machine
System Memory	214GB	2GB
No. of Logical Processors	24	4
No. of Thread per core	1	1
Processor	Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz	Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
Architecture	x86 64	x86 64

TABLE II: Software Specification

Aspect	Specification
No. of Rows	4
No. of Columns	3
Grid Size	12
No. of Processes	13
No. of Iterations	100
Time Interval between every receive (base station)	1 second
Time Interval between every send (sensor node)	0.5 second

B. Result Tabulation and Test Run

This section discusses the results of the simulation. Base station generates the log file upon receiving a message from a sensor node and logs the information such as sensor node ranks, coordinates, MAC address, IP address, temperature readings, reported time, etc. This is also added along with a summary of the entire simulation including total communication time, simulation time, no. of alerts, etc.

For readability purposes, only a section of the log files will be displayed in this report. The descriptions of the information in the log file are as follow.

- **Iteration**: the iteration number which an alert is detected
- **Logged Time**: the time where the log file was written
- **Alert Reported Time**: the time where the alert was sent by the sensor node to the base station
- **Alert Type**: true/false alert when compared with the satellite temperature readings
- **Node's rank**: rank number of the sensor node
- **Node's coordinate**: coordinate of the sensor node in the grid
- **Node's Temperature**: temperature detected by the node itself
- **Node's MAC address**: MAC address of the machine where the node is executed
- **Node's IP address**: IP address of the machine where the node is executed
- **Satellite's Reporting Time**: time where the temperature of the reporting node was simulated
- **Satellite's Reporting Temperature**: temperature of the reporting node simulated by the satellite

```
============================================================
Iteration: 11
Logged Time: Tue Oct 27 01:13:05 2020

Alert Reported Time: Tue Oct 27 01:12:57 2020
Alert Type: True
Number of Adjacent Matches to Reporting Node: 3
Communication Time (seconds): 8.014628

Reporting Node Information:
    Rank: 6
    Coordinate: (1, 2)
    Temperature: 88
    MAC Address: B0:02:D3:01:00:00
    IP Address: 192.168.122.1

Adjacent Nodes Information:
    Rank: 5
    Coordinate: (1, 1)
    Temperature: 88
    MAC Address: 40:01:DD:01:00:00
    IP Address: 192.168.122.1
    -------------------------
    Rank: 7
    Coordinate: (1, 3)
    Temperature: 88
    MAC Address: 00:57:ED:32:FD:7F
    IP Address: 192.168.122.1
    -------------------------
    Rank: 2
    Coordinate: (0, 2)
    Temperature: 111
    MAC Address: 20:B2:93:EA:71:7F
    IP Address: 192.168.122.1
    -------------------------
    Rank: 10
    Coordinate: (2, 2)
    Temperature: 93
    MAC Address: 00:00:00:00:05:00
    IP Address: 192.168.122.1
    -------------------------

Infrared Satellite Information:
    Reporting Time: Tue Oct 27 01:13:01 2020
    Reporting Temperature: 87
============================================================
```

Fig. 3: An extraction of the base station log file

Figure 3 shows the 11^{th} iteration of the simulation when executed on MonArch.

```
================================================
            Summary Report
================================================
Total Simulation Time (seconds): 103.042538
Shortest Communication Time (seconds): 0.000000
Longest Communication Time (seconds): 93.041109

Total Communication Time (seconds): 3502.018781
Total Messages Received: 100
Average Communication Time (seconds): 35.020188

Total True Alerts Count: 37
Total False Alerts Count: 63
```

Fig. 4: Summary of the base station log file

Figure 4 displays the summary section of the log file that is generated at the end, that is when the program shuts down gracefully. This summary report includes the total simulation time, shortest and longest communication time, total communication time, total messages received, average communication time and total number of true and false alerts.

C. Result Analysis

In this section, we analyze the wireless sensor network from the log file produced by the simulation. The formula below shows the calculation for the average communication time between the sensor nodes and the base station. We will use this to as our analysis tool. The simulation is executed on both machines as specified in Table I.

$$c = \frac{1}{m} \sum_{i=1}^{m} r_i - s_i$$

Where c is the average communication time of sending a report, r is the receive time of the report, s is the sending time of the report and m is the number of report within a time window.

