


ASSESSMENT COVER SHEET

	Unit Name and Code:	FIT3143 Parallel Programming	

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

Implementation of hybrid MPI / OpenMP simulated wireless sensor network

Faculty of Information Technology
Monash University, Clayton
Melbourne, Australia
student.monash.edu

Abstract—The problem of simulating efficient communications between processes which have a certain topology is explored for a 2D grid of nodes along with a common base station. Nodes are allowed to contact its direct neighbour and the base station, but communications between the base station and nodes should be minimized. In this work we explore using the Message Passing Interface for communications, defining a topology on the shared communicator of the node network. This results in substantially simpler code and drastically reduces the number of messages sent from the nodes to the base station. OpenMP is used for the encryption of all messages before transmission and the decryption of the received messages, with a speedup of greater than one.

Index Terms—cartesian topology, IPC, MPI, OpenMP, encryption

I. INTRODUCTION

The need for communication between different instances of software arises in many applications. Modern Inter-Process Communication (IPC) technologies attempt to allow the efficient exchange of data between processes [1]. One way to achieve IPC is to use the Message-Passing-Interface (MPI), which is a specification that allows processes to send messages to other processes to achieve communication [2]. MPI effectively allows the user to abstract the concept of communicating across a network into the sharing (point-to-point or broadcast) of messages.

In this work, we examine the problem of effective communication between *nodes* of a simulated wireless network, each of which runs on a separate process. The nodes in this network can only communicate directly with their adjacent nodes. We hypothesize that there is a clean, simple way to implement this in MPI; the method chosen is to give these nodes their own communicator endowed with a topology that allows nodes to query the communicator about its neighbours, in this case its adjacent nodes. By using the functionality of the OpenMPI library, we can group nodes which can communicate with each other into a 'neighbourhood'; using MPI Neighbourhood Collectives we can then aggregate data from every neighbour of a node through a single function call [3]. The simplicity and efficiency demonstrates MPI's power in abstracting such communications.

Another aspect to be explored is the speedup in data encryption when multiple logical cores are used in the encryption algorithm. MPI was designed for the communication between distributed memory, single-core machines running on

a common network — this has lead to the development of other standards such as OpenMP and OpenCL to provide parallelism on shared-memory architectures [2]. One common strategy employed in many situations is a hybrid MPI/OpenMP solution: use MPI for the inter-node communications and use OpenMP to leverage the multiple cores of a single node. We will attempt to make use of OpenMP to speed up the encryption / decryption algorithm on the nodes.

II. IPC IMPLEMENTATION IN MPI

A. Topology and Neighbourhoods

1) *Implementing a grid topology*: For this project we need to simulate a 5×4 wireless grid of sensors as nodes and a single base station. In terms of MPI, each node will be a process and the base station will be its own process. The layout of the grid is depicted in Figure 1.

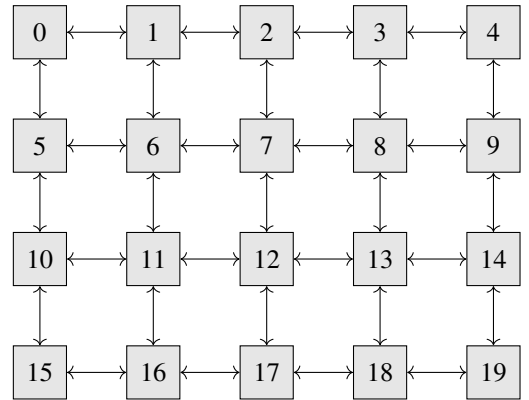


Fig. 1: The grid topology of the wireless sensor network. Processes 0 to 19 share an MPI communicator which encodes the adjacency information, represented here as arrows between processes indicating possible direct communication. Not shown is the base station (process 20) which is not in the wireless sensor communicator.

This type of topology (cartesian) is very common among scientific applications — for example, see [4] for using this type of IPC for solving Laplace's equation on a 2D grid topology. As such, the MPI specification has a function for creating a n dimensional cartesian grid with dimensions of $\{x_1, \dots, x_n\} \in \mathbb{N}^n$. In our case, $n = 2$ and $x_1 = 5$, $x_2 = 4$.

