Activity 1: 25 mins

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

MPI Scatter	**MPI Broadcast**
Both involve one pro[illegible]her processes data In code, both sendin[illegible] run function • The sending p[illegible]ding portion • The receiving [illegible]he receiving portions	
Sends **chunks of array** to **diff processes**	Sends **same data to everyone**
parameters: • Send buffer • Sendcount • sendtype • Recvcount • Recvtype • Root • Comm ○ Communication "world"	

MPI Scatter v
Use if need to send different sized values

Same as MPI Scatter except for
• Sendcount (changed from int ->arr of ints)
  ○ integer array (of length group size) specifying the number of elements to send to each processor
• Displs (new parameter)
  ○ integer array (of length group size). Entry i specifies the displacement (relative to sendbuf from which to take the outgoing data to process i
  ○ Means displacement from start
  ○ Essentially starting index of where data to be sent starts???

MPI Gather	**MPI Reduce**
Collects data from all processes into one process	
Parameters (same as scatter) • Send buffer • Send count	Parameters • Send buffer • Receive buffer

• Send type
• Receive buffer
• Receive count
  ○ Assumption that all processes sending same sized values
  ○ If not use mpi_gatherv
• Receive type
• Root
• Communicati
	• Count
	• Datatype
	• MPI_OP
	• Root
	• Comm

MPI ALL Gather (ev

Activity 2: 25 mins

Modify its code to replace the MPI Send and Recv functions with MPI Scatter and MPI Gather functions.

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


```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>
#include <string.h>
#include <mpi.h>

// Function prototype
int* ReadFromFile(char *pFilename, int *pOutRow, int *pOutCol);
void WriteToFile(char *pFilename, int *pMatrix, int inRow, int inCol);

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 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 function
        pArray nFile("VB.txt", &row2, &col2);

        if(pArr
        {
                  Read failed.\n", my_rank);
            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 startPoint = my_rank * elementsPerProcess;
    int endPoint  = startPoint + elementsPerProcess;
```

```
	// 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
		offset = elementsPerProcess;

/* replace w/mpi_sca
		for(i =
ArrayNum1 + offset, elementsPerProcess, MPI_INT, i,
0, MPI_COMM_WO
ArrayNum2 + offset, elementsPerProcess, MPI_INT, i,
0, MPI_COMM_WO
			offset += elementsPerProcess;
		}
	}else{
		pArrayNum1 = (int*)malloc((endPoint-startPoint) * sizeof(int));
		pArrayNum2 = (int*)malloc((endPoint-startPoint) * sizeof(int));
		pArrayNum3 = (int*)malloc((endPoint-startPoint) * sizeof(int));
		MPI_Recv(pArrayNum1, (endPoint - startPoint), MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);
		MPI_Recv(pArrayNum2, (endPoint - startPoint), MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);
	}
*/

mpi_scatter((int*)pArrayNum1, elementsPerProcess, MPI_INT,(int*)pArrayNum1,
elementsPerProcess, MPI_INT, 0, MPI_COMM_WORLD, &status);

mpi_scatter((int*)pArrayNum2, elementsPerProcess, MPI_INT,(int*)pArrayNum2,
elementsPerProcess, MPI_INT, 0, MPI_COMM_WORLD, &status);


	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++){
		// The second loop is intentionally included to increase the computational time
		for(j = 0; j< 500; j++){
			pArrayNum3[i] = pArrayNum1[i] * pArrayNum2[i];
		}
	}
```

```
        // STEP 4 - S                                    ults back to the root process

/* replace w/ MPI ga
        if(my_rank ==
        {
                // Initia                                d on Rank 0's workload
                offset = elementsPerProcess;

                for(i = 1; i < p; i++){
                        MPI_Recv((int*)pArrayNum3 + offset, elementsPerProcess, MPI_INT, i,
0, MPI_COMM_WORLD, &status);
                        offset += elementsPerProcess;
                }
                // 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);
        }else{
                MPI_Send((int*)pArrayNum3, (endPoint - startPoint), MPI_INT, 0, 0,
MPI_COMM_WORLD);
        }
*/
mpi_gather((int*)pArrayNum3, elementsPerProcess, MPI_INT,(int*)pArrayNum3,
elementsPerProcess, MPI_INT, 0, MPI_COMM_WORLD, &status);



        free(pArrayNum1);
        free(pArrayNum2);
        free(pArrayNum3);

        // 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;
}

// Function definition
int* ReadFromFile(char *pFilename, int *pOutRow, int *pOutCol)
```

```
{
        int i, j;
        int row, col;
        FILE *pFile = fopen(pFilename, "r");
        if(pFile == NULL)
        {
                printf("                file\n");
                return
        }

        fscanf(pFile, '                l);
        int *pMatrix =                 col * sizeof(int)); // Heap array

        // Reading a 2D matrix into a 1D heap array
        for(i = 0; i < row; i++){
            for(j = 0; j < col; j++){
                fscanf(pFile, "%d", &pMatrix[(i * col) + j]);
                }
        }
        fclose(pFile);

        *pOutRow = row; // Dereferencing the pointer
        *pOutCol = col; // Dereferencing the pointer
        return pMatrix;
}

void WriteToFile(char *pFilename, int *pMatrix, int inRow, int inCol)
{
        int i, j;
        FILE *pFile = fopen(pFilename, "w");
        fprintf(pFile, "%d\t%d\n", inRow, inCol);
        for(i = 0; i < inRow; i++){
            for(j = 0; j < inCol; j++){
                fprintf(pFile, "%d\t", pMatrix[(i * inCol) + j]);
                }
                fprintf(pFile, "\n");
        }
        fclose(pFile);
}
```

Vector_Cell_Product_MPI_SendRecv.c
Displaying Vector_Cell_Product_MPI_SendRecv.c.

