FIT3143 Tutorial Week 10

PARALLEL ALGORITHM DESIGN – ADVANCED MPI TOPICS

OBJECTIVES

- Understanding the principles of MPI Scatter, MPI Gather and Virtual topologies.
- Solving parallel programming problems using MPI Scatter and Gather functions.
- Design solutions using MPI virtual topologies.

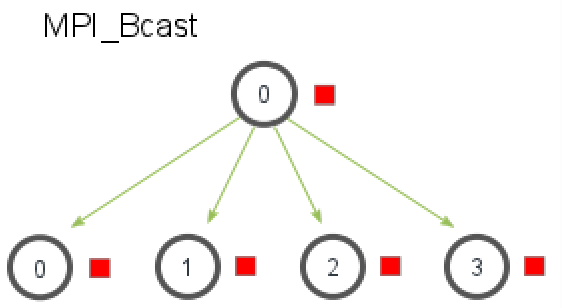
Note: Tutorials are not assessed. Nevertheless, please attempt the questions to improve your unit comprehension in preparation for the labs, assignments, and final assessments.

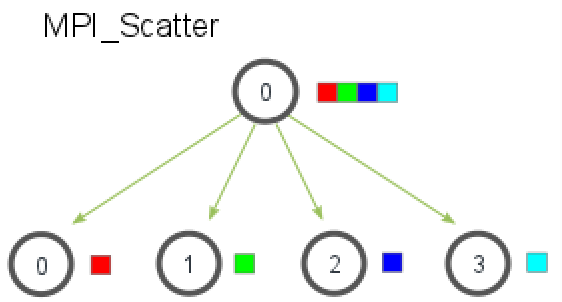
QUESTIONS

1. How is MPI Scatter different from MPI Broadcast? In addition, how is MPI Gather different from MPI Reduce?

   Solution referred from: https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

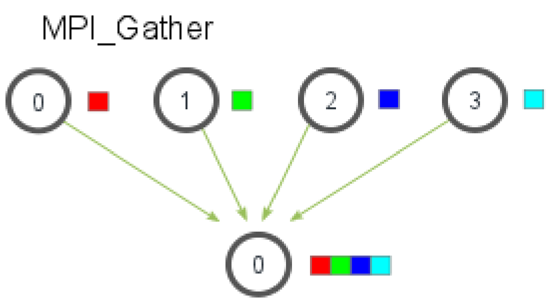
   `MPI_Scatter` is a collective routine that is very similar to `MPI_Bcast`. `MPI_Scatter` involves a designated root process sending data to all processes in a communicator. The primary difference between `MPI_Bcast` and `MPI_Scatter` is small but important. `MPI_Bcast` sends the ***same*** piece of data to all processes while `MPI_Scatter` sends ***chunks of an array*** to different processes. Check out the illustration below for further clarification.





In the illustration, `MPI_Bcast` takes a single data element at the root process (the red box) and copies it to all other processes. `MPI_Scatter` takes an array of elements and distributes the elements in the order of process rank. The first element (in red) goes to process zero, the second element (in green) goes to process one, and so on. Although the root process (process zero) contains the entire array of data, `MPI_Scatter` will copy the appropriate element into the receiving buffer of the process.

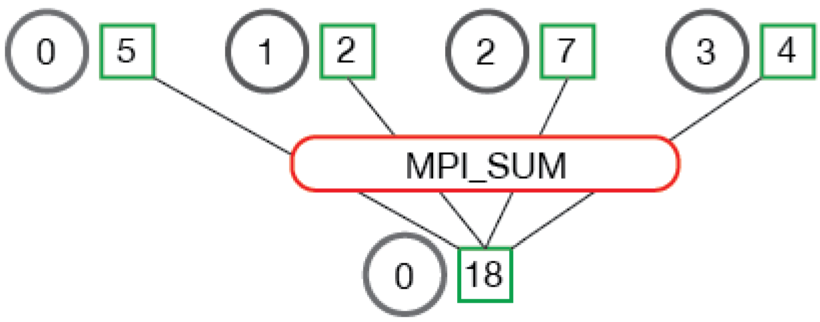
`MPI_Gather` is the inverse of `MPI_Scatter`. Instead of spreading elements from one process to many processes, `MPI_Gather` takes elements from many processes and gathers them to one single process. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching. Below is a simple illustration of this algorithm.



Similar to `MPI_Scatter`, `MPI_Gather` takes elements from each process and gathers them to the root process. The elements are ordered by the rank of the process from which they were received.

