MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

VI Semester MIDTERM TEST
PARALLEL COMPUTER ARCHITECTURE AND PROGRAMMING (CSE_3252)

Time Duration: 2 Hours **Date: 19.03.2024** **Max marks: 30 MARKS**

Scheme and Solution

Question No	Topic	Marks	BL	CO
1	Which type of memory in OpenCL is shared among all work-items? A) Local memory B) **Global memory** C) Constant memory D) Private memory	1	B1	CO3
2	Which component of OpenCL is responsible for managing device resources? A) Compiler **B) Runtime** C) Scheduler D) Kernel	1	B1	CO3
3	What does the term "host" refer to in the context of OpenCL? A) The main processing unit of a GPU **B) The central processing unit (CPU) of a computer** C) A specialized type of kernel D) A cloud computing service	1	B1	CO3
4	In OpenCL, what is a kernel? A) A type of memory storage **B) An execution unit for parallel computing tasks** C) A graphical user interface element D) A data structure for file management	1	B1	CO3
5	------------model defines how the concurrency model is mapped to physical hardware A) Platform model B) Execution model C) Memory model D) **Programming model**	1	B1	CO3
6	Which generation of computer systems used Small-scale integrated (SSI) and Medium-scale integrated (MSI) circuits as the basic building blocks A) Second generation **B) Third generation** C) First generation D) Fourth Generation	1	B1	CO1
7	#include <mpi.h>	1	B3	CO2

	#include <stdio.h>
	int main(int argc, char *argv[]) {
	    int rank, size, value, sum;
	    MPI_Init(&argc, &argv);
	    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	    MPI_Comm_size(MPI_COMM_WORLD, &size);
	    value = rank;
	    MPI_Reduce(&value, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
	    if (rank == 0) {
	        printf("Sum of ranks: %d\n", sum);
	    }
	    MPI_Finalize();
	    return 0;
	}
	Assuming we run this MPI program with 4 processes, what will be the output?
	A) Sum of ranks: 6
	B) Sum of ranks: 10
	C) Sum of ranks: 4
	D) Sum of ranks: 0			

8	#include <mpi.h>
	#include <stdio.h>
	
	int main(int argc, char *argv[]) {
	    int rank, size, data;
	    MPI_Init(&argc, &argv);
	    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	    MPI_Comm_size(MPI_COMM_WORLD, &size);
	    if (rank == 0) {
	        data = 100;
	        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
	    } else if (rank == 1) {
	        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
	        printf("Process 1 received data: %d\n", data);
	    }
	    MPI_Finalize();
	    return 0;
	}
	What will be the output of the program when executed with 2 MPI processes?
	A) Process 1 received data: 0
	B) Process 1 received data: 100
	C) Process 1 received data: 1
	D) Compilation Error
		1	**B3**	**CO2**

9	Which of the following is a characteristic of GPU architecture? A) High clock speeds and large cache sizes B) **Large numbers of cores optimized for parallel processing** C) Low power consumption and compact design D) Support for complex branch prediction and out-of-order execution	**1**	**B1**	**CO1**
10	In which architecture all PEs receive the same instruction broadcast from the control unit but operate on different data sets from distinct data streams A) Single instruction stream-single data stream (SISD) B) **Single instruction stream-multiple data stream (SIMD)** C) Multiple instruction stream-single date stream (MISD) D) Multiple instruction stream-multiple data stream (MIMD)	**1**	**B1**	**CO1**
11 a.	Using collective communication routines, implement an efficient MPI program which reads matrix *A* of size 4*x*4 and it produces a resultant matrix *RES* of size 4*x*4 such that every row elements of *A* are added with a key value. The key value for the first row is the minimum element of the last row and the key value for remaining row is the minimum element from the previous row of matrix *A*. Make use of 4 processes (including root) to perform this task. Example: A RES 1 2 3 4 **3 4 5 6** 5 6 7 8 6 7 8 9 2 4 3 5 7 9 8 10 **2** 3 4 6 4 5 6 8 ***use of collective communication-MPI_Bcast() and MPI_Gather()-2 marks** ***correct logic in code – 2 marks** **Solution:**	**4**	**B3**	**CO2**

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ROWS 4
#define COLS 4

// Function to find the minimum element in a row
int min_row(int row[]) {
   int min = row[0];
   for (int i = 1; i < COLS; i++) {
      if (row[i] < min) {
         min = row[i];
      }
   }
   return min;
```

```
}

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

    // Check if the number of processes is not equal to 4
    if (size != 4) {
        if (rank == 0) {
            printf("Error: Number of processes must be 4\n");
        }
        MPI_Finalize();
        return 1;
    }

    int A[ROWS][COLS];
    int key;

    // Root process reads the matrix A
    if (rank == 0) {
        printf("Enter elements of matrix A (4x4):\n");
        for (int i = 0; i < ROWS; i++) {
            for (int j = 0; j < COLS; j++) {
                scanf("%d", &A[i][j]);
            }
        }
    }

    // Broadcast matrix A from root process to all other processes
    MPI_Bcast(A, ROWS * COLS, MPI_INT, 0, MPI_COMM_WORLD);

    // Calculate key value based on rank
    if (rank == 0) {
        key = min_row(A[ROWS - 1]);  // Key for the first row
    } else {
        key = min_row(A[rank - 1]);  // Key for the remaining rows
    }

    // Broadcast key value from root process to all other processes
    MPI_Bcast(&key, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Add key value to each element of the row
    for (int i = 0; i < COLS; i++) {
        A[rank][i] += key;
    }