Algorithm 1 Splitting the processes into the grid network with its own communicator.

```

1: procedure SPLITCOMM(comm)
2:   reorder_ranks  $\leftarrow$  0
3:    $n \leftarrow 2$ 
4:   dimensions[1, 2]  $\leftarrow$  {5, 4}
5:   periodicity[1, 2]  $\leftarrow$  {0, 0}
6:   new_communicator  $\leftarrow$  MPI_CART_CREATE(comm,
   n, dimensions, periodicity, reorder_ranks)
7:   if new_communicator  $\neq$  MPI_COMM_NULL then
8:     WIRELESS_NODE_CODE(new_communicator)
9:   else
10:    BASE_STATION_CODE()

```

In Algorithm 1 we split the processes in the old communicator into two sets of processes. Once we give **MPI_Cart_create** the information about the topology we want, it returns a new communicator for the processes which have been selected to be in the grid, or a null communicator for the processes which are not in the grid. In our case if **MPI_COMM_WORLD** has 21 processes, then on splitting, $5 \times 4 = 20$ processes will be in the new wireless grid communicator and we tell these processes to run the simulated wireless sensor node code. The remaining single process will act as the base station. Note that we can communicate between nodes in the grid using the new communicator, whilst still being able to communicate from node to base station using the old communicator.

2) *Communication between nodes in the grid:* Once we have a cartesian topology communicator we can use neighbourhood collectives to allow communication between nodes which share an edge in Figure 1.

In Algorithm 2 every process has its own array of readings which it wishes to send to its direct neighbours. It also has a buffer **neighbour_readings** which it puts the received data from its neighbours in. The main idea of this algorithm is to find out its neighbours through querying the topology of the communicator, then instructing all processes to send its readings and gather all neighbour readings. The sending and receiving is all done in a single function call. For the receiving operation, a custom defined MPI datatype **recv_type** is used to interleave the data received so that they are stored in chronological order.

3) *Advantages of this method:* One of the main advantages of using MPI Cartesian topology and MPI Neighbourhood Collectives is to abstract and encapsulate the concept of sending and receiving to nearest neighbours — once the communicator topology and all the parameters are set up then a single **MPI_Neighbour_Allgatherv** call will do the following:

- 1) Locate all targets for sending
- 2) Send to targets
- 3) Loop over neighbours and receive
- 4) Organise the data in chronological order

Internally, the function uses **MPI_Cart_shift** and

Algorithm 2 Finding node degree $\in [0, 2n]$ and locations of neighbours — then using it to gather N chunks of data from each nearest-neighbour (stencil operation).

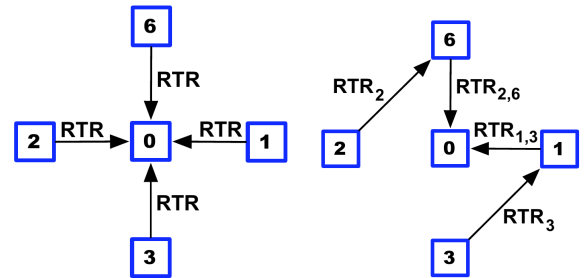
```

1: procedure STENCILGATHER(comm, readings[1, ..., N],
   neighbour_readings[1, ..., 2nN])
2:   locations[1, ..., 2n] is empty array
3:   degree  $\leftarrow$  0
4:   for  $i \in [1, \dots, n]$  do
5:     source, dest  $\leftarrow$  MPI_CART_SHIFT(comm, i, -1)
6:     if dest  $\neq$  MPI_PROC_NULL then
7:       location[degree]  $\leftarrow 2 \times i$ 
8:       degree  $\leftarrow$  degree + 1
9:     source, dest  $\leftarrow$  MPI_CART_SHIFT(comm, i, 1)
10:    if dest  $\neq$  MPI_PROC_NULL then
11:      location[degree]  $\leftarrow 2 \times i + 1$ 
12:      degree  $\leftarrow$  degree + 1
13:   recv_counts[1, ..., degree]  $\leftarrow$  {1, ..., 1}
14:   displacements[1, ..., degree] is empty array
15:   for  $i \in [0, \dots, \text{degree} - 1]$  do
16:     displacements[locations[i]]  $\leftarrow$  i
17:   MPI_NEIGHBOUR_ALLGATHERV(readings,
   N, MPI_INT, neighbour_readings, recv_counts,
   displacements, recv_type, comm)

```

MPI_Sendrecv to do all the work [5]. Other implementations of sharing data in a stencil pattern such as the one found in [6] need to do a lot more work in keeping track of array reads/writes and micro-manage the communications. Using our method, we do not need to worry about the how many **MPI_Sends** and **MPI_Recvs** we need to call, or the order of calling such that the blocking operations do not lock.