Similar to `MPI_Gather`, `MPI_Reduce` takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result.
Below is an illustration of the communication pattern of `MPI_Reduce`.

MPI_Reduce



In the above, each process contains one integer. `MPI_Reduce` is called with a root process of 0 and using `MPI_SUM` as the reduction operation. The four numbers are summed to the result and stored on the root process.

2. This following code file implements a simple parallel vector multiplication using MPI. Modify its code to replace the MPI Send and Recv functions with MPI Scatter and MPI Gather functions.

   Note: There is no need to compile the code, focus on writing a logically correct code to replace the MPI Send and Recv functions with MPI scatter and gather functions.

   Solution (focus on the main() function):

```
int main()
{
    int row1, col1, row2, col2;
    int i, j;
    int my_rank;
    int p;
    int *pArrayNum1 = NULL;
    int *pArrayNum2 = NULL;
    int *pArrayNum3 = NULL;

    int *pSubArrayNum1 = NULL;
    int *pSubArrayNum2 = NULL;
    int *pSubArrayNum3 = NULL;

    int offset;
    struct timespec start, end, startComm, endComm;
    double time_taken;

    MPI_Status status;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
 // Get current clock time.
clock_gettime(CLOCK_MONOTONIC, &start);

if(my_rank == 0)
{
 // STEP 1: Only the root process reads VA.txt and VB.txt
into its own memory
 printf("Rank: %d. MPI Implementation version 2. Commence
Reading\n", my_rank);

      // Call the read from file function
 pArrayNum1 = ReadFromFile("VA.txt", &row1, &col1);

      if(pArrayNum1 == 0)
 {
      printf("Rank: %d. Read failed.\n", my_rank);
      MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
      return 0;
 }


      // Call the read from file function
 pArrayNum2 = ReadFromFile("VB.txt", &row2, &col2);

      if(pArrayNum2 == 0)
 {
      printf("Rank: %d. Read failed.\n", my_rank);
      free(pArrayNum1);
      MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
      return 0;
 }

 if(row1 != row2 || col1 != col2)
 {
     printf("Rank: %d. Not matching row and column values
between the arrays.\n", my_rank);
     free(pArrayNum1);
     free(pArrayNum2);
     MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
     return 0;
 }
 printf("Rank: %d. Read complete\n", my_rank);

}

// Broadcast the arrays to all other MPI processess in the
group
MPI_Bcast(&row1, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&row2, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Basic workload distribution among MPI processes
// Row based partitioning or row segmentation
```

```
int elementsPerProcess = row1 / p;
int elementsPerProcessRemain = row1 % p;

int startPoint = my_rank * elementsPerProcess;
int endPoint  = startPoint + elementsPerProcess;
if(my_rank == p-1){
 // Last node, factor in the remainder
 endPoint += elementsPerProcessRemain;
}


pSubArrayNum1   =   (int*)malloc((endPoint-startPoint)   *
sizeof(int));
pSubArrayNum2   =   (int*)malloc((endPoint-startPoint)   *
sizeof(int));
pSubArrayNum3   =   (int*)malloc((endPoint-startPoint)   *
sizeof(int));

// STEP 2: Send relevant portions the arrays to all other
MPI processess in the group
clock_gettime(CLOCK_MONOTONIC, &startComm);
if(my_rank == 0){
 pArrayNum3 = (int*)malloc(row1 * sizeof(int)); // Can use
row2 as an alternative
}

// Solution for q2
MPI_Scatter(pArrayNum1,    elementsPerProcess,    MPI_INT,
pSubArrayNum1,     elementsPerProcess,     MPI_INT,     0,
MPI_COMM_WORLD);
MPI_Scatter(pArrayNum2,    elementsPerProcess,    MPI_INT,
pSubArrayNum2,     elementsPerProcess,     MPI_INT,     0,
MPI_COMM_WORLD);

clock_gettime(CLOCK_MONOTONIC, &endComm);
time_taken = (endComm.tv_sec - startComm.tv_sec) * 1e9;
 time_taken    =    (time_taken    +    (endComm.tv_nsec    -
startComm.tv_nsec)) * 1e-9;
printf("Rank:  %d.  Comm  time  (s):  %lf\n\n",  my_rank,
time_taken);


// STEP 3 - Parallel computing takes place here
printf("Rank: %d. Compute\n", my_rank);
for(i=0; i<elementsPerProcess;i++){
 for(j = 0; j< 500; j++){
          pSubArrayNum3[i]    =    pSubArrayNum1[i]    *
pSubArrayNum2[i];
  }
}
```

```
// STEP 4 - Send the arrays results back to the root process
// Solution for q2
MPI_Gather(pSubArrayNum3, elementsPerProcess, MPI_INT,
pArrayNum3, elementsPerProcess, MPI_INT, 0,
MPI_COMM_WORLD);

if(my_rank == 0){

 // STEP 5: Write to file
 printf("Rank: %d. Commence Writing\n", my_rank);
 WriteToFile("VC.txt", pArrayNum3, row1, col1);
 printf("Rank: %d. Write complete\n", my_rank);

 free(pArrayNum3);
}

free(pSubArrayNum1);
free(pSubArrayNum2);
free(pSubArrayNum3);

// Get the clock current time again
// Subtract end from start to get the CPU time used.
clock_gettime(CLOCK_MONOTONIC, &end);
time_taken = (end.tv_sec - start.tv_sec) * 1e9;
 time_taken = (time_taken + (end.tv_nsec - start.tv_nsec))
* 1e-9;
printf("Rank: %d. Overall time (s): %lf\n\n", my_rank,
time_taken); // tp

MPI_Finalize();
return 0;
}
```