    // Gather the resultant matrix RES to root process
    int RES[ROWS][COLS];
    MPI_Gather(A[rank], COLS, MPI_INT, RES, COLS, MPI_INT, 0,
MPI_COMM_WORLD);

    // Print resultant matrix RES in root process
    if (rank == 0) {
        printf("Resultant Matrix (RES):\n");
        for (int i = 0; i < ROWS; i++) {
            for (int j = 0; j < COLS; j++) {
                printf("%d ", RES[i][j]);
            }
            printf("\n");
        }
    }

    MPI_Finalize();
    return 0;
}
```

11 b.	Differentiate between following collective communication routines with the help of example code i) MPI_Allgather( ) and MPI_Alltoall( ) -2 marks ii) MPI_Reduce( ) and MPI_Scan( ) -2 marks *Function prototype and sample code	**4**	**B4**	**CO2**

MPI_Allgather() gathers data from all processes and scatters it to all processes, while MPI_Alltoall() sends distinct data from each process to all other processes.
MPI_Reduce() performs a reduction operation on data from all processes and sends the result to one process,
while MPI_Scan() performs a parallel prefix operation and sends the partial result to each process.

```
MPI_Allgather(&send_data, 1, MPI_INT, recv_data, 1, MPI_INT,
MPI_COMM_WORLD);

MPI_Alltoall(send_data, 1, MPI_INT, recv_data, 1, MPI_INT,
MPI_COMM_WORLD);

MPI_Reduce(&send_data, &recv_data, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);
```

	MPI_Scan(&send_data, &recv_data, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);			

11 c.	Demonstrate the process of creating kernel and setting the kernel arguments.
	
	ClCreatekernel and clSetkernelArg function protype with working -2 marks
	
	To create a kernel in OpenCL, you typically define a kernel function in a separate .cl file. Here's an example kernel function that performs element-wise addition of two arrays:
	
	`__kernel void add_arrays(__global const float* a, __global const float* b, __global float* result, const int size) {`
	`    int i = get_global_id(0);`
	`    if (i < size) {`
	`        result[i] = a[i] + b[i];`
	`    }`
	`}`
	
	In this kernel, __kernel is the kernel specifier, add_arrays is the name of the kernel function, and it takes input arrays a and b, as well as an output array result, along with the size of the arrays.
	Setting Kernel Arguments:
	After creating the kernel, you need to set kernel arguments before enqueuing the kernel for execution. Here's how you set kernel arguments using the OpenCL API in C:
	// Assume you have already created an OpenCL context, command queue, and program object
	
	// Create kernel object
	cl_kernel clCreateKernel(
	cl_program program,
	const char *kernel_name ,
	cl_int *errcode_ret);
	
	cl_kernel kernel = clCreateKernel(program, "add_arrays", &err);
	
	// Set kernel arguments
	cl_int clSetKernelArg(
	cl_kernel kernel,
	cl_uint arg_index ,
	size_t arg_size,
	const void *arg_value);
	
	clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&buffer_a);
	clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&buffer_b);
	clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*)&buffer_result);
	clSetKernelArg(kernel, 3, sizeof(int), (void*)&size);
		2	**B3**	**CO3**

12 a.	Why do we need Parallelism? Demonstrate the following parallel computer structures with diagram. i. Pipeline computer ii. Array computer iii. Multiprocessor systems ***Need for parallelism – 1 marks for each** ***3 parallel computer structures with diagram – 3*1 marks**	**4**	**B3**	**CO1**

12 b.	What are two types of barrier operations for a command queue in OpenCL. Write an OpenCL kernel code that takes a string S as input and one integer value N. The program produces the output string N times as shown below in parallel:
	I/p: S = Hello N = 3
	O/p String: HelloHelloHello
	Note: Each work item copies entire S
	
	Barrier functions – 2marks
	Openecl kernel code -2 marks
	
	• Two types of barrier operations for a command queue:
	✓ cl_int clFinish (cl_command_queue cmdQueue);
	This function blocks until all of the commands in a command queue have completely executed.
	✓ cl_int clFlush (cl_command_queue cmdQueue);
	
	This function blocks until all of the commands in a command queue have been removed from the command queue.
	This means that the commands will defienetly be in-flight but will not necessarily have completed.
	
	`__kernel void repeat_string(__global const char* input_string, __global char* output_string, const int n) {`
	`  int global_id = get_global_id(0);`
	`  int input_length = strlen(input_string);`
	`  int output_index = global_id * input_length;`
	`  // Repeat the input string N times`
	`  for (int i = 0; i < n; i++) {`
	`    for (int j = 0; j < input_length; j++) {`
	`      output_string[output_index++] = input_string[j];`
	`    }`
	`  }`
	`}`
		4	**B3**	**CO3**

12 c.	Compare multi-core trajectory and many-core trajectory. • **Any 4 difference – 2 marks**	**2**	**B4**	**CO1**

	The multi-core trajectory: • It seeks to maintain the execution speed of sequential programs while moving into multiple cores. • *Multicore trajectory* began as two-core processors, with the number of cores doubling every generation of MP. ➢ Eg: Intel Corei7 MP which has 4 processor cores, each of which is an out-of-order, multiple instruction issue processor implementing the full x86 instruction set. ➢ This MP supports *hyperthreading* with two *hardware threads* per core and is designed to maximize the exec speed of sequential programs. ***The many-core (many-thread) trajectory:*** • It focuses more on the *execution throughput* of parallel applications • The many-cores began as a *large number of much smaller cores*, and, once again, the number of cores doubles with each generation			