In addition, we may benefit from optimizations if the MPI provider changes the implementation to be more efficient. In [7], the authors discuss ways to optimize the stencil gather operation (shown in Figure 2).



(a) Point-to-point communication (b) Collective tree communication

Fig. 2: Gathering data from process 0's neighbours directly in (a), and a possible optimization in (b). Images from [7].

B. Event detection criteria (extended)

We will use the 'sliding window' method of detecting events. In this method, we cache the activation values of the

neighbours. We then look up to W iterations of activation values from the node's neighbours, where W is the window size. Figure 3 shows a window of size $W = 3$ of a node with 3 neighbours, with the I being the current iteration — the values from the previous two iterations are being consider for possible repeats.

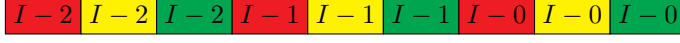


Fig. 3: The elements received from neighbours (Three neighbours in this diagram, each with its own color). For a window size of $W = 3$, the activation values of three iterations $\{I - 2, I - 1, I\}$ are considered. Since this node has three neighbours, this corresponds to $3W = 9$ activation values being searched for repetitions .

Algorithm 3 Check for $\geq \text{minrepeats}$ of activation values on the neighbour's activation values using a sliding window approach and the Misra-Gries streaming algorithm.

```

1: procedure CHECKFORDUPLICATES(degree,
  neighbour_readings[1, ..., 2nN], minrepeats)
2:   totalEvents  $\leftarrow$  0
3:   log is empty list
4:   lower  $\leftarrow$  0
5:   upper  $\leftarrow$  0
6:   for  $I \in [1, \dots, I_{\max}]$  do
7:     if upper  $< I_{\max} \times \text{degree}$  then upper  $\leftarrow$  upper + 1
8:     (events, values, repeats)  $\leftarrow$ 
      MISRA_GRIES(neighbour_readings, lower, upper,
        minrepeats)
9:     if events  $> 0$  then
10:       log[totalEvents]  $\leftarrow$  (events, values, repeats)
11:       totalEvents  $\leftarrow$  totalEvents + 1
12:     if  $I > W$  then lower  $\leftarrow$  lower + 1
13:   return totalEvents, log

```

Algorithm 3 shows how the sliding window works — we move the window's upper and lower bounds on each iteration and then pass the readings within these bounds to a streaming algorithm by Misra and Gries to check for reaped elements greater than or equal to the given **minrepeats** value [8]. The full Misra-Gries algorithm can be found in the provided citation and is not reproduced here for brevity purposes. The nodes each compute their local count of events and a log of which iteration, the number of repeats and the activation values to report to the base station.

C. Base station reporting

Once the node finishes determining events and finishes storing the values to log, it encrypts and sends the log to the base station where it is decrypted. The base station tallies the total number of activations and writes the node's report to the log file.

III. ENCRYPTION AND DECRYPTION

The encryption algorithm used in this assignment is the Skipjack algorithm modified to 32-bit block size . The Skipjack algorithm was original designed by the United States' National Security Agency (NSA). It has a unbalanced Feistel network using an 80-bit key [9]. The original source code can be found at <https://github.com/gray/crypt-skip32-xs/blob/master/skip32.c>.

In the encryption process, the plaintext is taken four bytes at a time. The four bytes are put through 24 Feistel rounds; Figure 4 depicts four rounds of the algorithm. The XOR operation is denoted by \oplus and the box represents a round function G where the key is used as a seed in producing output. To decrypt a ciphertext, the entire process is reversed using the same key.

To speed up encryption and decryption with OpenMP, we note that since the Feistel network depends on the result of the previous round in its calculation of the next round, a single 32-bit block cannot be accelerated using parallelism. However, if we have multiple blocks to encrypt/decrypt, since there exists no dependency between the blocks, we can give the different blocks to different threads for encryption and decryption.

The chosen encryption/decryption algorithm performs well generally across all architectures since the main operation used is XOR, where as other algorithms such as AES require special instruction sets (SSE etc.) to perform well. This makes it suited towards embedded environments if the simulation code were ever turned into production code compiled to the embedded architectures.