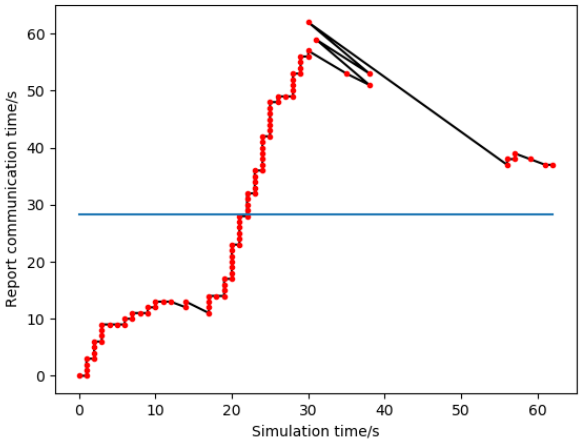


Fig. 5: The communication time between the sensor nodes and the base station for each iteration against the simulation time, executed on MonARCH

Figure 5 illustrates the communication time between the sensor nodes and the base station versus the simulation time when executed on MonARCH. There are 100 red dots where each red dot indicates each report sent (since there are 100 iterations, so there are 100 red dots or reports). While the blue line indicates the average communication time.

It is observed that the communication time increases over time, this is attributed to fact that base station can only receive one report at any given time interval, thus increasing the size of the queue for pending reports, where some reports will have higher communication time than the reports sent earlier by certain sensor nodes. When the communication time of certain alerts is raised, this pushes the average (overall) communication time upwards (blue line). It is also observed that there is little decline in the communication time between 30^{th} to 55^{th}. This could be due to the latency communication issues in MonARCH, resulting less reports to be sent at certain given time, hence allowing some pending reports to be served earlier, lowering the communication time of those reports. We now proceed to examine and analyze the simulation over local virtual machine.

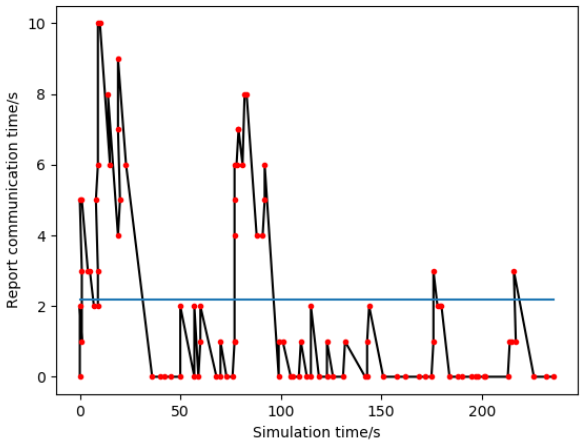


Fig. 6: The communication time between the sensor nodes and the base station for each iteration against the simulation time, executed on local virtual machine

Figure 6 shows the communication time between the sensor nodes and the base station versus the simulation time when running on a local virtual machine. Similarly, there are 100 red dots where each red dot signify the iteration / report received and the blue line signifies the average communication time.

It is observed that it takes approximately 60 seconds to complete 100 iterations in MonARCH based on Figure 5. While in local virtual machine, Figure 6 tells us that 100 iterations take approximately 250 seconds. In addition of that, the average communication time in MonARCH from Figure 5 is about 30 seconds, while in virtual machine (Figure 6) the average communication time is about 2 seconds. These time differences are the result of difference in hardware utilization as described in Table I. Local virtual machine is running on a slower 4-cores processor, thus these processes have additional context switching time between nodes when oversubscribed to execute 13 processes, lengthening the total time taken (degrading the performance) [9], [10] to complete one iteration and reducing the number of reports sent simultaneously.

At this point, running on MonARCH is the best platform to provide us an accurate measurement of the communication time (with increasing number of simultaneous alerts sending), because it does not oversubscribe and degrade the performance. So, the subsequent section will discuss on the simulation results for average communication time when executed on MonARCH.

TABLE III: Statistical results of the various grid sizes with 5 test runs

	grid size	5×5	4×3	2×2
Average Communication Time/s	**Run #1**	37.53	28.23	0.23
	Run #2	35.33	32.91	27.71
	Run #3	36.52	25.64	17.47
	Run #4	37.04	29.26	11.29
	Run #5	36.23	30.27	0.72
	Average	36.53	29.262	11.484
Maximum Communication Time/s	**Run #1**	73	62	2
	Run #2	67	65	50
	Run #3	70	46	41
	Run #4	69	60	22
	Run #5	68	50	4
	Maximum	73	65	50
Total Number of Simultaneous Report Sent	**Run #1**	82	64	16
	Run #2	74	72	70
	Run #3	79	54	65
	Run #4	77	63	53
	Run #5	78	64	26
	Average	78	63.4	46