IMPORTANT: You should modify the code using MPI Scatterv and MPI Gatherv if the vector is not evenly distributable among processes.

3. Explain the concept of MPI virtual topologies and its benefits.

MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology.

Virtual topologies are built upon MPI communicators and groups. Must be ”programmed” by the application developer.

Two Types: Cartesian, Graphs Cartesian: 1D, 2D, 3D arrangements

Benefits:

- Convenient: Useful for applications with specific communication patterns - patterns that match an MPI topology structure.
- Improved communication efficiency: hardware architectures may impose penalties for communications between successively distant ”nodes”.
- A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.

4. A high-rise building management is planning to install a series of fire alarm sensors representing a form of a 3D mesh architecture as illustrated in Figure 1.

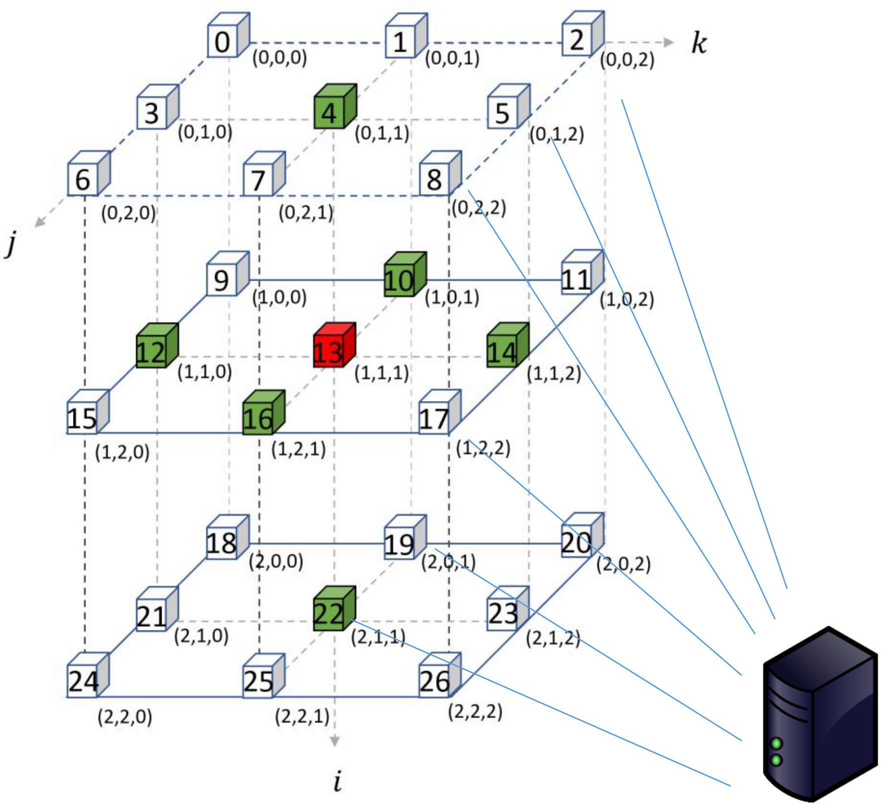


Figure 1: 3D mesh architecture of the fire alarm sensors.