IV. RESULTS AND DISCUSSION

Four runs were completed on MonARCH, each with 1000 iterations. To test the effectiveness of the OpenMP speedup, two runs were done with one logical CPU core per node and two runs were done with two logical CPU cores per node. The results are tabulated in the Appendix, where a chart of the events over time can also be found.

A. OpenMP encryption/decryption speedup

We hypothesized that increasing the number of logical cores runs on will increase the speed of the encryption / decryption process. However, it is surprising how little speed up was observed — $1.36\times$ for the encryption and $1.71\times$ for the decryption. Given that twice as many cores are used in OpenMP for the encryption / decryption, we should see a speedup close to 2, but this is not realized, probably due to the large overhead of the fork-join of OpenMP threads when using more than one logical core.

We can assume that encryption and decryption take similar times on the same amount of data. Since nodes can have 2 to 4 times the data to decrypt than to encrypt, the overhead from the fork-join process accounts for less time proportionally, which is why the speedup factor is greated for decryption than encryption.

Note that since the encryption/decryption algorithm is so fast, the times have large margins of error.

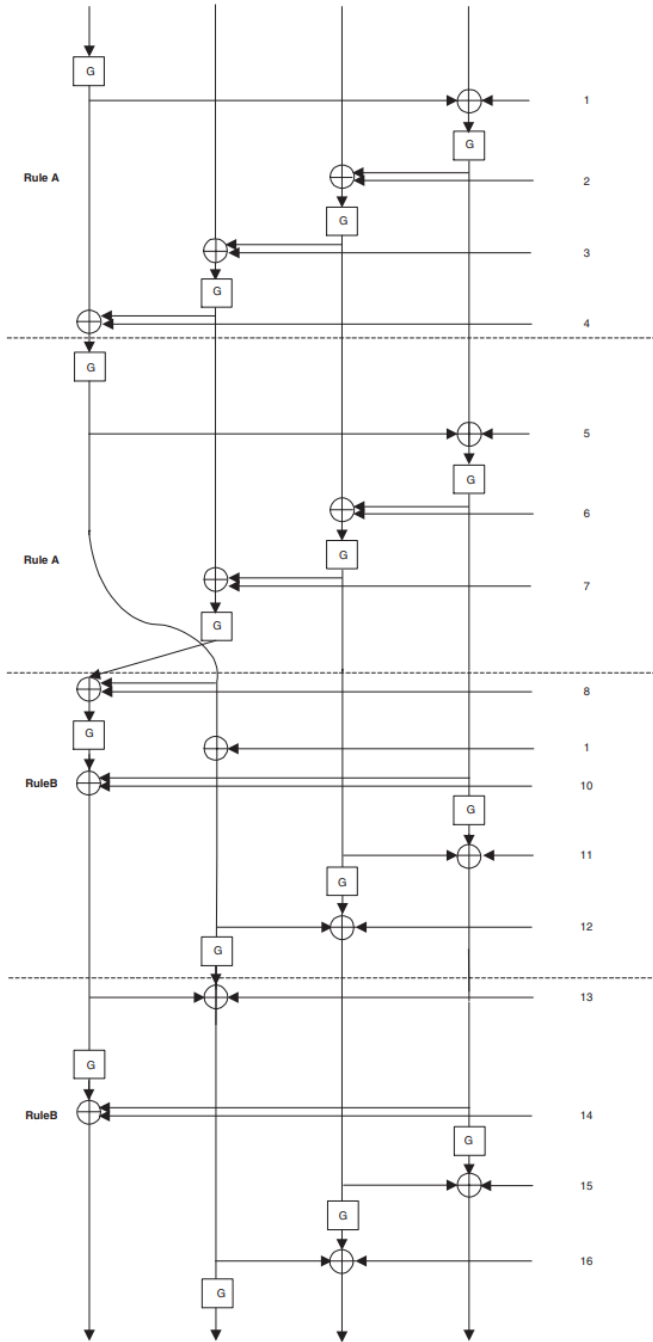


Fig. 4: Four Feistel rounds of the 32-bit Skipjack algorithm. Each line represents one byte. Image taken from [10]

B. Example data encryption and decryption

To show how scrambled the data is, here are 10 random integers between 0 and 100 on the first line, and their encrypted representations on each line afterwards.

```
20 17 55 16 25 7 60 47 38 26
-374268971
1641427967
1953390658
1023324914
-1147701219
2066851471
-854172993
1536280991
1687832615
-1401317755
```

Some integers are mapped to negative numbers, and there is seemingly no relation between the input integer and the output integer after the encryption.