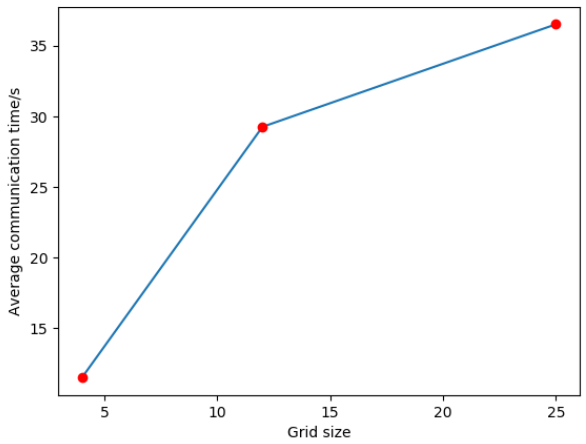


Fig. 7: Average communication time of the 3 different grid sizes (2×2, 4×3, 5×5)

Table III shows the statistical results obtained from running 3 different grid sizes - namely 2×2, 4×3, 5×5 for 5 times, when executed on MonARCH and the average of the communication time is obtained. The average communication time is then plotted against the grid sizes as shown in Figure 7.

Here, it is seen that when the grid sizes increase, the average communication time between the sensor nodes and the base station increases accordingly. This can be inferred from the standpoint that when the number of grid sizes increase, the number of sensor nodes increase, thereby causing the number of reports sent simultaneously to the base station at a given time interval to increase as well. The receive of only one report by base station at a time interval makes these pending reports sent to reside in the queue, increasing the communication time for waiting in the queue and raising the overall (average) communication time. This strengthens our hypothesis that the communication time increases linearly along with an increasing number of simultaneous alerts, as observed from Figure 7.

IV. Conclusion

In conclusion, we have demonstrated the simulation of a wireless sensor network with a 2D Cartesian topology with MPI. By using the topology functions provided by the MPI, we are able to create communication between the sensor nodes. Taking into account the hypothesis made in Section I, and all observations and analysis made via the simulations in Section III, we can conclude our hypothesis that the higher the number of reports sent simultaneously, the higher the average communication time throughout the simulation. The objective of analyzing the communication time between sensor nodes against simultaneous alerts sending were also achieved.

From the analysis above, one observational conclusion that we can make is the communication time does not provide an accurate measurement between the sensor nodes and the station. One possible idea that could be expanded on this work is to establish synchronization between the processes. This will produce a more accurate timestamp to calculate the communication time and resolve the latency issues between the nodes. For instance, we could achieve clock synchronization by using Cristian's Algorithm where we let the base station to be the time server and all sensor nodes' clock are adjusted to the base station's time. We also noticed that the messages sent from sensor nodes to the base station are not encrypted. This makes the message to be insecure and could leak along the transmission. Thus, another possible path to expand on this work is to establish a secure communication between the sensor nodes, satellite and the base station. For instance, we could utilize a proposed identity-based encryption and allow only decryption by certain members of the wsn [11], security the message transmitted from the sensor nodes to the base station.

References

[1] Yick, J., Mukherjee, B., & Ghosal, D. (2008). Wireless sensor network survey. Computer networks, 52(12), 2292-2330.

[2] Open, MPI. (2013). A high performance message passing library. Available on: http://www.open-mpi.org.

[3] Othman, M. F., & Shazali, K. (2012). Wireless sensor network applications: A study in environment monitoring system. Procedia Engineering, 41, 1204-1210.

[4] Ke, W. C., Liu, B. H., & Tsai, M. J. (2007). Constructing a wireless sensor network to fully cover critical grids by deploying minimum sensors on grid points is NP-complete. IEEE Transactions on Computers, 56(5), 710-715.

[5] Gropp, W. D. (2018, September). Using node information to implement mpi cartesian topologies. In Proceedings of the 25th European MPI Users' Group Meeting (pp. 1-9).

[6] Sun, X. H. (2003, December). Improving the performance of MPI derived datatypes by optimizing memory-access cost. In 2003 Proceedings IEEE International Conference on Cluster Computing (pp. 412-419). IEEE.

[7] Hoefler, T., Rabenseifner, R., Ritzdorf, H., de Supinski, B. R., Thakur, R., & Träff, J. L. (2011). The scalable process topology interface of MPI 2.2. Concurrency and Computation: Practice and Experience, 23(4), 293-310.

[8] Luecke, G. R., Zou, Y., Coyle, J., Hoekstra, J., & Kraeva, M. (2002). Deadlock detection in MPI programs. Concurrency and Computation: Practice and Experience, 14(11), 911-932.

[9] Kamal, H., & Wagner, A. (2012, September). Added concurrency to improve MPI performance on multicore. In 2012 41st International Conference on Parallel Processing (pp. 229-238). IEEE.

[10] Open, MPI. (2013). FAQ: Running MPI jobs. Available on: https://www.open-mpi.org/faq/?category=running.

[11] Chu, C. K., Liu, J. K., Zhou, J., Bao, F., & Deng, R. H. (2010, April). Practical ID-based encryption for wireless sensor network. In Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (pp. 337-340).