In Figure 1, each sensor can directly communicate with its immediate adjacent sensors (i.e., top, bottom, left, right, front, and back). Each sensor can also directly communicate with the server.

Based on this architecture, there are two options to implement the fire alarm computing and communication system.

Option A:

I) At each interval, the sensor measures the temperature and exchanges the temperature with its neighbours.

II) If the exchanged temperature values and measured values exceed a particular threshold, the sensor sends an alert to the server, which is located outside of the building.

III) The server listens for incoming alerts from the sensor nodes and logs it.

Option B:

I) At each interval, the sensor measures the temperature and directly sends the measured value to the server.

II) The server periodically receives temperatures readings from all sensors. At each iteration, the server then compares the temperature values of each node with the adjacent nodes to determine if a fire is detected. In other words, all of the computations are done at the server.

Before implementing the architecture, a simulator is created using Message Passing Interface (MPI). Based on the aforementioned description and illustration, answer the following questions:

a) Compare **Options A** and **B**. In particular, what type of distributed computing architectures (in relation to computation and communication) do **Options A** and **B** represent respectively?

b) What is the advantage of **Option A** to that of **Option B** in terms of message passing communication?

The following code snippet describes an attempt to simulate the sensor based on **Option A**. This code first splits the communicator between the server and sensor nodes. Then, a 3D grid using MPI virtual topology is created for the MPI processes simulating the sensors. This code however is incomplete. Based on the given code snippet, answer the remaining questions.

c) Why should the **MPI_Cart_create()** function be invoked by all of the MPI processes simulating the sensor nodes? What happens if any one of the MPI processes simulating the sensor nodes does not invoke the **MPI_Cart_create()** function?

d) When passing in the first argument into the **MPI_Cart_create()** function, why doesn't this function use the default **MPI_COMM_WORLD** communicator?

e) The **MPI_Cart_coords()** function computes the process coordinates in a 3D cartesian topology based on the given rank in a group. This function essentially performs a 1D (i.e., rank index) to 3D (i.e., coordinates) mapping based on the dimension of the grid. Assuming this function is not available and that you are required to manually calculate the coordinates, what are the equations which map a 1D rank value, ***x*** to the 3D coordinates ***i, j, k*** based on the ***row width***, ***column width*** and ***depth*** of the grid?

f) The **MPI_Cart_rank()** function computes the process rank in communicator based on the given Cartesian coordinate. This function essentially performs a 3D (i.e., coordinates) to 1D (i.e., rank index) mapping based on the dimension of the grid. Assuming this function is also not available and that you are required to manually calculate the the 1D cartesian rank, what is the equation to which maps the 3D coordinates ***i, j, k*** to a 1D rank value, ***x***, based on the ***row width***, ***column width*** and ***depth*** of the grid?

Hint: Refer to this website on mapping for some guidance.

g) The **sensor_io()** function in the given code below requires each node to exchange the temperature values with its adjacent nodes. However, this region of the code is incomplete. Complete this region of the code by using non-blocking MPI send and receive functions to exchange the temperature values. You do not need to copy the entire given code into your answer template. Only write the missing code in your answer template. Use a **for** loop to implement the send and receive functions and use the available variables in the given code below. You may opt to create new variables or arrays.

Note: There is no need to compile the code, focus on writing a logically correct code.