C. Communication between base and nodes

In the design of our program we have effectively distributed the task of finding events (repeated activation values) to the nodes themselves. This means that we only need to send the base station a message for each event rather than sending all activation values and letting the base station determine events. Furthermore, since the project specifications does not state that the event reporting has to be done in real time, we have each node keep a log of events in its own memory and only send the full report at the conclusion of the simulation.

This means that effectively one message to the base can report multiple events and we see in the logs that the communication time between the base and nodes is negligible and barely detectable using `MPI_Wtime`.

This does cause some issues — the nodes themselves have to have enough memory to fit all its own readings and its neighbours readings, but this can be adjusted by taking blocks of readings at a time and just appending the events of the current block of readings to the internal event log.

Compared to the nodes which have a very well defined topology, the ad hoc communication between the base and the nodes requires much more code and is more difficult to comprehend. This furthers our hypothesis in that using MPI topologies simplifies the code and abstracts the concepts to something familiar, i.e. a 2D cartesian grid.

V. CONCLUSION

We have demonstrated the possibility to model a simulated wireless network consisting of a 2D grid of nodes using the MPI framework of IPC. By leveraging abstractions of topology and neighbourhoods provided by MPI, we are able to exchange data between nodes using a single MPI Neighbourhood Collective function call, which is performant and simple conceptually. The encryption and decryption of data using OpenMP has been shown to have a speedup of greater than one. This type of hybrid MPI / OpenMP solution is extremely applicable to other scientific computations which use multiple CPUs per node using OpenMP and communications between nodes using MPI. For example, we could adapt the program to 3D topologies — each MPI process would be responsible for a

subspace of Euclidean space \mathbb{R}^3 . Then, rather than encryption, OpenMP could be used to speed up partial differential equation (PDE) solvers for problems such as electromagnetics (e.g. solving Laplace's equation $\nabla^2\Phi = 0$) or thermodynamics (e.g. solving the heat equation $\nabla^2u = \partial_t u$).

REFERENCES

- [1] M. Khan and M. A. Shah, "Inter-process communication, MPI and MPICH in microkernel environment: A comparative analysis," in *2017 23rd International Conference on Automation and Computing (ICAC)*. IEEE, Sep. 2017.
- [2] A. Friedley, G. Bronevetsky, T. Hoefer, and A. Lumsdaine, "Hybrid MPI," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*. ACM Press, 2013.
- [3] S. H. Mirsadeghi, J. L. Traff, P. Balaji, and A. Afsahi, "Exploiting common neighborhoods to optimize MPI neighborhood collectives," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, Dec. 2017.
- [4] C. Guiffaut and K. Mahdjoubi, "A parallel FDTD algorithm using the MPI library," *IEEE Antennas and Propagation Magazine*, vol. 43, no. 2, pp. 94–103, Apr. 2001.
- [5] T. Hoefer and J. L. Traff, "Sparse collective operations for MPI," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, May 2009.
- [6] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Review*, vol. 51, no. 1, pp. 129–159, Feb. 2009.
- [7] T. Hoefer and T. Schneider, "Optimization principles for collective neighborhood communications," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2012.
- [8] J. Misra and D. Gries, "Finding repeated elements," *Science of Computer Programming*, vol. 2, no. 2, pp. 143–152, Nov. 1982.
- [9] L. Knudsen and D. Wagner, "On the structure of skipjack," *Discrete Applied Mathematics*, vol. 111, no. 1-2, pp. 103–116, Jul. 2001.
- [10] A. Biryukov, "Skipjack," in *Encyclopedia of Cryptography and Security*. Springer US, 1999, pp. 586–587.

APPENDIX

Raw data from test runs — all timings are listed in seconds.

Computer specifications		MonARCH Xeon-E5-2667-v3 at 3.30 GHz 1 or 2 cores per node 1GB memory per node Gigabit Ethernet			
	Cores per node	Reported events	Total communication time	Total encryption time	Total decryption time
Run #1	1	151	0.256215	0.014389	0.044088
Run #2	1	138	0.666689	0.01385	0.042485
Run #3	2	134	0.119435	0.009367	0.028729
Run #4	2	152	0.147993	0.011361	0.0219
Average	1	143.75	0.297583	0.0141195	0.0432865
	2			0.010364	0.0253145
Speedup	N/A	N/A	N/A	1.36	1.71