Activity 3: 15 mins

Explain the concept of MPI virtual topologies and its benefits.

Activity 4: 25 mins

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.

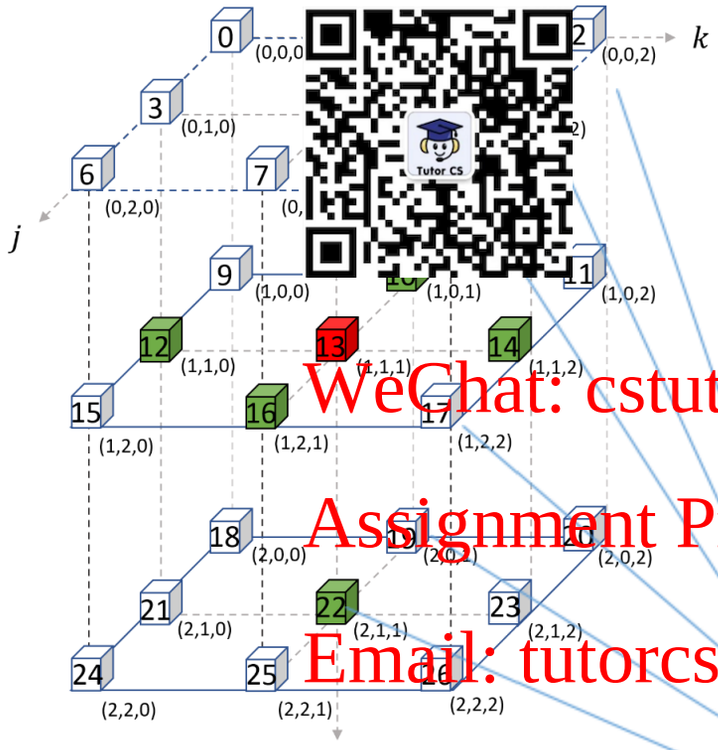

Figure 1: 3D mesh architecture of the fire alarm sensors.

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 temperature 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 terms of computation and communication) do **Options A** and **B** represent respectively?

Option A	Option B
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.	I) At each interval, the sensor measures the temperature and directly sends the measured value to the server. II) The server periodically receives temperature 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
Cartesian topology ● Decentralised ○ Better no single point of failure ○ Communications reduced as local comms are done ->(communication overhead reduced)	Master-slave system ● Centralised system

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

Less communication overhead

The following code s[illegible]n attempt to simulate the sensor based on **Option A**. This code first spl[illegible]tor between the server and sensor nodes. Then, a 3D grid using MPI vir[illegible]ated for the MPI processes simulating the sensors. This code however is[illegible]d 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?

Cart_create creates cartesian communication world

Why is it called by all processes?

- Needs to call in order to be a part of the cart comm world

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?

Passing base communicator has base process rank

- Use to create new structure with cart create

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 to the 3D coordinates ***i, j, k*** based on the ***row width***, ***column width*** and ***depth*** of the grid?

rank/row

K = rank // (num_row*num_col)

I = rank % (num_col)

J = rank % num_row

Rank = (num_row*num_col)/k

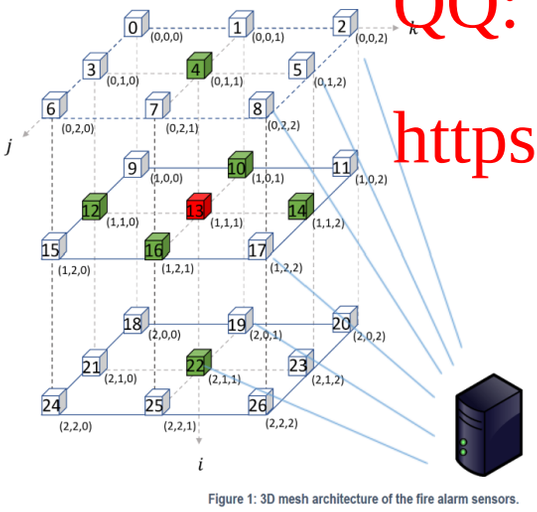

Figure 1: 3D mesh architecture of the fire alarm sensors.

$k = 13 // (3 \times 3) = 13 // 9 = 1$

$I = 13 \% 3 = 1$

$J = 13 \% 3 = 1$

$\therefore$ Rank is @ (1, 1, 1)

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then −

Show Examples

Operator	De	Example
+	Adds two operand	A + B = 30
−	Subtracts second	st. A − B = -10
*	Multiplies both op	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

MPI comm spilt
Params: ● Base comm ● Colour (basically group) ○ control of subset assignment (nonnegative integer). **Processes with the same color are in the same new communicator** ● Key ○ control of rank assignment (integer) ○

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?

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_Fina
    return 0
}
int sensor_i                    comm, MPI_Comm comm) {
    int ndi                     nk;
    int reo                     k, ierr, worldSize;
    int nbr
    int nbr
    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 CheckTe                    cvValues, int temp){
    int ret
    for (in                     ++) {
    retVal                      Values[i] == temp || recvValues[i] == -1);
    }
    return
}
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 applicable to the context of the question

    }
```