Code snippet implementing Option A (Refer to the /* INCOMPLETE REGION - START */ in the code to complete part (g)).

```
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <unistd.h>
#include <string.h>

#define NUM_RANGE 100
#define SHIFT_ROW 0
#define SHIFT_COL 1
#define SHIFT_DEP 2
#define DISP 1

int sensor_io(MPI_Comm world_comm, MPI_Comm comm);
int MeasureTemperature();
bool CheckTemperature(int* recvValues, int temp);
int server_io(MPI_Comm world_comm, MPI_Comm comm);

int main(int argc, char **argv){
    int rank, size;
    MPI_Comm new_comm;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MPI_Comm_split( MPI_COMM_WORLD,rank == size-1, 0, &new_comm);
    if (rank == size-1)
     server_io( MPI_COMM_WORLD, new_comm );
    else
     sensor_io( MPI_COMM_WORLD, new_comm );
```

```
    MPI_Finalize();
    return 0;
}
int sensor_io(MPI_Comm world_comm, MPI_Comm comm) {
     int ndims=3, size, my_rank;
     int reorder, my_cart_rank, ierr, worldSize;
     int nbr_i_lo, nbr_i_hi;
     int nbr_j_lo, nbr_j_hi;
     int nbr_k_lo, nbr_k_hi;
     MPI_Comm comm3D;
     int dims[ndims],coord[ndims];
     int wrap_around[ndims];
     char buf[256];

     MPI_Comm_size(world_comm, &worldSize); // size of the world
     communicator
     MPI_Comm_size(comm, &size); // size of the slave communicator
     MPI_Comm_rank(comm, &my_rank); // rank within the slave communicator

     dims[0]=dims[1]=dims[2]=0;
     MPI_Dims_create(size, ndims, dims);

     wrap_around[0] = 0;
     wrap_around[1] = 0;
     wrap_around[2] = 0;
     reorder = 1;
     ierr = 0;
     ierr = MPI_Cart_create(comm, ndims, dims, wrap_around, reorder,
     &comm3D);
     if(ierr != 0) printf("ERROR[%d] creating CART\n",ierr);

     MPI_Cart_coords(comm3D, my_rank, ndims, coord);
     MPI_Cart_rank(comm3D, coord, &my_cart_rank);

     MPI_Cart_shift( comm3D, SHIFT_ROW, DISP, &nbr_i_lo, &nbr_i_hi);
     MPI_Cart_shift( comm3D, SHIFT_COL, DISP, &nbr_j_lo, &nbr_j_hi);
     MPI_Cart_shift( comm3D, SHIFT_DEP, DISP, &nbr_k_lo, &nbr_k_hi);

     MPI_Request send_request[6];
     MPI_Request receive_request[6];
     MPI_Status send_status[6];
     MPI_Status receive_status[6];

     sleep(my_rank);
     int temp = MeasureTemperature();
     int recvValues[6] = {-1, -1, -1, -1, -1, -1};

     /* INCOMPLETE REGION - START */
     /* COMPLETE PART (g) HERE */

     /* INCOMPLETE REGION - END */

     if(CheckTemperature(recvValues, temp) == 1){
     sprintf(buf, "Fire alert from slave %d at Coord: (%d, %d, %d).
     Temperature: %d\n", my_rank, coord[0], coord[1], coord[2], temp);
     MPI_Send(buf, strlen(buf) + 1, MPI_CHAR, worldSize-1, 0, world_comm);
     }
     MPI_Comm_free( &comm3D );
     return 0;
}
bool CheckTemperature(int* recvValues, int temp){
     int retVal = 0;
```

```
      for (int i = 0; i < 6; i++) {
       retVal = retVal && (recvValues[i] == temp || recvValues[i] == -1);
      }
      return retVal;
}
int MeasureTemperature() {
      srand(time(NULL));
      int number;
      number = rand() % (NUM_RANGE + 1);
      return number;
}
int server_io(MPI_Comm world_comm, MPI_Comm comm){
      // Not applied to the context of the question

      }
```

Solution:

a)
Option A - Decentralized computing architecture
Option B - Centralized computing architecture

b) With Option A, the sensor node only communicates with the server should there be an alert. In Option B, each sensor node sends a temperature value to the server at each iteration. The communication overhead is higher for Option B as compared to Option A.

c)
MPI_CART_CREATE returns a handle to a new communicator to which the cartesian topology information is attached. Therefore, each MPI process which belongs to the virtual topology must call this function to obtain a new communicator representing the virtual topology.

The program risks returning an error at runtime

d)
The default MPI_COMM_WORLD communicator includes the server as one of the process ranks. As such, MPI_COMM_WORLD needs to be split into a separate communicator for the MPI processes simulating the sensor nodes, which will then be used to create the virtual topology.

e)

k = x / (row_width * column_width)
x -= (k * row_width * column_width)
i = x % row_width
j = (x / row_width)

f)

x = i +(j * row_width) + (k * row_width * column_width) g)

```
int neighbors[6] = {nbr_j_lo, nbr_j_hi, nbr_i_lo, nbr_i_hi, nbr_k_lo, 
nbr_k_hi};


      for (int i = 0; i < 6; i++) {

            MPI_Isend(&temp, 1, MPI_INT, neighbors[i], 0, comm3D, 
&send_request[i]);

            MPI_Irecv(&recvValues[i], 1, MPI_INT, neighbors[i], 0, comm3D, 
&receive_request[i]);

      }

      MPI_Waitall(6, send_request, send_status);
      MPI_Waitall(6, receive_request, receive_status);
```